Patrick R. Schaumont

A Practical Introduction to Hardware/Software Codesign

A Practical Introduction
to Hardware/Software Codesign

Patrick R. Schaumont

A Practical Introduction to Hardware/Software Codesign

Springer

Dr. Patrick R. Schaumont
Virginia Tech
Bradley Dept. Electrical & Computer Engineering
Whittemore Hall 302
24061 Blacksburg VA
USA
schaum@vt.edu

The most important day is today.

Preface

This is a practical book for computer engineers who want to understand or implement hardware/software systems. It focuses on problems that require one to combine hardware design with software design – such problems can be solved with hardware/software codesign. When used properly, hardware/software codesign works better than hardware design or software design alone: it can improve the overall performance of digital systems, and it can shorten their design time.

Hardware/software codesign can help a designer to make trade-offs between the flexibility and the performance of a digital system. To achieve this, a designer needs to combine two radically different ways of design: the sequential way of decomposition in time, using software, with the parallel way of decomposition in space, using hardware.

Intended Audience

This book assumes that you have a basic understanding of hardware that you are familiar with standard digital hardware components such as registers, logic gates, and components such as multiplexers and arithmetic operators. The book also assumes that you know how to write a program in C. These topics are usually covered in an introductory course on computer engineering or in a combination of courses on digital design and software engineering.

The book is suited for advanced undergraduate students and beginning graduate students. I believe the book will also be useful for researchers from other (non-computer engineering) fields, who have a need for hardware/software codesign. For example, I often work with cryptographers who have no formal training in hardware design but still want to obtain dedicated architectures for highly specialized algorithms. In my experience, it is very rewarding to explain codesign ideas to a cryptographer (or any other domain expert, for that matter).

A key learning objective of the book is to provide a hands-on, practical introduction to design that combines sequential decomposition (software) and spatial decomposition (hardware). There is a surprisingly large mental gap between these two approaches to design. In fact, undergraduate courses tend to emphasize only one way of thinking at a time. Undergraduates will take a course on programming (for

a stored-program computer) or a digital hardware design course. Considering how dramatic the changes to the computer architecture landscape have been in recent years (multicore, system-on-chip, ultra-low power computing, ...), it's becoming essential to look beyond these boundaries.

Organization

The book puts equal emphasis on design methods and modeling (design languages). Design modeling helps a designer to think about a design problem and to capture a solution for the problem. Design methods are systematic transformations that convert design models into implementations. There are four parts in this book: Basic Concepts, the Design Space of Custom Architectures, Hardware/Software Interfaces, and Applications.

Figure 1 illustrates how everything fits together. The first part of the book introduces modeling techniques concepts, required to understand how software can be converted into hardware and vice versa. The second part of the book describes the design space of custom architectures, starting at fully dedicated, high-performance
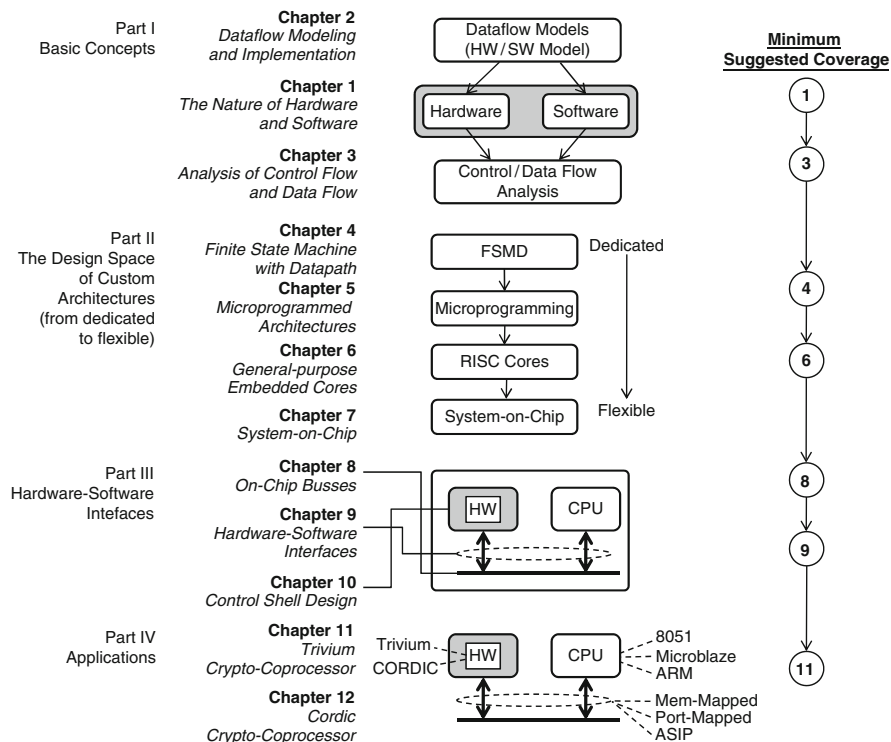

Fig. 1 Outline of the book

hardware implementations, and gradually evolving into flexible system-on-chip (SoC) architectures. The third part of the book goes into the details of hardware/software interface design. The final part of the book shows two application case studies.

Basic Concepts: The first chapter covers the fundamental properties of hardware and software and discusses the motivation for hardware/software codesign (*Chap. 1*). *Chapter 2* describes Dataflow models, a modeling technique that can target hardware as well as software. *Chapter 3* introduces control-flow and data-flow as the common underlying properties of hardware and software. By analyzing the control dependencies and the data dependencies of a C program, a designer obtains insight into possible hardware implementations of that C program.

The Design Space of Custom Architectures: The second part is a tour along the vast design space of flexible, customized architectures. A review of four architectures shows how hardware gradually evolves into software. The Finite State Machine with Datapath (FSMD) of *Chap. 4* is the starting point. FSMD models are the equivalent of hardware modeling at the register-transfer level (RTL). *Chapter 5* introduces micro-programmed architectures. These are still very much like RTL machines, but they have a flexible controller, which allows them to be reprogrammed with software. *Chapter 6* reviews general-purpose embedded RISC cores. These processors are the heart of typical contemporary hardware/software systems. Finally, *Chap. 7* ties the general-purpose embedded core back to the FSMD in the context of a System-on-Chip architecture (SoC). The SoC sets the stage for the hardware/software codesign problems that are addressed in the third part.

Hardware/Software Interfaces: The third part describes the link between hardware and software in the SoC architecture, in three chapters. *Chapter 8* discusses a typical on-chip bus structure and explains how it can efficiently move information between hardware and software. *Chapter 9* presents three different locations in the SoC architecture where a designer could attach custom hardware. This includes the memory-mapped interface, the coprocessor interface, and the custom processor datapath. Finally, *Chap. 10* shows how a designer can take an arbitrary hardware module and attach it to one of the three hardware/software interfaces described in *Chap. 9*.

Applications: The final part gives two sample applications of hardware–software codesign and shows how the different techniques and hardware–software interfaces can be used in actual design. *Chapter 11* presents a crypto-coprocessor, while *Chap. 12* describes a CORDIC coprocessor, including a prototype and an FPGA board.

There are several subjects which are *not* mentioned or discussed in the book. As an introductory discussion on a complex subject, I tried to find a balance between detail and complexity. For example, I did not include a discussion of advanced concepts in software concurrency, such as threads, and software architectures, such as operating systems and drivers. I also did not discuss software interrupts, or advanced system operation concepts such as Direct Memory Access.

I assume that the reader will go through all the chapters in sequence. The minimum suggested coverage which will still provide the reader an understandable introduction in hardware–software codesign includes Chaps. 1, 3, 4, 6, 8, 9, and 11.

Making it practical

The book emphasizes ideas and design methods, in combination with hands-on, practical experiments. The book therefore discusses detailed examples throughout the chapters, and it includes an entire section (*Applications*) to discuss the overall design process. The hardware descriptions are made in GEZEL, an open-source cycle-accurate hardware modeling language. The GEZEL website, which distributes the tools, examples, and other documentation, is at

```
http://rijndael.ece.vt.edu/gezel2
```

There are several reasons why I choose not to use a mainstream HDL such as VHDL, Verilog, or SystemC.

- A first reason is *reduced modeling overhead*. Although models are crucial for embedded system construction, detailed modeling issues often distract the readers' attention from the key issues. For example, modeling the clock signal in hardware requires a lot of additional effort, and it is not essential when doing single-clock synchronous design (which covers the majority of digital hardware design today).
- A second reason is that GEZEL comes with *support for cosimulation* built-in. GEZEL models can be cosimulated with different processor simulation models, including ARM, 8051, and others. GEZEL includes a library-block modeling mechanism that enables one to define new cosimulation interfaces with other simulation engines.
- A third reason is *conciseness*. This is a practical book with many design examples. Listings are unavoidable, but they need to be short. Of the 78 captioned listings in this book, 64 of them fit on a single page. Chapter 4 further illustrates the point of conciseness with a single design example in GEZEL, VHDL, Verilog, and SystemC side-by-side.
- A fourth reason is the path to implementation. GEZEL models can be translated (automatically) to VHDL. These models can be synthesized using standard HDL logic synthesis tools.

I use the material in this book in a class on hardware/software codesign. The class hosts senior-level undergraduate students, as well as first-year graduate-level students. For the seniors, this class ties many different elements of computer engineering together: computer architectures, software engineering, hardware design, debugging, and testing. For the graduate students, it is a refresher and a starting point of their graduate researcher careers in computer engineering.

In the class on codesign, the GEZEL experiments connect to an FPGA back-end and an FPGA prototyping kit. These experiments are implemented as homework.

Modeling assignments in GEZEL alternate with integration assignments on FPGA. Through the use of the GEZEL backend support, students even avoid writing of VHDL code, but directly switch from cosimulation to FPGA implementation. At the end of the course, there is a 'contest'. The students receive a reference implementation in C that runs on their FPGA prototyping kit. They need to accelerate this reference as much as possible using codesign techniques.

Blacksburg   Patrick Schaumont
July 2010   schaum@vt.edu

Contents

Part IV   Applications

Part I
Basic Concepts

This part introduces important concepts in hardware–software codesign. We compare and contrast the two classic 'mindsets of design': the hardware mindset and the software mindset, and we point out that hardware/software codesign is more than just glueing hardware and software components together; instead, it's about finding the correct balance between flexibility and performance in a design. The trade-off between parallel and sequential implementations is another fundamental idea for a codesigner; we will discuss a concurrent system model (data-flow) that can be converted into hardware (parallel) as well as into software (sequential). We will also discuss the analysis of C programs in terms of control-flow and data-flow.

Chapter 1
The Nature of Hardware and Software

Abstract Hardware/software codesign is the activity of partitioning an application into a flexible part (software) and a fixed part (hardware). The flexible part includes C programs, configuration data, parameter settings, bitstreams, and so forth. The fixed part consists of programmable components such as microprocessors and coprocessors. There are several technological and economical reasons for implementing electronic systems in this fashion, and we will discuss them upfront in this chapter. Next, we consider the design space of programmable components: what distinguishes one component from the other, and how do we select one in a given hardware–software codesign. A key observation is that there is a trade-off between flexibility and efficiency. A third part in this chapter will define the abstraction levels for hardware and software for the purpose of this book. And finally, we will also define three terms that play a vital role in this book, namely the terms concurrent, parallel, and sequential.

1.1 Introducing Hardware/Software Codesign

This chapter describes the main motivators for hardware/software codesign. However, *hardware* and *software* may mean very different things to different people. Therefore, we start by providing a simple definition of each. This will help to put readers from different backgrounds on the same line. This section will also provide a small example of hardware/software codesign and define several key terms, which are used later.

1.1.1 Hardware

In this book, we will model hardware by means of single-clock synchronous digital circuits created using word-level combinational logic and flip-flops.

These circuits can be modeled with building blocks such as registers, adders, and multiplexers. Cycle-based hardware modeling is often called register-transfer-level

(RTL) modeling because the behavior of a circuit can be thought of as a sequence of transfers between registers, with logical and arithmetic operations performed on the signals during the transfers.

Figure 1.1a gives an example of a hardware module captured in RTL. A register can be incremented or cleared depending on the value of the control signal rst. The register is updated on the up-going edges of a clock signal clk. The wordlength of the register is 8 bit. Even though the connections in this figure are drawn as single lines, each line represents a bundle of 8 wires. Figure 1.1a uses graphics to capture the circuit; in this book, we will be using a hardware description language (HDL) called GEZEL. Figure 1.1b shows the equivalent description of this circuit in GEZEL language. Chapter 4 will describe GEZEL modeling in detail.

Figure 1.1c illustrates the behavior of the circuit using a timing diagram. In such a diagram, time runs from left to right, and the relationships between signals of the circuit are shown vertically. In this case, the register is cleared on clock edge 2, and it is incremented on clock edge 3, 4, and 5. Before clock edge 2, the value of the


```
dp count( in rst: ns(1)) {
  reg c : ns(8);
  always  {
    q = c;
    c = rst ? 0 : c + 1;
  }
}
```
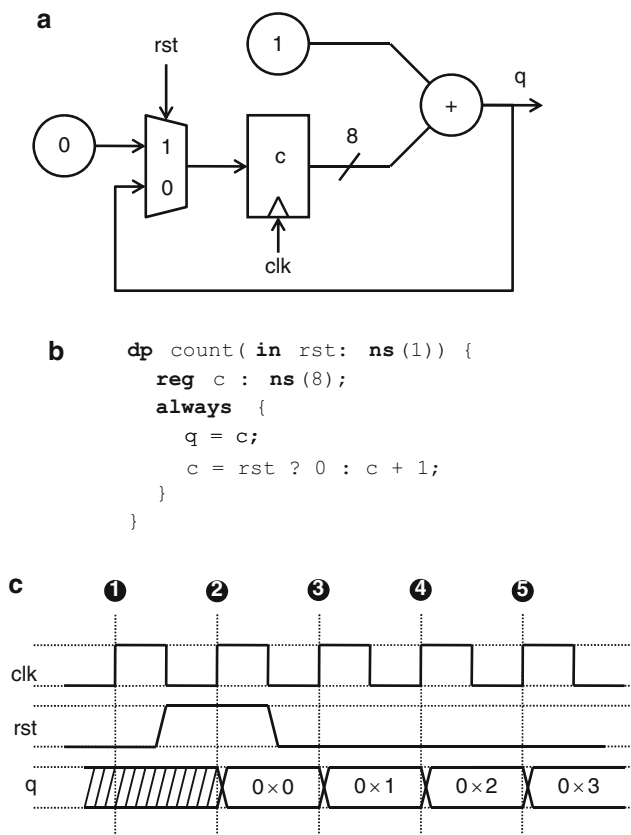
Fig. 1.1 (**a**) Hardware schematic for a counter, (**b**) GEZEL model of this counter, (**c**) Timing Diagram

register is assumed to be unknown, and the timing diagram indicates q's value as a shaded area. We will be using timing diagrams to describe the low-level behavior of hardware–software interfaces, as well as on-chip buses.

The single-clock synchronous model cannot express every possible hardware circuit. For example, it cannot model events at a time resolution smaller than a clock cycle. As a result, there are many forms of hardware that cannot be captured with it, including asynchronous hardware, dynamic logic, multiphase clocked hardware, and hardware with latches. However, single-clock synchronous hardware will be sufficient to explain the key concepts of hardware–software codesign for this book. The single-clock model is a very convenient abstraction for a designer who maps behavior (e.g., an algorithm) into discrete steps of one clock cycle. It enables this designer to envision how the hardware implementation of a particular algorithm should look like.

1.1.2 Software

Hardware/software codesign deals with hardware/software interfaces. The low-level construction details of software are important, because they can directly affect the performance and implementation cost of the interface. Hence, the book will consider important software implementation aspects such as the various sections of memory (global, stack, heap), the different kinds of memory (registers, caches, RAM, and ROM), and the techniques to control their use from within a high-level programming language such as C.

We will model software as single-thread sequential programs, written in C or assembly. Programs will be shown as listings, for example, Listing 1.1 and Listing 1.2. Most of the discussions in this book will be processor-independent, and they will assume 32-bit architectures (ARM, Microblaze) as well as 8-bit architectures (8051, Picoblaze).

The choice for single-thread sequential C is simply because it matches so well to the actual execution model of a typical microprocessor. For example, the sequential execution of C programs corresponds to the sequential instruction fetch-and-execute cycle of microprocessors. The variables of C are stored in a single, shared-memory

Listing 1.1 C example

```
1
2  int max;
3
4  int findmax(int a[10]) {
5    unsigned i;
6    max = a[0];
7    for (i=1; i<10; i++)
8      if (a[i] > max) max = a[i];
9  }
```

Listing 1.2 ARM assembly example

```
    .text
findmax:
    ldr r2, .L10
    ldr r3, [r0, #0]
    str r3, [r2, #0]
    mov ip, #1
.L7:
    ldr r1, [r0, ip, asl #2]
    ldr r3, [r2, #0]
    add ip, ip, #1
    cmp r1, r3
    strgt   r1, [r2, #0]
    cmp ip, #9
    movhi   pc, lr
    b    .L7
.L11:
    .align  2
.L10:
    .word   max
```

space, corresponding to the memory attached to the microprocessor. There is a close
connection between the storage concepts of a microprocessor (registers, stack) and
the storage types supported in C (`register int`, local variables). Furthermore,
common datatypes in C (`char`, `int`) directly map onto microprocessor variables
(byte, word). A detailed understanding of C execution is closely related to a detailed
understanding of the microprocessor activity at a lower abstraction level.

Of course, there are many forms of software that do not fit the model of a
single-thread sequential C program. Multi-threaded software, for example, creates
the illusion of *concurrency* and lets users execute multiple programs at once. Other
forms of software, such as object-oriented software and functional programming,
substitute the simple machine model of the microprocessor with a more sophisti-
cated one. Such more advanced forms of software are crucial to master complex
software applications. However, the link between execution of these more sophisti-
cated forms of software and microprocessor activity is less obvious. Therefore, we
will focus on the most simple form of C.

The material in this book does not follow any particular assembly language.
It assumes that the reader is familiar with the concepts of assembly. The book
emphasizes the relationship between C and assembly code. A hardware/software
codesigner often needs to handle optimization problems that can only be solved at
the level of assembly coding. In that case, the designer needs to be able to link the
software, as captured in C, with the program executing on the processor, as rep-
resented by assembly code. Most C compilers offer the possibility to generate an
assembly listing of the generated code, and we will make use of that feature. Listing
1.2 for example, was generated out of 1.1.

We will put some emphasis on being able to link the statements of a C program
to the assembly instructions. This is easier than you would think. As an example,

```
int max;                                      .text
                                 findmax:     ldr       r2, .L10
int findmax(int a[10]) {         .L10         ldr       r3, [r0, #0]
  unsigned i;                                 str       r3, [r2, #0]
  max = a[0];                                 mov       ip, #1
  for (i=1; i<10; i++)           .L7:         ldr       r1, [r0, ip, asl #2]
    if (a[i] > max)  max = a[i];              ldr       r3, [r2, #0]
}                                             add       ip, ip, #1
                                              cmp       r1, r3
                                              strgt     r1, [r2, #0]
                                              cmp       ip, #9
                                              movhi     pc, lr
                                              b.        .L7
                                 .L11:        .align    2.
                                 L10:         .word     max
```

Fig. 1.2 Mapping C to Assembly

let's compare Listing 1.1 and Listing 1.2. An ideal starting point when matching a C program to an assembly program is to look for similar structures: loops in C will be reflected as branches in assembly; if-then-else statements in C will be reflected as conditional branches in assembly. Even if you're unfamiliar with the assembly from a microprocessor, you can often derive such structures easily.

Figure 1.2 gives an example for the programs in Listing 1.1 and Listing 1.2. The for-loop in C is marked with a label and a branch instruction. All the assembly instructions in between the branch and the label are part of the body of the loop. Once the loop structure is identified, it is easy to derive the rest of the code, as the following examples show.

- The if-statement in C requires the evaluation of a greater-than condition. In assembly, an equivalent cmp (compare) instruction can be found. This allows you to conclude that the operands r1 and r3 of the compare instruction must contain a[i] and max of the C program. Both of these variables are stored in memory; a[i] because it's an indexed variable, and max because it's a global variable. Indeed, looking at the preceding instruction in the C program, you can see that both r1 and r3 are defined with ldr (load-register) instructions, which require an address.
- The address for r1 equals [r0, ip, asl #2], which stands for the expression (r0 + ip) << 2. This may not be obvious if this is the first time you are looking at ARM assembly; but it's something you will remember quickly. In fact, the format of this expression is easy to explain. The register ip contains the loop counter, since ip is incremented once within the loop body, and the value of ip is compared with the loop boundary value of 9. The register r0 is the *base address* of a[], the location in memory where a[0] is stored. The shift-over-2 is needed because a[] is an array of integers. Microprocessors use byte-addressable memory, which means that each integer requires 4 byte address locations.
- Finally, the conditional assignment of the max variable in C is not implemented using conditional branch instructions in assembly. Instead, a strgt

(store-if-greater) instruction is used. This is a *predicated* instruction, an instruction that executes only when a given conditional flag is true.

The bottom line of this analysis is that, with a minimal amount of effort, you are able to understand a great deal on the behavior of a microprocessor simply by comparing C programs with equivalent assembly programs. In Chap. 6, you will see how you can use the same approach to evaluate the quality of the assembly code generated by a compiler out of C code.

1.1.3 Hardware and Software

The objective of this book is to discuss the combination of hardware design and software design in all its forms. Hardware as well as software can be modeled using RTL programs and C programs, respectively. A term *model* merely indicates they are not the actual implementation, but only a representation of it. An RTL program is a *model* of a network of logic gates; a C program is a *model* of a binary image of microprocessor instructions. It is not common to talk about C programs as *models*; in fact, software designers think of C programs as actual implementations. In this book, we will therefore refer to hardware *models* and C or assembly *programs*.

Models are an essential part of the design process. They are a formal representation of a designers' intent, and they are used as input for simulation tools and implementation tools. In hardware/software codesign, we are working with models that are partly written as C programs, and partly as RTL programs. We will discuss this idea by means of a simple example.

Figure 1.3 shows an 8051 microcontroller and an attached coprocessor. The coprocessor is attached to the 8051 microcontroller through two 8-bit ports P0 and P1. A C program executes on the 8051 microcontroller, and this program contains instructions to write data to these two ports. When a given, predefined value appears on port P0, the coprocessor will make a copy of the value present on port P1 into an internal register.
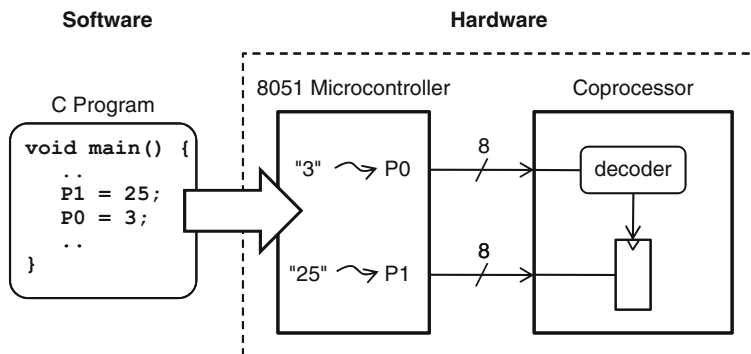
Fig. 1.3 A codesign model

Listing 1.3 8051 driver program

```
1  #include <8051.h>
2
3  enum {ins_idle, ins_hello};
4
5  void sayhello(char d) {
6    P1 = d;
7    P0 = ins_hello;
8    P0 = ins_idle;
9  }
10
11 void terminate() {
12   // special command to stop simulator
13   P3 = 0x55;
14 }
15
16 void main() {
17   sayhello(3);
18   sayhello(2);
19   sayhello(1);
20   terminate();
21 }
```

This very simple design can be addressed using hardware/software codesign; it includes the design of a hardware model and the design of a C program. The hardware model contains the 8051 processor, the coprocessor, and the connections between them. During execution, the 8051 processor will execute a software program written in C. Listing 1.3 shows that C program. Listing 1.4 shows an RTL hardware model for this design, written in the GEZEL language.

The C driver sends three values to port P1. Each time, it also cycles the value on port P0 between ins_hello and ins_idle, which are encoded as value 1 and 0, respectively.

The hardware model in Listing 1.4 is slightly longer. It includes both the microcontroller and the coprocessor. The coprocessor is on lines 1–18. This particular hardware model is a combination of a finite state machine (lines 10–18) and a datapath (lines 1–8). This modeling method is called FSMD (for finite-state-machine with datapath), and we will discuss FSMD in detail in Chap. 4. This FSMD is quite easy to understand. The datapath contains several instructions: decode and hello. The FSM controller selects, each clock cycle, which of those instructions to execute. For example, lines 14–15 shows the following control statement.

```
@s1 if (insreg == 1) then (hello, decode) -> s2;
                     else (decode)         -> s1;
```

This means: when the value of insreg is 1 and the FSM controller current state is s1, the datapath will execute instructions hello and decode, and the FSM controller next-state is s2. When the value of insreg would be 0, the datapath will execute only instruction decode and the FSM controller next-state is s1. The

Listing 1.4 GEZEL model for 8051 platform

```
1  dp hello_decoder(in   ins : ns(8);
2                   in   din : ns(8)) {
3    reg insreg : ns(8);
4    reg dinreg : ns(8);
5    sfg decode   { insreg = ins;
6                   dinreg = din; }
7    sfg hello    { $display($cycle, " Hello! You gave me ", dinreg); }
8  }
9
10 fsm fhello_decoder(hello_decoder) {
11   initial s0;
12   state s1, s2;
13   @s0 (decode) -> s1;
14   @s1 if (insreg == 1) then (hello, decode) -> s2;
15                        else (decode)         -> s1;
16   @s2 if (insreg == 0) then (decode)         -> s1;
17                        else (decode)         -> s2;
18 }
19
20 ipblock my8051 {
21   iptype "i8051system";
22   ipparm "exec=driver.ihx";
23   ipparm "verbose=1";
24   ipparm "period=1";
25 }
26
27 ipblock my8051_ins(out data : ns(8)) {
28   iptype "i8051systemsource";
29   ipparm "core=my8051";
30   ipparm "port=P0";
31 }
32
33 ipblock my8051_datain(out data : ns(8)) {
34   iptype "i8051systemsource";
35   ipparm "core=my8051";
36   ipparm "port=P1";
37 }
38
39 dp sys {
40   sig ins, din : ns(8);
41   use my8051;
42   use my8051_ins(ins);
43   use my8051_datain(din);
44   use hello_decoder(ins, din);
45 }
46
47 system S {
48   sys;
49 }
```

overall coprocessor behavior is like this: when the `ins` input changes from 0 to 1, then the `din` input will be printed in the next clock cycle.

The 8051 microcontroller is captured in Listing 1.4 as well. However, the internals of the microcontroller are not shown; only the hardware interfaces relevant to the coprocessor are included. The 8051 microcontroller is captured with three `ipblock` (GEZEL library modules), on lines 20–37. The first `ipblock` is an `i8051system`. It represents the 8051 microcontroller core, and it indicates the name of the compiled C program that will execute on this core (`driver.ihx` on line 22). The other two `ipblock` are two 8051 output ports (`i8051systemsource`), one to model port `P0`, and the other to model port `P1`.

Finally, the coprocessor and the 8051 ports are wired together in a top-level module, shown in lines 39–49. We can now simulate the entire model, including hardware and software, as follows. First, the 8051 C program is compiled to a binary executable. Next, the GEZEL simulator will combine the hardware model and the 8051 binary executable in a *cosimulation*. The output of the simulation model is shown below.

```
> sdcc driver.c
> /opt/gezel/bin/gplatform hello.fdl
i8051system: loading executable [driver.ihx]
9662 Hello! You gave me 3/3
9806 Hello! You gave me 2/2
9950 Hello! You gave me 1/1
Total Cycles: 10044
```

You can notice that the model produces output on cycles 9662, 9806, and 9950, while the complete C program executes in 10044 cycles. The evaluation and analysis of cycle-accurate behavior is a very important aspect of codesign, and we will address it throughout the book.

1.1.4 Defining Hardware/Software Codesign

The previous example motivates the following traditional definition of hardware/-software codesign.

> Hardware/Software codesign is the design of cooperating hardware components and software components in a single design effort.

For example, if you would design the architecture of a processor and at the same time develop a program that could run on that processor, then you would be using hardware/software codesign. However, this definition does not tell precisely what *software* and *hardware* mean. In the previous example, the software was a C program, and the hardware was an 8051 microcontroller with a coprocessor. In reality, there are *many forms of hardware and software*, and the distinction between them can easily become blurred. Consider the following examples.

- A Field Programmable gate Array (FPGA) is a hardware circuit that can be reconfigured to a user-specified netlist of digital gates. The program for an FPGA is a 'bitstream', and it is used to configure the netlist topology. Writing 'software' for an FPGA really looks like hardware development – even though it *is* software.
- A soft-core is a processor implemented in the bitstream of an FPGA. However, the soft-core itself can execute a C program as well. Thus, software can execute on top of other 'software'.
- A Digital-Signal Processor (DSP) is a processor with a specialized instruction-set, optimized for signal-processing applications. Writing efficient programs for a DSP requires detailed knowledge of these specialized instructions. Very often, this means writing assembly code, or making use of a specialized software library. Hence, there is a strong connection between the efficiency of the software and the capabilities of the hardware.
- An Application-Specific Instruction-set Processor (ASIP) is a processor with a customizable instruction set. The hardware of such a processor can be extended, and these hardware extensions can be encapsulated as new instructions for the processor. Thus, an ASIP designer will develop a hardware implementation for these custom instructions and subsequently write software that uses those instructions.
- The CELL processor, used in the Playstation-3, contains one control processor and 8 slave-processors, interconnected through a high-speed on-chip network. The software for a CELL is a set of 9 concurrent communicating programs, along with configuration instructions for the on-chip network. To maximize performance, programmers have to develop CELL software by describing simultaneously the computations and the communication activities in each processor.

These examples illustrate a few of the many forms of hardware and software that designers use today. A common characteristic of all these examples is that creating the 'software' requires intimate familiarity the 'hardware'. In addition, hardware includes much more than RTL models: it also includes specialized processor instructions, the FPGA fabric, multicore architectures, and more. Let us define the *application* as the overall function of a design, including hardware as well as software. This allows to define hardware/software codesign as follows:

> Hardware/Software codesign is the partitioning and design of an application in terms of fixed and flexible components.

Notice that we use the term 'fixed' instead of hardware, and 'flexible' instead of software. In the remainder of this chapter, we will discuss the big picture of hardware/software codesign. This will clarify the choice of these terms.

1.2 The Quest for Energy Efficiency

Choosing between implementing a design in hardware or implementing it in software may seem like a no-brainer. Indeed, from a designers' point-of-view, the easiest approach is to write software, for example in C. Software is easy and flexible, software compilers are fast, there are large amounts of source code available, and all you need to start development is a nimble personal computer. Furthermore, why go through the effort of designing a hardware architecture when there is already one available (namely, the RISC processor)?

1.2.1 Relative Performance

Proponents of hardware implementation will argue that *performance* is a big plus of hardware over software. Specialized hardware architectures have a larger amount of parallelism than software architectures. We can measure this as follows: *Relative performance* means the amount of useful work done per clock cycle. Under this metric, highly parallel implementations are at an advantage because they do many things at the same time.

Figure 1.4 illustrates various cryptographic implementations in software and hardware that have been proposed over the past few years (2003–2008). These are all designs proposed for embedded applications, where the trade-off between hardware and software is crucial. As demonstrated by the graph, hardware crypto-architectures have, on the average, a higher relative performance compared to embedded processors.
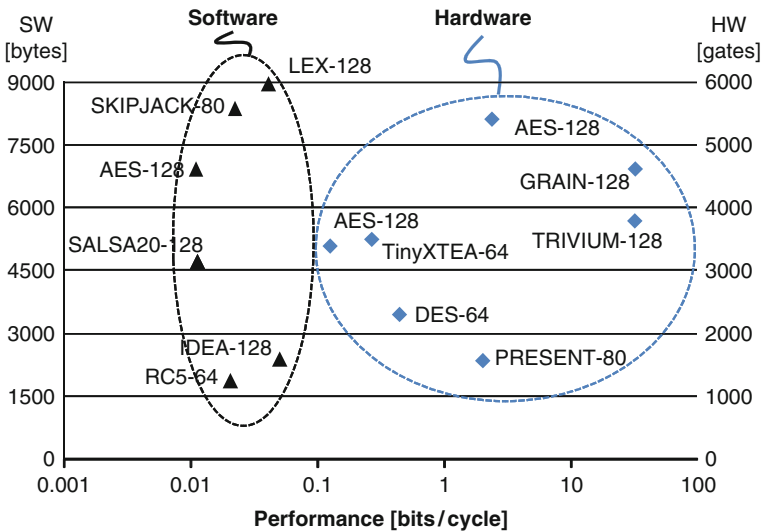
Fig. 1.4 Cryptography on small embedded platforms

However, relative performance may not be a sufficient argument to motivate the use of a dedicated hardware implementation. Consider for example a specialized Application-Specific Integrated Circuit (ASIC) versus a high-end (workstation-class) processor. The hardware inside of the ASIC can execute many operations in parallel, but the processor runs at a much higher clock frequency. Furthermore, modern processors are very effective in completing multiple operations per clock cycle. As a result, an optimized software program on top of a high-end processor may outperform a quick-and-dirty hardware design job on an ASIC. Thus, the *absolute performance* of software may very well be higher than the absolute performance of hardware. In contrast to relative performance, the absolute performance needs to take clock frequency into account.

1.2.2 Energy Efficiency

There is another metric which is independent from clock frequency, and which can be applied to all architectures. That metric is *energy-efficiency*: the amount of useful work done per unit of energy.

Figure 1.5 shows the example of a particular encryption application (AES) for different target platforms (Refer to Further Reading for source references). The flexibility of these platforms varies from very high on the left to very low on the right. The platforms include: Java on top of a Java Virtual machine on top of an embedded processor; C on top of an embedded processor; optimized assembly-code on top of a Pentium-III processor; Verilog code on top of a Virtex-II FPGA; and an ASIC implementation using 0.18 micron CMOS standard cells. The logarithmic Y-axis
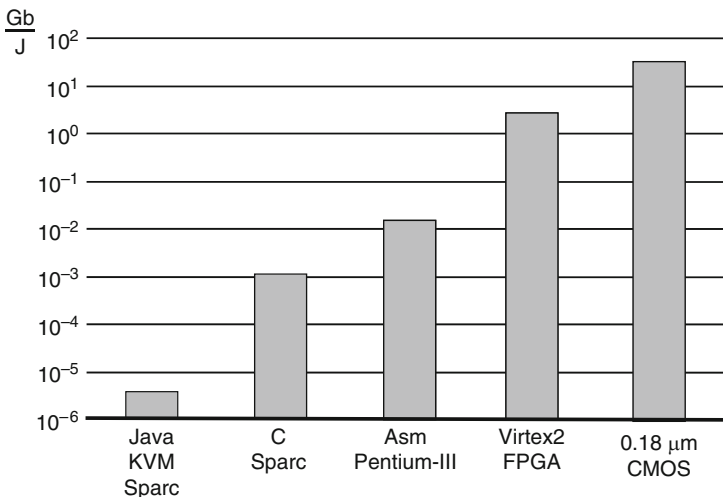
Fig. 1.5 Energy efficiency

shows the amount of gigabits that can be encrypted on each of these platforms using a single Joule of energy. Keep in mind that the application is the same for all these architectures and consists of encrypting bits. As indicated by the figure, the energy-efficiency varies over many *orders of magnitude*. If these architectures are being used in hand-held devices, where energy is a scarce resource, obviously there is a strong motivation to use a less flexible, more specialized architecture. For the same reason, you will never find a high-end workstation processor in a cell phone.

1.3   The Driving Factors in Hardware/Software Codesign

As pointed out in the previous section, energy-efficiency and relative performance are important factors to prefer a (fixed, parallel) hardware implementation over a (flexible, sequential) software implementation. The complete picture, however, is much more complicated. In the design of modern electronic systems, many trade-offs have to be made, often between conflicting objectives. Figure 1.6 shows that some factors argue for *more* software while other factors argue for *more* hardware. The following are arguments in favor of increasing the amount of on-chip dedicated hardware.

- **Performance:** The classic argument in favor of dedicated hardware design has been increased performance: more work done per clock cycle. That is still one of the major factors. Increased performance is obtained by reducing the flexibility of an application or by specializing the architecture that the application is mapped onto. In both cases, this implies the implementation of dedicated hardware components.
- **Energy Efficiency:** Almost every electronic consumer product today carries a battery (iPod, PDA, mobile phone, Bluetooth device, ..). This makes these products energy-constrained. At the same time, such consumer appliances are used for similar applications as traditional high-performance personal computers. In order to become sufficiently energy-efficient, consumer devices are implemented using a combination of embedded software and dedicated hardware components. Thus, a well-known use of hardware–software codesign is to trade function-specialization and energy-efficiency by moving (part of) the flexible software of a design into fixed hardware.
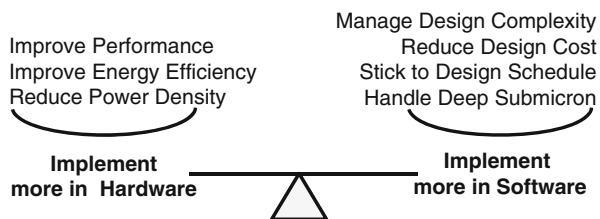
Fig. 1.6   Driving factors in hardware/software codesign

- **Power Densities** of modern high-end processors are such that their performance can no longer be increased by making them run faster. Instead, there is a broad and fundamental shift toward parallel computer architectures. However, at this moment, there is no dominant parallel computer architecture that has shown to cover all applications. Some of the candidates may include symmetric multiprocessors attached to the same memory; FPGAs used as accelerator engines for classic processors; Multicore and many-core architectures such as Graphics Processing Engines with general-purpose compute capabilities. The bottom line is that the software designer will not be able to 'ignore' or 'abstract' the computer architecture in the coming years. This architecture-awareness comes natural with hardware–software codesign.

The following arguments, on the other hand, argue for flexibility and thus for increasing the amount of on-chip software.

- **Design Complexity:** Today, it is common to integrate multiple microprocessors together with all related peripherals and hardware components on a single chip. This approach has been touted system-on-chip (SoC). Modern SoC are extremely complex. The conception of such a component is impossible without a detailed planning and design phase. Extensive simulations are required to test the design upfront, before committing to a costly implementation phase. Since software bugs are easier to address than hardware bugs, there is a tendency to increase the amount of software.
- **Design Cost:** New chips are very expensive to design. As a result, hardware designers make chips programmable so that these chips can be reused over multiple products or product generations. The SoC is a good example of this trend. However, 'programmability' can be found in many different forms other than embedded processors: reconfigurable systems are based on the same idea of reuse-through-reprogramming.
- **Shrinking Design Schedules:** Each new generation of technology tends to replace the older one more quickly. In addition, each of these new technologies is exponentially more complex than the previous generation. For a design engineer, this means that each new product generation brings more work that needs to be completed in a shorter period of time. Shrinking design schedules require engineering teams to work on multiple tasks at once: hardware and software are developed concurrently. A software development team will start software development as soon as the characteristics of the hardware platform are established, even *before* an actual hardware prototype is available.
- **Deep-Submicron Effects:** Designing new hardware from-scratch in high-end silicon processes is difficult due to second-order effects in the implementation. For example, each new generation of silicon technology has an increased variability and a decreased reliability. Programmable, flexible technologies make the hardware design process simpler, more straightforward, and easier to control. In addition, programmable technologies can be created to take the effects of variations into account.

Finding the correct balance between all these factors is obviously a very complex problem. In this book, we do not claim to look for an optimal solution that considers all of them. Instead, our primary cost factors will be: performance versus resource cost. Adding hardware to a software solution may increase the performance of the overall application, but it will also require more resources. In terms of the balance of Fig. 1.6, this means that we will balance *Design Cost* versus *Performance*.

1.4 The Hardware–Software Codesign Space

The trade-offs discussed in the previous section need to be made in the context of a *design space*. For a given application, there are many different possible solutions. The collection of all these implementations is called the hardware–software codesign space. Figure 1.7 gives a symbolic representation of this design space and indicates the main design activities in this design space.

On top is the *application*, and a designer will map this application onto a platform. A *platform* is a collection of programmable components. Figure 1.7 illustrates several different platforms. Mapping an application onto a platform means writing software for that platform, and if needed, customizing the hardware of the platform. The format of the software varies according to the components of the platform.
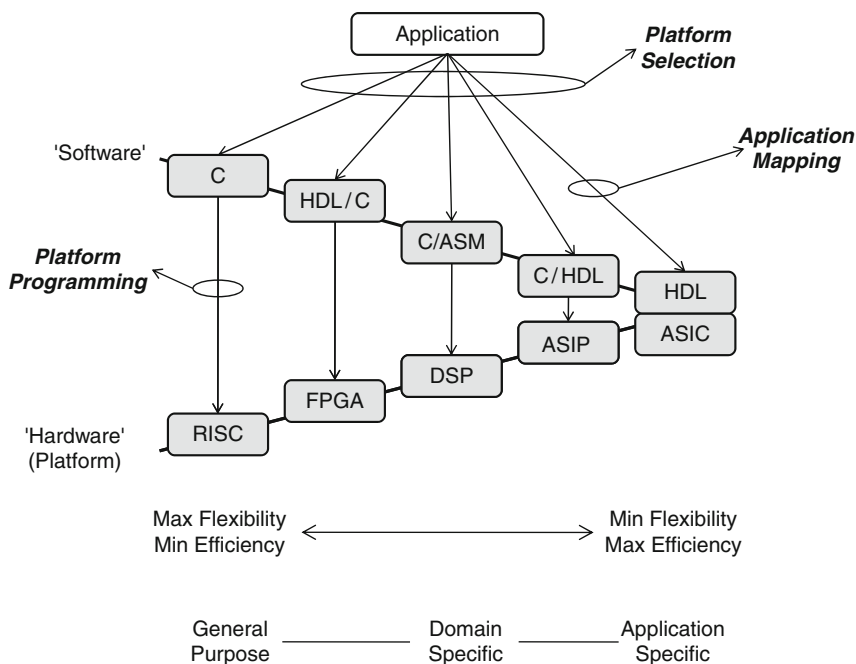
Fig. 1.7 The hardware–software codesign space

For example, a RISC processor may be programmed in C, while an FPGA could be programmed starting from a HDL program. In this section, we will describe the platform design space and we will discuss how an application is mapped onto a platform.

1.4.1 The Platform Design Space

A *specification* is a description of the desired application. A new application could be for example a novel way of encoding audio in a more economical format than current encoding methods. Often, applications start out as informal ideas, uncoupled from the actual implementation. Designers then write C programs to render their ideas in detail. In that case, the specification is a C program. The C program is not yet the final implementation, it is only a description of what the application should do. Very often, a specification is just a piece of English text, typically resulting in ambiguity and open questions.

The objective of the design process is to implement the application on a target platform. In hardware–software codesign, we are interested in using programmable components. Figure 1.7 illustrates several examples: A RISC microprocessor, a FPGA, a DSP, an ASIP, and finally an ASIC. The ASIC is normally not considered as a programmable component, but is included in this list to reflect a boundary in the codesign space.

Software as well as hardware have a very different meaning depending on the platform.

- In the case of a RISC processor, software is written in C, while the hardware is a general-purpose processor.
- In the case of a FPGA, software is written in a HDL. When the FPGA contains a soft-core processor, as discussed above, we will also write additional platform software in C.
- A DSP uses a combination of C and assembly code for software. The hardware is a specalized processor architecture, adapted to signal processing operations.
- An ASIP is a processor that can be specialized to a particular application domain, for example, by adding new instructions and by extending the processor datapath. The 'software' of an ASIP thus can contain C code as well as a hardware description of the processor extensions.
- Finally, in the case of an ASIC, the application is written in HDL, which is then converted into a hardcoded netlist. In contrast to other platforms, ASICs are non-programmable. In an ASIC, the application and the platform have merged to a single entity.

Figure 1.7 also shows a *domain-specific* platform. General-purpose platforms, such as RISC and FPGA, are able to support a broad range of applications. Application-specific platforms, such as the ASIC, are optimized to execute a single application. In between general-purpose and application-specific architectures,

there is a class called *domain-specific* platforms. These platforms are optimized to execute a particular range of applications, also called an *application domain*. Signal-processing, cryptography, networking, are all examples of domains. A domain may have sub-domains. For example, even though signal-processing is considered to be a domain, one could further distinguish voice-signal processing from video-signal processing and devise optimized platforms for each of these cases. The DSP and the ASIP are two examples of domain-specific platforms.

1.4.2 Application Mapping

Each of the above platforms in Fig. 1.7 presents a trade-off between application *flexibility* and platform *efficiency*. The wedge-shape of Fig. 1.7 expresses this idea, and it can be explained as follows.

Flexibility means how well the platform can be adapted to different applications. Flexibility in platforms is desired because it allows designers to make changes to the application after the platform is fabricated. Very flexible platforms, such as RISC and FPGA, are programmed with general-purpose languages. When a platform becomes more specialized, the programming tends to become more specialized as well. We visualize this by drawing the application closer to the platform. This expresses that the software becomes more specific to the hardware.

Different platforms may also provide different levels of efficiency. *Efficiency* can either relate to absolute performance (i.e., time-efficiency) or to the efficiency in using energy to implement computations. For a given application, a specialized platform tends to be more efficient than a general platform because its hardware components are optimized for that application. We can visualize this by moving the platform closer to the application in the case of specialized platforms.

The effect of flexibility-efficiency trade-off on the source code of software can be illustrated with a small example. Consider the execution of the dot-product on a DSP processor such as TI's C64x. In C, the dot-product is a vector operation that can be expressed in single compact loop:

```
sum=0;
for (i=0; i<N; i++)
    sum += m[i]*n[i];
```

Listing 1.5 shows the body of the loop, optimized as assembly code for the TI C64x DSP processor. We only point out why this assembly code is architecture specific. The TI C64x is a highly parallel processor that has two multiply-accumulate units. It can compute *two* loop iterations of the C loop at the same time. In Listing 1.5, several instructions are preceded by ||. Those instructions will be executing in parallel with the previous instructions. Even though Listing 1.5 spans 9 lines, it consists of only *three* instructions. Moreover, the program will complete with only half the amount of iterations of the C program. Thus, Listing 1.5 is more efficient than the original C program, but it is also optimized for a single specialized architecture. A gain in efficiency was obtained at the cost of flexibility (or portability).

Listing 1.5 dot product in C64x DSP processor

```
        LDDW   .D2T2    *B_n++,B_reg1:B_reg0
||      LDDW   .D1T1    *A_m++,A_reg1:A_reg0

        DOTP2  .M2X     A_reg0,B_reg0,B_prod
||      DOTP2  .M1X     A_reg1,B_reg1,A_prod

        SPKERNEL 4, 0
||      ADD    .L2      B_sum,B_prod,B_sum
||      ADD    .L1      A_sum,A_prod,A_sum
```

Platform Programming is the task of mapping software onto hardware. When standard programming languages are used, a compiler (for C) or a synthesis tool (for HDL) will do this job for us. However, many platforms are not just simple components. In that case, the application contains multiple pieces of software, possibly in different programming languages. For example, the platform may consist of a RISC processor and a specialized hardware coprocessor. In the case of a RISC with a coprocessor for example, the software consists of C (for the RISC) as well as dedicated coprocessor instruction-sequences (for the coprocessor). Thus, while Fig. 1.5 suggests that platform programming is just a matter of using automated tools and compilers, the reality may be more complicated.

An interesting, but very difficult question is *how* one can select a platform for a given specification, and *how* one can map an application onto a selected platform. Of these two questions, the first one is the hardest. Designers typically answer it based on their previous experience with similar applications. The second question is also very challenging, but it is possible to answer it in a more systematic fasion, using a design methodology. A *design method* is a systematic sequence of steps to convert a specification into an implementation. Design methods cover many aspects of application mapping, such as optimization of memory usage, design performance, resource usage, precision and resolution of data types, and so on. A design method is a canned sequence of design steps. You can learn it in the context of one design, and next apply this design knowledge in the context of another design.

1.5 The Dualism of Hardware Design and Software Design

In the previous sections, we discussed the driving forces in hardware/software codesign, as well as its design space. Clearly, there are compelling reasons for hardware–software codesign, and there is a significant design space available. A key challenge in hardware–software codesign is that a designer needs to combine two radically different design paradigms. In fact, hardware and software are each other's dual in many respects. In this section, we examine these fundamental differences. Table 1.1 provides a synopsis.

Table 1.1 The dualism of hardware and software design

	Hardware	Software
Design paradigm	Decomposition in space	Decomposition in time
Resource cost	Area (# of gates)	Time (# of instructions)
Constrained by	Time (clock cycle period)	Area (CPU instruction set)
Flexibility	Must be designed-in	Implicit
Parallelism	Implicit	Must be designed-in
Modeling	Model $\neq$ Implementation	Model $\sim$ Implementation
Reuse	Uncommon	Common

- **Design Paradigm:**
  In a hardware model, circuit elements operate in parallel. Thus, by using more circuit elements, more work can be done within a single clock cycle. Software, on the other hand, operates sequentially. By using more operations, a software program will take more time to complete. Thus, a hardware designer solves problems by *spatial decomposition*, while a software designer solves problems by *temporal decomposition*.
- **Resource Cost:**
  The dualism in decomposition methods leads a similar dual resource cost. Decomposition in space, as used by a hardware designer, means that more gates are required for when a more complex design needs to be implemented. Decomposition in time, as used by a software designer, implies that a more complex design will take more instructions to complete. Therefore, the resource cost for hardware is circuit *area*, while the resource cost for software is execution *time*.
- **Design Constraints:**
  There is a similar dualism in terms of design constraints. A hardware designer is constrained by the clock cycle period of a design. A software designer, on the other hand, is limited by the capabilities of the processor instruction set and the memory space available with the processor. Thus, the design constraints for hardware are in terms of a *time* budget, while the design constraints for software are fixed by the CPU. So, a hardware designer invests circuit area to maintain control over execution time, and a software designer invests execution time for an almost constant circuit area.
- **Flexibility:**
  Software excels over hardware in the support of application flexibility. Flexibility is the ease by which the application can be modified or adapted after the target architecture for that application is manufactured. In software, flexibility is essentially free. In hardware on the other hand, flexibility is not trivial. Hardware flexibility requires that circuit elements can be easily reused for different activities or functions in a design. A hardware designer has to think carefully about such reuse: flexibility needs to be designed into the circuit by means of multiplexers and additional control signals!

- **Parallelism:**
  A dual of flexibility can be found in the ease with which parallel implementations can be created. Parallelism is the most obvious approach to improving performance. For hardware, parallelism comes for free as part of the design paradigm. For software, on the other hand, parallelism is a major challenge. If only a single processor is available, software can only implement concurrency, which requires the use of special programming constructs such as threads. When multiple processors are available, a truly parallel software implementation can be made, but interprocessor communication and synchronization become a challenge.
- **Modeling:**
  In software, modeling and implementation are very close. Indeed, when a designer writes a C program, the compilation of that program for the appropriate target processor will also result in the implementation of the program! In hardware, the model and the implementation of a design are distinct concepts. Initially, a hardware design is modeled using a HDL. Such a hardware description can be simulated, but it is not an implementation of the actual circuit. Hardware designers use a hardware *description* language, and their programs are models which are later transformed to implementation. Software designers use a software *programming* language, and their programs are an implementation by itself.
- **Reuse:**
  Finally, hardware and software are also quite different when it comes to Intellectual Property Reuse or IP-reuse. The idea of IP-reuse is that a component of a larger circuit or a program can be packaged, and later reused in the context of a different design. In software, IP-reuse has known dramatic changes in recent years due to open source software and the proliferation of open platforms. When designing a complex program these days, designers will start from a set of standard libraries that are well-documented and available on a wide range of platforms. For hardware design, IP-reuse is – to put it bluntly – still in its infancy. Hardware Designers are only starting to define standard exchange mechanisms. IP-reuse of hardware has a long way to go compared to the state of reuse in software.

This summary comparison indicates that in many aspects, hardware design and software design use dual concepts. Hence, being able to effectively transition from one world of design to the other is an important asset to your skills as a computer engineer. You can try to combine the best of both worlds. Complex control processing? Use a software core. Hard-real-time processing requirement? Add some hardware. In this book, we will focus on this dualism and transition from concepts in hardware design to concepts in software design and back. Our objective is not only to excel as a hardware designer or a software designer; our objective is to excel as a system designer.

1.6 More on Modeling

As discussed earlier, we will use single-clock synchronous hardware and single-thread sequential software to discuss hardware–software codesign. Yet, even for these simple cases, there are many different approaches to describe these models, at different abstraction levels.

1.6.1 Abstraction Levels

We will differentiate the abstraction levels based on their time-granularity. A smaller time-granularity typically implies that activities are expressed in more detail. There are five abstraction levels commonly used by computer engineers for the design of electronic hardware–software systems. Starting at the lowest abstraction level, we enumerate the five levels. Figure 1.8 illustrates the hierarchy among these abstraction levels.

1. **Continuous Time:** At the lowest abstraction level, we describe operations as continuous actions. For example, electric networks can be described as systems of interacting differential equations. Solving these equations leads to an estimate for voltages and currents in these electric networks. This is a very detailed level, useful to analyze analog effects. However, this level of abstraction is not used to describe typical hardware–software systems.
2. **Discrete-event:** At the next abstraction level, we lump activity at discrete points in time called events. Those events can be possibly irregularly spaced. For example, when the inputs of a digital combinatorial circuit change, the effect of those changes will ripple from inputs to outputs, toggling nets at intermediate circuit nodes as they change. The changes on those nodes will be delayed by one or more gate propagation-delays. Discrete-event simulation is very popular to model hardware at low abstraction level. It gets rid of the differential equations and the complexity of continuous-time simulation, yet it captures all relevant information such as glitches and clock cycle edges. Discrete-event simulation is also used to model systems at high abstraction level, to simulate abstract event with irregular spacing in time. For example, discrete-event simulation can be used to model customer queues at a bank. In the context of hardware–software system design however, discrete-event modeling refers to a low abstraction level.
3. **Cycle-accurate:** Single-clock synchronous hardware circuits have the important property that all interesting things happen at regularly spaced intervals, namely at the clock edge. This abstraction is important enough to merit its own abstraction level, and it is called cycle-accurate modeling. A cycle-accurate model does not capture propagation delays or glitches. All activities that fall 'in between' clock edges are concentrated at the clock edge itself. As a result, activities happen either immediately (for combinatorial circuits for example), or else after an integral number of clock cycles (for sequential circuits). The cycle-accurate level is
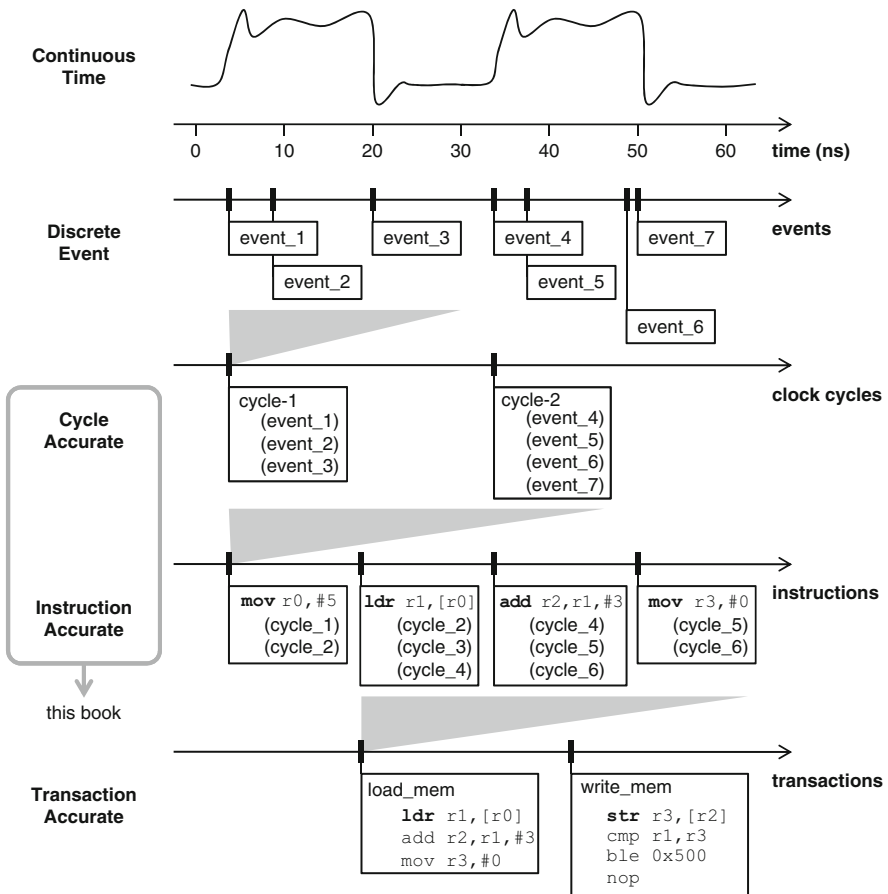
Fig. 1.8 Abstraction levels for hardware–software codesign models

very important for hardware–software system modeling and very often serves as
the 'golden reference' for a hardware–software implementation. Cycle-accurate
modeling will be used extensively throughout this book.

4. **Instruction-accurate:** RTL models are great but may be too slow for com-
   plex systems. For example, your laptop has a processor that probably clocks
   over 1 GHz (one billion cycles). Assuming that you could write a C func-
   tion that expresses a single clock cycle of processing, you would have to call
   that function one billion times to simulate just a single second of processing.
   Clearly, further abstraction can be useful to build leaner and faster models. The
   instruction-accurate modeling level expresses activities in steps of one micropro-
   cessor instruction. Each instruction lumps together several cycles of processing.
   Instruction-accurate simulators are used to verify complex software systems,
   such as complete operating systems. Instruction-accurate simulators count time

in terms of an instruction count, but not a cycle count. Thus, this abstraction level may not show you the real-time performance of a model – unless you are able to map instructions back to clock cycles.

5. **Transaction-accurate:** For very complex systems, even instruction-accurate models may be too slow or require too much modeling effort. For these models, yet another abstraction level is introduced: the transaction-accurate level. In this type of model, the behavior is expressed in terms of the interactions between the components of a system. These interactions are called transactions. For example, one could model a system with a disk drive and a user application (as discussed earlier), and create a simulation that focuses on the commands exchanged between the disk drive and the user application. A transaction-accurate model allows considerable simplification of the disk drive and the user application. Indeed, in between two transactions, millions of instructions can be lumped together and simulated as a single, atomic function call. Transaction-accurate models are important in the exploratory phases of a design, where a designer is interested in defining the overall characteristics of a design without going through the effort of developing detailed models.

In summary, there are five abstraction levels that are commonly used for hardware–software modeling: transaction-accurate, instruction-accurate, cycle-accurate, event-driven, and continuous-time. We are most interested in the instruction-accurate and cycle-accurate levels.

1.7 Concurrency and Parallelism

Concurrency and parallelism are terms that often occur in the context of hardware–software codesign. They mean very different things. Concurrency is the ability to execute simultaneous operations because these operations are completely independent. Parallelism is the ability to execute simultaneous operations because the operations can run on different processors or circuit elements. Thus, concurrency relates to an application model, while parallelism relates to the implementation of that model.

Hardware is always parallel. Software on the other hand can be sequential, concurrent, or parallel. Sequential and concurrent software requires a single processor, parallel software requires multiple processors. The software running on your laptop (email, WWW, word processing, and so on) is concurrent. The software running on the 65536-processor IBM Blue Gene/L is parallel.

An important incentive for a software designer to use hardware–software codesign is the ability it provides to create a parallel implementation. Making efficient use of the parallelism in the architecture implies that you also have a specification available that contains enough concurrency.

There is a well-known law in supercomputing, called Amdahl's law, that states that the maximal speedup for any application that contains q % sequential code is 1 / (q/100). For example, if your application is 33% of its runtime executing like

a sequential process, the maximal speedup is 3. So given enough hardware, you can make the parallel part of the application run arbitrarily fast. The sequential part however still has to run step by step and does not benefit from parallelism. Hence, the speedup cannot be higher than 3. Thus, you see that we don't only need to have parallel platforms, we also need a way to write parallel programs to run on those platforms. This is not obvious. C programs for example are sequential, and so are typical instruction-set architectures.

Surprisingly, even algorithms that seem sequential at first can be executed (and specified) in a parallel fashion. The following examples are discussed by Hillis and Steele. They describe the 'Connection Machine' (CM), a massively parallel processor. The CM contains a network of processors, each with their own local memory, and each processor in the network is connected to each other processor. The original CM machine contained 65536 processors, each of them with 4Kbits of local memory. Interestingly, while the CM dates from the 1980s, multiprocessor architectures recently regained a lot of interest with the modern design community. Figure 1.9 illustrates an 8-node CM.

The question relevant to our discussion is: how hard is it to write programs for such a CM? Of course you can write individual C programs for each node in the network, but that is not easy, nor is it very scalable. Remember that the original CM had 64K nodes! Yet, as Hillis and Steele have shown, it is possible to express algorithms in a concurrent fashion, such that they can map to a CM. Consider taking the sum of an array of numbers, illustrated in Fig. 1.10. To take the sum, we distribute the array over the CM processors so that each processor holds one number. We can now take the sum over the entire array in $log(n)$ steps (n being the number of processors) as follows. We perform 2^i parallel additions per time step, for i going from $log(n-1)$ down to 0. For example, the sum of 8 numbers can be computed in three time steps on a CM machine. In the following figure, time steps run vertically and each processor is drawn left to right. The communication activities between the processors are represented by means of arrows.
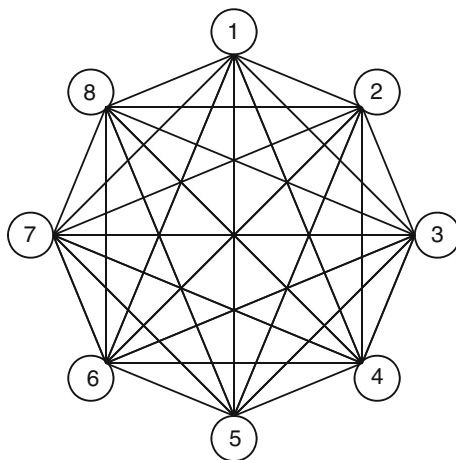
Fig. 1.9 Eight node connection machine (CM)
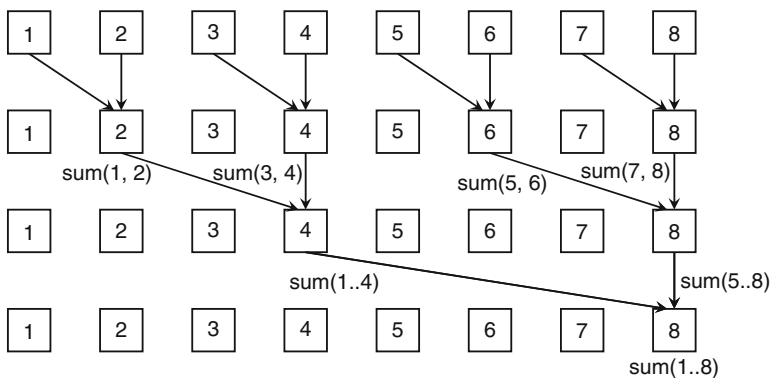
Fig. 1.10 Parallel sum
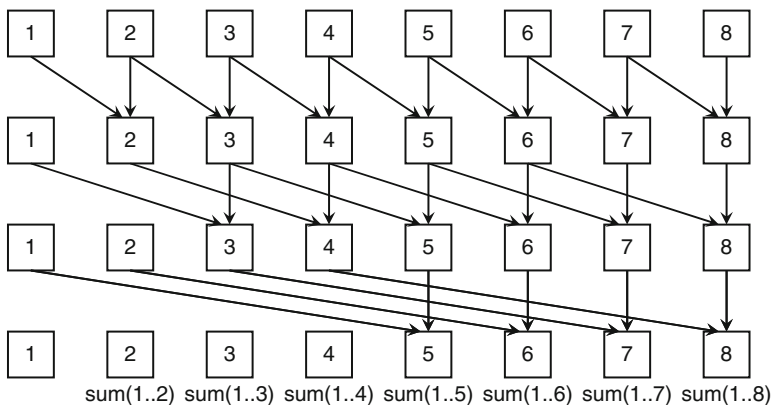
Fig. 1.11 Parallel partial sum

Compare the same algorithm running on a sequential processor. In that case, the array of numbers would be stored in a single memory and the processor needs to iterate over each element, requiring a minimum of 8 time steps! You can also see that the parallel sum still wastes a lot of potential computing power. We have in total $3 * 8 = 24$ computation time-steps available, and we are only using 7 of them. One extension of this algorithm is to evaluate all partial sums (i.e., the sum of the first two, three, four, etc numbers). A parallel algorithm that performs this in 3 time-steps, using 17 computation time-steps, is shown in Fig. 1.11.

For hardware–software codesign, the importance of concurrent specifications is the following. If you develop a concurrent specification, you will be able to make use of a parallel implementation. In contrast, if you restrict yourself to a sequential specification from the start, it will be much harder to make use of parallel hardware. Consequently, do not settle for a sequential language (such as C) as a universal specification mechanism. The use of C is excellent to make an executable, functional

model of what you want to build. But other concurrent specification mechanisms (such as data-flow, which we will discuss in the next chapter) may be better suited as a starting point for parallel implementation than C.

1.8 Summary

Thanks to Moore's law, chip companies are able to offer us an ever growing selection of programmable components. In this introductory chapter, we have defined hardware/software codesign as the partitioning and design of an application in terms of fixed and flexible parts. The flexible parts run as programs on those programmable components. Traditional microprocessors are only one of the many options, and we briefly described other components including FPGAs, DSPs and ASIPs. Platform selection is the job of determining which programmable component (or combination of components) is the best choice for a given application. Application mapping is the effort of transforming a specification into a program. Platform programming is the effort of converting an application program into low-level instructions for each programmable component. We also discussed the modeling abstraction levels for hardware and software, and we highlighted cycle-based synchronous RTL for hardware, and single-thread C for software as the golden level for this book. Finally, we also made careful distinction between a parallel implementation and a concurrent program.

1.9 Further Reading

Many authors have pointed out the advantages of dedicated hardware solutions when it comes to Energy Efficiency. A very comprehensive coverage of the problem can be found in Rabaey (2009).

Figure 1.4 is based on results published between 2003 and 2008 by various authors including Good and Benaissa (2007), Bogdanov et al. (2007), Satoh and Morioka (2003), Leander et al. (2007), Kaps (2008), Meiser et al. (2007), Ganesan et al. (2003), and Karlof et al. (2004).

Further discussion on the driving forces that require chips to become programmable is found in Keutzer et al. (2000). A very nice and accessible discussion of what that means for the hardware designer is given in Vahid (2003).

Hardware–software codesign, as a research area, is at least two decades old. Some of the early works are collected in Micheli et al. (2001), and a retrospective of the main research problems is found in Wolf (2003). Conferences such as the *International Conference on Hardware/Software Codesign and System Synthesis* (CODES+ISSS) cover the latest evolutions in the field.

Hardware/software codesign, as an educational topic, is still evolving. A key challenge seems to be to find a good common ground to jointly discuss hardware

and software. Interesting ideas can be found, for example in Madsen et al. (2002) and Vahid (2007b).

Despite its age, the paper on data-parallel algorithms by Hillis and Steel is still a great read Hillis and Steele (1986).

1.10 Problems

1.1. Use the World Wide Web to find a data sheet for the following components. What class of components are they (RISC/FPGA/DSP/ASIP/ASIC)? How does one write software for each of them? What tools are used to write that software?

- `TMS320DM6437`
- `EP3C25`
- `Xtensa LX2`
- `ADSP-BF526`
- `XC5VFX100T`

1.2. Develop a sorting algorithm for an 8-node CM, which can handle up to 16 numbers. Show that an N-node CM can complete the sorting task in a time propertional to N.

1.3. A single-input, single-output program running on an ARM processor needs to be rewritten such that it will run on three parallel ARM processors. As shown in Fig. 1.12, each ARM has its own, independent data- and instruction memory. For this particular program, it turns out that it can be easily rewritten as a sequence of three functions fA, fB, and fC which are also single-input, single-output. Each of these three functions can be executed on a separate ARM processor so that we get an arrangement as shown below. The sub-functions fA, fB, and fC contain 40, 20, and 40%, respectively of the instructions of the original program. You can ignore the time needed for communication of variables (out, in, t1, and t2 are integers).
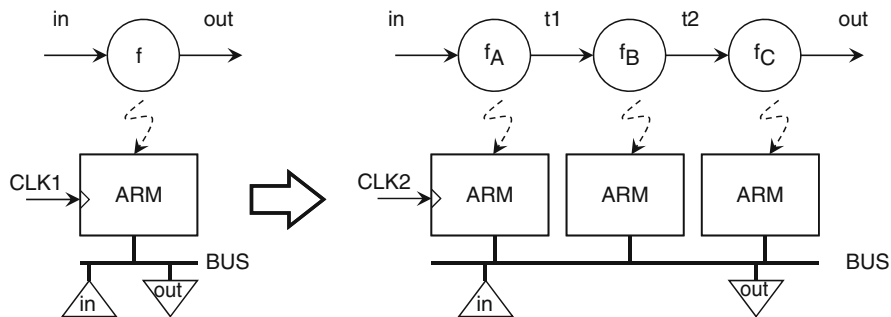
Fig. 1.12 Multiprocessor system for Problem 1.3

(a) Assume that all ARMs have the same clock frequency (CLK1 = CLK2). Find the maximal speedup that the parallel system offers over the single-ARM system. For example, a speedup of 2 would mean that the parallel system could process two times as much input data as the single-ARM system in the same amount of time.

(b) For the parallel system of three ARM described above, we can reduce the power consumption by reducing their clock frequency CLK and their operating voltage V. Assume that both these quantities scale linearly (i.e., Reducing the Voltage V by half implies that the clock frequency must be reduced by half as well). We will scale down the voltage/clock of the parallel system such that the scaled-down parallel system has the same performance as the original, single-ARM sequential system. Find the ratio of the power consumption of the original sequential system to the power consumption of the scaled-down, parallel system (i.e., find the power-savings factor of the parallel system). You only need to consider dynamic power consumption. Recall that Dynamic Power Consumption is proportional to voltage and clock frequency.

1.4. Describe a possible implementation for each of the following C statements in hardware. You can assume that all variables are integers and that each of them is stored in a register.

(a) `a = a + 1;`
(b) `if (a > 20) a = 20;`
(c) `while (a < 20) a = a + 1;`

1.5. The function in Listing 1.6 implements a CORDIC algorithm. It evaluates the cosine of a number with integer arithmetic and using only additions, subtractions,

Listing 1.6 Listing for Problem 1.5.

```c
int cordic_cos(int target) {
  int X, Y, T, current;
  unsigned step;
  X       = AG_CONST;
  Y       = 0;
  current = 0;
  for(step=0; step < 20; step++) {
    if (target > current) {
      T             =   X - (Y >> step);
      Y             = (X >> step) + Y;
      X             = T;
      current    += angles[step];
    } else {
      T             = X + (Y >> step);
      Y             = -(X >> step) + Y;
      X             = T;
      current    -= angles[step];
    }
  }
  return X;
}
```

and comparisons. The `angles []` variable is an array of constants. Answer each of the following questions. Motivate your answer.

• Do you think it is possible to implement this function on any architecture within 1,000 clock cycles?
• Do you think it is possible to implement this function on any architecture within 1,000 μs?
• Do you think it is possible to implement this function on any architecture within 1 clock cycle?
• Do you think it is possible to implement this function on any architecture within 1 μs?

Chapter 2
Data Flow Modeling and Implementation

Abstract In this chapter, we will learn how to create data flow models, and how to implement those models in hardware and software. Unlike C programs, data flow models are concurrent: they can express activities that happen simultaneously. This property makes data flow well suited for a parallel hardware implementation as well as a sequential software implementation.

2.1 The Need for Concurrent Models: An Example

By nature, hardware is parallel and software is sequential. As a result, software models (C programs) are not very well suited to capture hardware implementations, and vice versa, hardware models (RTL programs) are not a good abstraction to describe software. However, designers frequently encounter situations in which they cannot predict if the best solution for a design problem is a hardware implementation or a software implementation. Trying to do both (writing a full C program *as well as* a full RTL program) is not on option; it requires the designers to work twice as hard.

In signal processing, this problem is well known. Signal processing domain experts are used to describe complex systems, such as digital radios and radar processing units, using *block diagrams*. A block diagram is a high-level representation of the target system as a collection of smaller functions. A block diagram does not specify if a component should be hardware or software; it only shows a chain of signal processing algorithms and the data samples to send to each other. We are specifically interested in *digital* signal processing systems. Such systems represent signals as streams of discrete samples rather than continuous signal shapes.

Figure 2.1a shows the block diagram for a simple digital signal processing system. It's a pulse-amplitude modulation (PAM) system, and it is used to transmit digital information over bandwidth-limited channels. A PAM signal is created from binary data in two steps. First, each word in the file needs to be mapped to PAM symbols, which are just pulses of different heights. An entire file of words will thus be converted to a stream of symbols or pulses. Next, the stream of pulses needs to be converted to a smooth shape using pulse-shaping. Pulse-shaping ensures that the bandwidth of the resulting PAM signal bandwidth does not exceed the PAM symbol
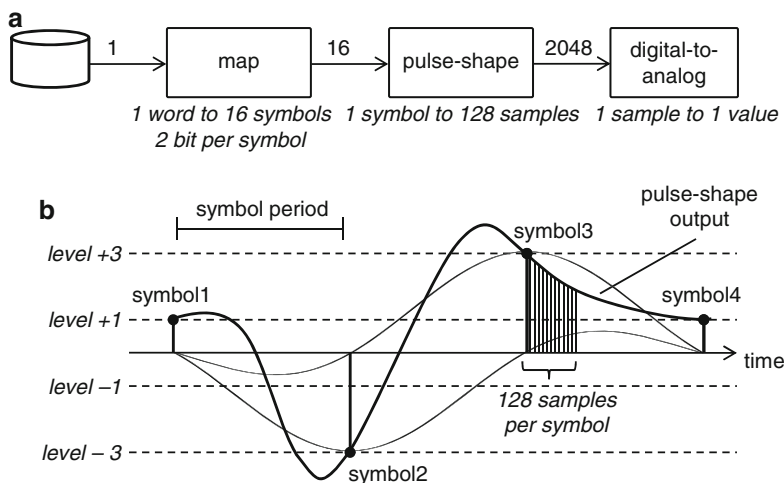
Fig. 2.1 (**a**) Pulse-amplitude modulation (PAM) system (**b**) Operation of the pulse-shaping unit

rate. For example, if a window of 1,000 Hz transmission bandwidth is available, then we can transmit 1,000 PAM symbols per second. In a digital signal processing system, a *smooth* curve is achieved by *oversampling*: calculating many closely-spaced discrete samples. The output of the pulse-shaping unit produces many samples for each input symbol pulse, but it is still a stream of discrete samples. The final module in the block diagram is the digital-to-analog module, which will convert the stream of discrete samples into a continuous signal.

Figure 2.1a shows a PAM-4 system, which uses four different symbols. Since there are four different symbols, each PAM symbol holds 2 bits of source information. A 32-bit word from a binary file thus needs 16 PAM-4 symbols. The first block in the PAM transmission system makes the conversion of a single word to a sequence of sixteen PAM-4 symbols. Figure 2.1b shows that each PAM-4 symbol is mapped to a pulse with four possible signal levels: $\{-3, -1, 1, 3\}$. Once the PAM-4 signals are available, they are shaped to a smooth curve using a pulse-shape filter. The input of this filter is a stream of symbol pulses, while the output is a stream of samples at a much higher rate. In this case, we generate 128 samples for each symbol.

Figure 2.1b illustrates the operation of the pulse-shape filter. The smooth curve at the output of the pulse-shape filter connects the top of each pulse. This is achieved by an interpolation technique, which extends the influence of a single symbol pulse over many symbol periods. The figure illustrates two such interpolation curves, one for `symbol2` and the other for `symbol3`. The pulse-shape filter will produce 128 samples for each symbol entered into the pulse-shape filter.

Now let us consider the construction of a simulation model for this system. We focus on capturing its functionality, and start with a C program as shown in Listing 2.1. We ignore the implementation details of the function calls right now and only focus on the overall structure of the program.

Listing 2.1 C example

```
1   extern int read_from_file();
2   extern int map_to_symbol(int, int);
3   extern int pulse_shape(int, int);
4   extern void send_to_da(int);
5
6   int main() {
7     int word, symbol, sample;
8     int i, j;
9     while (1) {
10      word = read_from_file();
11      for (i=0; i<16; i++) {
12        symbol = map_to_symbol(word, i);
13        for (j=0; j<128; j++)
14          sample = pulse_shape(symbol, j);
15          send_to_da(sample);
16      }
17    }
18  }
```
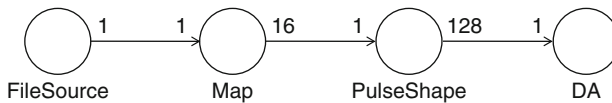

Fig. 2.2 Dataflow model for the PAM system

The program in Listing 2.1 is fine as a system simulation. However, as a model for the implementation, this C program is too strict, since it enforces a *sequential* execution of all functions. If we observe Fig. 2.1a carefully, we can see that the block diagram does not require a sequential execution of the symbol mapping function and the pulse shaping function. The block diagram *only* specifies the flow of data in the system but not the execution order of the functions. The distinction is subtle but important. For example, in Fig. 2.1a, it is possible that the map module and the pulse-shape module work in parallel, each on a different symbol. In Listing 2.1 on the other hand, the map_to_symbol() function and the pulse_shape() function will *always* execute sequentially. In hardware–software codesign, the implementation target could be either parallel or else sequential. The program in Listing 2.1 favors a sequential implementation, but it does not encourage a parallel implementation in the same manner as a block diagram.

In this chapter we will discuss a modeling technique, called Data Flow, which avoids this problem. Data Flow models closely resemble block diagrams. The PAM-4 system, as a Data Flow model, is shown in Fig. 2.2. In this case, the different functions of the system are mapped as individual entities or *actors* such as FileSource, Map, PulseShape, and DA. These actors are linked through communication channels or *queues*. The numbers at the input and output of each actor indicate the relative rate of communications. For example, there are 16 samples

produced by Map for each input sample. Each actor is an *independent unit*. It continuously checks its input for the availability of data, and as soon as data appear, calculates the corresponding output, and sends the result to the next actor in the chain. In the remainder of this chapter, we will discuss the precise construction details of dataflow diagrams. For now, we only point out the major differences of this modeling style compared to modeling in C.

- The strongest point of Data Flow models, and the main reason why signal processing engineers love to use them, is that a Data Flow model is a *concurrent* model. Indeed, the actors in Fig. 2.2 operate and execute as individual concurrent entities. A concurrent model can be mapped to a parallel or a sequential implementation, and so they can model hardware targets as well as software targets.
- Data Flow models are distributed, and there is no need for central controller or 'conductor' in the system to keep the individual system components in pace. In Fig. 2.2, there is no central controller that tells the actors when to operate; each actor can determine for itself when it is time to work.
- Data Flow models are modular. We can develop a design library of data flow components and then use that library in a plug-and-play fashion to construct data flow systems.
- Data Flow systems are easy to analyze, and properties such as deadlock and stability can be evaluated based on inspection of the model. This is an important advantage, which is not at all obvious for C programs or hardware circuit descriptions. In fact, a designer typically does not know if a C program will work or not until the program runs.

Data Flow has been around for a surprisingly long time, yet it has been largely overshadowed by the stored-program (Von Neumann) computing model. Data Flow concepts have been explored since the early sixties. By 1974, Jack Dennis had developed a language for modeling data flow and described data flow using graphs, similar to our discussion in this chapter. In the 1970s and 1980s, an active research community was building not only data flow-inspired programming languages and tools but also computer architectures that implement data flow computing models. Today, data flow remains very popular to describe signal processing systems. For example, commercial tools such as Simulink® are based on the ideas of data flow. An interesting example of an academic environment is the Ptolemy project at UC Berkeley (http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm). The Ptolemy design environment can be used for many different types of system specification, including data flow. The examples on the website can be run inside of a web browser as Java applets.

In the following sections, we will consider the elements that make up a data flow model, and we will discuss a technique for formal analysis of data flow models called Synchronous Data Flow (SDF) graphs. After that, we look into systematic conversion of SDF graphs into a hardware or software implementation.

2.1.1 Tokens, Actors, and Queues

Figure 2.3 shows the data flow model of a simple addition.

- *Actors* contain the actual operations. Actors have a bounded behavior (meaning that they have a precise beginning and ending), and they iterate that behavior from start to completion. One such iteration is called an actor firing. In the example above, each actor firing would execute a single addition.
- *Tokens* carry information from one actor to the other. A token has a value, such as '1', '4', '5', and '8' in Fig. 2.2.
- *Queues* are unidirectional communication links that transport tokens from one actor to the other. Data Flow queues have an infinite amount of storage so that tokens will never get lost in a queue. Data Flow queues are first-in first-out. In Fig. 2.3, there are two tokens in the upper queue, one with value '1' and one with value '4'. The '4' token was entered first into the queue, the '1' token was entered after that. When the 'add' actor will read a token from that queue, the actor will first read the token with value '4' and then the token with value '1' (Fig. 2.4).
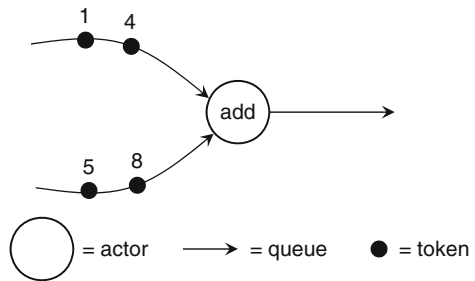
Fig. 2.3 Data flow model of an addition
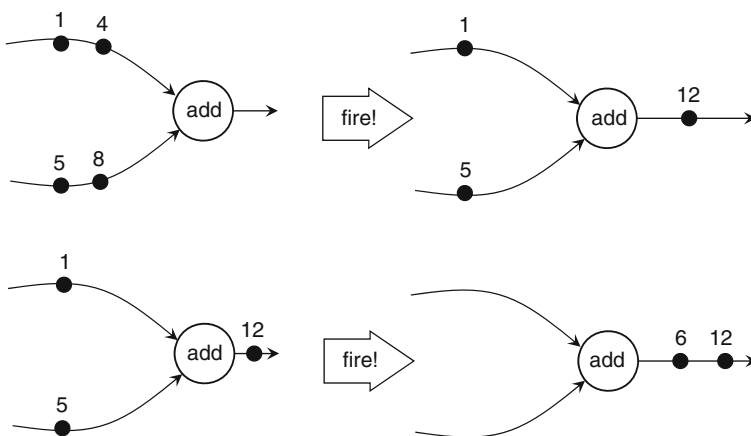
Fig. 2.4 Actor firing moves a data flow model through markings

When a data flow model executes, the actors will read tokens from their input queues, transform the values of the input tokens into output values, and generate new tokens on their output queues. Each single execution of an actor is called the firing of that actor. Data flow execution then is expressed as a sequence of (possibly concurrent) actor firings.

Conceptually, data flow models are untimed. The firing of an actor takes zero time, even though any real implementation of an actor will require a certain amount of time. Untimed means that time is irrelevant for data flow models: the execution is guided only by the presence of data. An actor will never fire if there is no input data, but instead it will wait until data becomes available at its inputs.

A graph with tokens is called a marking of a data flow model. When a data flow model executes, the graph goes through a series of markings that drive data from the inputs of the data flow model to the outputs. Each marking corresponds to a different state of the system, and the execution of a data flow model is determined by the order in which each marking appears. To an external observer, this marking (i.e., the distribution of the tokens on the queues) is the only observable state in the system. This has a subtle side-effect: the behavior of an actor cannot contain internal state variables. A work-around is to express state variables as tokens.

2.1.2 Firing Rates, Firing Rules, and Schedules

When should an actor fire? The conditions under which an actor fires are called the firing rule of that actor. Simple actors such as the add actor can fire when there is a single token on each of its queues. A firing rule thus involves testing the number of tokens present on the input queues. The required number of tokens can be symbolically indicated next to the actor input. Similarly, the amount of tokens that an actor produces per firing can be written next to the actor output. These numbers are called the token consumption rate (at the actor inputs) and token production rate (at the actor outputs). The production/consumption rates of the add actor therefore could be written like as shown in Figs. 2.5 or 2.6.

Data Flow actors may also consume more then one token per actor firing. When they do, we have a multirate data flow model. For example, the actor in Fig. 2.7 has a consumption rate of 2 and a production rate of 1. It will consume two tokens per firing from its input, add them together, and produces the result at the output.
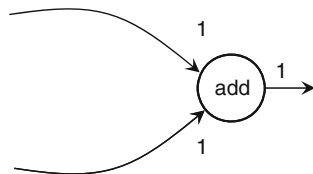
Fig. 2.5 Data flow actor with production/consumption rates

Fig. 2.6 Can this model
do any useful computations?
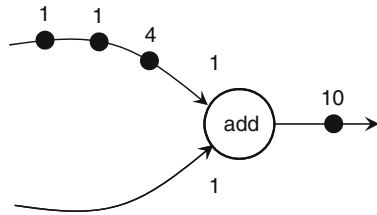
Fig. 2.7 Example of a multi-rate data flow model

2.1.3 Synchronous Data Flow Graphs

When the number of tokens consumed/produced per actor firing is a fixed and constant value, the resulting class of systems are called synchronous data flow graphs or SDF graphs. The term synchronous means that the token production and consumption rates are known, fixed numbers. The SDF semantics are not universal. For example, not every C program can be translated to an equivalent SDF graph.

On the other hand, SDF graphs can be formally analyzed. By considering the structure of an SDF graph, and the production and consumption rates on the actor inputs and outputs, we can predict if the graph can result in a working, stable implementation or not. This means that the schedule of an SDF graph, if it exists, can be predicted completely beforehand. We will discuss a technique that allows one to find such a schedule automatically.

2.1.4 SDF Graphs are Determinate

Assuming that each SDF actor implements a deterministic function, then the entire SDF graph is determinate. This means that the results computed in an SDF graph will always be the same, regardless of the actual firing order of the actors. Figure 2.8 illustrates this property. This graph contains actors with unit production/consumption rates. One actor adds tokens, the second actor increments the value of tokens. As we start firing actors, tokens are transported through the graph. After the first firing, an interesting situation occurs: both the add actor as well as the plus1 actor can fire. Going down at the left side, we assume that the plus1 actor fires first. Going down at the right side, we assume that the add actor fires first. However, regardless of this choice, the graph eventually converges to the situation shown at the bottom.

Why is this property so important? Assume for a moment that the add actor and the plus1 actor execute on two different processors, a slow one and a fast one. Depending upon which actor runs on the fast processor, the SDF execution will
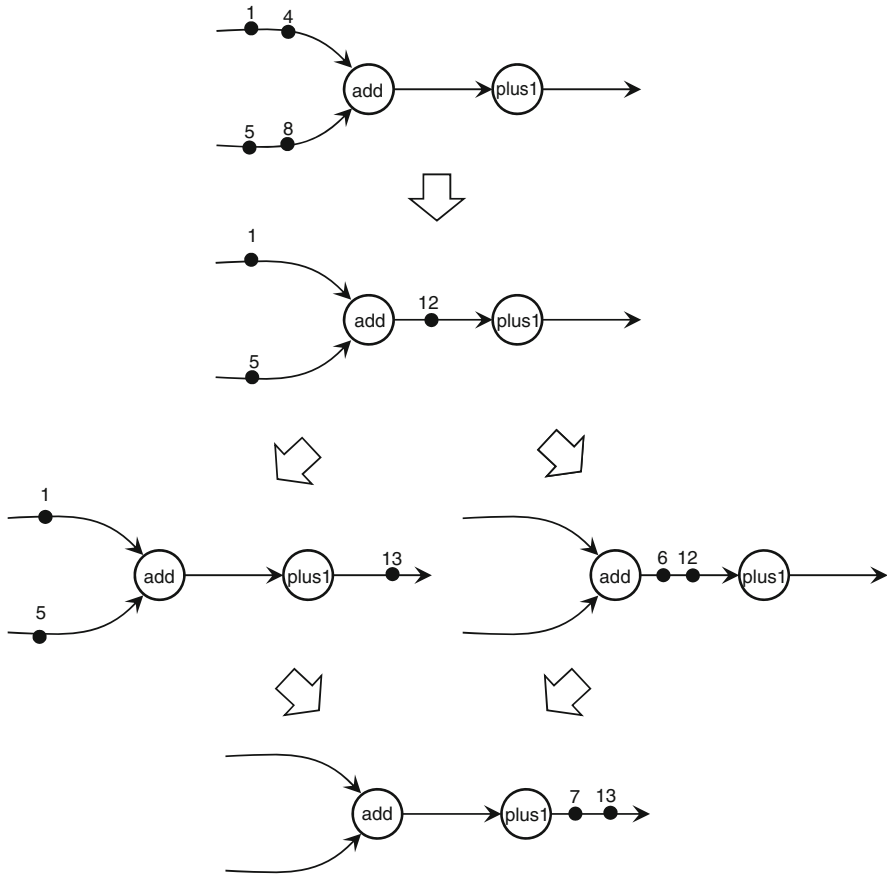
Fig. 2.8 SDF graphs are determinate

come down through the left or the right path in the above figure. Thanks to the determinate property of SDF, it doesn't matter which processor runs on what actor: the results will be always the same. In other words, no matter what technology we are using to implement actors, the system will work as specified as long as we implement the firing rules correctly. Determinate behavior is vital in many embedded applications, especially in applications that involve risk.

2.2 Analyzing Synchronous Data Flow Graphs

An admissible schedule for an SDF graph is one that can run forever without deadlock or without storing an infinite amount of tokens on one of the communication queues. A deadlock situation occurs if, at a given moment, it is no longer possible to fire any actor.
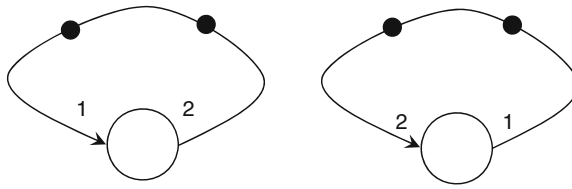
Fig. 2.9 Which SDF graph will deadlock, and which is unstable?

Here are two SDF graphs where these two problems are apparent. Considering Fig. 2.9, which graph will deadlock, and which graph will result in an infinite amount of tokens?

Is it possible to create a systematic method that can show the absence of deadlock or token build-up for an arbitrary SDF topology? It turns out that this is the case. We will study the method of Lee to create so-called Periodic Admissible Sequential Schedules (PASS). A PASS is defined as follows:

- A schedule is the order in which the actors must fire.
- An admissible schedule is a firing order that will not cause deadlock or token-build-up.
- Finally, a periodic admissible schedule is a schedule that can continue forever, because it is periodic (meaning that after some time, the same marking sequence will start). We will consider Periodic Admissible Sequential Schedules, or PASSs for short. Such a schedule requires only a single actor to fire at a time. A PASS would be used, for example, to execute an SDF model on top of a microprocessor.

2.2.1 Deriving Periodic Admissible Sequential Schedules

We can create a PASS for an SDF graph (and test if one exists) with the following four steps.

1. Create the topology matrix G of the SDF graph
2. Verify the rank of the matrix to be one less than the number of nodes in the graph
3. Determine a firing vector
4. Try firing each actor in a round robin fashion, until it reaches the firing count as specified in the firing vector

We will demonstrate each of these steps using the example of the three-node SDF graph shown in Fig. 2.10.

Step 1. Create a topology matrix for this graph. This topology matrix has as many rows as there are graph edges (FIFO queues) and as many columns as there are graph nodes. The entry (i, j) of this matrix will be positive if the node j produces tokens into graph edge i. The entry (i, j) will be negative if the node j consumes tokens from graph edge i. For the above graph, we thus can create the following topology

Fig. 2.10 Example SDF
graph for PASS construction
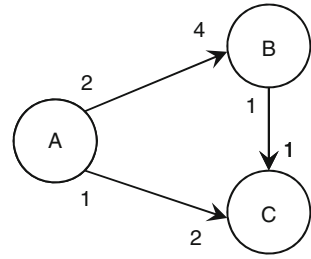
matter. Note that G does not have to be square – it depends on the amount of queues
and actors in the system.

$$G = \begin{bmatrix} 2 & -4 & 0 \\ 1 & 0 & -2 \\ 0 & 1 & -1 \end{bmatrix} \begin{array}{l} \leftarrow \text{edge}(A, B) \\ \leftarrow \text{edge}(A, C) \\ \leftarrow \text{edge}(B, C) \end{array} \qquad (2.1)$$

Step 2. The condition for a PASS to exist is that the rank of G has to be one
less than the number of nodes in the graph. The proof of this theorem is beyond
the scope of this book but can be consulted in Lee and Messerschmitt (1987). The
rank of a matrix is the number of independent equations in G. It can be verified
that there are only two independent equations in G. For example, multiply the first
column with -2 and the second column with -1, and add those two together to find
the third column. Since there are three nodes in the graph and the rank of G is 2, a
PASS is possible.

Step 2 verifies that tokens cannot accumulate on any of the edges of the graph.
We can find the resulting number of tokens by choosing a firing vector and making
a matrix multiplication. For example, assume that A fires two times, and B and C
each fire zero times. This yields the following firing vector:

$$q = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \qquad (2.2)$$

The residual tokens left on the edges after these firings are two tokens on
edge(A, B) and a token on edge(A, C):

$$b = Gq = \begin{bmatrix} 2 & -4 & 0 \\ 1 & 0 & -2 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \qquad (2.3)$$

Step 3. Determine a periodic firing vector. The firing vector indicated above is
not a good choice to obtain a PASS: each time this firing vector executes, it adds

three tokens to the system. Instead, we are interested in firing vectors that leave no additional tokens on the queues. In other words, the result must equal the zero-vector.

$$Gq_{\text{PASS}} = 0 \tag{2.4}$$

Since the rank of G is less than the number of nodes, this system has an infinite number of solutions. Intuitively, this is what we should expect. Assume a firing vector (a, b, c) would be a solution that can yield a PASS, then also $(2a, 2b, 2c)$ will be a solution, and so is $(3a, 3b, 3c)$, and so on. You just need to find the simplest one. One possible solution that yields a PASS is to fire A twice, and B and C each once:

$$q_{\text{PASS}} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \tag{2.5}$$

The existence of a PASS firing vector does not guarantee that a PASS will also exist. For example, just by changing the direction of the (A, C) edge, you would still find the same q_{PASS}, but the resulting graph is deadlocked since all nodes are waiting for each other. Therefore, there is still a fourth step: construction of a valid PASS.

Step 4. Construct a PASS. We now try to fire each node up to the number of times specified in q_{PASS}. Each node which has the adequate number of tokens on its input queues will fire when tried. If we find that we can fire no more nodes, and the firing count of each node is less than the number specified in q_{PASS}, the resulting graph is deadlocked.

We apply this on the original graph and using the firing vector $(A = 2, B = 1, C = 1)$. First we try to fire A, which leaves two tokens on (A, B) and one on (A, C). Next, we try to fire B – which has insufficient tokens to fire. We also try to fire C but again have insufficient tokens. This completes our first round through – A has fired already one time. In the second round, we can fire A again (since it has fired less than 2 times), followed by B and C. At the end of the second round, all nodes have reached the firing count specified in the PASS firing vector, and the algorithm completes. The PASS we are looking for is (A, A, B, C).

The same algorithm, when applied to the deadlocked graph in Fig. 2.11, will immediately abort after the first iteration, because no node was able to fire.
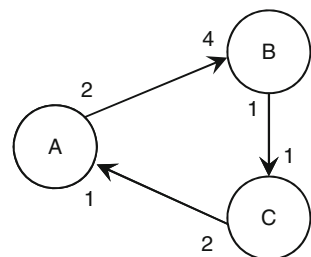
Fig. 2.11 A deadlocked graph

Note that the determinate property of SDF graphs implies that we can try to fire actors in any order of our choosing. So, instead of trying the order (A, B, C) we can also try (B, C, A). In some SDF graphs (but not in the one discussed above), this may lead to additional PASS solutions.

2.2.2 Example: Euclid's Algorithm as an SDF Graph

The graph in Fig. 2.12 evaluates the greatest common divisor of two numbers a and b. The sort actor reads two numbers, sorts them, and copies them to the output. The diff actor subtracts the smallest number from the largest one, as long as they are different. If we run this system for a while, we see that the value of the token flowing around converges to the greatest common divisor of the two numbers a and b. For example, assume

$$(a_0, b_0) = (16, 12) \tag{2.6}$$

then we see the following sequence of token values.

$$(a_1, b_1) = (4, 12), (a_2, b_2) = (8, 4), (a_3, b_3) = (4, 4), \ldots \tag{2.7}$$

a_i and b_i are the token values upon iteration i of the PASS. Since this sequence converges to the tuple $(4, 4)$, we conclude that the greatest common divisor of 12 and 16 is 4.

We will now derive a PASS for this system. The topology matrix G for this graph is shown below. The columns, left to right, correspond to each node from the SDF graph, left to right.
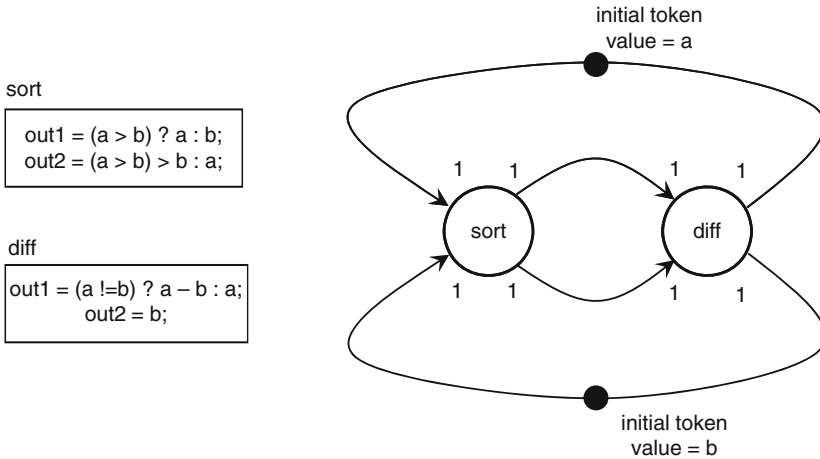
Fig. 2.12 Euclid's greatest common divisor as an SDF graph

$$G = \begin{bmatrix} +1 & -1 \\ +1 & -1 \\ -1 & +1 \\ -1 & +1 \end{bmatrix} \begin{array}{l} \leftarrow \text{edge(sort, diff)} \\ \leftarrow \text{edge(sort, diff)} \\ \leftarrow \text{edge(diff, sort)} \\ \leftarrow \text{edge(diff, sort)} \end{array} \qquad (2.8)$$

One can determine that the rank of this matrix is one, since the columns complement each other. Since there are two actors in the graph, we conclude that the condition for PASS (i.e., rank$(G) =$ nodes-1) is fulfilled. A valid firing vector for this system is one in which each actor fires exactly once per iteration.

$$q_{\text{PASS}} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad (2.9)$$

A working schedule for this firing vector would be to fire each of the actors in the graph in sequence, using the order (sort, diff). Note that in the graph as shown, there is only a single, strictly sequential schedule possible. For now, we will also ignore the stopping condition (i.e., detecting that a and b are equal and that the system can terminate).

This completes our discussion of PASS. SDF has very powerful properties, which allow a designer to predict critical system behavior upfront (such as determinism, deadlock, storage requirements). Yet SDF is not a universal specification mechanism; it is not a good replacement for any possible hardware/software system. The next part will further elaborate some of the difficulties of data flow modeling.

2.3 Control Flow Modeling and the Limitations of Data Flow Models

SDF systems are distributed, data-driven systems. They execute whenever there is data to process, and remain idle when there is nothing to do. However, SDF seems to have trouble to model control-related aspects. Control appears in many different forms in system design, for example:

- **Stopping and Restarting**. As we saw in the Euclid example (Fig. 2.12), an SDF model never terminates; it just keeps running. Stopping and restarting is a control-flow property that cannot be addressed well with SDF graphs.
- **Mode-Switching**. When a cell-phone switches from one standard to the other, the processing (which may be modeled as an SDF graph) needs to be reconfigured. However, the topology of an SDF graph is fixed and cannot be modified at runtime.
- **Exceptions**. When catastrophic events happen, processing may suddenly need to be altered. SDF cannot model exceptions that affect the entire graph topology. For example, once a token enters a queue, the only way of removing it is to read the token out of the queue. It is not possible to suddenly 'empty' the queue on a global, exceptional condition.

- **Run-Time Conditions**. A simple if-then-else statement (choice between two activities depending on an external condition) is troublesome for SDF. An SDF node cannot simply 'disappear' or become inactive – it is always there. Moreover, we cannot generate conditional tokens, as this would violate SDF rules which require fixed production/consumption rates. Thus, SDF cannot model conditional execution such as required for if-then-else statements.

There are two solutions to the problem of control flow modeling in SDF. The first one is to try using SDF anyhow, but emulate control flow at the cost of some modeling overhead. The second one is to extend the semantics of SDF. We give a short example of each strategy.

2.3.1 Emulating Control Flow with SDF Semantics

Figure 2.13 shows an example of an if-then-else statement, SDF-style. Each of the actors in the above graph is an SDF actors. The last one is a selector-actor, which will transmit either the A or B input to the output depending on the value of the input condition. Note that when Sel fires, it will consume a token from each input, so both A and B have to run for each input token. So this is not really an if-then-else in the same sense as in C programming. The approach taken by this graph is to implement both the if-leg and the else-leg and afterwards transmit only the required result. This approach is sometimes taken when there is sufficient parallelism in the underlying implementation. For example, in hardware design, the equivalent of the Sel node would be a multiplexer (Fig. 2.14).

2.3.2 Extending SDF Semantics

Researchers have also proposed extensions on SDF models. One of these models is called BDF, or Boolean Data Flow Buck (1993). The idea of BDF is to make the production and consumption rate of a token dependent on the value of an external
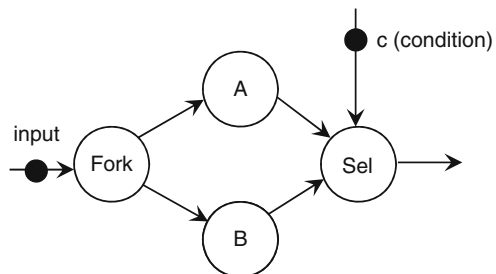
Fig. 2.13 Emulating if-then-else conditions in SDF

Fig. 2.14 Implementing if-then-else using boolean data flow
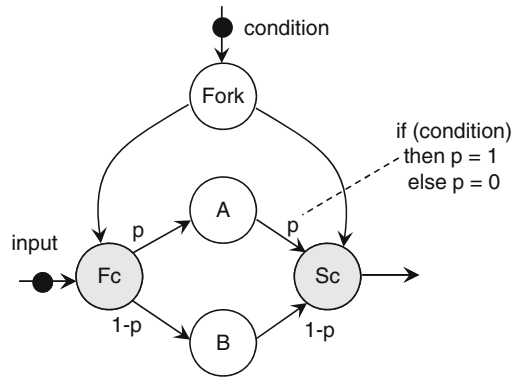
control token. In the above graph, the condition token is distributed over a fork to a conditional fork and a conditional merge node. These conditional nodes are BDF.

- The conditional fork will fire when there is an input token and a condition token. Depending on the value of the condition token, it will produce an output on the upper or the lower queue. We used a conditional production rate p to indicate this. It is impossible to determine the value of p upfront – this can only be done at runtime.
- The conditional merge will fire when there is a condition token. If there is, it will accept a token from the upper input or the lower input, depending on the value of the condition. Again, we need to introduce a conditional consumption rate.

The overall effect is that either node A or else node B will fire, but never both. Even a simple extension on SDF already takes jeopardizes the basic properties which we have enumerated above. For example, a consequence of using BDF instead of SDF is that we now have data flow graphs that are only conditionally admissible. Moreover, the topology matrix now will include symbolic values (p), and becomes harder to analyze. For 5 conditions, we would have to either a matrix with 5 symbols, or else enumerate all possible condition values and analyze 32 different matrices (each of which can have a different series of markings). In other words, while BDF can help solving some of practical cases of control, it quickly becomes impractical for analysis.

Besides BDF, researchers have also proposed other flavors of control-oriented data flow models, such as Dynamic Data Flow (DDF) which allows variable production and consumption rates, and Cyclo-Static Data Flow (CSDF) which allows a fixed, iterative variation on production and consumption rates. All of these extensions break down the elegance of SDF graphs to some extent. SDF remains a very popular technique for Digital Signal Processing applications. But the use of BDF, DDF, and the like has been limited.

2.4   Software Implementation of Data Flow

In this section, we will consider the implementation of SDF graphs in software.

2.4.1   Converting Queues and Actors into Software

Figure 2.15 demonstrates several different approaches to map dataflow into software. A first distinction will be made between mapping dataflow to multiprocessor systems and to single-processor systems. Our first concern is the implementation of dataflow graphs on single-processor systems. Such implementations require a sequential scheduling of dataflow actors. There are two methods to implement such a sequential schedule.

- We can do this using a *dynamic schedule*, which means that the software processor will evaluate, during execution of the SDF graph, the order in which actors should execute. This can be implemented in several ways. Essentially, it implies that the CPU will test the actors' firing rule at runtime to evaluate which actor can run. Dynamic scheduling of an SDF system can be done using a single-thread executive, or else using multithreading.
- We can also use a *static schedule*, which means that we will determine upfront exactly in what order the actors need to run (fire). This can be implemented using a single-threaded executive. However, because the static schedule fixes the execution order of the actors, there is an additional important optimization opportunity: we can inline the entire dataflow graph in a single function.
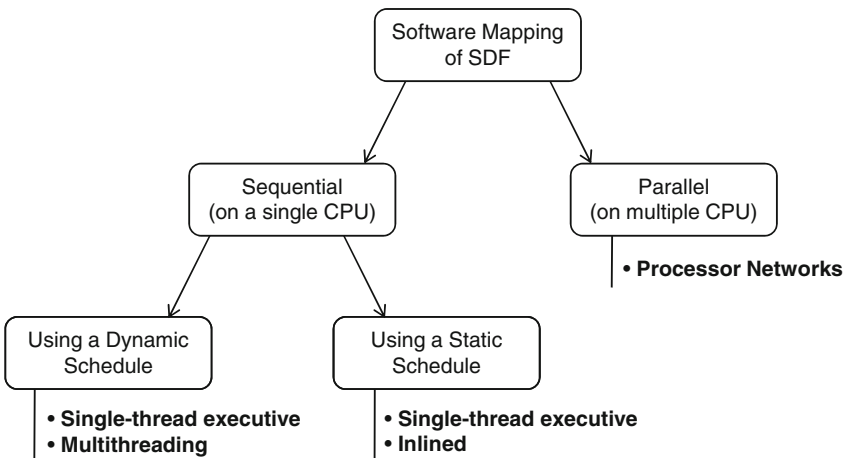
Fig. 2.15   Overview of possible approaches to map dataflow into software

Before studying the various means to implement SDF schedules, let us first recall the essential features of SDF graphs. SDF graphs represent concurrent systems and use actors which communicate over FIFO queues. The firing only depends on the availability of data (tokens) in the FIFO queues, and it can be described with the firing rule for that actor. The amount of tokens produced/consumed per firing at the output/input of an actor is specified by production rate/consumption rate for that output/input. When implementing an SDF graph in software, we have to map *all* elements of the SDF graph in software: actors, queues, and firing rules. Eventually, the implementation will always need to follow the rules of dataflow.

2.4.1.1 FIFO Queues

An SDF system requires, in principle, infinitely large FIFO queues. In practice however, you will not implement infinite FIFO queues, but instead create a queue with a limited number of positions and overflow detection. Another approach is to create a FIFO that grows dynamically in length – for example, by doubling the amount of memory allocated for it, each time the FIFO overflows. Note also that if we know a PASS, we can create a static schedule and determine the maximum number of tokens on each queue, and then appropriately choose the size of each queue. Here is a software object Q (in C) which could model such a FIFO of limited length (Fig. 2.16).

The typical software interface of a FIFO queue has two parameters and three methods.

- The number of elements N that can be stored by the queue. (parameter)
- The data type element of a queue elements. (parameter)
- A method to put elements into the queue.
- A method to get elements from the queue.
- A method to test the number of elements in the queue.

The storage organization can be done with a standard data structure such as a circular queue. A circular queue is a data structure consisting of an array of memory
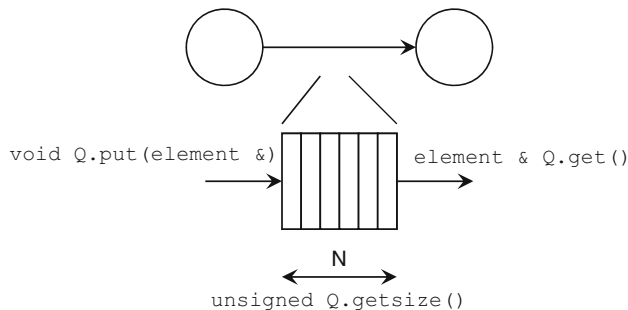
Fig. 2.16 A software queue

locations, a write-pointer, and a read-pointer. These pointers map relative queue addresses to array addresses using modulo addressing as illustrated in the figure below. The head of the queue is at Rptr. Element I of the queue is at (Rptr+I) mod array_size. The tail of the queue is at (Wptr−1) mod array_size. Figure 2.17 illustrates the operation of a 2-element circular queue.

2.4.1.2 Actors

A data flow actor can be captured as a function, with some additional support to interface with the FIFO queues (Fig. 2.18). Designers will often differentiate between the internal activities of an actor and the input–output behavior. The behavior corresponding to actor firing can be implemented as a simple C function. The logic
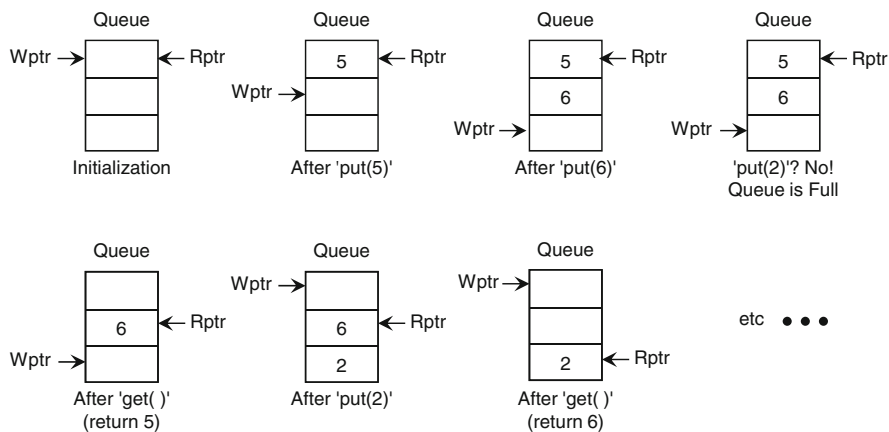
Fig. 2.17 Operation of the circular queue
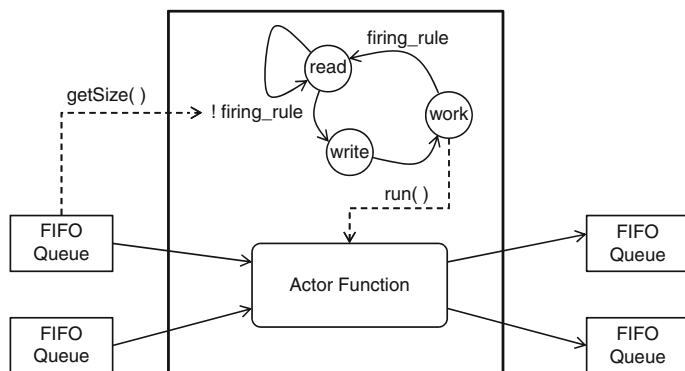
Fig. 2.18 Software implementation of the dataflow actor

around this function tests the firing rule and manipulates the input queues and the output queues. One can think of this logic as a small controller on top of the actor function.

The local controller of an actor goes through three states. In the read state, it remains idle until a token arrives at an input queue. The actor then proceeds to the work state. The controller reads one token from the input queue, extracts the value of that token, and runs the function. This yields the value of the output token of the actor. Finally, the output value can be entered into the output queue, which is done while the actor controller transitions through the write state.

However we must take care that the firing rule is implemented correctly. When an SDF actor fires, it has to read all input queues and it has to write into all output queues according to the specified production and consumption rates.

We can now implement the actor in C. As mentioned before, we will implement all the actors as C functions. The following struct collects all the inputs and outputs of an actor.

```
typedef struct actorio {
  fifo_t *in1;
  fifo_t *in2;
  fifo_t *out1;
  fifo_t *out2;
} actorio_t;
```

We can use that struct to model actors as functions, for example:

```
void sort_actor(actorio_t *g) {
  int r1, r2;
  if ((fifo_size(g->in1) > 0) &&
      (fifo_size(g->in2) > 0)) {
    r1 = get_fifo(g->in1);
    r2 = get_fifo(g->in2);
    put_fifo(g->out1, (r1 > r2) ? r1 : r2);
    put_fifo(g->out2, (r1 > r2) ? r2 : r1);
  }
}
```

Finally, the actor_io and queue objects can be instantiated in the main program, and the actor functions can be called using a system scheduler.

2.4.2 Sequential Targets with Dynamic Schedule

A software implementation of SDF is obtained by combining several different actor descriptions, by interconnecting those actors using FIFO queues, and by executing the actors through a system schedule. In a *dynamic* system schedule, the firing rules of the actors will be tested at runtime; the system scheduling code consists of the firing rules, as well as the order in which the firing rules are tested.

Listing 2.2 FIFO object in C
```c
#define MAXFIFO 1024

typedef struct fifo {
  int data[MAXFIFO]; // array
  unsigned wptr;      // write pointer
  unsigned rptr;      // read pointer
} fifo_t;

void init_fifo(fifo_t *F) {
  F->wptr = F->rptr = 0;
}

void put_fifo(fifo_t *F, int d) {
  if (((F->wptr + 1) % MAXFIFO) != F->rptr) {
    F->data[F->wptr] = d;
    F->wptr = (F->wptr + 1) % MAXFIFO;
    assert(fifo_size(F) <= 10);
  }
}

int get_fifo(fifo_t *F) {
  int r;
  if (F->rptr != F->wptr) {
    r = F->data[F->rptr];
    F->rptr = (F->rptr + 1) % MAXFIFO;
    return r;
  }
  return -1;
}

unsigned fifo_size(fifo_t *F) {
  if (F->wptr >= F->rptr)
    return F->wptr - F->rptr;
  else
    return MAXFIFO - (F->rptr - F->wptr) + 1;
}

int main() {
  fifo_t F1;
  init_fifo(&F1); // resets wptr, rptr;
  put_fifo(&F1, 5); // enter 5
  put_fifo(&F1, 6); // enter 6
  printf("%d_%d\n", fifo_size(&F1), get_fifo(&F1));// prints: 2 5
  printf("%d\n", fifo_size(&F1)); // prints: 1
}
```

2.4.2.1 Single-Thread Dynamic Schedules

Following the FIFO and actor modeling in C, as discussed in Sect. 2.4.1, we can implement a system schedule as a function that instantiates all actors and queues, and next calls the actors in a round-robing fashion.

```
void main() {
  fifo_t F1, F2, F3, F4;
  actorio_t sort_io;
  ..
  sort_io.in1 = &F1;
  sort_io.in2 = &F2;
  sort_io.out1 = &F3;
  sort_io.out2 = &F4;
  while (1) {
    sort_actor(&sort_io);
    // .. call other actors
  }
}
```

The interesting question, of course, is: what is the most appropriate call order of the actors in the system schedule? First, note that it is impossible to call the actors in the 'wrong' order, because each of them still has a firing rule that protects them from running when there is no data available. Consider the example in Fig. 2.19. Even though snk will be often called as often as src, the firing rule of snk will only allow that actor to run when there is sufficient data available. In Fig. 2.19a, this means that the snk actor will only fire every other time the main function calls it. However, while this technique of dynamic scheduling will prevent actors from running prematurely, it is still possible that some actors run too often, resulting in the number of tokens on the interconnection queues slowly growing. This happens, for example, in Fig. 2.19b. In this case, the src actor will produce two tokens each time the main function calls it, but the snk actor will only read one of these tokens per firing.

The problem of the system schedule in Fig. 2.19b is the firing rate provided by the system schedule differs from the required firing rate for a PASS. Indeed, the PASS for this system would be (src, snk, snk). The dynamic system schedule,
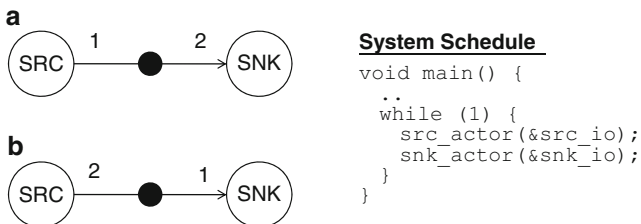
Fig. 2.19 (**a**) A graph which will simulate fine under a single rate system schedule, (**b**) a graph which will cause extra tokens under a single rate system schedule

given by the code below, cannot make the firing rate of SNK higher than that of SRC. This problem can be addressed in several ways.

- **Solution 1:** We could adjust the system schedule to reflect the firing rate predicted by the PASS. Thus, the code for the system scheduler becomes:

```
void main() {
  ..
  while (1) {
    src_actor(&src_io);
    snk_actor(&snk_io);
    snk_actor(&snk_io);
  }
}
```

This solution is not very elegant, because it destroys the idea of having a dynamic scheduler. If we have to obtain the PASS firing rate first, we may as well forget about using a dynamic schedule.

- **Solution 2:** We could adjust the code for the `snk` actor to continue execution as long as there are tokens present. Thus, the code for the `snk` actor becomes:

```
void snk_actor(actorio_t *g) {
  int r1, r2;
  while ((fifo_size(g->in1) > 0)) {
    r1 = get_fifo(g->in1);
    ... // do processing
  }
}
```

This is a better solution as the previous one, because it keeps the advantages of a dynamic system schedule.

2.4.2.2   MultiThread Dynamic Schedules

The actor functions, as described above, are captured as real functions. They exit completely in between invocations. As a result, actors cannot maintain state in local variables, but have to use global variables instead. We will discuss a solution based on multithreaded programming, in which each actor lives in a separate thread.

A multithreaded C program is a program that has two concurrent threads of execution. For example, in a program with two functions, one thread could be executing the first function, while the other thread could be executing the second function. Since there is only a single processor to execute this program, we need to switch the processor back and forth between the two threads of control. This is done by a scheduler – similar to a scheduler used for scheduling actors, a thread scheduler will switch between threads.

We will illustrate the use of *cooperative* multithreading. In this model, the threads of control indicate at which point they release control back to the scheduler. The scheduler then decides which thread can run next.

Figure 2.20 shows an example with two threads. Initially, the user has provided the starting point of each thread using `create()`. Assume that the upper thread
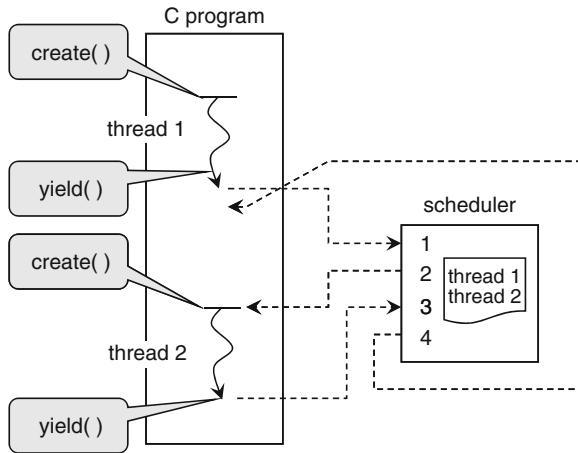
Fig. 2.20 Example of cooperative multithreading

(thread1) is running and arrives at a yield() point. This is a point where the thread returns control to the scheduler. The scheduler maintains a list of threads under its control, and therefore knows that the lower thread (thread2) is ready to run. So it allows thread2 to run until that thread, too, comes at a yield point. Now the scheduler sees that each thread had a chance to run, so it goes back the first thread. The first thread then will continue just after the yield point.

We thus see that two functions are enough to build a threading system: create() and yield(). The scheduler can apply different strategies to select each thread, but the simplest one is to let each thread run in turn – this is called a 'round-robin' scheduling strategy. We will look at the functions in a cooperative multithreading library called quickthreads. The quickthreads API (Application Programmers' Interface) consists of four function calls.

- stp_init() initializes the theading system.
- stp_create(stp_userf_t *F, void *G) creates a thread that will start execution with user function F. The function will be called with a single argument G. The thread will terminate when that function completes, or when the thread aborts.
- stp_yield() releases control over the thread to the scheduler.
- stp_abort() terminates a thread so that it will be no more scheduled.

Listing 2.3 is a small program that uses the QuickThread library.

This program creates two threads (line 21–22), one which starts at function hello, and another which starts at function world. Function hello (line 3–9) is a loop that will print "hello" three times, and yield after each iteration. After the third time, the function will return, which also terminates the thread. Function world (line 11–17) is a loop that will print "world" five times, and yield at the end of each iteration. When all threads are finished, the main function will terminate. We compile and run the program as follows.

Listing 2.3 Example of QuickThreads

```c
#include "../qt/stp.h"
#include <stdio.h>

void hello(void *null) {
  int n = 3;
  while (n-- > 0) {
    printf("hello\n");
    stp_yield();
  }
}

void world(void *null) {
  int n = 5;
  while (n-- > 0) {
    printf("world\n");
    stp_yield();
  }
}

int main(int argc, char **argv) {
  stp_init();
  stp_create(hello, 0);
  stp_create(world, 0);
  stp_start();
  return 0;
}
```

```
>gcc -c ex1.c -o ex1../qt/libstp.a ../qt/libqt.a
./ex1
hello
world
hello
world
hello
world
world
world
```

Indeed the printing of hello and world are interleaved for the first three iterations, and then the world thread runs through completion.

We can now use this multithreading system to create a multithread version of the SDF scheduler. Here is the example of a sort actor, implemented using the cooperative threading model.

```c
void sort_actor(actorio_t *g) {
  int r1, r2;
  while (1) {
    if ((fifo_size(g->in1) > 0) &&
        (fifo_size(g->in2) > 0)) {
      r1 = get_fifo(g->in1);
      r2 = get_fifo(g->in2);
      put_fifo(g->out1, (r1 > r2) ? r1 : r2);
```

```
      put_fifo(g->out2, (r1 > r2) ? r2 : r1);
    }
    stp_yield();
  }
}
```

The system scheduler now will call threads rather than actors:

```
void main() {
  fifo_t F1, F2, F3, F4;
  actorio_t sort_io;
  ..
  sort_io.in1 = &F1;
  sort_io.in2 = &F2;
  sort_io.out1 = &F3;
  sort_io.out2 = &F4; // connect queues
  stp_create(sort_actor, &sort_io); // create thread
  stp_start(); // start system schedule
}
```

Similar to what is discussed before, the execution rate of the actor code must be equal to the PASS firing rate in order to avoid unbounded growth of tokens in the system. A typical cooperative multithreading system uses round-robin scheduling: all actor threads in the system need to run() after one actor thread calls yield(). Therefore, Solution 1 (as discussed before under the single-thread executive method) cannot work. Instead, we need Solution 2, and allow an actor thread to fire several times before it yields:

```
void sort_actor(actorio_t *g) {
  int r1, r2;
  while (1) {
    while ((fifo_size(g->in1) > 0) &&
           (fifo_size(g->in2) > 0)) {
      r1 = get_fifo(g->in1);
      r2 = get_fifo(g->in2);
      put_fifo(g->out1, (r1 > r2) ? r1 : r2);
      put_fifo(g->out2, (r1 > r2) ? r2 : r1);
    }
  stp_yield();
  }
}
```

2.4.3 Sequential Targets with Static Schedule

When we have completed the PASS analysis for an SDF graph, we know at least one solution for a feasible sequential schedule. We can use this to optimize the implementation in several ways:

- First, since we know the exact sequential schedule, we are able to remove the firing rules of the actors. This will yield a small performance advantage. Of course, we can no longer use such actors with dynamic schedulers.

- Next, we can also investigate the optimal interleaving of the actors such that the storage requirements for the queues are reduced. This is illustrated below.
- Finally, we can create a fully inlined version of the SDF graph, by exploiting our knowledge on the static, periodic behavior of the system as much as possible. We will see that this not only allows us to get rid of the queues but also allows us to create a fully inlined version of the entire SDF system.

Consider the example in Fig. 2.18. From this SDF topology, we know that the relative firing rates of A, B, and C must be 4, 2, and 1 to yield a PASS. The right side of the code shows an example implementation of this PASS. The A, B, C actors are called in accordance with their PASS rate. Due to this particular interleaving, it is easy to see that in a steady state condition, the queue AB will carry four tokens maximum, while the queue BC will contain two tokens maximum. This is not the most optimal interleaving. By calling the actors in the sequence (A,A,B,A,A,B,C), the maximum amount of tokens on any queue is reduced to two. Finding an optimal interleaving in an SDF graph is an optimization problem. While an in-depth discussion of this optimization problem is beyond the scope of this book, it is important to keep in mind that the solution determined using PASS is not necessarily optimal (Fig. 2.21).

Implementing a truly *static* schedule means that we will no longer test firing rules when calling actors. In fact, when we call an actor, we will have to guarantee that the required input tokens are available. In a system with a static schedule, all SDF-related operations get a fixed execution order: the actor firings and the sequences of `put` and `get` operations on the FIFO queues. This provides the opportunity to optimize the resulting SDF system.

We will discuss optimization of single-thread SDF systems with a static schedule using an example we discussed before – the GCD. From our earlier analysis, we know that a valid PASS fires each node a single time. Listing 2.4 is a description for each of the actors (`sort`, `diff`).

These actors are interconnected in the main program by means of queues. The main program also executes the PASS in the form of a while loop (Listing 2.5).


```
while(1) {
  // call A four times
  A(); A(); A(); A();

  // call B two times
  B(); B();

  // call C one time
  C();
}
```
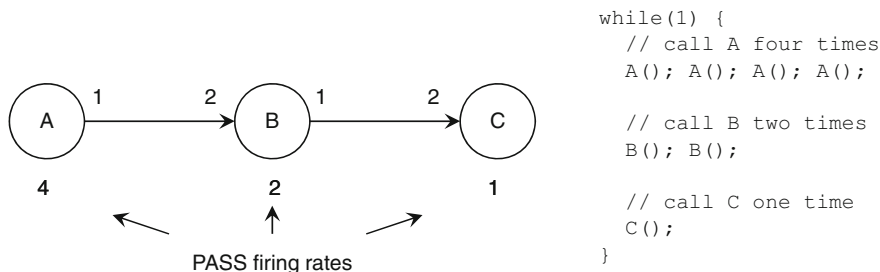
Fig. 2.21 System schedule for a multirate SDF graph

Listing 2.4 Actors for Euclid's Algorithm

```
void sort_actor(actorio_t *g) {
  int r1, r2;
  if ((fifo_size(g->in1) > 0) &&
  (fifo_size(g->in2) > 0)) {
    r1 = get_fifo(g->in1);
    r2 = get_fifo(g->in2);
    put_fifo(g->out1, (r1 > r2) ? r1 : r2);
    put_fifo(g->out2, (r1 > r2) ? r2 : r1);
  }
}

void diff_actor(actorio_t *g) {
  int r1, r2;
  if ((fifo_size(g->in1) > 0) &&
  (fifo_size(g->in2) > 0)) {
    r1 = get_fifo(g->in1);
    r2 = get_fifo(g->in2);
    put_fifo(g->out1, (r1 != r2) ? r1 - r2 : r1);
    put_fifo(g->out2, r2);
  }
}
```

Listing 2.5 System Schedule for Euclid's Algorithm

```
void main() {
  fifo_t F1, F2, F3, F4;
  actorio_t sort_io, diff_io;
  sort_io.in1  = &F1;
  sort_io.in2  = &F2;
  sort_io.out1 = &F3;
  sort_io.out2 = &F4;
  diff_io.in1  = &F3;
  diff_io.in2  = &F4;
  diff_io.out1 = &F1;
  diff_io.out2 = &F2;

  // initial tokens
  put_fifo(&F1, 16);
  put_fifo(&F1, 12);

  // system schedule
  while (1) {
    sort_actor(&sort_io);
    diff_actor(&diff_io);
  }
}
```

The optimizations are as follows:

1. Because the firing order of actors can be completely fixed, the access order on queues can be completely fixed as well. This latter fact will allow the queues themselves to be optimized out and replaced with fixed variables. Indeed, assume

Listing 2.6 Optimized System Schedule for Euclid's Algorithm

```
void main() {
  int f1, f2, f3, f4;

  // initial token
  f1 = 16;
  f2 = 12;

  // system schedule
  while (1) {

    // code for actor 1
    f3 = (f1 > f2) ? f1 : f2;
    f4 = (f1 > f2) ? f2 : f2;

    // code for actor 2
    f1 = (f3 != f4) ? f3 - f4;
    f2 = f4;
  }
}
```

for example that we have determined that the access sequence on a particular FIFO queue will always be as follows:

```
loop {
  ...
  F1.put(value1);
  F1.put(value2);
  ...
  .. = F1.get();
  .. = F1.get();
}
```

In this case, only two positions of FIFO F1 are occupied at a time. Hence, FIFO F1 can be replaced by two single variables.

```
loop {
  ...
  r1 = value1;
  r2 = value2;
  ...
  .. = r1;
  .. = r2;
}
```

2. As a second optimization, we can inline actor code inside of the main program and the main scheduling loop. In combination with the above optimization, this will allow to drop the firing rules and to collapse an entire dataflow graph in a single function.

When we apply these optimizations to the Euclid example, each queue (F1, F2, F3, and F4) will contain no more than a single token, which means that each queue can be replaced by a single integer (Listing 2.6).

In the above example, we can expect the runtime to decrease significantly. We have dropped testing of the firing rules, FIFO manipulations, and function boundaries. This is possible because we have determined a valid PASS for the initial data flow system, and we have chosen a fixed schedule to implement that PASS.

2.5 Hardware Implementation of Data Flow

We can implement SDF actors also as dedicated hardware engines. While we could map FIFO queues and actor firing rules directly in hardware, we are especially interested in simple, optimized implementations. The use of hardware FIFOs (that require handshake synchronization protocols) will be covered later when we discuss advanced hardware/software interfaces.

2.5.1 Single-Rate SDF Graphs

As an optimized implementation, consider simplest case, in which there is a direct, one-to-one mapping from SDF graphs to hardware elements. Each actor then translates to a single combinational hardware module, and each FIFO queue translates to wires or registers. Figure 2.22 illustrates how this works out for the Euclid example.

We create the following implementation:

1. Map each queue to a wire.
2. Map each queue containing a token to a register. The initial value of the register must equal the initial value of the token.
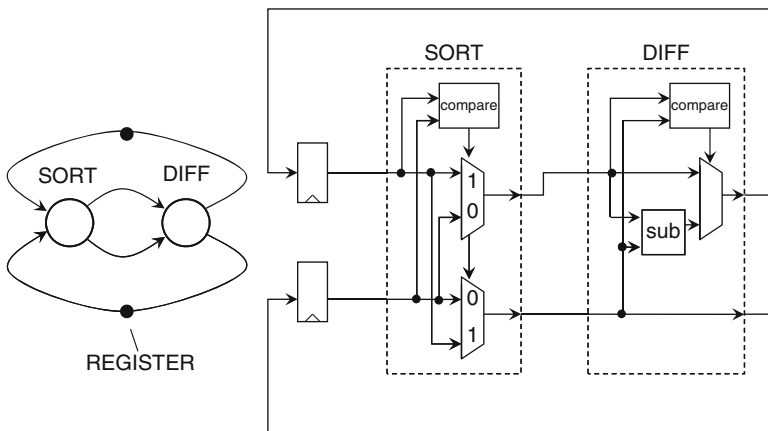
Fig. 2.22 Hardware implementation of Euclid's algorithm

Fig. 2.23 SDF graph
of a simple moving-average
application
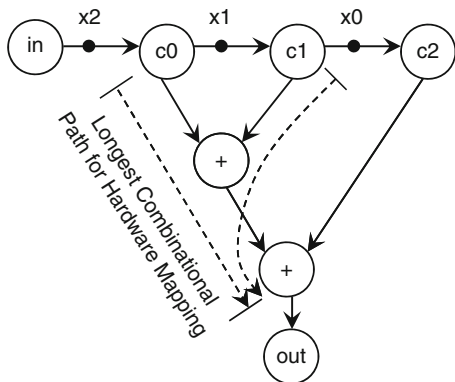
3. Map each actor to a combinational circuit, which completes a firing within a
   clock cycle. Both the sort and diff actors require no more than a comparator
   module, a few multiplexers, and a subtractor.

Does this circuit work? Yes, it does. The circuit evaluates a single iteration
through a PASS per clock cycle.

Is this translation procedure general so that it would work for any SDF graph?
No, it is not. The translation procedure is restricted to the following SDF graphs.

- We need a single-rate SDF graph, which has a PASS firing vector with all ones
  in it.
- All actors need to be implemented using combinational logic.

In addition, the above method may result in circuits with a very long combina-
tional path. In the circuit above for example, the maximal operating clock frequency
is determined by the combined delay of the sort circuit and the diff circuit. Still, the
concept of this transformation is useful, in particular when it is used with the trans-
formations which will be discussed next (Fig. 2.23).

2.5.2 Pipelining

Pipelining of SDF graphs helps to break long combinational paths that may exist in
circuits. Consider the example shown below.

This is a data flow specification of a digital filter. It evaluates a weighted sum of
samples of an input stream, with the sum defined as $out = x0.c2 + x1.c1 + x2.x0$. It
can be seen from this graph that the critical path is equal to a constant multiplication
(with $c0$ or $c1$) and two additions. We would like to 'push down' initial tokens into
the adder tree. With the rules of data flow execution, this is easy. Consider a few
subsequent markings of the graph. We let the in actor produce additional tokens, and
then let the $c0$, $c1$, $c2$ and add actors fire so that additional tokens start to appear on

Fig. 2.24 Pipelining the moving-average filter by inserting additional tokens (1)
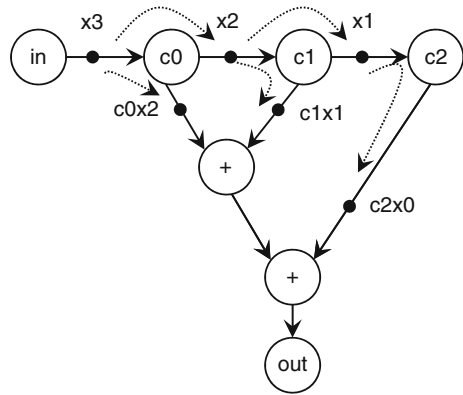
Fig. 2.25 Pipelining the moving-average filter by inserting additional tokens (2)
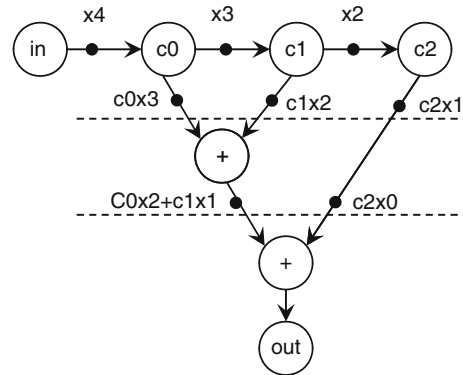
queues that have no such tokens. For example, assume that the in actor produces a single additional token $x3$. Then the resulting graph looks as follows (Fig. 2.24):

In this graph, the longest combinational path is reduced to only two additions. By letting the in actor produce another token, we will be able to reduce the longest combinational path to a single addition (Fig. 2.25).

The resulting SDF graph can be implemented as in Fig. 2.26.

Remember that it is not possible to introduce arbitrary initial tokens in a graph without following the rules for actor firing – doing so will almost certainly change the behavior of the system. This change in behavior is obvious in the case of feedback loops, such as shown in the accumulator circuit in Fig. 2.27. Using a single token in the feedback loop of an add actor will accumulate all input samples. Using two tokens in the feedback loop will accumulate the odd samples and even samples separately. When pipelining a SDF graph, make sure to follow the normal steps for data flow marking (i.e., do not introduce any initial token unless it can be obtained by a sequence of firings from actors).

Fig. 2.26 Hardware implementation of the moving-average filter
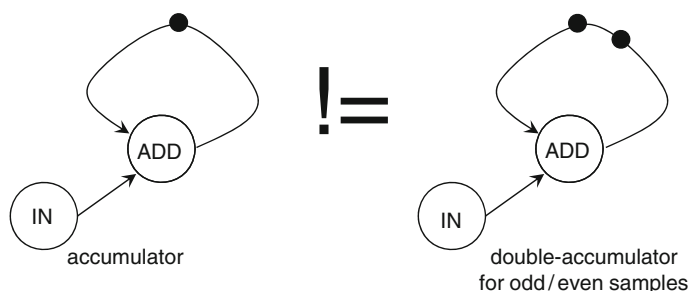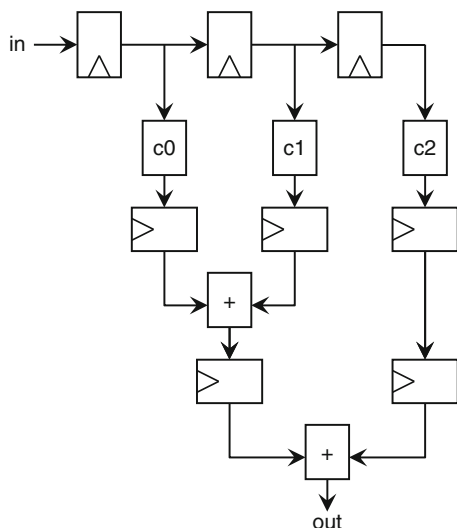
Fig. 2.27 Loops in SDF graphs cannot be pipelined

2.5.3 Multirate Expansion

Another interesting transformation concerns multirate data flow graphs. It is possible to transform such graphs systematically to single-rate SDF graphs. These single-rate SDF graphs can then be directly mapped into hardware circuits. We go through the following steps to convert a multirate graph to a single-rate graph.

1. Determine the PASS firing rates of each actor.
2. Duplicate each actor the number of times indicated by its firing rate. For example, given an actor A with a firing rate of 2, we create A0 and A1. These actors are identical.
3. Convert each multirate actor input/output to multiple single-rate input/outputs. For example, if an actor input has a consumption rate of 3, we replace it with three single-rate inputs.

4. Reintroduce the queues in the data flow system to connect all actors. Since we are building a PASS system, the total number of actor inputs will be equal to the total number of actor outputs.
5. Reintroduce the initial tokens in the system, distributing them sequentially over the single-rate queues.

Consider the following example of a multirate SDF graph. Actor A produces three tokens per firing, actor B consumes two tokens per firing. Their resulting firing rates are two and three, respectively (Fig. 2.28).

After completing steps 1–5 discussed above, we obtain the following SDF graph. The actors have duplicated according to their firing rates, and all multirate I/O were converted to single-rate I/O. The initial tokens are distributed over the queues connecting A and B. The distribution of tokens follows the sequence of queues between A and B (i.e., follows the order a, b, etc..) (Fig. 2.29).

The resulting single-rate graph can now be mapped directly into a hardware circuit. This circuit will have two inputs (IN0, IN1) and three outputs (OUT0, OUT1, OUT2). This corresponds to the original specification: the inputs are being consumed at rate 2, and the outputs are being produced at rate three.
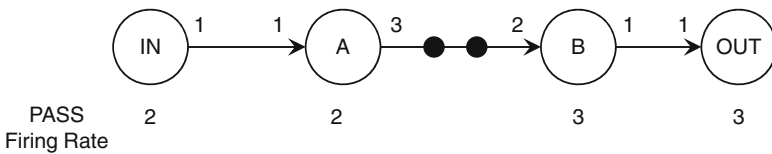
Fig. 2.28 Multi-rate data flow-graph
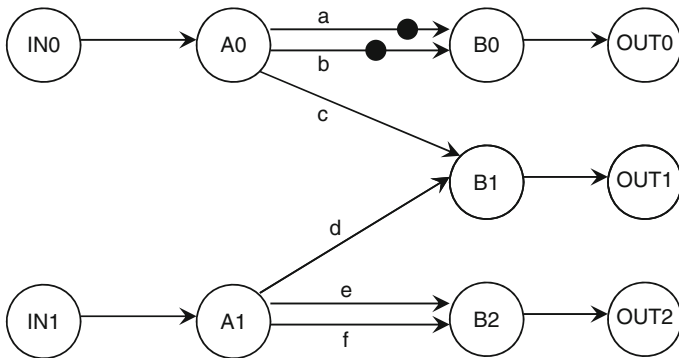
Fig. 2.29 Multi-rate SDF graph expanded to single-rate

2.6   Summary

Data Flow models express concurrent systems in such a way that the models can map into hardware as well as into software. Data Flow models consist of actors which communicate by means of tokens which flow over queues from one actor to the other. A data flow model can precisely and formally express the activities of a concurrent system. An interesting class of data flow systems are SDF models. In such models, all actors can produce or consume a fixed amount of tokens per iteration (or invocation).

By concerting a given SDF graph to a topology matrix, it is possible to derive stability properties of a data flow system (deadlock and limited number of tokens) automatically. A stable data flow system can be executed using a periodic admissible sequential schedule (PASS), a fixed period sequence of actor firings.

We also discussed how a data flow model can be automatically converted into sequential software or parallel hardware. For a sequential software implementation, we can use either threads or else static scheduling of C functions to capture the concurrent behavior of a data flow system. For hardware, a simple one-to-one conversion technique exists to translate single-rate SDF graphs into hardware. Several optimization techniques, such as pipelining and multirate expansion, can deal with graphs which are difficult or impossible to map as single-rate SDF graphs.

Data Flow modeling remains an important and easy-to-understand technique. Data Flow models are useful in signal-processing applications, in which the infinite streams of signal samples are captured as token streams and the signal processing functions as actors.

2.7   Further Reading

Dataflow analysis and implementation have been well researched over the past few decades, and Dataflow enjoys a rich body of literature.

In the early seventies, dataflow has been considered as a replacement for traditional instruction-fetch machines. Actual data-flow computers were built that operate very much according to the SDF principles discussed here. Those early years of dataflow have been documented very well at a retrospective conference called *Dataflow to Synthesis Retrospective*. The conference honored Arvind, one of dataflows' pioneers, and the online proceedings include a talk by Jack Dennis Dennis (2007).

In the eighties, dataflow garnered attention because of its ability to describe signal processing problems well. For example, Lee and Messerschmitt described SDF scheduling mechanisms Lee and Messerschmitt (1987). Parhi and Messerschmitt discussed unfolding transformations of SDF graphs Parhi and Messerschmitt (1989). Interestingly, and perhaps not unexpected, digital signal processors became a commodity around the same time. This work eventually gave rise to the Ptolemy environment Eker et al. (2003). Despite these successes, dataflow never became truly dominant compared to existing control-oriented paradigms.

However, dataflow excels in the description of streaming processing, and therefore remains very popular for signal processing applications. In particular, the recent trend toward multiprocessors has spurred a new interest in streaming applications. System specification is done in a dataflow-variant or language, and an automatic design environment maps this to a multiprocessor target. Some of the recent work in this area includes StreamIt (which maps to an IBM Cell Processor) Thies (2008) and Brook (which maps to a Graphics Processor) Stanford Graphics Lab (2003).

2.8 Problems

2.1. Consider the single-rate SDF graph in Fig. 2.30. The graph contains three types of actors. The fork actor reads one token and produces two copies of the input token, one on each output. The add actor adds up two tokens, producing a single token that holds the sum of the input token. The snk actor is a token-sink which records the sequence of tokens appearing at its input. A single initial token, with value 1, is placed in this graph. Find the value of tokens that is produced into the snk actor. Find a short-hand notation for this sequence of numbers.

2.2. The Fibonacci Number series F is defined by $F(0) = 0$, $F(1) = 1$, $F(i) = F(i-1) + F(i-2)$ when i is greater than 1. By changing the marking of the SDF graph in Fig. 2.27, it is possible to generate the Fibonacci series into the snk actor. Find the location and the initial value of the actors you will add.

2.3. Consider the SDF graph in Fig. 2.31. Transform that graph such that it will produce the same sequence of tokens twice as fast. To implement this, replace the snk actor with snk2, an actor which requires two tokens on two different inputs in order to fire. Next make additional transformations to the graph and its marking so that it will produce this double-rate sequence into snk2.

2.4. Data Flow actors cannot contain state variables. Yet, we can 'simulate' state variables with tokens. Using only an adder actor, show how you can implement an accumulator that will obtain the sum of an infinite series of input tokens.

2.5. For the SDF graph of Fig. 2.32, find a condition between x and y for a PASS to exist.
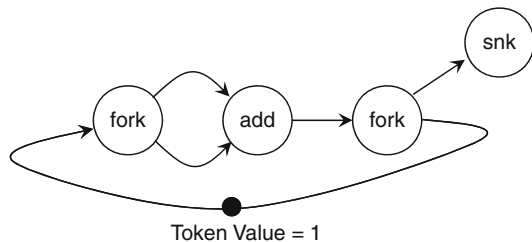
Fig. 2.30 SDF graph for Problem 2.1

Token Value = 1
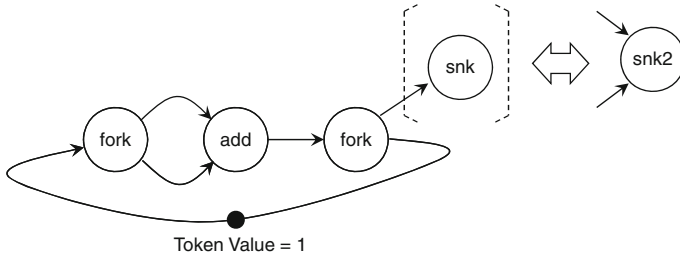
Fig. 2.31 SDF graph for Problem 2.3

Fig. 2.32 SDF graph
for Problem 2.5
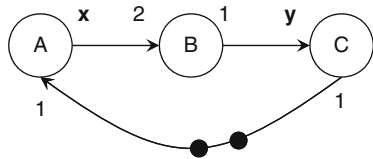
Fig. 2.33 SDF graph
for Problem 2.6
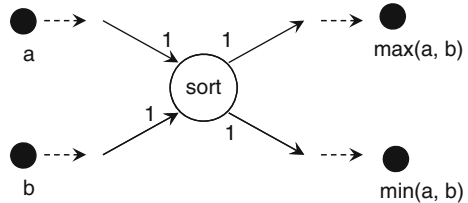
2.6. Given the 2-input sorting actor shown in Fig. 2.33. Using this actor, create a SDF graph of a sorting network with 4 inputs and 4 outputs.

2.7. Using an accumulator actor, as derived in Problem 2.4, implement the following C program as a SDF graph. The graph reads a single input token in and produces a single output token out, corresponding to the return value of the function.

```
int graph(int in) {
  int i, j, k = 0;
  for (i=0; i<10; i++)
    for (j=0; j<10; j++)
      k = k + j * (i + in);
  return k;
}
```

2.8. Assume a C function with only expressions on scalar variables (no pointers) and for-loops. Show that such a C function can be translated to a SDF graph if and only if the loop-bound expressions are manifest, that is, they only depend on compile-time constant values and loop counters.
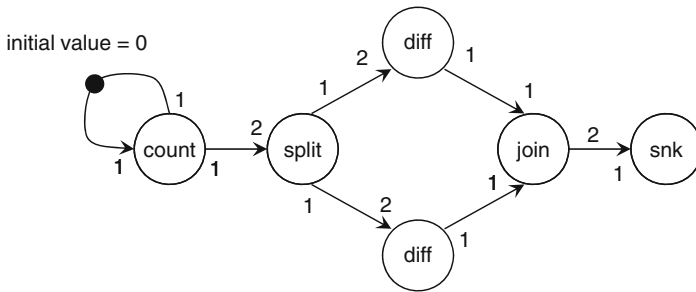
Fig. 2.34 SDF graph for Problem 2.9

2.9. Using a PASS analysis, find a stable firing rate for each actor of the SDF graph in Fig. 2.34. The 5 actors in this graph have the following functionality. count increments a token at the input, and produces a copy of the incremented value on each of its outputs. split reads two tokens at the input and distributes these tokens over each output. diff reads two tokens at the input and produces the difference of these tokens (first minus last) at the output. join reads a token on each input and produces a merged stream. join is the complement of split. snk prints the input token.

2.10. Using the quickthreads API defined earlier, create a data flow simulation for the SDF graph shown in Fig. 2.34.

2.11. Optimize the SDF graph shown in Fig. 2.34 to a single C function by implementing the schedule at compile-time and by optimizing the FIFO queues into single variables.

2.12. Convert the SDF graph in Fig. 2.34 to a single-clock hardware implementation. Perform first a multirate expansion. You can assume that the snk actor is implemented as a system-level output port.

Chapter 3
Analysis of Control Flow and Data Flow

Abstract In this chapter, we analyze the control flow and dataflow of a C program. Understanding these properties helps a designer to understand the relationship between a C program and an equivalent hardware implementation of that C program. Control edges and data edges reflect relationships between the operations of the C program, and we distinguish *control edges* from *data edges*. A control edge specifies the execution order of these operations. A data edge specifies that data produced by one operation is consumed by the second. By representing the operations of the C program as nodes of a graph, control edges and data edges define a structure called a Control Flow Graph (CFG) and a Data Flow Graph (DFG), respectively. For a hardware–software codesigner, the distinction between control edges and data edges is of great importance. Data edges will appear in any implementation target – hardware or software – of the algorithm. Control edges, on the other hand, may be removed when the algorithm executes on an architecture with sufficient implementation parallelism.

3.1 Data and Control Edges of a C Program

In the previous chapter, we discussed the data flow model of computation. Fundamental to this model is the decomposition of a system into individual nodes (*actors*), which communicates through unidirectional, point-to-point channels (*queues*). The resulting system model is represented as a graph. Such a data flow system is able to express concurrent computations that map easily into hardware as well as into software. So, the data flow model of computation illustrates how we can build system models that are equally well suited for hardware implementation as well as for software implementation.

Our objective in this chapter is to think of a C program in a similar target-independent fashion. For a software designer, a C program is always software. For a hardware–software codesigner however, a C program may be hardware or software, depending on the requirements and needs of the application. Obviously, one cannot make a direct conversion of C into hardware – a major roadblock is that hardware is parallel by nature, while C is sequential. But we can look at a C program as a

high-level description of the behavior of an implementation, without deciding on the exact nature of the implementation. At that point, we become interested in analyzing the C program in terms of its fundamental building blocks. These building blocks are the operations of the C program and the relations between them.

We define two types of relationships between the operations of a C program: data edges and control edges. At first glance, data edges and control edges are quite similar.

> A **data edge** is a relation between two operations, such that data which is produced by one operation is consumed by the other.
> A **control edge** is a relation between two operations, such that one operation has to execute after the other.

This looks similar, but it's not identical. Consider for example the following C function, which finds the maximum of two variables.

```c
int max(int a, b) {
  int r;
  if (a > b)
    r = a;
  else
    r = b;
  return r;
}
```

This function contains two assignment statements and an if-then-else branch. For the purpose of this analysis, we will equate statements in C with 'operations'. In addition, we define the entry and exit points of the function as two additional operations. Therefore, the max function contains five operations:

```c
int max(int a, b) {      // operation 1 - enter the function
  int r;
  if (a > b)             // operation 2 - if-then-else
    r = a;               // operation 3
  else
    r = b;               // operation 4
  return r;              // operation 5 - return max
}
```

To find the control edges in this function, we need to find what chains of operations can execute, based on the usual C semantics. For example, the operation 2 will always execute after operation 1. Therefore, there is a control edge from operation 1 to operation 2. An if-then-else statement introduces two control edges, one for each of the possible outcomes of the if-then-else test. If `a > b` is true, then operation 3 will follow operation 2, otherwise operation 4 will follow operation 2. Therefore, there is a control edge from operation 2 to each of operations 3 and 4. Finally, operation 5 will follow either execution of operation 3 or 4. There is a control edge from each of operations 3 and 4 to operation 5. Summing up, finding control edges

corresponds to finding the possible execution paths in the C program, and linking up the operations in these execution paths with edges.

To find the data edges in this function, we examine the data production/ consumption patterns of each operation.

```
int max(int a, b) {      // operation 1 - produce a, b
  int r;
  if (a > b)             // operation 2 - consume a, b
    r = a;               // operation 3 - consume a and (a>b),
                         //               produce r
  else
    r = b;               // operation 4 - consume b and (a>b),
                         //               produce r
  return r;              // operation 5 - consume r
}
```

The data edges are defined between operations of corresponding production/consumption. For example, operation 1 defines the value of a and b. Several operations will make use of those values. The value of a is used by operations 2 and 3. Therefore, there is a data edge from operation 1 to operation 2, as well as a data edge from operation 1 to operation 3. The same goes for the value of b, which is produced in operation 1 and consumed in operation 2 and operation 4. There is a data edge for b from operations 1 to 2, as well as from operations 1 to 4.

Control statements in C may produce data edges as well. In this case, the if-then-else statement evaluates a flag, and the *value* of that flag is needed before subsequent operations can execute. For example, operation 3 will only execute when the conditional expression (a>b) is true. We can think of a boolean flag carrying the value of (a>b) from operations 2 to 3. Similarly, operation 4 will only execute when the conditional expression (a>b) is false. There is a boolean flag carrying the value of (a>b) from operations 2 to 4.

The data edges and control edges of the operations from the max function can now be arranged in a graph, where each operation represents a node. The result is shown in Fig. 3.1, and it represents the control flow graph (CFG) and the data flow graph (DFG) for the program. In contrast to control edges, data edges are valid for a specific variable. Therefore, data edges are labeled with that variable.

We will now explore the properties of control edges and data edges more carefully and evaluate how the CFG and DFG can be created systematically for a more complicated C program.

3.2   Implementing Data and Control Edges

In the context of hardware–software codesign, the implementation target of a C program may be either hardware or software. The data edges and control edges of the C program give important clues on the implementation alternatives for that C program.
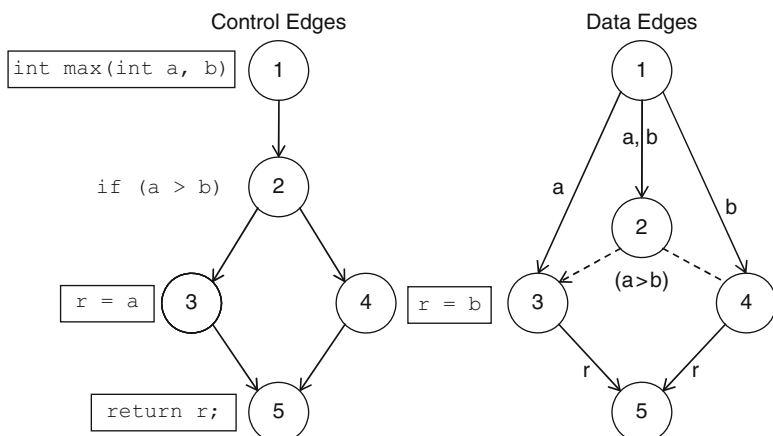
Fig. 3.1 Control edges and data edges of a simple C program

- A data edge reflects a requirement on the flow of information. If you change the flow of information, you change the meaning of the algorithm. For this reason, a data edge is a fundamental property of the behavior expressed in that C program.
- A control edge, on the other hand, is a consequence of the execution semantics of the program language, but it is not fundamental to the behavior expressed in that C program.

In hardware–software codesign, we are looking to design the architecture that fits best to a given algorithm. Even though we may start from a C program, the eventual target of this program may not be a processor. It may be a processor with a coprocessor, or a full hardware implementation. One question then inevitably arises: what are the important parts of a C program that will be present in any implementation of that program? The answer to this question is given by the control edges and data edges of the program, and is summarized as follows:

> A **data edge** must always be implemented regardless of the underlying architecture.
> A **control edge** may be removed if the underlying architecture can handle the resulting concurrency.

In other words, control edges can be removed by providing enough parallelism in the underlying architecture. For example, even though the semantics of C assume sequential execution, modern microprocessors are able to run multiple instructions in parallel, even when they would belong to two different sequential C statements. These microprocessors are able to modify the control edges of the flow of instructions at runtime, without breaking the data edges within that instruction flow.
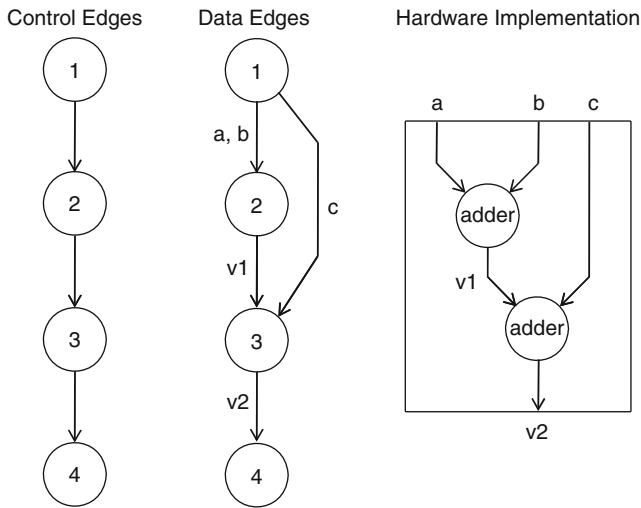
Fig. 3.2 Hardware implementation of a chained addition

Here is another example. The following function adds up three numbers using multiple operations.

```
int sum(int a, b, c) {   // operation 1
  int v1;
  v1 =  a + b;           // operation 2
  v2 = v1 + c;           // operation 3
  return v2;             // operation 4
}
```

It is straightforward to draw a fully parallel hardware implementation of this function. This implementation is shown, together with the DFG and CFG of the function, in Fig. 3.2. The similarity between the set of *data* edges and the interconnection pattern of the hardware is obvious. The control edges, however, carry no meaning for the hardware implementation, since hardware is parallel. The structure shown on the right of Fig. 3.2 will complete the addition in a single clock cycle.

In the next chapter, we will develop a systematic method to derive the CFG and the DFG of C programs. As an application, we will then use these data structures to convert C programs into a hardware implementation.

3.3   Contruction of the Control Flow Graph

A C program can be systematically converted into an intermediate representation called a CFG. A CFG is a graph that contains all the control edges of a program. Each node in the graph represents a single operation (or C statement). Each edge of the graph indicates a control edge, i.e., an execution order for the two operations connected by that edge.
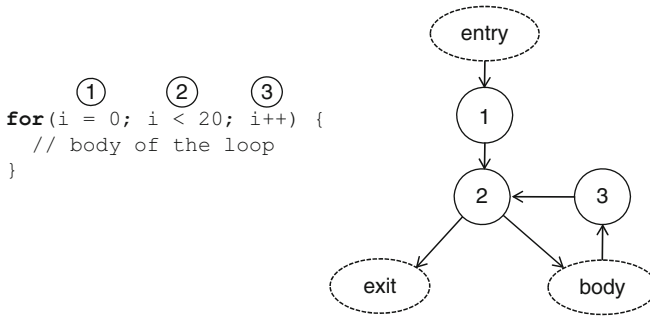
Fig. 3.3 Control flow graph (CFG) of a `for` loop

Since C executes sequentially, this conversion is straightforward. However, some cases require further attention. Control statements (such as loops) may require multiple operations. In addition, when decision-making is involved, multiple control edges may originate from a single operation.

Consider the `for` loop in C, as illustrated next.

```
for (i=0; i < 20; i++) {
  // body of the loop
}
```

This statement includes four distinct parts: the loop initialization, the loop condition, the loop-counter increment operation, and the body of the loop. The `for` loop thus contributes three operations to the CFG, as shown in Fig. 3.3. The dashed nodes in this figure (`entry`, `exit`, `body`) represent other parts of the C program and may contain a complete single-entry, single-exit CFG.

The `do-while` loop and the `while-do` loop are similar iterative structures. We can draw a template for each of them, including the `if-then-else` statement. This is illustrated in Fig. 3.4.

As an example, let's create the CFG of the following C function. This function calculates the Greatest Common Divisor (GCD) using Euclid's algorithm.

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

To construct the CFG of this program, we convert each statement to one or more operations in the CFG, and next connect the operations using control edges. The result of this conversion is shown in Fig. 3.5.

In a CFG it is useful to define a *control path*, a path between two nodes in the CFG. For example, each nonterminating iteration of the `while` loop of the

```
    (1)
if (a < b) {
    // true branch
} else {
    // False branch
}
```

```
    (1)
while (a < b) {
    // loop body
}
```

```
do        {
    // loop body
} while (a<b)
    (1)
```
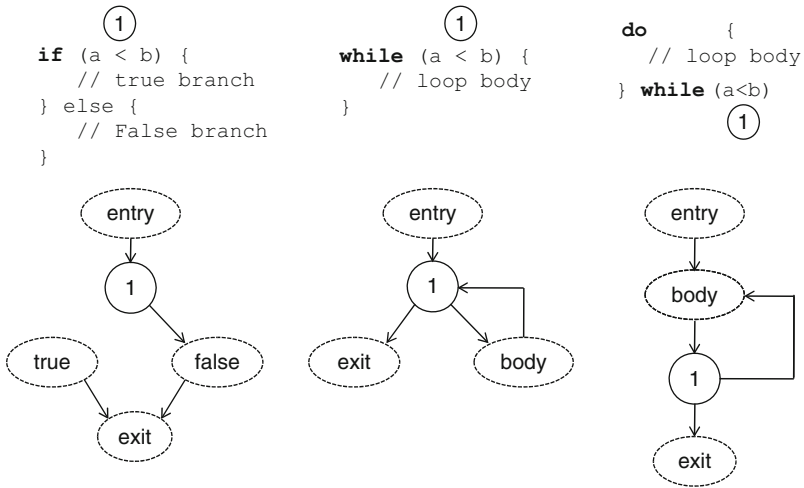
Fig. 3.4 CFG of if-then-else, while-do, do-while


```
1: int gcd(int a, int b) {
2:    while(a != b) {
3:      if(a > b)
4:        a = a - b;
        else
5:        b = b - a;
        }
6:    return a;
      }
```
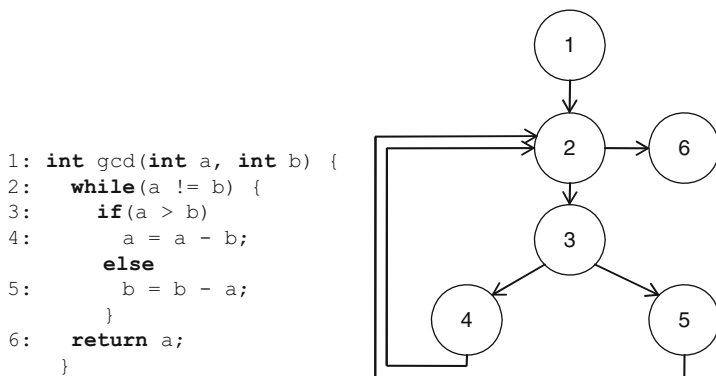
Fig. 3.5 CFG of the CGD program

C program will follow either the path 2-3-4-2 or else 2-3-5-2. Control paths will be important in the construction of the DFG, which is discussed next.

3.4 Construction of the Data Flow Graph

A C program can be systematically converted into a data structure called a DFG. A DFG is a graph that reflects all the data edges of a program. Each node in the graph represents a single operation (or C statement). Each edge of the graph indicates a data edge, i.e., a production/consumption relationship between two operations in the program.

Obviously, the CFG and the DFG will contain the same set of nodes. Only the edges will be different. While it is possible, in principle, to directly derive the DFG from a C program, it is easier to create the CFG first and then to derive the DFG out of it. The trick is to trace control paths and at the same time identify corresponding read- and write operations of variables.

Let us assume initially that we will analyze programs without array expressions and pointers, and extend our conclusions later to those cases as well. The procedure to recover the data edges related to assignment statements is as follows:

1. In the CFG, select a node where a variable is used as an operand in an expression. Mark that node as a read-node.
2. Find the CFG nodes that assign that variable. Mark those nodes as write-nodes.
3. If there exists a direct control path from a write-node into a read-node that does not pass through another write-node, then you have identified a data edge. The data edge originates at the write-node and ends at the read-node.
4. Repeat the previous steps for all variable-read nodes.

This procedure identifies all data edges related to assignment statements, but not those originating from conditional expressions in control flow statements. However, these data edges are easy to find: they originate from the condition evaluation and affect all the operations whose execution depends on that condition.

Let us derive the DFG of the GCD program given in Fig. 3.5. According to the procedure, we pick a node where a variable is read. For example, node 5 in the CFG reads variables a and b.

```
    b = b - a;
```

Concentrate on the b operand first. We need to find all nodes that write into b. If the CFG is available, we can start by tracing precedessor nodes of this node until we hit one that writes into b. The predecessors of node 5 include: node 3, node 2, node 1, node 4, and node 5. Both nodes 1 and nodes 5 write into b. In addition, there is a path from node 1 to node 5 (e.g., 1-2-3-5), and there is also a path from node 5 to node 5 (e.g., 5-2-3-5). In each of these paths, no other nodes write into b apart from the final node 5. In this case, this path is called a *direct* path. A data edge connects the start and end node of a direct path. Thus, there is a data edge for b from node 1 to node 5 and from node 5 to node 5. Starting from the same read-node 5, we can also find all predecessors that define the value of operand a. In this case, we find that node 1 and node 4 write into a, and that there is a direct path from node 1 to node 5, as well as from node 4 to node 5. Therefore, there is a data edge for a from node 1 to node 5, and from node 4 to node 5.

To complete the set of data edges into node 5, we also need to identify all conditional expressions that affect the outcome of node 5. Considering the control statements in this function, we see that node 5 depends on the condition evaluated in node 3 (a > b) as well as the condition evaluated in node 2 (a != b). There is a data edge from each of node 2 and node 3 to node 5, carrying the outcome of this condition. The collection of all data edges into node 5 can now be annotated into the DFG, resulting in the partial DFG of Fig. 3.6.

Fig. 3.6 Incoming data
edges for node 5 in the CGD
program
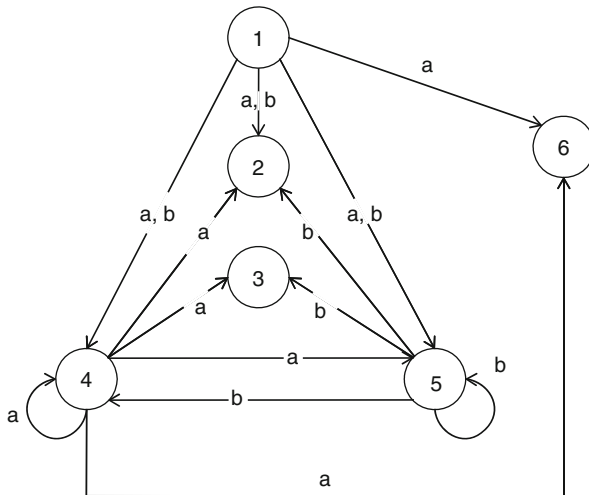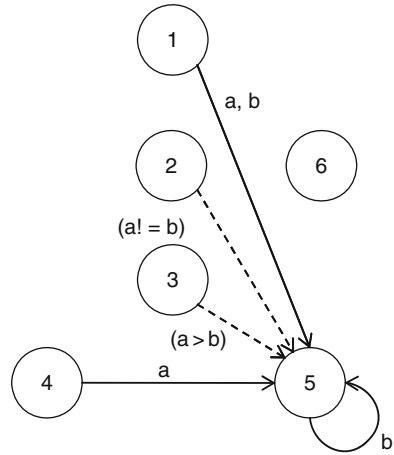
Fig. 3.7 Data edges for all nodes in the greatest common divisor (GCD) program, apart from
edges carrying condition variables

We can repeat this procedure for each other node of the graph in order to construct the complete DFG. The result of this analysis is shown in Fig. 3.7. This graph does not contain the data edges originating from conditional expressions.

How do we draw a DFG of a program with pointers and arrays? There are several approaches to solve this, and they depend on the requirements of the analysis and the amount of effort we are able to invest in the analysis.

First, observe that an indexed variable is not really different from a scalar variable as long as we can exactly determine the value of the index during the data-flow analysis. Similarly, data edges resulting from pointers are easy to find if we can

```
1: int L[3] = {10, 20, 30};
2:    for (int i=1; i<3; i++)
3:       L[i] = L[i] + L[i-1];
```
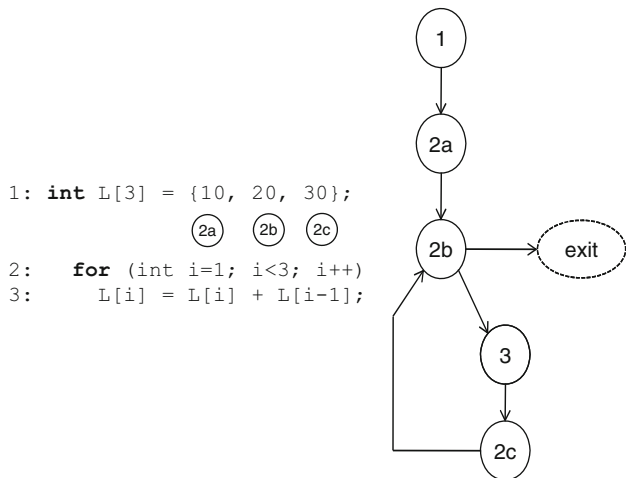
Fig. 3.8 CFG for a simple loop with an indexed variable

exactly determine the value of the pointer. However, in practice, this may be hard. An indexed variable may have an index expression that combines multiple loop counters or even a variable which is unknown at compile time.

We may be able to relax the analysis requirements and simplify the data-flow analysis. In many applications, the upper bound and lower bound of an index expression are known. In that case, we may consider any *write* operation into the range of indices as a single write, and any *read* operation into the range of indices as a single read. For cases when an entire range of indices would map into a single memory (a single register file, or a single-port RAM memory), this type of data-flow analysis may be adequate.

We illustrate this approach using the following example. The CFG of this loop is shown in Fig. 3.8.

```
int L[3] = {10, 20, 30};
for (int i=1; i<3; i++)
  L[i] = L[i] + L[i-1];
```

To create the DFG of this program, proceed as before. Find all nodes that read from a variable, and find the nodes that write into that variable over a direct path in the CFG. As discussed above, we can handle the analysis of the indexed variable L in two different ways: In the first approach, we look upon L as a single monolithic variable, such that a read from any location from L is treated as part of the same data edge. In the second approach, we distinguish individual locations of L, such that each location of L may contribute to a different data edge. The first approach is illustrated in Fig. 3.9a, while the second approach is illustrated in Fig. 3.9b.

When the individual locations of L cannot be distinguished by a data edge, additional information is needed to extract the entry of interest. For this reason, node 3 in Fig. 3.9a has an additional data edge to provide the loop counter i. Thus, in
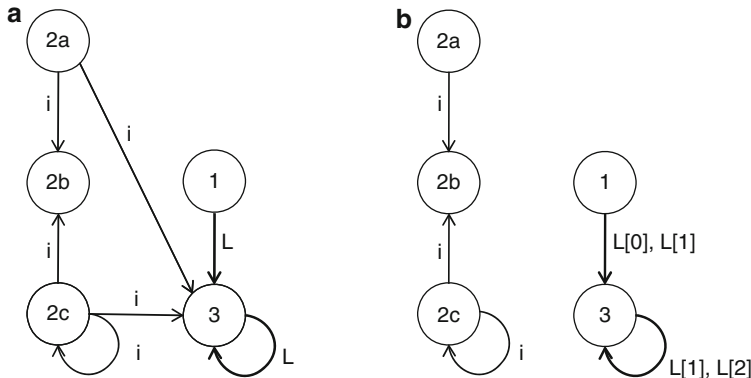
Fig. 3.9 DFG for a simple loop with an indexed variable: (**a**) Treating the indexed variable as a single variable; (**b**) treating the indexed variable as a collection of variables

Fig. 3.9a, reading entry `L[i]` means: read all the entries of `L` and then select one using `i`. In Fig. 3.9b, reading entry `L[i]` means three different read operations, one for each value of `i`.

Index analysis on arbitrary C programs quickly becomes very hard to solve. Yet, hardware–software codesigners often only have a C program to start their design with. Insight into the data-flow of a complex C program is essential for a successful hardware–software codesign.

This concludes an introduction to control-flow and data-flow analysis of C programs. The next section shows an application for the techniques we have covered so far. By deriving the CFG and the DFG, we are to translate simple C programs systematically into hardware. This technique is by no means a universal one; its purpose is to clarify the meaning of control edges and data edges.

3.5 Application: Translating C to Hardware

A nice application of data-flow and control-flow analysis of a C program is the systematic translation of C into hardware. In the general case, this is a very complex problem. A given C program can be mapped into hardware in many different ways. Instead, our objective is to illustrate how data-edges and control-edges can be used in one possible translation strategy. We will focus on a simplified version, which includes the following restrictions for the input programs.

- We translate only scalar C code (no pointers and no arrays).
- We implement each C statement in a single clock cycle.

3.5.1 Designing the Datapath

Given a C program, we first create the CFG and the DFG. Next, we use the data edges and control edges to implement the hardware. The data edges will help us define the connectivity in the datapath. The control edges will help us define the control signals used by the datapath. The data edges show us how to interconnect the datapath components. With the CFG and DFG available, the following rules will define the implementation of the hardware datapath.

1. Find the C expression embedded in each node of the CFG, and create an equivalent combinational circuit to implement the expression. For example, if a node in the CFG corresponds to the C statement `a = b - a;`, then the C expression embedded in that statement is `b - a`. The combinational circuit required to implement this expression is a subtractor. Conditional expressions generate datapath elements, too. The outputs of these expressions become the flags used by the hardware controller of this datapath.
2. Each variable in the C program is translated into a register with a multiplexer in front of it. The multiplexer is needed when multiple sources may update the register. By default, the register will update itself. The selection signals of the multiplexer will be driven by the controller.
3. The datapath circuit and the register variables are connected based on the nodes and data edges in the DFG. Each assignment operation connects a combinational circuit with a register. Each data edge connects a register with the input of a combinational circuit. Finally, we also connect the system-inputs and system-outputs to inputs of datapath circuits and register outputs, respectively.

The GCD program can now be converted into a hardware implementation as shown in Fig. 3.10. We need two registers in this datapath for each of the variables `a` and `b`. The conditional expressions for the `if` and `while` statement need an equality-comparator and a bigger-than comparator. The subtractions `b-a` and `a-b` are implemented using a subtractor. The connectivity of the components is defined by the data edges of the DFG.

The resulting datapath has two data inputs (`in_a` and `in_b`) and one data output (`out_a`). The circuit requires two control variables (`upd_a` and `upd_b`) to operate, and it produces two flags (`flag_while` and `flag_if`). The control variables and the flags are used by the controller of this datapath.

3.5.2 Designing the Controller

How do we create the controller for this datapath such that it implements the GCD algorithm? This control information is present in the C program and is captured in the CFG. In fact, we can translate the CFG almost straight into hardware by considering it to be a finite state machine (FSM) specification.

```
1: int gcd(int a, int b) {
2:    while (a != b) {
3:       if (a > b)
4:          a = a - b;
       else
5:          b = b - a;
    }
6:    return  a;
   }
```
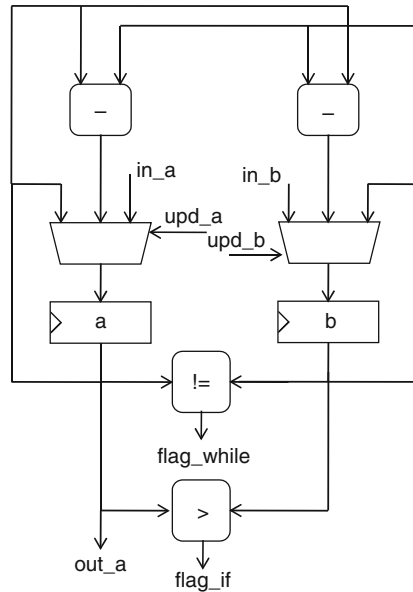

Fig. 3.10 Hardware implementation of GCD datapath
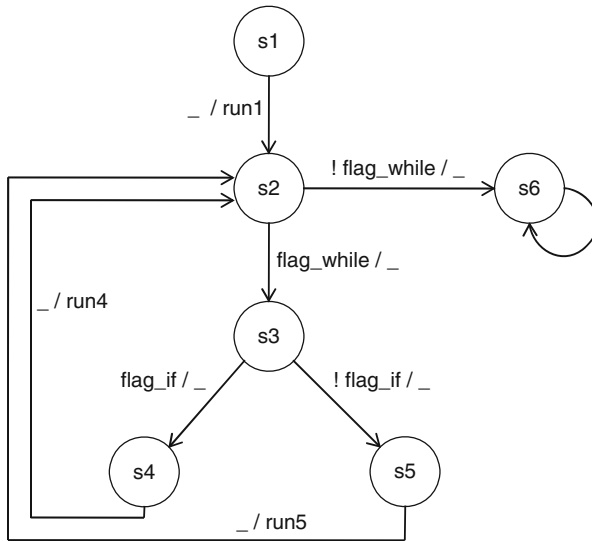
Fig. 3.11 Control specification for the GCD datapath

A FSM specification for the GCD algorithm is shown in Fig. 3.11. The correspondence with the CFG is obvious. Each of the transitions in this FSM takes 1 clock cycle to complete. The activities of the FSM are expressed as condition/ command tuples. For example, _ / run1 means that during this clock cycle, the

value of the condition flags is a don't-care, while the command for the datapath is
the symbol run1. Similarly, flag_while / _ means that this transition is condi-
tional on flag_while being true, and that the command for the datapath is a hold
operation. A *hold operation* is one which does not change the state of the datapath,
including registers. The command set for this FSM includes (_, run1, run4,
run5). Each of these symbols represents the execution of a particular node of the
CFG. The datapath control signals can be created by additional decoding of these
command signals. In this case of the GCD, the datapath control signals consist of
the selection signals of the datapath multiplexers.

A possible implementation of the GCD controller is shown in Fig. 3.12. Each
clock cycle, the controller generates a new command based on the current state
and the value of flag_while and flag_if. The commands run1, run4, and
run5 are decoded into upd_a and upd_b. The table in Fig. 3.12 indicates how
each command maps into these control signals. The resulting ensemble of datapath
and FSM, as illustrated in Fig. 3.12 is called a Finite State Machine with Datapath
(FSMD). This concept is central to custom hardware design, and we will discuss
design and modeling of FSMD in further detail in Chap. 4.

The operation of this hardware design is illustrated with an example in Table 3.1.
Each row of the table corresponds to 1 clock cycle. It takes 8 clock cycles to evaluate
the GCD of 6 and 4.

In conclusion, this example shows an application of data-flow and control-flow
analysis of a C program. It illustrates that the control-flow and data-flow can lie at
the basis of hardware design as well as software design. There are many suboptimal
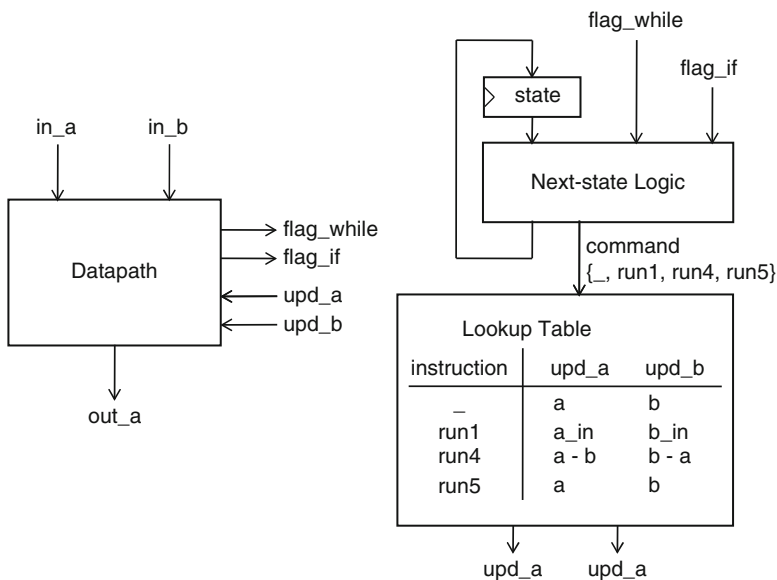
Fig. 3.12 Controller implementation for the GCD datapath

Table 3.1 Operation of the hardware to evaluate GCD(4,6)

Cycle	a	b	State	`flag_if`	`flag_while`	Next State	`upd_a`	`upd_b`
1	_	_	s1	–	–	s2	`in_a`	`in_b`
2	6	4	s2	1	1	s3	`a`	`b`
3	6	4	s3	1	1	s4	`a`	`b`
4	6	4	s4	1	1	s2	`a-b`	`b`
5	2	4	s2	0	1	s3	`a`	`b`
6	2	4	s3	0	1	s5	`a`	`b`
7	2	4	s5	0	1	s2	`a`	`b-a`
8	2	2	s2	0	0	s6	`a`	`b`
9	2	2	s6	–	–	s6	`a`	`b`

elements left. First, we did not address the use of arrays and pointers. Second, the resulting implementation in hardware is not very impressive: the resulting parallelism is limited to a single C statement per clock cycle, and operations cannot be shared over operator implementations. For instance, two subtractors are implemented in hardware in Fig. 3.10, but only one is used at any particular clock cycle.

3.6 Single-Assignment Programs

Converting C programs into hardware at one cycle per C-statement is not very impressive, in particular because most microprocessors can already do that. Can we do any better than this? That is, can we create a translation strategy that will allow the execution of multiple C statements per clock cycle? To answer this question, we need to understand the limitations of our current translation strategy, as described in Sect. 3.5. In our current approach, each C statement takes a single clock cycle to execute because each variable is mapped into a register. Information takes a full clock cycle to propagate from the input of a register to the output. Therefore, each variable assignment takes a full clock cycle to take effect; it takes a full clock cycle before the value of a variable assignment is available at the correponding register output. Thus, it seems that the mapping of each variable into a register, which by itself seemed a sensible decision, also introduces a performance bottleneck. If we want to run at a faster pace than one statement per clock cycle, we will need to revise this variable-to-register mapping strategy.

The above observation triggers another question, namely: how did it help us to map each variable into a register ? Consider that each variable may be assigned in multiple C statements, while the same variable may also be used as an operand in multiple C expressions. By allocating a register for each variable, we concentrate all the assignments in one location, ready to be used for all expressions that may require the resulting value. Thus, it is as if the register concentrates all the data edges for a given variable to flow through a single, global storage location. This makes assignment statements easy to manage, but it hurts performance.

We can do better than that. It is possible to formulate a C program in a form called a *single-assignment program*. The key property of a single-assignment program is exactly what its name refers: each variable in that program is assigned only a single time within a single lexical instance of the program. In order to convert a C program into a single-assignment program, we may need to introduce extra variables, as illustrated by the following example. Assume we have a C snippet that looks as follows:

```
a = a + 1;
a = a * 3;
a = a - 2;
```

This section of code contains three assignments on `a`. Using our previous strategy, we would need 3 clock cycles to execute this fragment. Instead, we can rewrite this program so that each variable is assigned only a single time. This requires the introduction of additional variables.

```
a2 = a1 + 1;
a3 = a2 * 3;
a4 = a3 - 2;
```

The difference with the previous program is that each assignment is matched by a single read operation. In other words, the single-assignment form of the program indicates the data edges of the program in the source code: assigning a given variable indicates the start of data edges, while reading the same variable indicates the end of the data edge.

After a program is in single-assignment form, we can apply a better register-assignment strategy. For instance, in the example above, the cycle count may be reduced by mapping `a2` and `a3` to a wire, while keeping `a4` in a register. This would group the three C statements in a single clock cycle.

In converting normal C programs to single-assignment form, care must be taken that all assignments are taken into account. In particular, when variables are assigned under different control conditions or in different levels of a loop nesting structure, the single-assignment form may become ambiguous. Consider the following example: a loop which makes the sum of the numbers 1–5.

```
a = 0;
for (i = 1; i < 6; i++)
  a = a + i;
```

In the single-assignment form, the assignments to `a` can be made unique, but it remains unclear what version of `a` should be read inside of the loop.

```
a1 = 0;
for (i = 1; i < 6; i++)
  a2 = a + i;  // which version of a to read?
```

The answer is that both `a1` and `a2` are valid solutions for this program: it really depends on the iteration within the loop. When we first enter the loop, we would write:

```
  a2 = a1 + 1;
```

After the first loop iteration, we would write instead:

```
a2 = a2 + i;   // when i > 1
```

To resolve this ambiguity, single-assignment programs use a `merge` function and operation that can merge multiple data edges into one. We can introduce a new variable `a3` to hold the result of the `merge` function, and now formulate the program into single-assignment form as follows:

```
a1 = 0;
for (i = 1; i < 6; i++) {
  a3 = merge(a2, a1);
  a2 = a3 + i;
```

The hardware implementation of the `merge` function would be a multiplexer, under control of an appropriate selection signal. In this case, `(i==0)` would be an appropriate selection signal. The above translation rules can be used to more complicated programs as well. For example, the GCD program can be converted as follows:

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

The equivalent single-assignment form of the GCD is shown below. The conditional expression in the `while` statement uses variables from either the function input or else the body of the loop. For this reason, the conditional expression uses the `merge` function as an operand.

```
int gcd(int a1, int b1) {
  while (merge(a1, a2) != merge(b1, b2)) {
    a3 = merge(a1, a2);
    b3 = merge(b1, b2);
    if (a3 > b3)
      a2 = a3 - b3;
    else
      b2 = b3 - a3;
  }
  return a2;
}
```

Note that a single assignment program by itself is not sufficient to have an efficient hardware implementation (or an implementation that performs better than a single statement per clock cycle). We have only provided a hint to how we may be able to obtain eventually a better implementation.

There are algorithms to derive the single-assignment form of a program automatically. Software compilers use these representations, called *static single-assignment form*, extensively to implement advanced code optimizations.

3.7 Summary

In this chapter we discussed the data flow and control flow properties of a C program. These properties can be modeled into two graph structures, called the DFG (data flow graph) and CFG (control flow graph). The DFG and CFG can be derived systematically starting from the C source. The data flow and control flow of a program help the designer to understand the implementation alternatives for that program. We showed that data flow is preserved over different implementations in hardware and software, while control flow may change drastically. Indeed, a sequential or a parallel implementation of a given algorithm may have a very different control flow. We used this insight to define a simple mechanism to translate C programs into hardware.

3.8 Further Reading

The material discussed in this chapter will be found, in expanded form, in a textbook on compiler construction such as Muchnick (1997) or Appel (1997). In particular, these books provide further details on the analysis available on the CFG and DFG.

High-level synthesis is a research area that investigates the automated mapping of programs written in C and other high-level languages into lower-level architectures. In contrast to compilers, which target a processor with a fixed architecture, high-level synthesis does support some freedom in the target architecture. High-level synthesis has advantages and limitations; proponents and opponents. Refer to Gupta et al. (2004) to see what can be done; read Edwards (2006) as a reminder of the pitfalls.

During our discussion on the mapping of C programs into hardware, we did explicitly rule out pointers and arrays. In high-level synthesis, the design problems related to implementation of memory elements are collected under the term *memory management*. This includes for example the systematic mapping of array variables into memory elements, and the efficient conversion of indexed expressions into memory addresses. Refer to Verbauwhede (1994) for an introduction to memory management issues.

The original work on Static Single Assignment (SSA) was by Cytron Cytron et al. (1991). A discussion on how the SSA form can assist in the translation of C software into hardware may be found in Kastner et al. (2003).

3.9 Problems

3.1. Do the following for the C program in Listing 3.1.

Listing 3.1 Program for Problem 3.1
```c
int addall(int a, int b, int c, int d) {
  a = a + b;
  a = a + c;
  a = a + d;
  return a;
}
```

(a) Derive and draw the CFG and the DFG.
(b) The *length* of a path in a graph is defined as the number of edges in that path.
    Find the longest path in the DFG and give an interpretation for that quantity.
(c) Rewrite the program in Listing 3.1 so that the maximal path length in the DFG
    decreases. Assume that you can do only a single arithmetic operation per C
    statement. Draw the resulting DFG.

3.2. Draw the CFG and the DFG of the program in Listing 3.2. Include all control
dependencies in the CFG. Include the data dependencies for the variables a and b in
the DDG.

Listing 3.2 Program for Problem 3.2
```c
int count(int a, int b) {
  while (a < b)
    a = a * 2;
  return a + b;
}
```

3.3. Design a datapath in hardware for the program shown in Listing 3.3. Allocate
registers and operators. Indicate control inputs required by the datapath, and condi-
tion flags generated by the datapath.

Listing 3.3 Program for Problem 3.3
```c
unsigned char mysqrt(unsigned int N) {
  unsigned int x,j;
  x = 0;
  for(j= 1<<7; j != 0; j>>=1) {
    x = x + j;
    if( x*x > N)
      x = x - j;
  }
  return x;
}
```

3.4. A well-structured C program is a program that only contains the following
control statements: if-then-else, while, do-while, and for. Consider the four CFG
in Fig. 3.13. Which of these CFG does correspond to a well-structured C program?
Note that a single node in the CFG may contain more than a single statement, but it
will never contain more than a single decision point.
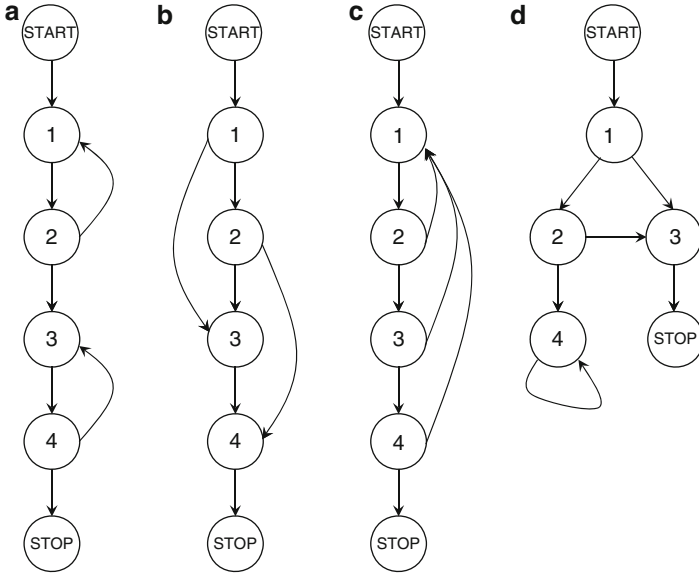
Fig. 3.13 CFG for Problem 3.4: (**a**) Case 1 (**b**) Case 2 (**c**) Case 3 (**d**) Case 4

3.5. Draw the DFG for the program in Listing 3.4. Assume all elements of the array `a[]` to be stored in a single resource.

Listing 3.4 Program for Problem 3.5

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int findmax() {
  int max, i;

  max = a[0];
  for (i=1; i<10; i++)
    if (max < a[i])
      max = a[i];

  return max;
}
```

3.6. Design a hardware implementation (datapath and controller) for the program in Listing 3.4. Assuming that the elements of array `a[]` are all stored in a memory with a single read port. Figure 3.14 illustrates such a memory. The time to lookup an element is very short; thus, you can think of this memory as a combinational element.

3.7. Convert the program in Listing 3.3 to single-assignment form. Compare the location of the `merge()` functions with the solution of Problem 3.3. Using the SSA form of the program, optimize the solution so that each iteration of the `for`-loop will take no more than a single clock cycle to complete.

Fig. 3.14 A single-port
read-only memory, used to
solve Problem 3.6
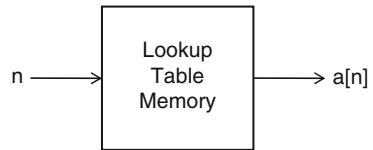
3.8. This problem requires access to a GNU Compiler (gcc) version 4.0 or above.
Start by writing up the listing of Problem 3.3 in a file can call the file mysqrt.c.

(a) Compile this function using the following command line.

```
gcc -c -fdump-tree-cfg mysqrt.c
```

The compiler generates an object file `mysqrt.o`, as well as a file with debug
information. Under `gcc 4.0.2`, the name of that file is `mysqrt.c.t13.
cfg`. Open `mysqrt.c.t13.cfg` in a text editor. This is a textual represen-
tation of a CFG as produced by the GCC compiler. Compare this CFG to one
you would draw by hand. In particular, comment on the following two obser-
vations: (1) Nodes in a CFG can be grouped together when they all belong to a
single path of the CFG with a single exit point. (2) `goto` and `if-then-else`
are adequate to capture all control statements in C (such as `for`, `while`, and
so on).

(b) Compile this function using the following command line. `O2` turns on the com-
piler optimizer so that GCC will try to produce better code.

```
gcc -c -O2 -fdump-tree-ssa mysqrt.c
```

The compiler generates an object file and a file with debug information. Under
`gcc 4.0.2`, the name of that file is `mysqrt.c.t16.ssa`.
Open `mysqrt.c.t16.ssa` in a text editor. This is a textual representation of
the SSA as produced by the GCC compiler. Find the `merge` functions in this
file and compare the number and location of these functions in the CFG. Did
you find the same number of `merge` functions in Problem 3.7? Do they have
the same location?

Part II
The Design Space of Custom Architectures

This second part of this book describes a range of custom architectures, which have varying degrees of complexity and flexibility. Starting from very simple cycle-based hardware models, we gradually add control structures to increase their flexibility. This leads the discussion into FSMD (Finite State Machine with Datapath), micro-programmed architectures, general-purpose embedded cores, and finally system-on-chip architectures. An over-arching theme, besides flexibility, is the trade-off of that flexibility with performance. This trade-off helps designers to navigate the design space of custom architectures.

Chapter 4
Finite State Machine with Datapath

Abstract In this chapter, we introduce an important building block for efficient custom hardware design: the *Finite State Machine with Datapath* (FSMD). An FSMD combines a controller, modeled as a finite state machine (FSM) and a datapath. The datapath receives commands from the controller and performs operations as a result of executing those commands. The controller uses the results of datapath operations to make decisions and to steer control flow. The FSMD model will be used throughout the remainder of the book as the reference model for the 'hardware' part of hardware/software codesign. We will introduce a syntax for FSMD by means of the GEZEL language. The cosimulation tools used in conjunction with this book rely on the GEZEL language. We will discuss several alternate language mappings for the FSMD language in GEZEL, including VHDL, Verilog, and SystemC. Thus, GEZEL is used as a generic shorthand modeling mechanism for FSMD. Finally, we will also describe a few formal properties of the FSMD model, and we define a *proper FSMD* as one which leads to a race-free and deterministic hardware implementation.

4.1 Cycle-Based Bit-Parallel Hardware

In this chapter, we develop a model to systematically describe custom hardware consisting of a controller and a datapath. Together with the model, we will also learn to capture hardware designs into a language, called GEZEL. This section and the next one describe how to create datapath modules. Further sections will explain the control model and the combination of control and datapath into an Finite State Machine with Datapath (FSMD).

We will create cycle-based hardware models. In such models, the behavior of a circuit is expressed in steps of a single clock cycle. This abstraction level is very common in digital design, and it is captured with the term *synchronous* design. We will design circuits with a single, global clock signal.

4.1.1 Wires and Registers

We start by discussing the variables that are used to describe synchronous digital hardware. The example in Listing 4.1 shows a 3-bit counter.

Listing 4.1 A 3-bit counter module
```
1  reg a : ns(3); // a 3-bit unsigned register
2  always {
3    a = a + 1;   // each clock cycle, increment a
4  }
```

This fragment represents a 3-bit register, called a. Each clock cycle, the value of a is incremented by one. Since a is a 3-bit value, the register will count from 0 to 7, and then wrap around. The initial value of a is not specified by this code. However, we will use the convention that the initial value of registers is zero.

Let us look a bit closer at the expression 'a = a + 1' and describe precisely what it means. The right-hand side of this expression, a+1, reads the value of the register a, and adds one to that value. The left-hand side of this expression assigns that value to a, and thus writes into the register. Figure 4.1 gives an equivalent circuit diagram for 'a = a + 1', and it illustrates a key feature of a register: the input and the output of a register can have a different value. The input and the output are each connected to a different bus of wires. The timing diagram in Fig. 4.1 illustrates the circuits' operation. Before the very first clock edge, the output of a is initialized to its initial value of 0. At the very first clock edge, the register will be incremented, which means that the value at the input of a is copied to the output of a.

Going back to Listing 4.1, we see that the always statement describes the activities in the model as a result of *updating* the registers. As shown in the diagram, this happens at the upgoing clock edge. This is the only time-related aspect of
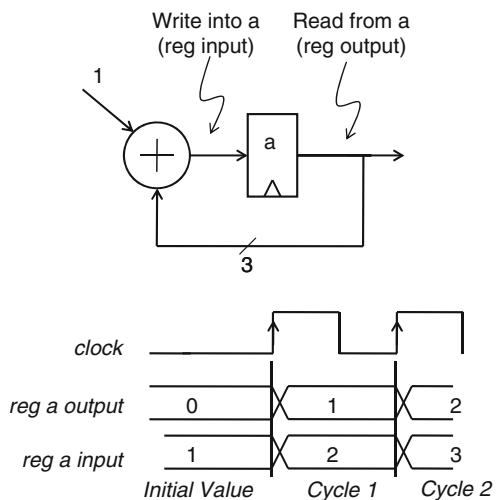
Fig. 4.1 Equivalent Hardware for the 3-bit counter

Listing 4.2   Another 3-bit counter module
```
1  reg a : ns(3); // a 3-bit unsigned register
2  sig b : ns(3); // a 3-bit unsigned signal
3  always {
4     b = a + 1;
5     a = b;
6  }
```

Listing 4.3   Yet another 3-bit counter module
```
1  reg a : ns(3); // a 3-bit unsigned register
2  sig b : ns(3); // a 3-bit unsigned signal
3  always {
4     a = b;
5     b = a + 1;
6  }
```

the model. The time required to execute a+1 is unspecified, and the expression will be evaluated as soon as the output of the a register changes. Note also that Listing 4.1 does not contain an explicit clock signal: the model executes simply based on the semantics of a register variable.

Besides a register variable, there is another variable type, called a *signal*. A signal has the same meaning as a wire or, in case of a multibit signal, as a bundle of wires. Listing 4.2 illustrates how a signal is created and used. A signal instantly takes up the value of the expression assigned to it. Thus, the value of b in Listing 4.2 will instantly reflect the value of a+1. The circuit diagram corresponding to this program looks identical to the diagram shown in Fig. 4.1. However, in this case, we have a specific name for the value at the *input* of the register a, namely the signal b.

A signal has no memory. When a signal value is used on the right-hand side of an expression, it will return the value assigned to the signal during that clock cycle. This has a particular effect on the program shown in Listing 4.2: the lexical order of expressions has no meaning. Only the data flow between reading/writing registers and signals is important. For example, the program in Listing 4.3 has exactly the same behavior as the program in Listing 4.2. One can think of the difference between registers and signals also as follows: When a register is used as an operand in an expression, it will return the value assigned to that register during the *previous* clock cycle. When a signal is used as an operand in an expression, it will return the value assigned to that signal during the *current* clock cycle. Thus, registers implement communication across clock cycles, while signals implement communication within a single clock cycle.

Because a signal has no memory, it cannot have an initial value. Therefore, the value of a signal remains undefined when it is not assigned during a clock cycle. It is illegal to use an undefined signal as an operand in an expression, and the GEZEL simulator will flag this as a runtime error. Another case which is unsupported is the use of signals in a circular definition, such as for example shown in Listing 4.4. It is impossible to determine a stable value for a or b during any clock cycle. This type of code will be rejected as well by the GEZEL simulator with a runtime error message. In Sect. 4.6, we define the rules for a properly formed FSMD more precisely.

Listing 4.4 A broken 3-bit counter module

```
1  sig a : ns(3);
2  sig b : ns(3);
3  always {
4    a = b;
5    b = a + 1;  // this is not a valid GEZEL program!
6  }
```

4.1.2 Precision and Sign

In contrast to C, hardware registers and signals can have an arbitrary wordlength, from a single bit up to any value. It is also possible to mix multiple wordlengths in an expression. In addition, registers and signals can be unsigned or signed.

The wordlength and the sign of a register or signal are specified at the creation of that register or signal. The following example creates a 4 bit unsigned register a and a 3 bit signed signal b. The representation of b follows two's complement format: the weight of the most significant bit of b is negative.

```
reg a : ns(4);    // unsigned 4-bit value
sig b : tc(3);    // signed 3-bit value
```

In an expression, registers and signals of different wordlengths can be combined. The rules that govern the precision of the resulting expression are as follows:

- The evaluation of an expression will never loose precision. All operands will automatically adapt their precision to a compatible wordlength.
- Assigning the result of an expression, or casting an expression type, will adjust the precision of the result.

As an example, the code shown in Listing 4.5 will store the value 12 in register a. Walking step by step through this code, the precision of each expression is evaluated as follows: First, the constant 3 needs to be assigned to b. A constant is always represented with sufficient bits to capture it as a two's complement number. In this case, you can express the constant 3 as a 3-bit two's complement number with the bit pattern 011. When assigning this 3-bit value to b, the lower two bits will be copied, and the bitpattern in b becomes 11. On line 5 of the code, we add the constant 9 to b. The bitpattern corresponding to the decimal constant 9 is 1001. To add the bitpattern 11 and 1001 as unsigned numbers, we extend 11 to 0011 and perform the addition to find 1100, or 12 in decimal. Finally, the bitpattern 1100 is assigned to a, which can accommodate all bits of the result.

When the length of an operand is extended, the rules of sign extension will apply. The additional bits beyond the position of the most significant bit are copies of the sign bit, in the case of two's complement numbers, or zeros, in the case of unsigned numbers. In Listing 4.6, a is a 6-bit unsigned number, and b is a 2-bit *signed* number. After assigning the constant 3 to b, the value of b will be -1, and the bit pattern of b equals 11. The result of subtracting b and 3 is -4 in decimal, which is 100 as a bitpattern (with the msb counting as a sign bit). Finally, assigning -4 to a 6-bit

Listing 4.5 Adding up 4 bit and 2 bit

```
1  reg a : ns(4);   // a 4-bit unsigned number
2  sig b : ns(2);   // a 2-bit unsigned number
3  always {
4    b = 3;          // assign 0b(011) to b
5    a = b + 9;      // add 0b(11) and 0b(1010)
6  }
```

Listing 4.6 Subtracting 2 bit and 4 bit

```
1  reg a : ns(6); // a 6-bit unsigned number
2  sig b : tc(2); // a 2-bit signed number
3  always {
4    b = 3;        // assign 0b(011) to b
5    a = b - 3;    // subtract 0b(11) and 0b(011)
6  }
```

number will result in the bitpattern `111100` to be stored in `a`. Since `a` is an unsigned number, the final result is the decimal number `60`.

The effect of an assignment can also be obtained immediately by means of a cast operation, expressed by writing the desired type between brackets in front of an expression. For example, `(tc(1)) 1` has the value `-1`, while `(ns(3)) 15` has the value `8`.

4.1.3 Hardware Mapping of Expressions

For each expression involving signals and registers of a specified sign, and precision, there is an equivalent hardware circuit. This correspondence is quite easy to derive, once we know how each operator is mapped into hardware. We will discuss a list of common operations and indicate how they map into hardware logic. A summary of this discussion is shown in Table 4.1.

Arithmetic Operations. Addition (`+`), subtraction (`-`), and multiplication (`*`) are commonly used in datapath hardware design. Division (`/`) and modulo (`%`) on the other hand are uncommon because of the higher cost to implement these operations. Left-shift (`<<`) and right-shift (`>>`) are used to implement multiplication/division with powers of two. Constant-shifts are particularly advantageous for hardware implementation since they translate to simple hardware wiring.

Bitwise Operations. All of the bitwise operations, including AND (`&`), OR (`|`), XOR (`^`), and NOT (`~`) have a direct equivalent to logic gates. Bitwise operations are defined as bit-by-bit operations. The same precision rules as for all other operators apply: when the operands of a bitwise operation are of unequal length, they will be extended until they match. For example, if `w` is a word and `u` is a bit, then the expression

```
w & (tc(1))
```

will AND each bit of `w` with the bit in `u`.

Table 4.1 Operations in GEZEL and equivalent hardware implementation

Operation	Operator	Implementation	Precedence
Addition	+	Adder	4
Subtraction	–	Subtractor	4
Unary Minus	–	Subtractor	7
Multiplication	*	Multiplier	5
Right-shift	>> (variable)	Variable-shifter	0
Left-shift	<< (variable)	Variable-shifter	0
Constant Right-shift	>> const	Wiring	4
Constant Left-shift	<< const	Wiring	4
Lookup Table	A(n)	Random logic	10
AND	&	AND-gate	2
OR	|	OR-gate	2
XOR	^	XOR-gate	3
NOT	~	NOT-gate	8
Smaller-then	<	Subtractor	3
Bigger-then	>	Subtractor	3
Smaller-equal-then	<=	Subtractor	3
Bigger-equal-then	>=	Subtractor	3
Equal-to	==	Comparator	3
Not-equal-to	!=	Comparator	3
Bit Selection	[const]	Wiring	9
Bit-vector Selection	[const:const]	Wiring	9
Bit Concatenation	#	Wiring	4
Type cast	(type)	Wiring	6
Precedence ordering	()		11
Selection	? :	Multiplexer	1

const is a constant number

Comparison Operations. All of the comparison operations return a single unsigned bit (ns(1)). These operations use a subtractor to compare two numbers and then use the sign/overflow flags of the result to evaluate the result of the comparison. Exact comparison (== or !=) can be done by matching the bitpattern of each operand. In contrast to arithmetic operations, the comparison operations are implemented differently for signed and unsigned numbers. Indeed, the bit pattern 111 is smaller than the pattern 001 for signed numbers, but the same pattern 111 is bigger than the pattern 001 for unsigned numbers.

Bitvector Operations. Single bits, or a vector of several bits, can be extracted out of a word using the bit-selection operator.

```
reg a : ns(5);
reg b : ns(1);
reg c : ns(2);
always {
  b = a[3];   // if a = 10111, then b = 0
  c = a[4:2]; // if a = 10111, then a[4:2] = 101, so c = 01
}
```

The type of a bit-selection operation is unsigned, and just wide enough to hold all the bits. The bits in a bit vector are counted from right to left, with bit 0 holding the least significant bit. The opposite operation of bit-selection is bit-concatenation (#), which sticks bits together in a larger word.

```
reg a : ns(5);
reg b : ns(1);
reg c : ns(2);
always {
  a = c # b; // if b = 0, c = 11, then a = 00110
}
```

Selection. The ternary operator `a ? b : c` is the equivalent notation for a multiplexer. The result of the ternary operation will be `b` or `c` depending on the value of `a`. The wordlength of the result will be long enough to accommodate the largest word of either input of the multiplexer.

Indexed Storage. There is no array construction in GEZEL. However, it is possible to capture lookup tables. Lookup tables can be mapped into random logic using logic minimization.

```
  lookup T : ns(12) = {0x223, 0x112, 0x990};
  reg a : ns(12);
always {
  a = T(2);   // a = 0x990
}
```

Organization and Precedence. Finally, brackets may be used to group expressions and change the evaluation order. The default evaluation order is determined by the precedence of each operator. The precedence is shown as a number in Table 4.1, where a higher number corresponds to a higher precedence, meaning that operator will be evaluated before others.

Each expression involving registers, signals and the operations of Table 4.1, corresponds to a hardware datapath. A few examples are shown next. The first one, in Listing 4.7, shows Euclid's Greatest Common Divisor (GCD) algorithm. Two registers m and n are compared, and each clock cycle, the smallest one is subtracted from the largest one. Note that Listing 4.7 does not show how m and n are initialized.

Describing datapaths with expressions results in compact hardware descriptions. An excellent example are shift registers. Figure 4.2 illustrates a Linear Feedback Shift Register, which is a shift register with a feedback loop created by XORing bits within the shift register. The feedback pattern is specified by a polynomial,

Listing 4.7 Datapath to evaluate Greatest Common Divisor

```
1 reg m,n : ns(16);
2 always {
3   m = (m > n) ? (m - n) : m;
4   n = (n > n) ? (n - m) : m;
5 }
```

Fig. 4.2 Linear feedback
shift register for
$p(x) = x^4 + x^3 + 1$
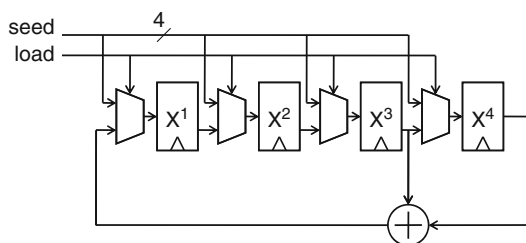
Listing 4.8 Linear Feedback Shift Register

```
1  reg shft      : ns(4);
2  sig shft_new  : ns(4);
3  sig load      : ns(1);
4  sig seed      : ns(4);
5  always {
6    shft_new  = (shft << 1) | (shft[2] ^ shft[3]);
7    shft      = load ? seed : shft_new;
8  }
```

and the polynomial used for Fig. 4.2 is $p(x) = x^4 + x^3 + 1$. LFSRs are used for pseudorandom sequence generation. If a so-called maximum-length polynomial is chosen, the resulting sequence of pseudorandom bits has a period of $2^n - 1$, where n is the number of bits in the shift register. Thus, an LFSR is able to create a long nonrepeating sequence of pseudorandom bits with a minimal amount of hardware.

The shift register used to implement the LFSR must always contain at least one nonzero bit. It is easy to see in Fig. 4.2 that an all-zero pattern in the shift register will only produce itself. Therefore, an LFSR must be initialized with a nonzero seed value. The seed value is programmed using a multiplexer in front of each register.

Although the structure of Fig. 4.2 is quite complex to draw, it remains very compact when written using word-level expressions. This is shown in Listing 4.8. Line 6 of the code represents the shift-and-feedback operation. Line 7 of the code represents the loading of the seed value into the LFSR register.

In summary, using two variable types (signals and registers), it is possible to describe synchronous hardware by means of expressions on those signals and registers. Remember that the order in which expressions are written is irrelevant: they will all execute within a single clock cycle. In the next section, we will group expressions into modules and define input/output ports on those modules.

4.2 Hardware Modules

A hardware module defines a level of hierarchy for a hardware netlist. In order to communicate across the levels of hierarchy, hardware modules define ports. Figure 4.3 shows the 3-bit counter, encapsulated as a module. There is a single input port, `clr`, which allows to synchronously clear the register. There is also a 3-bit output port `c` that holds the current count value. The equivalent description in

Fig. 4.3 3-bit counter
module
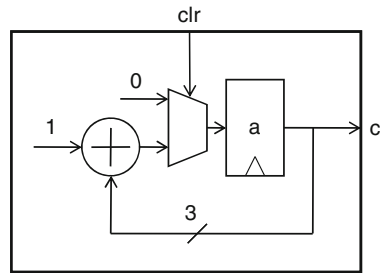
Listing 4.9 3-bit counter module with reset

```
1  dp count(in  clr : ns(1);
2            out c  : ns(3)) {
3    reg a : ns(3);
4    always {
5      a = clr ? 0 : a + 1;
6      c = a;
7    }
8  }
```

GEZEL language of this structure is shown in Listing 4.9. The `always` block is
included in a `dp` (datapath), which defines a list of `in` and `out` ports. There can
be as many input and output ports as needed, and they can be created in any or-
der. Registers and signals are local to a single module and invisible outside of the
module boundary. Input ports and output ports are equivalent to wires and there-
fore behave identical to signals. Input ports and output ports are subject to similar
requirements as signals: it is not allowed to assign an output port more than once
during a clock cycle, and each output must be assigned at least once during each
clock cycle. We will further investigate this while discussing the formal properties
of the FSMD model in Sect. 4.6.

After hardware is encapsulated inside of a module, the module itself can be used
as a component in another hardware design. This principle is called *structural hi-
erarchy*. As an example, Listing 4.10 shows how the 3-bit counter is included in a
testbench structure that clears the counter as soon as it reaches three. The module
is included by the `use` keyword, which also shows how ports should be connected
to local signals and registers. The equivalent hardware structure of Listing 4.10 is
shown in Fig. 4.4.

The `countrun` module in Listing 4.10 has no inputs nor outputs. Such mod-
ules have no practical value for implementation, but they are useful for testbenches.
The listing also illustrates the use of a `$display` statement, which is a simulator
directive to print the value of a signal or register. Several other directives will be
discussed in Sect. 4.5.

Once a module has been included inside of another one by means of the `use`
statement, it cannot be included again: each module can be used only once. How-
ever, it is easy to create a duplicate of an existing module by means of a *cloning*
statement. Listing 4.11 shows how to create 3-bit counters, `count0`, `count1` and
`count2`.

Listing 4.10 Encapsulated counter module

```
1  dp countrun {
2    sig clearit : ns(1);
3    sig cnt     : ns(3);
4    use count(clearit, cnt);
5    always {
6      clearit = cnt[0] & cnt[1];
7      $display("cnt = ", cnt);
8    }
9  }
```

Fig. 4.4 Hardware
equivalent of Listing 4.10
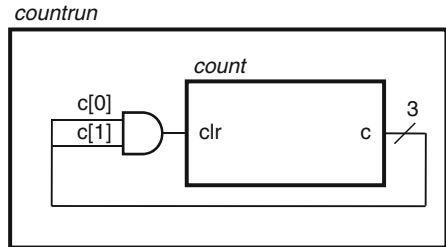
Listing 4.11 Cloning of modules

```
1  dp count0(in  clr : ns(1);
2            out c   : ns(3)) {
3    reg a : ns(3);
4    always {
5      a = clr ? 0 : a + 1;
6      c = a;
7    }
8  }
9  dp count1 : count0
10 dp count2 : count0
```

4.3 Finite State Machines

We will next describe a control model for hardware circuits. As discussed before, the expressions that are part of an `always` block are evaluated at each clock cycle, and it is not possible to conditionally evaluate an expression. Even the selection operator (`c ? expr1 : expr2`) will evaluate the true-part as well as the false-part regardless of the condition value `c`. Assume that `expr1` and `expr2` would contain an expensive operator, then we would need two copies of that operator to implement `c ? expr1 : expr2`.

A control model, on the other hand, allows us to indicate what expressions should execute during each clock cycle. Very simple control models will only select the sequence of expressions to execute over multiple clock cycles. More complex control models will also allow decision making. Advanced control models also consider issues such as exceptions, recursion, out-of-order execution, and so forth. In this

section, we describe a common control model for hardware description, called Finite State Machine (FSM). An FSM can be used to describe sequencing and decision making. In the next section, we will combine the FSM with expressions in a datapath.

A FSM is a sequential digital machine which is characterized by

- A set of states
- A set of inputs and outputs
- A state transition function
- An output function

An FSM has a current state, equal to an element from the set of states. Each clock cycle, the state transition function selects the next value for the current state, and the output function selects the value on the output of the FSM. The state transition function and the output function are commonly described in terms of a graph. In that case, the set of states becomes the set of nodes of the graph, and the state transitions become edges in the graph.

The operation of a FSM is best understood by means of an example. Suppose we need to observe a (possibly infinite) sequence of bits, one at a time. We need to determine at what point the sequence contains the pattern '110'. This problem is perfectly suited for an FSM. We can distinguish three relevant states for the FSM, by realizing that sequential observation of the input bits transforms this pattern recognition problem into an incremental process.

1. State S0: We have not recognized any useful pattern.
2. State S1: We have recognized the pattern '1'.
3. State S2: We have recognized the pattern '11'.

When we consider each state and each possible input bit, we can derive all state transitions, and thus derive the state transition function. The output function can be implemented by defining a successful recognition as an input bit of '0' when the FSM is in state S2. This leads to the state transition graph (or state transition diagram) shown in Fig. 4.5. The notation used for Fig. 4.5 is that of a *Mealy* FSM. The output of a Mealy FSM is defined by the present state, as well as the input.

There is a different formulation of a FSM known as a *Moore* state machine. The output of a Moore state machine is only dependent on the current state, and not on the current input. Both forms, Mealy and Moore, are equivalent formulations of FSM. A Mealy machine can be converted into an equivalent Moore machine using a simple conversion procedure.

To convert the Mealy form into a Moore form, make a state transition table for the Mealy FSM. This table contains the next-state of each state transition combined with the relevant output. For example, in state S1 of Fig. 4.5, the input '1' leads to state S2 with output 0. Annotate this in the Table as: S2, 0. The entire diagram can be translated this way, leading to Table 4.2.

Using the conversion table, an equivalent Moore FSM can be constructed. There is one Moore state for each unique *(next-state, output)* pattern. From Table 4.2, we can find 4 Moore states: (S0, 0), (S0, 1), (S1, 0), and (S2, 0). To find the Moore FSM
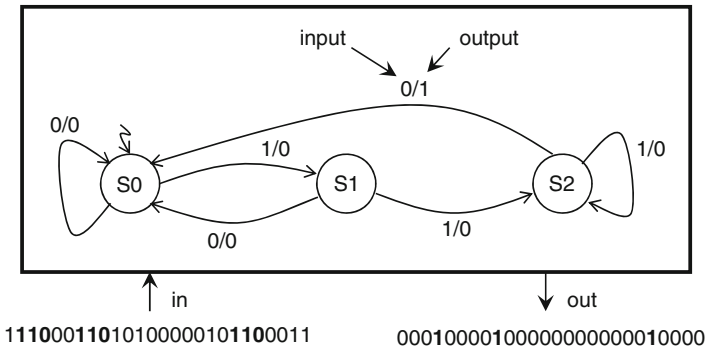
Fig. 4.5 Mealy FSM of a recognizer for the pattern '110'

Table 4.2 Conversion of Mealy to Moore FSM for Fig. 4.5

Current-State	Input = 0	Input = 1
S0	S0, 0	S1, 0
S1	S0, 0	S2, 0
S2	S0, 1	S2, 0

Table 4.3 Resulting moore state transition table

Current-State	Input = 0	Input = 1	Output
SA = S0, 0	SA	SC	0
SB = S0, 1	SA	SC	1
SC = S1, 0	SA	SD	0
SD = S2, 0	SB	SD	0

state transitions, we replicate the corresponding Mealy transitions for each Moore state. There may be multiple Moore transitions for a single Mealy transition. For example, Mealy state S0 is replicated into Moore states (S0, 0) and (S0, 1). Thus, each of the state transitions out of S0 will be replicated two times. The resulting Moore state transition table is shown in Table 4.3, while the resulting Moore FSM graph is drawn in Fig. 4.6. A small ambiguity was removed from Fig. 4.6 by making state SA = (S0, 0) the initial state of the Moore machine. That is, the Mealy machine does not specify the initial value of the output. Since the Moore machine ties the output to the current state, an initial output value must be assumed as well.

In summary, a FSM is a common control model for hardware design. We can use it to model conditional execution of expressions. In that case, we will use the outputs of the FSM to control the execution of expressions. Similarly, we will use the inputs to feed runtime conditions into the FSM so that conditional sequencing can be implemented. In the next section, we will use a modified form of a FSM, called FSMD, which is the combination of an FSM and a datapath (modeled using expressions).
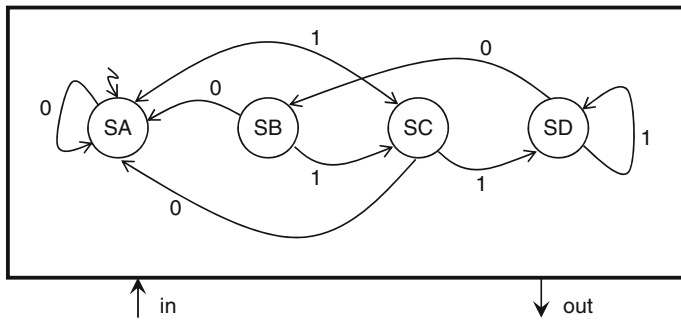
Fig. 4.6 Moore FSM of a recognizer for the pattern '110'

4.4 Finite State Machines with Datapath

A FSM with Datapath combines a hardware control model (an FSM) with a datapath. The datapath is described as cycle-based bit-parallel hardware using expressions, as discussed in Sect. 4.1. However, in contrast to datapaths using only `always` blocks, FSMD datapaths define also one or more *instructions*: conditionally executed 'always' blocks.

4.4.1 Modeling

Listing 4.12 shows an example of a datapath with an `always` block and three instructions: `inc`, `dec`, and `clr`. The meaning of the `always` block is the same as before: it contains expressions that will execute every clock cycle. In contrast to an `always` block, instructions will only execute when told to do so by a controller. Thus, the three instructions of the datapath in Listing 4.12 can increment, decrement, or clear register `a`, depending on what the controller tells the datapath to do.

An instruction has the same meaning for expressions as an `always` block: it specifies operations in combinational logic and describes register-updates. An instruction has a name such as `inc`, `dec`, and `clr` so it can be referenced by a controller. The keyword `sfg` precedes the instruction name, and this keyword is an acronym for *signal flow graph*: An instruction is a snippet of a dataflow graph of a hardware description during 1 clock cycle of processing.

A datapath with instructions needs a controller to select which instruction should execute in each clock cycle. The FSMD model uses an FSM for this. Listing 4.13 shows a FSM controller for the datapath of Listing 4.12. In this case, the controller will steer the datapath such that it first counts up from 0 to 3, and next counts down from 3 to 0. The FSM is a textual format for a state transition diagram. Going through the listing, we can identify the following features.

Listing 4.12 Datapath for an up-down counter with three instructions

```
1  dp updown(out c   : ns(3)) {
2    reg a : ns(3);
3    always { c = a; }
4    sfg inc { a = a + 1; } // instruction inc
5    sfg dec { a = a - 1; } // instruction dec
6    sfg clr { a = 0;     } // instruction clr
7  }
```

Listing 4.13 Controller for the up-down counter

```
1  fsm ctl_updown(updown) {
2    initial s0;
3    state s1, s2;
4    @s0 (clr) -> s1;
5    @s1 if (a < 3) then (inc) -> s1;
6                   else (dec) -> s2;
7    @s2 if (a > 0) then (dec) -> s2;
8                   else (inc) -> s1;
9  }
```

- Line 1: The `fsm` keyword defines a FSM with name `ctl_updown` and tied to the datapath `updown`.
- Line 2–3: The FSM contains three states. One state, called `s0`, will be the initial state. Two other states, called `s1` and `s2`, are regular states. The current state of the FSM will always be one of `s0`, `s1`, or `s2`.
- Line 4: When the current state is `s0`, the FSM will unconditionally transition to state `s1`. State transitions will be taken at each clock edge. At the same time, the datapath will receive the instruction `clr` from the FSM, which will cause the register `a` to be cleared (See Listing 4.12).
- Line 5–6: When the current state of the FSM equals `s1`, a conditional state transition will be taken. Depending on the value of the condition, the FSM will transition either to state `s1` (Line 5), or else to state `s2` (Line 6), and the datapath will receive either the instruction `inc` or else `dec`. This will either increment or else decrement register `a`. The state transition condition is given by the expression `a < 3`. Thus, the FSM will remain in state `s1` as long as register `a` is below three. When `a` equals three, the FSM will transition to `s2` and a decrementing sequence is initiated.
- Line 7–8: When the current state of the FSM equals `s2`, a conditional state transition to either `s1` or else `s2` will be taken. These two state transitions will issue a decrement instruction to the datapath as long as register `a` is above zero. When the register equals zero, the controller will transition to `s1` and restart the incrementing sequence.

Thus, the FSM controller determines the schedule of instructions on the datapath. There are many possible schedules, and the example shown in Listing 4.9 is one of them. However, since an FSM is not a programmable construct, the implementation of the FSM will fix the schedule of instructions provided to the datapath.

Table 4.4 Behavior of the FSMD in Listing 4.13

Cycle	FSM curr/next	DP instr	DP expr	a curr/next
0	s0/s1	clr	a = 0;	0/0
1	s1/s1	inc	a = a + 1;	0/1
2	s1/s1	inc	a = a + 1;	1/2
3	s1/s1	inc	a = a + 1;	2/3
4	s1/s2	dec	a = a − 1;	3/2
5	s2/s2	dec	a = a − 1;	2/1
6	s2/s2	dec	a = a − 1;	1/0
7	s2/s1	inc	a = a + 1;	0/1
9	s1/s1	inc	a = a + 1;	1/2
9	s1/s1	inc	a = a + 1;	2/3

Table 4.4 illustrates the first 10 clock cycles of operation for this FSMD. Each row shows the clock cycle, the current and next FSM state, the datapath instruction selected by the controller, the datapath expression, and the current and next value of the register a.

In the up/down counter example, each datapath instruction contains a single expression, and the FSM selects a single datapath instruction for execution during each clock cycle. This is not a strict requirement. An instruction may contain as many expressions as needed, and the FSM controller may select multiple instructions for execution during any clock cycle. You can think of a group of scheduled instructions and the always block as a single, large always block that is active for a single clock cycle. Of course, not all combinations will work in each case. For example, in the datapath shown in Listing 4.12, the instructions clr, inc, and dec are all exclusive since all of them modify register a. The set of expressions that execute during a given clock cycle (as a result of the always block and the scheduled instructions) have to be conform to the same rules as if there were only a single always block. We will define these rules precisely in Sect. 4.6.

Listing 4.14 shows the implementation of Euclid's algorithm as an FSMD. In this case, several datapath instructions contain multiple expressions. In addition, the controller runs multiple datapath instructions during one state transition (line 21). The body of the reduce instruction was presented earlier as the computational core of GCD (Listing 4.14). The additional functionality provided by the controller is the initialization of the registers m and n, and the detection of the algorithm completion.

We can now provide a more precise description of an FSMD and its associated execution model. An FSMD consists of two stacked FSMs, as illustrated in Fig. 4.7. The top FSM contains the controller and is specified using a state transition diagram. The bottom FSM contains the datapath and is specified using expressions. The top FSM send instructions to the bottom FSM and receives status information in return. Both FSM operate synchronously and are connected to a single clock.

Each clock cycle, the two FSM go through the following activities.

1. Just after the clock edge, the state variables of both FSM are updated. For the controller, this means that a state transition is completed and the state register holds the new current-state value. For the datapath, this means that the register variables are updated as a result of assigning expressions to them.

Listing 4.14 Euclid's GCD as an FSMD

```
1  dp euclid(in  m_in, n_in : ns(16);
2            out gcd        : ns(16)) {
3    reg m, n               : ns(16);
4    reg done               : ns(1);
5    sfg init    { m    = m_in;
6                  n    = n_in;
7                  done = 0;
8                  gcd  = 0; }
9    sfg reduce  { m    = (m >= n) ? m - n : m;
10                 n    = (n >  m) ? n - m : n;  }
11   sfg outidle { gcd  = 0;
12                 done = ((m == 0) | (n == 0)); }
13   sfg complete{ gcd  = ((m > n) ? m : n);
14                 $display("gcd = ", gcd); }
15 }
16 fsm euclid_ctl(euclid) {
17   initial s0;
18   state s1, s2;
19   @s0 (init) -> s1;
20   @s1 if (done) then (complete)          -> s2;
21                 else (reduce, outidle) -> s1;
22   @s2 (outidle) -> s2;
23 }
```

Fig. 4.7 An FSMD consists of two stacked FSMs
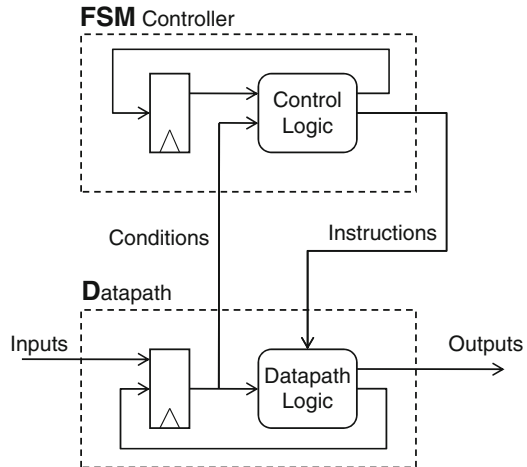
2. The control FSM combines the control-state and datapath-state to evaluate the new next-state for the control FSM. At the same time, it will also select what instructions should be executed by the datapath.
3. The datapath FSM will evaluate the next-state for the state variables in the datapath, using the updated datapath state as well as the instructions received from the control FSM.

4. Just before the next clock edge, both the control FSM and the datapath FSM have evaluated and prepared the next-state value for the control state as well as the datapath state.

What makes a controller FSM different from a datapath FSM? Indeed, as illustrated in Fig. 4.7, both the datapath and the controller are sequential digital machines. Yet, from a designers' viewpoint, the creation of datapath logic and control logic is very different.

- Control logic tends to have an irregular structure. Datapath logic tends to have a regular structure (especially once you work with multibit words).
- Control logic is easy to describe using a finite state transition diagram and hard to describe using expressions. Datapath logic is just the opposite: easy to describe using expressions but hard to capture in state transition diagrams.
- The registers (state) in a controller have a different purpose than those in the datapath. Datapath registers contain algorithmic state (like m and n in Listing 4.10). Control registers contain sequencing state.

In conclusion, we have discussed FSMD, a standard model for digital hardware design. Our discussion covered the modeling syntax as well as the operation. FSMDs are useful because they capture control flow as well as data flow in hardware. Recall that C programs are also a combination of control flow and dataflow (Chap. 3). This results in an implied connection between hardware FSMD models and C programs.

4.4.2 An FSMD is Not Unique

Figure 4.7 demonstrated how an FSMD actually consists of two stacked FSM. This has an interesting implication. From an implementation perspective, the partitioning between control logic and datapath logic is not unique. To illustrate this on Fig. 4.7, assume that we would merge the control logic and datapath logic into a single logic module, and assume that we would combine the registers in the controller with those in the datapath. The resulting design would still look as an FSM. The implementation thus makes no clear distinction between the Datapath part and FSM part.

In the previous subsection, we showed that the modeling of the FSM controller and the Datapath is very different: using state transition graphs and expressions, respectively. Since the partitioning between a controller and the datapath is not unique, this means that we should be able to write up an FSM using expressions. This is illustrated in Listings 4.15 and 4.16. The first listing shows the FSM state transition diagram for the up-down counter from Sect. 4.4.1. The second listing shows an equivalent description using expressions. The key difference between both is that in Listing 4.16, we have chosen the encoding for controller states. Also note that the 'controller' function in Listing 4.16 is captured in an always block.

Since we can model an FSM with expressions, we can also merge it with the datapath controlled by this FSM. The resulting design for the up-down counter is

Listing 4.15 FSM controller for updown counter

```
1  fsm ctl_updown(updown) {
2    initial s0;
3    state s1, s2;
4    @s0 (clr) -> s1;
5    @s1 if (a < 3) then (inc) -> s1;
6                   else (dec) -> s2;
7    @s2 if (a > 0) then (dec) -> s2;
8                   else (inc) -> s1;
9  }
```

Listing 4.16 FSM controller for updown counter using expressions

```
1  dp updown_ctl(in  a_sm_3, a_gt_0 : ns(1);
2                out instruction     : ns(2)) {
3    reg state_reg : ns(2);
4    // state encoding: s0 = 0, s1 = 1, s2 = 2
5    // instruction encoding: clr = 0, inc = 1, dec = 2
6    always {
7      state_reg   = (state_reg == 0) ? 1 :
8                    ((state_reg == 1) &  a_sm_3) ? 1 :
9                    ((state_reg == 1) & ~a_sm_3) ? 2 :
10                   ((state_reg == 2) &  a_gt_0) ? 2 : 1;
11     instruction = (state_reg == 0) ? 0 :
12                   ((state_reg == 1) &  a_sm_3) ? 1 :
13                   ((state_reg == 1) & ~a_sm_3) ? 2 :
14                   ((state_reg == 2) &  a_gt_0) ? 2 : 1;
15   }
16 }
```

shown in Listing 4.17. This description shows that an FSMD can be captured as a simple datapath. When choosing the writing style for an FSMD (using either a separate FSM and datapath description, or else as a single datapath), you would make the following trade-off considerations.

- When capturing the FSMD in a single datapath, the expressions in the datapath include scheduling as well as data processing. On the other hand, in an FSMD with separate FSM and datapath description, the datapath expressions represent only data processing. Often, data-processing expressions are directly related to the original specification. Compare, for example, Listings 4.13 and 4.17 while considering that the purpose of this design is an up-down counter. Obviously, Listing 4.13 is easier to understand.
- When the datapath expressions include scheduling as well as data processing, they become harder to reuse in a different schedule. Using an FSMD with separate FSM and datapath description on the other hand will allow changes to the scheduling (the FSM) while reusing most of the datapath description.
- When capturing the FSMD in a single datapath, the state assignment is chosen by the designer and may be optimized for specific applications. In an FSMD with separate FSM and datapath description, the state assignment is left to the logic synthesis tool.

Listing 4.17 Updown counter using expressions

```
1   dp updown_ctl(out c : ns(3)) {
2     reg a        : ns(3);
3     reg state_reg   : ns(2);
4     sig a_sm_3 : ns(1);
5     sig a_gt_0 : ns(1);
6     // state encoding: s0 = 0, s1 = 1, s2 = 2
7     always {
8       state_reg = (state_reg == 0) ? 1 :
9                     ((state_reg == 1) &  a_sm_3) ? 1 :
10                    ((state_reg == 1) & ~a_sm_3) ? 2 :
11                    ((state_reg == 2) &  a_gt_0) ? 2 : 1;
12      a_sm_3 = (a < 3);
13      a_gt_0 = (a > 0);
14            a = (state_reg == 0) ? 0 :
15                  ((state_reg == 1) &  a_sm_3) ? a + 1 :
16                  ((state_reg == 1) & ~a_sm_3) ? a - 1 :
17                  ((state_reg == 2) &  a_gt_0) ? a + 1 : a - 1;
18      c = a;
19    }
20  }
```

- An FSMD captured as a single datapath is good for designs with simple or no control scheduling, such as designs found in very high-throughput applications. An FSMD with separate FSM and datapath description is good for more complicated, structured designs.

In the above examples, we showed how an FSM can be modeled using datapath expressions, and how this allowed to capture an entire FSMD in a single datapath. The opposite case (modeling a datapath as a state transition diagram) is very uncommon. Problem 4.9 investigates some of the reasons for this.

4.4.3 Implementation

How to determine the hardware implementation of an FSMD? The basic rules for mapping expressions on registers/signals into synchronous digital hardware are the same as before. However, there is also an important difference. When an expression occurs inside of a datapath instruction, it should execute only when that instruction is selected by the controller. Thus, the final datapath structure will depend on the schedule of instructions selected by the FSMD controller.

To clarify this point, consider again the up-down counter in Listing 4.18. From the FSM model, it is easy to see that clr, inc, and dec will always execute in different clock cycles. Therefore, the datapath operators used to implement each of clr, inc, and dec can be shared. A possible implementation of the FSMD is shown in Fig. 4.8. The datapath includes an adder/subtractor and a multiplexer, which implement the instruction-set (clr, inc, dec). A local decoder is needed to convert the encoding used for these instructions into local control signals for the

Listing 4.18 Datapath for an up-down counter with three instructions

```
1  dp updown(out c    : ns(3)) {
2    reg a : ns(3);
3    always  { c = a; }
4    sfg inc { a = a + 1; }
5    sfg dec { a = a - 1; }
6    sfg clr { a = 0;     }
7  }
8  fsm ctl_updown(updown) {
9    initial s0;
10   state s1, s2;
11   @s0 (clr) -> s1;
12   @s1 if (a < 3) then (inc) -> s1;
13                  else (dec) -> s2;
14   @s2 if (a > 0) then (dec) -> s2;
15                  else (inc) -> s1;
16 }
```

Fig. 4.8 Implementation of the up-down counter FSMD
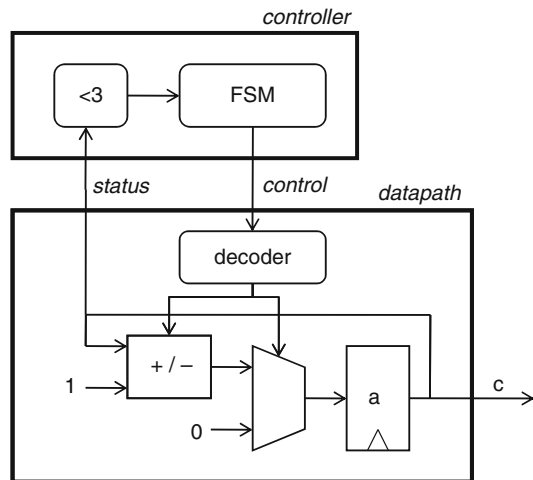
multiplexer and the adder subtractor. The datapath is attached to a controller, which is implemented with an FSM and a local decoder for datapath status information. In this case, the value in register a is datapath status.

The implementation shown in Fig. 4.8 can be completely refined using logic synthesis tools: the contents of the decoder and the FSM for example will be defined using low-level logic synthesis. Through the FSMD description, a designer can still influence the following aspects of the synthesis process.

- The designer determines the amount of work done in a single clock cycle. This is simply the set of all datapath instructions which will execute concurrently in a single clock cycle. Hence, in order to obtain the best possible sharing, a designer must distribute similar operations over multiple clock cycles. For example, if there are 16 multiplications to perform with a clock cycle budget of 4 clock cycles, then an implementation with 4 multiplies each clock cycle

will most likely be smaller than one which performs 16 multiplies in the first clock cycle and nothing in the next three cycles.

- The designer also influences indirectly the critical path of the design by the complexity of the expressions given in the datapath instructions. In synchronous hardware design, the minimum clock period of a design is bounded by the critical path. Hence, the critical path of the FSMD datapath should be kept small. Obviously, small and short expressions will result in smaller and faster logic in the datapath. If the expressions used to capture the datapath become too complex, the resulting design may be too slow for the intended system clock period.

4.5 Simulation and RTL Synthesis of FSMD

In this section we will illustrate the use of the GEZEL simulation and code generation tools for standalone FSMD models. These tools can be downloaded online.

4.5.1 Simulation

We will illustrate the simulation tools on the GCD module discussed earlier in Sect. 4.4.1. Listing 4.19 shows the gcd module as well as a simulation test-bench. In this case, the testbench test_euclid drives the inputs m and n to two constant values. The system module shown at the bottom of the listing tells the simulator what module should be considered the top-level module in the simulation. Only modules which are listed in a system block (as well as the modules included by the top module) will take part in the simulation.

The simulation command is fdlsim. The arguments include the name of the file that holds the FSMD descriptions and the number of clock cycles to simulate. For example, assume Listing 4.19 would be included in a file euclid.fdl, then the following command would simulate 20 clock cycles.

```
> fdlsim euclid.fdl 20
 m = 0/2322 n = 0/654
 m = 2322/1668 n = 654/654
 m = 1668/1014 n = 654/654
 m = 1014/360 n = 654/654
 m = 360/360 n = 654/294
 m = 360/66 n = 294/294
 m = 66/66 n = 294/228
 m = 66/66 n = 228/162
 m = 66/66 n = 162/96
 m = 66/66 n = 96/30
 m = 66/36 n = 30/30
 m = 36/6 n = 30/30
 m = 6/6 n = 30/24
 m = 6/6 n = 24/18
```

Listing 4.19 Euclid's GCD as an FSMD

```
1  dp euclid(in   m_in, n_in : ns(16);
2             out gcd        : ns(16)) {
3    reg m, n                 : ns(16);
4    reg done                 : ns(1);
5    always       { $display($dec, " m = ", m, " n = ", n); }
6    sfg init     { m    = m_in;
7                   n    = n_in;
8                   done = 0;
9                   gcd  = 0; }
10   sfg reduce   { m    = (m >= n) ? m - n : m;
11                  n    = (n >  m) ? n - m : n;  }
12   sfg outidle  { gcd  = 0;
13                  done = ((m == 0) | (n == 0)); }
14   sfg complete { gcd  = ((m > n) ? m : n);
15                  $display("cycle = ", $cycle, " gcd = ", gcd); }
16 }
17 fsm euclid_ctl(euclid) {
18   initial s0;
19   state s1, s2;
20   @s0 (init) -> s1;
21   @s1 if (done) then (complete)          -> s2;
22                 else (reduce, outidle) -> s1;
23   @s2 (outidle) -> s2;
24 }
25
26 dp test_euclid {
27   sig m, n, gcd : ns(16);
28   use euclid(m, n, gcd);
29   always {
30     m = 2322;
31     n = 654;
32   }
33 }
34
35 system S {
36   test_euclid;
37 }
```

```
 m = 6/6 n = 18/12
 m = 6/6 n = 12/6
 m = 6/0 n = 6/6
 m = 0/0 n = 6/6
 m = 0/0 n = 6/6
cycle = 18 gcd = 6
 m = 0/0 n = 6/6
```

Due to the $display statements in the always block and the complete
instruction in Listing 4.19, the simulation generates intermediate output. Note the
particular way of printing register variables, using a current/next value representa-
tion. Clock cycles are counted by the simulator starting from zero, so the simulation
executes the complete instruction in clock cycle 19. There are several other
simulation directives similar to $display, as listed in Table 4.5.

Table 4.5 Simulation directives

Directive	Use
`$display(arg, ..)`	Used inside sfg. Prints strings and expressions.
`$cycle`	Used as argument of `$display`. Returns current clock cycle.
`$toggle`	Used as argument of `$display`. Returns overall toggle count.
`$sfg`	Used as argument of `$display`. Returns name of current `sfg`.
`$dp`	Used as argument of `$display`. Returns name of current `dp`.
`$hex, $dec, $bin`	Used as argument of `$display`.
	Changes output format for values to hex, decimal or binary.
`$finish`	Used inside sfg. Terminates the simulation immediately.
`$trace(expr,filename)`	Used inside a dp. Records values of register/signal in file.
`$option "string"`	Used at top of file. Enables additional profiling and VCD tracing.

4.5.2 Code Generation and Synthesis

Once an FSMD has been successfully simulated, it can be converted into synthesizable format. The design in Listing 4.19 can be converted into RTL-VHDL code using the following command:

```
fdlvhd euclid.fdl
```

The tool generates a single VHDL file for each datapath in the `euclid.fdl` file.

```
>fdlvhd euclid.fdl
Pre-processing System ...
Output VHDL source ...
---------------------------
Generate file: euclid.vhd
Generate file: test_euclid.vhd
Generate file: system.vhd
Generate file: std_logic_arithext.vhd
```

4.6 Proper FSMD

A *proper FSMD* is one which has deterministic behavior. In general, a model with deterministic behavior is one which will always show the same response given the same initial state and the same input stimuli. Deterministic behavior is a desirable feature for many applications and definitely for software programs. For hardware/-software codesign applications, it makes sense to enforce a uniform approach toward determinacy across the boundaries of hardware and software.

For a hardware FSMD implementation, deterministic behavior means that the hardware is free of race conditions. Without determinacy, a hardware model may end up in an unknown state (often represented using an 'X' in multivalued logic

hardware simulation) . We will avoid this situation by enforcing modeling conditions to the FSMD, leading to a proper FSMD.

A proper FSMD is obtained by enforcing four properties in the FSMD model. These properties are easy to check, both by the FSMD developer as well as by the simulation tools. The four properties are the following:

1. Neither registers nor signals can be assigned more than once during a clock cycle.
2. No circular definition exists between signals (wires).
3. If a signal is used as an operand of an expression, it must have a known value in the same clock cycle.
4. All datapath outputs must be defined (assigned) during all clock cycles.

The first rule is obvious and ensures that there will be at most a single assignment per register/signal and per clock cycle. Recall from our earlier discussion that in a synchronous hardware model, all expressions are evaluated simultaneously according to the data dependencies of the expressions. If we allow multiple assignments per register/signal, the resulting value in the register or signal will become ambiguous.

The second rule ensures that any signal will carry a single, stable value during a clock cycle. Indeed, a circular definition between signals (e.g., as shown in Listing 4.4) may result in more than a single valid value. For example, circular definitions would occur when you try to model flip-flops with combinational logic (state). A proper FSMD model enforces you to use `reg` for all state variables. Another case where you would end up with circular definition between signals is when you create free-running ring-oscillators. In a cycle-based hardware description language, all events happen at the pace of the global clock, and free-running oscillators cannot be modeled.

The third rule ensures that no signal can be used as an operand when the signal value would be undefined. Indeed, when an undefined signal is used as an operand in an expression, the result of the expression may become unknown. Such unknown values propagate in the model and introduce ambiguity on the outputs.

The fourth rule deals with hierarchy and makes sure that rule 2 and 3 will hold even across the boundaries of datapaths. As we discussed earlier, datapath inputs and outputs have the same semantics as wires. The value of a datapath input will be defined by the datapath output connected to it. Rule 4 says that this datapath output will always have a known and stable value. Rule 4 is stricter than required. For example, if we don't read a datapath input during a certain clock cycle, the corresponding connected datapath output could remain undefined without causing trouble. However, requiring all outputs to be always defined is much easier to remember for the FSMD designer.

All of the above rules are enforced by the simulation tools for FSMD, either at runtime (through an error message), or else when the model is parsed. The resulting hardware created by these modeling rules is determinate and race-free.

4.7 Language Mapping for FSMD by Example

Even though we will be using the GEZEL language throughout this book for modeling of FSMD, all concepts covered so far are equally valid in other modeling languages including Verilog, VHDL, or SystemC. We use GEZEL because of the following reasons:

- It is easier to set up cosimulation experiments in GEZEL. We will cover different types of hardware–software interfaces, and all of these are directly covered using GEZEL primitives.
- More traditional modeling languages include additional concepts (such as multivalued logic and event-driven simulation), which, even though important by themselves, are less relevant in the context of an introduction to hardware–software codesign.
- GEZEL designs can be expanded into Verilog, VHDL, or SystemC, as will be illustrated in this section. In fact, the implementation path of GEZEL works by converting these GEZEL designs into VHDL, and then using hardware synthesis on the resulting design.

The example we will discuss is the *binary* GCD algorithm, a lightly optimized version of the classic GCD that makes use of the odd-even parity of the GCD operands. The implementation of the design is illustrated in Listings 4.20, 4.21, 4.22, 4.23 for the case of GEZEL, Verilog, and SystemC, respectively.

4.7.1 GCD in GEZEL

Listing 4.20 Binary GCD in GEZEL

```
1  dp euclid(in  m_in, n_in : ns(16);
2            out gcd        : ns(16)) {
3    reg m, n               : ns(16);
4    reg done               : ns(1);
5    reg factor             : ns(16);
6
7    sfg init    { m = m_in; n = n_in; factor = 0; done = 0; gcd = 0;
8                  $display("cycle=", $cycle, " m=", m_in, " n=",
9                  n_in); }
10   sfg shiftm  { m = m >> 1; }
11   sfg shiftn  { n = n >> 1; }
12   sfg reduce  { m = (m >= n) ? m - n : m;
13                 n = (n >  m) ? n - m : n; }
14   sfg shiftf  { factor = factor + 1; }
15   sfg outidle { gcd = 0; done = ((m == 0) | (n == 0)); }
16   sfg complete{ gcd = ((m > n) ? m : n) << factor;
17                 $display("cycle=", $cycle, " gcd=", gcd); }
18 }
19 fsm euclid_ctl(euclid) {
20   initial s0;
21   state s1, s2;
```

```
22    @s0 (init) -> s1;
23    @s1 if (done)                         then (complete)              -> s2;
24        else if ( m[0] &  n[0]) then (reduce, outidle)     -> s1;
25        else if ( m[0] & ~n[0]) then (shiftn, outidle)    -> s1;
26        else if (~m[0] &  n[0]) then (shiftm, outidle)    -> s1;
27                                 else (shiftn, shiftm,
28                                       shiftf, outidle)    -> s1;
29    @s2 (outidle) -> s2;
30 }
```

4.7.2   GCD in Verilog

Listing 4.21 Binary GCD in Verilog
```
1  module euclid(m_in, n_in, gcd, clk, rst);
2    input  [15:0] m_in;
3    input  [15:0] n_in;
4    output [15:0] gcd;
5    reg    [15:0] gcd;
6    input  clk;
7    input  rst;
8
9    reg [15:0] m, m_next;
10   reg [15:0] n, n_next;
11   reg done, done_next;
12   reg [15:0] factor, factor_next;
13   reg [1:0] state, state_next;
14
15   parameter s0 = 2'd0, s1 = 2'd1, s2 = 2'd2;
16
17   always @(posedge clk)
18    if (rst) begin
19      n      <= 16'd0;
20      m      <= 16'd0;
21      done   <= 1'd0;
22      factor <= 16'd0;
23      state  <= s0;
24    end else begin
25      n      <= n_next;
26      m      <= m_next;
27      done   <= done_next;
28      factor <= factor_next;
29      state  <= state_next;
30    end
31
32   always @(*) begin
33     n_next      <= n;        // default reg assignment
34     m_next      <= m;        // default reg assignment
35     done_next   <= done;     // default reg assignment
36     factor_next <= factor;   // default reg assignment
37     gcd         <= 16'd0;    // default output assignment
38
39     case (state)
40
```

```
41          s0: begin
42              m_next       <= m_in;
43              n_next       <= n_in;
44              factor_next <= 16'd0;
45              done_next    <= 1'd0;
46              gcd          <= 16'd0;
47              state_next <= s1;
48              end
49
50          s1: if (done) begin
51                  gcd <= ((m > n) ? m : n) << factor;
52                  state_next <= s2;
53              end else if (m[0] & n[0]) begin
54                  m_next       <= (m >= n) ? m - n : m;
55                  n_next       <= (n >  m) ? n - m : n;
56                  gcd          <= 16'd0;
57                  done_next <= ((m == 0) | (n == 0));
58                  state_next <= s1;
59              end else if (m[0] & ~n[0]) begin
60                  n_next <= n >> 1;
61                  gcd        <= 16'd0;
62                  done_next <= ((m == 0) | (n == 0));
63                  state_next <= s1;
64              end else if (~m[0] & n[0]) begin
65                  m_next <= m >> 1;
66                  gcd        <= 16'd0;
67                  done_next <= ((m == 0) | (n == 0));
68                  state_next <= s1;
69              end else begin
70                  n_next <= n >> 1;
71                  m_next <= m >> 1;
72                  factor_next <= factor + 1;
73                  gcd        <= 16'd0;
74                  done_next <= ((m == 0) | (n == 0));
75                  state_next <= s1;
76              end
77
78          s2: begin
79              gcd        <= 16'd0;
80              done_next <= ((m == 0) | (n == 0));
81              state_next<= s2;
82              end
83
84          default: begin
85                  state_next <= s0;  // jump back to init
86                  end
87      endcase
88    end
89
90  endmodule
```

4.7.3 GCD in VHDL

Listing 4.22 Binary GCD in VHDL

```vhdl
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4
5  entity gcd is
6    port( m_in, n_in : in  std_logic_vector(15 downto 0);
7          gcd        : out std_logic_vector(15 downto 0);
8          clk, rst   : in  std_logic
9        );
10 end gcd;
11
12 architecture behavior of gcd is
13   type statetype is (s0, s1, s2);
14   signal state, state_next : statetype;
15   signal m, m_next : std_logic_vector(15 downto 0);
16   signal n, n_next : std_logic_vector(15 downto 0);
17   signal done, done_next : std_logic;
18   signal factor, factor_next : std_logic_vector(15 downto 0);
19   begin
20
21   update_regs: process(clk, rst)
22   begin
23     if (rst='1') then
24       m      <= (others => '0');
25       n      <= (others => '0');
26       done   <= '0';
27       factor <= (others => '0');
28       state  <= s0;
29     elsif (clk='1' and clk'event) then
30       state  <= state_next;
31       m      <= m_next;
32       n      <= n_next;
33       done   <= done_next;
34       factor <= factor_next;
35       state  <= state_next;
36     end if;
37   end process;
38
39   eval_logic: process(m_in, n_in, state)
40   begin
41   n_next      <= n;
42   m_next      <= m;
43   done_next   <= done;
44   factor_next <= factor;
45   gcd         <= (others => '0');
46
47   case state is
48
49     when s0 =>
50        m_next       <= m_in;
```

```
51          n_next      <= n_in;
52          factor_next <= (others => '0');
53          done_next   <= '0';
54          gcd         <= (others => '0');
55          state_next  <= s1;
56
57      when s1 =>
58          if (done = '1') then
59              if (m > n) then
60                  gcd <= conv_std_logic_vector(shl(unsigned(m),
61                                                 unsigned(factor)),16);
62              else
63                  gcd <= conv_std_logic_vector(shl(unsigned(n),
64                                                 unsigned(factor)),16);
65              end if;
66              state_next  <= s2;
67          elsif ((m(0) = '1') and (n(0) = '1')) then
68              if (m >= n) then
69                  m_next <= unsigned(m) - unsigned(n);
70                  n_next <= n;
71              else
72                  m_next <= m;
73                  n_next <= unsigned(n) - unsigned(m);
74              end if;
75              gcd        <= (others => '0');
76              if ((m = "0000000000000000") or (n = "0000000000000000"))
77                  then
78                  done_next <= '1';
79              else
80                  done_next <= '0';
81              end if;
82              state_next <= s1;
83          elsif ((m(0) = '1') and (n(0) = '0')) then
84              n_next <= '0' & n(15 downto 1);
85              gcd        <= (others => '0');
86              if ((m = "0000000000000000") or (n = "0000000000000000"))
87                  then
88                  done_next <= '1';
89              else
90                  done_next <= '0';
91              end if;
92              state_next <= s1;
93          elsif ((m(0) = '0') and (n(0) = '1')) then
94              m_next <= '0' & m(15 downto 1);
95              gcd        <= (others => '0');
96              if ((m = "0000000000000000") or (n = "0000000000000000"))
97                  then
98                  done_next <= '1';
99              else
100                 done_next <= '0';
101             end if;
102             state_next <= s1;
103         else
104             n_next <= '0' & n(15 downto 1);
```

```
105              m_next <= '0' & m(15 downto 1);
106              factor_next <= conv_std_logic_vector(unsigned(factor) +
107                          conv_unsigned(1,16),16);
108              gcd         <= (others => '0');
109              if ((m = "0000000000000000") or (n = "0000000000000000"))
110                 then
111                 done_next <= '1';
112              else
113                 done_next <= '0';
114              end if;
115              state_next <= s1;
116           end if;
117
118        when s2 =>
119           gcd <= (others => '0');
120           if ((m = "0000000000000000") or (n = "0000000000000000"))
121              then
122              done_next <= '1';
123           else
124              done_next <= '0';
125           end if;
126           state_next<= s2;
127
128        when others =>
129           state_next <= s0;
130
131        end case;
132     end process;
133  end behavior;
```

4.7.4 GCD in SystemC

Listing 4.23 Binary GCD in SystemC

```
1  #include "systemc.h"
2
3  enum statetype {s0, s1, s2};
4
5  SC_MODULE(gcd_fsmd) {
6    sc_in <bool>          clk;
7    sc_in <bool>          rst;
8    sc_in <sc_uint<16> >  m_in, n_in;
9    sc_out <sc_uint<16> > gcd;
10
11   sc_signal<statetype>  state, state_next;
12   sc_uint<16>           m,      m_next;
13   sc_uint<16>           n,      n_next;
14   sc_uint<16>           factor, factor_next;
15   sc_uint< 1>           done,   done_next;
16
17   void update_regs();
18   void eval_logic();
19   SC_CTOR(gcd_fsmd) {
```

```
20       SC_METHOD(eval_logic);
21       sensitive << m_in << n_in << state;
22       SC_METHOD(update_regs);
23       sensitive_pos << rst << clk;
24     }
25   };
26
27   void gcd_fsmd::update_regs() {
28     if (rst.read() == 1) {
29       state  = s0;
30       m      = 0;
31       n      = 0;
32       factor = 0;
33       done   = 0;
34     } else {
35       state  = state_next;
36       m      = m_next;
37       n      = n_next;
38       factor = factor_next;
39       done   = done_next;
40     }
41   }
42
43   void gcd_fsmd::eval_logic() {
44
45     n_next      = n;
46     m_next      = m;
47     done_next   = done;
48     factor_next = factor;
49     gcd         = 0;
50
51     switch(state) {
52       case s0:
53         m_next      = m_in;
54         n_next      = n_in;
55         factor_next = 0;
56         done_next   = 0;
57         gcd         = 0;
58         state_next  = s1;
59         break;
60       case s1:
61         if (done == 1) {
62           gcd = ((m > n) ? m : n) << factor;
63           state_next = s2;
64         } else if (m[0] & n[0]) {
65           m_next      = (m >= n) ? m - n : m;
66           n_next      = (n >  m) ? n - m : n;
67           gcd         = 0;
68           done_next   = ((m == 0) | (n == 0));
69           state_next = s1;
70         } else if (m[0] & ~n[0]) {
71           n_next      = (n >> 1);
72           gcd         = 0;
73           done_next   = ((m == 0) | (n == 0));
```

```
74          state_next = s1;
75        } else if (~m[0] & n[0]) {
76          m_next     = m >> 1;
77          gcd        = 0;
78          done_next  = ((m == 0) | (n == 0));
79          state_next = s1;
80        } else {
81          n_next     = n >> 1;
82          m_next     = m >> 1;
83          factor_next= factor + 1;
84          gcd        = 0;
85          done_next  = ((m == 0) | (n == 0));
86          state_next = s1;
87        }
88        break;
89      case s2:
90        gcd  = 0;
91        done_next  = ((m == 0) | (n == 0));
92        break;
93      default:
94        state_next = s0;
95      }
96  }
```

4.8 Summary

In this section, we discussed a synchronous hardware modeling mechanism, consisting of a datapath in combination with an FSM controller. The resulting model is called FSMD (Finite State Machine with Datapath). An FSMD models datapath instructions with expressions, and control with a state transition graph. Datapath expressions are created in terms of register variables and signals (wires). Register variables are implicitly attached to the global clock signal. Datapath instructions (groups of datapath expressions) form the connection between the controller and the datapath.

A given FSMD design is not unique. A given design can be decomposed into many different, equivalent FSMD descriptions. It is up to the designer to pick a modeling style that feels natural and that is useful for the problem at hand.

We discussed a modeling syntax for FSMD called GEZEL. GEZEL models can be simulated and converted into synthesizable VHDL code. However, the FSMD model is generic and can be captured into any suitable hardware description language. At the end of the chapter, we showed equivalent synthesizable implementations of an FSMD in GEZEL, Verilog, VHDL, and SystemC.

4.9 Further Reading

The FSMD model has been recognized as a universal model for RTL modeling of hardware. See Vahid (2007a) for a textbook that starts from combinational and sequential logic, and gradually works up to FSMD based design. FSMD were

popularized by Gajski, and are briefly covered in Gajski et al. (2009). Going back earlier in time, one can find an excellent development of the FSMD model in Davio, Deschamps, and Thaysse Davio et al. (1983).

The GEZEL toolset can be downloaded from `http://rijndael.ece.vt.edu/gezel2`. *Proper* FSMD, as defined in this chapter, are race-free. A mathematical proof of this can be found in Schaumont et al. (2006).

There is an ongoing discussion how to improve the productivity of hardware design. Some researchers believe that *high-level synthesis*, the automatic generation of RTL starting from high-level descriptions, is unavoidable. Several academic and commercial design tools that support such high level synthesis are described in Gajski et al. (2009). See Gupta et al. (2004) for a detailed description of one such an environment. On the other hand, the nature of hardware design is such that designers like to think about clock cycles when they think about architecture. Hence, abstraction should be applied with utmost care. See Hoe (2000) and Qin (2004) for examples for such carefully abstracted hardware design and modeling paradigms.

4.10 Problems

4.1. Which of the circuits (a, b, c, d) in Fig. 4.9 can be simulated using a cycle-based simulator?

4.2. Design a high-speed sorter for four 32-bit registers (Fig. 4.10). Show how to create a sorting network for four numbers, using only simple two-input comparator modules. The comparator modules are built with combinational logic and have a constant critical path. Optimize the critical path of the overall design and create a maximally parallel implementation. You may make use of comparator modules, registers, and wiring. The input of the sorter comes from four registers marked 'input', the output of the sorter needs to be stored in four registers marked 'output'.
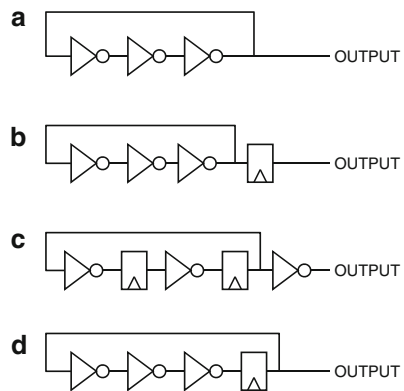
Fig. 4.9 Sample circuits for Problem 4.1
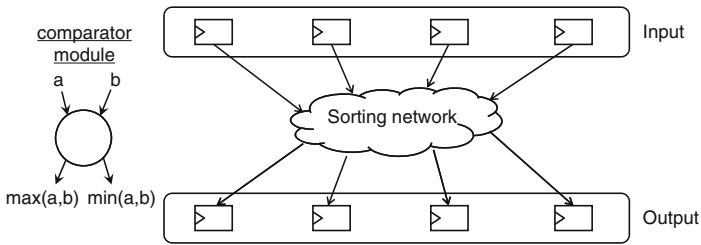
Fig. 4.10 Sorter design for Problem 4.2

Fig. 4.11 Pattern recognizer
for Problem 4.3
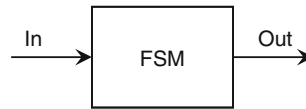

```
In    1 0 1 1 0 0 1 1 1 0 1 0 1 ...
Out   0 1 0 0 1 0 1 0 0 1 0 1 0 ...
```

4.3. Design a FSM that recognizes the pattern '10' and the pattern '01' in an infinite
stream of bits. Make sure that the machine recognizes only one pattern at a time, and
that it is not triggered by overlapping patterns. Figure 4.11 shows an example of the
behavior of this FSM.

1. Draw a Mealy-type state diagram of this FSM.
2. Draw an RTL schematic of an implementation for this machine. Draw your im-
   plementation using registers and logic gates (AND, OR, NOT, and XOR). Make
   your implementation as compact as possible.

4.4. Design a Mealy-type FSM that recognizes either of the following two patterns:
1101 or 0111. The patterns should be read left to right (i.e., the leftmost bit is seen
first), and they are to be matched into a stream of single bits.

4.5. Design an FSMD to divide natural numbers. The dividend and the divider each
have 8 bits of resolution. The quotient must have 10 bits of resolution, and the
remainder must have 8 bits of resolution. The divider has the following interface:

```
dp divider(in  x     : ns(8);
           in  y     : ns(8);
           in  start : ns(1);
           out q     : ns(10);
           out r     : ns(8);
           out done  : ns(1)) {

 // Define the internals of the FSMD here ..

}
```

Given a dividend X and a divider Y, the divider will evaluate a quotient Q on p bits of precision and a remainder R such that

$$X.2^p = Q.Y + R \qquad (4.1)$$

For example, if $p = 8$, $X = 12$, $Y = 15$, then a solution for Q and R is $Q = 204$ and $R = 12$ because $12 . 28 = 204 . 15 + 12$.

Your implementation must obtain the quotient and remainder within 32 clock cycles. To implement the divider, you can use the restoring division algorithm as follows. The basic operation evaluates a single bit of the quotient according to the following pseudocode:

```
basic_divider(input a, b;
              output q, r) {
  z := 2 * a - b;
  if (z < 0) then
     q = 0;
     r = 2 * a;
  else
     q = 1;
     r = z;
}
```

To evaluate the quotient over p bits, you repeat the basic 1-bit divider p times as follows:

```
r(0) = X;
for i is 1 to p do
basic_divider(r(i-1), Y, q(i), r(i));
```

Each iteration creates one bit of the quotient, and the last iteration returns the remainder. For example, if $p = 8$, then $Q = q(0), q(1), q(2), ..., q(7)$ and $R = r(8)$.

Create a hardware implementation which evaluates one bit of the quotient per clock cycle.

4.6. How many flip-flops and how many adders do you need to implement the FSMD description in Listing 4.24? Count each single bit in each register, and assume binary encoding of the FSM state, to determine the flip-flop count.

4.7. FSMD models provide modeling of control (conditional execution) as well as data processing in hardware. Therefore, it is easy to mimic the behavior of a C program and build an FSMD that reflects the same control flow as the C program. Write an FSMD model for the following C function. Assume that the arguments of the function are the inputs of the FSMD, and that the result of the function is the FSMD output. Develop your model so that you need no more than a single multiplier.

Listing 4.24 Program for Problem 4.5

```
1   dp mydp(in i : ns(5); out o : ns(5)) {
2     reg a1, a2, a3, a4 : ns(5);
3     sfg f1 { a1 = i;
4               a2 = 0;
5               a3 = 0;
6               a4 = 0; }
7     sfg f2 { a1 = a2 ? (a1 + a3) : (a1 + a4); }
8     sfg f3 { a3 = a3 + 1; }
9     sfg f4 { a4 = a4 + 1; }
10    sfg f5 { a2 = a2 + a1; }
11  } fsm mydp_ctl(mydp) {
12    initial s0;
13    state s0, s1, s2;
14    @s0 (f1) -> s1;
15    @s1 if (a1) then (f2, f3) -> s2;
16               else (f4)      -> s1;
17    @s2 if (a3) then (f2)     -> s1;
18               else (f5)      -> s2;
19  }
```

Listing 4.25 Program for Problem 4.6

```
1   int filter(int a) {
2     static int taps[5];
3     int c[] = {-1, 5, 10, 5, -1};
4     int r;
5
6     for (i=0; i<4; i++)
7       taps[i] = taps[i+1];
8     taps[4] = a;
9     r = 0;
10    for (i=0; i<5; i++)
11      r = r + taps[i] * c[i];
12
13    return r;
14  }
```

To model an array of constants in GEZEL, you can make use of the lookup table construct as follows:

```
dp lookup_example {

    lookup T : ns(8) = {5, 4, 3, 2, 1, 1, 1, 1};

    sig a, b : ns(3);

    always {
      a = 3;
      b = T[a]; // this assigns the fourth element of T to b
    }
}
```

Listing 4.26 Program for Problem 6.8

```
1  dp D( in fp, i1, i2 : ns(4); out mul: ns(4);
2       in mul_st: ns(1);
3       out mul_done : ns(1)) {
4    reg acc, sr2, fpr, r1 : ns(4);
5    reg mul_st_cmd : ns(1);
6    sfg ini { // initialization
7       fpr       = fp;
8       r1        = i1;
9       sr2       = i2;
10      acc       = 0;
11      mul_st_cmd = mul_st;
12   }
13   sfg calc { // calculation
14      sr2 = (sr2 << 1);
15      acc = (acc << 1) ^ (r1 & (tc(1))  sr2[3])  // add a if b=1
16            ^ (fpr & (tc(1)) acc[3]); // reduction if carry
17   }
18   sfg omul { // output inactive
19      mul      = acc;
20      mul_done = 1;
21      $display("done. mul=", mul);
22   }
23   sfg noout { // output active
24      mul      = 0;
25      mul_done = 0;
26   }
27 }
28 fsm F(D) {
29   state s1, s2, s3, s4, s5;
30   initial   s0;
31   @s0 (ini,  noout) -> s1;
32   @s1 if (mul_st_cmd) then (calc, noout) -> s2;
33                       else (ini, noout)  -> s1;
34   @s2 (calc, noout) -> s3;
35   @s3 (calc, noout) -> s4;
36   @s4 (calc, noout) -> s5;
37   @s5 (ini,  omul ) -> s1;
38 }
```

4.8. Repeat problem 4.6, but develop your FSMD so that the entire function completes in a single clock cycle.

4.9. Write the FSMD of Listing 4.26 in a single `always` block. This FSMD presents a Galois Field multiplier.

4.10. In this chapter, we discussed how FSM can be expressed as datapath expressions (See Sect. 4.4.2 and Problem 4.8). It is also possible to go the opposite way, and model datapaths in terms of FSMs.

1. Write an FSM for the datapath shown in Listing 4.27.
2. Discuss why it is a bad idea to model datapath expressions as FSM, while it can still be useful to model FSM as datapath expression.

Listing 4.27 Program for Problem 4.9

```
1  dp tester(out o: ns(2)) {
2    reg a1 : ns(1);
3    reg a2 : ns(2);
4    always {
5      a1 = a1 + 1;
6      a2 = a2 + a1;
7      o  = a2;
8    }
9  }
```

Chapter 5
Microprogrammed Architectures

Abstract The Finite State Machine controller in an FSMD is nonprogrammable. By substituting this FSM for a programmable controller, you obtain a micropro-grammed architecture. The advantage of a programmable architecture is obviously the flexibility to implement multiple functionalities. This chapter discusses the design of microprogrammed controllers and datapaths, and it explains the advantages and limitations of microprogramming. In particular, you will see that complex, pipelined datapaths are not easy to handle because of the bare-bones approach to control.

5.1 Limitations of Finite State Machines

Finite State Machines are well suited to capture the control flow of algorithms, and to support their decision-making. Recall for example how FSM state transition diagrams resemble control dependency graphs (CDG). However, FSM are no universal solution for control modeling. They suffer from several modeling weaknesses, especially when dealing with complex control requirements.

A key issue is that FSMs are flat. They don't express any hierarchy. A *flat* control model is like a C program where the entire program logic is captured in a single function. Real systems do not use a flat control model: they need a control hierarchy. Of course, there have been several proposals for hierarchical modeling extensions for FSMs, such as the Statecharts from David Harel. Currently, none of these are widely used for hardware design. Many of the limitations of FSMs stem from their lack of hierarchy.

5.1.1 State Explosion

A flat FSM suffers from *state explosion*, which occurs when multiple independent activities interfere in a single model. Assume that an FSM has to capture two independent activities, each of which can be in one of three states. The resulting FSM,
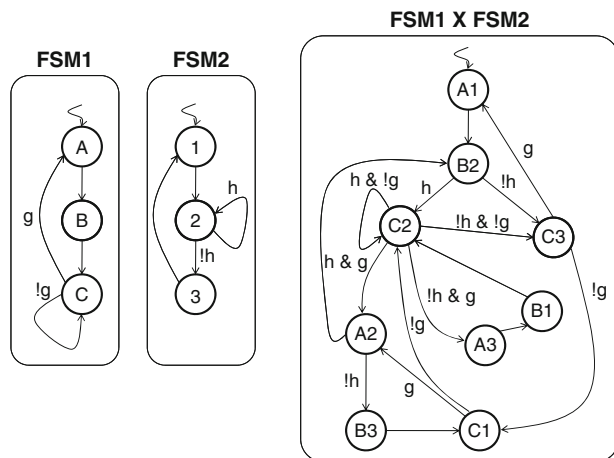
Fig. 5.1 State explosion in FSM when creating a product state-machine

called a *product state-machine*, needs nine states to represent the overall model. The product state-machine needs to keep track of the current state from two independent state machines at the same time. Due to conditional state transitions, one state machine can remain in a single state while the other state machine proceeds to the next state. This results in multiple intermediate states such as A1, A2, and A3. Figure 5.1 illustrates the effect of state explosion in a product state-machine. Two state machines, FSM1 and FSM2, need to be merged into a single product state-machine FSM1xFSM2. In order to represent all individual states, 9 states are needed in total. The resulting number of state transitions (and state transition conditions) is even higher. Indeed, if we have n independent state transition conditions in the individual state machines, the resulting product state-machine can have up to 2^n state transition conditions.

5.1.2 Exception Handling

A second issue with a flat FSM is the problematic handling of exceptions. An exception is a condition which may cause an immediate state transition, regardless of the current state of the finite state machine. The purpose of an exception is to abort the regular flow of control and to transfer the control to a dedicated exception-handler. An exception may have internal causes, such as an overflow condition in a datapath, or external causes, such as an interrupt. Regardless of the cause, the effect of an exception on a finite state machine model is dramatic: an additional state transition needs to be added to every state of the finite state machine. For example, assume that the product state-machine in Fig. 5.1 needs to include an exception input called

Fig. 5.2 Adding a single global exception deteriorates the readability of the FSM significantly
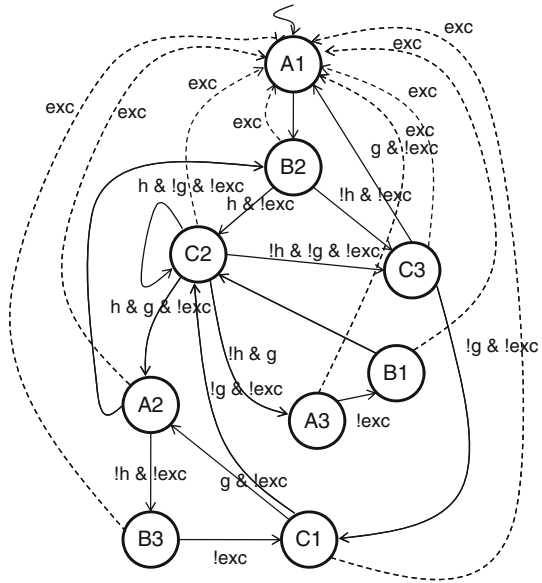
exc, and that the assertion of that input requires immediate transition to state A1. The resulting FSM, shown in Fig. 5.2, shows how exceptions degrade the clarity of the FSM state transition graph.

5.1.3 Runtime Flexibility

Finally, and perhaps the biggest issue from the viewpoint of hardware–software codesign, a finite state machine is a nonflexible model. Once the states and state transitions are defined, the control flow of the FSM is fixed. The hardware implementation of a FSM leads to a hardwired controller that cannot be modified after implementation.

As a result, designers have proposed improved techniques for specifying and implementing control, in order to deal with flexibility, exceptions, and hierarchical modeling. Microprogramming is one such a technique. Originally introduced in the 1950s by Maurice Wilkes as a means to create a programmable instruction-set for mainframe computers, it became very popular in the 1970s and throughout the 1980s. Microprogramming was found to be very useful to develop complex microprocessors with flexible instruction-sets. Currently (2008), microprogramming is often ignored because of the extreme polarization between hardware and software. Mainstream design has evolved into an activity where flexibility is almost always implemented on microprocessors, in software, while dedicated and hard-coded design is implemented in hardware. However, newer architectures, such as FPGAs and ASIPs, suggest that flexibility is not the exclusive domain of software.

In this chapter, we will investigate the microprogramming technique and learn how it can be used to provide flexibility to hardware circuits while maintaining full customizability.

5.2 Microprogrammed Control

Figure 5.3 shows a microprogrammed machine next to an FSMD. The fundamental idea in microprogramming is to replace the next-state logic of the finite state-machine with a *programmable* memory, called the control store. The control store holds microinstructions, and is addressed using a register called CSAR (Control Store Address Register). The CSAR is equivalent to a program counter in micro-processors. The next-value of CSAR is determined by the next-address logic, using the current value of CSAR, the current microinstruction and the value of status flags evaluated by the datapath. The default next-value of the CSAR corresponds to the previous CSAR value incremented by one. This way, the control store will return a stream of microinstructions from sequential memory locations. However, the next-address logic can also implement conditional jumps or immediate jumps.

Thus, the next-address logic, the CSAR, and the control store implement the equivalent of an instruction-fetch cycle in a microprocessor. In the design of Fig. 5.3, each microinstruction takes a single clock cycle to execute. Within a single clock cycle, the following activities occur:

- The CSAR provides an address to the control store which retrieves a microin-struction. The microinstruction is split into two parts: a command-field and a jump-field. The command-field serves as a command for the datapath. The jump-field goes to the next-address logic.
- The datapath executes the command encoded in the microinstruction, and returns status information to the next-address logic.
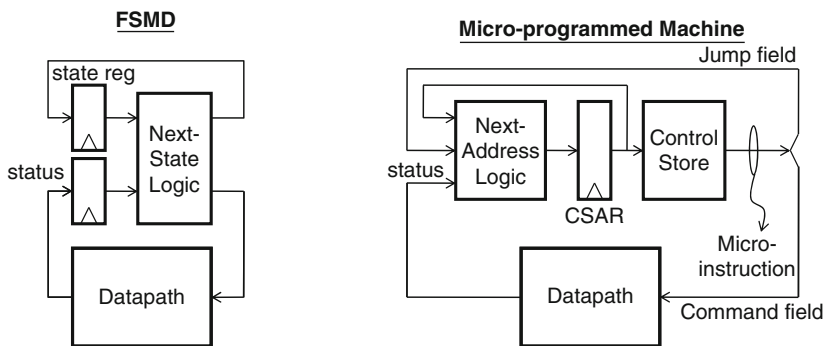
Fig. 5.3 In contrast to FSM-based control, microprogramming uses a flexible control scheme

- The next-address logic combines datapath states, microinstruction jump-field, and status returned from the datapath. The next-address logic will eventually update the CSAR. Consequently, the critical path of the microprogrammed machine in Fig. 5.3 is determined by the combined logic delay through the control store, the next-address logic, and the datapath.

While the microprogrammed controller is more complicated than the finite state machine, it also addresses the problems of FSMs very effectively.

1. The microprogrammed controller scales well with complexity. For example, a 12-bit CSAR will allow a control store with 4,096 locations, and therefore a microprogram with 4,096 steps. An equivalent FSM diagram with 4,096 states, on the other hand, would be horrible to draw!
2. A microprogrammed machine deals very well with control hierarchies. With small modifications to the microprogrammed machine in Fig. 5.3, we can save the CSAR in a separate register or on a stack memory, and later restore it. This requires the definition of a separate microinstruction to call a subroutine as well as a second microinstruction to return from it.
3. A microprogrammed machine can deal efficiently with exception handling, since global exceptions are managed directly by the next-address logic, independently from the control store. For example, the presence of a global exception can feed a hard-coded value into the CSAR, immediately transferring the microprogrammed machine to an exception-handling routine. Exception handling in a microprogrammed machine is similar to a jump instruction, but it does not affect every instruction of a microprogram in the same way as it affects every state of a finite state machine.
4. Finally, microprograms are flexible and very easy to change after the microprogrammed machine is designed. Simply changing the contents of the control store is sufficient to change the program of the machine. In a microprogrammed machine, there is a clear distinction between the architecture of the machine and the functionality implemented using that architecture.

5.3 Microinstruction Encoding

An interesting design problem that is part of the microprogrammed design is the format of microinstructions in the control store. In this section, we will discuss the design trade-offs that determine the microinstruction format.

5.3.1 Jump Field

We start with an example of microinstruction encoding, shown in Fig. 5.4. This is a 32-bit microinstruction word, with 16 bits reserved for the datapath, and 16 bits
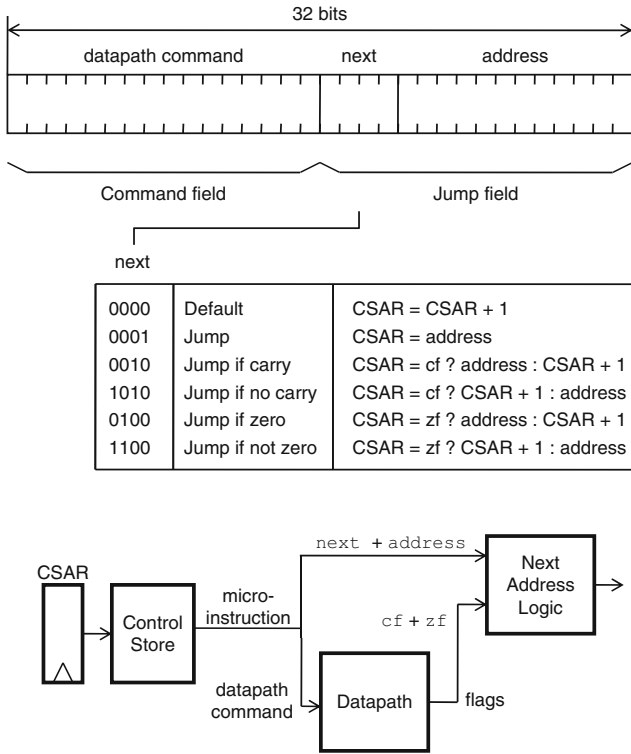
Fig. 5.4 Sample format for a 32-bit microinstruction word

reserved for the next-address logic. Let us first consider the part for the next-address logic. The `address` field holds an absolute target address, pointing to a location in the control store. In this case, the address is 12 bit, which means that this microinstruction format would be suited for a control store with 4,096 locations. The `next` field encodes the operation that will lead to the next value of CSAR. The default operation is, as discussed earlier, to increment CSAR. For such instructions, the address field remains unused. When `next` has values different from the default one, various jump instructions can be encoded. An absolute jump will transfer the value of the `address` field into CSAR. A conditional jump will use the value of a flag to conditionally update the CSAR or else increment it. Obviously, the format as shown is quite bulky and may consume a large amount of storage. For example, the `address` field is only used for jump instructions. In an average micro-processor program, about 1 instruction out of 5 is a jump. Therefore, if the microprogram contains only a few jump instructions, then the storage for the address field is wasted. To avoid this, we will need to optimize the microinstruction format. For example, when microinstructions are no jumps, then the bits used for the `address` field could be given a different purpose.

5.3.2 Command Field

The design of the datapath command format reveals another interesting trade-off: we can either opt for a very wide microinstruction word, or else we can prefer a narrow microinstruction word. A wide microinstruction word allows each control bit of the datapath to be stored separately. A narrow microinstruction word, on the other hand, will require the creation of "symbolic instructions", which are encoded groups of control-bits for the datapath. The FSMD model relies on such symbolic instructions. Each of the above approaches has a specific name. *Horizontal* microinstructions use no encoding at all. They represent each control bit in the datapath with a separate bit in the microinstruction format. *Vertical* microinstructions on the other hand encode the control bits for the datapath as much as possible. A few bits of the microinstruction can define the value of many more control bits in the datapath.

Figure 5.5 demonstrates an example of vertical and horizontal microinstructions in the datapath. We wish to create a microprogrammed machine with three instructions on a single register `a`. The three instructions do one of the following: double the value in `a`, decrement the value in `a`, or initialize the value in `a`. The datapath shown on the bottom of Fig. 5.5 contains two multiplexers and a programmable adder/subtractor. It can be easily verified that each of the instructions enumerated above can be implemented as a combination of control bit values for each multiplexer and for the adder/subtractor. The controller on top shows two possible encodings for the three instructions: a horizontal encoding and a vertical encoding.
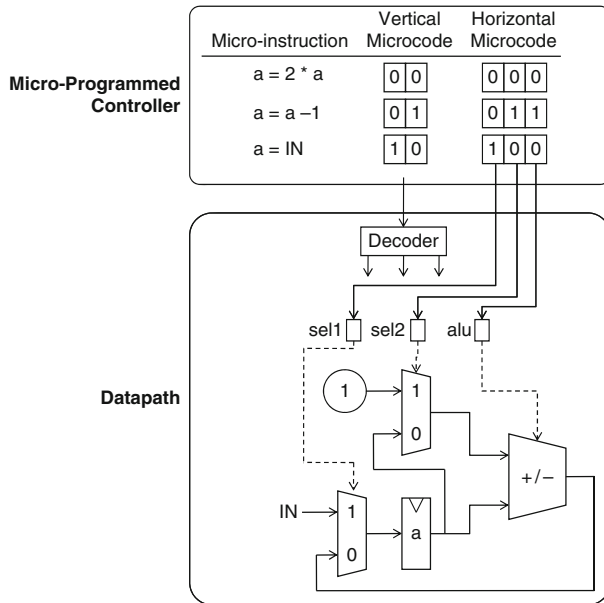
Fig. 5.5 Example of vertical versus horizontal microprogramming

- In the case of horizontal microcode, the control store will include each of the control bits in the datapath as a bit in the microinstruction word. Hence, in this case, the encoding of the instructions reflects exactly the required setting of datapath elements for each microinstruction.
- In the case of vertical microcode, the microinstructions will be encoded. Since there are three different instructions, we can implement this machine with a two-bit microinstruction word. To generate the control bits for the datapath, we will have to decode each of the microinstruction words into local control signals on the datapath.

We can describe the design trade-off between horizontal and vertical microprograms as follows: Vertical microprograms have a better code density, which is beneficial for the size of the control store. For the example in Fig. 5.5, the vertically-encoded version of the microprogram will be only 2/3 of the size of the horizontally-encoded version. On the other hand, vertical microprograms use an additional level of encoding, so that each microinstruction needs to be decoded before it can drive the control bits of the datapath. Thus, the machine with the vertically encoded microprogram may have a longer critical path.

Obviously, the choice between a vertical and horizontal encoding needs to be made carefully. In practice, designers use a combination of vertical and horizontal encoding concepts so that the resulting digital structure is compact yet efficient. Consider for example the value of the `next` field of the microinstruction word in Fig. 5.4. There are six different types of jump instructions, which would imply that a vertical microinstruction would need no more than three bits to encode these six jumps. Yet, four bits have been used, indicating that there is some redundancy left. The encoding was chosen to simplify the design of the next-address logic, which is shown in Fig. 5.6. Another reason to leave "open room" in the encoding of microinstructions is to allow future upgrades. For example, in the design in Fig. 5.6, it is quite easy to add an additional conditional jump that uses an arbitrary combination of `cf` and `zf`.
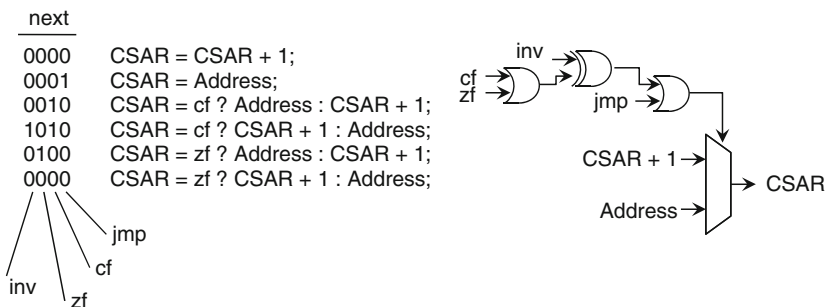
Fig. 5.6 Example of vertical versus horizontal microprogramming

5.4 The Microprogrammed Datapath

The datapath of a microprogrammed machine consists of three elements: computation units, storage, and communication busses. Each of these may contribute a few control bits to the microinstruction word. For example, multi-function computation units have selection bits that determine their specific function, storage units have address bits and read/write command bits, and communication busses have source/destination control bits. The datapath may also generate condition flags for the microprogrammed controller. Typically, these will be generated by the computation units.

5.4.1 Datapath Architecture

Figure 5.7 illustrates a microprogrammed controller with a datapath attached. The datapath includes an ALU with shifter unit, a register file with 8 entries, an accumulator register, and an input port.

The microinstruction word is shown on top of Fig. 5.7 and contains 6 fields. Two fields, Nxt and Address, are used by the microprogrammed controller. The other are used by the datapath. The type of encoding is mixed horizontal/vertical: the overall machine uses a horizontal encoding: each module of the machine is controlled independently. The submodules within the machine on the other hand use a vertical encoding. For example, the ALU field contains 4 bits. In this case, the ALU component in the datapath will execute up to 16 different commands.

The machine completes a single instruction per clock cycle. The ALU combines an operand from the accumulator register with an operand from the register file or the input port. The result of the operation is returned to the register file or the
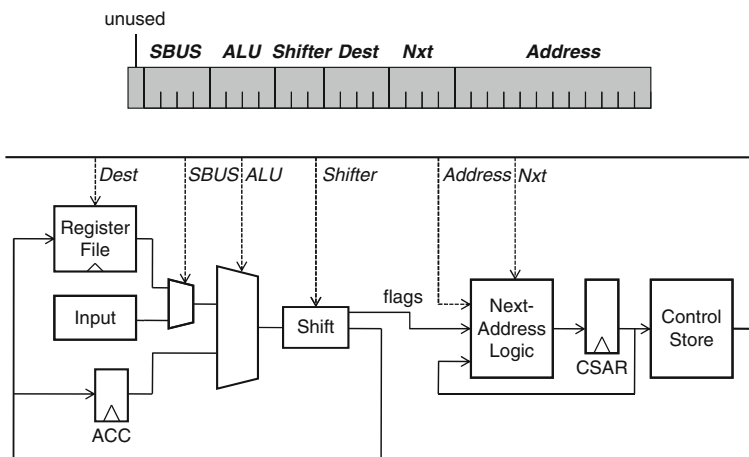
Fig. 5.7 A microprogrammed datapath

accumulator register. The communication used by datapath operations is controlled by two fields in the microinstruction word. The *SBUS* field and the *Dest* field indicate the source and destination, respectively.

The Shifter module also generates flags, which are used by the microprogrammed controller to implement conditional jumps. Two flags are created: a zero-flag, which is high (1) when the output of the shifter is all-zero, and a carry-flag, which contains the bit shifted-out at the most-significant position.

5.4.2 Writing Microprograms

Table 5.1 illustrates the encoding used by each module of the design from Fig. 5.7. A microinstruction can be formed by selecting a module function for each module of the microprogrammed machine, including a next-address for the Address field. When a field remains unused during a particular instruction, a don't care value can be chosen. The don't care value should be chosen so that unwanted state changes in the datapath are avoided.

For example, an instruction to copy register R2 into the accumulator register ACC would be formed as follows: The instruction should read out register R2 from the register file, pass the register contents over the SBus, through the ALU and the shifter, and write the result in the ACC register. This observation allows to determine the value of each field in the microinstruction.

- The SBus needs to carry the value of R2. Using Table 5.1 we find SBUS equals 0010.
- The ALU needs to pass the SBus input to the output based on Table 5.1, ALU must equal 0001.
- The shifter passes the ALU output unmodified, hence Shifter must equal 111.
- The output of the shifter is used to update the accumulator register, so the Dest field equals 1000.
- Assuming that no jump or control transfer is executed by this instruction, the next microinstruction will simply be one beyond the current CSAR location. This implies that Nxt should equal 0000 and Address is a don't-care, for example all-zeroes.
- Finally, we can find the overall microinstruction code by putting all instruction fields together. Figure 5.8 illustrates this process. We conclude that a microinstruction to copy R2 into ACC can be encoded as 0x10F80000 in the control store.

Writing a microprogram thus consists of formulating the desired behavior as a sequence of register transfers, and next encoding these register transfers as microinstruction fields. Higher level constructs, such as loops and if-then-else statements, can be expressed as a combination (or sequence) of register transfers. While this makes microprogramming look like a tedious process, keep in mind that this also offers the programmer full control over the hardware at every clock cycle.

Table 5.1 Microinstruction encoding of the example machine

Field	Width	Encoding			
SBUS	4	Selects the operand that will drive the S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	Input
		0100	R4	1001	Address/Constant
ALU	4	Selects the operation performed by the ALU			
		0000	ACC	0110	ACC — S-Bus
		0001	S-Bus	0111	not S-Bus
		0010	ACC + SBus	1000	ACC + 1
		0011	ACC − SBus	1001	SBus − 1
		0100	SBus − ACC	1010	0
		0101	ACC & S-Bus	1011	1
Shifter	3	Selects the function of the programmable shifter			
		000	logical SHL(ALU)	100	arith SHL(ALU)
		001	logical SHR(ALU)	101	arith SHR(ALU)
		010	rotate left ALU	111	ALU
		011	rotate right ALU		
Dest	4	Selects the target that will store S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	ACC
		0100	R4	1111	unconnected
Nxt	4	Selects next-value for CSAR			
		0000	CSAR + 1	1010	cf ? CSAR + 1 : Address
		0001	Address	0100	zf ? Address : CSAR + 1
		0010	cf ? Address : CSAR + 1	1100	zf ? CSAR + 1 : Address

As an example, let us develop a microprogram that reads two numbers from the input port and that evaluates their greatest common divisor (GCD) using Euclid's algorithm. The first step is to develop a microprogram in terms of register transfers. A possible approach is shown in Listing 5.1. Lines 2 and 3 in this program read in two values from the input port and store these values in registers R0 and ACC. At the end of the program, the resulting GCD will be available in either ACC or R0, and the program will continue until both values are equal. The stop test is implemented in line 4, using a subtraction of two registers and a conditional jump based on the zero-flag. Assuming both registers contain different values, the program will continue to subtract the largest register from the smallest one. This requires to find which of R0 and ACC is bigger, and it is implemented with a conditional jump in line 5. The bigger-than test is implemented using a subtraction, a left-shift, and a test on the resulting carry-flag. If the carry-flag is set, then the most-significant bit of the subtraction would be one, indicating a negative result in two's complement
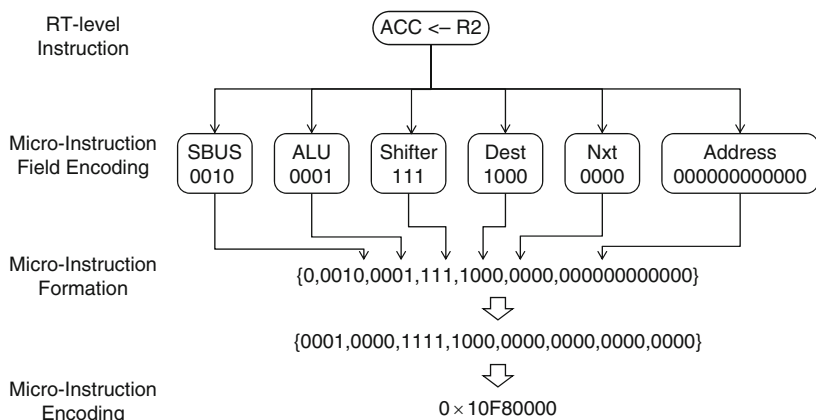
Fig. 5.8 Forming microinstructions from register-transfer instructions

Listing 5.1 Micro-program to evaluate a GCD

```
1  ;    Command Field      || Jump Field
2          IN -> R0
3          IN -> ACC
4  Lcheck: (R0 - ACC)      || JUMP_IF_Z Ldone
5          (R0 - ACC) << 1 || JUMP_IF_C LSmall
6          R0 - ACC -> R0  || JUMP Lcheck
7  Lsmall: ACC - R0 -> ACC || JUMP Lcheck
8  Ldone:                     JUMP Ldone
```

logic. This conditional jump-if-carry will be taken if R0 is smaller than ACC. The combination of lines 5, 6, and 7 shows how an if-then-else statement can be created using multiple conditional and unconditional jump instructions. When the program is complete, in line 8, it iterates in an infinite loop. Depending on how this microprogram is integrated into a bigger program, this infinite loop would have to be replaced with an appropriate control transfer to the next task.

5.5 Implementing a Microprogrammed Machine

In this section, we discuss a sample implementation of a microprogrammed machine in the GEZEL language. This can be used as a template for other implementations.

5.5.1 Microinstruction Word Definition

A convenient starting point in the design of a microprogrammed machine is the definition of the microinstruction. This includes the allocation of microinstruction control bits and the definition of the meaning of relevant bit-patterns.

The individual control fields are defined as subvectors of the microinstruction. Listing 5.2 shows the GEZEL implementation of the microprogrammed design discussed in the previous Section. The possible values for each microinstruction field are shown in Lines 5–65. The use of C macros simplifies the writing of microprograms.

The formation of a single microinstruction is done using a C macro as well, shown in Lines 68–75. Lines 78–128 show the microprogrammed controller, which includes a control store with a microprogram and the next-address CSAR logic. The control store is a lookup table with a sequence of microinstructions (lines 85–100). On line 110, a microinstruction is fetched from the control store, and broken down into individual fields which form the output of the microprogrammed controller (lines 110–117). The next-address logic uses the next-address control field to find a new value for CSAR each clock cycle (lines 120–126).

The microprogrammed machine includes several datapaths, including a register file (lines 130–161), an ALU (lines 163–185), a shifter (lines 187–211). Each of the datapaths is crafted along a similar principle: based on the control field input, the data-input is transformed into a corresponding data-output. The decoding process of control fields is visible as a sequence of ternary selection-operators.

The top-level cell for the microprogrammed machine is contained in lines 213–245. The top-level includes the controller, a register file, an ALU, and a shifter. The top-level module also defines a data-input port and a data-output port, and each has a strobe control signal that indicates a data-transfer. The strobe signals are generated by the top-level module based decoding of microinstruction fields. The input strobe is generated when the SBUS control field indicates that the SBUS will be reading an external input. The output strobe is generated by a separate, dedicated microinstruction bit.

A simple testbench for the top-level cell is shown on lines 247–266. The testbench feeds in a sequence of data to the microprogrammed machine, and prints out each number appearing at the data output port. The microprogram for this machine evaluates the GCD of each tuple in the list of numbers shown on line 255.

Listing 5.2 Micro-programmed controller in GEZEL

```
1   // wordlength in the datapath
2   #define WLEN 16
3
4   /* encoding for data output */
5   #define O_NIL     0     /* OT <- 0 */
6   #define O_WR      1     /* OT <- SBUS */
7
8   /* encoding for SBUS multiplexer */
9   #define SBUS_R0   0     /* SBUS <- R0  */
10  #define SBUS_R1   1     /* SBUS <- R1  */
11  #define SBUS_R2   2     /* SBUS <- R2  */
12  #define SBUS_R3   3     /* SBUS <- R3  */
13  #define SBUS_R4   4     /* SBUS <- R4  */
14  #define SBUS_R5   5     /* SBUS <- R5  */
15  #define SBUS_R6   6     /* SBUS <- R6  */
16  #define SBUS_R7   7     /* SBUS <- R7  */
```

```
17  #define SBUS_IN   8     /* SBUS <- IN  */
18  #define SBUS_X SBUS_R0 /* don't care */
19
20  /* encoding for ALU */
21  #define ALU_ACC  0    /* ALU <- ACC */
22  #define ALU_PASS 1    /* ALU <- SBUS */
23  #define ALU_ADD  2    /* ALU <- ACC + SBUS */
24  #define ALU_SUBA 3    /* ALU <- ACC - SBUS */
25  #define ALU_SUBS 4    /* ALU <- SBUS - ACC */
26  #define ALU_AND  5    /* ALU <- ACC and SBUS */
27  #define ALU_OR   6    /* ALU <- ACC or SBUS */
28  #define ALU_NOT  7    /* ALU <- not SBUS */
29  #define ALU_INCS 8    /* ALU <- ACC + 1 */
30  #define ALU_INCA 9    /* ALU <- SBUS - 1 */
31  #define ALU_CLR  10   /* ALU <- 0 */
32  #define ALU_SET  11   /* ALU <- 1 */
33  #define ALU_X ALU_ACC /* don't care */
34
35  /* encoding for shifter */
36  #define SHFT_SHL 1      /* Shifter <- shiftleft(alu) */
37  #define SHFT_SHR 2      /* Shifter <- shiftright(alu) */
38  #define SHFT_ROL 3      /* Shifter <- rotateleft(alu) */
39  #define SHFT_ROR 4      /* Shifter <- rotateright(alu) */
40  #define SHFT_SLA 5      /* Shifter <- shiftleftarithmetical (alu) */
41  #define SHFT_SRA 6      /* Shifter <- shiftrightarithmetical (alu) */
42  #define SHFT_NIL 7      /* Shifter <- ALU */
43  #define SHFT_X SHFT_NIL /* don't care */
44
45  /* encoding for result destination */
46  #define DST_R0  0       /* R0  <- Shifter */
47  #define DST_R1  1       /* R1  <- Shifter */
48  #define DST_R2  2       /* R2  <- Shifter */
49  #define DST_R3  3       /* R3  <- Shifter */
50  #define DST_R4  4       /* R4  <- Shifter */
51  #define DST_R5  5       /* R5  <- Shifter */
52  #define DST_R6  6       /* R6  <- Shifter */
53  #define DST_R7  7       /* R7  <- Shifter */
54  #define DST_ACC 8       /* IR  <- Shifter */
55  #define DST_NIL 15      /* not connected <- shifter */
56  #define DST_X   DST_NIL /* don't care instruction */
57
58  /* encoding for command field */
59  #define NXT_NXT  0     /* CSAR <- CSAR + 1 */
60  #define NXT_JMP  1     /* CSAR <- Address */
61  #define NXT_JC   2     /* CSAR <- (carry==1)? Address : CSAR + 1 */
62  #define NXT_JNC 10     /* CSAR <- (carry==0)? Address : CSAR + 1 */
63  #define NXT_JZ   4     /* CSAR <- (zero==1) ? Address : CSAR + 1 */
64  #define NXT_JNZ 12     /* CSAR <- (zero==0) ? Address : CSAR + 1 */
65  #define NXT_X NXT_NXT
66
67  /* encoding for the microinstruction word */
68  #define MI(OUT, SBUS, ALU, SHFT, DEST, NXT, ADR) \
69      (OUT   << 31) | \
70      (SBUS  << 27) | \
```

```
71        (ALU   << 23) | \
72        (SHFT  << 20) | \
73        (DEST  << 16) | \
74        (NXT   << 12) | \
75        (ADR)
76
77   dp control(in  carry, zero : ns(1);
78             out ctl_ot      : ns(1);
79             out ctl_sbus    : ns(4);
80             out ctl_alu     : ns(4);
81             out ctl_shft    : ns(3);
82             out ctl_dest    : ns(4)) {
83
84      lookup cstore : ns(32) = {
85        // 0 Lstart: IN -> R0
86        MI(O_NIL, SBUS_IN, ALU_PASS, SHFT_NIL, DST_R0,  NXT_NXT, 0),
87        // 1          IN -> ACC
88        MI(O_NIL, SBUS_IN, ALU_PASS, SHFT_NIL, DST_ACC, NXT_NXT, 0),
89        // 2 Lcheck: (R0 - ACC)       || JUMP_IF_Z Ldone
90        MI(O_NIL, SBUS_R0, ALU_SUBS, SHFT_NIL, DST_NIL, NXT_JZ,  6),
91        // 3         (R0 - ACC) << 1  || JUMP_IF_C LSmall
92        MI(O_NIL, SBUS_R0, ALU_SUBS, SHFT_SHL, DST_NIL, NXT_JC,  5),
93        // 4         R0 - ACC -> R0   || JUMP Lcheck
94        MI(O_NIL, SBUS_R0, ALU_SUBS, SHFT_NIL, DST_R0,  NXT_JMP, 2),
95        // 5 Lsmall: ACC - R0 -> ACC  || JUMP Lcheck
96        MI(O_NIL, SBUS_R0, ALU_SUBA, SHFT_NIL, DST_ACC, NXT_JMP, 2),
97        // 6 Ldone:  R0 -> OUT        || JUMP Lstart
98        MI(O_WR,  SBUS_R0, ALU_X,    SHFT_X,   DST_X,   NXT_JMP, 0)
99      };
100
101      reg csar        : ns(12);
102      sig mir         : ns(32);
103      sig ctl_nxt     : ns(4);
104      sig csar_nxt    : ns(12);
105      sig ctl_address : ns(12);
106
107      always {
108
109        mir = cstore(csar);
110        ctl_ot      = mir[31];
111        ctl_sbus    = mir[27:30];
112        ctl_alu     = mir[23:26];
113        ctl_shft    = mir[20:22];
114        ctl_dest    = mir[16:19];
115        ctl_nxt     = mir[12:15];
116        ctl_address = mir[ 0:11];
117
118        csar_nxt = csar + 1;
119        csar = (ctl_nxt == NXT_NXT)  ? csar_nxt :
120               (ctl_nxt == NXT_JMP)  ? ctl_address :
121               (ctl_nxt == NXT_JC)   ? ((carry==1) ? ctl_address : csar_nxt) :
122               (ctl_nxt == NXT_JZ)   ? ((zero==1) ? ctl_address : csar_nxt) :
123               (ctl_nxt == NXT_JNC)  ? ((carry==0) ? ctl_address : csar_nxt) :
124               (ctl_nxt == NXT_JNZ)  ? ((zero==0) ? ctl_address : csar_nxt) :
```

```
125                csar;
126        }
127    }
128
129    dp regfile (in  ctl_dest : ns(4);
130                 in  ctl_sbus : ns(4);
131                 in  data_in  : ns(WLEN);
132                 out data_out : ns(WLEN)) {
133      reg r0 : ns(WLEN);
134      reg r1 : ns(WLEN);
135      reg r2 : ns(WLEN);
136      reg r3 : ns(WLEN);
137      reg r4 : ns(WLEN);
138      reg r5 : ns(WLEN);
139      reg r6 : ns(WLEN);
140      reg r7 : ns(WLEN);
141      always {
142        r0 = (ctl_dest == DST_R0) ? data_in : r0;
143        r1 = (ctl_dest == DST_R1) ? data_in : r1;
144        r2 = (ctl_dest == DST_R2) ? data_in : r2;
145        r3 = (ctl_dest == DST_R3) ? data_in : r3;
146        r4 = (ctl_dest == DST_R4) ? data_in : r4;
147        r5 = (ctl_dest == DST_R5) ? data_in : r5;
148        r6 = (ctl_dest == DST_R6) ? data_in : r6;
149        r7 = (ctl_dest == DST_R7) ? data_in : r7;
150        data_out = (ctl_sbus == SBUS_R0) ? r0 :
151                   (ctl_sbus == SBUS_R1) ? r1 :
152                   (ctl_sbus == SBUS_R2) ? r2 :
153                   (ctl_sbus == SBUS_R3) ? r3 :
154                   (ctl_sbus == SBUS_R4) ? r4 :
155                   (ctl_sbus == SBUS_R5) ? r5 :
156                   (ctl_sbus == SBUS_R6) ? r6 :
157                   (ctl_sbus == SBUS_R7) ? r7 :
158                   r0;
159      }
160    }
161
162    dp alu (in  ctl_dest : ns(4);
163            in  ctl_alu  : ns(4);
164            in  sbus     : ns(WLEN);
165            in  shift    : ns(WLEN);
166            out q        : ns(WLEN)) {
167      reg acc : ns(WLEN);
168      always {
169        q = (ctl_alu == ALU_ACC)  ? acc :
170            (ctl_alu == ALU_PASS) ? sbus :
171            (ctl_alu == ALU_ADD)  ? acc + sbus :
172            (ctl_alu == ALU_SUBA) ? acc - sbus :
173            (ctl_alu == ALU_SUBS) ? sbus - acc :
174            (ctl_alu == ALU_AND)  ? acc & sbus :
175            (ctl_alu == ALU_OR)   ? acc | sbus :
176            (ctl_alu == ALU_NOT)  ? ~ sbus      :
177            (ctl_alu == ALU_INCS) ? sbus + 1   :
178            (ctl_alu == ALU_INCA) ? acc + 1    :
```

```
179            (ctl_alu == ALU_CLR)  ? 0          :
180            (ctl_alu == ALU_SET)  ? 1          :
181            0;
182       acc = (ctl_dest == DST_ACC) ? shift : acc;
183     }
184   }
185
186   dp shifter(in  ctl     : ns(3);
187              out zero    : ns(1);
188              out cy      : ns(1);
189              in  shft_in : ns(WLEN);
190              out so      : ns(WLEN)) {
191     always {
192       so = (ctl == SHFT_NIL) ? shft_in :
193            (ctl == SHFT_SHL) ? (ns(WLEN)) (shft_in << 1) :
194            (ctl == SHFT_SHR) ? (ns(WLEN)) (shft_in >> 1) :
195            (ctl == SHFT_ROL) ? (ns(WLEN)) (shft_in # shft_in [WLEN-1]) :
196            (ctl == SHFT_ROR) ? (ns(WLEN)) (shft_in[0] # (shft_in >> 1)):
197            (ctl == SHFT_SLA) ? (ns(WLEN)) (shft_in << 1) :
198            (ctl == SHFT_SRA) ? (ns(WLEN)) (((tc(WLEN)) shft_in) >> 1) :
199            0;
200       zero = (shft_out == 0);
201       cy  = (ctl == SHFT_NIL) ? 0 :
202            (ctl == SHFT_SHL) ? shft_in[WLEN-1] :
203            (ctl == SHFT_SHR) ? 0 :
204            (ctl == SHFT_ROL) ? shft_in[WLEN-1] :
205            (ctl == SHFT_ROR) ? shft_in[0] :
206            (ctl == SHFT_SLA) ? shft_in[WLEN-1] :
207            (ctl == SHFT_SRA) ? 0 :
208            0;
209     }
210   }
211
212   dp hmm(in  din  : ns(WLEN); out din_strb  : ns(1);
213          out dout : ns(WLEN); out dout_strb : ns(1)) {
214     sig carry, zero : ns(1);
215     sig ctl_ot      : ns(1);
216     sig ctl_sbus    : ns(4);
217     sig ctl_alu     : ns(4);
218     sig ctl_shft    : ns(3);
219     sig ctl_acc     : ns(1);
220     sig ctl_dest    : ns(4);
221
222     sig rf_out, rf_in : ns(WLEN);
223     sig sbus          : ns(WLEN);
224     sig alu_in        : ns(WLEN);
225     sig alu_out       : ns(WLEN);
226     sig shft_in       : ns(WLEN);
227     sig shft_out      : ns(WLEN);
228
229     use control(carry,    zero,
230                 ctl_ot,   ctl_sbus, ctl_alu,  ctl_shft, ctl_dest);
231     use regfile(ctl_dest, ctl_sbus, rf_in,    rf_out);
232     use alu    (ctl_dest, ctl_alu,  sbus,     alu_in,  alu_out);
```

```
233    use shifter(ctl_shft, zero,       carry,     shft_in,  shft_out);
234
235    always {
236     sbus      = (ctl_sbus == SBUS_IN) ? din  : rf_out;
237     din_strb  = (ctl_sbus == SBUS_IN) ? 1    : 0;
238     dout      = sbus;
239     dout_strb = (ctl_ot == O_WR) ? 1 : 0;
240     rf_in     = shft_out;
241     alu_in    = shft_out;
242     shft_in   = alu_out;
243    }
244  }
245
246  dp hmmtest {
247    sig din        : ns(WLEN);
248    sig din_strb   : ns(1);
249    sig dout       : ns(WLEN);
250    sig dout_strb  : ns(1);
251    use hmm(din, din_strb, dout, dout_strb);
252
253    reg dcnt       : ns(5);
254    lookup stim    : ns(WLEN) = { 14, 32, 87, 12, 23, 99, 32, 22};
255
256    always {
257      dcnt = (din_strb) ? dcnt + 1 : dcnt;
258      din  = stim(dcnt & 7);
259      $display($cycle, " IO ", din_strb, " ", dout_strb, " ", $dec,
260                din, " ", dout);
261    }
262  }
263
264  system S {
265   hmmtest;
266  }
```

This design can be simulated with the `fdlsim` GEZEL simulator. Because of the C macros included in the source, the program first needs to be processed using the C preprocessor. The following command line illustrates how to simulate the first 100 cycles of this design.

```
>cpp -P hmm2.fdl | fdlsim 100
```

The first few lines of output look as follows:

```
0 IO 1 0 14 14
1 IO 1 0 32 32
2 IO 0 0 87 14
3 IO 0 0 87 14
4 IO 0 0 87 14
...
```

The microprogrammed machine reads the numbers 14 and 32 in clock cycle 0 and 1, respectively, and starts the GCD calculation. To find the corresponding GCD, we look for a "1" in the fourth column (output strobe). Around cycle 21, the first one

appears. We can find that GCD(32,14)=2. Note that the testbench proceeds with the next two inputs in cycle 23 and 24.

```
18 IO 0 0 87 2
19 IO 0 0 87 2
20 IO 0 0 87 2
21 IO 0 1 87 2
22 IO 1 0 87 87
23 IO 1 0 12 12
24 IO 0 0 23 87
```

A quick command to filter out the valid outputs during simulation is the following.

```
> cpp -P hmm2.fdl | fdlsim 200| awk ''{if ($4 == "1") print $0}''
21 IO 0 1 87 2
55 IO 0 1 23 3
92 IO 0 1 32 1
117 IO 0 1 14 2
139 IO 0 1 87 2
173 IO 0 1 23 3
```

The above design illustrates how the FSMD model can be applied to create a more complex microprogrammed machine. In the following, we show how this can be used to create programming concepts at even higher levels of abstraction, using microprogram interpreters.

5.6 Microprogram Interpreters

A microprogram is a highly-optimized sequence of commands for a datapath. This sequence of register transfers is optimized for parallelism. Writing efficient microprograms is not so easy because it requires an in-depth understanding of the machine architecture. An obvious question is if a programming language, such as a pseudo-assembly language, would be of help in developing microprograms. Certainly, the writing process itself could be made more convenient. Tools can generate the content of the control store automatically. On the other hand, if we want to keep all the parallelism in the machine visible, the resulting programming language needs to have a low abstraction level.

A common usage of microprograms is therefore not to encode complete applications, but instead to serve as *interpreters* for other programs. An interpreter is a machine that decodes and executes instruction sequences of an abstract high-level machine, which we will call the macro-machine. The instructions from the macro-machine will be implemented in terms of microprograms for a microprogrammed machine. Such a construct is illustrated in Fig. 5.9, and is called a *microprogram interpreter*. We create a microprogram in the form of an infinite loop, which reads a macro-instruction byte and breaks down a byte in opcode and operand fields. It then takes specific actions depending on the values of the opcode.

Fig. 5.9 A microprogram
interpreter implements a more
abstract language
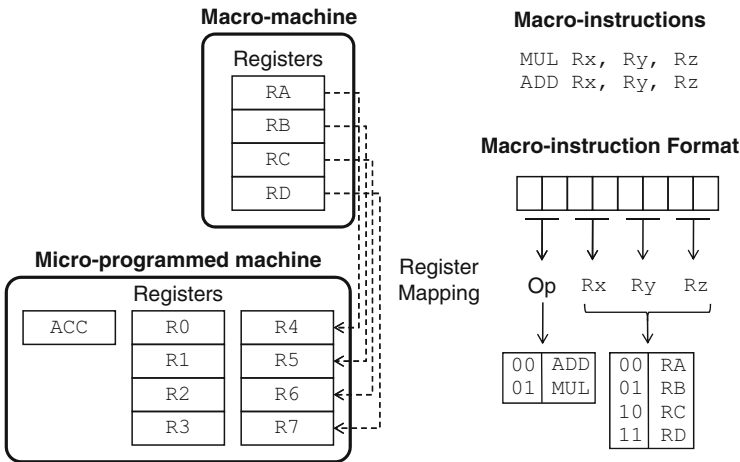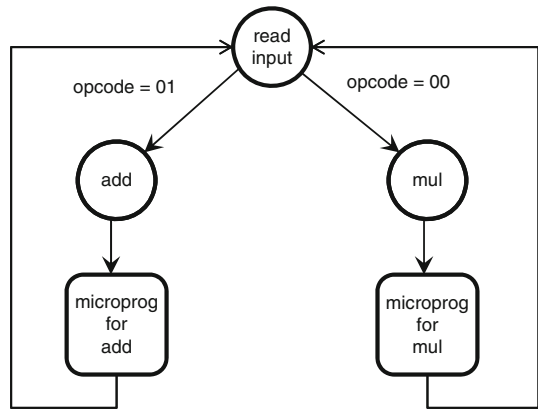
Fig. 5.10 Programmer's model for the macro-machine example

We will discuss, by means of an example, how such a macro-machine can be
implemented. Figure 5.10 shows the programmers' model of the macro-machine.
It is a very simple machine, with four registers RA through RD, and two instruc-
tions for adding and multiplying those registers. The macro-machine will have
the same wordlength as the microprogrammed machine, but it has fewer register
than the original microprogrammed machine. To implement the macro-machine,
we will map the macro-register set directly onto the micro-register set. In this
case, we will map register RA to RD onto register R4 to R7, respectively. This
leaves register R0 to R3, as well as the accumulator, available to implement macro-
instructions. The macro-machine has two instructions: add and mul. Each of
these instructions takes two source operands and generates one destination operand.
The operands are macro-machine registers. Because the micro-machine has to

decode the macro-instructions, we also need to choose the instruction-encoding of the macro-instructions. This is illustrated on the right of Fig. 5.10. Each macro-instruction is a single byte, with two bits for the macroopcode, and two bits for each of the macro-instruction operands.

Listing 5.3 shows a sample implementation for each of the ADD and MUL instructions. We have assumed that single-level subroutines are supported at the level of the micro-machine. See Problem 5.3 how such a subroutine call can be implemented in the microprogrammed machine.

The microinterpreter loop, on line 21–29, reads one macro-instruction from the input, and determines the macro-instruction opcode with a couple of shift instructions. Depending on the value of the opcode field, the microprogram will then jump to a routine to implement the appropriate macro-instruction, add or mul.

The implementation of ADD is shown in lines 35–39. The microinstructions use fixed source operands and a fixed destination operand. Since the macro-instructions can use one of four possible operand registers, an additional register-move operation is needed to prepare the microinstruction operands. This is done by the putarg and getarg subroutines, starting on line 62. The getarg subroutine copies data from the macro-machine source registers (RA through RD) to the micro-machine source working registers (R1 and R2). The putarg subroutine moves data from the micro-machine destination working register R1 back to the destination macro-machine register (RA through RD).

The implementation of the add instruction starts on line 35. At the start of this section of code, the accumulator contains the macro-instruction. The accumulator value is passed on to the getarg routine, which decodes the two source operand registers and copies them into micro-machine register R1 and R2. Next, the add macro-instruction performs the addition and stores the result in R1 (line 36–39). The putarg and getarg routines assume that the opcode of the macro-instruction is stored in the accumulator. Since the body of the add instruction changes the accumulator, it needs to be preserved before putarg is called. This is the purpose of the register-copy instructions on lines 36 and 39.

The implementation of the mul macro-instruction starts on line 46 and follows the same principles as the add instruction. In this case, the body of the instruction is more complex as the multiply operation needs to be performed using an add-and-shift loop. A loop counter is created in register R3 to perform 8 iterations of add-and-shift. Because the accumulator register is only 8 bit, the multiply instruction cannot capture all 16 output bits of an 8-by-8 bit multiply. The implementation of mul preserves only the least significant byte.

Listing 5.3 Implementation of the macro-instructions ADD and MUL

```
1  //-------------------------------------------------
2  // Macro-machine for the instructions
3  //
4  //     ADD Rx, Ry, Rz
5  //     MUL Rx, Ry, Rz
6  //
7  //     Macro-instruction encoding:
8  //     +----+----+----+----+
```

```
 9  //      | ii + Rx + Ry + Rz +
10  //      +----+----+----+----+
11  //
12  //      where ii = 00 for ADD
13  //                01 for MUL
14  //      where Rx, Ry and Rz are encoded as follows:
15  //                00 for RA (mapped to R4)
16  //                01 for RB (mapped to R5)
17  //                10 for RC (mapped to R6)
18  //                11 for RD (mapped to R7)
19  //
20  // Interpreter loop reads instructions from input
21  macro:    IN -> ACC
22            (ACC & 0xC0) >> 1 -> R0
23            R0 >> 1 -> R0
24            R0 >> 1 -> R0
25            R0 >> 1 -> R0
26            R0 >> 1 -> R0
27            R0 >> 1 -> R0            || JUMP_IF_NZ mul
28            (no_op)                  || JUMP add
29  macro_done: (no_op)                || JUMP macro
30
31  //-------------------------------------------------
32  //
33  // Rx = Ry + Rz
34  //
35  add:      (no_op)                  || CALL getarg
36            ACC -> R0
37            R2  -> ACC
38            (R1 + ACC) -> R1
39            R0 -> ACC                || CALL putarg
40            (no_op)                  || JUMP macro_done
41
42  //-------------------------------------------------
43  //
44  // Rx = Ry * Rz
45  //
46  mul:      (no_op)                  || CALL getarg
47            ACC -> R0
48            0   -> ACC
49            8   -> R3
50  loopmul:  (R1 << 1) -> R1          || JUMP_IF_NC nopartial
51            (ACC << 1) -> ACC
52            (R2 + ACC) -> ACC
53  nopartial: (R3 - 1) -> R3          || JUMP_IF_NZ loopmul
54            ACC -> R1
55            R0 -> ACC                || CALL putarg
56            (no_op)                  || JUMP macro_done
57
58  //-------------------------------------------------
59  //
60  // GETARG
61  //
62  getarg:   (ACC & 0x03) -> R0       || JUMP_IF_Z Rz_is_R4
```

```
63              (R0 - 0x1)                   || JUMP_IF_Z Rz_is_R5
64              (R0 - 0x2)                   || JUMP_IF_Z Rz_is_R6
65  Rz_is_R7:   R7 -> R1                     || JUMP get_Ry
66  Rz_is_R6:   R6 -> R1                     || JUMP get_Ry
67  Rz_is_R5:   R5 -> R1                     || JUMP get_Ry
68  Rz_is_R4:   R4 -> R1                     || JUMP get_Ry
69  get_Ry:     (ACC & 0x0C) >> 1 -> R0
70              R0 >> 1 -> R0                || JUMP_IF_Z Ry_is_R4
71              (R0 - 0x1)                   || JUMP_IF_Z Ry_is_R5
72              (R0 - 0x2)                   || JUMP_IF_Z Ry_is_R6
73  Ry_is_R7:   R7 -> R2                     || RETURN
74  Ry_is_R6:   R6 -> R2                     || RETURN
75  Ry_is_R5:   R5 -> R2                     || RETURN
76  Ry_is_R4:   R4 -> R2                     || RETURN
77
78  //-----------------------------------------------
79  //
80  // PUTARG
81  //
82  putarg:     (ACC & 0x30) >> 1  -> R0
83              R0 >> 1 -> R0
84              R0 >> 1 -> R0
85              R0 >> 1 -> R0                || JUMP_IF_Z Rx_is_R4
86              (R0 - 0x1)                   || JUMP_IF_Z Rx_is_R5
87              (R0 - 0x2)                   || JUMP_IF_Z Rx_is_R6
88  Rx_is_R7:   R1 -> R7                     || RETURN
89  Rx_is_R6:   R1 -> R6                     || RETURN
90  Rx_is_R5:   R1 -> R5                     || RETURN
91  Rx_is_R4:   R1 -> R4                     || RETURN
```

A microprogrammed interpreter can create the illusion of a machine that has more powerful instructions than the original microprogrammed architecture. The trade-off made by such an interpreter is that of abstraction versus performance: each instruction of the macro-machine may need many micro-machine instructions. The concept of microprogram interpreters has been used extensively to design processors with configurable instruction sets and was originally used to enhance the flexibility of expensive hardware. Today, the technique of microprogram interpreter design is still very useful to create an additional level of abstraction on top of a micropro-grammed architecture.

5.7 Microprogram Pipelining

As can be observed from Fig. 5.11, the microprogram controller may be part of a long chain of combinational logic. Pipeline registers can be used to break these long chains. However, the introduction of pipeline registers has a large impact on the design of microprograms. In this section, we will study these effects in more detail.
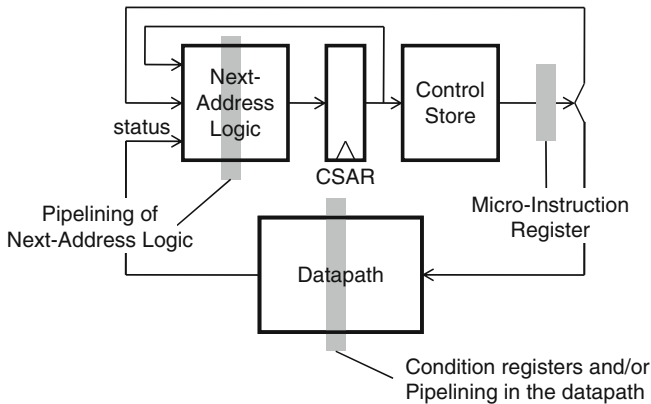
Fig. 5.11 Typical placement of pipeline registers in a microprogram interpreter

First, observe in Fig. 5.11 that the CSAR register is part of possibly three loops with logic. The first loop runs through the next-address logic. The second loop runs through the control store and the next-address logic. The third loop runs through the control store, the data path, and the next-address logic. These combinational paths may limit the maximum clock frequency of the microprogrammed machine. There are three common places where additional pipeline registers may be inserted in the design of this machine, and they are marked with shaded boxes in Fig. 5.11.

- A common location to insert a pipeline register is at the output of the control store. The register at that location is called the microinstruction register. Inserting a register there allows overlap of the datapath evaluation, the next address evaluation, and the microinstruction fetch.
- Another location for pipeline registers is the datapath. Besides pipeline register inside of the data path, additional condition-code registers can be placed at the datapath outputs.
- Finally, the next-address logic may be pipelined as well, in case high-speed operation is required and the target CSAR address cannot be evaluated within a single clock cycle.

5.7.1 Microinstruction Register

Because each of these registers cuts through a different update-loop of the CSAR register, and each of them has a different effect on the microprograms, let us first consider the effect of adding the microinstruction register. Because of this register, the microinstruction fetch (i.e., addressing the CSTORE and retrieving the next

Table 5.2 Effect of the microinstruction register on jump instructions

Cycle	CSAR	Microinstruction register
N	4	
$N + 1$	5	CSTORE(4) = JUMP 10
$N + 2$	10	CSTORE(5) *need to cancel*
$N + 3$	11	CSTORE(10) *execute*

Table 5.3 Effect of the microinstruction register and condition-code register on conditional jump instructions

Cycle	CSAR	Microinstruction register
N	3	
$N + 1$	4	CSTORE(3) = TEST R0 sets Z-flag
$N + 2$	5	CSTORE(4) = JZ 10
$N + 3$	10	CSTORE(5) *need to cancel*
$N + 4$	11	CSTORE(10) *execute*

microinstruction) is offset by one cycle from the evaluation of that microinstruction. For example, when the CSAR is fetching instruction i from a sequence of instructions, the datapath and next-address logic will be executing instruction $i - 1$.

Table 5.2 illustrates the effect of this offset on the instruction stream, when that stream contains a jump instruction. The microprogrammer entered a JUMP 10 instruction in CSTORE location 4, and that instruction will be fetched in clock cycle N. In clock cycle $N + 1$, the microinstruction will appear at the output of the microinstruction register. The execution of that instruction will complete in cycle $N + 2$. For a JUMP, this means that the value of CSAR will be affected in cycle $N + 2$. As a result, a JUMP instruction cannot modify the value of CSAR within a single clock cycle. If CSTORE(4) contains a JUMP, then the instruction located in CSTORE(5) will be fetched as well. The microprogrammer needs to be aware of this. The possible strategies are (a) take into account that a JUMP will be executed with one cycle of delay (so-called "delayed branch") or (b) include support in the microprogrammed machine to cancel the execution of an instruction in case of a jump.

5.7.2 *Datapath Condition-Code Register*

As a second case, let us assume that we have a condition-code register in the datapath, in addition to a microinstruction register. The result of a condition code register is that a condition value will only be available 1 clock cycle after the expression leading to that condition was evaluated on the datapath. As a result, a conditional-jump instruction can only operate on datapath conditions from the previous clock cycle. Table 5.3 illustrates this effect. The branch instruction in CSTORE(4) is a conditional jump. When the condition is true, the jump will be executed with 1 clock cycle delay, as was discussed before. However, the JZ is evaluated in cycle $N + 2$

Table 5.4 Effect of additional pipeline registers in the CSAR update loop

Cycle	CSAR_pipe	CSAR	Microinstruction register
0	0	0	CSTORE(0)
1	1	0	CSTORE(0) twice ?
2	1	1	CSTORE(1)
3	2	1	CSTORE(1) twice ?

based on a condition code generated in cycle $N + 1$. Thus, the microprogrammer needs to be aware that conditions need to be available 1 clock cycle before they will be used in conditional jumps.

5.7.3 Pipelined Next-Address Logic

Finally, let us assume that there is a third level of pipelining available inside of the next-address update loop. For simplicity, we will assume there are two CSAR registers back-to-back in the next-address loop. The output of the next-address-logic is fed into a register CSAR_pipe, and the output of CSAR_pipe is connected to CSAR. Table 5.4 shows the operation of this microprogrammed machine, assuming all registers are initially zero. As shown in the table, the two CSAR registers in the next-address loop result in two (independent) address sequences. When all registers start out at 0, then each instruction of the microprogram will be executed twice. Solving this problem is not easy. While one can do a careful initialization of CSAR_pipe and CSAR such that they start out at different values (e.g., 1 and 0), this reinitialization will need to be done at each jump instruction. This makes the design and the programming of pipelined next-address logic very hard.

   The previous three examples show that a microprogrammer must be aware of the implementation details of the microarchitecture, and in particular of all the delay effects caused by registers. This can significantly increase the complexity of the development of microprograms.

5.8 Picoblaze: A Contemporary Microprogram Controller

Although microprogramming originated many years ago, its ideas are still very useful. When complex systems are created in hardware, the design of an adequate control architecture is often a key problem. In this section, we illustrate a possible solution based on the use of a microcontroller.

   Most FPGA companies now offer small programmable, synthesizable controllers. This includes for example, Picoblaze (Xilinx) or Mico8 (Lattice Semiconductor). These controllers have only minimal computational capabilities, such as an ALU with basic logical and arithmetic operations. However, they do implement an instruction-fetch engine, and as such as they are well suited as controllers for
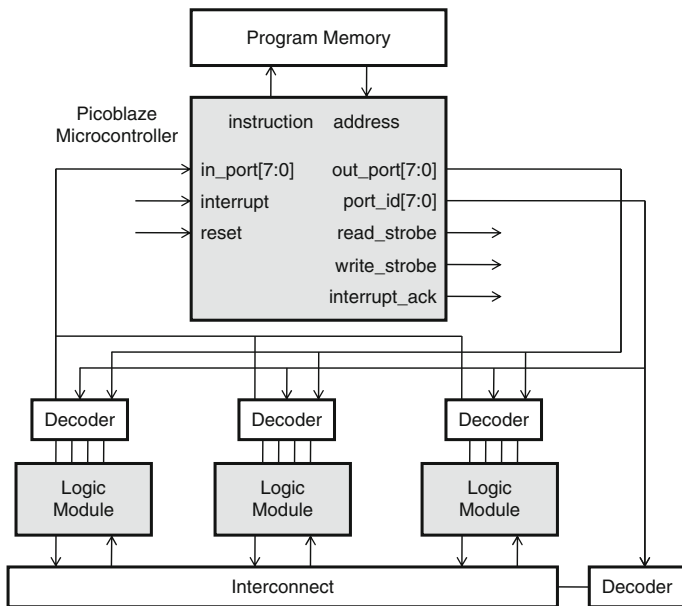
Fig. 5.12 Using a Picoblaze controller as a microprogram sequencer

larger circuits. They also come with a pseudoassembly instruction-set, that allows for easy design of control programs. For these reasons, these small controllers are well suited as replacement for finite machines.

In this section, we will discuss another use of these controllers – namely using them as microprogram controllers. Figure 5.12 shows an example of a micropro-grammed architecture based on Picoblaze. The Picoblaze controller is an 8-bit architecture with an internal program memory. The controller has several additional ports that are helpful to use this module as a system controller.

- An 8-bit input port `in_port` can read in 8-bit values.
- An 8-bit output port `out_port` can produce 8-bit values.
- A port identifier `port_id` carries a port address. This allows the picoblaze con-troller to distinguish 256 multiplexed input and output ports.
- A `read_strobe` and `write_strobe` synchronizes input/output operations on the I/O ports.
- Additional control lines will initialize the controller (`reset`) or will handle in-terrupts (`interrupt`, `interrupt_ack`).

The Picoblaze controller has several instructions to communicate through the I/O ports.

```
OUTPUT   sX, sY; // write contents of reg sX to port ID reg sY
OUTPUT   sX, kk; // write contents of reg sX to port ID const kk
INPUT    sX, sY; // read contents of port ID reg sY into reg sX
INPUT    sX, kk; // read contents of port ID const kk into reg sX
```

In the example of Fig. 5.12, the I/O ports are used to control several datapath submodules. Combining the port address and the port data, we can communicate up to 16 bits of data per Picoblaze instruction to the datapath submodules. Thus, we will have to use a verticallyencoded microinstruction to accommodate a large number of logic modules. For this reason, there are decoders on top of each logic module in Fig. 5.12. There can be up to 8 bits of status information from the logic modules back to the Picoblaze controller. If there are more than 8 status flags over the entire group of logic modules, additional decoding needs to be introduced based on the port identifier. An important characteristic of the architecture in Fig. 5.12 is the dedicated interconnect, which enables one logic module to be directly connected to the next one. This dedicated interconnect is, strictly speaking, not needed: a designer could simply use the Picoblaze I/O instructions to implement communication between datapath modules. However, this practice will almost certainly turn the controller into a bottleneck. A better strategy is to start off with a specialized interconnect.

5.9   Summary

In this section we introduced Microprogramming as means to deal with control design problems in hardware. Finite State Machines are good for small, compact specifications, but they do result in a few issues. Finite State Machines cannot easily express hierarchy (FSMs calling other FSMs). As a result, control problems can easily blow up when specified as a finite-state-machine, yielding so-called "state explosion". In a microprogrammed architecture, the hardcoded next-state logic of a finite state machine is replaced with a programmable control store and a program control (called CSAR or Control Store Address Register). This takes care of most problems: microprograms can call other microprograms using jump instructions or using the equivalent of subroutines. Microprograms have a much higher scalability than finite state machines.

Writing microprograms is more difficult than writing assembly code or C code. Therefore, instead of mapping a full application directly into microcode, it may be easier to develop a microprogrammed interpreter. Such an interpreter implements an instruction-set for a language at a higher level of abstraction. Still, even single microinstructions may be hard to write, and in particular the programmer has to be aware of all pipelining and delay effects inside of the microprogrammed architecture.

5.10   Further Reading

The limitations of FSM as a mechanism for control modeling have long been recognized in literature. While it has not been the topic of the chapter, there are several alternate control modeling mechanisms available. A key contribution to

hierarchical modeling of FSM was defined by David Harel in StateCharts (Harel 1987). Additionally, the development of so-called synchronous languages support the specification of control as event-driven programs. See for example Esterel by Berry (2000) and Potop-Butucaru et al. (2007).

A nice introduction and historical review of Microprogramming can be found online on the pages of Smotherman (2009). Most of the work on microprogramming was done in the late 1980s and early 1990s. Conference proceedings and computer-architecture books from that period are an excellent source of design ideas. For example, a extensive description of microprogramming is found in the textbook by Lynch (1993). Control optimization issues of microprogramming are discussed by Davio et al. (1983).

Documentation for the Picoblaze microcontroller, as well as the source code, can be found online (Xilinx 2009).

5.11 Problems

5.1. Figure 5.13 shows a microprogrammed datapath. There are six control bits for the datapath: two bits for each of the multiplexers M1 and M2 and two bits for the ALU. The encoding of the control bits is indicated in the figure.

(a) Develop a horizontal microinstruction encoding for the list of microinstructions shown in Table 5.5.
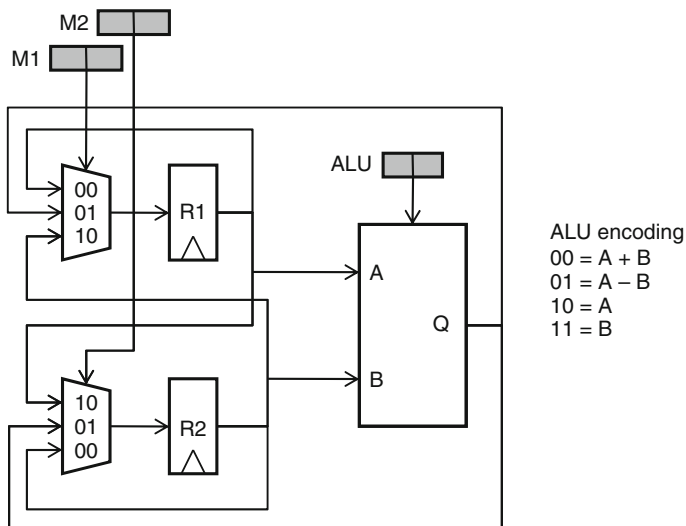
Fig. 5.13 Datapath for Problem 5.1

Table 5.5 Microinstructions for Problem 5.1

SWAP	Interchange the content of R1 and R2.
ADD Rx	Add the contents of R1 and R2 and store the contents in Rx, which is equal to R1 or R2. There are two variants of this instruction depending on Rx.
COPY Rx	Copy the contents of Ry into Rx. (Rx, Ry) is either (R1, R2) or (R2, R1). There are two variants of this instruction depending on Rx.
NOP	Do nothing.

Table 5.6 Next-address
instructions for Problem 5.3

```
NXT        CSAR = CSAR + 1;
JUMP k     CSAR = k;
GOSUB k    RET = CSAR + 1; CSAR = k;
RETURN     CSAR = RET;
SWITCH k   RET = CSAR + 1; CSAR = RET;
```

(b) Develop a vertical microinstruction encoding for the same list of instructions. Use a reasonable encoding that results in a compact and efficient decoder for the datapath.

5.2. Using the microprogrammed machine discussed in Sect. 5.5, create a program that reads in a number from the input and that counts the number of nonzero bits in that number. The resulting bitcount must be stored in register R7.

5.3. Design a next-address instruction decoder based on the set of microinstructions shown in Table 5.6. Design your implementation in GEZEL or Verilog. An example IO definition is shown next. The CSAR has to be 10 bit wide, the width of the Address field and the width of the next-address field must be chosen accordingly.

```
dp nextaddress_decoder(in csar     : ns(10);
                       out address : ns(x);
                       out next    : ns(y)) {
   // ...
}
```

5.4. Figure 5.14 shows the implementation of a next-address decoder. A total of 10 bits from the microinstruction are used to control the next-address logic: a six-bit address field, and four control bits, b0, b1, b2, and b3.

For each of the combinations of control bits shown in Table 5.7, find a good description of the instruction corresponding to the control bit values shown. Don't write generic descriptions (like "CSAR register is incremented by one"), but give a high-level description of the instruction they implement. Use terms that a software programmer can understand.

5.5. Your colleague asks you to evaluate an enhancement for a microprogrammed architecture, as illustrated in Fig. 5.15. The enhancement is to insert a pipeline register just in form of the control store.
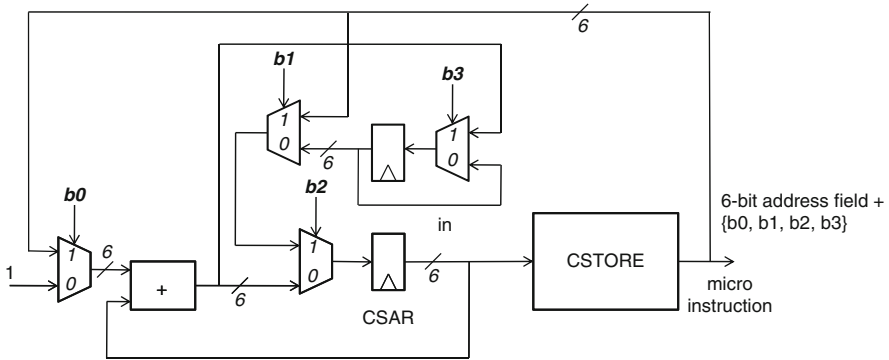
Fig. 5.14 Datapath for Problem 5.4

Table 5.7 Next-address
instructions for Problem 5.4

Combination	b0	b1	b2	b3
Instruction 1	1	X	0	X
Instruction 2	X	1	1	0
Instruction 3	X	1	1	1
Instruction 4	X	0	1	0

Fig. 5.15 Datapath for
Problem 5.5
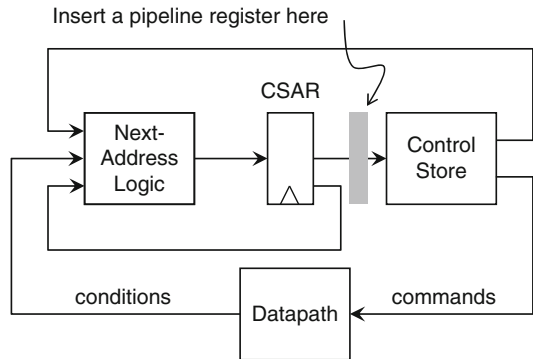
(a) Does this additional register reduce the critical path of the overall architecture?
(b) Your colleague calls this a dual-thread architecture and claims this enhancement
    allows the micro-control engine to run two completely independent programs in
    an interleaved fashion. Do you agree with this or not?

Chapter 6
General-Purpose Embedded Cores

Abstract The most successful programmable component on silicon is the micro-processor. Fueled by a well-balanced mix of efficient implementations, flexibility, and tool support, microprocessors have grown into a key component for electronic design. This chapter reviews the major features of microprocessor architectures, and in particular of RISC (Reduced Instruction Set Computer) processors. The topic of microprocessors is a very broad one; entire books are devoted to its discussion. The objectives of this chapter are more modest. Our objective is to get insight into the relation between a C program and the execution of that C program on a micro-processor. This will help us to understand the cost of the C program in terms of memory footprint and execution time. Later chapters build on this insight to discuss the detailed interaction of C programs with custom-hardware modules. The chapter covers four different aspects of C program execution on RISC processors. First, we will discuss the major architecture elements of a RISC processor and their role in C program execution. Second, we will discuss the path from C programs to assembly programs to machine instructions. Third, we will discuss the runtime organization of a C program at the level of the machine. And finally, we will discuss techniques to evaluate the quality of generated assembly code, and thus evaluate the quality of the C compiler. Much more can be said on microprocessors besides these points; the reader can find additional suggestions in the Further Reading section at the end of this chapter.

6.1 Processors

The most successful programmable component of the past decades is, without doubt, the microprocessor. Nowadays almost any electronic device more compli-cated than a pushbutton is fitted with a microprocessor or a microcontroller. Here are some of the driving factors in that evolution.

- Microprocessors, or the stored-program concept in general, separate software from hardware through the definition of an *instruction-set*. No other hardware development technique has ever been able to uncouple hardware and software in

a similar way. Think for example about microprogramming. Microprograms are really shorthand notations for the control specification of a specialized datapath. The notion of a microprogram as an entity that can be developed independently makes no sense because microinstructions are specialized.

- Microprocessors come with *tools* (compilers and assemblers), that help a designer to create applications. The availability of a compiler to translate a programming language into a program for a microprocessor is an enormous advantage. An embedded software designer can become proficient in a high level programming language – for example C – and move easily between different microprocessor architectures, without having to become an expert in any of those architectures.

- No other device has been able to cope as efficiently with *reuse* as microprocessors did. A general-purpose embedded core is an excellent example of reuse in itself. However, microprocessors have also dictated the rules of reuse in electronic system design in general. They have defined bus interfaces to support the physical integration of an electronic system consisting of multiple components. Their compilers have enabled the development of standard software libraries as well as the logical integration of a system.

- Fourth, no other programmable component has the same *scalability* as a microprocessor. The concept of the stored-program computer has been implemented across a large range of word-lengths (4-bit ... 64-bit). In addition, microprocessors have also extended their reach to entire chips, containing many other components besides a microprocessor, while staying "in command" of the system. This approach, commonly called System-On-Chip (SoC), will be discussed in the next chapter.

In summary, the combination of instruction-set, tools, reuse, and scalability have turned the microprocessor into a dominant component in electronic systems. In fact, hardware/software codesign by itself starts by combining software with dedicated hardware components attached to the microprocessor hardware. The starting point of this chapter is the architecture of the Reduced Instruction Set Computer (RISC) processor, a very common type of processor architecture. From the RISC architecture, the chapter then investigates the relation of a C program with its execution on a microprocessor.

6.1.1 The Toolchain of a Typical Microprocessor

Figure 6.1 illustrates the typical design flow to convert software source code into instructions for a processor. The figure introduces the following terminology used in this chapter. A *compiler* or an *assembler* is used to convert source code into *object code*. Each object code file contains a binary representation of the instructions and data constants corresponding to the source code, along with supporting information to organize these instructions and constants in memory. A *linker* is used to combine several object code files into a single, stand-alone *executable file*. A linker

Fig. 6.1 Standard design flow of software source code to processor instruction
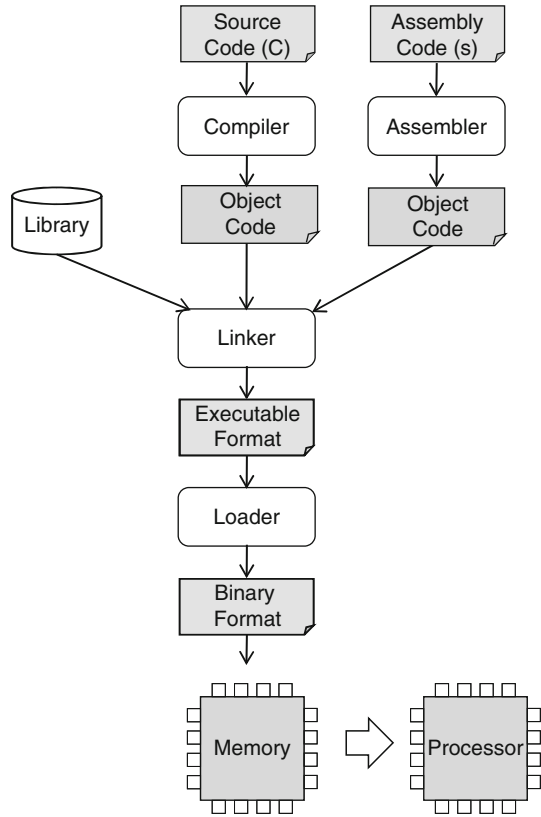
will resolve all unknown elements, such as external data or library routines, while creating the executable file. Finally, a *loader* program determines how the information in an executable file is organized into memory locations. Typically, there will be a part of the memory space reserved for instructions, another part for constant data, another part for global data with read/write access, and so on. A very simple microprocessor system contains at least two elements: the processor and a memory holding instructions for the processor. The memory is initialized with processor instructions by the loader. The processor will then fetch these instructions from memory and execute them on the processor datapath.

6.1.2 *From C to Assembly Instructions*

Let us demonstrate the tool flow explained above through an example. Listing 6.1 shows a C program that evaluates the largest among the common divisors of 5 pairs of numbers. The program illustrates a few interesting features, such as function calls, arrays, and global variables. We will inspect the C program at two lower levels of

Listing 6.1 A C program to find the maximum of greatest common divisors

```
1   int gcd(int a[5], int b[5]) {
2     int i, m, n, max;
3     max = 0;
4     for (i=0; i<5; i++) {
5       m = a[i];
6       n = b[i];
7       while (m != n) {
8         if (m > n)
9           m = m - n;
10        else
11          n = n - m;
12      }
13      if (max > m)
14        max = m;
15    }
16    return max;
17  }
18
19  int a[] = {26, 3,33,56,11};
20  int b[] = {87,12,23,45,17};
21
22  int main() {
23    return gcd(a, b);
24  }
```

abstraction. First, at the level of assembly code generated by the compiler, and next, at the level of the machine code stored in the executable generated by the linker.

In this book, we consider embedded microprocessor architectures such as ARM or Microblaze. We are making use of a *cross-compiler* to generate the executable for these microprocessors. A cross-compiler generates an executable for processor different than the machine used to run the compiler. In this case, we will generate an executable for an ARM processor using a PC workstation. The examples in this book make use of the GNU compiler toolchain. The command to generate the ARM assembly listing is as follows:

```
> /usr/local/arm/bin/arm-linux-gcc -c -S -O2 gcd.c -o gcd.s
```

The command to generate the ARM ELF executable is as follows:

```
> /usr/local/arm/bin/arm-linux-gcc -O2 gcd.c -o gcd
```

Both commands run the same program, `arm-linux-gcc`, but the specific function is selected through the use of command-line flags. Using `man gcc` or `gcc --help` on the command line will list and clarify the available command-line options.

Listing 6.2 is the assembly program generated out of the C program in Listing 6.1. Comparing the assembly program to the C program helps us in understanding low-level implementation details of the C program. Consider the following examples.

Listing 6.2 Assembly dump of Listing 6.1

```
1   gcd:
2           str     lr, [sp, #-4]!
3           mov     lr, #0
4           mov     ip, lr
5   .L13:
6           ldr     r3, [r0, ip, asl #2]
7           ldr     r2, [r1, ip, asl #2]
8           cmp     r3, r2
9           beq     .L17
10  .L11:
11          cmp     r3, r2
12          rsbgt   r3, r2, r3
13          rsble   r2, r3, r2
14          cmp     r3, r2
15          bne     .L11
16  .L17:
17          add     ip, ip, #1
18          cmp     lr, r3
19          movge   lr, r3
20          cmp     ip, #4
21          movgt   r0, lr
22          ldrgt   pc, [sp], #4
23          b       .L13
24  a:
25          .word   26, 3, 33, 56, 11
26  b:
27          .word   87, 12, 23, 45, 17
28  main:
29          str     lr, [sp, #-4]!
30          ldr     r0, .L19
31          ldr     r1, .L19+4
32          ldr     lr, [sp], #4
33          b       gcd
34          .align  2
35  .L19:
36          .word   a
37          .word   b
```

- **Program Structure.** The overall structure of the assembly program preserves the structure of the C program. The gcd function is on lines 1–23, the main function is on lines 28–34. The loop structure of the C program can be identified in the assembly program by inspection of the labels and the corresponding branch instructions. In the gcd function, the inner for loop is on lines 10–15, and the outer while loop is on line 5–23.
- **Storage.** The constant arrays a and b are directly encoded as constants in the assembly, on lines 24–27. The assembly code does not directly work with these constant arrays, but instead with *pointer* to these arrays. The storage location at label .L19 will hold a pointer to array a followed by a pointer to array b.
- **Function Calls.** Function calls in assembly code need to handle the same semantics as a C function call. The function call gcd(a,b) has two parameters

which needs to be passed from `main` to `gcd`. Lines 30–32 of the assembly program show how this C function call is implemented. The assembly program copies the starting address of these arrays into `r0` and `r1`. The `gcd` function in the assembly can make use of `r0` and `r1` as a pointer to array `a` and `b`, respectively.

The assembly program is the starting point to study the implementation details of software on a microprocessor. Further in this chapter, we will also discuss other implementation issues, such as handling of local variables, data types, memory allocation, and compiler optimizations.

The microprocessor works with object code, binary opcodes generated out of assembly programs. Compiler tools can recreate the assembly code out of the executable format. Once an executable is available, the following command shows how to retrieve the opcodes for the `gcd` program.

```
> /usr/local/arm/bin/arm-linux-objdump -d gcd
```

Listing 6.3 shows the object code dump for the `gcd` program. The instructions are mapped to *sections* of memory, and the `.text` section holds the instructions of the program. Each function as a particular starting address, measured to the start of the executable. In this case, the `gcd` function starts at `0x8380` and the `main` functions starts at `0x83cc`. Listing 6.3 also shows the *opcode* of each instruction, the binary representation of instructions handled by the microprocessor. As part of generating the executable, the address value of each label is added to each instruction. For example, the `b  .L13` instruction on line 23 of Listing 6.2 is encoded as a branch to address `0x838c` on line 22 of Listing 6.3.

When we study the low-level behavior of a microprocessor, we investigate how instructions execute on the microprocessor. We can use the assembly mnemonic (such as `cmp`, `mov`, and so on) to describe the instruction. Such mnemonics are just shorthand notations for the opcodes stored in memory.

6.1.3 Simulating a C Program Executing on a Microprocessor

Before going into the details of microprocessors, we briefly explain how to simulate an embedded microprocessor on a standard workstation. Such simulations are very common in hardware–software codesign; they are meant to test the executables created with a cross-compiler, and to evaluate the performance of the resulting program. Microprocessors such as ARM can be simulated with an instruction-set simulator, a simulation engine specialized at simulating the instruction-set for a particular microprocessor. The GEZEL cosimulation environment integrates several instruction-simulation engines, including one for the ARM processor, one for the 8051 microcontroller and one for the picoblaze microcontroller. These simulation engines are open-source software projects. SimIt-ARM was developed by Wei Qin, the Dalton 8051 simulator was developed by the team of Frank Vahid, and the Picoblaze simulator was developed by Mark Six.

Listing 6.3 Object dump of Listing 6.2

```
1  Disassembly of section .text:
2
3  00008380 <gcd>:
4      8380:   e52de004    str   lr, [sp, -#4]!
5      8384:   e3a0e000    mov   lr, #0   ; 0x0
6      8388:   e1a0c00e    mov   ip, lr
7      838c:   e790310c    ldr   r3, [r0, ip, lsl #2]
8      8390:   e791210c    ldr   r2, [r1, ip, lsl #2]
9      8394:   e1530002    cmp   r3, r2
10     8398:   0a000004    beq   83b0 <gcd+0x30>
11     839c:   e1530002    cmp   r3, r2
12     83a0:   c0623003    rsbgt r3, r2, r3
13     83a4:   d0632002    rsble r2, r3, r2
14     83a8:   e1530002    cmp   r3, r2
15     83ac:   1afffffa    bne   839c <gcd+0x1c>
16     83b0:   e28cc001    add   ip, ip, #1   ; 0x1
17     83b4:   e15e0003    cmp   lr, r3
18     83b8:   a1a0e003    movge lr, r3
19     83bc:   e35c0004    cmp   ip, #4   ; 0x4
20     83c0:   c1a0000e    movgt r0, lr
21     83c4:   c49df004    ldrgt pc, [sp], #4
22     83c8:   eaffffef    b     838c <gcd+0xc>
23 000083cc <main>:
24     83cc:   e52de004    str   lr, [sp, -#4]!
25     83d0:   e59f0008    ldr   r0, [pc, #8]     ; 83e0 <main+0x14>
26     83d4:   e59f1008    ldr   r1, [pc, #8]     ; 83e4 <main+0x18>
27     83d8:   e49de004    ldr   lr, [sp], #4
28     83dc:   eaffffe7    b     8380 <gcd>
29     83e0:   00010444    andeq r0, r1, r4, asr #8
30     83e4:   00010458    andeq r0, r1, r8, asr r4
```

Figure 6.2 shows how instruction-set simulators are integrated into the GEZEL cosimulation engine, gplatform. The software part of the application is written in C, and compiled into executable format using a cross compiler. The hardware part of the application is written in GEZEL, and it specifies the platform archi-tecture: the microprocessor, and its interaction with other hardware modules. The combination of the GEZEL program and the cross-compiled executable format is used in a cosimulation. All the instruction-set simulation engines in GEZEL are cycle-accurate simulators; they reflect the behavior of a processor clock-cycle by clock-cycle. Instruction-set simulation engines can also be instruction-accurate rather than cycle-accurate. Such simulators run faster than cycle-accurate simulation engines because they handle less detail.

Listing 6.4 shows a GEZEL program that simulates a stand-alone ARM core that executes the gcd program of Listing 6.1. Lines 1–4 define an ARM core which runs an executable program called gcd. The ipblock is a special type of GEZEL module which represents a black-box simulation model, a simulation model without internal details. This particular module does not have any input/output ports. We will

Fig. 6.2 An instruction-set
simulator, integrated into
GEZEL, can simulate
cross-compiled executables
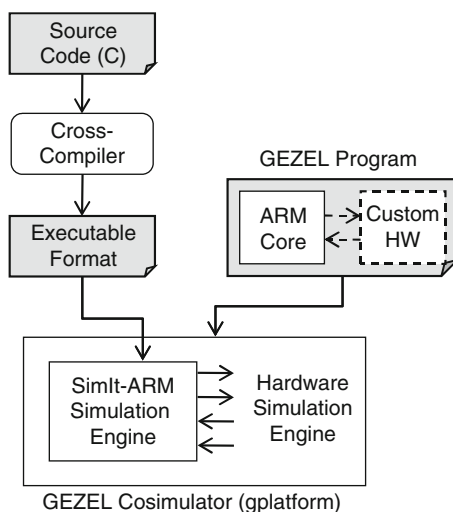
Listing 6.4 A GEZEL top-level module with a single ARM core

```
1  ipblock myarm {
2    iptype "armsystem";
3    ipparm "exec = gcd";
4  }
5
6  dp top {
7    use myarm;
8  }
9
10 system S {
11   top;
12 }
```

introduce such input/output ports while discussing the various hardware/software
interfaces (Chap. 11). Lines 6–12 of the GEZEL program simply configure the
myarm module for execution.

To simulate the program, we will need to cross-compile the C application
software for the ARM instruction-set simulator. Next, we run the instruction-set sim-
ulator. To generate output through the cosimulation, we modified the main function
of the C program as follows:

```
int main() {
  printf("gcd(a,b)=%d\n", gcd(a,b));
  return 0;
}
```

The compilation and cosimulation is now done through the following commands:

```
> /usr/local/arm/bin/arm-linux-gcc -static gcd.c -o gcd
> gplatform top.fdl
```

```
core myarm
armsystem: loading executable [gcd]
gcd(a,b)=3
Total Cycles: 14,338
```

The output of the simulation shows that the program takes 14,338 cycles to execute. In the next chapters, we will look at techniques to analyze this performance and to improve it.

6.2 The RISC Pipeline

This section describes the internal architecture of a very common type of microprocessor, the Reduced Instruction Set Computer (RISC). Our objective is a review of the basic ideas in RISC architecture design, with enough detail to enable us to deal with common hardware/software codesign problems. The material in this section is typically covered in-depth in a computer-architecture course.

In a RISC processor, the execution of a single instruction is split in different stages, which are chained together as a pipeline. Each instruction operates on a set of registers contained within the processor. For example, the ARM processor contains 17 registers: data register r0 to r14, a program counter register pc, and a processor status register cpsr. The Microblaze processor has 32 general-purpose registers (r0 to r31) and up to 18 special-purpose registers (such as the program counter, the status register, and more). Processor registers are used as operands or as targets for the processor instructions.

Each stage of a RISC pipeline takes 1 clock cycle to complete. A typical RISC pipeline has 3 or 5 stages, and Fig. 6.3 illustrates a 5-stage pipeline. The five stages
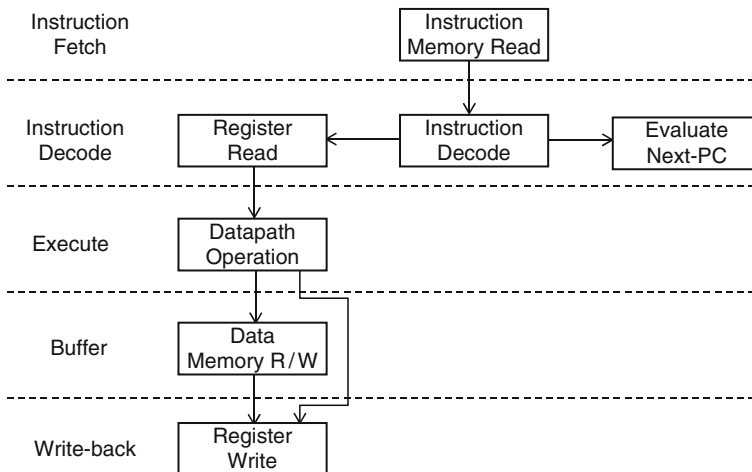
Fig. 6.3 A five-stage RISC pipeline

of the pipeline are called Instruction Fetch, Instruction Decode, Execute, Buffer, and Write-back. As an instruction is executed, each of the stages performs the following activities:

- **Instruction Fetch:** The processor retrieves the instruction addressed by the program counter register from the instruction memory.
- **Instruction Decode:** The processor examines the instruction opcode. For the case of a branch-instruction, the program counter will be updated. For the case of a compute-instruction, the processor will retrieve the processor data registers that are used as operands.
- **Execute:** The processor executes the computational part of the instruction on a datapath. In case the instruction will need to access data memory, the execute stage will prepare the address for the data memory.
- **Buffer:** In this stage, the processor may access the data memory, for reading or for writing. In case the instruction does not need to access data memory, the data will be forwarded to the next pipeline stage.
- **Write Back:** In the final stage of the pipeline, the processor registers are updated.

A 3-stage RISC pipeline is similar to a 5-stage RISC pipeline, but the Execute, Buffer, and Write-back stages are collapsed into a single stage.

Under ideal circumstances, the RISC pipeline is able to accept a new instruction every clock cycle. Thus, the instruction throughput in a RISC processor may be as high as one instruction every cycle. Because of the pipelining, each instruction may take up to 5 clock cycles to complete. The instruction latency therefore can be up to 5 clock cycles. A RISC pipeline improves instruction throughput at the expense of instruction latency. However, the increased instruction latency of a RISC processor is usually not a problem because the clock frequency of a pipelined processor is higher than that of a nonpipelined processor.

In some cases, it is not possible for an instruction to finish within 5 clock cycles. A *pipeline stall* occurs when the progress of instruction through the pipeline is temporarily halted. The cause of such a stall is a *pipeline hazard*. In advanced RISC processors, pipeline *interlock* hardware can detect and resolve pipeline hazards automatically. Even when interlock hardware is present, pipeline hazards may still occur. We discuss three different categories of pipeline hazards, along with examples for an ARMv6 processor. The three categories are the following.

- *Control hazards* are pipeline hazards caused by branches.
- *Data hazards* are pipeline hazards caused by unfulfilled data dependencies.
- *Structural hazards* are caused by resource conflicts and cache misses.

6.2.1 Control Hazards

Branch instructions are the most common form of pipeline stalls. As indicated in Fig. 6.3, a branch is only executed (i.e., it modifies the program counter register) in stage 2 of the pipeline. At that moment, another instruction has already entered the

Program

```
start:  mov    r0, #5
        cmp    r0, #5
        ble    TGT
        mov    r0, #0
        nop
TGT:
        add    r0, #10
```

Cycle	Fetch	Decode	Execute	Buffer	Writeback
0	cmp r0, #5				
1	ble TGT	cmp r0, #5	*interlock*		
2	mov r0, #0	ble TGT	cmp r0, #5		
3	TGT: add r0, #10	mov r0, #0		cmp r0, #5	
4		TGT: add r0, #10	unused		cmp r0, #5
5			TGT: add r0, #10	unused	
6				TGT: add r0, #10	unused
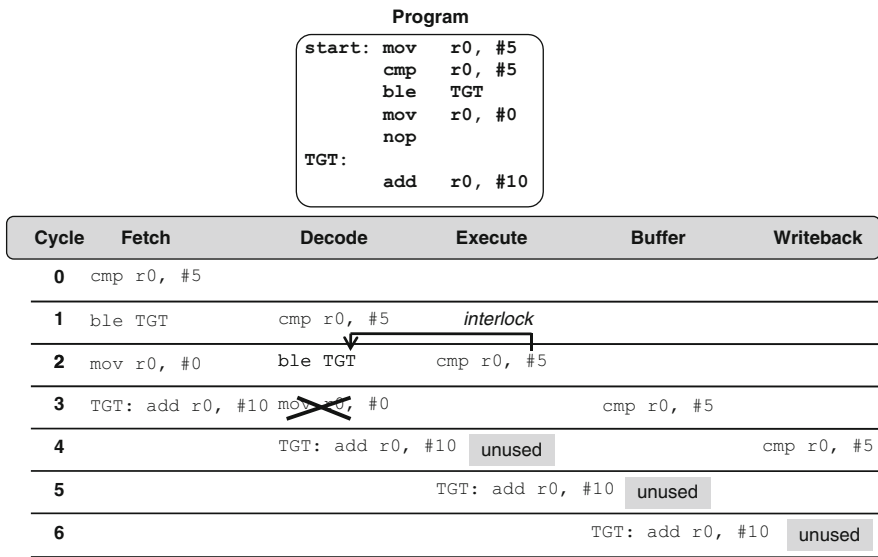
Fig. 6.4 Example of a control hazard

pipeline. As this instruction is located *after* the branch instruction, that instruction should be thrown away in order to preserve sequential execution semantics.

Figure 6.4 illustrates a *control hazard*. The pipeline is shown drawn on its side, running from left to right. Time runs down across the rows. A control hazard occurs because of the branch instruction `ble TGT`. In cycle 2, the new program counter value evaluates to the target address of the branch, `TGT`. Note that even though `ble` is a conditional branch that uses the result of the instruction just before that (`cmp r0, #5`), the branch condition is available in cycle 2 because of the interlock hardware in the pipeline (See Sect. 6.2.2). Starting in cycle 3, instructions from the target address `TGT` enter the pipeline. At the same time, the instruction just after the branch is canceled in the decode stage. This results in an unused instruction slot just after the branch instruction.

Some RISC processors, including the Microblaze, include a *delayed-branch* instruction. The purpose of this instruction is to allow the instruction just after the branch instruction to complete even when the branch is taken. This will prevent "unused" pipeline slots as shown in Fig. 6.4.

For example, the following C function:

```
1  int accumulate() {
2    int i,j;
3    for (i=0; i<100; i++)
4      j += i;
5    return j;
6  }
```

leads to the following assembly code for Microblaze:

```
      addk r4,r0,r0   ; clear r4 (holds i)
      addk r3,r3,r4   ; j = j + i
$L9:
      addik r4,r4,1   ; i = i + 1
      addik r18,r0,99 ; r18 <- 99
      cmp r18,r4,r18  ; compare i with 99
      bgeid r18,$L9   ; delayed branch if equal
      addk r3,r3,r4   ; j = j + i -> branch delay slot
```

The delayed-branch instruction is `bgeid`, which is a "branch if-greater-or-equal delayed". The instruction just after the branch corresponds to the loop body `j = j + i`. Because it is a delayed-branch instruction, it will be executed regardless if the conditional branch is taken or not.

6.2.2 Data Hazards

A second cause of pipeline stalls are *data hazards*: pipeline delays caused by the unavailability of data. Processor registers are updated at the end of each instruction, during the write-back phase. But what if the data is required before it has updated a processor register? After all, as indicated in the pipeline diagram in Fig. 6.3, the write-back stage is two cycles after the execute stage. So an instruction that reaches the write-back stage is two instructions after the instruction that is currently executing. In the following snippet, by the time the `mov` instruction reaches the write-back stage, the `add` instruction will be in the buffer stage, and the addition would have already completed.

```
  mov  r0, #5
  add  r1, r0, r1
```

In a RISC pipeline, this is handled by pipeline interlock hardware. The pipeline interlock hardware looks at the read/write patterns of all instructions currently flowing in the RISC pipeline and makes sure they take data from the right source. For the example above, when the `add` instruction is in the execute stage, it will take the result directly from the `mov` instruction result, which will be in the buffer stage. This activity is called *forwarding*, and it is handled automatically by the processor. In some cases, *forwarding* is not possible because the data is simply not yet available. This happens when a read-from-memory instruction is followed by an instruction that uses the data coming from memory. An example of this case is shown in Fig. 6.5.

The second instruction fetches data from memory and stores it in register `r1`. The following `add` instruction uses the data from that register as an operand. In cycle 4, the `add` instruction reaches the execute stage. However, at that moment, the `ldr` instruction is still accessing the data memory. The new value of `r1` is only available at the start of cycle 5. Therefore, the interlock hardware will stall all stages

Program

```
start: mov  r0, #5
       ldr  r1, [r0]
       add  r2, r1,  #3
       mov  r3, #0
       nop
```

Cycle	Fetch	Decode	Execute	Buffer	Writeback
0	mov r0, #5				
1	ldr r1,[r0]	cmp r0, #5			
2	add r2,r1,#3	ldr r1,[r0]	cmp r0, #5		
3	mov r3,#0	add r2,r1,#3	ldr r1,[r0]	cmp r0, #5	
4	wait	wait	wait	ldr r1,[r0]	cmp r0, #5
5		mov r3,#0	add r2,r1,#3	unused	ldr r1,[r0]
6			mov r3,#0	add r2,r1,#3	unused
7				mov r3,#0	add r2,r1,#3
8					mov r3,#0
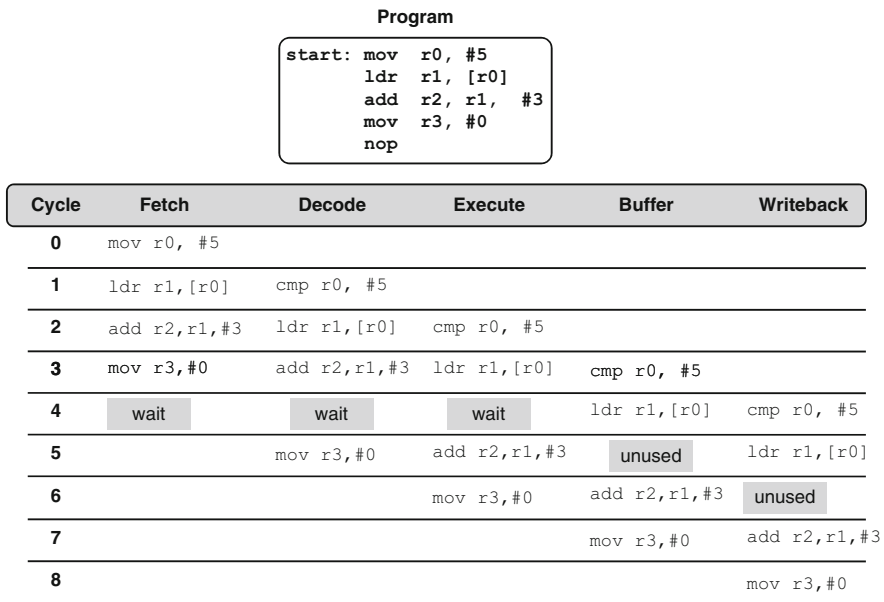
Fig. 6.5 Example of a data hazard

preceding the buffer stage in cycle 4. Starting in cycle 5, the entire pipeline moves forward again, but due to the stall in cycle 4, an unused pipeline slot flushes out in cycle 5 and cycle 6.

Data hazards can lengthen the execution time of an instruction that would normally finish in just 5 clock cycles. For classic RISC processors, data hazards can be predicted statically, by examining the assembly program. When the execution time of a program needs to be estimated exactly, a programmer will need to be able to identify all data hazards and their effects.

6.2.3 Structural Hazards

The third class of hazards are *structural hazards*. These are hazards caused by instructions that require more resources from a processor than those that are available. For example, a given instruction may require 5 concurrent additions while there is only a single ALU available. To address such a case, the execution phase of the instruction will be artificially extended over multiple clock cycles, while the pipeline stages before that will be stalled.

Another example of a structural hazard is illustrated in Fig. 6.6. The `ldmia` instruction is a load-multiple instruction that will read consecutive memory locations and store the resulting values in memory. In the example shown, the value stored in address `r0` will be copied to `r1`, while the value stored in address `r0+4` will be

Program

```
mov     r0,   #5
ldmia   r0,   {r1,r2}
add     r4,   r1,r2
add     r4,   r4,r3
```

Cycle	Fetch	Decode	Execute	Buffer	Writeback
0	mov r0, #5				
1	ldmia r0,{r1,r2}	mov r0, #5			
2	add r4,r1,r2	ldmia r0,{r1,r2}	mov r0, #5		
3	wait	wait	ldmia r0,{r1,r2}	mov r0, #5	
4	add r4,r4,r3	add r4,r1,r2	ldmia r0,{r1,r2}	load r1	mov r0, #5
5		add r4,r4,r3	add r4,r1,r2	load r2	update r1
6			add r4,r4,r3	add r4,r1,r2	update r2
7				add r4,r4,r3	add r4,r1,r2
8					add r4,r4,r3
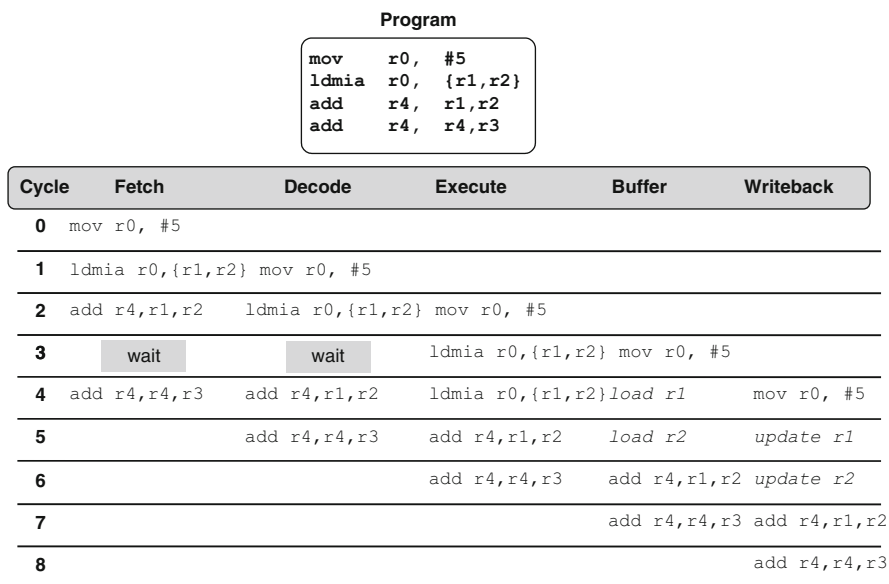
Fig. 6.6 Example of a structural hazard

copied to `r2`. When the `ldmia` instruction reaches the execute stage, the execute stage will be busy for 2 clock cycles in order to evaluate the memory addresses `r0` and `r0+4`. Therefore, all pipeline stages before the execute stage are halted for a single clock cycle. After that, the pipeline proceeds normally.

A structural hazard is caused by the processor architecture, but it may have a broad number of causes: the width of memory ports, the number of execution units in the datapath, or restrictions on the communication busses. A programmer can only predict structural hazards through a solid understanding of the processor architecture. Furthermore, memory latency effects can also cause the execution time of the buffer stage to vary. A cache miss for example can extend the latency of a load-memory instruction with hundreds of cycles. While the load-memory instruction is waiting for data to be returned from memory, it will stall the pipeline in a manner similar to a structural hazard.

6.3 Program Organization

For an efficient hardware/software codesign, a designer needs to have a simultaneous understanding of system architecture and software. This is different from traditional computer science, where a designer is typically interested in running a C program "as fast as possible", but without much concern for the computer hardware that runs the C program.

In this section, we will look at the relationship between a C program and its implementation on a RISC processor. This includes a discussion of the main parts of a C program and their mapping to instructions and into sections of memory, the organization of a C program into binary format, and the link between a C program and the RISC architecture. While the examples will be made for ARM and Micro-Blaze RISC processors, the ideas explained here are generic and applicable to many other RISC processors. In fact, a good hardware/software codesigner tries to be *as architecture-independent as possible*, which means that she will be able to easily move from one processor architecture to another. For example, we will show that it is possible to do decent performance analysis of a C program, at the cycle-accurate level, without detailed knowledge of the instruction-set of a processor.

6.3.1 Data Types

A good starting point to discuss the mapping of C programs to RISC processors are the datatypes used by C programs. Table 6.1 shows how C maps to the native datatypes supported by ARM and Microblaze processors. All C data types, apart from char, are treated as signed (two's complement) numbers.

The difference between operations on two's complement numbers and operations on unsigned numbers is minor, at least in terms of machine representation of the numbers. Signed numbers may require sign extension (see Sect. 4.1.2). In addition, the comparison of signed numbers is different from the comparison of unsigned numbers. Indeed, when comparing unsigned bytes, 0xff is bigger than 0x01. But, when comparing signed bytes, 0xff is smaller than 0x01.

The mapping of C datatypes to physical memory locations is affected by several factors. First, datatypes need to follow the rules of datatype *alignment*, which define what are the allowed starting addresses for datatypes in memory. A RISC processor will access the data memory at predefined physical boundaries, typically one word (32 bits) at a time. Thus, a single memory transfer may be able to access any of the four bytes in a word, but a group of four bytes across a word boundary cannot be accessed in a single memory transfer. For this reason, datatypes may need alignment in the physical memory organization, and this restricts the location of these datatypes in logical address space (Fig. 6.7a).

A second factor that affects the mapping of datatypes is the storage order, illustrated in Fig. 6.7b. A little-endian storage order will map the lower-significant

Table 6.1 C compiler data types

C data type	ARM	Microblaze
Char	Unsigned 8-bit	Unsigned 8-bit
Short	Signed 16-bit	Signed 16-bit
Int	Signed 32-bit	Signed 32-bit
Long	Signed 32-bit	Signed 32-bit
Long long	Signed 64-bit	Signed 64-bit
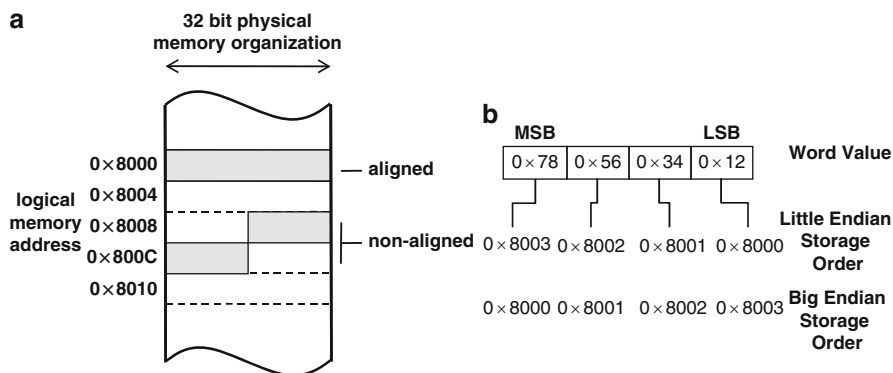
Fig. 6.7 (**a**) Alignment of data types (**b**) Little-endian and Big-endian storage order

bytes of a word into lower memory locations. A big-endian storage order, on the other hand, will map the higher-significant bytes to lower memory locations. If only C-programming is involved in a design, then the endianness is not important. In hardware/software codesign, the physical representation of datatypes is important in the transition of software to hardware and back. Hence, the endianness of a processor (and in some cases even the bit-ordering) is important. It is easy to check the endianness of a given processor using a small C program such as the following one.

```
int main() {
   char j[4];
   volatile int *pj;
   pj = (int *) j;

   j[0]  = 0x12;
   j[1]  = 0x34;
   j[2]  = 0x56;
   j[3]  = 0x78;

   printf("%x\n", *pj);
}
```

For this program, a little-endian processor will print 78,563,412, while a big-endian processor will print 12,345,678. A Microblaze processor is big-endian, while an ARM processor is (normally) little-endian.

6.3.2 Variables in the Memory Hierarchy

Next, we discuss the relationship between the variables of a C program and the memory locations used to store those variables. The *memory hierarchy* gives the RISC pipeline the illusion of a continuous and very fast memory space. As illustrated in Fig. 6.8, a memory hierarchy includes the processor registers, the cache
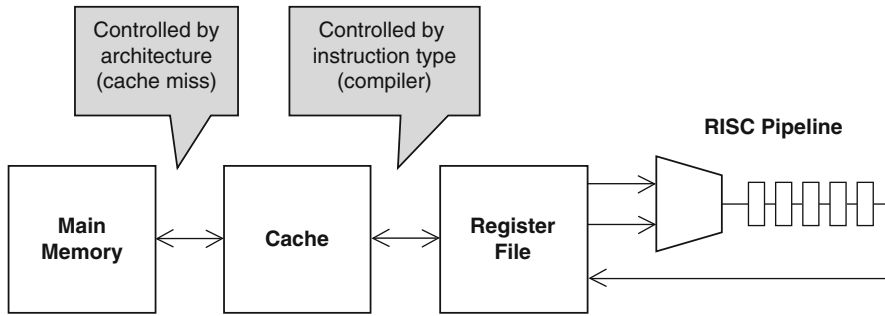
Fig. 6.8 Memory hierarchy

memory, and the main memory. In embedded processors, cache memory is optional; in high-end processors on the other hand, multiple levels of cache memory are used.

The cache operates as a fast local memory which holds the most-frequently used main-memory locations. Whenever the processor requests data from a memory location, the cache may report a *hit* and return a locally-stored copy of the desired memory location. The cache may also report a *miss* and instead first fetch the requested memory location from main memory. A cache memory improves program efficiency when data (or instructions) are used more than once by a C program. For example, a loop counter will be accessed at least once every iteration of the loop. As a result, the probability of a cache hit is much larger than that of a cache miss.

The memory hierarchy is transparent to a C programmer. Under normal circumstances, a C programmer will not worry what type of memory is used to store the data from a given program. It could be the processors' registers, the cache, or the main-memory. In reality, data travels up and down the memory hierarchy during program execution. This is illustrated by the following example, which shows a C function accumulating an array.

```
1  void accumulate(int *c, int a[10]) {
2    int i;
3    *c = 0;
4    for (i=0; i<10; i++)
5      *c += a[i];
6  }
```

We can now generate the ARM assembly code using the ARM cross-compiler. We will use optimization-level 2 for this.

```
/usr/local/arm/bin/arm-linux-gcc -O2 -c -S accumulate.c
```

This generates the following listing in `accumulate.s`:

```
1            mov     r3, #0
2            str     r3, [r0, #0]
3            mov     ip, r3
4  .L6:
5            ldr     r2, [r1, ip, asl #2] ; r2 <- a[i]
6            ldr     r3, [r0, #0]         ; r3 <- *c (memory)
```

```
7              add       ip, ip, #1              ; increment loop ctr
8              add       r3, r3, r2
9              cmp       ip, #9
10             str       r3, [r0, #0]            ; r3 -> *c (memory)
11             movgt     pc, lr
12             b         .L6
```

Let us consider how the accumulator variable is implemented. Looking at the C program, there is only a *single* placeholder for the accumulator, namely *c. In terms of physical memory, there are at least three different locations where a copy of *c may be found: the processor registers, the cache, and the main memory. For example, in the assembly implementation, we see how the *value* of the accumulator travels up in the memory hierarchy. According to the C function, the accumulator is provided through a pointer. This implies that the accumulator will be stored in main memory. On line 6 of the previous Listing, that variable is read from memory and stored in processor register r3. On line 10, the processor register r3 is written back to memory. Thus, depending on the nature and state of the cache memory, reading/writing processor registers from/to memory may trigger additional data transfer between the cache memory and the main memory. In the context of codesign, this difference between the physical implementation of a C program and its logical design is important. For example, when a communication link needs to be created between software and hardware, the physical mapping of variables is important.

A C programmer has a limited amount of control over the mapping of variables onto the memory hierarchy. The control is offered through the use of *storage class specifiers* and *type qualifiers*. The most important ones are enumerated in Table 6.2. A few example declarations are shown below.

```
volatile int *c;     // c is a pointer to a volatile int
int * const y;       // y is a constant pointer to an int
register int x;      // x is preferably mapped into a register
```

Table 6.2 C storage class specifiers and type qualifiers

Keyword	Function
Storage specifier	
Register	Indicates that the preferred mapping of the data type is in processor registers. This will keep the variable as class as possible to the RISC pipeline.
Static	Limits the scope (visibility) of the variable over multiple files. This specifier does not relate to the memory hierarchy, but to the functions where the variable may be accessed.
Extern	Extends the scope (visibility) of the variable to all files. This specifier does not relate to the memory hierarchy, but to the functions where the variable may be accessed.
Type qualifier	
Const	Indicates that the qualified variable cannot be changed.
Volatile	Indicates that the qualified variable can change its value at any time, even outside of the operations in the C program. As a result, the compiler will make sure to write the value always back to main memory after modification, and maintain a copy of it inside of the processor registers.

The use of type qualifiers and storage specifiers allows some control of the memory-hierarchy and the implementation of variables in C. Throughout this chapter, and later when discussing hardware/software interfaces, other examples of their use will appear.

6.3.3 Function Calls

Behavioral hierarchy – C functions calling other functions – is key to mastering complexity with C programs. In this section, we briefly describe the concepts of C function calls in the context of RISC processors. We use the example C program in Listing 6.5.

Let us assume that we have compiled this program for an ARM processor using the `arm-linux-gcc` cross compiler. It is possible to recreate the assembly listing corresponding to the object file by *disassembling* the object code. The utility `arm-linux-objdump` takes care of that. The `-d` flag on the command line selects the disassembler functionality. The utility supports many other functions (See Problem 6.8).

```
/usr/local/arm/bin/arm-linux-objdump -O2 -c accumulate.c -o
    accumulate.o
/usr/local/arm/bin/arm-linux-objdump -d accumulate
```

The ARM assembly listing of this program is shown in Listing 6.6.

Close inspection of the instructions will reveal many practical aspects of the runtime layout of this program, and in particular of the implementation of function calls. The instruction that branches into `accumulate` is implemented at address `0x2c` with a `bl` instruction – *branch with link*. This instruction will copy the program counter in a separate link register `lr`, and load the address of the branch target into the program counter. A return-from-subroutine can now be implemented by copying the link register back into the program counter. This is shown at address `0x1c` in `accumulate`. Obviously, care must be taken when doing

Listing 6.5 Accumulate Example in C
```
1  int accumulate(int a[10]) {
2    int i;
3    int c = 0;
4    for (i=0; i<10; i++)
5      c += a[i];
6    return c;
7  }
8
9  int a[10];
10 int one = 1;
11
12 int main() {
13   return one + accumulate(a);
14 }
```

Listing 6.6 Accumulate Example in Assembly

```
00000000 <accumulate>:
   0:   e3a01000        mov     r1, #0
   4:   e1a02001        mov     r2, r1
   8:   e7903102        ldr     r3, [r0, r2, lsl #2]
   c:   e2822001        add     r2, r2, #1
  10:   e3520009        cmp     r2, #9
  14:   e0811003        add     r1, r1, r3
  18:   c1a00001        movgt   r0, r1
  1c:   c1a0f00e        movgt   pc, lr
  20:   ea000000        b       8 <accumulate+0x8>

00000024 <main>:
  24:   e52de004        str     lr, [sp, -#4]!
  28:   e59f0014        ldr     r0, [pc, #20]    ; 44 <main+0x20>
  2c:   ebfffffe        bl      0 <accumulate>
  30:   e59f2010        ldr     r2, [pc, #16]    ; 48 <main+0x24>
  34:   e5923000        ldr     r3, [r2]
  38:   e0833000        add     r3, r3, r0
  3c:   e1a00003        mov     r0, r3
  40:   e49df004        ldr     pc, [sp], #4
        ...
```

multiple subroutine calls so that lr is not overwritten. In the main function, this is
solved at the entry, at address 0x24. There is an instruction that copies the current
contents of lr into a local area within the stack, and at the end of the main function
the program counter is directly read from the same location.

The arguments into (and out from) the accumulate function are passed
through register r0 rather than main memory. This is obviously much faster when
only a few data elements need to be copied. The input argument of accumulate
is the base address from the array a. Indeed, the instruction on address 8 uses r0
as a base address and adds the loop counter multiplied by 4. This expression thus
results in the effective address of element a[i] as shown on line 5 of the C program
(Listing 6.5). The return argument from accumulate is register r0 as well. On
address 0x18 of the assembly program, the accumulator value is passed from r1
to r0. For ARM processors, the full details of the procedure-calling convention are
defined in the *ARM Procedure Call Standard* (APCS), a document used by com-
piler writers and software library developers. In general, arguments are passed from
function to function through a data structure known as a *stack frame*. A stack frame
holds the return address, the local variables, the input and output arguments of the
function, and the location of the calling stack frame. A nice example of a stack frame
is found when the accumulate function described earlier is compiled *without* op-
timizations. In that case, the C compiler takes a very conservative approach and will
keep all local variables in main memory, rather than in registers.

```
/usr/local/arm/bin/arm-linux-gcc -c -S accumulate.c
```

Listing 6.7 shows the resulting nonoptimized assembly code of accumulate.
Figure 6.9 illustrates the construction of the stack frame.

Listing 6.7 Accumulate without compiler optimizations

```
1   accumulate:
2           mov     ip, sp
3           stmfd   sp!, {fp, ip, lr, pc}
4           sub     fp, ip, #4
5           sub     sp, sp, #12
6           str     r0, [fp, #-16]    ; base address a
7           mov     r3, #0
8           str     r3, [fp, #-24]    ; c
9           mov     r3, #0
10          str     r3, [fp, #-20]    ; i
11  .L2:
12          ldr     r3, [fp, #-20]
13          cmp     r3, #9            ; i<10 ?
14          ble     .L5
15          b       .L3
16  .L5:
17          ldr     r3, [fp, #-20]    ; i * 4
18          mov     r2, r3, asl #2
19          ldr     r3, [fp, #-16]
20          add     r3, r2, r3        ; *a + 4 * i
21          ldr     r2, [fp, #-24]
22          ldr     r3, [r3, #0]
23          add     r3, r2, r3        ; c = c + a[i]
24          str     r3, [fp, #-24]    ; update c
25          ldr     r3, [fp, #-20]
26          add     r3, r3, #1
27          str     r3, [fp, #-20]    ; i = i + 1
28          b       .L2
29  .L3:
30          ldr     r3, [fp, #-24]    ; return arg
31          mov     r0, r3
32          ldmea   fp, {fp, sp, pc}
```

The instructions on line 2 and 3 are used to create the stack frame. On line 3, the frame pointer (fp), stack pointer (sp), link register or return address (lr), and current program counter (pc) are pushed onto the stack. The single instruction stmfd is able to perform multiple transfers (m), and it grows the stack downward (fd). These four elements take up 16 bytes of stack memory.

On line 3, the frame pointer is made to point to the first word of the stack frame. All variables stored in the stack frame will now be referenced based on the frame pointer fp. Since the first 4 words in the stack frame are already occupied, the first free word is at address fp - 16, the next free word is at address fp - 20, and so on. These addresses may be found back in Listing 6.7.

The following local variables of the function accumulate are stored within the stack frame: the base address of a, the variable i, and the variable c. Finally, on line 31, a return instruction is shown. With a single instruction, the frame pointer fp, the stack pointer sp, and the program counter pc are restored to the values just before calling the accumulate function.

Before Calling Accumulate   **During Execution of Accumulate**

High   High

old fp →

sp →

Low

Stack Frame Construction

```
mov    ip, sp
stmfd  sp!, {fp, ip, lr, pc}
sub    fp, ip, #4
sub    sp, sp, #12
str    r0, [fp, #-16]
str    r3, [fp, #-20]
str    r3, [fp, #-24]
```

new fp →   old fp

−4   old sp

−8   return address

−12   pc

−16   base adr a

−20   i

Stack Frame Return

Low   c   ← new sp

```
ldmea   fp, {fp, sp, pc}
```
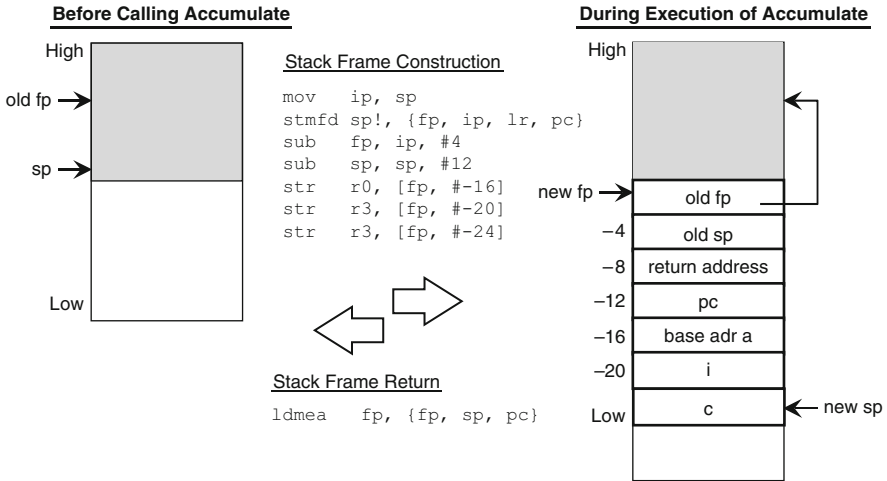
Fig. 6.9 Memory hierarchy

6.3.4 Program Layout

Another aspect of C program implementation is the physical representation of the program and its data structures in the memory hierarchy. This leads to the *program layout*, the template that is used to organize instructions and data. A distinction must be made between the organization of a compiled C program in an executable file (or a program ROM), and the memory organization of that C program during execution. The former case is a static representation of all the instructions and constants defined by the C program. The latter case is a dynamic representation of all the instructions and the runtime data structures such as the stack and the heap.

Figure 6.10 shows how a C program is compiled into an executable file, which in turn is mapped into memory. There are several standards available for the organization of executable files. In the figure, the example of ELF (Executable Linkable Format) is shown. An ELF file is organized into sections, and each of these can take up a variable amount of space in the file. The sections commonly found in an ELF file are the .text section which contains the binary instructions of the C program and the .data section which contains initialized data (constants). The ELF file may also contain debugging information, such as the names of the variables in the C program. This debugging information is utilized by source level debuggers to relate the execution of a binary program to actions of the C source code.

When a compiled C program is first loaded into memory for execution, the ELF file is analyzed by the loader and the sections with relevant information are copied into memory locations. In contrast to a file, the resulting organization of instructions and data into memory do not need to be contiguous or even occupy the same physical memory. Each section of an ELF file can be mapped at a different address, and possibly map into a different memory module. The example in Fig. 6.10 shows how
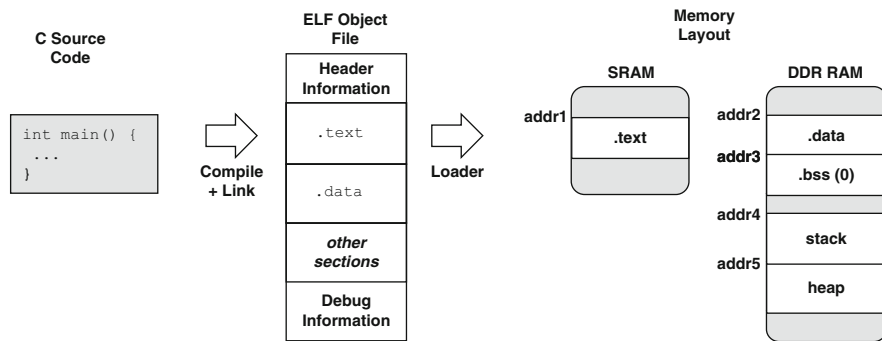
Fig. 6.10 Static and dynamic program layout

the .text segment maps into fast static ram memory (SRAM) while the .data, stack, and heap segments map into DDR RAM memory. During program execution, there may be sections of memory which do not appear in the ELF file, or which do not occupy any area within the ELF file. These sections include data storage areas: dynamic data (heap), local data (stack), and global data (bss).

A C compiler will typically come with utilities that allows inspection of the organization of an executable file or an object file. Based on this, we can understand why a given section may or may not fit into a given memory area. We will illustrate a few GNU utilities that can report the size of each executable segment. Assume that we have a compiled ELF executable for the C program from Listing 6.5, and that we need to know how much memory is required for the data and instructions from that program.

We assume that the program is called sections.c, that its corresponding object file is sections.o, and that the executable file is called sections. We make use of the ARM cross-compiler tools. The arm-linux-size utility gives a summary printout of the amount of memory required for a given C program.

```
> arm-linux-size sections.o
   text    data     bss     dec     hex filename
     76       4       0      80      50 sections.o
```

The output of the program on sections.o shows that there are 76 bytes in the text segment, 4 bytes in the initialized data-segment data, and 0 bytes in the non-initialized data-segment bss. Looking at the C program, we can conclude that there are 19 words (76/4) required for the instructions that implement accumulate and main. Assuming one instruction per word, there will be around 19 instructions required to implement these two functions. Besides the text segment, there is also a data segment with 4 bytes. These 4 bytes of *initialized* data are needed to store the variable one on line 10.

When we run the same utility on the executable file, we find that the amount of code and data increases significantly. This is because various C libraries have been linked into the program.

```
> arm-linux-size sections
   text    data     bss     dec     hex filename
 362095    4176    5204  371475   5ab13 sections
```

We now verify our assumption on the 19 instructions in the program. The assembly listing corresponding to `sections.o` is shown in Listing 6.6. We find that there are 17 instructions, while according to the `size` command there are 19 words in the text segment. Where do these two additional words come from? Inspection of the assembly program reveals the answer. On address `2c`, the first word *after* the last instruction is read and stored into `r0`. On address `30`, the second word after the last instruction is read and stored into `r2`. These two addresses correspond to the base addresses of the array `a` and the scalar variable `one`, respectively. Thus, the C compiler turns all global variables into pointers, which are stored as part of the text segment.

The above analysis of the executable program still leaves a few questions unanswered. One of them is where the text and data segment will fit into physical memory? This question can also be addressed using the same `arm-linux-objdump` utility. The `-h` flag turns on printout of *header* sections. This reveals the following information for the `sections` program.

```
> arm-linux-objdump -h sections

sections:      file format elf32-littlearm

Sections:
Idx Name           Size     VMA       LMA       File off  Algn
  0 .init          00000014 000080c0  000080c0  000000c0  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .text          00046de8 000080e0  000080e0  000000e0  2**4
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .fini          0000000c 0004eec8  0004eec8  00046ec8  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata        00011867 0004eed4  0004eed4  00046ed4  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
 ...
  7 .data          00001024 00068788  00068788  00058788  2**2
                   CONTENTS, ALLOC, LOAD, DATA
 ...
 14 .bss           00001454 000697e0  000697e0  000597e0  2**5
                   ALLOC
```

This listing illustrates the name of the sections, their size, the starting address (VMA and LMA), the offset within the ELF file, and the alignment in bytes as a power of 2. The number we are looking for is the starting address of each segment. VMA stands for *virtual memory address* and reflects the address of the section during execution. LMA stands for *load memory address* and reflects the address of the section when the program is first loaded into memory. In this case both are the same. The numbers would be different in cases where the program is stored in a different memory than the one that holds the program at runtime. For example, a Flash memory can store the program sections at the address reflected by LMA. When the program executes, it is copied into RAM memory at the address reflected by VMA.

The section dump shown above does not indicate the place where the stack and heap are allocated. The answer to that question is that an `arm-linux` executable does not decide by itself where the stack will be allocated. Instead, this executable is meant to run as a process in an operating system (linux) and follows the conventions for stack and heap in that operating system.

However, in the case when an operating system is not present, the executable will show sections for the stack and heap as well. As an example, we illustrate the output of the utilities discussed above when the program is executed stand-alone on a Microblaze core. To obtain this information, the C program from Listing 6.5 is compiled on a Microblaze C cross-compiler. Next, similar GNU utilities for Microblaze are used to analyze the object file and the executable file. Compared to the numbers found earlier, the final executable is considerably slimmer, due to the optimized libraries that are used.

```
> mb-size sections.o
   text    data     bss     dec     hex filename
     84       4       0      88      58 sections.o
> mb-size sections
   text    data     bss     dec     hex filename
   1528     304    2128    3960     f78 sections
```

The result of `objdump` listing the header sections is illustrated in Fig. 6.11. In this case, the listing contains, besides a `text`, `data`, and `bss` segment, also a `heap` and a `stack`. Using the numbers from the size and LMA columns, one can draw a memory map as shown on the right of the figure.
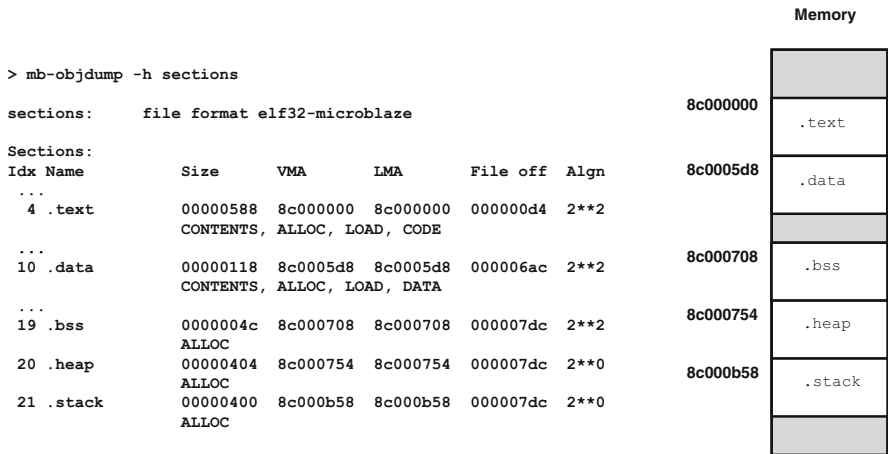
Fig. 6.11 Output of `objdump` on the sections program on Microblaze

6.4 Analyzing the Quality of Compiled Code

By the nature of the job, a hardware/software codesigner is likely to get into touch with the low-level details of a program. That, in turn, inevitably means handling assembly language of a processor. Since most likely you will encounter many different processor architectures over your engineering career, it's reasonable to ask how much knowledge is really required on the processor's instruction set in order to do useful hardware/software codesign. This section will attempt to answer that question. In short, it is not very useful to learn how to program assembly on a processor when there is a C compiler available for that processor. However, it is very useful to be able to analyze the C compiler output at the level of the assembly language. Thus, given a C program and an assembly program corresponding to the C, a hardware/software codesigner should be able to establish the links between the two. We call this concept *Program Analysis*: interpreting and understanding the performance of a program based on observing the assembly code of that program. Program Analysis is useful for addressing many design activities, such as the following examples.

- Program analysis provides insight into the optimizations that a C compiler can and cannot offer. Many C programmers are hopeful about the capabilities of a C compiler to produce efficient assembly code. While this is generally true for a high-quality compiler, there are many cases where a compiler cannot help. For example, a compiler is unable to make optimizations that require specific knowledge in the statistical properties of the program data input. A compiler will not transform a program with `double` or `float` variables into one with `int` variables, even if an `int` would give sufficient precision for a particular application. A compiler cannot perform optimizations at high abstraction levels, such converting one type of sorting algorithm into another, equivalent, but more efficient, sorting algorithm. To understand what a compiler can and cannot do, it is helpful to compare C code and the corresponding assembly code.
- Program analysis enables a programmer to make quite accurate predictions on the execution time of a given program. In cases where you are addressing low-level issues, such as controlling hardware modules or controlling hardware interface signals from within software, these timing predictions may be very important.

We distinguish *static* program analysis, which inspects only the assembly code, and *dynamic* program analysis, which observes program execution at runtime. We will illustrate the concept of each by means of examples from ARM and Microblaze assembly code.

6.4.1 Analysis Based on Static Assembly Code

The objective of Static Program Analysis is to quantify (appreciate) the performance of a given C program on a given processor by studying the assembly language produced by the C compiler, and while relying on the processor's documentation

Listing 6.8 A simple convolution function

```
1  int array[256];
2  int c[256];
3  int main() {
4    int i, a;
5    a = 0;
6    for (i=0; i<256; i++)
7      a += array[i] * c[256 - i];
8    return a;
9  }
```

as needed. We will do this by discussing a small example C program, shown in Listing 6.8. This program is a convolution operation: it evaluates the cross-product of a data-array with a reversed data-array.

We are interested in the efficiency of this program on a Microblaze processor. For this purpose, we generate the assembly listing of this program using the Microblaze GNU compiler. The optimization level is set at O2. The resulting assembly listing is shown in Listing 6.9.

The question we wish to address is: did the C compiler do a good job while converting the C program in Listing 6.8 into assembly? In previous sections, we already discussed several of the elements that help us answer this question, including the concept of a *stack frame*. Another concept is the *Application Binary Interface* (ABI). The ABI defines how a processor will use its registers to implement a C program. For the case of a Microblaze processor and the example in Listing 6.8, the following aspects are relevant.

- The Microblaze has 32 registers.
- Register r0 is always zero and used as a zero-constant.
- Register r1 is the stack pointer.
- Registers r19 through r31 are callee-saved registers: a function that wishes to use these registers must preserve their contents before returning to the caller.

  These elements will help to understand lines 8–17 from Listing 6.9.

- Line 8 grows the stack pointer by 44 bytes (11 words). Note that the Microblaze stack grows downwards.
- Lines 9, 10, 13, 14 save registers on the stack. These registers (r22, r23, r19, r24) will be used as temporary variables by the program. They are restored just before the main function terminates.

  From the values loaded into the working registers, we can infer what they actually represent.

- Register r22 is initialized with array, the starting address of the array.
- Register r23 is initialized with c+1024, which is the start of the c variable plus 1024. Since the c variable is an array with 256 integers, we conclude that r23 points to the end of the c variable.
- Register r19 is initialized to 255, which is the loop count minus 1.
- Register r24 is initialized to 0, and could be the loop counter or the accumulator.

Listing 6.9 Microblaze assembly for the convolution program

```
1          .text
2          .align 2
3          .globl main
4          .ent main
5   main:
6          .frame r1,44,r15
7          .mask 0x01c88000
8          addik r1,r1,-44
9          swi   r22,r1,32
10         swi   r23,r1,36
11         addik r22,r0,array
12         addik r23,r0,c+1024
13         swi   r19,r1,28
14         swi   r24,r1,40
15         swi   r15,r1,0
16         addk  r24,r0,r0
17         addik r19,r0,255
18  $L5:
19         lwi   r5,r22,0
20         lwi   r6,r23,0
21         brlid r15,__mulsi3
22         addik r19,r19,-1
23         addk  r24,r24,r3
24         addik r22,r22,4
25         bgeid r19,$L5
26         addik r23,r23,-4
27         addk  r3,r24,r0
28         lwi   r15,r1,0
29         lwi   r19,r1,28
30         lwi   r22,r1,32
31         lwi   r23,r1,36
32         lwi   r24,r1,40
33         rtsd  r15,8
34         addik r1,r1,44
35         .end  main
36  $Lfe1:
37         .size main,$Lfe1-main
38         .bss
39         .comm array,1024,4
40         .type array, @object
41         .comm c,1024,4
42         .type c, @object
```

We now know enough to tackle the loop body in lines 18–26. Loops in assembly code are easy to find since they always start with a label (like $L5) and terminate with a branch instruction to that label. In this case, the last instruction of the loop body is on line 25 because the branch on line 24 is a delayed-branch (ends with a "d"). The loop body reads an element from the variables array and c (line 18 and 19) and stores the result in r5 and r6. The next instruction is a function call. It is

implemented in a RISC processor as a branch which saves the return address on the stack. r15 is used to hold the return address. The function is called __mulsi3. From its name, this function hints to be a multiplication, indicating that the compiler generated code for a microprocessor without a built-in multiplier. The multiplication will support the implementation of the following C code.

```
a += array[i] * c[256 - i];
```

This is a clear example where the investigation of the assembly code helps to answer performance questions that are invisible at the level of C. For example, suppose that you would be testing a Microblaze program. To improve the program performance, you added a hardware multiplier to the process. However, you don't notice any improvements. Looking into the assembly code (and finding __mulsi3) will directly reveal why the hardware multiplier is not used.

The result of the function __mulsi3 is provided in registers r3 and r4 (this is another convention of the ABI). Indeed we see that r3 is accumulated to r24 on line 21. This clears up the meaning of register r24: it is the accumulator. Note that there are three adjustments to counter values in the loop body: Register r19 is decremented by 1, register r22 is incremented by 4, and register r23 is decremented by 4. The adjustment to r23 is still part of the loop because this instruction is located in the branch delay slot after the bgeid branch. We already know that register r22 and r32 are pointers pointing to the variables array and c. Register r19 is the loop counter. Thus, we conclude that the compiler was able to find out that the address expressions for c and array are sequential, just as the loop counter i. It is as if the compiler has automatically performed the following very effective optimization:

```
int array[256];
int c[256];
int main() {
  int i, a;
  int *p1, *p2;
  p1 = array;
  p2 = &(c[255]);
  a = 0;
  for (i=0; i<256; i++)
    a += (*(p1++)) * (*(p2--));
  return a;
}
```

We conclude that the C compiler is able to do fairly advanced dataflow analysis and optimization (when the optimization flag is turned on). Static program analysis does not reveal cycle counts and performance numbers. Rather, it provides a qualitative appreciation of a program. Being able to investigate assembly code, even for processors foreign to you, enables you to make accurate decisions on potential software performance.

6.4.2 Analysis Based on Execution of Object Code

Another way of looking at program performance is to analyze its behavior during execution. This is called *Dynamic Program Analysis*. Processor features such as pipeline stalls and cache misses are not easy to determine using static program analysis alone. Dynamic Program Analysis can reveal all these effects.

In order to observe a program during execution, we need to make use of a processor simulator. SimIt-ARM, one of the instruction simulators integrated in GEZEL, is able to report the activities of each instruction as it flows through the processor pipeline. This includes quantities such as the value of the program counter, the simulation cycle-count, and the instruction completion time. Obviously, collecting this type of information will generate huge amounts of data, and a programmer needs to trace instructions selectively. SimIt-ARM provides the means to turn the instruction-tracing feature on or off so that a designer can focus on a particular program area of interest.

Listing 6.10 shows the listing of a GCD function. Lines 11–13 illustrate a pseudosystemcall that is used to turn the instruction-tracing feature of SimIt-ARM on and off. This pseudosystemcall is simulator-specific, and will be implemented differently when a different processor or simulation environment is used. As shown in the `main` function on lines 17–19, the `gcd` function is called after turning the tracing feature on, and it is turned-off again after that.

We will also generate the assembly code for the gcd function, which is useful as a guide during instruction tracing. Listing 6.11 shows the resulting code,

Listing 6.10 Microblaze assembly for the convolution program

```
1   int gcd (int a, int b) {
2     while (a != b) {
3       if (a > b)
4         a = a - b;
5       else
6         b = b - a;
7     }
8     return a;
9   }
10
11  void instructiontrace(unsigned a) {
12    asm("swi 514");
13  }
14
15  int main() {
16    int a, i;
17    instructiontrace(1);
18    a = gcd(6, 8);
19    instructiontrace(0);
20    printf("GCD=%d\n", a);
21    return 0;
22  }
```

Listing 6.11 ARM assembly code for gcd function

```
1   gcd:
2       mov    ip, sp                      ; set up stack frame
3       stmfd  sp!, {fp, ip, lr, pc}
4       sub    fp, ip, #4
5       sub    sp, sp, #8
6       str    r0, [fp, #-16]              ; storage for var_a
7       str    r1, [fp, #-20]              ; storage for var_b
8   .L2:
9       ldr    r2, [fp, #-16]
10      ldr    r3, [fp, #-20]
11      cmp    r2, r3                      ; while (var_a != var_b)
12      bne    .L4
13      b  .L3
14  .L4:
15      ldr    r2, [fp, #-16]              ; if (var_a > var_b)
16      ldr    r3, [fp, #-20]
17      cmp    r2, r3
18      ble    .L5
19      ldr    r3, [fp, #-16]
20      ldr    r2, [fp, #-20]
21      rsb    r3, r2, r3                  ; var_a = var_a - var_b;
22      str    r3, [fp, #-16]
23      b  .L2
24  .L5:                                   ; else
25      ldr    r3, [fp, #-20]
26      ldr    r2, [fp, #-16]
27      rsb    r3, r2, r3                  ; var_b = var_b - var_a;
28      str    r3, [fp, #-20]
29      b  .L2
30  .L3:
31      mov    r0, r3
32      ldmea  fp, {fp, sp, pc}
```

annotated with the corresponding C statements in Listing 6.10. We will first look at the execution without compiler optimization.

```
/usr/local/arm/bin/arm-linux-gcc -static -S gcd.c -o gcd.S
```

Next, we prepare the processor simulator to collect the instruction trace. Processor simulators will typically require additional configuration parameters for the memory subsystem. In this simulation, we use the following parameters:

- D-cache of 16 KByte, organized as a 32-set associative cache with a line size of 32-bytes.
- I-cache of 16 KByte, organized as a 32-set associative cache with a line size of 32-bytes.
- 64-cycle memory-access latency, 1-cycle cache-access latency.

For completeness, we will recall the operation of a set-associative cache. Consider the address mapping used by a 16KByte set-associative cache with 32 sets and a line size of 32 bytes. Since the entire cache is 16KByte, each of the 32 sets in the
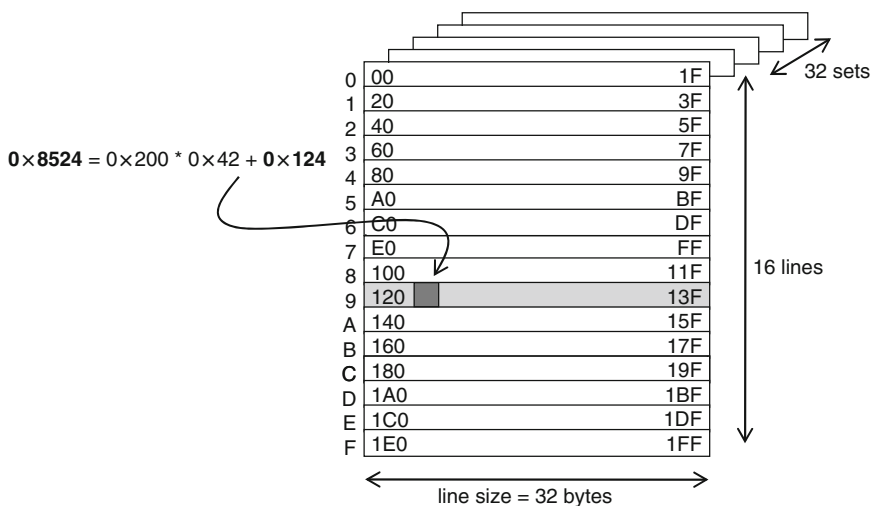
Fig. 6.12 Mapping of address 0x8524 in a 32-set, 16-line, 32-bytes-per-line set-associative cache

cache contains 512 bytes or 16 lines. If we number the cache lines from 0 to 15, then address n from the address space will map into line $(n/32)mod16$. For example, assume that the processor performs an instruction fetch from address 0x8524. Figure 6.12 shows how this address maps into the second word of the tenth line of the cache. The cache will thus check each tenth line in each of the 32 sets before declaring a cache-miss.

We can now perform the simulation with instruction tracing on. The output is shown, in part, below. The columns in this listing have the following meaning.

- **Cycle:** The simulation cycle count at the instruction fetch
- **Addr:** The location of that instruction in program memory
- **Opcode:** The instruction opcode
- **P:** Pipeline misspeculation. If a 1 appears in this column, then the instruction is not completed but removed from the pipeline
- **I:** Instruction-cache miss. If a 1 appears in this column, then there is a cache miss when this instruction is fetched
- **D:** Data-cache miss. If a 1 appears in this column, then there is a data cache miss when this instruction executes
- **Time:** The total time that this instruction is active in the pipeline, from the cycle it is fetched to the cycle it is retired
- **Mnemonic:** Assembly code for this instruction

```
Cycle     Addr    Opcode    P I D Time   Mnemonic
30601     81e4    e1a0c00d  0 1 0   70   mov   ip, sp;
30667     81e8    e92dd800  0 0 0    8   stmdb sp!, {fp, ip, lr, pc};
30668     81ec    e24cb004  0 0 0    8   sub   fp, ip, #4;
30672     81f0    e24dd008  0 0 0    5   sub   sp, sp, #8;
```

```
30673    81f4    e50b0010 0 0 0     5   str   r0, [fp, #-16];
30674    81f8    e50b1014 0 0 0     5   str   r1, [fp, #-20];
30675    81fc    e51b2010 0 0 0     5   ldr   r2, [fp, #-16];
30676    8200    e51b3014 0 1 0    70   ldr   r3, [fp, #-20];
30742    8204    e1520003 0 0 0     6   cmp   r2, r3;
30743    8208    1a000000 0 0 0     3   bne   0x8210;
30745    820c    ea00000d 1 0 0     1   b     0x8248;
30746    8210    e51b2010 0 0 0     5   ldr   r2, [fp, #-16];
30747    8214    e51b3014 0 0 0     5   ldr   r3, [fp, #-20];
30748    8218    e1520003 0 0 0     6   cmp   r2, r3;
30749    821c    da000004 0 0 0     3   ble   0x8234;
30751    8220    e51b3010 1 1 0     1   ldr   r3, [fp, #-16];
30752    8234    e51b3014 0 1 0    69   ldr   r3, [fp, #-20];
30817    8238    e51b2010 0 0 0     5   ldr   r2, [fp, #-16];
30818    823c    e0623003 0 0 0     6   rsb   r3, r2, r3;
30819    8240    e50b3014 0 0 0     6   str   r3, [fp, #-20];
30821    8244    eaffffec 0 0 0     2   b     0x81fc;
30822    8248    e1a00003 1 0 0     1   mov   r0, r3;
30823    81fc    e51b2010 0 0 0     5   ldr   r2, [fp, #-16];
30824    8200    e51b3014 0 0 0     5   ldr   r3, [fp, #-20];
30826    8208    1a000000 0 0 0     3   bne   0x8210;
30828    820c    ea00000d 1 0 0     1   b     0x8248;
```

First, find a few instructions in the table that have a "1" in the P column. These are pipeline misspeculations. They happen for example at cycle 30,745 and cycle 30,751. You can see that these instructions come just *after* a branch instruction, and thus they are caused by a control hazard. Next observe the execution time of the instructions. Most instructions take less than 6 clock cycles, but a few take over 50 clock cycles. As indicated in the I and D column, these instructions are slowed down by cache misses. For example, the instruction at cycle 30,676, address 0x8200, is an instruction-cache miss, and so is the instruction at cycle 30,752, address 0x8234.

It is possible to explain why an instruction causes an I-cache miss? Indeed, this is possible, and there are two cases to consider. The first case is when a linear sequence of instructions is executing. In that case, I-cache misses will occur at the boundary of the cache-lines. In a cache organization with a line size of 32 bytes, cache misses will thus occur at multiples of 32 bytes (0x20 in hex). The instruction at address 0x8200 is an example of this case. This is the first instruction of a cache line which is not in the cache. Therefore, the instruction-fetch stage stalls for 64 clock cycles in order to update the cache. The second case is when a jump instruction executes and moves to a program location which is not in the cache. In that case, the target address may be in the middle of a cache line, and a cache miss may still occur. The instruction at address 0x8234 is an example of that case. That instruction is executed as a result of jump. In fact, the instruction just before that (at address 0x8220) is also cache miss. That instruction does not complete, however, because it is part of a control hazard.

Finally, observe also that some regular instructions take 5 clock cycles to complete, while others take 6 clock cycles. A relevant example are the instructions on address 0x8214 and address 0x8218. The first of these instructions is a memory-fetch

that loads the value of a local variable (b) into register r3. The following instruction is a compare instruction that uses the value of r3. As discussed earlier, this is an example of a data hazard, where the value of a register is only available after the buffer stage of the RISC pipeline. The compare-instruction at address 0x8218 cannot benefit from pipeline interlock hardware and it must be stalled for 1 clock cycle until the result is available from data-memory.

As a result, dynamic instruction-tracing makes it possible to determine the cause of pipeline hazards in a program. This technique is therefore useful for low-level performance optimization.

6.5 Summary

In this chapter, we discussed the organization and operation of typical RISC processors, using the ARM and the Microblaze as an example. In hardware–software codesign, processors are the entry-point of software into the hardware world. Hence, to analyze the operation of a low-level hardware–software interface, it is very useful to understand the link between a C program, its assembly instructions, and the behavior of these instructions in the processor pipeline. The execution of software by a RISC processor is affected by the behavior of the RISC pipeline, its memory hierarchy, and the organization of instructions and data into memory. Through the understanding of a limited set of concepts in C, these complex interactions can be understood and controlled to a fairly detailed level. For example, the mapping of datatypes to memory can be influenced with storage qualifiers, and detailed performance optimization is possible through careful rewriting of C code in combination with study of the resulting program through static and dynamic analysis. This chapter has prepared us for the next big step in a hardware/software codesigned system: the extension of a simple RISC processor into a system-on-chip architecture that integrates software, processors, and custom hardware functions. Clearly, the RISC processor will play a pivotal role in this story.

6.6 Further Reading

The classic work on RISC processor architectures is by Hennessy and Patterson (2006). It is essential reading if you want to delve into the internals of RISC processors. Good documentation on the low-level software tools such as size and objdump is not easy to find; the manual pages unfortunately are rather specialized. Books on Embedded Linux Programming, such as Yaghmour et al. (2008), are the right place to start if the man pages do not help. The ELF format is described in detail in the Tool Interface Standard ELF format (ELF Committee 1995). Processor documentation can be found with the processor designers or processor vendors. For example, ARM has an extensive on-line library documenting all the features of

ARM processors (ARM 2009b), and Xilinx provides a detailed specification of the Microblaze instruction-set (Xilinx 2009).

An effective method to learn about the low-level implementation of a RISC core is to implement one, for example starting from open source code. The LEON series of processors by Gaisler Aeroflex, for example, provides a complete collection of HDL source code, compilers, and debuggers (Aeroflex 2009). The internals of a processor simulator, and of the SimIt-ARM instruction-set simulator are described by Qin in several articles (D'Errico and Qin 2006; Qin and Malik 2003).

6.7 Problems

6.1. Write a short C program that helps you to determine if the stack grows upwards or downwards.

6.2. Write a short C program that helps you to determine the position of the stack segment, the text segment, the heap, and data segment (global variables).

6.3. Explain the difference between the following terms:

- Control hazard and data hazard
- Delayed branch and conditional branch
- Little Endian and Big Endian
- `volatile int * a` and `int * const a`
- Virtual Memory Address (VMA) and Load Memory Address (LMA)

6.4. This problem considers C Qualifiers and Specifiers.

(a) Correct or not: The `volatile` qualifier will prevent a processor from storing that variable in the cache ?
(b) When writing a C program, you can create an integer variable a as follows: `register int a`. This specifier tells the compiler that a should be preferably kept in a register as much as possible, in the interest of program execution speed. Explain why this specifier cannot be used for the memory-mapped registers in a hardware coprocessor.

6.5. The following C function was compiled for Microblaze with optimization-level O2. It results in a sequence of 4 assembly instructions. Carefully examine the C code (Listing 6.12) and the assembly code (Listing 6.13), and answer the following questions. Note that register `r5` holds the function argument and register `r3` holds the function return value.

(a) Explain why the assembly code does not have a loop?
(b) Suppose line 5 of the C program reads `a = a - 1` instead of `a = a + 1`. Determine how the assembly code would change.

Listing 6.12 C Listing for Problem 6.5
```
1  int dummy(int a) {
2    int i, j = a;
3    for (i=0; i<3; i++) {
4      j += a;
5      a  = a + 1;
6    }
7    return a + j;
8  }
```

Listing 6.13 Assembly Listing for Problem 6.5
```
1      muli  r3, r5,   4
2      addk  r4, r3, r5
3      rtsd  r15, 8
4      addik r3, r3, 6
```

Listing 6.14 Assembly Listing for Problem 6.6
```
1      ldr   r3, [fp, #-16];  // load-register
2      mov   r2, r3, lsr #1;  // lsr = shift-right
3      ldr   r3, [fp, #-16];
4      and   r3, r3, #1;
5      rsb   r3, r3, #0;
6      and   r3, r3, #-805306367;
7      eor   r3, r2, r3;
8      str   r3, [fp, #-16];  // store-register
```

6.6. The following C statement implements a pseudorandom generator. It translates to the sequence of assembly instructions as shown in Listing 6.14. The assembly instructions are those of a 5-stage pipelined StrongARM processor.

```
unsigned rnstate;
rnstate = (rnstate >> 1) ^ (-(signed int)(rnstate &1)
          & 0xd0000001u);
```

Answer the following questions:

(a) What is the purpose of line 5 in Listing 6.14 (the `rsb` instruction) in the StrongArm Code? Point out exactly what part of the C expression it will implement.
(b) What types of hazard can be caused by line 3 in Listing 6.14?

6.7. The C in Listing 6.15 was compiled for StrongARM using the following command:

```
/usr/local/arm/bin/arm-linux-gcc -O -S -static loop.c -o loop.s
```

The resulting assembly code is shown in Listing 6.16.

(a) Draw a dataflow diagram of the assembly code.
(b) Identify all instructions in this listing that are directly involved in the address calculation of a data memory read.

Listing 6.15 C program for Problem 6.7

```
1  int a[100]
2    int b[100];
3    int i;
4
5    for (i=0; i<100; ++i) {
6      a[b[i]] = i + 2;
7    }
8    return 0;
```

Listing 6.16 Assembly listing for Problem 6.7

```
1       mov     r1, #0
2  .L6:
3       add     r0, sp, #800
4       add     r3, r0, r1, asl #2
5       ldr     r3, [r3, #-800]
6       add     r2, r0, r3, asl #2
7       add     r3, r1, #2
8       str     r3, [r2, #-400]
9       add     r1, r1, #1
10      cmp     r1, #99
11      ble     .L6
```

6.8. Listing 6.17 shows a routine to evaluate the CORDIC transformation (Coordinate Digital Transformation). CORDIC procedures are used to approximate trigonometric operations using simple integer arithmetic. In this case, we are interested in the inner loop of the program, on lines 16–28. This program will be compiled with a single level of optimization as follows:

```
arm-linux-gcc -O1 -g -c cordic.c
```

Next, the assembly code of the program is created using the objdump utility. The command line flags are chosen to generate the assembly code, interleaved with the C code. This is possible if the object code was generated using *debug information* (-g flag above). The resulting file is shown in Listing 6.18.

```
arm-linux-objdump -S -d cordic.o
```

(a) Study the listing in Listing 6.18 and explain the difference between an addgt and an addle instruction on the ARM processor.
(b) Using objdump, find the size of the text segment and the data segment.
(c) Study the listing in Listing 6.18 and point out what are the *callee-saved* registers in this routine.
(d) Estimate the execution time for the cordic routine, ignoring the cache misses.

Listing 6.17 C listing for Problem 6.8

```
1  #define AG_CONST 163008218
2
3  static const int angles[] = {
4    210828714, 124459457,  65760959,  33381289,
5     16755421,   8385878,   4193962,   2097109,
6      1048570,    524287,    262143,    131071,
7        65535,     32767,     16383,       8191,
8         4095,      2047,       1024,       511 };
9
10 void cordic(int target, int *rX, int *rY) {
11   int X, Y, T, current;
12   unsigned step;
13   X       = AG_CONST;
14   Y       = 0;
15   current = 0;
16   for(step=0; step < 20; step++) {
17     if (target > current) {
18       T          =  X - (Y >> step);
19       Y          = (X >> step) + Y;
20       X          = T;
21       current    += angles[step];
22     } else {
23       T          = X + (Y >> step);
24       Y          = -(X >> step) + Y;
25       X          = T;
26       current    -= angles[step];
27     }
28   }
29   *rX = X;
30   *rY = Y;
31 }
```

Listing 6.18 Mixed C-assembly listing for Problem 8.8

```
1  void cordic(int target,     int *rX, int *rY) {
2    0:  e92d40f0    stmdb   sp!, {r4, r5, r6, r7, lr}
3    4:  e1a06001    mov r6, r1
4    8:  e1a07002    mov r7, r2
5  int X, Y, T, current;
6  unsigned step;
7  X       = AG_CONST;
8    c:  e59fe054    ldr lr, [pc, #84]
9  Y       = 0;
10   10:  e3a02000    mov r2, #0  ; 0x0
11 current = 0;
12   14:  e1a01002    mov r1, r2
13 for(step=0; step < 20; step++) {
14   18:  e1a0c002    mov ip, r2
15   1c:  e59f5048    ldr r5, [pc, #72]
16   20:  e1a04005    mov r4, r5
17   if (target > current) {
18   24:  e1500001    cmp r0, r1
19     T          = X - (Y >> step);
20   28:  c04e3c52    subgt   r3, lr, r2, asr ip
```

```
21        Y            = (X >> step) + Y;
22   2c:   c0822c5e    addgt   r2, r2, lr, asr ip
23        X            = T;
24   30:   c1a0e003    movgt   lr, r3
25        current   += angles[step];
26   34:   c795310c    ldrgt   r3, [r5, ip, lsl #2]
27   38:   c0811003    addgt   r1, r1, r3
28     } else {
29        T            = X + (Y >> step);
30   3c:   d08e3c52    addle   r3, lr, r2, asr ip
31        Y            = -(X >> step) + Y;
32   40:   d0422c5e    suble   r2, r2, lr, asr ip
33        X            = T;
34   44:   d1a0e003    movle   lr, r3
35        current   -= angles[step];
36   48:   d794310c    ldrle   r3, [r4, ip, lsl #2]
37   4c:   d0631001    rsble   r1, r3, r1
38   50:   e28cc001    add     ip, ip, #1
39   54:   e35c0013    cmp     ip, #19
40   58:   8586e000    strhi   lr, [r6]
41     }
42   }
43   *rX = X;
44   *rY = Y;
45   5c:   85872000    strhi   r2, [r7]
46 }
47   60:   88bd80f0    ldmhiia sp!, {r4, r5, r6, r7, pc}
48   64:   ea000007    b   24 <cordic+0x24>
49   68:   09b74eda    ldmeqib r7!, {r1, r3, r4, r6, r7, r9, sl,
50                             fp, lr}
51   6c:   00000000    andeq   r0, r0, r0
```

Chapter 7
System On Chip

Abstract There is no generally accepted, universally-available machine abstraction above that of a RISC processor. However, the RISC is a key component in a very successful heterogeneous architecture: the System-on-Chip. A system-on-chip architecture combines one or more microprocessors, an on-chip bus system, several dedicated coprocessors, and on-chip memory, all on a single chip. An SoC architecture provides general-purpose computing capabilities along with a few highly specialized functions, adapted to a particular design domain. This chapter reviews the cast of players in the system-on-chip concept, and it describes its key characteristics. The chapter also documents how GEZEL SoC models can be constructed as a combination of custom FSMD hardware modules and simulation primitives to capture the RISC cores.

7.1 The System-on-Chip Concept

Figure 7.1 illustrates a typical System-on-chip. It combines several components on a bus system. One of these components is a microprocessor (typically a RISC), which plays the role of conductor in the SoC. Other components include on-chip memory, off-chip-memory interfaces, dedicated peripherals, hardware coprocessors, and component-to-component communication infrastructure.

The application domain greatly affects the type of hardware peripherals, the size of memories, and the nature of on-chip communications. A particular configuration of these elements is also called a *platform*. Just like a personal computer is a platform for general-purpose computing, a system-on-chip is a platform for domain-specialized computing, i.e., for an ensemble of applications that are typical for a given application domain. Examples of application domains are mobile telephony, video processing, or high-speed networking. The set of applications in the video-processing domain for example could include image transcoding, image compression and decompression, image color transformations, and so forth. Domain specialization in a System-on-Chip is advantageous for several reasons.
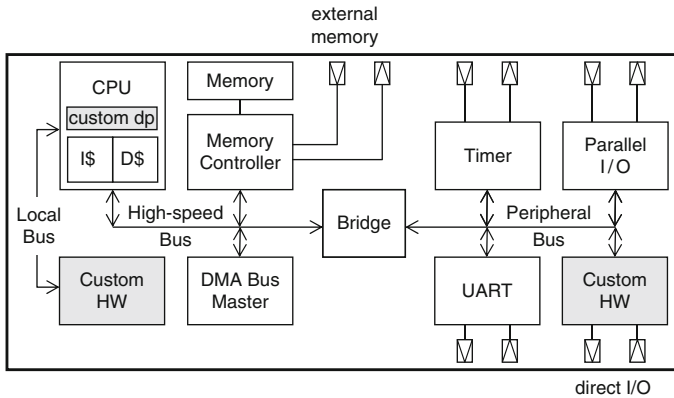
Fig. 7.1 Generic template for a system-on-chip

- The *specialization* of the platform ensures that its processing efficiency is higher compared to that of general-purpose solutions. Increased processing efficiency means lower power consumption (longer battery lifetime) or higher absolute performance.
- The *flexibility* of the platform ensures that it is a reusable solution that works over multiple applications. As a result, the design cost-per-application decreases, applications can be developed faster, and the SoC itself becomes cheaper because it can be manufactured for a larger market.

7.1.1 The Cast of Players

An architecture such as in Fig. 7.1 can be analyzed along 4 orthogonal dimensions: control, communication, computation, and storage. The role of central controller role is given to the microprocessor, who is responsible of issuing control signals to, and collecting status signals from, the various components in the system. The microprocessor may or may not have a local instruction memory. In case it does not have a local instruction memory, caches may be utilized to improve instruction-memory bandwidth. The I$ and D$ symbols in Fig. 7.1 represent the instruction- and data-caches in the microprocessor. In the context of an SoC architecture, these caches will only benefit the microprocessor they are serving. This obvious fact has an often-ignored consequence: whenever the microprocessor needs to interact with a peripheral, it needs to transfer data beyond the cache. Thus, while the microprocessor in an SoC is in the role of central commander, it is not a good idea to make the microprocessor also responsible for data-movement or data-handling.

The SoC implements communication using system-wide busses. Each bus is a bundle of signals including address, data, control, and synchronization signals. The data transfers on a bus are expressed as read- and write-operations with a particular

memory address. The bus control lines indicate the nature of the transfer (read/write, size, source, destination), while the synchronization signals ensure that the sender and receiver on the bus are aligned in time during a data transfer. Each component connected to a bus will respond to a particular range of memory addresses. The ensemble of components can thus be represented in an *address map*, a list of all system-bus addresses relevant in the system-on-chip.

It is common to split SoC busses into *segments*. Each segment connects a limited number of components, grouped according to their communication needs. In the example, a high-speed communication bus is used to interconnect the microprocessor, a high-speed memory interface, and a Direct Memory Access (DMA) controller. A DMA is a device specialized in performing block-transfers on the bus, for example to copy one memory region to another. Next to a high-speed communication bus, you may also find a peripheral bus, intended for lower-speed components such as a timer and input–output peripherals. Segmented busses are interconnected with a *bus bridge*, a component that translates bus transfers from one segment to another segment. A bus bridge will only selectively translate transfers from one bus segment to the other. This selection is done based on the address map.

The bus control lines of each bus segment are under command of the *bus master*, the component that decides the nature of a given bus master. Other components, the *bus slaves*, will follow the directions of the bus master. Each bus segment can contain one or more bus masters. In case there are multiple masters, the identity of the bus master can be rotated among bus-master components at run time. In that case, a *bus arbiter* will be needed to decide which component can become a bus master for a given bus transfer. A bus bridge can be either master or bus slave, depending on the direction of the transfers. For example, when going from the high-speed bus to the peripheral bus, the bus bridge will act as a bus slave on the high-speed bus and as a bus master on the peripheral bus. Each of the transfers on the high-speed bus and on the peripheral bus will be handled independently. Therefore, the segmentation of busses using bus bridges leads to a dual advantage. First, bus segments can group components with matching read- and write-speed together, thus providing optimal usage of the available bus bandwidth. Second, the bus segments enable communication parallelism.

7.1.2 SoC Interfaces for Custom Hardware

Let us consider the opportunities to attach custom hardware modules in the context of an SoC architecture. In the context of this chapter, a "custom hardware module" means a dedicated digital machine described as an FSMD or as a microprogrammed machine. Eventually, all custom hardware will be under control of the central processor in the SoC. The SoC architecture offers several possible hardware-software interfaces to attach custom hardware modules. Three approaches can be distinguished in Fig. 7.1 as shaded blocks.

- The most general approach is to integrate a custom hardware module as a standard peripheral on a system bus. The microprocessor communicates with the custom hardware module by means of read/write memory accesses. Of course, the memory adddresses occupied by the custom hardware module cannot be used for other purposes (i.e., as addressable memory). For the memory addresses occupied by the custom hardware module, the microprocessors' cache has no meaning, and the caching effect is unwanted. Microcontroller chips with many different peripherals typically use this memory-mapped strategy to attach peripherals. The strong point of this approach is that a universal communication mechanism (memory read/write operations) can be used for a wide range of custom hardware modules. The corresponding disadvantage, of course, is that such a bus-based approach to integrate hardware is not very scalable in terms of performance: the system bus quickly becomes a bottleneck when intense communication between a microprocessor and the attached hardware modules is needed.
- A second mechanism is to attach custom hardware through a local bus system or coprocessor interface provided by the microprocessor. In this case, the communication between the hardware module and the microprocessor will follow a dedicated protocol, defined by the local bus system or coprocessor interface. In comparison to system-bus interfaces, coprocessor interfaces have a high-bandwidth and a low latency. The microprocessor may also provide a dedicated set of instructions to communicate over this interface. Typical coprocessor interfaces do not involve a memory addresses. This type of coprocessor obviously requires a microprocessor with a coprocessor- or local-bus interface.
- Microprocessors may also provide a means to integrate a custom-hardware datapath inside of the microprocessor. The instruction set of the microprocessor is then extended with additional, new instructions to drive this custom hardware. The communication channel between the custom datapath and the processor is typically through the processor register file, resulting in a very high communication bandwidth. However, the very tight integration of custom hardware with a microprocessor also means that the traditional bottlenecks of the microprocessor are also a bottleneck for the custom-hardware modules. If the microprocessor is stalled because of external events (such as memory-access bandwidth), the custom data-datapath is stalled as well.

These observations show that there is no single *best* way to integrate hardware and software. There are many possible solutions, each with their advantages and disadvantages. Selecting the right approach involves trading-off many factors, including the required communication bandwidth, the design complexity of the custom hardware interface, the software, the available design time, and the overall cost budget. The following chapters will cover some of the design aspects of hardware-software interfaces. In the end, however, it is the hardware-software codesigner who must identify the integration opportunities of a given System-on-Chip architecture, and who must realize their potential.

7.2 Four Design Principles in SoC Architecture

A SoC is very specific to an application domain. Are there any guiding design principles that are relevant to the design of *any* SoC? This section will address this question in more detail. The objective is to clarify four design principles that govern the majority of modern SoC architectures. These four principles include heterogeneous and distributed communications, heterogeneous and distributed data processing, heterogeneous and distributed storage, and hierarchical control. We will review each of these four points in more detail. This will demonstrate the huge potential of the SoC, in particular when the technological dimension is brought into the picture.

7.2.1 Heterogeneous and Distributed Data Processing

A first prominent characteristic of an SoC architecture is heterogeneous and distributed data processing. An SoC may contain multiple independent (distributed) computational units. Moreover, these units can be heterogenous and include FSMD, microprogrammed engines, or microprocessors.

One can distinguish three forms of data-processing parallelism. The first is word-level parallelism, which enables the parallel processing of multiple bits in a word. The second is operation-level parallelism, which allows multiple instructions to be executed simultaneously. The third is task-level parallelism, which allows multiple independent threads of control to be executed independently. Word-level parallelism and operation-level parallelism are available on all of the machine architectures we discussed so far: FSMD, Microprogrammed machines, RISC, and also SoC. However, only an SoC supports task-level parallelism. Note that multithreading in a RISC is not task-level parallelism; it is task-level concurrency on top of a sequential machine.

Each of the computational units in an SoC can be specialized to a particular function. The overall SoC therefore includes a collection of *heterogeneous* computational units. For example, a digital signal processing chip in a camera may contain specialized units to perform image-processing. Computational specialization is the key to obtain an efficient chip. In addition, the presence of all forms of parallelism (word-level, operation-level, task-level) ensures that an SoC can fully exploit the technology.

In fact, integrated circuit technology is *extremely* effective to provide computational parallelism. Consider the following numerical example. A 1-bit full-adder cell can be implemented in about 28 transistors. The Intel Core 2 processor contains 291 million transistors in 65 nm CMOS technology. This is sufficient to implement 325,000 32-bit adders. Assuming a core clock frequency of 2.1 GHz, we thus find that the silicon used to create a Core 2 can theoretically implement 682,000 Giga-operations per second. We call this number the *intrinsic computational efficiency* of silicon. Of course, we don't know how to build a machine that would have this efficiency, let alone that such a machine would be able to cope with the resulting

power density. The intrinsic computational efficiency merely represents a theoretical upperbound.

$$\text{Eff}_{\text{intrinsic}} = \frac{291.10^6}{28.32}.2.1 \approx 682{,}000 \, \text{Gops} \tag{7.1}$$

The actual Core 2 architecture handles around 9.24 instructions per clock cycle, in a single core and in the most optimal case. The actual efficiency of the 2.1 GHz Core 2 therefore is 19.4 Giga-operations per second. We make the (strong) approximation that these 9.24 instructions each correspond to a 32-bit addition, and call the resulting throughput the actual Core2 efficiency. The ratio of the intrinsic Core2 efficiency over the actual Core2 efficiency illustrates the efficiency of silicon technology compared to the efficiency of a processor core architecture.

$$\text{Efficiency} = \frac{\text{Eff}_{\text{intrinsic}}}{\text{Eff}_{\text{actual}}} \approx \frac{682{,}000}{19.4} = 35{,}150 \tag{7.2}$$

Therefore, bare silicon can implement computations 35,000 times more efficient than a Core2! While this is a very simple and crude approximation, it demonstrates why specialization of silicon using multiple, independent computational units is so attractive.

7.2.2 Heterogeneous and Distributed Communications

The central bus in a system-on-chip is a critical resource. It is shared by many components in an SoC. One approach to prevent this resource from becoming a bottleneck is to split the bus into multiple bus segments using bus bridges. The bus bridge is therefore a mechanism to create distributed on-chip communications. The on-chip communication requirements typically show large variations over an SoC. Therefore, the SoC interconnection mechanisms should be heterogeneous as well. There may be shared busses, point-to-point connections, serial connections, and parallel connections.

Heterogeneous and distributed SoC communications enable a designer to exploit the on-chip communication bandwidth. In modern technology, this bandwidth is extremely high. An illustrative example by Chris Rowen mentions the following numbers. In a 90 nm 6-layer metal processor, we can reasonably assume that metal layers will be used as follows: two metal layers are used for power and ground, respectively, two metal layers are used to route wires in the X direction, and two metal layers are used to route wires in the Y direction. The density of wires in a 90 nm process is 4 wires per micron (one thousandth of a millimeter), and the bit frequency is at 500 MHz. Consequently, in a chip of 10 millimeter on the side, we will have 40,000 wires per layer on a side. Such a chip can transport 80,000 bits in any direction at a frequency of 500 MHz. This corresponds to 40 terabits per second! Consequently, on-chip communications have a high bandwidth – the real challenge is how to organize it efficiently.

The same cannot be said for *off-chip* communication bandwidth. In fact, off-chip bandwidth is very expensive compared to on-chip bandwidth. For example, consider the latest Hypertransport 3.1 standard, a serial link developed for high-speed processor interconnect. Usually, a (high-end) processor will have one to four of such ports. The maximum aggregate data bandwidth for such a port is around 20.8 GByte per second. Thus, we will find less than 80 GByte per second input/output bandwidth on a state-of-the-art processor today. That is still 62 times less than the 40 Tb/s on-chip bandwidth in a standard 90 nm CMOS process! This clearly indicates the potential of on-chip integration.

7.2.3 Heterogeneous and Distributed Storage

A third characteristic of System-on-Chip architectures is a distributed and heterogeneous storage architecture. Instead of a single, central memory, an SoC will use a collection of dedicated memories. Processors and microcoded engines may contain local instruction memories. Processors may also use cache memories to maintain local copies of data and instructions. Coprocessors and other active components will use local register files. Specialized accelerators can use dedicated memories for specific applications such as for video frame buffering or as local scratchpad.

This storage is implemented with a collection of different memory technologies. There are five broad categories of silicon-based storage available today.

- *Registers* are the fastest type of memory available. Registers are also called *foreground memory*. They reside the closest to the computation elements of an architecture. A register does not have the concept of addressing unless it is organized in a *register file*.
- *Dynamic Random Access Memory* (DRAM) provides cheap storage at very high densities. Today (2009), one in 4 memory chips sold is a DRAM (or a related category such as SDRAM, DDR, DDR2). DRAM, and all the following categories are called *background memory*. Unlike registers, DRAM cells use a different processing technology as normal logic. Therefore, DRAM memories are unsuited to be integrated on a single-chip SoC.
- *Static Random Access Memory* (SRAM) is used where fast read–write storage is required. SRAM has lower density and higher power consumption than DRAM. It is not used for the bulk of computer storage, but rather for specialized tasks such as caches, video buffers, and so on. On the plus side, SRAM can be implemented with the same process technology as normal logic gates. It is therefore easy to mix SRAM and computational elements in an SoC.
- *Nonvolatile Read-Only Memory* (NVROM) is used for applications that only require read access on a memory, such as for example to store the instructions of a program. Nonvolatile memories have a higher density than SRAM. There is a range of technologies that can be used to implement a NVROM (mask-programmed ROM, PROM, EPROM, EEPROM).

Table 7.1 Types of memories

Type	Register Register file	DRAM	SRAM	NVROM (ROM, PROM, EPROM)	NVRAM (Flash, EEPROM)
Cell size (bit)	10 transistors	1 transistor	4 transistors	1 transistor	1 transistor
Retention	0	Tens of ms	0	∞	10 years
Addressing	Implicit	Multiplexed	Non-muxed	Non-muxed	Non-muxed
Access time	Less then 1 ns	Less then 20 ns	Less then 10 ns	20 ns	20 ns (read) 100 μs (write)
Power consumption	High	Low	High	Very low	Very low
Write durability	∞	∞	∞	1-time	One million times

- *Nonvolatile Random Access Memory* (NVRAM) is used for applications that need read–write memories that do not loose data when power is removed. The read- and write-speed in a NVRAM can be asymmetrical (write being slower) so that in the limit the distinction between NVROM and NVRAM is not sharp.

Table 7.1 summarizes the key characteristics of these different types of memory. The entries in the table have the following meaning:

- The *cell size* is the silicon area required to store a single bit. The cell size is only part of the complete memory – additional hardware is needed for address decoding, multiplexing bits from the data bus, and so on. High-density storage technologies use only a single transistor per bit, and make use of low-level physical properties of that transistor (parasitic capacitance, floating gate, etc) to hold the bit.
- The *retention time* expresses how long a bit can be held in a nonpowered memory.
- The *addressing* mechanism shows how bits are retrieved from memory. In multiplexed addressing, such as used by DRAM, the address is cut in two parts which are provided sequentially to the memory.
- The *access time* is the time required to fetch a data element from memory. Note that the access time is a coarse number, as it does not capture the detailed behavior of a memory. For example, in NVRAM technologies, the read and write access time is asymmetrical: write takes longer than read. Modern DRAM memories are very efficient in providing consecutive memory locations (burst access), but individual random locations take longer. Finally, modern memories can be internally pipelined, such that they can process more than one read or write command at the same time.
- The *power consumption* is a qualitative appreciation for the power consumption of a memory, as measured per access and per bit. Fast read/write storage is much more power-hungry than slow read-only storage.

The presence of distributed storage significantly complicates the concept of a centralized memory address space, which is so useful in SoC. As long as the data

within these distributed memories is local to a single component, this does not cause any problem. However, it becomes troublesome when data needs to be shared among components. First, when multiple copies of a single data item exist in different memories, all these copies need to be kept consistent. Second, updating of a shared data item needs to be implemented in a way that will not violate data dependencies among the components that share the data item. It is easy to find a few examples where either of these two requirements will be violated (see Problem 7.1).

In 1994, Wulf and McKee wrote a paper entitled *Hitting the Memory Wall: Implications of the Obvious*. The authors used the term *memory wall* to indicate the point at which the performance of a (computer) system is determined by the speed of memory, and is no longer dependent on processor speed. While the authors conclusions were made for general-purpose computing architectures, their insights are also valid for mixed hardware/software systems such as those found in System-on-Chip. Wulf and McKee observed that the performance improvement of processors over time was higher than the performance improvement of memories. They assumed 60% performance improvement per year for processors, and 10% for memories – valid numbers for systems around the turn of the century. The memory wall describes the specific moment when the performance of a computer system becomes performance-constrained by the memory subsystem. It is derived as follows:

In general-purpose computer architectures with a cache, the memory access time of a processor with cycle time t_c, cache hit-rate p, and memory access time t_m, is given by

$$t_{\text{avg}} = p \times t_c + (1 - p) \times t_m \tag{7.3}$$

Now assume that on the average, one in 5 processor instructions will require a memory reference. Under this assumption, the system becomes memory-access constrained when t_{avg} reaches 5 times the cycle time t_c. Indeed, no matter how good the processor is, it will spend more time waiting for memory than executing instructions. This point is called the memory wall.

How likely is it for a computer to hit the memory wall? To answer this question, we should observe that the cache hit rate p cannot be 100%. Data-elements stored in memory have to be fetched at least once from memory before they can be stored in a cache. Let us assume a factor of $p = 0.99$ (rather pessimistic), a cycle time t_c of 1, and a cycle time t_m of 10.

Under this assumption,

$$(t_{\text{avg}})_{\text{now}} = 0.99 \times 1 + 0.01 \times 10 = 1.09 \tag{7.4}$$

One year from now, memory is 1.1 times faster and the processor is 1.6 times faster. Thus, one year from now, t_{avg} will change to

$$(t_{\text{avg}})_{\text{now}+1} = 0.99 \times 1 + 0.01 \times 10 \times \frac{1.6}{1.1} = 1.135 \tag{7.5}$$

And after N years it will be

$$(t_{\text{avg}})_{\text{now}+\text{N}} = 0.99 \times 1 + 0.01 \times 10 \times \frac{1.6}{1.1}^N \tag{7.6}$$

The memory wall will be hit when t_{avg} equals 5, which can be solved according to the above equation to be $N = 9.8$ years. Now, more than a decade after this 1994 paper, it is unclear if the doomsday scenario has really materialized. Many other factors have changed in the meantime as well, making the formula an unreliable predictor. For example, current processor workloads are very different than those from 10 years ago. Multimedia, gaming, and internet have become a major factor. In addition, current processor scaling is no longer done by increasing clock frequency but by more drastic changes at the architecture-level (multiprocessors). Finally, new limiting factors, such as power consumption density and technological variability, have retargeted the quest for performance into one for efficiency.

Despite this, memory remains a crucial element in SoC design, and it still has a major impact on system performance. In many applications, the selection of memory elements, their configuration and layout, and their programming is a crucial design task.

7.2.4 Hierarchical Control

The final concept in the architecture of an SoC is the hierarchy of control among components. A hierarchy of control means that the entire SoC operates as a single logical entity. This implies that all components in an SoC will need to synchronize at some point. For example, consider a C program (on a RISC) that uses a coprocessor implemented as a peripheral. The C program will need to send arguments to the coprocessor, wait for the coprocessor to finish execution, and finally retrieve the result from the coprocessor. From the perspective of the coprocessor, the custom hardware will first wait for someone to provide it with operands over the peripheral bus, next it will do active processing, and finally it will signal completion of the operation to the caller (for example, by setting a status flag). It is clear that the control on the RISC processor as well as on the coprocessor are not independent. The local controller in the coprocessor can be developed with an FSM or a microprogramming technique. The RISC processor will maintain overall control in the system and distribute commands to custom hardware.

The design of a good control hierarchy is a challenging problem. On the one hand, it should exploit the distributed nature of the SoC as good as possible – this implies doing many things in parallel. On the other hand, it should minimize the number of conflicts that arise as a result of running things in parallel. Such conflicts can be the result of overloading the available bus-system or memory bandwidth, or overscheduling a coprocessor. Due to the control hierarchy, all components of an SoC are logically connected to each other, and each of them may cause a system bottleneck. The challenge for the SoC designer (or platform programmer) is to be aware of the location of such system bottlenecks and to control them.

7.3 Example: Portable Multimedia System

In this section we illustrate the four key characteristics discussed earlier (distributed and heterogeneous memory, interconnect, computing, and hierarchical control) by means of an example. Figure 7.2 shows the block diagram of a digital media processor by Texas Instruments. The chip is used for the processing of still images, video, and audio in portable, battery-operated devices. It is manufactured in 130 nm CMOS, and the entire chip consumes no more than 250 mW in default-preview mode and 400 mW when video encoding and decoding is operational.

The chip supports a number of device modes. Each mode corresponds to typical user activity. The modes include the following:

- Live preview of images (coming from the CMOS imager) on the video display.
- Live-video conversion to a compressed format (MPEG, MJPEG) and streaming of the result into an external memory.
- Still-image capturing of a high-resolution image and conversion to JPEG.
- Live audio capturing and audio compression to MP3, WMA, or AAC.
- Video decode and playback of a recorded stream onto the video display.
- Still image decode and playback of a stored image onto the video display.
- Audio decode and playback.
- Photo printing of a stored image into a format suitable for a photo printer.

The central component of the block diagram in Fig. 7.2 is the SDRAM memory controller. During operation, image data is stored in off-chip SDRAM memory. The SDRAM controller organizes memory traffic to this large off-chip memory.
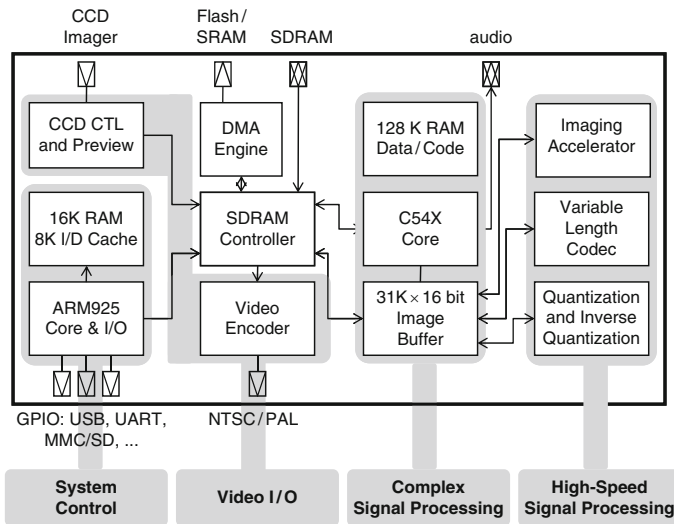

Fig. 7.2 Generic template for a system-on-chip

Around the controller, four different subsystems are organized. They deal with video input/output, complex signal processing, high-speed signal processing, and system control, respectively.

The video input/output subsystem includes the CCD sensor interface and a video encoder. The CCD interface is capable of sampling up to 40 MHz at 12 bits per pixel, and it needs to provide high-resolution still images (2–5 Mpixels) as well as moving images (up to 30 frames/s at 640×480 pixels). Most CCD sensors record only a single color per pixel. Typically there are 25% red pixels, 25% blue pixels, and 50% green pixels, which are arranged in a so-called Bayer pattern. This means that, before images can be processed, the missing pixels need to be filled in (interpolated). This task is a typical example of streaming and dedicated processing. The video subsystem also contains a video encoder, capable of merging two video streams on screen, and providing picture-in-picture functionality. The video coder also includes on-screen menu subsystem functionality. The output of the video coder goes to an attached LCD or a TV. The video coder provides in the range of 100 operations per pixel, while the power budget of the entire video subsystem is less than 100 mW. Such numbers are clearly out of range for a software-driven processor.

The complex signal-processing subsystem is created on top of a C54x digital signal processor (DSP) with 128 Kbytes of RAM and operating at 72 MHz. The DSP processor performs the main processing and control logic for the wide range of signal processing algorithms that the device has to perform (MPEG-1, MPEG-2, MPEG-4, WMV, H.263, H.264, JPEG, JPEG2K, M-JPEG, MP3, AAC, WMA).

The third subsystem is the high-speed signal processing subsystem, needed for encoding and decoding of moving images. Three coprocessors deliver additional computing muscle for the cases where the DSP falls short. There is a DMA engine that helps moving data back and forth between the memory attached to the DSP and the coprocessors. The three coprocessors implement the following functions: The first one is a SIMD-type of coprocessor to provide vector-processing for image processing algorithms. The second is a quantization coprocessor to perform quantization in image encoding algorithms. The third coprocessor performs Huffman encoding for those image encoding standards. The coprocessor subsystem increases the overall processing parallelism of the chip, as they can work concurrently with the DSP processor. This allows the system clock to be decreased.

Finally, the system ARM subsystem is the overall system manager. It synchronizes and controls the different subcomponents of the system. It also provides interfaces for data input/output and user interface support.

Each of the four properties discussed in the previous section can be identified in this chip.

- The SoC contains *heterogeneous and distributed processing*. There is hardwired processing (video subsystem), signal processing (DSP), and general-purpose processing on an ARM processor. All of this processing can have overlapped activity.
- The SoC contains *heterogeneous and distributed interconnect*. Instead of a single central bus, there is a central "switchbox" that multiplexes accesses to the off-chip memory. Where needed, additional dedicated interconnections are

implemented. Some examples of dedicated interconnections include the bus between the DSP and its instruction memory, the bus between the ARM and its instruction memory, and the bus between the coprocessors and their image buffers.

- The SoC contains *heterogeneous and distributed storage*. The bulk of the memory is contained within an off-chip SDRAM module, but there are also dedicated instruction memories attached to the TI DSP and the ARM, and there are dedicated data memories acting as small dedicated buffers.
- Finally, there is a hierarchy of control that ensures the overall parallelism in the architecture is optimal. The ARM will start/stop components and control data streams depending on the mode of the device.

The DM310 chip is an excellent example of the balancing effort required to support real-time video and audio in a portable device. The architects (hardware and software people) of this chip have worked closely together to come up with the right balance between flexibility and energy-efficiency.

7.4 SoC Modeling in GEZEL

In the last section of this chapter, we consider how a System-on-Chip can be modeled in GEZEL, building on our previous experience with FSMD design, microprogrammed design, and general-purpose processors. The approach used by GEZEL is to implement SoC modeling through the inclusion of instruction-set simulators in the simulator. A typical example configuration is shown in Fig. 7.3. It includes several components. Custom hardware modules are captured as FSMD models. The microprocessor cores are captured as custom library modules, called `ipblock`.
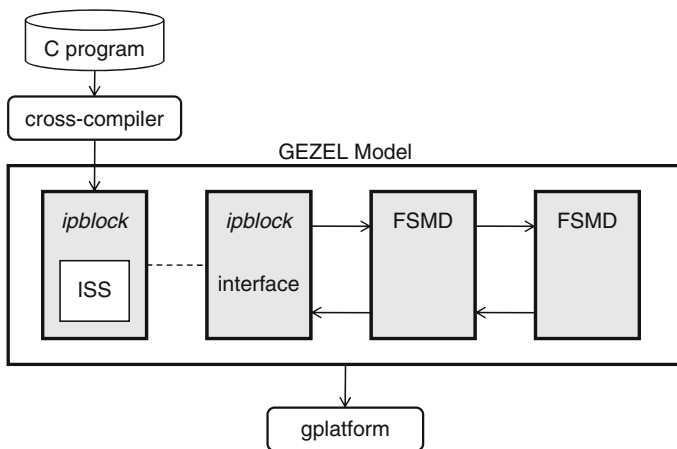

Fig. 7.3 A GEZEL system-on-chip model

Each microprocessor core, when integrated in an SoC, can offer different types of interfaces. Each of these interface types is captured in a different `ipblock`. The software executed by the microprocessor core is developed in C or assembly and converted to binary format using a cross-compiler or cross-assembler. The binary is used to initialize the instruction-set simulator, embedded in an `ipblock`. The entire system simulation is executed by the GEZEL platform simulator `gplatform`.

7.4.1 An SoC with a StrongARM Core

We describe some of the features of SoC modeling, including two cores included in the `gplatform` simulator. The first core is a StrongARM core, modeled with the Simit-ARM v2.1 simulator developed by W. Qin at Boston University. Listing 7.1 shows a simple, standalone ARM core, similar to the model covered in Listing 6.4. Line 2 of this listing tells that this module is an ARM core with attached instruction memory (`armsystem`). Line 3 names the ELF executable that must be loaded into the ARM simulator when the simulation starts. The format of an `ipblock` is generic and can be applied to many different types of cosimulation entities.

The model shown in Listing 7.1 is not very interesting since it does not show any communication between hardware and software. We will extend this model with a *memory-mapped* interface, as shown in Listing 7.2. Figure 7.4 illustrates how this model corresponds to a System-on-Chip architecture. In Listing 7.2, a hardware-to-software interface is defined on lines 6–10. This particular example shows a memory-mapped interface. The interface has a single output port `data`. The port can be thought of as a register that is written by the software. The software can update the value of the register by writing to memory address `0x80000000`. After each update, the output port `data` will hold this value until the software writes to the register again. Note that the association between the memory-mapped interface and the ARM core is established based on the name of the core (line 8 in Listing 7.2). Lines 12–28 show a custom hardware module, modeled as an FSMD, which is attached to this memory-mapped interface. The FSM uses the least-significant bit from the memory-mapped register as a state transition condition. Whenever this bit changes from 0 to 1, the FSMD will print the value of the memory-mapped register.

Listing 7.1 A GEZEL top-level module with a single ARM core

```
1  ipblock myarm {
2    iptype "armsystem";
3    ipparm "exec = hello";
4  }
5
6  system S {
7    myarm;
8  }
```

Listing 7.2 A GEZEL module with an ARM core and a memory-mapped interface on the ARM

```
1   ipblock myarm {
2     iptype "armsystem";
3     ipparm "exec=hello";
4   }
5
6   ipblock port1(out data    : ns(32)) {
7     iptype "armsystemsource";
8     ipparm "core=myarm";
9     ipparm "address = 0x80000000";
10  }
11
12  dp portreader {
13    sig data : ns(32);
14    use myarm;
15    use port1(data);
16    reg changed : ns(1);
17    always   { changed = data[0]; }
18    sfg show { $display($cycle,": The MM interface is now ", $dec,
19                data); }
20    sfg nil  { }
21  }
22  fsm f_portreader(portreader) {
23    initial s0;
24    state   s1;
25    @s0 if (~changed) then (nil)  -> s0;
26                      else (show) -> s1;
27    @s1 if (changed)  then (nil)  -> s1;
28                      else (nil)  -> s0;
29  }
30
31  system S {
32    portreader;
33  }
```

To cosimulate this model, we proceed as follows: First, we cross-compile a C program to run on the ARM. Next, we execute the cosimulation. The following is an example C program that we will run on top of this system architecture.

```
#include <stdio.h>

int main() {
  int y;
  volatile int * a = (int *) 0x80000000;

  *a = 25;
  *a = 0;
  *a = 39;
  *a = 0;

  return 0;
}
```
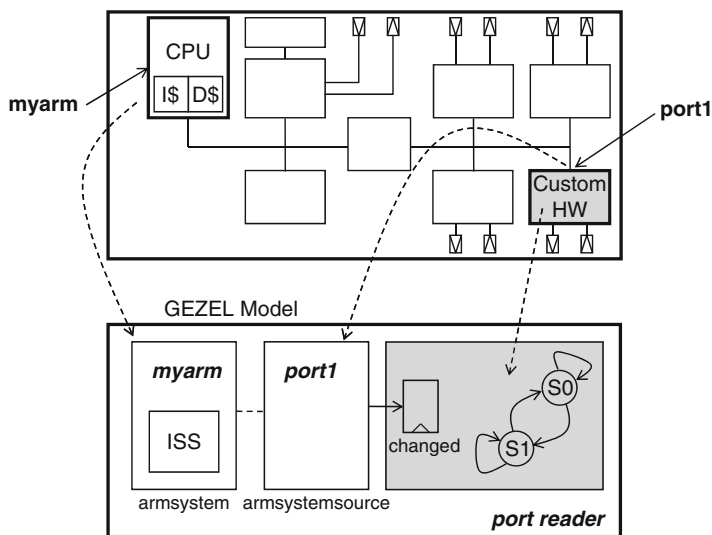
Fig. 7.4 Correspondence of Listing 7.2 to SoC architecture

This program creates a pointer to the absolute memory address `0x80000000`, which corresponds to the memory-mapped port of the custom hardware module in Listing 7.2. The C program then writes a sequence of values to this address. The nonzero values will trigger the `$display` statement shown on line 18 of Listing 7.2. Compilation of this program and execution of the cosimulation are done through the following commands.

```
> arm-linux-gcc -static  hello.c -o hello
> gplatform  armex.fdl
core myarm
armsystem: loading executable [hello]
7063: The MM interface is now 25
7069: The MM interface is now 39
Total Cycles: 7595
```

The cosimulation verifies that data is passed correctly from software to hardware. The first print statement only happens at cycle 7063. This caused by the need to initialize the C runtime environment on the ARM (changing the C runtime environment to a faster library may reduce this delay significantly).

The relation between the GEZEL model and the System-on-Chip architecture, as illustrated in Fig. 7.4, shows that the FSMD captures the internals of a shaded "custom hardware" module in a System-on-Chip architecture. The memory-mapped register captured by `port1` is located at the input of this custom hardware module. Thus, the GEZEL model in Listing 7.2 does not capture the bus infrastructure (the peripheral bus, the bus bridge, the high-speed bus) of the SoC. This has an advantage as well as a disadvantage. On the plus side, the resulting simulation model is easy to build and will have a high simulation speed. On the down side,

the resulting simulation does not capture the bus conflicts that occur in the real SoC architecture, and therefore the simulation results may show a difference with the real chip. Ultimately, the choice of modeling accuracy is with the designer. A more detailed GEZEL model could capture the transactions on an SoC bus as well, but this would cost an additional effort, and the resulting model may simulate at a lower speed. For a cosimulation that focuses on verifying the functionality of a hardware-software codesign, a model such as shown in Listing 7.2 is adequate.

7.4.2 Ping-Pong Buffer with an 8051

As a second example, we show how an 8051 microcontroller core can be cosimulated in a GEZEL system model. Figure 7.5a shows a system with an 8-bit 8051 microcontroller, a dual-port RAM with 64 locations, and a hardware module. The microcontroller, as well as the hardware module, can access the RAM. The 8051 microcontroller has several 8-bit I/O ports, and two of them are used in this design. Port P0 is used to send a data byte to the hardware, while port P1 is used to retrieve a data byte from the hardware.

The idea of this design is the implement a *ping-pong buffer* as follows. The RAM is split up in two sections of 32 locations each. When the 8051 controller is writing into the lower section of the RAM, the hardware will read out the upper section of the RAM. Next, the 8051 will switch to writing the higher section of the RAM, while the hardware module will scan out the lower section of the RAM. This
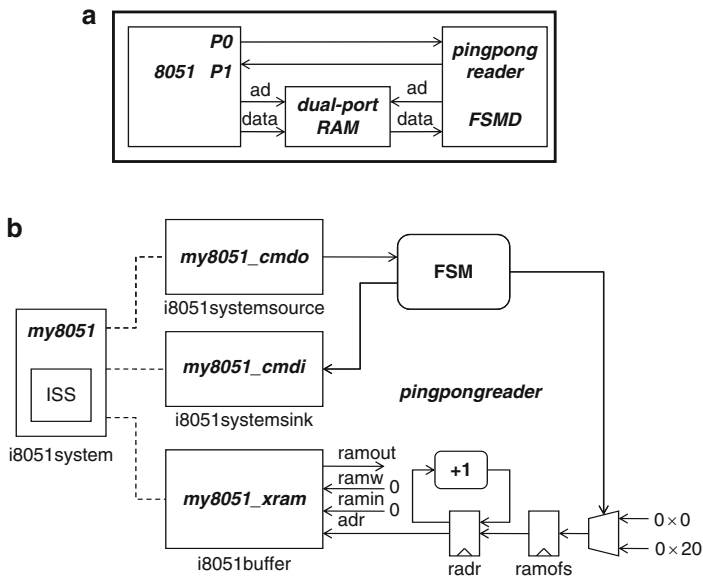

Fig. 7.5 (**a**) 8051 microcontroller with a coprocessor; (**b**) Corresponding GEZEL model structure

double-buffering technique is frequently used to emulate a dual-port shared RAM with single-port RAM modules. Switching between the two operational modes of the system is implemented using a two-way handshake between the 8051 controller and the hardware. The two ports on the 8051 are used for this purpose.

Figure 7.5b and Listing 7.3 show the GEZEL design that implements this model. The 8051 microcontroller is captured with three different ipblock: one for the microcontroller (line 1–6), a second one for port P0 configured as input port (line 8–12), and a third one for port P1 configured as output port (line 14–18). Similar to the StrongARM simulation model, the 8051 microcontroller is captured with an instruction-set simulator, in this case the *Dalton* ISS from the University of California at Riverside. The shared buffer is captured in an ipblock as well, starting on line 20. The shared buffer is specific to the 8051 microcontroller and is attached to the 8051s xbus (expansion bus). The buffer provides one read/write port for the hardware, while the other port is only accessible from within the 8051 software. The hardware module that accesses the ping-pong buffer is listed starting at line 30. The FSMD will first read locations 0 through 0x1F, and next locations 0x20 through 0x3F. The handshake protocol is implemented through the 8051s P0 and P1 port.

Listing 7.3 GEZEL Model of a ping-pong buffer on between an 8051 microcontroller and a hardware FSMD

```
1   ipblock my8051 {
2     iptype "i8051system";
3     ipparm "exec=ramrw.ihx";
4     ipparm "verbose=1";
5     ipparm "period=1";
6   }
7
8   ipblock my8051_cmdo(out data : ns(8)) {
9     iptype "i8051systemsource";
10    ipparm "core=my8051";
11    ipparm "port=P0";
12  }
13
14  ipblock my8051_cmdi(in data : ns(8)) {
15    iptype "i8051systemsink";
16    ipparm "core=my8051";
17    ipparm "port=P1";
18  }
19
20  ipblock my8051_xram(in  idata   : ns(8);
21                      out odata   : ns(8);
22                      in  address : ns(6);
23                      in  wr      : ns(1)) {
24    iptype "i8051buffer";
25    ipparm "core=my8051";
26    ipparm "xbus=0x4000";
27    ipparm "xrange=0x40"; // 64 locations at address 0x4000
28  }
29
30  dp pingpongreader {
31    reg rreq, rack, rid : ns(1);
```

```
32     reg radr              : ns(6);
33     reg ramofs            : ns(6);
34     sig adr               : ns(6);
35     sig ramin, ramout     : ns(8);
36     sig ramw              : ns(1);
37     sig P0o, P0i          : ns(8);
38     use my8051;
39     use my8051_cmdo(P0o);
40     use my8051_cmdi(P0i);
41     use my8051_xram(ramin, ramout, adr, ramw);
42     always        { rreq  = P0o[0];
43                      adr   = radr;
44                      ramw  = 0;
45                      ramin = 0;          }
46     sfg noack     { P0i   = 0;           }
47     sfg doack     { P0i   = 1;           }
48     sfg getramofs0 { ramofs = 0x0;       }
49     sfg getramofs2 { ramofs = 0x20;      }
50     sfg readram0  { radr   = ramofs;  }
51     sfg readram1  { radr   = radr + 1;
52                      $display($cycle, " ram radr ", radr, " data ", ramout);
53                    }
54  }
55
56  fsm fpingpongreader(pingpongreader) {
57     initial s0;
58     state s1, s2, s3, s4, s5, s6;
59     @s0 if (~rreq) then (noack)                  -> s1;
60                 else (noack)                     -> s0;
61
62     @s1 if (rreq) then (doack, getramofs0)       -> s2;
63                         else (noack)             -> s1;
64
65     @s2 (readram0, doack)                        -> s3;
66     @s3 if (radr == 0x5) then (doack)            -> s4;
67                     else (readram1, doack)  -> s3;
68
69     @s4 if (~rreq) then (noack, getramofs2)      -> s5;
70                 else (doack)                     -> s4;
71
72     @s5 (readram0, noack)                        -> s6;
73     @s6 if (radr == 0x25) then (doack)           -> s1;
74                     else (readram1, doack) -> s6;
75  }
76
77  system S {
78    pingpongreader;
79  }
```

Listing 7.4 shows the driver software for the 8051 microcontroller. This software was written for the Small Devices C Compiler (sdcc), a C compiler that supports a broad range of microcontrollers. This compiler directly supports 8051 port access through symbolic names (P0, P1, and so on). In addition, the shared memory accesses can be modeled through an initialized pointer.

Listing 7.4 8051 software driver for the ping-point buffer

```
1  #include <8051.h>
2
3  void main() {
4    int i;
5
6    volatile xdata unsigned char *shared =
7                  (volatile xdata unsigned char *) 0x4000;
8
9    for (i=0; i<64; i++) {
10     shared[i] = 64 - i;
11   }
12
13   P0 = 0x0;
14   while (1) {
15
16     P0 = 0x1;
17     while (P1 != 0x1) ;
18
19     // hw is accessing section 0 here.
20     // we can access section 1
21     for (i = 0x20; i < 0x3F; i++)
22       shared[i] = 0xff - i;
23
24     P0 = 0x0;
25     while ((P1 & 0x1)) ;
26
27     // hw is accessing section 1 here
28     // we can access section 0
29     for (i = 0x00; i < 0x1F; i++)
30       shared[i] = 0x80 - i;
31   }
32 }
```

To cosimulate the 8051 and the hardware, we first cross-compiler the 8051 C
code to binary format. Next, we use the gplatform cosimulator to execute the simula-
tion. Because the microcontroller will execute an infinite program, the cosimulation
is terminated after 60,000 clock cycles. The program output shows that the GEZEL
model scans out the lower part of the ping-pong buffer starting at cycle 36,952,
and the upper part starting at cycle 50,152. The cycle count is relatively high because
the instruction length of a traditional 8051 microcontroller is high: each instruction
takes 12 clock cycles to execute.

```
> sdcc --model-large ram.c
> gplatfrom -c 60000 block8051.fdl
i8051system: loading executable [ramrw.ihx]
0x00    0x00    0xFF    0xFF
0x01    0x00    0xFF    0xFF
36952 ram radr 0/1 data 40
36953 ram radr 1/2 data 3f
36954 ram radr 2/3 data 3e
```

```
36955 ram radr 3/4 data 3d
36956 ram radr 4/5 data 3c
0x00    0x01    0xFF    0xFF
50152 ram radr 20/21 data df
50153 ram radr 21/22 data de
50154 ram radr 22/23 data dd
50155 ram radr 23/24 data dc
50156 ram radr 24/25 data db
Total Cycles: 60000
```

7.5 Summary

System-on-chip architectures implement a combination of flexibility and specialization. The RISC core, the champion of flexibility in embedded designs, takes care of general-purpose processing, and acts as a central controller in SoC. Multiple additional specialized components, including memories, peripherals, and coprocessors, are helping the RISC to address specialized tasks. The interconnect infrastructure, consisting of on-chip bus segments, bus bridges, and specialized connections, help integrating everything together.

All of this makes the SoC a wide-spread paradigm that will be around for some years to come. It is a pragmatic solution that addresses several problems of modern electronic design at the same time. First, an SoC maintains flexibility and is applicable as a platform for several applications within an application domain. This reusability makes the SoC economically advantageous. Compared to a dedicated hardware design, the SoC chip is cheaper, and a given application can be created faster. Second, an SoC contains specialized processing capabilities where needed, and this allows it to be energy-efficient. This greatly expands to potential applications of SoC.

In this chapter, we have reached the summit of our architecture exploration. The key objective of this journey was to investigate how we can make hardware more flexible. We started from custom-hardware models coded as FSMD models. Next, we replaced the fixed finite state machine of an FSMD with a flexible microcoded controller, and obtained a microprogrammed architecture. Third, we turned to RISC processors, which are greatly improved microprogrammed architectures that shield software from hardware. Finally, we used the RISC as a central element in the System-on-Chip architecture.

7.6 Further Reading

System-on-chip is a broad concept with many different dimensions. One of these dimensions is easier and faster design through reuse (Saleh et al. 2006). Another is that SoC technology is critical for modern consumer applications because of the

optimal balance between energy-efficiency and flexibility (Claasen 2006). In recent years, alternative visions on SoC architectures have been given, and an interesting one is given in the book of Rowen (2004). The example on the efficiency of on-chip interconnect comes from the same book.

The definition of *intrinsic computational power* of silicon is elaborated in the ISSCC99 article by Claasen (1999). The paper by Wulf and McKee on the Memory Wall can be found online (Wulf and McKee 1995). In 2004, one of the authors provided an interesting retrospective (McKee 2004).

The digital media processor discussed in this chapter is described in more detail by Talla and colleagues in Talla et al. (2004).

The Dalton 8051 Instruction Set Simulator can be found online, including a synthesizable VHDL view of the 8051 processor (Vahid 2009).

7.7 Problems

7.1. Consider Fig. 7.1 again.

 (a) Explain why the memory area occupied by the UART peripheral cannot be cached by the RISC processor.
 (b) Assume that the high-speed bus would include a second RISC core, which also has an instruction-cache and a data-cache. Explain why, without special precautions, caching can cause problems with the stable operation of the system.
 (c) A quick fix for the problem described in (b) could be obtained by dropping one of the caches in each processor. Which cache must be dropped: the instruction-cache or the data-cache?

7.2. Consider the simple SoC model in Fig. 7.6. Assume that the high-speed bus can carry 200 MWord/sec, and the peripheral bus can carry 30 MWord/sec. The CPU has no cache and requests the following data streams from the system: 80 MWord/sec of read-only bandwidth for instructions, 40 Mword/sec of read/write bandwidth for data, and 2 MWord/sec for Ethernet packet input/output.

 (a) What is the data bandwidth through the bus bridge?
 (b) Assume you have to convert this architecture into a dual-core architecture, where the second core has the same data stream requirements as the first
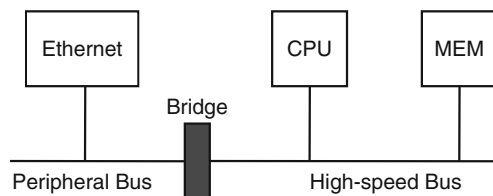

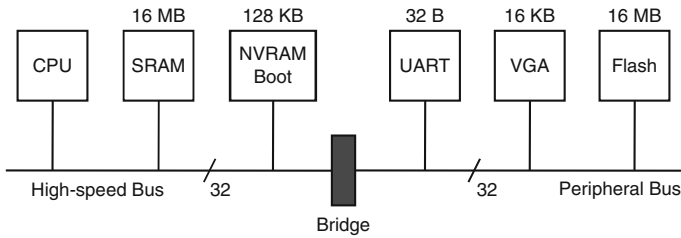**Fig. 7.6** System-on-chip model for Problem 7.2

Fig. 7.7 System-on-chip model for Problem 7.3

core. Discuss how you will modify the SoC. Keep in mind that you can add components and busses, but that you cannot change their specifications. Don't forget to add bus arbitration units, if you need them.

7.3. You have to design a memory map for the SoC shown in Fig. 7.7. The system contains a high-speed bus and a peripheral bus, both of them with a 32-bit address space and both of them carrying words (32 bit). The components of the system include a RISC, a 16 MByte RAM memory, 128 KByte of nonvolatile program memory, a 16 MByte Flash memory. In addition, there is a VGA peripheral and a UART peripheral. The VGA has a 16 KByte video buffer memory, and the UART contains 32 bytes of transmit/receive registers.

(a) Draw a possible memory map for the processor. Keep in mind that the Bus Bridge can only convert bus transfers within a single, continuous address space.
(b) Define what address range can be cached by the processor. A "cached address range" means that a memory-read to an address in that range will result in a backup copy stored in the cache.

7.4. Consider Listing 7.3 and 7.4 again. Modify the GEZEL program and the C program so that the FSMD *writes* into the shared memory, and the C program *reads* from the shared memory. Cosimulate the result to verify the solution is correct.

Part III
Hardware/Software Interfaces

The third part of this book is a systematic overview of hardware/software interfaces, describing all the elements that connect low-level hardware to custom hardware. We start with an overview of on-chip communication busses. Next, we describe various forms of hardware interfaces, along with their counterpart in software. Finally, we show how to develop efficient control mechanisms for custom-hardware modules.

Chapter 8
On-Chip Busses

Abstract The on-chip bus is the backbone of any SoC, and it is a means to efficiently connect various components including processors, memory, and peripherals. The challenges for an on-chip bus are not minor: it has to accommodate a wide range of communication needs with a single, unified architecture. In this chapter we review the key characteristics of the on-chip bus, using several existing on-chip bus standards as examples: ARM/AMBA, IBM/Coreconnect, and Wishbone. We also look at some of the long-term challenges for on-chip interconnect, and how this will affect the design of hardware–software interfaces.

8.1 Connecting Hardware and Software

Over the next few chapters, we will discuss various forms of interconnecting hardware components and software drivers, and in this chapter we will focus on the on-chip bus system. As shown in Fig. 8.1, an on-chip bus connects a microprocessor with a coprocessor. Several elements play a role in the design of the overall system. First, the microprocessor is attached to the on-chip bus through a microprocessor *bus interface*. On the microprocessor, several layers of software transform the bus interface into an *Application Programmers' Interface (API)*, through the use of drivers and low-level software programming. A similar bus interface exists on the hardware coprocessor. The custom hardware module within the coprocessor is isolated from the bus interface by means of a *control shell*, the hardware equivalent of a software driver on the microprocessor. Thus, the link of software to hardware includes a software driver, an on-chip interconnection bus, and a hardware control shell. A good interface (in hardware or software) will shield the software programmer or hardware designer as much as possible from the low-level communication details. This makes the design of software drivers and hardware control shells a challenging task. A factor further complicating this problem is that bus-systems need to be very flexible and scalable so that they can adapt to a wide range of situations and needs. The bus systems, which we will discuss in the section, will illustrate how this flexibility is achieved.
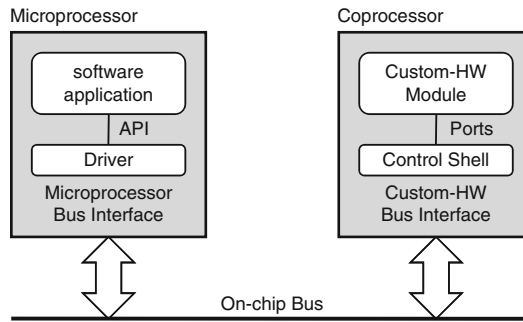
Fig. 8.1 Elements in a bus-based hardware–software interface

8.2 On-Chip Bus Systems

This section describes the basic structure of an on-chip bus, and provides some common terminology and notations.

8.2.1 Some Existing On-Chip Bus Systems

The discussions in this chapter are based on three different bus systems: the AMBA bus, the CoreConnect bus and the Wishbone bus.

- **AMBA** is the bus system used by ARM processors. Originally developed in 1995, the AMBA bus is now in its third generation, and it has evolved into a general on-chip interconnect mechanism. The third generation of AMBA provides three variants of interconnect. A general-purpose, low-bandwidth bus called APB (Advanced Peripheral Bus), a high-speed single-frequency bus called AHB (Advanced High-performance Bus), and a high-speed multifrequency bus called AXI (AMBA Advanced Extensible Interface).
- **CoreConnect** is a bus system proposed by IBM for its PowerPC line of processors. Similar to the AMBA bus, the CoreConnect bus comes in several variants. The main variants include a general-purpose, low-bandwidth bus called OPB (On-chip Peripheral Bus) and a high-speed single-frequency bus called PLB (Processor Local Bus).
- **Wishbone** is an open-source bus system proposed by SiliCore Corporation. The bus is used by many open-source hardware components, for example those in the OpenCores project (http://www.opencores.org). The Wishbone bus is simpler than AMBA and CoreConnect. The specification defines two interfaces (a master-interface and a slave-interface) from which various bus topologies can be derived.

   Rather than describing each bus separately, we will unify them in a generic bus system that reflects the characteristics of each of them. We then will point out how each of AMBA, CoreConnect, and Wishbone implement the features of this generic bus.

8.2.2 Bus Elements

An on-chip bus system implements a bus protocol: a sequence of steps to transfer data in an orderly manner. A typical on-chip bus system will consist of one or more bus segments, as shown in Fig. 8.2. Each bus segment groups one or more bus *masters* with bus *slaves*. Bus bridges are directional components to connect bus segments. A bus-bridge acts as a slave at the input, and as a master at the output. At any particular moment, a bus segment is under the control of either a bus master or a bus arbiter. A bus arbiter's role is to decide what bus master is allowed to control the bus at a particular moment. The arbitration should be done in a fair manner such that no bus masters get permanently locked out of bus access. The bus slaves can never obtain control over a bus segment, but instead have to follow the directions of the bus master that owns the bus.

   A bus system uses an *address space* to organize the communication between components. A sample address space is shown on the right of Fig. 8.2. Usually, the smallest addressable entity in an address space is one byte. Each data transfer over the bus is associated with a given destination address. The destination address determines what component should pick up the data. Bus bridges are address-transparent: they will merge the slave address spaces from their output and transform it to a single slave address space at their input.

   An on-chip bus physically consists of a bundle of wires, which includes one of the following four categories: address wires, data wires, command wires, and synchronization wires.
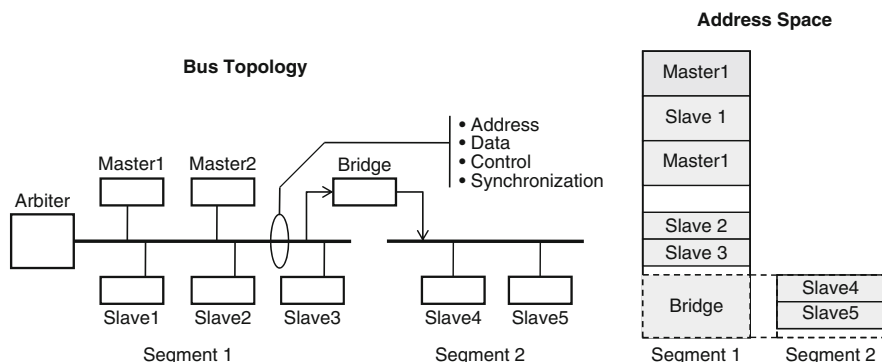

Fig. 8.2 Bus topology and components

- Data wires transfer data items between components. As discussed in the previous chapter, on-chip wiring is very dense, and data wires do not have to be multiplexed. Masters, slaves, and bridges will have separate data inputs and data outputs.
- Address wires carry the address that goes with a given data item. The process of recognizing the destination address is called *address decoding*. One approach to implement address decoding is to implement it inside of the bus slave. Another approach is to perform address decoding centrally, and to distribute the decoded address signals directly to the slaves.
- Command wires describe the nature of the transfer to be performed. Simple commands include *read* and *write*, but larger on-chip bus systems may contain a wide variety of commands, that either speed up or else qualify a given read or write command. Several examples will be discussed later for actual on-chip busses.
- Synchronization wires ensure that bus masters and bus slaves are synchronized during data transfer. Common on-chip bus systems today are synchronous. They use a single clock signal per bus segment: all data, address, and command wires are referenced to the edges of the bus clock. Besides the clock signal, additional control signals are used to synchronize a bus master and bus slave, for example to indicate time-outs and to support request-acknowledge signalling.

8.2.3 Bus Signals

Figure 8.3 shows the physical layout of a typical bus segment with two masters and two slaves. The AND and OR gates in the center of the diagram serve as multiplexers. Several signals are merged this way into bus-wide address and data signals. For example, the address generated by the bus masters is merged into a single bus address, and this bus address is distributed to the bus slaves. Similarly, the data from the masters to the slaves is merged into a bus-wide write-data signal, and the data from the slaves to the masters is merged into a bus-wide read-data signal.

The convention that associates the direction of data with *reading* and *writing* the data is as follows. Writing data means: sending it from a master to a slave.
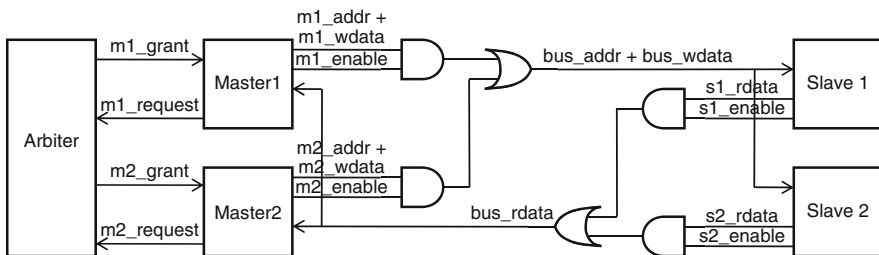
Fig. 8.3 Physical Interconnection of a bus. The *_addr, *_wdata, *_sdata signals are signal vectors. The *_enable, *_grant, *_request signals are single-bit signals

Reading data means: sending it from a slave to a master. Both the AMBA bus and the CoreConnect bus use this convention, and it affects the input/output direction of bus signals on slave or master components.

In Fig. 8.3, each component generates its own bus-enable signal in order to drive a data item or an address onto the bus. This scheme is used by the Coreconnect bus system. For example, when `Master1` will write data, then `m1_enable` will be high while `m2_enable` will be low. If both enable signals would be high, the resulting bus address and write-data will be undefined. Thus, the bus protocol will only work when the components collaborate and follow the rules of the protocol. The AMBA bus system follows a more strict mechanism: in AMBA, the bus-enable signals are generated by a central bus controller (usually the bus arbiter).

Since a bus segment can potentially group a large amount of signals, bus systems will follow a *naming convention*. The objective of a naming convention is to infer the functionality of a wire based on its name. A naming convention is very helpful to read a timing diagram, discussed below. The naming convention can be slightly different for each bus system, as the following examples show.

- The IBM/Coreconnect bus signals are prefixed with the component identifier that drives the signal. For example, `M0_ABus[0:31]` is a 32-bit address bus signal driven by master `M0`. The address signals from the masters are combined into a common bus address signal using a technique similar to Fig. 8.3. The common bus signal has a different prefix and is called `PLB_ABus[0:31]`. PLB stands for Processor Local Bus, the high speed bus of the IBM/Coreconnect bus system.
- The ARM/AMBA bus signals are prefixed with a single letter that indicates which type of bus segment the signal is part of. For example, `HREADY` is a signal of the AMBA Advanced High-Performance Bus (AHB), while `PREADY` is a signal of the AMBA Advanced Peripheral Bus (APB). This means that bus signals are not always unique. For example, in an AMBA bus, two bus slaves can both have an output port `HRDATA`, which would be the read-data signal from the slave to the master. In contrast, the same signals in a Coreconnect PLB bus would be called `SL0_RdDBus` and `SL1_RdDBus`. The cause of this difference is that the names of the AMBA bus signals are attributed with component pins, not with bus wires.
- The Wishbone bus signals are suffixed with a single letter that indicates the direction of the signal. For example, `CLK_I` is an input signal carrying the clock, which `ACK_O` is an output signal carrying the acknowledge part of a handshake. Similar to the AMBA bus, the Wishbone bus defines names for component pins, but not for the bus wires.

8.2.4 Bus Timing Diagram

Because a bus system reflects a complex, highly parallel entity, timing diagrams are extensively used to describe the timing relationships of one signal to the other.
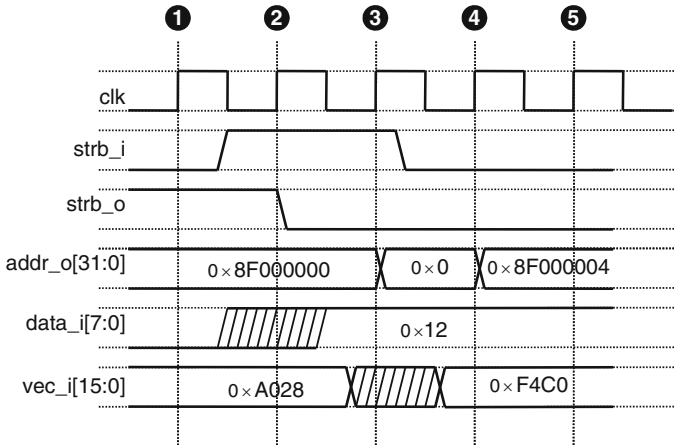
Fig. 8.4 Bus timing diagram notation

Figure 8.4 illustrates a timing diagram of the activities in a generic bus over 5 clock cycles. The clock signal is shown on top, and all signals are referenced to the upgoing clock edge. Dashed vertical lines indicate the timing reference.

When discussing timing diagrams, one must make a distinction between clock edges and clock cycles. The difference between them is subtle but often causes confusion. The term *clock cycle* is ambiguous, because it does not indicate a singular point in time: it has a beginning and an end. A *clock edge*, on the other hand, is an atomic event that cannot be partitioned further (at least not under a single-clock synchronous paradigm). A consequence of the ambiguous term *clock cycle* is that the meaning of the term can change with the direction of the signals. When discussing input signals, designers usually mean that these signals must be stable at the start of the clock cycle, just *before* a clock edge. When discussing output signals on the other hand, designers usually talk about signals that are stable at the end of the clock cycle, so *after* a clock edge. Consider for example signal strb_o in Fig. 8.4. The signal goes down just after the clock edge labeled 2. As strb_o is an output signal, a designer would say that the signal is low in clock cycle 2: the output should reach a stable value after clock edge 2. In contrast, consider the signal strb_i. This input signal is high at the clock edge labeled 2. Therefore, a designer would say this input is high in clock cycle 2. This means that the signal should reach a stable value before clock edge 2. Therefore, talking about clock edges will be less ambiguous than talking about clock cycles, especially when discussing bus protocols that involve multiple interconnected components.

Signal busses of several wires can be collapsed into a single trace in the timing diagram. Examples in Fig. 8.4 are for example addr_o, data_i, and vec_i. The label indicates when the bus changes value. For example, addr_o changes from 0x8F000000 to 0x00000000 at the third clock edge, and it changes back to

`0x8F000004` 1 clock cycle later. Various schemes exist to indicate that a signal or a bus has an unknown or don't care value. The value of `data_i` at the second clock edge and the value of `vec_i` at the third clock edge are all unknown.

Bus timing diagrams are a very useful form to describe the activities on a bus as a function of time. They are also a central piece of documentation for the design of a hardware–software interface.

8.3 Bus Transfers

In this section, we will discuss several examples of data transfer between a bus master and a bus slave. We will also discuss several strategies commonly used by on-chip busses to improve the overall system performance. Over the next few sections, we will describe a generic bus and, where appropriate, define the relationship of signals on this generic bus to AMBA and CoreConnect. Table 8.1 gives a summary of the control signals that will appear in timing diagrams. The exact meaning of these signals will be explained throughout this chapter.

8.3.1 Simple Read and Write Transfers

Figure 8.5 illustrates a write transfer on a generic peripheral bus. A bus master will write the value `0xF000` to address `0x8B800040`. We will assume this bus has only a single bus master and that it does not need arbitration. On clock edge 2, the master takes control of the bus by driving the master select line `m_sel` high. This

Table 8.1 Signals on the generic bus

| Signal name	Meaning
clk	Clock signal. All other bus signals are references to the upgoing clock edge.
m_addr	Master address bus.
m_data	Data bus from master to slave (write operation).
s_data	Data bus from slave to master (read operation).
m_rnw	Read-not-Write. Control line to distinguish read from write operations.
m_sel	Master select signal, indicates that this master takes control of the bus.
s_ack	Slave acknowledge signal, indicates transfer completion.
m_addr_valid	Used in place of m_sel in split-transfers.
s_addr_ack	Used for the address in place of s_ack in split-transfers.
s_wr_ack	Used for the write-data in place of s_ack in split-transfers.
s_rd_ack	Used for the read-data in place of s_ack in split-transfers.
m_burst	Indicates the burst type of the current transfer.
m_lock	Indicates that the bus is locked for the current transfer.
m_req	Requests bus access to the bus arbiter.
m_grant	Indicates bus access is granted.
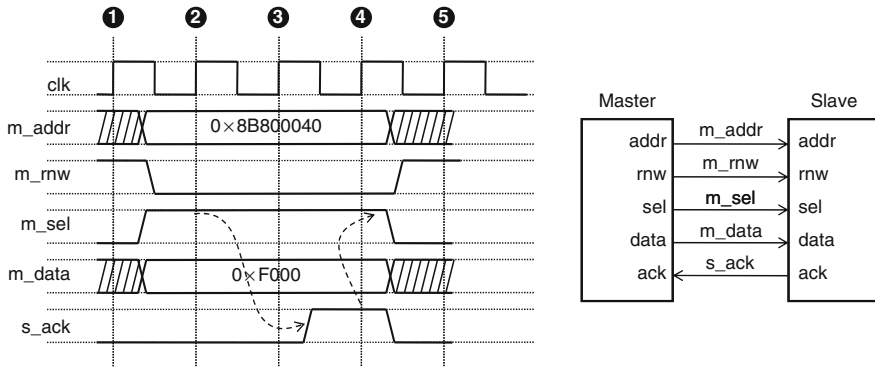
Fig. 8.5 Write transfer with one wait state on a generic peripheral bus

indicates to the bus slave that a bus transaction has started. Further details on the nature of the bus transaction are reflected in the state of the bus address m_addr and the bus read/write control signal m_rnw. In this case, the transfer is a write, so the *read-not-write* (m_rnw) signal goes low.

Bus requests from the master are acknowledged by the slave. A slave can extend the duration of a transfer in case the slave cannot immediately respond to the request of a master. In Fig. 8.5, the slave issues an acknowledge signal s_ack on clock edge 4. This is 1 clock cycle later than the earliest possible clock edge 3. Such a cycle of delay is called a *wait state*: the bus protocol is extended for a few cycles. Wait states enable communication between bus components of very different speed. However, wait states are also a disadvantage. During a wait state, the bus is tied-up and inaccessible to other masters. In a system with many slow slaves, this will significantly affect the overall system performance. A *bus timeout* can be used to avoid that a slave takes over a bus completely. If, after a given amount of clock cycles, no response is obtained from the bus slave, the bus arbiter can declare a timeout condition. This is a signal to the bus master to give up the bus and abort the transfer.

Figure 8.6 shows a read transfer with no wait states. The protocol is almost identical as a write transfer. Only the direction of data is reversed (from slave to master), and the m_rnw control line remains high to indicate a read transfer. The bus protocols for read and write described here are typical for peripheral busses. Table 8.2 makes a comparison between the signal names of the generic bus discussed above, the CoreConnect/OPB bus, the AMBA/APB bus, and the Wishbone bus.

8.3.2 Transfer Sizing and Endianess

By default, all masters and slaves on an on-chip bus will use a uniform wordlength and a uniform endianess. For example, the masters, the slaves, and the bus could be
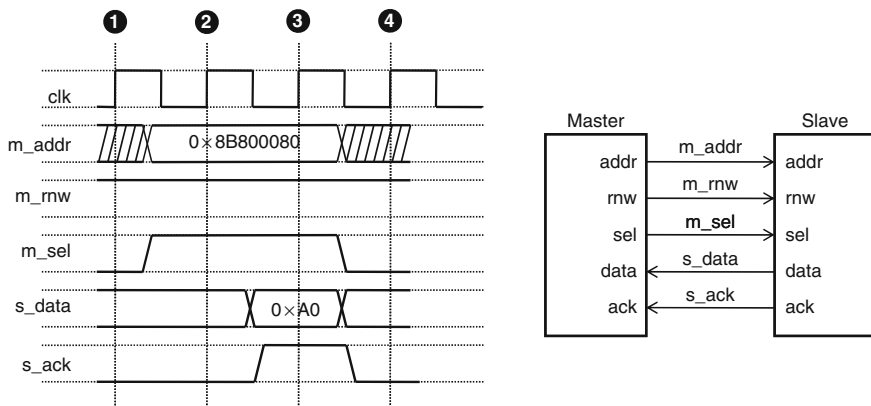
Fig. 8.6 Read transfer with no wait state on a generic peripheral bus

Table 8.2 Bus signals for simple read/write on Coreconnect/OPB, ARM/APB and Wishbone Busses

Generic	CoreConnect/OPB	AMBA/APB	Wishbone
clk	OPB_CLK	PCLK	CLK_I (master/slave)
m_addr	Mn_ABUS	PADDR	ADDR_O (master)
			ADDR_I (slave)
m_rnw	Mn_RNW	PWRITE	WE_O (master)
m_sel	Mn_Select	PSEL	STB_O (master)
m_data	OPB_DBUS	PWDATA	DAT_O (master)
			DAT_I (slave)
s_data	OPB_DBUS	PRDATA	DAT_I (master)
			DAT_O (slave)
s_ack	Sl_XferAck	PREADY	ACK_O (slave)

using 32-bit little-endian words. This would mean that each data transfer transports 32 bits, and that the least significant byte would be found in the lower byte of the 32-bit word. As long as the master, the bus, and the slave make identical assumptions on the data format, a single request and a single acknowledge signal will be adequate to control the transfer of data.

The specific wire numbering within a signal vector can depend on the type of bus. The documentation of the bus should be consulted to determine the name of the least significant bit of a word. As an example, Table 8.3 illustrates the signal naming for a bus slave input under various bus schemes.

While endianess can be configured on most busses, it is a static selection, and dynamic endianess switching is not supported by AMBA, Coreconnect, or Wishbone. The additional hardware and complexity introduced in the bus system do not justify the benefit. Indeed, as illustrated in Fig. 8.7, heterogeneous endianess can be resolved while interconnecting bus components to the bus system.

A bus system will also need to provide a mechanism for *transfer sizing*: selecting what part of a given word belongs to the actual data transfer. In a 32-bit data-bus

Table 8.3 Signal naming and numbering for a bus slave input

Signal part	Offset	CoreConnect/OPB	AMBA/APB	Wishbone
Word		Sl_DBUS[0..31]	PWDATA[31..0]	DAT_I[31..0]
Most significant bit		Sl_DBUS[0]	PWDATA[31]	DAT_I[31]
Little endian byte	0	Sl_DBUS[24..31]	PWDATA[7..0]	DAT_I[7..0]
Big endian byte	0	Sl_DBUS[0..7]	PWDATA[31..24]	DAT_I[31..24]
Little endian byte	3	Sl_DBUS[0..7]	PWDATA[31..24]	DAT_I[31..24]
Big endian byte	3	Sl_DBUS[24..31]	PWDATA[7..0]	DAT_I[7..0]
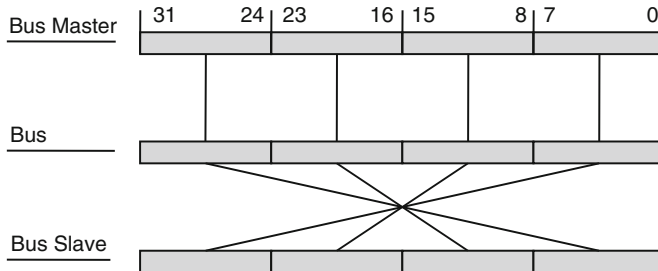
Fig. 8.7 Connecting a big-endian slave to little-endian master

for example, it is useful to be able to transfer a single byte or a halfword (16 bit).
For example, this would allow a C program to write a single char (8 bit) to memory.
Transfer sizing is expressed using byte-enable signals, or else by directly encoding
the size of the transfer as part of the bus control signals. The former method, using
byte-enable signals, is slightly more general than the latter, because it allows one to
cope with unaligned transfers.

To see the difference between the two, consider the difference in performance for
a processor running the following C program.

```
int main() {
  unsigned i;
  char a[32], *p = a;

  for (i=0; i<32; i++)
    *p++ = (char) (i + 4);

  return 0;
}
```

As the processor moves through all iterations of the i-loop, it will generate *byte-aligned* write operations to all addresses occupied by the a array. Assume that this
happens in a system with a 32-bit data bus, and that the third byte of a 32-bit word
needs to be written during a particular iteration. When the bus does not support
unaligned data transfers, the processor will first need to *read* the word that contains
the byte, update the word by modifying a single byte, and write it back to memory.
When the bus does support unaligned data transfers, on the other hand, the processor

can directly write to the third byte in a word. Therefore, the example program will complete quicker on systems that support unaligned transfers. Note that unaligned transfers can also lead to exceptions. For example, processors with a word-level memory organization do not support transfer of unaligned words. If a programmer attempts to perform such a transfer, an exception will result, which usually halts the execution of the program.

Endianess and byte-transfer sizing help bus components to deal with the ordering of individual bytes within a bus word. However, it is also possible that the bus wordlength of the master or the slave is *physically* different from the wordlength provided by the bus. For example, a bus slave could have an 8-bit data bus but needs to be connected to a 32-bit bus. Or, a bus master could have a 64-bit data bus but needs to be connected to a 32-bit bus. These cases will involve the addition of extra hardware to accommodate the interface. Figure 8.8 shows how a 64-bit bus slave and a 16-bit bus slave can be connected to a 32-bit bus. In the case of the 64-bit bus slave, a data-write will transfer only 32 bits at a time; the upper 32 bits are wired to zero. In the case of a data-read, one of the address lines, Addr[2], needs to be used to multiplex the 64 bits of data produced by the bus slave. The net effect of the multiplexing is that the bus slave appears as a continuous memory region when data is read. The case of the 16-bit bus slave is opposite: the 32-bit bus system can deliver more data than the 16-bit bus slave can handle, and an additional address bit, Addr[1] is used to determine which part of the 32-bit bus will be transferred to the 16-bit bus slave.

In summary, busses are able to deal with varying wordlength requirements by the introduction of additional control signals (byte-select signals) and by adding additional multiplexing hardware around the bus slaves or bus masters. A designer also needs to be aware of the endianess assumptions held by the on-chip bus, the bus master, and the bus slave.
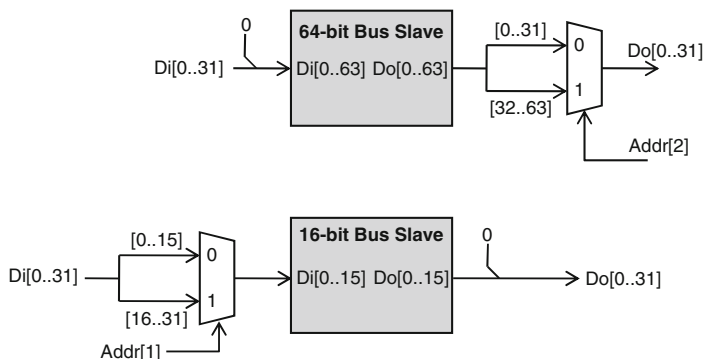
Fig. 8.8 Connecting a 16-bit resp. 64-bit Bus slave to a 32-bit bus

8.3.3   Improved Bus Transfers

As discussed above, each bus data transfer will go through multiple phases. First, the bus master has to negotiate bus access with the bus arbiter. Next, the bus master has to issue a bus address and a bus command. Third, the bus slave has to acknowledge the data transfer. Finally, the bus master has to terminate the bus transfer and release control over the bus. Each of these activities takes a finite amount of time to complete. Moreover, all of these activities are sequential so that the overall system is limited by the speed of the slowest component. For high-speed busses, this is too slow.

On-chip busses use three mechanisms to speed up these transfers. The first mechanism, *transaction-splitting*, separates each bus transaction in multiple phases, and allows each phase to complete separately. This prevents locking up the bus over an extended period of time. The second mechanism, *pipelining*, introduces overlap in the execution of bus transfer phases. The third mechanism, *burstmode operation*, transfers multiple data items, located at closely related addresses, during a single bus transaction.

8.3.3.1   Pipelined Transfers and Transaction Splitting

A bus may use one or several of these mechanisms at the same time. AMBA and CoreConnect treat the transfer of an address as a separate transaction from the transfer of data, and each transaction has its own acknowledge-signal. The rationale is that a bus slave will need some time after the reception of an address in order to prepare for the data transfer. This separate acknowledgement of address and data is also the basis for pipelining and transaction-splitting. Figure 8.9 gives an example
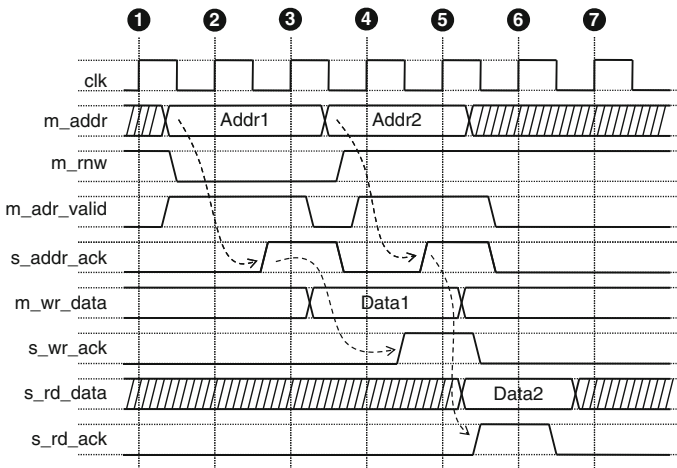

Fig. 8.9 Example of pipelined read/write on a generic bus

of overlapped read/write transfers for a generic bus. Two transfers are shown in the figure: a write followed by a read. The bus used in this figure is slightly different from the one used in Figs. 8.5 and 8.6. The difference is that there are *three* acknowledge signals rather than a single one. On clock edge 2, the bus master indicates a write to address `Addr1`. The bus slave acknowledges this address on clock edge 3. However, at that moment the data transfer is not yet completed. By acknowledging the address, the slave merely indicates it is ready to accept data. From clock edge 4, the bus master executes two activities. First, it sends the write-data `Data1` to the bus slave. Then, it initiates the next transfer by driving a new address `Addr2` on the bus. On clock edge 5, two events take place: the bus slave accepts `Data1`, and it also acknowledges the read-address `Addr2`. Finally, on clock edge 6, the bus slave returns the data resulting from that read operation, `Data2`. Thus, through multiple control/status signals, the bus masters and bus slaves are able to implement bus transfers in an overlapped fashion. Obviously, this will require additional hardware in the bus interface for the master and the slave.

Both the AMBA/AHB and the Coreconnect/PLB support overlapped and pipelined bus transfers, although the detailed implementation of the protocol on each bus system is different. The Wishbone bus does not support splitting or pipelining of bus transfers.

8.3.3.2 Burstmode Transfers

The third technique, burstmode transfer, will transfer multiple data items from closely related addresses in one bus transfer. In computer systems, this works well because of the locality of data and instruction accesses. For example, in case of a cache miss on a processor, and entire cache *line* needs to be replaced. This will require reading or writing of 32 or more consecutive bytes from memory. Burst-mode transfer is also advantageous for *paged* memory architectures, such as DRAM. In a paged memory architecture, a memory access goes through two stages: the first stage selects a single page in memory, while the next stage accesses a single element within the page. Accessing multiple data elements within a single page is easier and quicker than accessing data elements from different pages. The burst-mode transfer of a bus can exploit this when all data elements in a single burst are located on the same memory page.

Burst-mode transfers can have a fixed or a variable length. In a fixed-length burst-mode scheme, the bus master will negotiate the burst properties at the start of the burst transfer, and next perform each transfer within the burst. In a variable-length scheme, the bus master (or the bus slave) has the option of terminating the burst after every transfer. The addresses within a burst are usually incremental, although there are also applications where the address needs to remain constant, or where the address increments with a modulo operation. Thus, as part of the burst specification, a bus may allow the user to specify the nature of the burst address sequence. Finally, the address step will depend on the size of the data within the burst: bytes, half-words, and words will increment addresses by 1, 2, and 4, respectively. Obviously,

Table 8.4 Burst transfer schemes

Burst property	CoreConnect/OPB	AMBA/APB	Wishbone
Burst length	Fixed (2 .. 16)	Fixed (4, 8, 16)	Variable
	or Variable	or Variable	
Address sequence	Incr	Incr/Mod/Const	Incr/Const
Transfer size	Fixed by bus	Byte/Halfword/Word	Byte/Halfword/Word
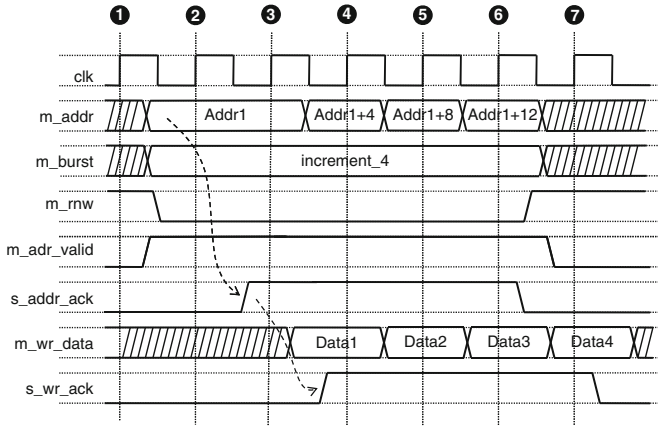
Fig. 8.10 A 4-beat incrementing write burst

all of these options involve adding extra control-signals on the bus, at the side of
the master as well as the slave. Table 8.4 shows the main features for burst-mode
support on Coreconnect, AMBA, and Wishbone.

An example of a burst-mode transfer is shown in Fig. 8.10. This transfer illus-
trates a burst transfer of 4 incrementally addressed words. Besides the commands
discussed before (m_addr, m_rnw, m_adr_valid), a new command m_burst
is used to indicate the type of burst transfer performed by the master. Since this
is a generic example, we will just assume that one of the burst types is encoded as
increment_4, meaning a burst of 4 consecutive transfers with incrementing ad-
dress. On clock edge 3, the slave accepts this transfer, and after that the master will
provide 4 data words in sequence. The address information provided by the master
after the first address is, in principle, redundant. The addresses are implied from
the burst type (increment_4) and the address of the first transfer. Figure 8.10
assumes that the wordlength of the transfers will equal 4 bytes (one word). There-
fore, the address sequence increments by 4. The scheme in this figure is similar
to the scheme used by AMBA/AHB. The Coreconnect/PLB system is slightly more
general (and as a consequence, more complicated), although the ideas of burst trans-
fers are similar to those explained above.

This completes our discussion on bus data transfers. As demonstrated in this sec-
tion, there are many variations and enhancements possible for data transfer over
a bus. Optimal bus performance requires both the master as well as the slave to

be aware of all features provided by a bus protocol. For the hardware–software codesigner, understanding the bus protocols is useful to observe the hardware–software communication at its lowest abstraction level. For example, it is very well possible to associate the behavior of a C program with the data transfers observed on a bus (see Problem 10.3).

So far, we made the implicit assumption that there is only a single master on the bus. In the next section, we will discuss the concepts of bus arbitration, when there are multiple masters on the bus.

8.4 Multimaster Bus Systems

When there is more than a single master on a bus, each bus transfer will need to be negotiated. A *bus arbiter* will control this negotiation process and allocate each transfer to a bus master. Because the specific bus signals are different for AMBA/AHB and Coreconnect/PLB, we will discuss the case of a generic bus and clarify bus-specific implementation features separately.

Figure 8.11 shows the topology of a multimaster bus with two masters and an arbiter circuit. The slaves are not shown in the figure. Of the regular bus features, only the address bus, and a transfer-acknowledge signal are visible. Each master can request access to the bus through the `request` connection. The arbiter uses `grant` to indicate the master that it can access the bus. Once a master has control over the bus, it will proceed through one of the regular bus transfer schemes as discussed before. The `lock` signals are used by a master to grab exclusive control over the bus, and will be clarified later.

Figure 8.12 shows how two masters compete for the bus over several clock cycles. On clock edge 2, master 1 requests the bus through `req1`. Since master 2 is not in need for the bus at that moment, the arbiter will grant the bus to master 1. Note that the `grant` signal comes as an immediate response to the `request` signal.
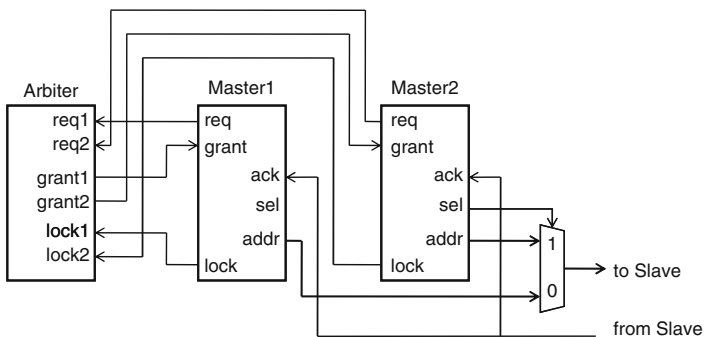

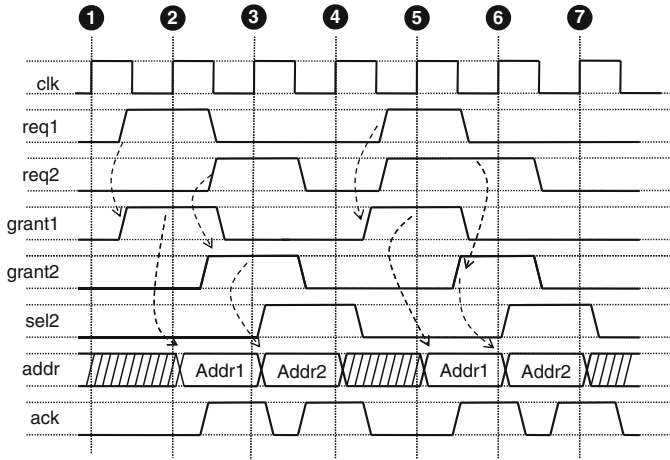**Fig. 8.11** Multi-master arbitration

Fig. 8.12 Multi-master arbitration timing

This means that the bus negotiation process can complete within a single clock cycle. In addition, it implies that the arbiter will need to use combinational logic to generate the `grant` signal based on the `request` signal.

After clock edge 2, master 1 drives an address onto the address bus and completes a regular bus transfer. We assume that the slave acknowledges the completion of this transfer on clock edge 3, by pulling `ack` high. The earliest time when the next arbitration for a bus transfer takes place is clock edge 3. This is called an *overlapping* arbitration cycle, because the arbitration of the next transfer happens at the same moment as the completion of the current transfer. The second transfer is granted to master 2, and completes on clock edge 4.

Between clock edge 4 and 5, the bus sits idle for one cycle, because no master has requested access to the bus. On clock edge 5, both master 1 and 2 request access to the bus. Only one master is allowed to proceed, and this means that there is a *priority resolution* implemented among the masters. In this case, master 1 has fixed priority over master 2, which means that master 1 will always get access to the bus, and master 2 will get access to the bus only when master 1 does not need it. The transfer of master 1 completes at clock edge 6. Since master 2 is still waiting for access to be granted, it can proceed at clock edge 6 because master 1 no longer needs to bus. The fourth and final transfer then completes on clock edge 7.

8.4.1 Bus Priority

The timing diagram in Fig. 8.12 reveals the interesting concept of priority. When multiple masters want to access the bus at the same time, only a single master is

allowed to proceed based on priority resolution. The simplest priority scheme is to allocate a fixed priority, strictly increasing, to every master. While this is easy to implement, it is not necessarily the best solution. When a high-priority master continuously accesses the bus, other low-priority masters can be denied bus transfers for extended amounts of time.

A common case of a multimaster configuration is when multiple processors on a single bus access the same memory. In case the processors work as a symmetrical entity, no processor should have priority over the other. In a situation where all masters have equivalent access rights, priority resolution is implemented using *round-robin* scheme. In that case, each master takes turns to get access to the bus. When two masters request the bus continuously, then the bus transfers of master 1 and master 2 will be interleaved. Another scheme is *least-recently-used*, in which the master that was waiting for the bus for the longest time will get access first. Equal-priority schemes such as round-robin or least-recently-used avoid *starvation* of the bus masters, but, they also make the performance of the bus somewhat unpredictable. When working with latency-critical applications, this can be a problem. To address this, designers can use a mixed scheme that combines multiple levels of priority with an equal-priority scheme to allow several masters to share the same priority level.

The priority algorithm used by the bus arbiter is not part of the definition of the bus transfer protocol. Therefore, the Coreconnect/PLB and AMBA/AHB bus schemes only describe the arbitration connections, but not the priority schemes. The Wishbone bus is special in that it does not define special bus request/grant signals. Instead, Wishbone leaves the design of the bus topology to the designer.

Table 8.5 makes a comparison between the generic bus arbitration signals defined above, and those of CoreConnect and AMBA. The table also lists a few arbitration signals that are unique to each individual bus protocol. The bus locking signals will be explained shortly. The other signals have the following meaning.

- `Mx_priority[..]` allows a PLB master to select its priority for each transfer. This scheme allows the master to change its priority level dynamically depending on the needs of the bus transfer.
- `HMASTER[..]` is an encoding of the identity of the master that was granted bus access by the arbiter. The signal is used to drive the bus address multiplexer.

Table 8.5 Arbitration signals on CoreConnect/OPB and AMBA/AHB

Signal	CoreConnect/PLB	AMBA/AHB
reqx	Mx_request	HBUSREQ
grantx	PLB_PAValid	HGRANT
lock	Mx_Buslock	HLOCK
	PLB_Buslock	HMASTLOCK
	Mx_priority[..]	
sel		HMASTER[..]

8.4.2 Bus Locking

The final concept in multimaster bus schemes is *bus locking*: the exclusive allocation of a bus to a single master for the duration of multiple transfers. There are several reasons why bus locking may be needed. First, when large blocks of data need to be transferred with strict latency requirements, exclusive access to the bus may be required. While burst-mode transfers can help a master to complete these transfers quickly, these transfers can still be interrupted by another master with higher priority. By locking the bus, the master can be sure this will not happen.

The second need for bus locking is when a master needs to have guaranteed, exclusive access to consecutive transfers, typically a read transfer followed by a write transfer. This mechanism can be used to implement a *test-and-set* instruction, a well-known primitive used to implement mutual exclusion in software. When two bus masters have access to a single, shared region of memory, access to that shared region needs to be exclusive.

An example implementation of test-and-set is shown below. This C program runs on each of two processors (bus masters) attached to the same bus. They share a memory location at address `0x8000`. By calling `testandset`, a processor will try to read this memory location, and write into it in the span of a single locked bus transfer. This means that the function `test_and_set()` cannot be interrupted: only one processor will be able to read the value of the `mutex` when its value is low. The two processors use this function as follows. Before accessing the shared resource, the processors will call `enter()`, while they will call `leave()`. The shared resource can be anything that needs exclusive access by one of the processors.

```
int *mutex = (int *) 0x8000; // location of mutex

int test_and_set() {
  int a;
  lock_bus();
  a = *mutex;
  *mutex = 1;
  unlock_bus();
}

void leave() {
  *mutex = 0;
}

void enter() {
  while (test_and_set()) ;
}
```

Figure 8.13 shows an example of `test-and-set` with bus-locking. On clock edge 2, master 2 requests access to the bus using `req2`. This access is granted by the arbiter through `grant2`. After clock edge 2, this master grabs the bus using `sel2` and locks it using `lock2`. Master 2 will now perform a test-and-set operation, which involves a read of a memory address immediately followed by a write to the
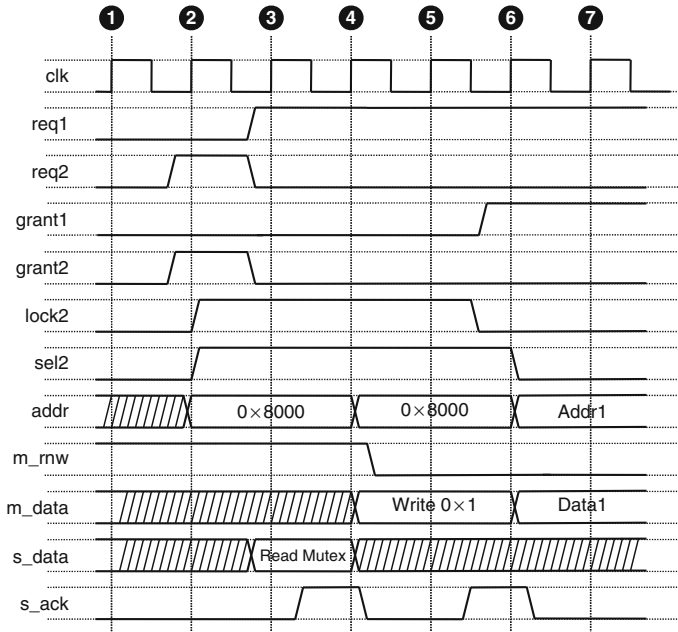
Fig. 8.13 Test-and-set operation with bus locking

same memory address. The read operation starts on clock edge 3 and completes on clock edge 4. On clock edge 3, the master drives an address onto the bus and signals a read operation (m_rnw). On clock edge 4, the slave delivers the data stored at this address and completes the transfer using s_ack.

Meanwhile, another master requested bus access starting on clock edge 3 (using req1). However, because master 2 has locked the bus, the arbiter will ignore this request. Master 2 will be granted further use of the bus until it release the lock. This access is guaranteed even if master 2 has a lower priority than other masters requesting the bus.

After performing the reading part of the test-and-set instruction, master 2 will now write a "1" into the same location. At clock edge 5, master 2 puts the address and the data on the bus, and at clock edge 6 the slave accepts the data. The lock can be released after clock edge 5. Note that, should the write operation to the slave fail, then the complete test-and-set instruction has failed. We assume however that the write operation completes correctly. As soon as master 2 releases lock2, control will go to master 1 because of the pending request on req1. As a result, starting with clock edge 6, a new bus transfer can start, which is allocated to master 1.

In conclusion, when multiple masters are attached to a single bus, individual bus transfers need to be arbitrated. In addition, a priority scheme may be used among masters to ensure latency requirements for particular masters. Finally, bus-locking

can be used to implement guaranteed access for an extended amount of time. Since all of these techniques are implemented in hardware, at the level of a bus transfer, they are very fast and efficient. As such, bus systems play an important role in building efficient hardware–software communication.

8.5   On-Chip Networks

A key issue with on-chip bus systems is that they are a global resource, shared among all modules in a chip: processors, memories, coprocessors, and peripherals. As a result, the communication among these modules is sequentialized. While it is possible to split a bus in smaller segments using bus segments, this remains only a partial solution. Bus bridges assume an implicit hierarchy among bus segments: a bus bridge is a master on one side and a slave on the other. There are many cases, such as with symmetric multiprocessor architectures, where a hierarchy among the processors is not wanted or even counter-productive.

Besides the logical constraints originating from a bus, there are also significant technological issues. Implementing very long wires on a chip is hard, and distributing high-frequency signals and clocks using such wires is even harder. The power consumption of a wire will be proportional to the length of the wire and the switching frequency of the signals on that wire. Hence, global wires will consume significantly more power than small, local wires. Chips with a centralized interconnection system will consume more energy for the same task than chips with a distributed interconnection system.

In this section, we review some of the ongoing developments in on-chip interconnection systems. At the start, it is instructive to consider the approach taken by the Wishbone bus. In contrast to Coreconnect and AMBA, the Wishbone bus does not assume a predefined topology. This has some disadvantages, such as for example the absence of an arbitration scheme at the bus protocol level (see Sect. 8.4). On the other hand, Wishbone allows masters and slaves to be connected using arbitrary topologies. Depending on the interconnection scheme, additional bus-interconnect components need to be designed. Figure 8.14 illustrates the Wishbone master/slave interface definition and three possible topologies that are made using this interface. The easiest topology, and the only one that does not need additional hardware, is a data-flow type of interconnection that alternates master- and slave-interfaces. The standard bus topology requires selection of one of several slave interfaces by means of a decoder. A bus topology can also be extended with multimaster capabilities by adding arbitration hardware. A *cross-bar* interconnect system enables multiple masters to communicate, concurrently, with multiple slaves.

Cross-bar hardware supports multiple concurrent communications, and this may require multiple arbiters, multiplexers, and decoders. Recent generations of the AMBA bus support the cross-bar concept partially through so-called *multilayer* busses. The idea of such a bus is to implement a cross-bar or interconnect matrix for a limited number of master and slave interfaces. Each of these interfaces can be
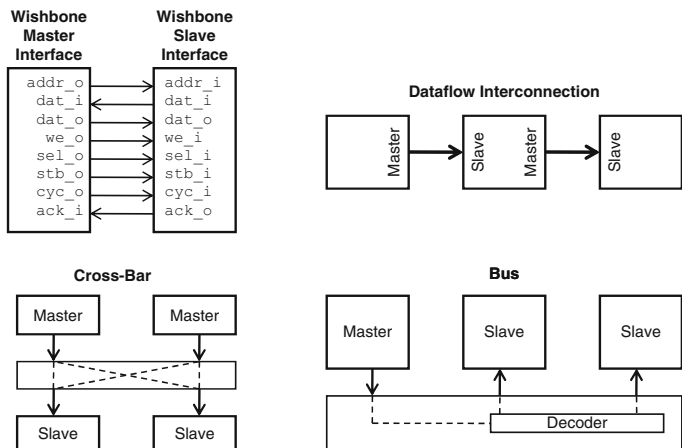
**Wishbone
Master
Interface**

**Wishbone
Slave
Interface**

```
addr_o  ────→  addr_i
dat_i   ←────  dat_i
dat_o   ────→  dat_o
we_o    ────→  we_i
sel_o   ────→  sel_i
stb_o   ────→  stb_i
cyc_o   ────→  cyc_i
ack_i   ←────  ack_o
```

Dataflow Interconnection

| Master | Slave | Master | Slave |

Cross-Bar

| Master | Master |

| Slave | Slave |

Bus

| Master | Slave | Slave |

Decoder

Fig. 8.14 Bus topologies with Wishbone

| AMBA Master | AMBA Slave | | Arbiter | AMBA Slave |

Layer 1 → Decoder

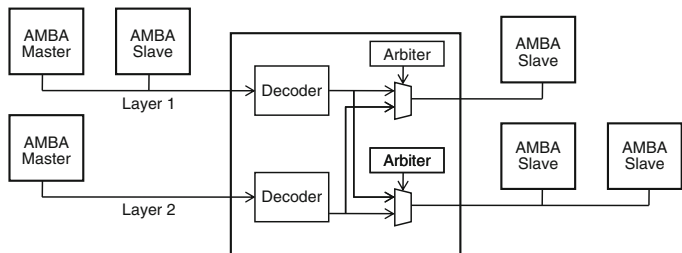| AMBA Master | | Arbiter | AMBA Slave | AMBA Slave |

Layer 2 → Decoder

Fig. 8.15 A multi-layer bus with AMBA

attached to a standard bus system. This way, the interconnect matrix is equivalent to a multiport bridge. Figure 8.15 illustrates a two-layer bus that connects two slave interfaces to two master interfaces.

Cross-bar interconnect and multilevel busses have very limited scalability because they are a centralized resource. The complexity of a cross-bar increases with the square of the number of connected components. In a modern SoC, distributed computing is quickly becoming the norm. Think for example of multiprocessor System-on-Chip architectures. The shift of centralized computing to distributed computing results in a corresponding shift in the communication paradigm as well. The present understanding of this new communication paradigm is the "network on chip", conceptually presented in Fig. 8.16. The computational elements of the chip and their interconnections are organized in a geometrical pattern, typically a matrix. The elements in this matrix are the *tiles* of the network on chip. Every tile can directly communicate with its neighboring tiles. In addition, every tile has an address, symbolically indicated by the matrix indices. This allows any tile to communicate with any other tile. A *route* for the communication is selected, and a data

Fig. 8.16 A generic
network-on-chip

Processor Tile   Communication Link

Route

Node
Address
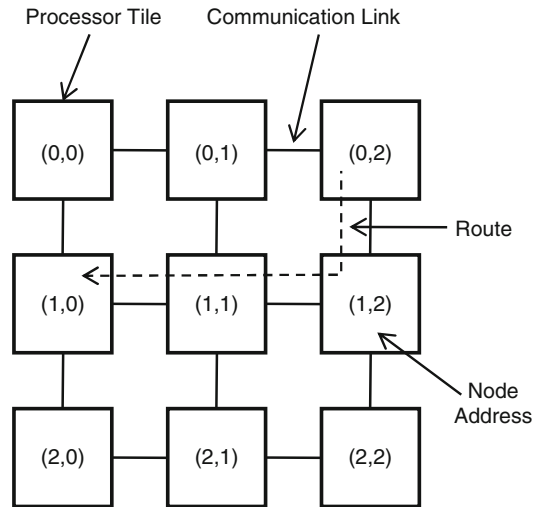
packet travels through a number of *hops*, from a source tile to a destination tile
over a number of intermediate tiles. Figure 8.16 illustrates a route from tile (0,2)
to tile (1,0).

The design of a network on-chip, and its operation, introduces a challenging
set of problems. At the basic level, the communication and data representation is
very different from the approach used in on-chip busses. In a network-on-chip, data
items are encapsulated in a *packet* before being transmitted. Once a packet leaves a
source tile and travels to a destination tile, it needs to find a route. As can be seen
in Fig. 8.16, a route is not unique. Between a given source tile and destination tile,
many different routes are possible. Hence, one needs a distributed routing algorithm,
which will try to select segments such that the overall network on-chip has the low-
est amount of congestion. There are also important system-level questions. What
is the best network-on-chip topology? Can we optimize minimum-latency (fewest
hops) with maximum-throughput (least congestion)? Can we accommodate differ-
ent communication behaviors, such as short high-throughput bursts with regular
low-throughput traffic?

The CELL processor is a well-known multiprocessor device that relies on
network-on-chip technology to provide on-chip communications. The CELL com-
bines 8 regular processing components called *SPE* (synergistic processing element).
In addition, there is a control processor called *PPE* (power processor element) and
an off-chip interface unit. All of these components are connected to the same
network on chip, called the *EIB* (element interconnect bus). As illustrated in
Fig. 8.17, the EIB consists of 4 ring structures, each 16 bytes wide. The com-
munication model of the CELL processors assumes that processors will work using
local memory, and that communication is implemented by moving blocks of data
from one local memory location to the other. A *Direct Memory Access* (DMA) unit
is used to handle communication between the bus interface and the local memory.
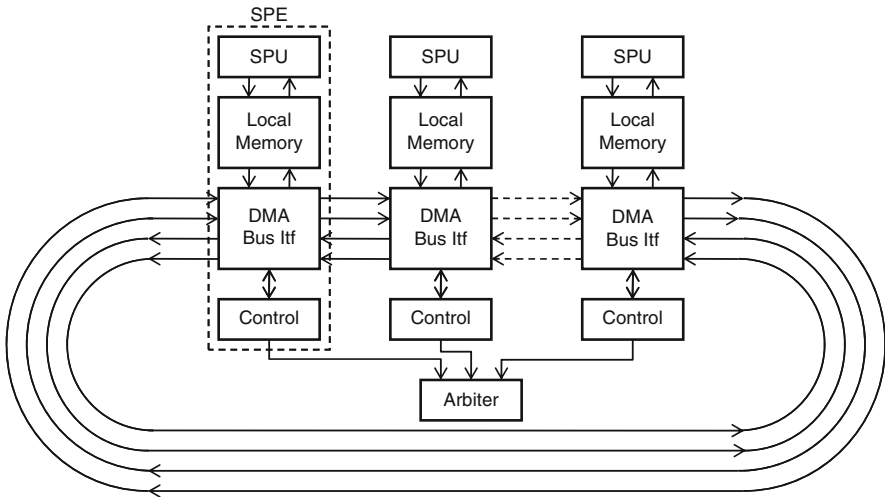
Fig. 8.17 On-chip network in the CELL processor

The 4 rings run in opposite directions, so that each SPE can directly talk to its neighbors. Communication with other SPE is possible by taking several hops over the communication bus. Each time an SPE wants to transmit a block of data over EIB, it will send an appropriate request to the central on-chip arbiter. The arbiter will schedule all outstanding requests over the 4 rings. Up to three transfers can be concurrently scheduled over each ring, provided that these transfers use different segments on the ring.

The resulting data bandwidth on the CELL processor is impressive. In a 3.2 GHz CELL chip, the interface from the SPE to the rest of the chip supports 25.6 GBytes/s data bandwidth in each direction. The downside is that the CELL must be programmed in a very particular manner, as a set of concurrent programs that pass messages to one another. Optimizing the performance of a parallel CELL program is complex, and requires attention to a large collection of details, such as the granularity of tasks and message blocks, the synchronization mechanisms between processors, and the locality of data.

8.6 Summary

In this chapter, we discussed the concepts of on-chip interconnection busses, the lowest abstraction level where software and hardware meet. On-chip busses are shared communication resources to connect a bus master components with bus slave components. We discussed several examples, including the AMBA, CoreConnect, and Wishbone bus. These busses support basic read/write transfers between bus masters and slaves. They also include enhancements to improve the performance

of on-chip communication. These enhancements include pipelining, split-transfers, and burstmode transfers. When multiple masters are present on a bus, each transfer needs to be arbitrated. These transfers can be prioritized based on the bus master. Recent developments in on-chip communication with System-on-Chip emphasize distributed solutions in the form of multilevel busses or network-on-chip. The net effect of this evolution is that on-chip communication becomes a design challenge on its own, with many different abstraction layers to tackle. This makes the design and optimization of on-chip communication schemes, at different abstraction levels, also an interesting problem from the hardware–software codesign perspective.

8.7 Further Reading

The best reference to study on-chip bus systems is obviously the documentation from the vendors themselves. The AMBA bus specification can be obtained online from ARM (ARM 2009a). Likewise, the CoreConnect bus specification can be obtained online from IBM (IBM 2009). An in-depth discussion of contemporary on-chip bus systems, including AMBA and CoreConnect, is available from Pasricha and Dutt (2008). The same book also reviews ongoing research topics for on-chip bus systems.

Recently research efforts have focused on network-on-chip. An overview of the design principles may be found in De Micheli's book (Micheli and Benini 2006). A recent special issue of IEEE Design and Test Magazine has reviewed several proposals and open research issues (Ivanov and De Micheli 2005).

8.8 Problems

8.1. Find the maximum communication speed from CPU1 to CPU2 in the system architecture shown in Fig. 8.18. Assume that the CPUs have a dedicated synchronization channel available so that they will be able to choose the most optimal moment to perform a read- or a write-transaction. Use the following design constants.

- Each bus transaction on the high-speed bus takes 50 ns.
- Each bus transaction on the low-speed bus takes 200 ns.
- Each memory access (read or write) takes 80 ns.
- Each bridge transfer takes 100 ns.
- The CPU's are much faster than the bus system, and can read/write data on the bus at any chosen data rate.

8.2. The timing diagram in Fig. 8.19 illustrates a write operation on the AMBA peripheral bus, AMBA APB. A *memory-mapped* register is a register which is able to intercept bus transfers from a specific address. In this case, we wish to create
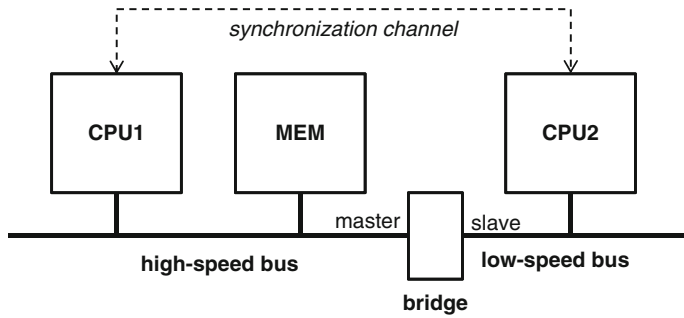
Fig. 8.18 System topology for Problem 10.1
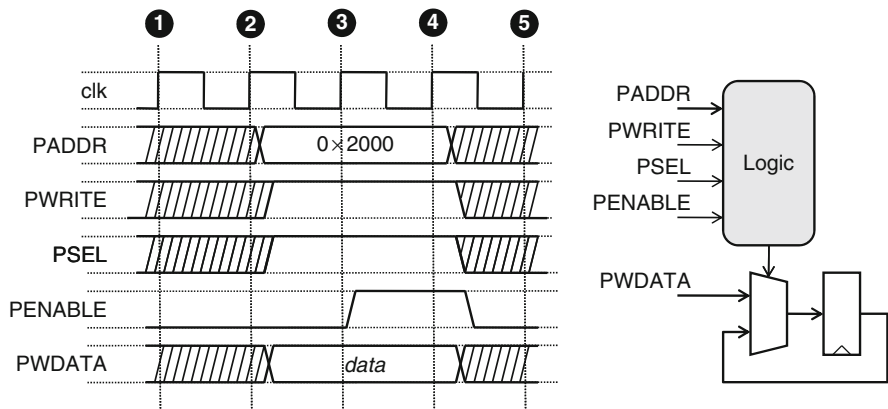
Fig. 8.19 Timing diagram and schematic for Problem 10.2

logic which will write PWDATA into a register when a write to address 0x2000 occurs. Develop a logic expression for the logic module shown in Fig. 8.19. Assume a 16-bit address.

8.3. The system-on-chip in Fig. 8.20 combines a coprocessor, a processor, and on-chip data- and instruction-memory. The processor will copy each element of an array a[] to the coprocessor, each time storing the result as an element of an array r[]. The C program that achieves this is shown on the right of Fig. 8.20. All the operations in this architecture take *zero* time to execute, apart from the following two: accessing the on-chip data memory takes 3 cycles and processing a data item on the coprocessor takes 5 cycles.

(a) Find the resulting execution time of the C program.
(b) Show how you can rewrite the C program so that the resulting execution time becomes smaller than 8,000 cycles.

Fig. 8.20 Architecture and C program for Problem 8.3

Listing 8.1 Program for Problem 8.4
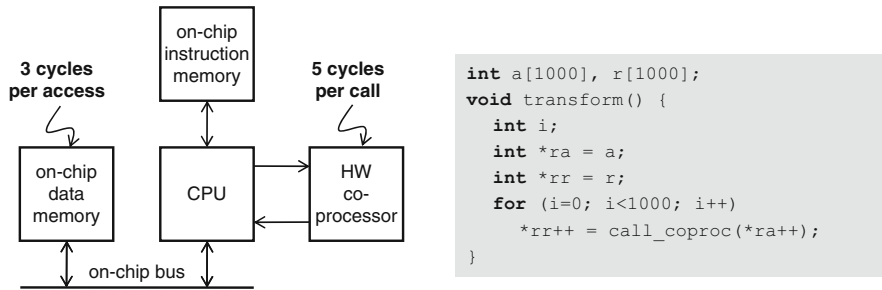```
#include <stdio.h>
void main() {
  int i, a[0x40];
  for (i=0; i< 0x40; i++)
    if (i > 0x23)
      a[i] = a[i-1] + 1;
    else
      a[i] = 0x5;
}
```

**8.4.** While debugging a C program on a 32-bit microprocessor (shown in
Listing 8.1), you capture the following bus transfer. The microprocessor is at-
tached to an off-chip memory that holds the program and the data. The text and data
segment both are stored in an off-chip memory starting at address `0x44000000`.
The array a[] starts at address `0x44001084`. The instructions from the body
of the loop start at address `0x44000170`. Observe closely the timing diagram in
Fig. 8.21 and answer the questions below.

- The cursor X in Fig. 8.21 is positioned at a point for which the address bus con-
  tains 0x4400111C and the data bus contains 0x8. Is this a memory read of a
  memory write?
- For the same cursor position "X", is this memory access for an instruction-fetch
  or for a data-memory read?
- For the same cursor position "X", what is the value of the loop counter i from the
  C program?

**8.5.** The timing diagram in Fig. 8.22 shows the arbitration process of two masters,
M1 and M2, requesting access to a shared bus. Answer the questions below using
the information provided in the timing diagram.

(a) Based on the timing diagram, which master has the highest priority for bus
    transfers: M1, M2, or impossible to tell?
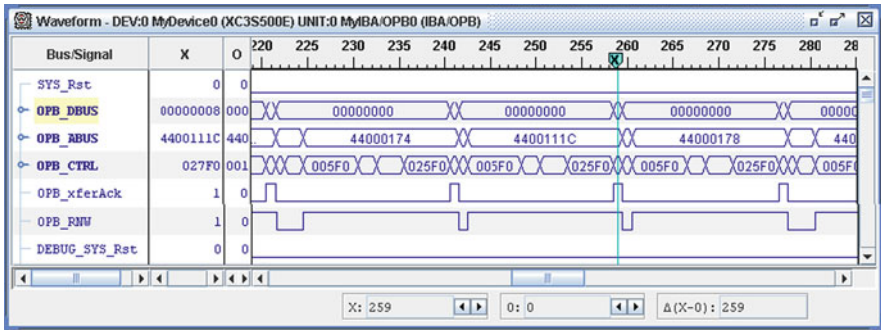
**Fig. 8.21** Timing diagram for Problem 8.4



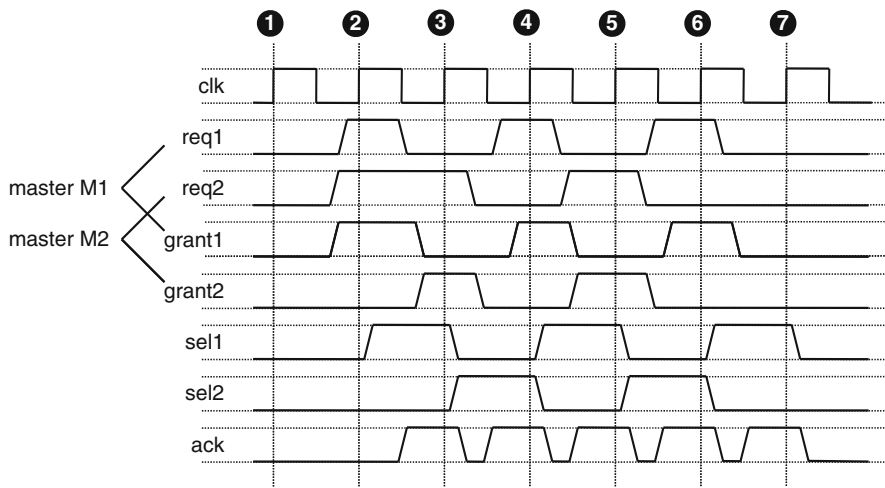**Fig. 8.22** Timing diagram for Problem 8.5

(b) Which master has control over the address bus between clock edge 3 and clock edge 4: M1, M2, or impossible to tell?

(c) What type of component determines the value of the grant*x* signals: a bus master, an bus arbiter, or a bus slave?

(d) What type of component determines the value of the ack signal: a bus master, an bus arbiter, or a bus slave?

# Chapter 9
# Hardware/Software Interfaces

**Abstract** The objective of a hardware/software interface is to enable communication between software and custom hardware. The software runs on a microprocessor, while the custom hardware is attached to that microprocessor. We will consider how to implement stable data transfers by synchronizing software and hardware. Next, we will discuss the various implementations of hardware/software interfaces, including memory-mapped interfaces, coprocessor interfaces, and custom-instruction interfaces.

## 9.1 The Hardware/Software Interface

Figure 9.1 illustrates the elements that make up a "hardware/software" interface.

- The *microprocessor interface* includes the hardware and low-level firmware that allows a software program to get "out" of the microprocessor. A microprocessor can use several different mechanisms for this, such as coprocessor instructions, or memory load/store instructions.
- The *on-chip bus* transports data from the microprocessor module to the custom-hardware module. While typical on-chip buses are shared among several masters and slaves, they can also be implemented as dedicated point-to-point connections. For example, coprocessors are often attached to a dedicated link.
- The *custom-hardware interface* handles data coming from, and going to, the on-chip bus. The custom-hardware interface will decode the on-chip bus protocol, and make the data available to the custom-hardware module through a register or a dedicated memory.

In this chapter, we will discuss three different implementations of the hardware/software interface. In a *memory-mapped* interface, the microprocessor interface is the memory interface of a microprocessor, and the communication with the custom-hardware module is implemented through load/store instructions. A *coprocessor* interface uses a dedicated coprocessor port on a microprocessor. The custom-hardware module is controlled with specialized coprocessor instructions

**Fig. 9.1** The hardware/
software interface
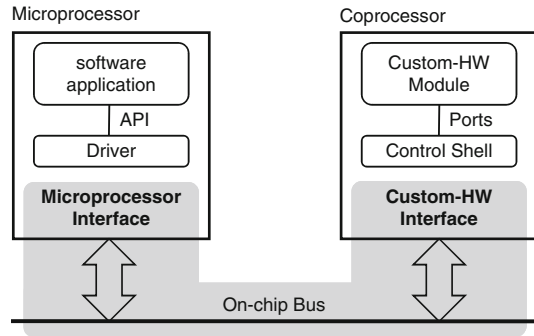


on the microprocessor. Finally, a *custom-instruction interface* integrates a custom-hardware module inside of a microprocessor and defines a new instruction on the microprocessor to control this custom-hardware module.

A hardware/software codesigner will choose among these three options based on the capabilities of a given microprocessor, the required performance of the hardware/software interface in terms of throughput and latency, and the design cost of attaching (porting) software and hardware to this interface.

## 9.2 Synchronization Schemes

Before discussing the three types of interfaces, we turn to a fundamental question: how can we guarantee that the software application and the custom-hardware module will remain synchronized, given that they are independently executing entities? How does a hardware module know that a software program wishes to communicate with it?

### 9.2.1 Synchronization Concepts

We define *synchronization* as the structured interaction of two otherwise independent and parallel entities. Figure 9.2 illustrates the key idea of synchronization. Two entities, in this case a microprocessor and a coprocessor, each have an independent thread of execution. Through synchronization, one point in the execution thread of the microprocessor is tied to one point in the control flow of the coprocessor. This is the synchronization point. Synchronization must guarantee that when the microprocessor is at point A then the coprocessor will be at point B. There are several mechanisms to implement this objective, and we will discuss these later.

Synchronization is needed to implement communication in parallel systems. Obviously, if the parallel components never interact, there's no point trying to keep them synchronized. We discussed communication within parallel systems before:
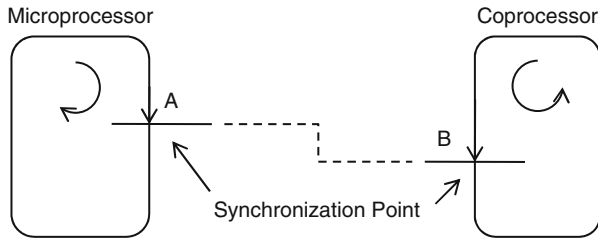
**Fig. 9.2** Synchronization point

recall our discussion on the implementation of data-flow (Chap. 2). In data-flow, different actors communicate with one another through the exchange of tokens. Assume that one actor is implemented in software and another one is implemented as a custom-hardware module. Also, assume that the software actor sends tokens to the hardware actor. According to the rules of data-flow, each token produced must eventually be consumed, and this implies that the hardware actor must somehow know when the software actor will be sending that token. In other words: the hardware and software actors will need to synchronize when communicating a token. Of course, there are many different approaches to realize a data-flow communication, depending on how we realize the data-flow edge. But, regardless of the realization, the requirement to synchronize does not go away. For example, one may argue that a FIFO memory could be used to buffer the tokens going from software to hardware, thereby allowing hardware and software actors to run "independently." This is not the case: FIFO memories do not remove the requirement to synchronize. When the FIFO is empty, the hardware actor will need to wait until a token appears, and when the FIFO is full, the software actor will need to wait until a free space appears.

Synchronization is an interesting problem because it has several dimensions, each with several levels of abstraction. Figure 9.3 shows the three dimensions of interest: time, data, and control. In this section, we explain the meaning of these dimensions. In further sections, we discuss several examples of synchronization mechanisms.

The *dimension of time* expresses the granularity at which two parallel entities synchronize. Clock-cycle accuracy is needed when we interface two hardware components with each other. Bus-transfer accuracy is needed when the granularity of synchronization is expressed in terms of a specific bus protocol, such as a data transfer from a master to a slave. Finally, transaction accuracy is needed when the granularity of synchronization is a logical transaction from one entity to the next. Note that the meaning of *time* varies with the abstraction level, and does not have to correspond to wall-clock time. Instead, synchronization only needs to control the execution order of operations: clock cycles, bus transfers, and logical transfers.

In practice, a synchronization scheme between hardware and software covers all abstraction levels in time. This is because high-level transactions are typically implemented in terms of bus-transfers, and bus-transfers imply the use of
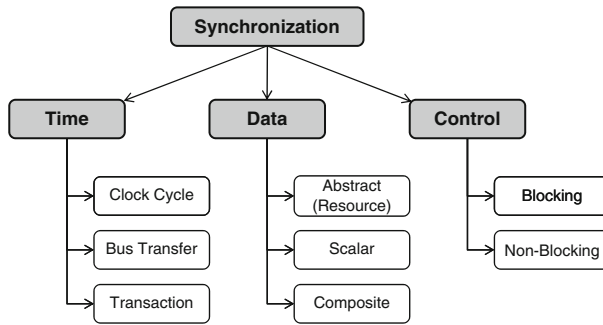
**Fig. 9.3** Dimensions of the synchronization problem

synchronization at the cycle-accurate level. However, for a hardware–software code-signer, it is useful and often easier to think about synchronization problems at higher abstraction levels, before pinning down the implementation details.

The *data dimension of synchronization* determines the size of the container involved in the synchronization. When no data is involved at all, the synchronization between two entities is abstract. Abstract synchronization is useful to handle access to a shared resource. On the other hand, when two parallel entities communicate data values, they will need a shared data container. The communication scheme works as follows: one entity dumps information in the data container, and synchronizes with the second entity. Next, the second entity retrieves information of the data container. In this scheme, synchronization is used to indicate when there is something of interest in the shared data container. Under scalar data synchronization, the two entities will synchronize for every single data item transferred. Under composite data synchronization, the two entities will synchronize only once per several data transfers.

The *control dimension of synchronization* indicates how the local behavior in each entity will implement synchronization. In a blocking scheme, the synchronization can stall local behavior. In a nonblocking scheme, the local behavior will not be stalled, but instead a status signal is issued that the synchronization primitive did not succeed.

A hardware–software co-designer is able to make decisions along each of these three dimensions separately. In the following sections, several examples of synchronization will be described.

### 9.2.2 Semaphore

A *semaphore* is a synchronization primitive which does not involve the transfer of data, but instead controls access over an abstract, shared resource. A semaphore $S$ is a shared resource that supports two operations: grabbing the semaphore ($P(S)$) and releasing the semaphore ($V(S)$). These operations can be executed by several

concurrent entities. In this case, we will assume there are two entities competing for the semaphore. The $P$ and $V$ are the first letters of the Dutch verbs "proberen" and "verhogen," chosen by the scientist who proposed using semaphores in system software, Edgser Dijkstra.

The meaning of $P(S)$ and $V(S)$ is as follows. $P(S)$ and $V(S)$ are indivisible operations that manipulate the value of a semaphore. Initially, the value of the semaphore is 1. The operation $P(S)$ will decrement the semaphore by one. If an entity tries to $P(S)$ the semaphore while it is zero, then $P(S)$ will stall further execution of that entity until the semaphore is nonzero. Meanwhile, another entity can increment the semaphore by calling $V(S)$. When the value of the semaphore is nonzero, any entity which was stalled on a $P(S)$ operation will decrement the semaphore and proceed. In case multiplie entities are blocked on a semaphore, one of them, chosen at random, will be able to proceed. The maximum value of the basic binary semaphore is 1. Calling $V(S)$ several times will not increase the semaphore above 1, but it will not stall either. This is the basic semaphore operation, and several enhancements have been defined in the context of operating system software. For our discussion on synchronization however, this basic definition is sufficient.

Using semaphore operations, it is possible to describe the synchronization of two concurrent entities. The pseudocode in Listing 9.1 is an example using a single semaphore. A first of two concurrent entities needs to send data to the second entity through a shared variable shared_data. When the first entity starts, it immediately decrements the semaphore. Entity two, on the other hand, waits for a short while, and then will stall on the semaphore. Meanwhile, entity one will write into the shared variable, and increment the semaphore. This will unlock the second entity,

**Listing 9.1** One-way synchronization with a semaphore

```
int shared_data;
semaphore S1;

entity one {
  P(S1);
  while (1) {
    short_delay();
    shared_data = ...;
    V(S1);   // synchronization point
  }
}

entity two {
  short_delay();
  while (1) {
    P(S1);   // synchronization point
    received_data = shared_data;
  }
}
```
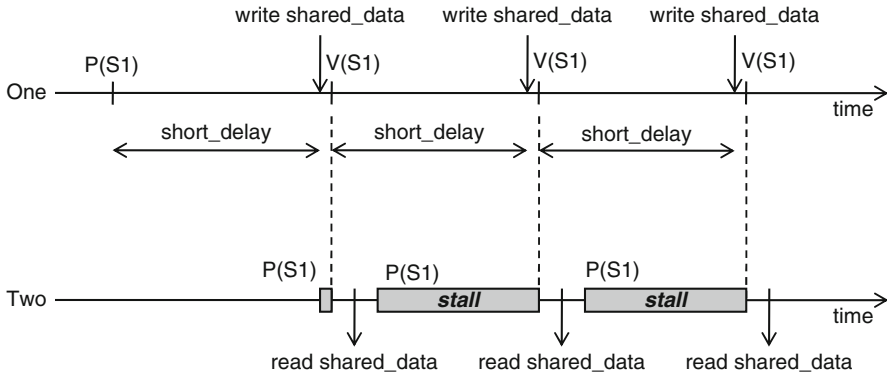
**Fig. 9.4** Synchronization with a single semaphore

which can now read the shared variable. The moment when entity one calls $V(S1)$ and entity two is stalled on $P(S1)$ is of particular interest: it is the synchronization point between entity one and entity two.

Figure 9.4 illustrates the interaction between entities one and two. The dashed lines indicate the synchronization points. Because entity two keeps on decrementing the semaphore faster than entity one can increment it, entity two will always stall. As a result, each write of shared_data by entity one is followed by a matching read in entity two.

Yet, this synchronization scheme is not perfect, because it assumes that entity two will always arrive first at the synchronization point. Now assume that the slowest entity would be entity two instead of entity one. Under this assumption, it is possible that entity one will write shared_data several times before entity two can read a single item. Indeed, $V(S1)$ will not stall even if it is called several times in sequence. It is easy to make entity one faster, even for the simple example shown earlier: just move the *short_delay* () function call from the while-loop in entity one to the while-loop in entity two.

This observation leads to the conclusion that the general synchronization of two concurrent entities needs to work in two directions: one entity needs to be able to wait on the other, and vice versa. In the *producer/consumer* scenario explained earlier, the producer will need to wait for the consumer if that consumer is slow. Conversely, the consumer will need to wait for the producer if the producer is slow. We can address the situation of unknown delays with a two-semaphore scheme, as shown in Listing 9.2. At the start, each entity decrements a semaphore. $S1$ is used to synchronize entity two, while $S2$ is used to synchronize entity one. Each entity will release its semaphore only after the read-operation (or write-operation) is complete.

Figure 9.5 illustrates the case where two semaphores are used. On the first synchronization, entity one is quicker than entity two, and the synchronization is done using semaphore $S2$. On the second synchronization, entity two is faster, and in this case the synchronization is done using sempahore $S1$.

**Listing 9.2** Two-way synchronization with two semaphores

```
int shared_data;
semaphore S1, S2;

entity one {
  P(S1);
  while (1) {
    var_delay();
    shared_data = ...;
    V(S1);  // synchronization point 1
    P(S2);  // synchronization point 2
  }
}

entity two {
  P(S2);
  while (1) {
    var_delay();
    P(S1);  // synchronization point 1
    received_data = shared_data;
    V(S2);  // synchronization point 2
  }
}
```
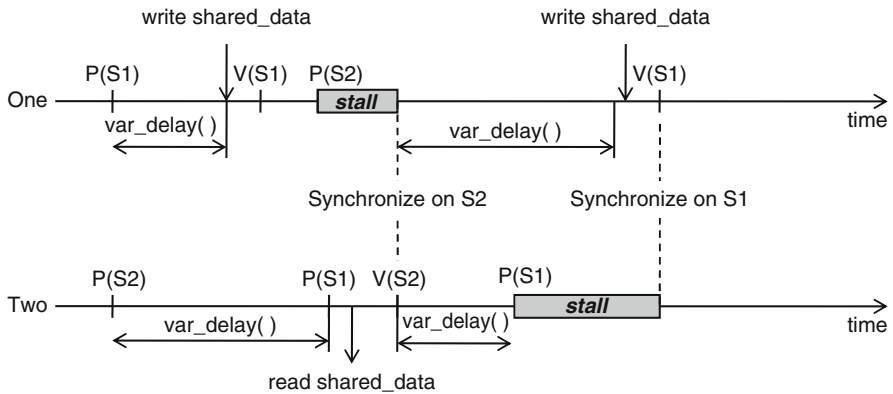

Fig. 9.5 Synchronization with two semaphores

9.2.3 One-Way and Two-Way Handshake

In parallel systems, concurrent entities may be physically distinct, and a centralized semaphore may not be feasible. For this situation, we will use a *handshake*: a signaling protocol based on signal levels. The concepts of semaphore-based synchronization will still apply. We will implement a synchronization point by making one entity wait for another one.

Entity One   Entity Two
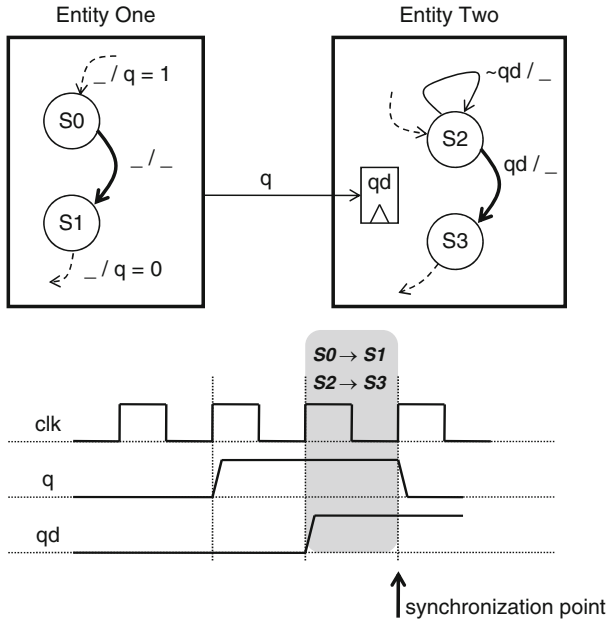
Fig. 9.6 One-way handshake

The most simple implementation of a handshake is a *one-way handshake*, which only needs one wire. Figure 9.6 clarifies the implementation of this handshake for the case of two hardware modules. When we will discuss hardware/software interfaces, we will also consider handshakes between hardware and software. In this figure, entity one transmits a query signal to entity two. Entity two captures this signal in a register, and uses its value as a state transition condition. The synchronization point is the transition of S0 to S1 in entity one, with the transition of S2 to S3 in entity two. Entity two will wait for entity one until both of them can make these transitions in the same clock cycle. Entity one needs to set of acknowledge signal to high one cycle *before* the actual synchronization point, because the request input in entity two is captured in a register.

The limitation of a one-way handshake is similar to the limitation of a one-semaphore synchronization scheme: it only enables a single entity to stall. To accomodate arbitrary execution delays, we need a two-way handshake as shown in Fig. 9.7. In this case, two symmetrical handshake activities are implemented. Each time, the query signal is asserted during the transition preceding the synchronization point. Then, the entities wait until they receive a matching response. In the timing diagram of Fig. 9.7, entity one arrives first in state S0 and waits. Two clock cycles later, entity two arrives in state S2. The following clock cycle is the synchronization point: as entity one proceeds from S0 to S1, entity two makes a corresponding transition from S2 to S3. Because the handshake process is bidirectional, the synchronization point is executed correctly regardless of which entity arrives first at that point.
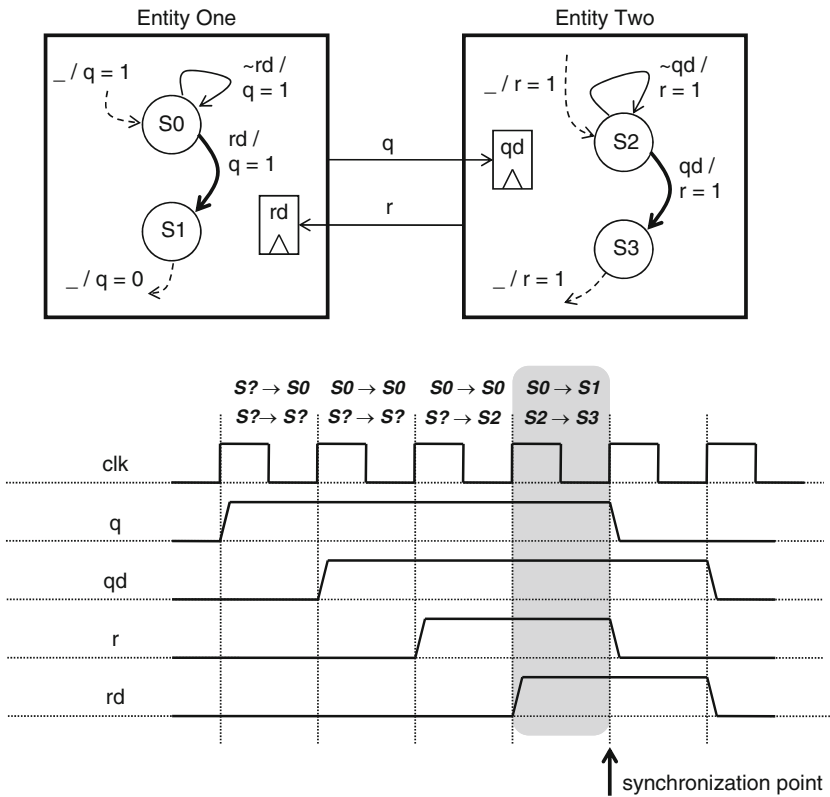
Fig. 9.7 Two-way handshake

There are still some opportunities for optimization. For example, we can de-assert the response signal already during the synchronization point, which will make the complete handshake cycle faster to complete. We can also design the protocol such that it uses level transitions rather than absolute signal levels. Some of these optimizations are explored in the Problems.

9.2.4 Blocking and Nonblocking Data-Transfer

Semaphores and handshakes implement the idea of a synchronization point. A hardware/software interface uses a synchronization point to transfer data. The actual data transfer is implemented using a suitable hardware/software interface, as will be described later in this chapter.

An interesting aspect of the data transfer is how a synchronization point should be implemented in terms of the execution flow of the sender or receiver. If a sender or receiver arrives too early at a synchronization point, should it wait idle until

the proper condition comes along, or should it go off and do something else? In terms of the send/receive operations in hardware and software, these two cases are distinguished as *blocking* data transfers and *nonblocking* data transfers.

A blocking data transfer will stall the execution flow of the software or hardware until the data-transfer completes. For example, if software has implemented the data transfer using function calls, then a blocking transfer would mean that these functions do not return until the data transfer has completed. From the perspective of the programmer, these primitives are the easiest to work with. However, they can stall the entire program.

A nonblocking data transfer will not stall the execution flow of software or hardware, but the data transfer may be unsuccessful. So, a software function that implements a nonblocking data transfer will need to introduce an additional status flag that can be tested. Nonblocking data transfers will not stall an entire program, but they require additional attention of the programmer to deal with exception cases.

Both of the semaphore and handshake schemes discussed earlier implement a blocking data transfer. The key observation of this section was that the data transfer between two parallel entities needs a synchronization mechanism. Semaphores and handshakes are two well known schemes to implement this synchronization. In the following sections, we look into the actual implementation of these communication channels in hardware and software.

9.3   Memory-Mapped Interfaces

A memory-mapped interface allocates part of the address space of a processor for communication between hardware and software. The memory-mapped interface is the most general, most wide-spread type of hardware/software interface. This is no surprise: memory is a central concept in software, and it's supported at the level of the programming language through the use of pointers. In this section, we look into the operation of memory-mapped registers, and into extended concepts such as mailboxes, queues, and shared memory. We also discuss the GEZEL modeling of memory-mapped interfaces.

9.3.1   The Memory-Mapped Register

A memory-mapped interface can be as simple as a register which can be read and written through bus transfers on an on-chip bus. Figure 9.8 illustrates the generic setup of memory-mapped register, and it identifies the main components of a memory-mapped interface. The register will be accessed when a given memory address, or an address within a given range, appears on the bus. The memory address, and the related bus command, is analyzed by an address decoder. This decoder
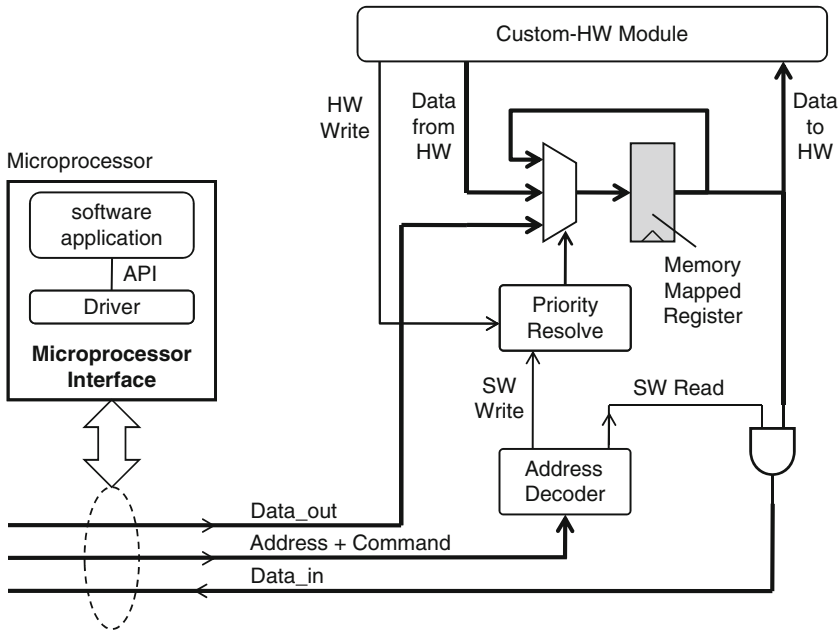
Fig. 9.8 A memory-mapped register

will generate a read pulse or a write pulse for the register. A full decoder will generate these pulses for a single address value. However, the complexity of the decoder is proportional to the number of bits that must be decoded. Therefore, it may be cheaper to build a decoder for a range of memory addresses (See Problem 9.4). The result of such a decoder is that a register is *aliased* as multiple memory locations.

A memory-mapped register works as a shared resource between software and hardware. A write-conflict may occur if the hardware and the software attempt to write into the register during the same clock cycle. To resolve this case, a priority decoder can be added that will either give preference to the hardware or the software on these conflicting write operations. Note that it typically does not make sense to *sequentialize* the write operations into the register, since one value would overwrite the other.

In software, the representation of a memory-mapped register is easy to do using an initialized pointer as follows.

```
volatile unsigned int *MMRegister = (unsigned int *) 0x8000;

// write the value '0xFF' into the register
*MMRegister = 0xFF;

// read the register
int value = *MMRegister;
```

```
volatile int *p = (int *) 0x8000;
int *p = (int *) 0x4000;
```
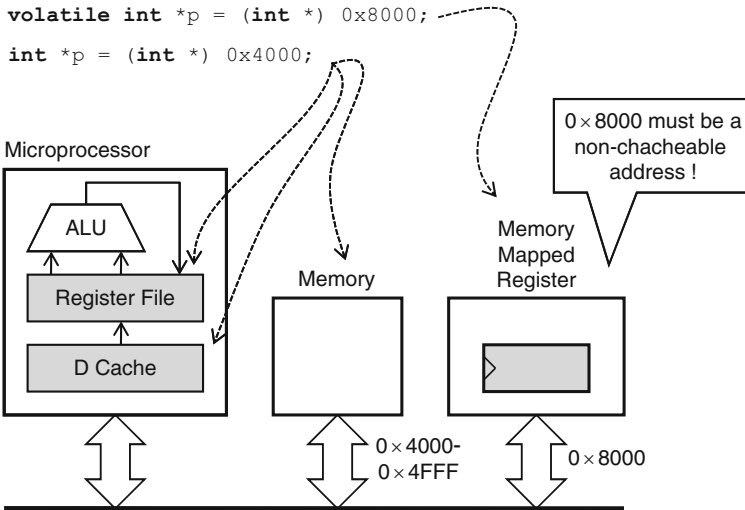
Fig. 9.9 Integrating a memory-mapped register in a memory hierarchy

Figure 9.9 explains why the pointer must be a `volatile` pointer. A memory-mapped register is integrated into the memory hierarchy of a processor, at the level of main memory. When a processor instruction will read or write from that register, it will do so through a memory-load or memory-store operation. Through the use of the `volatile` qualifier, the C compiler will treat the memory hierarchy slightly different.

- When using normal pointer operations, the processor and the compiler will attempt to minimize the number of operations to the main memory. This means that the value stored at an `int *` can appear in three different locations in the memory hierharchy: in main memory, in the cache memory, and in a processor register.
- By defining a register as a `volatile int *`, the compiler will avoid keeping a copy of the memory-mapped register in the processor registers. This is essential because a memory-mapped register can be updated by a custom-hardware module, outside of the control of a microprocessor. Hence, keeping backup copies inside of the processor cache or the processor register file cannot work.

However, defining a memory-mapped register with a `volatile` pointer will not prevent that memory address from being cached. Therefore, the memory addresses that include memory-mapped register must always be allocated into a noncacheable memory area of a processor. This is a part of the configuration of the processor cache. Building on the principle of a memory-mapped register, we will now create communication structures to tie hardware and software together.

9.3.2 Mailboxes

A mailbox is a simple extension of a memory-mapped register with a handshake mechanism. The obvious problem with a memory-mapped register by itself is that the hardware cannot tell when the software has written or read the register, and vice versa. Thus, what we need is the equivalent of a mailbox: a box with a little flag to signal its state. Suppose that we are sending data from software to hardware, then the software writes into the register, and next sets a "mailbox full" flag. The hardware, which keeps an eye on the mailbox flag, will then read the value in the register, and clears the "mailbox full" flag.

This construct is easy to build using three memory mapped registers, as illustrated in Fig. 9.10. In this case, the sender is the software program on the left of the figure, and the receiver is the hardware module on the right of the figure. After writing fresh data into the `data` memory-mapped register, the `req` flag is raised. The hardware component is a finite state machine that scans the state of the `req` flag and, as soon as the flag goes high, will capture the data, and raise the `ack` flag in response. Meanwhile, the software program is waiting for the `ack` flag to go high. Once both `ack` and `req` are high, a similar sequence is followed to reset them again.

The entire protocol thus goes through four phases: `req` up, `ack` up, `req` down, and `ack` down. Because both parties transition through the protocol in an interlocked fashion, the protocol automatically adapts to the speed of the slowest component. The protocol has two synchronization points: once just after both `ack` and `req` have transitioned high, and a second time just after both `ack` and `req` are low. This means that it is quite easy to double the throughput of the protocol in Fig. 9.10 (See Problem 9.5).

A mailbox based on memory-mapped registers has a high overhead, in terms of design cost as well as in terms of performance. The frequent synchronization of hardware and software through handshakes has two disadvantages. First it requires
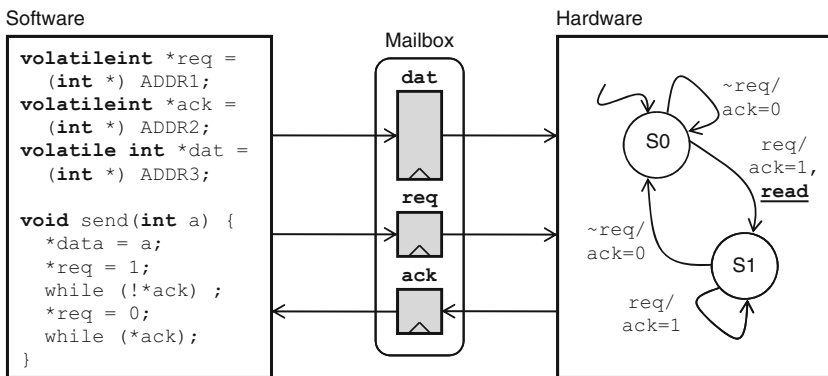

Fig. 9.10 A mailbox register between hardware and software

a fine-grained interlocking of the execution flow of hardware and software. Keep in mind that each four-phase handshake implies two synchronization points. These interlocked execution flows are harder to design and to control. The second disadvantage is that frequent synchronization will generate many additional bus transfers. For example, consider the `while` statements in the C program in Fig. 9.10. Each iteration in the while loop generates one read from a `volatile` pointer, resulting in one bus transfer.

Both of these problems – tight coupling and extra bus transfers – can be solved by improving the buffer mechanism between hardware and software. We will discuss two examples. The first is to use *FIFO queues* instead of mailboxes to uncouple hardware and software and to remove the need for interlocked read/write of the memory-mapped interface. The second is to use *shared memory*. This can reduce the need for synchronization by increasing the granularity of the data transfer, from a single word to an entire memory block.

9.3.3   First-In First-Out Queues

When a handshake protocol is used to implement a mailbox, the write and read operations are interleaved. This is inconvenient when the rates of write operations and read operations is very different. For example, the writes into the FIFO could be bursty, so that several tokens are written in rapid succession, while the reads from the FIFO could be very regular, so that tokens are read with a regular pace. The role of the FIFO is to store the extra tokens during write operations, and to gradually release them during read operations. Of course, in the long term, the average number of writes into such a buffer must be equal to the average number of reads: a FIFO only addresses a short-term need.

Handshakes are also a solution for this type of communication channel. Instead of having a single pair of `request/acknowledge` signals, we will now have *two* pairs. One pair controls the write operations into the FIFO, while the second pair controls the read operations into the FIFO. Figure 9.11 illustrates a FIFO queue with individual handshakes for read and write operations into the queue. In this case, we have assumed a FIFO with eight positions. The GEZEL code on the left of Fig. 9.11 shows the register-transfer level operations. In this case, the handshake operations are tied to incrementing a read-pointer and a write-pointer of a dual-port memory. The increment operations are conditional on the state of the FIFO, and the level of the `request` inputs. The state of the FIFO can be empty, full or nonempty, and this condition is evaluated based on comparing the read and write pointer values. Encoding the status of the FIFO using only the value of the read and write pointer values has a negative side-effect: the code shown in Fig. 9.11 requires always at least a single empty space in the FIFO buffer. By introducing a separate `full` flag, all spaces in the buffer can be utilized (See Problem 9.6).
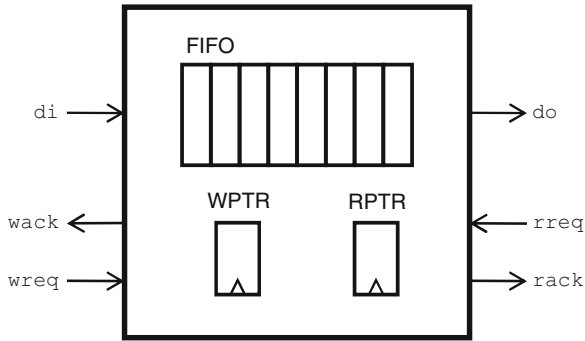
```
dp fifo(in  di   : ns(8);
        in  wreq : ns(1);
        out wack : ns(1);
        out do   : ns(8);
        in  rreq : ns(1);
        out rack : ns(1)) {
   sig read, write : ns(1);
   reg rptr, wptr : ns(3);
   use dualport_mem(di, wptr, write,  // write port
                    do, rptr, read);  // read port
 always{
   read  = (rreq & (wptr != rptr)) ? 1 : 0;
   write = (wreq & ((wptr + 1) != rptr) ? 1 : 0;
   wptr  = write ? wptr + 1 : wptr;
   rptr  = read ? rptr + 1 : rptr;
   wack  = write;
   rack  = read;
 }
}
```

Fig. 9.11 A FIFO with handshakes on the read and write ports

9.3.4 Slave and Master Handshakes

The FIFO shown in Fig. 9.11 has two *slave* interfaces: one for writing and one for reading. A slave interface waits for a control signal of a connecting interface and reacts to it. Thus, the acknowledge signals will be set in response to the request signals. There is a matching master protocol required for a slave protocol. In the hardware/software interface of Fig. 9.10, the software interface uses a master-protocol and the hardware interface uses a slave-protocol.

By building a FIFO with a slave input and a master output, multiple sections of FIFO can be connected together to build a larger FIFO. An example of this scheme is shown in Fig. 9.12. In this implementation, we use a FIFO with a single storage location, implemented as a register. The updates of this register are under control of a finite state machine, which uses the request/acknowledge handshake signals as
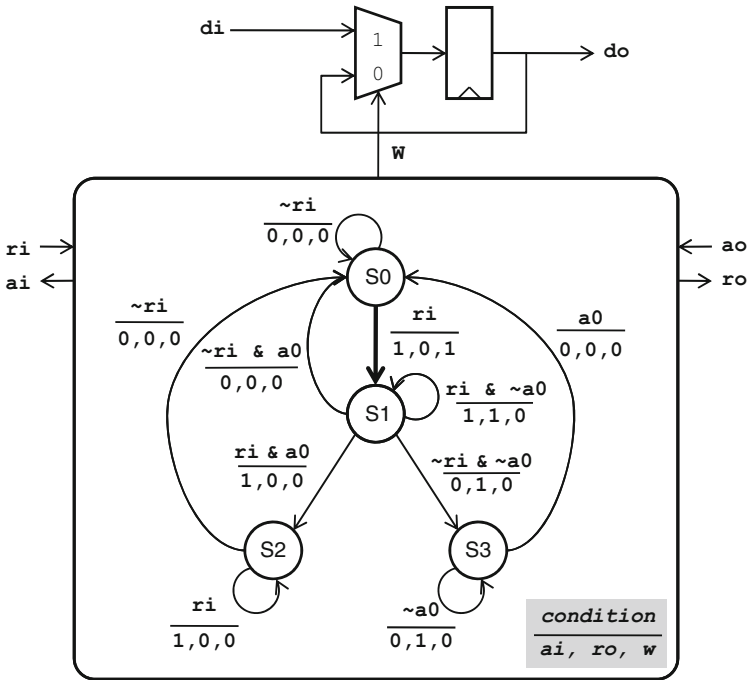
Fig. 9.12   A one-place FIFO with a slave input handshake and a master output handshake

inputs. Note the direction of the request/acknowledge arrows on the input port and the output port of the FIFO. At the input port, the request signal is an input, while at the output port, the request signal is an output.

A master-type of handshake interface can only be connected to a slave-type of handshake interface, otherwise the handshake protocol does not work. The FIFO in Fig. 9.12 operates as follows. Initially, the FSM is in state S0, waiting for the input request signal to be set. Once it is set, it will write the value in the input port into the register and transition to state S1. In state S1, the FSM sets the request signal for the output port, indicating that the FIFO stage is nonempty. From state S1, three things can happen, depending on which handshake (input or output) completes first. If the input handshake completes (ri falls low), the FSM goes to state S4. If the output handshake responds (a0 raises high), the FSM goes to state S2. If both handshakes complete at the same time, the FSM directly goes back to S0.

9.3.5   Shared Memory

Instead of controlling the access on a single register, a single handshake can also be used to control access to a region of memory. In that case, a shared-memory scheme is obtained, such as illustrated in Fig. 9.13.

Software

```
volatile int *req =
   (int *) ADDR1;
volatile int *ack =
   (int *) ADDR2;

void send(int a) {
 // modify region 1
 // ...
 *req = 1;
 while (!*ack) ;

 // modify region 2
 // ...
 *req = 0;
 while (*ack);
}
```

Shared Memory

ack	req

Region 2

Region 1

Hardware

On-chip Bus
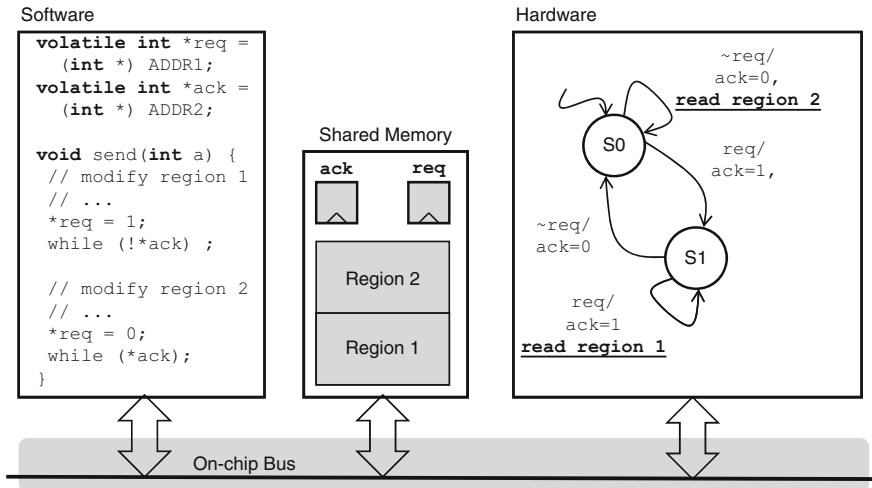
Fig. 9.13 A double-buffered shared memory with a memory-mapped request/acknowledge hand-shake

In this example, a memory module is combined with two memory-mapped registers. The registers are used to implement a two-way handshake. The memory is split up in two different regions. The handshake protocol is used control access to these regions. In one phase of the protocol, changes are only done to one region of the memory. In the second phase of the protocol, changes are only done to the other region of the memory. This way, the hardware module can be sure that all data values in a given section of memory are consistent. This can be used to exchange large data records such as images, internet packets, file headers, and so on. Shared memory is a convenient approach to implement distributed memory, as was discussed earlier in Chap. 7. In addition, memory access may be optimized by clever organization of the read/write access patterns into a shared memory.

9.3.6 GEZEL Modeling of Memory-Mapped Interfaces

To conclude our discussion on memory-mapped interfaces. We describe the modeling of memory-mapped interfaces in GEZEL. A memory-mapped interface is represented using a dedicated simulation primitive called an `ipblock`. There is a separate primitive for a read-interface and a write-interface, and each is mapped to a user-specified memory address.

The following example shows the modeling of a memory-mapped interface for a coprocessor that evaluates the Greatest Common Divisor Algorithm. The coprocessor uses a single input and a single output, and is controlled using memory-mapped registers. Listing 9.3 illustrates the design of the coprocessor. Lines 1–28

contain 5 `ipblock`, representing the software processor and the memory-mapped hardware/software interfaces. These interfaces are not modeled as actual registers, but merely as modules capable of decoding read operations and write operations to a given memory bus. The decoded addresses are given as a parameter to the `ipblock`. For example, the request signal of the handshake is mapped to address `0x80000000`, as shown on line 6–10. The coprocessor kernel is shown on lines 30–43, and is a standard implementation of the greatest-common-divisor algorithm similar to the one used in earlier examples in this book. The hardware–software interface logic is embedded in the interface module, included on lines 45–101, which links the `ipblock` with this datapath. This module is easiest to understand by inspecting the FSM description. For each GCD computation, the hardware will go through two complete two-way handshakes. The first handshake (lines 84–89) provides the two operands to the GCD hardware. These operands are provided sequentially, over a single input port. After the first handshake, the computation starts (line 92). The second handshake (lines 93–100) is used to retrieve the result. This approach of tightly coupling the execution of the algorithm with the hardware/software interface logic has advantages and disadvantages: it results in a compact design, but it also reduces the flexibility of the interface. In the next chapter, we will discuss design techniques to avoid this tight coupling.

Listing 9.3 A memory-mapped coprocessor for GCD

```
1   ipblock my_arm {
2     iptype "armsystem";
3     ipparm "exec=gcddrive";
4   }
5
6   ipblock m_req(out data : ns(32)) {
7     iptype "armsystemsource";
8     ipparm "core=my_arm";
9     ipparm "address=0x80000000";
10  }
11
12  ipblock m_ack(in data : ns(32)) {
13    iptype "armsystemsink";
14    ipparm "core=my_arm";
15    ipparm "address=0x80000004";
16  }
17
18  ipblock m_data_out(out data : ns(32)) {
19    iptype "armsystemsource";
20    ipparm "core=my_arm";
21    ipparm "address=0x80000008";
22  }
23
24  ipblock m_data_in(in data : ns(32)) {
25    iptype "armsystemsink";
26    ipparm "core=my_arm";
27    ipparm "address=0x8000000C";
28  }
29
```

```
30  dp euclid(in  m_in, n_in : ns(32);
31            in  go         : ns( 1);
32            out ready      : ns( 1);
33            out gcd        : ns(32)) {
34    reg m, n : ns(32);
35    sig done : ns(1);
36
37    always { done  = ((m==0) | (n==0));
38            ready = done;
39            gcd   = (m > n) ? m : n;
40            m     = go ? m_in : ((m > n) ? m - n : m);
41            n     = go ? n_in : ((n >= m) ? n - m : n);
42          }
43  }
44
45  dp tb_euclid {
46    sig m, n  : ns(32);
47    sig ready : ns(1);
48    sig go    : ns(1);
49    sig gcd   : ns(32);
50    use euclid(m, n, go, ready, gcd);
51
52    use my_arm;
53
54    sig req, ack, data_out, data_in : ns(32);
55    use m_req(req);
56    use m_ack(ack);
57    use m_data_out(data_out);
58    use m_data_in(data_in);
59
60    reg r_req    : ns(1);
61    reg r_done   : ns(1);
62    reg r_m, r_n : ns(32);
63
64    always   { r_req  = req;
65              r_done  = ready;
66              data_in = gcd;
67              m       = r_m;
68              n       = r_n;
69            }
70    sfg ack1 { ack = 1;        }
71    sfg ack0 { ack = 0;        }
72    sfg getm { r_m = data_out; }
73    sfg getn { r_n = data_out; }
74    sfg start{ go  = 1;        }
75    sfg wait { go  = 0;        }
76  }
77
78  fsm ctl_tb_euclid(tb_euclid) {
79    initial s0;
80    state s1, s2, s3, s4, s5, s6;
81
82    @s0 (ack0, wait) -> s1;
83
```

```
84     // read m
85     @s1 if (r_req)   then (getm, ack1, wait) -> s2;
86                      else (ack0, wait)        -> s1;
87     // read n
88     @s2 if (~r_req) then (getn, ack0, wait) -> s3;
89                      else (ack1, wait)        -> s2;
90
91     // compute
92     @s3 (start, ack0) -> s4;
93     @s4 if (r_done) then (ack0, wait)         -> s5;
94                      else (ack0, wait)         -> s4;
95
96     // output result
97     @s5 if (r_req)   then (ack1, wait)         -> s6;
98                      else (ack0, wait)         -> s5;
99     @s6 if (~r_req) then (ack0, wait)         -> s1;
100                     else (ack1, wait)         -> s6;
101 }
102
103 system S {
104   tb_euclid;
105 }
```

Listing 9.4 shows a C driver program that matches the coprocessor design of Listing 9.3. The program evaluates the GCD operation of the numbers 80 and 12, followed by the GCD of the numbers 80 and 13. Note the difference between a master-handshake protocol, as shown in the functions sync1() and sync0(), and a slave-handshake protcol, as illustrated in the FSM transitions in Listing 9.3. In a master handshake, the request signals are first written and followed by a test on the acknowledge signals. In a slave handshake, the request signals are first tested, and followed by a write on the acknowledge signals.

Listing 9.4 A C driver for the GCD memory-mapped coprocessor

```
1   #include <stdio.h>
2   volatile unsigned int *req = (unsigned int *) 0x80000000;
3   volatile unsigned int *ack = (unsigned int *) 0x80000004;
4
5   void sync1() {
6     *req = 1;  while (*ack == 0) ;
7   }
8
9   void sync0() {
10    *req = 0;  while (*ack == 1) ;
11  }
12
13  int main() {
14    volatile unsigned int *di  = (unsigned int *) 0x80000008;
15    volatile unsigned int *ot  = (unsigned int *) 0x8000000C;
16
17    *di = 80;
18    sync1();
19    *di = 12;
20    sync0();
```

```
21    sync1();
22    printf("gcd(80,12) = %d\n", *ot);
23    sync0();
24
25    *di = 80;
26    sync1();
27    *di = 13;
28    sync0();
29    sync1();
30    printf("gcd(80,13) = %d\n", *ot);
31    sync0();
32
33    return 0;
34  }
```

Executing this cosimulation is easy, and follows similar steps as discussed on Chap. 7. We first cross-compile the C program to an executable. Next, we run the cosimulator on the executable and the GEZEL program.

```
> arm-linux-gcc -static  gcddrive.c -o gcddrive
> gplatform  gcd.fdl
core my_arm
armsystem: loading executable [gcddrive]
armsystemsink: set address 2147483652
armsystemsink: set address 2147483660
gcd(80,12) = 4
gcd(80,13) = 1
Total Cycles: 11814
```

In conclusion, memory-mapped interfaces are a general-purpose and easy-to-use mechanism to create hardware/software interfaces. The principle of a two-way handshake is applicable to many different situations, and it ensures that a simple shared storage location is sufficient to synchronize hardware and software and to implement communications. Because memory-mapped interfaces rely on a general-purpose on-chip bus, they become easily constrained when throughput requirements increase. In addition, because the on-chip bus is shared with other components, a memory-mapped interface will also show a varying latency. For cases that require a dedicated, tightly controlled link, we will use a coprocessor interface.

9.4 Coprocessor Interfaces

In cases where high-throughput between the software and the custom hardware is needed, it makes sense to have a dedicated interface to attach this hardware. Such a dedicated interface is called a coprocessor interface. As illustrated in Fig. 9.14, a coprocessor interface does not make use of the on-chip bus, but instead uses a dedicated port on the processor. This port is driven using a particular set of instructions, the coprocessor instructions. The coprocessor instruction set is different for each type of processor, since it depends on the properties of the coprocessor interface.
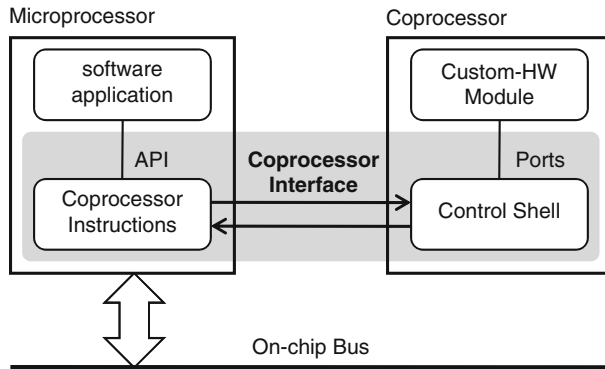
Fig. 9.14 Coprocessor interface

A classic example of a coprocessor is a floating-point calculation unit, which is attached to an integer core. Not all processors have a coprocessor interface.

The choice of designing a custom hardware module for a particular coprocessor interface is an important design decision: it locks the custom hardware design into a particular processor. For example, a hardware module designed for an AMBA *bus* is likely to have a wider range of users than a hardware module designed for an ARM *coprocessor interface*. The former module can run on any system with an AMBA bus (even those which do not have an ARM processor), while the latter module is restricted to systems with an ARM processor.

The main advantages of a coprocessor interface over an on-chip bus are higher throughput and fixed latency. We consider each of these aspects.

- Coprocessor interfaces have a higher throughput then memory-mapped interfaces because they are not constrained by the processor wordlength. For example, coprocessor ports on 32-bit CPUs may support 64-bit or 128-bit interfaces, allowing them to transport four words per coprocessor instruction. Hardware/software interfaces based on coprocessor instructions may also be implemented more efficiently than load/store instructions. A coprocessor instruction commonly specifies two source operands and one destination operand. In contrast, a load/store instruction will specify only a single operand. A complete hardware/software interaction over a coprocessor interface may thus be specified with fewer instructions as the same interaction over an on-chip bus.
- A coprocessor interface can also maintain fixed latency, so that the execution timing of software and hardware is precisely known. Indeed, a coprocessor bus is a dedicated connection between hardware and software, and it has a stable, predictable timing. This, in turn, can simplify the implementation of hardware/software synchronization mechanisms. In contrast, an on-chip bus interface uses a communication medium which is shared between several components, and which may include unknown factors such as bus bridges. This leads to unpredictable timing for the hardware/software interaction.

9.4.1 Tight and Loose Coupling

A comparison between the key features of a coprocessor interface and a memory-mapped interface is shown in Table 9.1. An interesting point in this table is the distinction between a *tightly coupled* and a *loosely coupled* interface. Coupling indicates the level of interaction between the execution flow in software and the execution flow in custom-hardware. In a tight coupling scheme, custom-hardware and software synchronize often, and exchange data often, for example at the granularity of a few instructions in software. In a loose coupling scheme, hardware and software synchronize infrequently, for example at the granularity of a function or a task in software.

A given application can use either tight- or loose-coupling. Figure 9.15 shows how the choice for loose-coupling of tight-coupling can affect the latencies of the application. The left side of the figure illustrates a tight-coupling scheme. The software will send four separate data items to the custom hardware, each time collecting the result. The figure assumes a single synchronization point which sends the operand and retrieves the result. This is the scheme that would be used by a coprocessor interface. The synchronization point corresponds to the execution of a coprocessor instruction in the software. The right side of the figure illustrates a loosely coupled scheme. In this case, the software provides a large block of data to the custom hardware, synchronizes with the hardware, and then waits for the custom hardware to complete processing and return the result. This scheme would be used by a memory-mapped interface, for example using a shared memory. The correct choice between tight coupling and loose coupling depends on the application and the target architecture. Loosely coupled schemes tend to yield slightly more complex hardware designs because the hardware needs to deal more extensively with data movement between hardware and software. However, there is no single correct choice between tight and loose coupling.

In the following sections, we will discuss two coprocessor interfaces: the Fast Simplex Link (FSL) interface, used by the Microblaze soft-core processor, and the LEON-3 floating point interface.

Table 9.1 Comparing a coprocessor interface with a memory-mapped interface

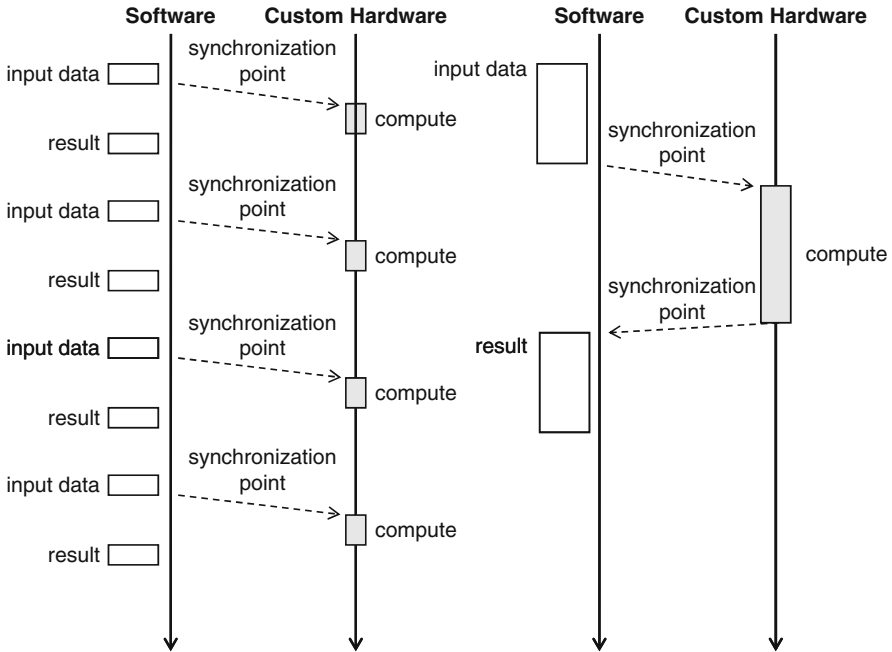| Factor	Coprocessor interface	Memory-mapped interface
Addressing	Processor-specific	On-chip bus address
Connection	Point-to-point	Shared
Latency	Fixed	Variable
Throughput	Higher	Lower
Typical-use	Tightly coupled	Loosely coupled

Fig. 9.15 Tight coupling vs. loose coupling

9.4.2 The Fast Simplex Link

The Microblaze processor, a soft-core processor that can be configured in a Xilinx FPGA, is configurable with up to 8 Fast Simplex Link (FSL) interfaces. An FSL link consists of an output port with a master-type handshake, and an input port with a slave-type handshake. The *simplex* aspect of FSL relates to the direction of data, which is either output or input. The Microblaze processor has separate instructions to write to the output port and read from the input port.

Figure 9.16 shows a single FSL interface. In between the hardware coprocessor and the Microblaze, FIFO memories can be added to adjust for differences in the rates for reading and writing. Data going from the Microblaze to the FIFO goes over a master interface consisting of the signals data, write, and full. Data going from the FIFO to the Microblaze goes over a slave interface which includes the signals data, exists, and read. The labeling of handshake signals is slightly different than what we discussed before: write and exists correspond to req, while full and read correspond to ack.

The operation of the FSL interface for a FIFO with two positions is shown in Fig. 9.17. This figure shows the activities of writing three tokens into the FIFO, and reading them out again. The operation will be familiar because the FSL protocol uses a two-way handshake. On clock edge 2, the first data item is written into the FIFO. The exists flag is raised because the FIFO is nonempty. On clock edge 3,

Microblaze Processor

Coprocessor
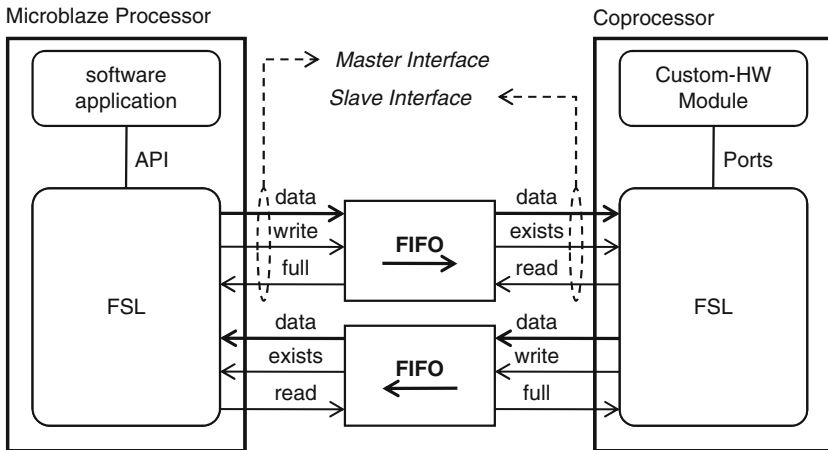
Fig. 9.16 The fast simplex link interface
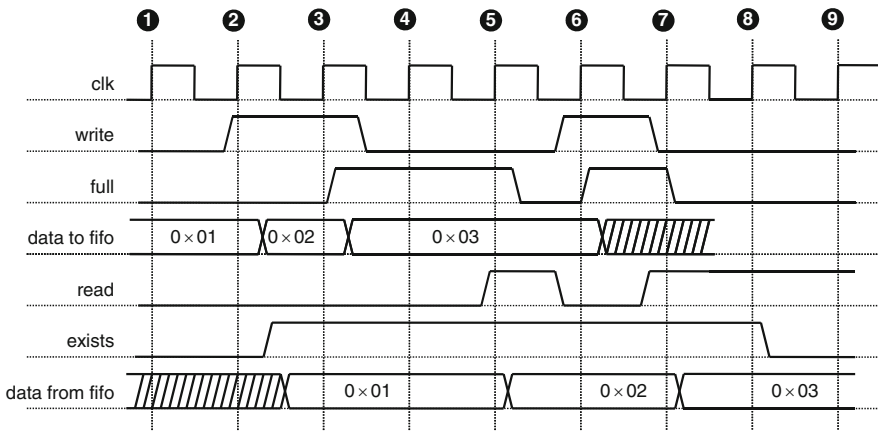
Fig. 9.17 The FSL handshake protocol

a second data item is written into the FIFO. We assume that the FIFO holds no more than two places, so that this second write will fill the FIFO completely, and the `full` flag is raised as a result. In order to write more tokens into the FIFO, at least one read operation needs to complete first. This happens on clock edge 5. The third data item is written into the FIFO on clock edge 6, which completely fills the FIFO again. The FIFO is emptied by read operations on clock edge 7 and 8. From clock edge 9 on, the FIFO is empty.

The read and write operations on an FSL interface are controlled using dedicated Microblaze instructions. The basic read and write operations are of the form:

```
put  rD, FLSx   // copy register rD to FSL interface FSLx
get  rD, FSLx   // copy FSL interface FSLx to register rD
```

There are many variants of these instructions, and we only discuss the main features.

- The instruction can be formulated as blocking as well as nonblocking operations. When a nonblocking instruction is unable to complete a read or a write operation, it will reset the carry flag in the processor. This way, a conditional jump instruction can be used to distinguish a successful transfer from a failed one.
- The FSL I/O instructions can also read a control status flag directly from the hardware interface: The *data* bus shown in Fig. 9.16 physically consists out of a 32-bit data word and a single-bit control flag. An exception can be raised if the control bit is different from the expected value. This allows the hardware to influence the control flow in the software.
- The FSL I/O instructions can also be formulated as *atomic* operations. In that case, a group of consecutive FSL instructions will run as a single set, without any interrupts. This is useful when the interface to a hardware module is created using several parallel FSL interfaces. By disallowing interrupts, the designer can be sure that all FSL interfaces are jointly updated.

The FSL interface is a very popular coprocessor interface in the context of FPGA designs. It uses a simple hardware protocol, and is supported with a flexible, yet dedicated instruction set on the processor. However, it's only a data-moving interface. In the next section, we will discuss a floating-point coprocessor interface. Such an interface has a richer execution semantics.

9.4.3   The LEON-3 Floating Point Coprocessor Interface

The interface in this section is a tightly coupled interface to attach a floating-point unit (FPU) to a processor. While a high-end desktop processor has the FPU built-in, embedded processors often configure this as an optional extension. The FPU interface discussed in this section is the one used in the LEON-3 processor, designed by Gaisler Research. It is a tightly coupled interface because instructions executed on the FPU need to remain in sync with the instructions executing on the main processor.

Figure 9.18 illustrates the main signals of the FPU interface. The LEON-3 32-bit microprocessor includes an integer-instruction pipeline, a set of floating-point registers, and an instruction-fetch unit. When the microprocessor fetches a floating-point instruction, it will dispatch that instruction to the floating-point coprocessor. After the result of the floating point operation is returned to the microprocessor, it is merged with the flow of instructions in the integer pipeline. There are several interesting issues with this scheme, and the signals on the coprocessor interface can best be understood by examining the interaction between the FPU and the microprocessor in detail.

The FPU contains two different datapaths. One is a *linear* pipeline with three pipeline stages. The second is a *nonlinear* pipeline, and it consists of a pipeline
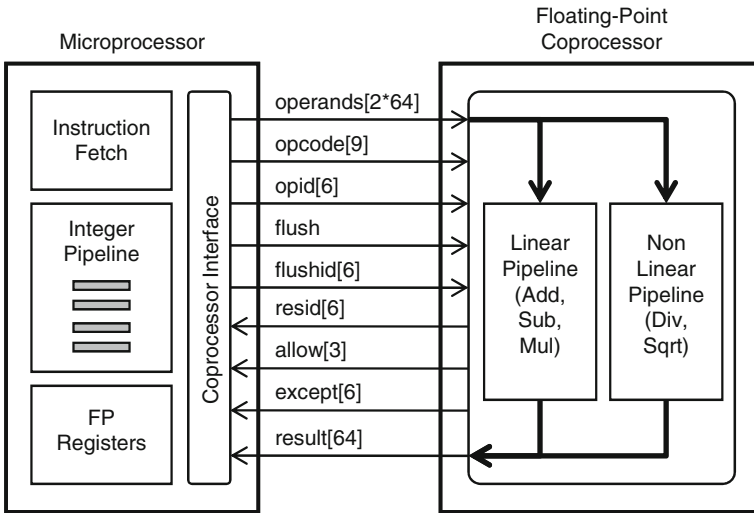
Fig. 9.18 GRFPU floating point coprocessor interface

with feedback, so that one pipeline stage remains in use for several cycles. FPU operations such as add, subtract, and multiply are handled by the linear pipeline, while operations such as divide and square-root are handled by the nonlinear pipeline. FPU instructions through the linear pipeline have a latency of three clock cycles and a throughput of one instruction per clock cycle. However, FPU instructions through the nonlinear pipeline can have a latency of up to 24 clock cycles, and their throughput can be as low as 1 instruction every 24 clock cycles.

The challenge of the coprocessor interface is to maintain the instruction sequence in the FPU synchronized with the microprocessor. This is nontrivial because the latency and throughput of intructions in the FPU is irregular. Indeed, results must be merged in the microprocessor in the same order as their operands are dispatched to the FPU. Figure 9.19 demonstrates that, due to the complex pipeline architecture of the FPU however, results may arrive out-of-order. Here is how the interface handles this problem.

- Each operation for the FPU is provided as a set of operands, with a given opcode, and an instruction identifier opid. When the FPU operation finishes, it returns the result, toghether with a corresponding instruction identifier resid. By examining the value of resid, the processor can determine what result corresponds to what set of operands. The instruction identifier is generated by the processor, but is typically a simple counter (similar to the labels in the grids on Fig. 9.19). For each result, the coprocessor will also generate an exception code except, which allows the detection of overflow, divide-by-zero, and so on.
- When a floating-point instruction appears just after a conditional branch, the floating-point instruction may need to be cancelled when the conditional branch
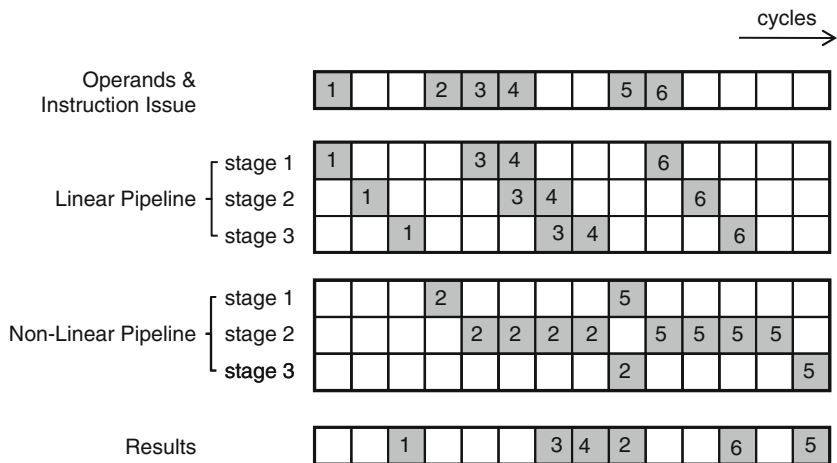
Fig. 9.19 GRFPU Instruction issue and result return

would be taken. Through the control signal `flush`, the microprocessor can indicate that the coprocessor should cancel an instruction. The instruction identifier is provided through `flushid`.

- Because of the nonlinear pipeline in the coprocessor, not all types of FPU instructions can be accepted every clock cycle. The `allow` signal indicates to the microprocessor what instructions can start at a given clock cycle.

Clearly, this interface definition is highly specialized toward floating-point coprocessors. In addition, the interface is tightly coupled with the microprocessor. Instead of handshake signals, a series of control signals is defined that ensures that the execution flow of the coprocessor and the microprocessor will stay synchronized. This leads to an even more processor-specific interface: the custom-instruction interface.

9.5 Custom-Instruction Interfaces

The integration of hardware and software can be considerably accelerated with the following idea: reserve part of the opcodes of a microprocessor for new instructions, and use these opcodes to execute custom-hardware modules. Next, integrate the custom-hardware modules directly into the microarchitecture of the microprocessor. As this solution does not make use of an on-chip bus interface or a coprocessor interface, it is highly processor specific. The resulting design is called an *Application-Specific Instruction-set Processor* or ASIP for short.

In an ASIP design, we benefit from the instruction-fetch and dispatch mechanism in the microprocessor to ensure that custom-hardware and software remain synchronized. In addition, the hardware/software codesign problem is formulated

as a problem of finding the proper application-specific instruction-set, which many designers experience as an easier problem. This has made the ASIP concept very populuar.

9.5.1 ASIP Design Flow

A very appealing aspect of ASIP design is that it uses an incremental design process. In constrast, the traditional hardware design process is bottom-up, exact and rigorous. When creating an ASIP, a designer can start with a C program that captures the required functionality of a function, and that maps the C program on a basic processor. After the performance of this function on the processor is evaluated, adjustments to the program/processor can be made. Such adjustments include for example new hardware in the processor datapath, and new processor instructions. After the processor hardware is adjusted, the C program can be tuned as well to make use of these instructions. This leads to a design flow as illustrated in Fig. 9.20.

The design starts with a C program and a description of the processor. The *processor description* is not a hardware description in terms of FSMD, but instead a specialized description of processor resources. It contains a description of instructions supported by the processor, the configuration and size of register files, and the memory architecture. Using the processor description, an *ASIP generator* will create design components for the ASIP. This includes a software development toolkit (compiler, assembler, linker, and debugger) as well as a synthesizable hardware description of the processor. Using the software development toolkit, the application
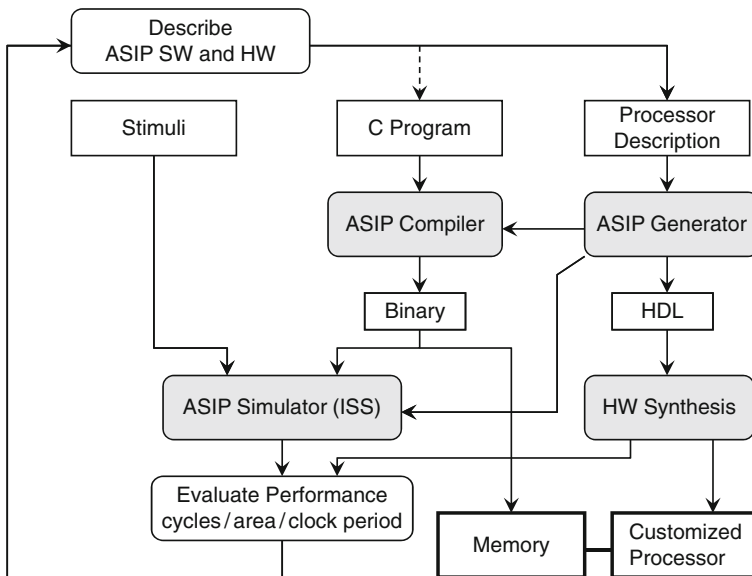

Fig. 9.20 ASIP design flow

program in C can be compiled, simulated, and evaluated. The hardware description can be technology-mapped onto gates or FPGA, which yields the processor implementation as well as technology metrics such as area and achievable processor clock. Based on the performance evaluation, the processor description can be reworked to give a better performance for a given application. This may also require rework of the application in C.

Compared to SoC design based on custom-hardware modules, will the ASIP design flow in Fig. 9.20 deliver better performance? Not in the general case. Keep in mind that the basic architecture template of an ASIP is a sequential processor. The fundamental bottlenecks of the sequential processor (memory-access, sequential execution of code) are also fundamental bottlenecks of an ASIP. Compared to SoC design based on custom-hardware modules, can the above design flow deliver less error-prone results? Yes, it can for the following two reasons. First, the design process is incremental. A functional error will be detected very quickly, in the early phases of the design. Second, it works at a higher level of abstraction. The application is modeled in C. The processor description language (which will be discussed later) is also at a higher level of abstraction than hardware description languages. In the past few years, a tremendous progress has been made on design tools that support the ASIP design flow. All of the shaded boxes in Fig. 9.20 can be obtained as commercial tools.

9.5.2 Example: Endianess Byte-Ordering Processor

We describe an example of ASIP design, including GEZEL modeling of the custom-instruction. The application is an endianess byte-ordering processor.

Figure 9.21 illustrates the problem. Processors have a chosen byte-ordering or endianess. When a processor transmits data words over a network, the byte order of the transmitted packets is converted from the processor endianess into network byte-order, which is big-endian in practice. For example, a strongARM processor is little endian, and will store the word `0x12345678` with the lowest significant byte in the lowest memory address. However, when that word is transmitted over a network, the packet byte-order must follow a big-endian convention that will send `0x78` first and `0x12` last. The TCP/IP protocol stack on StrongARM will, therefore, convert each word from little-endian format to big-endian format before providing it to the network buffer on the Ethernet card.

For a 32-bit processor, endianness conversion involves byte-level manipulation using shifting and masking. The following is an example of a function that converts little-endian to big-endian (or vice versa) in C.

```
for (i=0; i<4096; i++)
   ot[i] = ( ((in[i] & 0x000000ff) << 24) |
             ((in[i] & 0x0000ff00) <<  8) |
             ((in[i] & 0x00ff0000) >>  8) |
             ((in[i] & 0xff000000) >> 24));
```
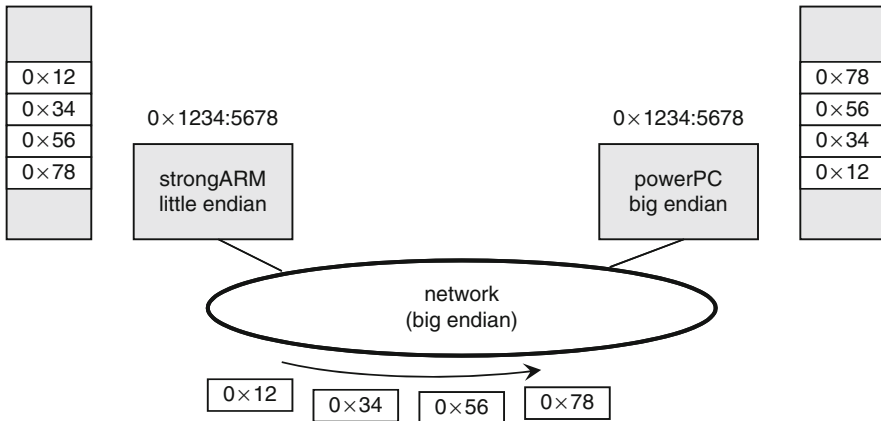
Fig. 9.21 Endianess byte-ordering problem

On a strongARM processor, this loop requires 13 cycles per iteration (assuming no cache misses). Examining the assembly, we would find that there are 11 instructions inside of the loop, leaving 2 cycles of pipeline stalls per iteration – one for the branch and one data-dependency (`ldr` instruction).

```
.L6:
    ldr r3, [r4, ip, asl #2]    ; read in[i]
    and r2, r3, #65280          ; evaluate ot[i]
    mov r2, r2, asl #8          ;    ...
    orr r2, r2, r3, asl #24     ;    ...
    and r1, r3, #16711680       ;    ...
    orr r2, r2, r1, lsr #8      ;    ...
    orr r2, r2, r3, lsr #24     ;    ...
    str r2, [r0, ip, asl #2]    ;   and write back
    add ip, ip, #1              ; next iteration
    cmp ip, lr                  ;
    ble .L6
```

We now consider improvements to this design. In hardware, the endianess-conversion is obviously very simple: it is a simple wiring pattern. The key problem in this design is how to move data from data memory and back. Before trying an ASIP, let's try to solve this problem with a memory-mapped coprocessor.

The GEZEL coprocessor, without the interface definition, looks as follows. It uses two memory-mapped registers, one for input and one for output.

```
dp mmswap(in d1 : ns(32); out q1 : ns(32)) {
  always {
    q1 = d1[ 7: 0] #
         d1[15: 8] #
         d1[23:16] #
         d1[31:24];
  }
}
```

The C driver program for this memory-mapped coprocessor looks as follows.

```
volatile unsigned int * mmin = (unsigned int *) 0x80000000;
volatile unsigned int * mmot = (unsigned int *) 0x80000004;
for (i=0; i<4096; i++) {
  *mmin = in[i];
  ot[i] = *mmot;
}
```

Looking at the assembly, the execution time of the loop body now takes 10 clock cycles per iteration (7 instructions, 1 stall for branching, 2 stall for data-dependencies). This is a gain of three cycles. We assume that we have single-cycle access to the memory-mapped coprocessor, which is unlikely in practice. Hence, the gain of three cycles will probably be lost in a real design. The disadvantage of this design is apparent from the assembly program: each data element travels *four* times over the memory bus in order to be converted.

```
.L21:
    ldr r3, [r0, ip, asl #2]   ; load in[i]
    str r3, [r4, #0]           ; send to coprocessor
    ldr r2, [r5, #0]           ; read from coprocessor
    str r2, [r1, ip, asl #2]   ; write ot[i]
    add ip, ip, #1             ; next iteration
    cmp ip, lr
    ble .L21
```

Using an ASIP design, this wasteful copying over the memory bus can be avoided: we can retrieve in[i] a single time, convert it inside of the processor, and write back the converted result to ot[i]. In order to do this as part of an instruction, we need to modify (extend) the datapath of the processor with a new operation. Figure 9.22 illustrates how this works: the execution stage of the pipeline is extended with a new datapath (endianess conversion), and a new instruction is integrated into the instruction decoder. While this approach looks conceptually simple, it is not straightforward to implement. First, opcode space is a scarce resource. In a processor with high code density, there will be very few redundant opcodes inside of an instruction set. Second, the datapath of a RISC processor is optimized for speed. All hardware added into the pipeline must run faster then the slowest pipeline stage, otherwise it will become a bottleneck for the processor.

GEZEL supports experiments with custom datapaths in a StrongARM processor, by using some of the unused opcodes of that processor. In particular, GEZEL supports 2-by-2 and 3-by-1 custom datapaths in a StrongARM. A 2-by-2 custom datapath reads two register operands and produces two register operands. A 3-by-1 custom datapath reads three register operands and produces a single register operand. The following GEZEL program shows a 2-by-2 custom datapath for endianess conversion. An ipblock of type armsfu2x2 is used to represent a custom instruction.

```
ipblock myarm {
  iptype "armsystem";
  ipparm "exec=nettohost_sfu";
}
```
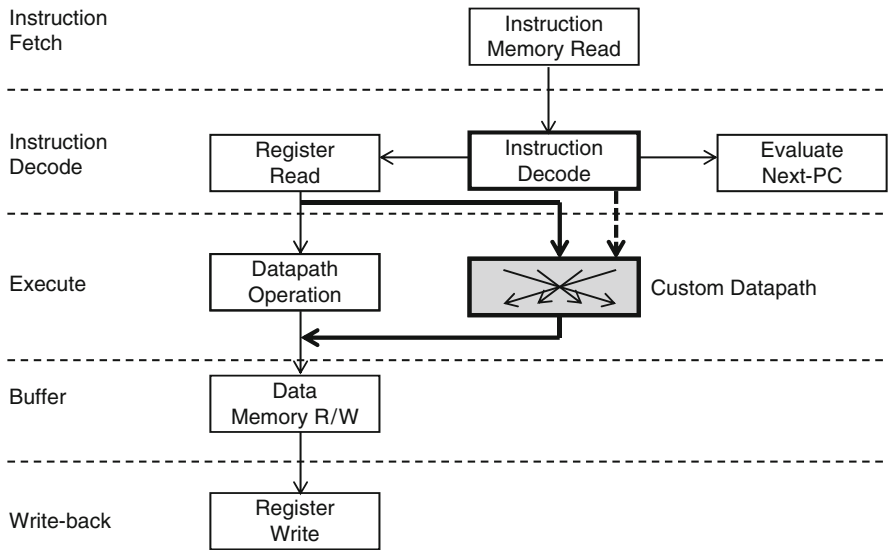
Fig. 9.22 Single-argument custom-datapath in an ASIP

```
ipblock armsfu(out d1, d2 : ns(32);
               in  q1, q2 : ns(32)) {
  iptype "armsfu2x2";
  ipparm "core = myarm";
  ipparm "device=0";
}

dp swap(in  d1, d2 : ns(32);
        out q1, q2 : ns(32)) {
  always {
    q1 = d1[ 7: 0] #
         d1[15: 8] #
         d1[23:16] #
         d1[31:24];
    q2 = d2[ 7: 0] #
         d2[15: 8] #
         d2[23:16] #
         d2[31:24];
  }
}

dp top {
  sig d1, d2, q1, q2: ns(32);
  use armsfu(d1, d2, q1, q2);
  use swap (d1, d2, q1, q2);
  use myarm;
}
```

We will now write a C program to use this custom datapath. We need to make use of a custom opcode to trigger execution of the custom datapath. The GEZEL `armsfu` interface relies on the `smullnv` opcode, which is unused by the StrongARM. As `smullnv` cannot be produced directly in C, its opcode is inserted in a regular program by making use of inline assembly. The following C snippet shows how to define an inline assembly macro, and how to call this instruction for a single-argument and a double-argument instruction.

```
#define OP2x2_1(D1, D2, S1, S2) \
  asm volatile ("smullnv %0, %1, %2, %3" : \
                "=&r"(D1), "=&r"(D2) : \
                "r"(S1), "r"(S2));

// use as a single-argument instruction
for (i=0; i<4096; i++) {
  OP2x2_1(ot[i], dummy1, in[i], dummy2);
}

// use as a dual-argument instruction
for (i=0; i<2048; i++) {
  OP2x2_1(ot[2*i], ot[2*i+1], in[2*i], in[2*i+1]);
}
```

The resulting assembly for the single-argument case now looks as follows. The loop requires 8 cycles: 6 instructions and two stalls. This is a gain of 2 clock cycles compared to the previous case. Equally important is that the program now only needs half the bus transfers, because the coprocessor is integrated *inside* of the StrongARM.

```
.L38:
    ldr     r3, [r4, lr, asl #2]   ; load in[i]
    smullnv r2, r7, r3, ip         ; perform conversion
    str     r2, [r1, lr, asl #2]   ; write ot[i]
    add     lr, lr, #1             ; next iteration
    cmp     lr, r0
    ble     .L38
```

The dual-argument design is even more efficient, because the loop management code is now shared over two endianess conversions. We have 9 cycles per loop iteration: 7 instructions and two stalls. However, each iteration of the loop performs two conversions, so that the effective cycle cost is 4.5 cycles per endianess conversion (compared to 8 cycles in the previous case).

```
.L53:
    ldr     r1, [r5], #4     ; read in[2*i], adjust pointer
    ldr     r2, [r5], #4     ; read in[2*i+1], adjust pointer
    smullnv r0, ip, r1, r2   ; perform conversion
    str     r0, [r4], #4     ; store ot[2*i], adjust pointer
    subs    lr, lr, #1       ; next iteration
    str     ip, [r4], #4     ; sotre ot[2*i+1], adjust pointer
    bne     .L53
```

Summing up, by converting an all-software design to an ASIP-type design, the cycle cost for endianess conversion on a StrongARM reduces from 13 cycles per

word to 4.5 cycles per word, an improvement of 2.9 times. Observe that in the final design, almost all execution time is spent in moving data from memory to the processor and back. This illustrates a point we made earlier: the strength of an ASIP design is also its weakness. An instruction-set architecture is convenient to build extensions, but bottlenecks in the instruction-set architecture will also be bottlenecks in the resulting hardware/software codesign.

### 9.5.3 Finding Good ASIP Instructions

How do we identify good ASIP instructions? Most likely, the application domain itself will suggest what type of primitives are most frequently needed. For example, image processing often works with 8-bit pixels. Hence, instructions that support efficient 8-bit operations will be useful for an image-processing ASIP. Another example, coding and cryptography make use of modular arithmetic. Hence, in an ASIP for coding algorithms, it makes sense to have support for modular addition and multiplication.

The study of automatic instruction definition is a research field on its own, and is of particular interest to compiler developers. We will describe two basic techniques that work directly at the level of assembly language, and that are not specific to a given application domain. The first technique is called *operator fusion*, and the second technique is called *operator compounding*.

In operator fusion, we define custom instructions as a combination of dependent operations. The dependencies are found by means of data flow analysis of the code. After data flow analysis, we can cluster assembly operations together. Each time we group two operations together, the intermediate register storage required for the individual operations disappears. Figure 9.23a illustrates operator fusion. There are obviously a few limitations to the clustering process.

- All operations that are fused are jointly executed. Hence, they must be at the same loop hierarchy level, and they must be within the same branch of an if-then-else statement. Note that it is still possible to integrate an entire if-then-else statement as a custom instruction; see Problem 9.7.
- The number of input and output operands must be limited to ensure that the register-file bandwidth stays within bounds. Indeed, as the new custom instruction executes, it will require all the input arguments to be available at the same clock cycle, and it will produce all output arguments at the same clock cycle.
- The length of the chained operations must not be too long, since this adversely affects the critical path of the processor.

Figure 9.23b shows an example of operator compounding. In this case, we are combining multiple possibly unrelated operations together, especially when they share common inputs. The operator compounds are identified using data-flow analysis of the assembly code, and they have similar limitations as fused operators.
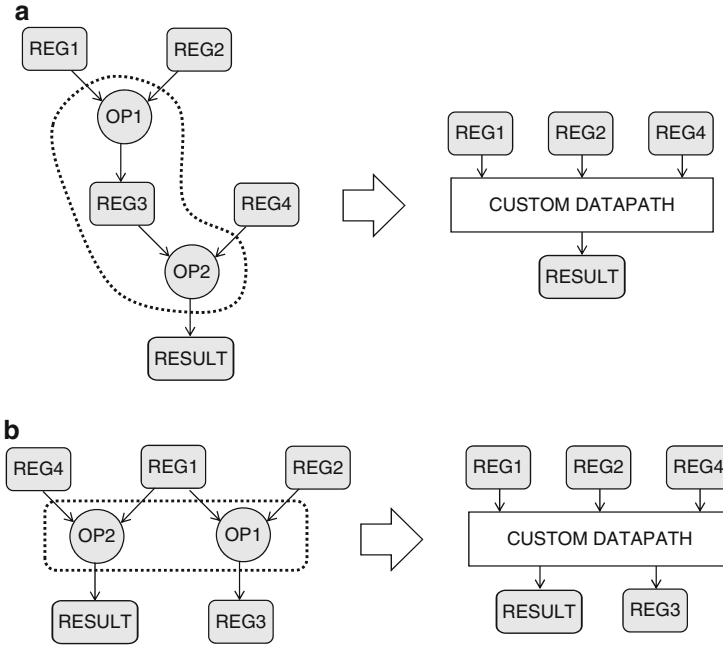
Fig. 9.23 (**a**) Operator fusion (**b**) Operator compounding

We'll now consider the endianess-conversion example once more and consider how much more it can be optimized beyond a single OP2x2 ASIP instruction for endianess conversion. In this case, we consider a complete C function as follows. Notice how this code has been optimized with incremental pointer arithmetic.

```
void byteswap(unsigned *in, unsigned *ot) {
  int i;
  int d1, d2, d3, d4;
  unsigned *q1 = in;
  unsigned *q2 = ot;
  for (i=0; i<2048; i++) {
    d1 = *(q1++);
    d2 = *(q1++);
    OP2x2_1(d3, d4, d1, d2);
    *(q2++) = d3;
    *(q2++) = d4;
  }
}
```

The assembly code for this functions looks as follows. The loop body contains nine instructions. Only a single instruction performs the actual byte swap operation! Four other instructions are related to moving data into and out of the processor (ldr, str), two instructions do loop counter management (cmp, add), one instruction

implements a conditional return (`ldmgtfd`) and one instruction is a branch. There
will be two pipeline stalls: one for the branch, and one for the second memory-load
(`ldr`).

```
byteswap:
    stmfd      sp!, {r4, r5, lr}       ; preserve registers
    mov        ip, #0                  ; init loop counter
    mov        lr, #2032               ; loop counter limit
    add        lr, lr, #15
.L76:
    ldr        r2, [r0], #4            ; load in[i]
    ldr        r3, [r0], #4            ; load in[i+1]
    smullnv    r4, r5, r2, r3          ; endianess conversion
    str        r4, [r1], #4            ; store ot[i]
    str        r5, [r1], #4            ; store ot[i+1]
    add        ip, ip, #1              ; increment loop counter
    cmp        ip, lr                  ; test
    ldmgtfd    sp!, {r4, r5, pc}       ; and conditionally return
    b          .L76
```

To optimize this assembly code with additional ASIP instructions, we can con-
struct a data-flow diagram for the assembly code, and investigate the opportunities
for operation fusion and compounding. Figure 9.24 shows the dataflow analy-
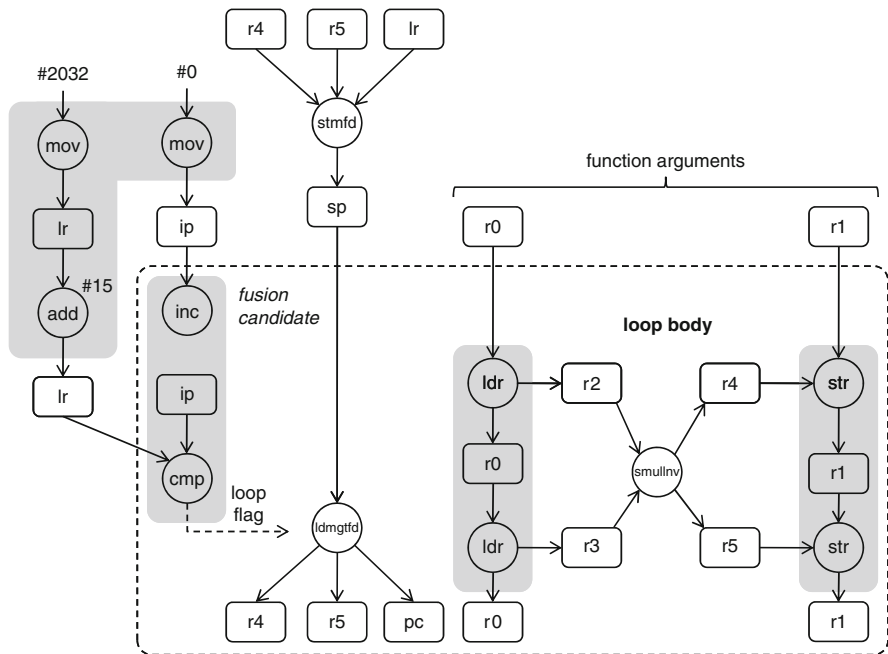sis diagram. The boxes represent registers, while the circles represent operations.

Fig. 9.24 Analysis of the `byteswap` function for ASIP instructions

A distinction is made between operations inside of the loop and those outside of it; recall that fusion and compounding only work within a single loop level. The shaded clusters are examples of possible operation fusion.

- A set of fused operations could merge the loop counter logic into two new operations. The first operation initializes two registers. The second operation increments one register, compares it to the next, and sets a loop flag as a result of it. We will call these new operations `initloop` and `incloop`, respectively.
- The second set of fused operations is trickier: they involve memory access (`str` and `ldr`), so they cannot be implemented with modifications to the execution stage of the RISC pipeline alone. In addition, because the StrongARM has only a single memory port, fusing these operations into a single operation will not provide performance improvement if there is not a similar modification to the memory architecture as well. Thus, we cannot define new instructions for these fusion candidates. However, we may still reduce the footprint of the loop body by collapsing the store and load instructions in store-multiple and load-multiple instructions.

The resulting assembly code after these transformations would therefore look as follows. The loop body contains six instructions, but will have three stalls (one for `ldm`, one for the data dependency from `ldm` to `smullnv`, and one for the branch). A loop round-trip now costs 9 cycles, or 4.5 cycles per word. This is as fast as we found before, but in this version of the code, function call overhead is included.

```
byteswap:
        stmfd     sp!, {r4, r5, lr}   ; preserve registers
        initloop  ip, lr, #0, #2047   ; loop-counter instruction 1
.L76:
        ldm       r0!, {r2, r3}       ; load in[i], in[i+1]
        smullnv   r4, r5, r2, r3      ; endianess conversion
        stm       r1!, {r4, r5}       ; store ot[i], ot[i+1]
        incloop   ip, lr              ; loop counter instruction 2
        ldmgtfd   sp!, {r4, r5, pc}   ; and conditionally return
        b         .L76
```

This concludes our discussion of the last hardware/software interface, the custom instruction. The design of customized processors is a hot research topic, and optimizations have been investigated that go far beyond the examples we discussed earlier. For the hardware/software codesigner, it's important to understand that ASIP design takes a top–down view on the codesign problem: one starts with a C program, and next investigates how to improve its performance. In a classic coprocessor design, the view is bottom-up: one starts with a given kernel in hardware, and next investigates how to integrate it efficiently into a system. Both approaches are viable, and in both cases, a codesigner has to think about interfacing hardware and software efficiently.

9.6 Summary

Hardware/Software interfaces are at the core of the hardware/software codesign problem. While the previous chapters have discussed the means to build communication channels between hardware and software, this chapter has explained how to do it. The starting point of all hardware/software interfaces is a means to synchronize two parallel behaviors, one in software and one in hardware. A semaphore implements an abstract solution for this problem. Master handshakes and slave handshakes are a method for synchronization that works in hardware as well as in software.

We made a distinction between three classes of interfaces, each with a different integration within the System on Chip architecture. A memory-mapped interface reuses the addressing mechanism of an on-chip bus to reserve a few slots in the address space for hardware/software communication. Single memory locations can be implemented using memory-mapped registers. A range of memory locations can be implemented using a shared memory. Specialized communication mechanisms, such as FIFO queues, further improve the characteristics of the hardware/software communication channel.

The coprocessor interface is a second type of hardware/software interface. It requires a dedicated port on a microprocessor, as well as a few predefined instructions to move data through that port. We discussed two examples of this interface, including the Fast Simplex Link interface, and a floating-point coprocessor interface for the LEON-3 processor.

The final hardware/software interface is the custom-instruction, created by modifying the instruction-set architecture of a microprocessor. This interface requires a rather detailed understanding of microprocessor architecture, and is often supported with a dedicated toolchain.

As a hardware/software codesigner, it is useful to spend time thinking about the breadth of this design space, which is enormous. Probably, the most important point is to realize that there is no single silver bullet that can capture all the variations of interconnections for hardware and software. There are many trade-offs to make for each variation, and different solutions can tackle different bottlenecks: computational power, data bandwidth, power consumption, design cost, and so on. Also, keep in mind that no system can be free of bottlenecks: the objective of hardware/software codesign is not to remove bottlenecks, but rather to locate and understand them. In the next chapter, we will focus on the hardware-side of custom-hardware modules, and describe how hardware can be controlled from within software through a hardware/software interface.

9.7 Further Reading

The theory of synchronization is typically discussed in depth in textbooks on Parallel or Concurrent Programming, such as Taubenfeld (2006) or Moderchai (2006). The original documents by Dijkstra are available from the E. W. Dijkstra Archive

at the University of Texas Dijkstra (2009). They're in Dutch! Een uitstekende gelegenheid dus, om Nederlands te leren.

Early research in hardware/software codesign suggested that much of the hardware/software interface problem can be automated. Chapter 4 in Micheli et al. (2001) describes some of the work in this area. To date however, no standards to create hardware/software interfaces exist, and the design of such interfaces largely remains an ad-hoc process.

Yet, design support is critical to ensure error-free design. Designers often build virtual platforms of a chip during the implementation. A virtual platform is a complete simulation of the entire chip, emphasizing a detailed representation of the hardware/software interactions.

Memory-mapped interfaces are ubiquitous in the design of System-on-Chip architectures. One can consult the datasheet of a typical microcontroller and find that all peripherals are programmed or configured using memory-mapped input/output. For example, check the datasheet of Atmel AT91SAM7L128, an ARM-based microcontroller with numerous on-chip peripherals Atmel (2008).

In contrast to general hardware/software interfaces, the literature on custom processor design is rich and detailed. In fact, the ASIP approach is one of the most succesful models for hardware/software codesign when practical implementations are considered. A comprehensive treatment of the ASIP design process is provided by Rowen in Rowen (2004). Leupers and Ienne discuss customized processor architectures in Leupers and Ienne (2006). There are numerous publications on design applications based on ASIP, and as many conferences that cover them (a major event is Embedded Systems Week, grouping 3 conferences together on compiler design, on system design, and on embedded software implementation).

9.8 Problems

9.1. Consider the two-way handshake in Fig. 9.25. A sender synchronizes with a receiver and transmits a sequence of data tokens through a data register.

(a) Describe under what conditions register `r1` can be removed without hurting the integrity of the communication. Assume that, after taking `r1` away, the `req` input of the receiver is tied to logic-1.
(b) Describe under what conditions register `r3` can be removed without hurting the integrity of the communication. Assume that, after taking `r3` away, the `req` input of the sender is tied to logic-1.
(c) Assume that you would substitute register `r1` by two registers in series, so that the entire transition from sender-ack to receiver-req now takes two clock cycles instead of one. Describe the effect of this change on the throughput of the communication, and describe the effect of this change on the latency of the communcation.
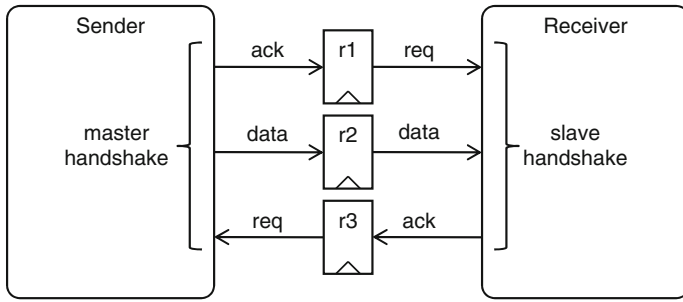
Fig. 9.25 Two-way handshake for Problem 9.1 and Problem 9.5

9.2. A C function has 10 inputs and 10 outputs, all of them integers. The function takes 1,000 cycles to execute in software. You need to evaluate if it makes sense to build a coprocessor for this function. Assume that the function takes K cycles to execute in hardware, and that you need Q cycles to transfer a word between the software and the coprocessor over a system bus. Draw a chart that plots Q in terms of K, and indicate what regions in this chart justify a coprocessor.

9.3. A C function requires 8 programmable constants that change very infrequently, and 1 data input that changes upon each call. Design a memory-mapped interface that minimizes the amount of memory locations required, while at the same time introducing minimal impact on the runtime performance of the design. Be precise: show a CPU bus on one side, and 8 + 1 registers on the other side, with your memory-mapped interface design in between. Assume the interface is mapped starting at address 0x100.

9.4. Build a memory-mapped register for the address bus described in Table 8.1. Evaluate the complexity of two different designs. What is the general recommendation for the design of memory-mapped address decoders you can make?

(a) A decoder that maps the register to any address of the range `0x3F000000` - `0x3F00FFFF`.
(b) A decoder that maps the register to the single address `0x3F000000`.

9.5. Consider the two-way handshake in Fig. 9.25. Implement this two-way handshake by developing an FSMD for the sender and the receiver. Next, optimize the two-way handshake so that *two* tokens have been transferred each time `req` and `ack` have make a complete handshake and returned to the logic-0 state.

9.6. Modify the design of Fig. 9.11 so that all positions of the FIFO are used before the `full` flag is set high.

9.7. Consider the following C program. You have to optimize it using custom instructions.

(a) Study the following C program and the corresponding assembly code out of it.

```
int findmax(unsigned int data[1024]) {
  unsigned int max = 0;
  int i;
  for (i=0; i<1024; i++)
    if (max < data[i])
      data[i] = max;
  return max;
}

findmax:
    mov         r2, #0
    mov         r1, #1020
    mov         ip, r2
    add         r1, r1, #3
.L7:
    ldr         r3, [r0, r2, asl #2]
    cmp         ip, r3
    strcc       ip, [r0, r2, asl #2]
    add         r2, r2, #1
    cmp         r2, r1
    movgt       r0, #0
    movgt       pc, lr
    b           .L7
```

(b) Define a custom instruction max rx, ry, which compares ry to the current value of rx and replaces that value if ry is bigger than rx. Assume that rx and ry are unsigned, 32-bit values. Show how the assembly code must be modified.
(c) Design a GEZEL datapath for this custom instruction, following the example in Sect. 9.5.2.

9.8. The following C function is compiled to a Microblaze processor and results in assembly code as shown. The input arguments of the assembly code are r5 and r6; the return argument is r3; the return instruction was left out of the assembly.

```
int absmax(int v, int w) {
  return (v * 6 + w * 4);
}

// input arg: r5, r6
// return arg: r3
addk    r3,r5,r5    // op1
addk    r3,r3,r5    // op2
addk    r3,r3,r3    // op3
addk    r6,r6,r6    // op4
addk    r6,r6,r6    // op5
addk    r3,r3,r6    // op7
```

(a) Perform dataflow analysis on the assembly code and draw the data dependency diagram later. Use rectangles to indicate registers and circles to indicate operations. Label the operations "op1" to "op7."
(b) When you have drawn the dataflow dependency diagram, define several ASIP candidate instructions (at least 2) using operation fusion. Indicate clearly which operations you could merge into ASIP instructions.

9.9. An ASIP processor performs operations on a stream samples, fed into the ASIP at a fixed rate. The samples are processed using an algorithm A. Which of the following optimizations will reduce the energy consumption E needed to process a single sample?

(a) Rewrite the algorithm A so that it requires fewer MOPS from the processor (MOPS = Million Operations per Second). Does this reduce the energy consumption E?

(b) Add a custom instruction B that will make algorithm A complete in only half the clock cycles. You can assume the power consumed by added hardware is negligible. Does this reduce the energy consumption E?

(c) Increase the clock frequency of the ASIP. Does this reduce the energy consumption E?

(d) Lower the voltage of the ASIP. Does this reduce the energy consumption E?

Chapter 10
Coprocessor Control Shell Design

Abstract This chapter discusses the design practice of attaching a custom hardware module to a hardware/software interface, an activity referred to as *control shell design*. It involves the encapsulation of a custom hardware module on a standard hardware/software interface, and the development of a software driver to control the custom hardware module through this hardware/software interface. There are two orthogonal aspects to control shell design. The first, *data design*, defines how to transfer data from software to custom hardware and back. The second, *control design*, defines how to implement a software control strategy for the custom hardware. The outcome of data design and control design define the *programmer's model*, the abstract software view of the hardware module. This chapter will address each of these aspects, and discuss an example based on a control shell for an encryption coprocessor for the Advanced Encryption Standard (AES).

10.1 The Coprocessor Control Shell

A control shell connects a custom hardware module to a coprocessor bus or an on-chip bus. The control shell steers the input and output ports of the custom hardware module. This may affect many different activities of the custom hardware module, including data transfer as well as control. Figure 10.1 shows the location of the control shell in the overall integration of a microprocessor with a custom-hardware module.

10.1.1 Functions of the Coprocessor Control Shell

The design of the control shell is a classic hardware/software codesign problem, that matches the flexibility of custom-hardware design to the realities of hardware/software interfaces. The services that are implemented by the control shell include the following.

Fig. 10.1 The control shell
maps a custom-hardware
module to a
hardware–software interface
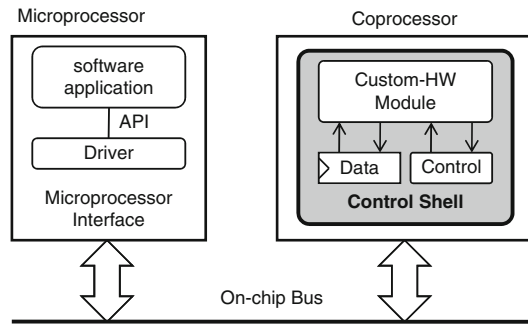
- **Data Transfer:** The control shell implements read/write transactions on the on-chip bus, using a master-protocol or a slave-protocol. In other cases, the control shell implements handshake sequences for a coprocessor bus. A control shell can be optimized for high-throughput and bursty data transfers, for example with a dedicated Direct Memory Address Controller.
- **Wordlength Conversion:** The control shell converts data operands of the custom-hardware module, which can be arbitrary in size and number, into data structures suitable for on-chip bus communication. For example, in a 32-bit system, software can deliver only 32-bits of data at a time, while the custom hardware module may need a much wider 1024-bit input bus. In that case, the control shell needs to support a conversion of a single 1024-bit operand into an array of 32-bit words. Furthermore, the control shell takes care of the bit- and byte-level organization of the data in case conversions between software and hardware are needed.
- **Operand Storage:** The control shell provides local and/or temporary storage for arguments and parameters for the custom-hardware component. Besides arguments and parameters, the control shell can also include local buffering to support the on-chip interface. The distinction between arguments and parameters is in fact very important. Arguments are updated *every time* the custom-hardware module executes. Parameters, on the other hand, may be updated only infrequently. Hence, to minimize the hardware–software communication bandwidth, parameters are transmitted only once from software to hardware, and then held in a local memory in the control shell.
- **Instruction Set:** The control shell defines the programmer's model, the software-view of a custom-hardware component in terms of instructions and data structures. The design of instruction-sets for custom hardware modules is a particularly interesting and important problem. A carefully designed custom instruction-set can make the difference between an efficient coprocessor, and a confusing blob of logic.
- **Local Control:** The control shell implements local control interactions with the custom hardware component, such as sequencing a series of micro-operations in response to a single software command.

10.1.2 Layout of the Coprocessor Control Shell

Figure 10.2 illustrates the layout of a generic coprocessor control shell, which connects a custom hardware module and which attaches it to an on-chip bus interface or coprocessor interface. As the control shell itself is a user-defined hardware component, it can have an arbitrary architecture. The following components are commonly found in a control shell.

- A data input buffer for *Argument Storage*.
- A data output buffer for *Result Storage*.
- A *Command Interpreter* to control the data buffers and the custom hardware module based on commands from software.

The control shell has several *ports*, addressable inputs/outputs of the coprocessor. For example, an on-chip bus interface uses an address decoder, while a coprocessor interface may have dedicated ports. From the perspective of the custom hardware module, it is common to partition the collection of ports into data input/output ports and control/status ports.

The separation of control and data is an important aspect in the design of coprocessors. Indeed, in a software program on a microprocessor, control and data are tightly connected through the instruction-set of the microprocessor. In custom-hardware however, the granularity of interaction between data and control is chosen by the designer. Observe that in Fig. 10.2, control flows vertically through the hardware while data runs horizontally from left to right.

Despite the relatively simple organization of the control shell, the design space of the data buffers and the command interpreter is rich and deep. In the following, we will discuss mechanisms to help us design the control shell efficiently. We will differentiate between *data-design* and *control-design*. Data-design implements data dependencies between software and hardware. Control-design implements control dependencies between software and hardware. First, we investigate the performance limits of a control shell.
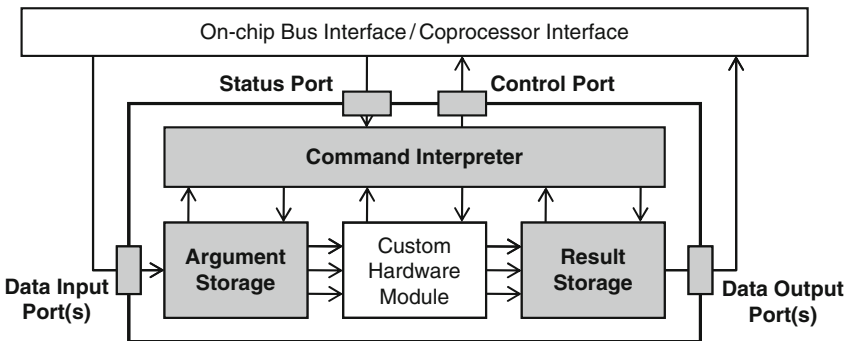

Fig. 10.2 Layout of a coprocessor control shell

10.1.3 Communication-Constrained vs. Computation-Constrained Coprocessors

We start by evaluating the performance of the control shell shown in Fig. 10.3. This shell uses addressable ports to steer 3 input ports and 1 output port of a custom hardware module. If we add up the wordlengths of the actual input ports and output ports of the custom hardware module, we find it requires transfer of $128 + 128 + 32 + 32 = 320$ bits per execution. This assumes that all of the data ports carry an operand, not a constant parameter. We will also assume that the custom hardware module takes 5 cycles to compute a result. Hence, when this module is connected to software, and we wish to run the module at full performance, we will need to offer a data bandwidth of $320/5 = 64$ bits per cycle. This data bandwidth needs to be delivered through a hardware/software interface. Let's assume the interface provides 128 bits per transfer, and that each transfer will take 1 clock cycle. In that case, the available bandwidth is $128/1 = 128$ bits per cycle. Clearly, the interface can provide data two times faster than required to keep the custom hardware fully utilized. This is an example of a *computation-constrained* system: the bottleneck is in the hardware computation unit. Now let's assume that we have a pipelined custom hardware module, and that the module accepts new operands every clock cycle. In that case, the module bandwidth increases to $320/1 = 320$ bits per clock cycle. The interface cannot keep up with the bandwidth required to keep the coprocessor fully utilized. The system is now *communication-constrained*.

Figure 10.4 summarizes these observations. The distinction between a communication-constrained system and a computation-constrained system is important, because it tells the designer where to put design effort. In a communication-constrained system, it does not make sense to implement a more powerful coprocessor, because it will remain under-utilized. Conversely, in a computation-constrained system, we do not need to look for a faster hardware/software interface.
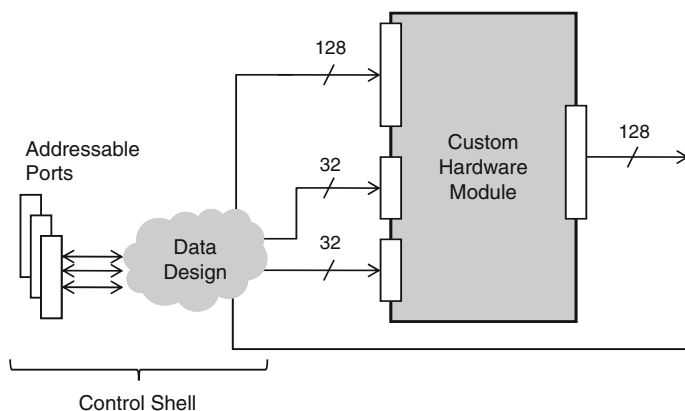

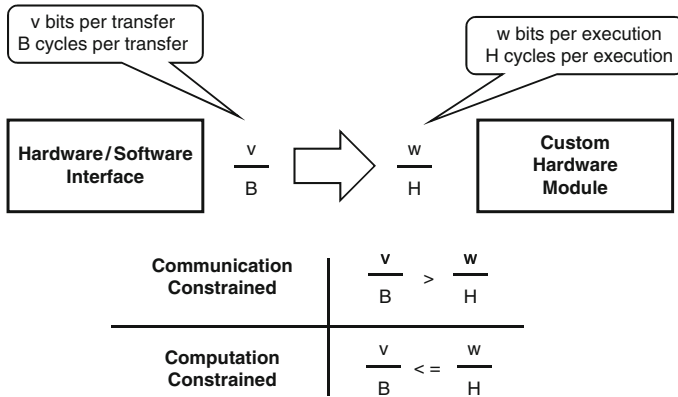**Fig. 10.3** Data design in a control shell

Fig. 10.4 Communication-constrained system vs computation-constrained system

Architecture	HSF
Systolic array processor	1
Bit-parallel processor	1–10
Bit-Serial processor	10–100
Microcoded processor	>100

Table 10.1 Hardware sharing factor

An additional insight can be gained from the number of clock cycles needed per execution of the custom hardware module. This number is called the *hardware sharing factor* or HSF. The HSF is defined as the number of clock cycles that are available in between each input/output event. For example, an HSF of 10 would mean that a given hardware architecture has a cycle budget of 10 clock cycles between successive input/output events. Thus, if this architecture would contain two multiplier operators then these 10 clock cycles are adequate to support up to 20 multiplications. The HSF is useful in back-of-the-envelope calculations to evaluate if a given architecture is powerful enough to sustain a computational requirement. There is a strong correlation between the internal architecture of a hardware module and its HSF. This is ilustrated in Table 10.1.

- A systolic-array processor is a multidimensional arrangement of computation units (datapaths or dataflow-actor-like processors) that operate on one or more parallel streams of data items.
- A bit-parallel processor is a processor with bit-parallel operators such as adders and multipliers that operates under control of a simple engine (such as a FSMD or a microprogrammed controller).
- A bit-serial processor is a processor with bit-serial operators, i.e., operators that compute on a single bit at a time, under control of a simple engine (such as a FSMD or a microprogrammed controller).
- A microcoded processor is a processor with an instruction-fetch, similar to a general purpose processor.

The correlation between HSF and architecture is important: it helps a designer selecting the right architecture style at the start of a design, when only the HSF is known.

10.2   Data Design

Data Design is the implementation of a mapping between the control-shell ports and the custom-hardware ports. Typically, this includes the introduction of buffers and registers, as well as the creation of an address map.

10.2.1   Flexible Addressing Mechanisms

A data port on a coprocessor has three characteristics. The port has a *wordlength*, a *direction* and an *update rate*. For example, the wordlength and direction could be a 32-bit input. The update rate expresses how frequently a port changes value. The two extremes are a parameter, which needs to be set only a single time, and a function argument, which changes value upon each execution of the hardware module.

When these three characteristics (wordlength, direction, and update rate) are known, we can map the actual ports of the hardware module to the ports of the control shell. For example, consider a coprocessor for the greatest-common-divisor function, gcd. The high-level specification of this function would be:

```
int gcd(int m, int n);
```

The hardware module equivalent of gcd has two input ports m and n, which are 32-bit wide. The module also has a single output port of 32-bit. These three ports are the actual ports of the hardware module. When we implement this module as a memory-mapped coprocessor, the ports of the control shell will be implemented as memory-mapped registers. Therefore, we need to define a connection from the three ports of the hardware module to the memory-mapped registers.

This strategy makes each port of the hardware module independently addressable from software. However, it may not always be possible to allocate an arbitrary number of data ports in the control shell. In that case, we need to *multiplex* the hardware-module ports over the control shell ports.

10.2.2   Multiplexing and Masking

There are several occasions when the ports of the hardware module need to be multiplexed over the ports of the control shell.

- There may be insufficient control-shell ports available to implement a one-to-one mapping between hardware-module ports and control-shell ports.
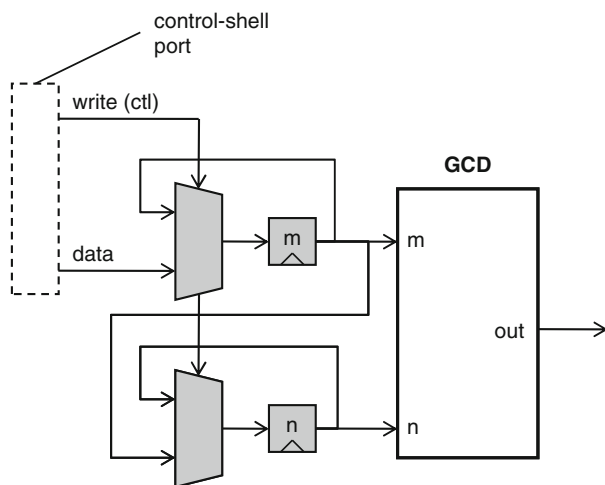
Fig. 10.5 Time-multiplexing of two hardware-module ports over a single control-shell port

- Some hardware-module ports need to be programmed only once (or very infrequently), so that it is inefficient to allocate a separate control-shell port for these ports.

Multiplexing will increase the control complexity of the control shell slightly. In addition, uncareful multiplexing will reduce the available input/output bandwidth of the hardware module. Thus, there is a risk that the module becomes communication-constrained because of the multiplexing process.

Multiplexing can be implemented in different ways. The first is to use *time-multiplexing* of the hardware module ports. The second is to introduce an *index register* in the control shell. Figure 10.5 shows an example of a time-multiplexed port for the GCD coprocessor. In this case, the arguments need to be provided by writing the value of m and n sequentially to the control-shell port.

The index-register technique works as follows. Several ports (say N) on the hardware module are mapped into two ports on the control shell. One port on the control shell is a data port of sufficient width to hold any single hardware module port. The second port is an index port and has width $log_2 N$ bits. The index port controls the mapping of the data port of the control shell to one of the ports on the hardware module. Figure 10.6 shows an example of the index-register technique to merge 8 outputs to a single data output. The index register technique is more flexible than time-multiplexing, because the interface can freely choose the readout order of the hardware-module output ports. At the same time, it also requires double the amount of interface operations. Hence, multiplexing with index-registers is most useful for ports that update very infrequently, such as parameters.

Multiplexing is also useful to handle operands with very long wordlengths. For example, if the hardware module uses 128-bit operands, while the control-shell ports are only 32-bit, then the operands can be provided one word at a time by means of time multiplexing. We will discuss an example further in this chapter.
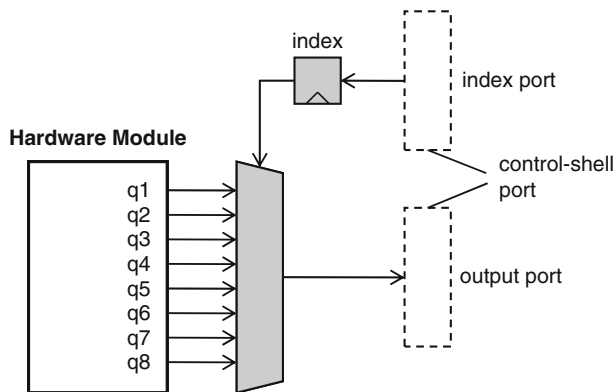
Fig. 10.6 Index-register to select one of eight output ports

Finally, we also mention a technique to work with very short operands, such as single-bit arguments. In this case, it is expensive to allocate a single control-shell port for each single-bit hardware-module port. Instead, a single control-shell port should be shared over several hardware-module ports. This may lead to unwanted bits when reading the control-shell port, and it may also result in unwanted updates when writing the control-shell port. To solve this, a *mask register* can be introduced. A mask register indicates which bits of a control-shell port should be taken into account when updating the hardware-module ports. The updated value is obtained by simple bit-masking of the previous value on the hardware ports with the new value of the control shell port.

```
new_hardware_port = (old_hardware_port & ~mask_value) |
                    (control_shell_value & mask_value);
```

10.3 Control Design

Control design in a coprocessor is the collection of activities to generate control signals and to capture status signals. The result of control design is a set of *commands* or instructions that can be executed by the coprocessor. These commands are custom-tailored for the design.

Figure 10.7 shows a generic architecture to control a custom hardware module through software. It includes a *command interpreter* which accepts commands from software and which returns status information. The command interpreter is the top-level controller in the coprocessor, and it communicates directly with software. We also make a distinction between a *command* and a *configuration*. A command is a one-time control operation. A configuration is a value which will affect the execution of the coprocessor over an extended period of time, possibly over multiple commands.

Fig. 10.7 Command design of a control shell
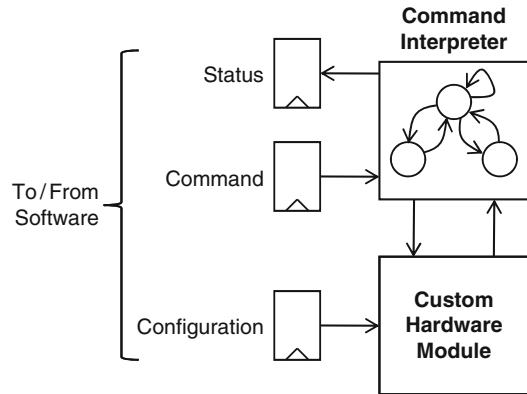
In the following sections, we discuss several architectural techniques that can be used to optimize the performance of the coprocessor.

10.3.1 Hierarchical Control

Figure 10.8 shows the architecture of a coprocessor that can achieve communication/computation overlap. The coprocessor has a hierarchy of controllers, which allow independent control of the input buffering, the computations, and output buffering. The command interpreter analyzes each command from software and splits it up into commands for the lower-level FSM. In the simplest form, these subcommands are simple start/done handshakes. Thus, for each command of software, the command interpreter can start a combination of lower-level FSM. Often, a single level of command decoding is insufficient. For example, we may want to use a coprocessor which has an addressable register set in the input or output buffer. In that case, we can embed the register address into the command coming from software. To implement these more complicated forms of subcontrol, the start/done handshakes need to be replaced with more complex command/status pairs.

A control hierarchy simplifies the design of control, as is shown in Fig. 10.9. The command interpreter can easily adapt to the individual schedules from the input FSM, compute FSM and output FSM. On the other hand, the method of using start/-done pulses is inconvenient when working with pipelined submodules, since a done pulse only appears after the pipeline latency of the submodule. This will require a modification to the start/done scheme, which will be discussed later.

First, we examine how a hierarchical controller as illustrated in Fig. 10.8 can help in achieving computation/communication overlap. The basic principle, of course, is well known: we need to pipeline the input/compute/output operations within the coprocessor. Table 10.2 illustrates a set of five software commands to achieve pipelined execution within the coprocessor. This scheme is called *block-level pipelining*. The commands obtain precise pipeline startup and shutdown. The first three commands
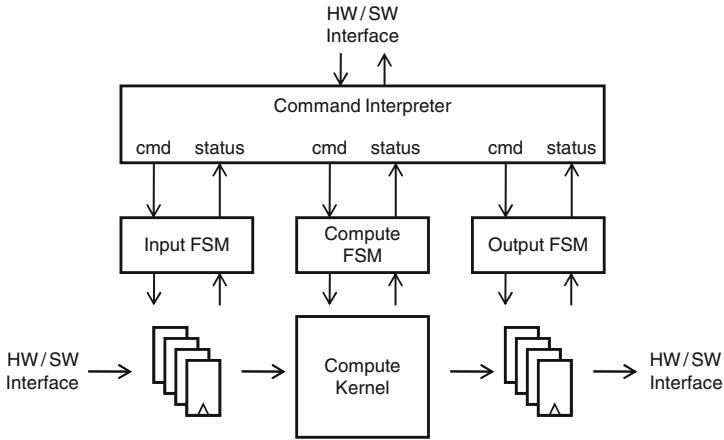
Fig. 10.8 Hierarchical control in a coprocessor
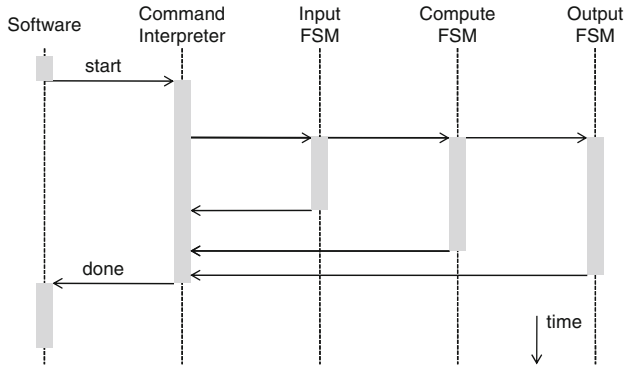
Fig. 10.9 Execution overlap using hierarchical control

Table 10.2 Command set to control block-level pipelining

Command	Input FSM	Compute FSM	Output FSM
pipe_load1	start		
pipe_load2	start	start	
pipe_continue	start	start	start
pipe_flush1		start	start
pipe_flush2			start

(pipe_load1, pipe_load2, pipe_continue) require the software to send an argument to the coprocessor. The last three commands (pipe_continue, pipe_flush1, pipe_flush2) require the software to retrieve a result from the coprocessor. Once the pipeline is filled up through the sequence of pipe_load1 and pipe_load2, the software can repeat the command pipe_continue as often as needed.

10.3.2 Control of Internal Pipelining

When a custom hardware module has internal pipeline stages, hierarchical control becomes more intricate. We need to address two issues during the design of a control shell. First, we need to find a way to generate control signals for the pipeline stages. Next, we need to define a proper mechanism to interface these control signals with the higher layers of control. Indeed, a simple start/done handshake is insufficient for a pipeline, because it does not reflect the pipeline effect. In this section we will address both of these aspects.

Figure 10.10 introduces some terminology on pipeline architectures. A pipeline consists of a number of pipeline stages separated by pipeline registers. The latency of a pipeline is the number of clock cycles it takes for an operand to move from the input of the pipeline to the output. The throughput of a pipeline measures the number of results produced per clock cycle. If a pipeline accepts a new operand each clock cycle, its throughput is one (per cycle). If, on the other hand, a pipeline accepts a new operand every N cycles, its throughput is $1/N$. In a *linear pipeline* architecture, there are no feedback connections. For such a pipeline, the latency equals the number of pipeline stages, and the throughput equals 1. In a *nonlinear pipeline* architecture, there are feedback connections. This happens when certain stages of a pipeline are reused more than a single time for each data token that enters the pipeline. In a nonlinear pipeline, the latency can be higher than the number of pipeline stages, and the throughput can be lower than 1.

In any pipelined coprocessor, the pipeline control signals are eventually be under the control of a higher-level controller. Pipeline control signals will be required in two cases.

- In case a pipeline needs to perform more than a single operation, there needs to be a way to send control information into the pipeline. This control information will determine the operation of individual pipeline stages.
- In a nonlinear pipeline architecture, multiplexers may be needed between the pipeline stages in order to feed operands from multiple locations within the pipeline. These multiplexers need additional control signals.
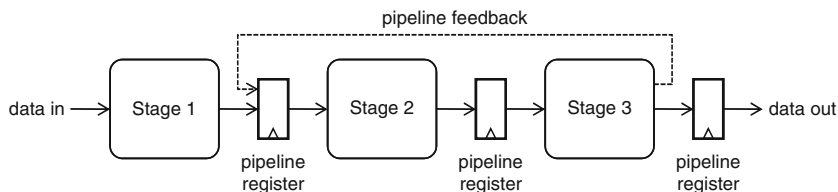
Fig. 10.10 Pipeline terminology

10.3.2.1   Control of Linear Pipelines

We first discuss the control of multifunction pipelines, which use one of the following two methods. The first is called *data-stationary* control, while the second is called *time-stationary* control. Figure 10.11 illustrates the differences between these two schemes.

- In a data-stationary scheme, control signals will travel along with the data through the pipeline. At each pipeline stage, the control word is decoded and transformed into appropriate control signals for that stage.
- In a time-stationary scheme, a single control word will control the entire pipeline for a single clock cycle. Because the pipeline contains fragments of different data items, each in a different stage of processing, a time-stationary control scheme will specify the operations to be performed on several data elements at the same time.

From a programmer's perspective, a data-stationary approach is more convenient because it hides the underlying pipeline structure in the program. A RISC instruction-set, for example, uses data-stationary encoding. On the other hand, time-stationary control makes the underlying machine structure explicitly visible in the control signals. Time-stationary control is therefore suitable for tasks that require access to the entire pipeline at once, such as exception handling. In addition, nonlinear pipeline architectures are easier to control with a time-stationary

Data-stationary Control

Time-stationary Control

Fig. 10.11 Data-stationary and time-stationary pipeline control

approach then with a data-stationary approach. Indeed, generating the control signals for multiplexers between pipeline stages requires an overview of the entire pipeline.

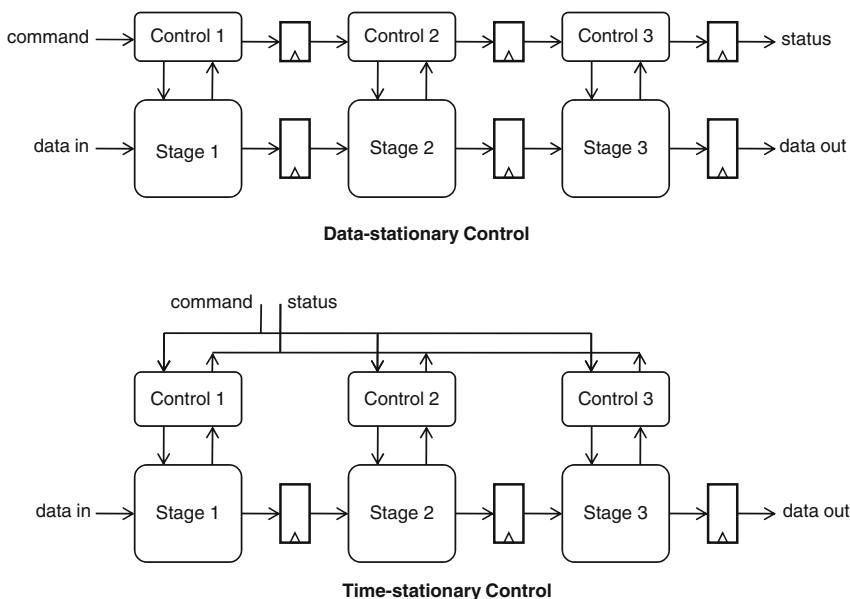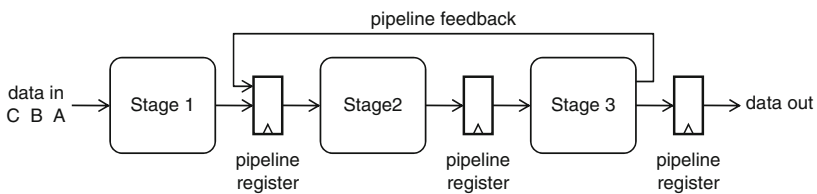### 10.3.2.2 Control of Nonlinear Pipelines

When a pipeline has feedback connections, the pipeline stages are reused multiple times per data item that enters the pipeline. As a result, the throughput of a pipeline can no longer be one.

Figure 10.12 illustrates the operation of a nonlinear pipeline structure with three stages. Each data item that enters the pipeline will use stage 2 and stage 3 two times. When a new data item enters the pipeline, it will occupy stage 1 in the first cycle, stage 2 in the second, and stage 3 in the third cycle. The data item is then routed back to stage 2 for the fourth cycle of processing, and into stage 3 for the fifth and final cycle of processing. We can thus conclude that this nonlinear, three-stage pipeline has a latency of 5. The diagram below Fig. 10.12 is a *reservation table*, a systematic representation of data items as they flow through the pipeline, with stages corresponding to rows and clock cycles corresponding to columns. The table demonstrates the pipeline processing of three data items A, B, and C. From the diagram, we can see that data items A and B are processed in subsequent clock cycles. However, item C cannot immediately follow item B: the pipeline is busy and will occupy stage 2 and stage 3. Hence, item C will need to wait. This situation is called a pipeline *conflict*. In cycle 5, the pipeline is available again and item C can start. We conclude that the pipeline is able to process two elements (A and B) in four clock cycles. Therefore, the throughput of this pipeline is 0.5.

The operation of nonlinear pipelines has been studied in detail, for example in the seminal work of Peter Kogge, but this material is beyond the scope of this chapter. Instead, we will consider the impact of pipelining on the generation of control handshake signals.

Cycle	1	2	3	4	5	6	7	8	9
Stage1	A	B			C				
Stage 2		A	B	A	B	C		C	
Stage 3			A	B	A	B	C		C

Fig. 10.12 A reservation table for a nonlinear pipeline

10.3.2.3 Control Handshakes for Pipelines

Earlier in this section we used a simple start/done handshake to implement hiearchical control for an iterated component. How do these handshake signals have to be modified for pipeline architectures?

In an iterated structure, a single done signal is sufficient to mark two distinguished but coinciding events. The first event is when a result is available at the output. The second event is when a new argument can be accepted at the input. In a pipeline structure however, input and output activities do not have to coincide. Indeed, in a pipeline, the latency does not have to be the reciprocal of the throughput. The case of a linear pipeline is still easy: the latency is equal to the number of pipeline stages, and the throughput is equal to one. The case of a nonlinear pipeline is more complex, and the latency as well as the throughput can both be different from one.

To distinguish input events from output events, we will use *two* handshake-acknowledge signals. The first one, done, indicates when the pipeline produces valid data. The second one, allow, indicates when the input is able to accept new arguments.

Figure 10.13 illustrates the relation of the handshake signals to the operation of the pipeline. The left side of the figure illustrates the interface to the pipeline, while the right side shows the values for start, done, and allow over several clock cycles. The beginning and end of a pipeline instruction are marked through start and done. The allow signal indicates if a new instruction can be started at the *next* clock cycle. If allow is zero, this means that starting an instruction will cause a pipeline conflict. You can observe that done is a delayed version of start, with the delay equal to the pipeline latency. The format of the allow signal is more complex, because it depends on the exact pattern of pipeline use. For example, allow must be zero in cycle 3 and cycle 4 because the second pipeline stage is occupied by instruction A and B.

Nevertheless the allow signal is easy to generate, as demonstrated in Fig. 10.14. The allow signal indicates when the pipeline is occupied and cannot accept a new instruction. For the reservation table shown in Fig. 10.14, this happens two clock cycles after a pipeline instruction starts. These two clock cycles are a

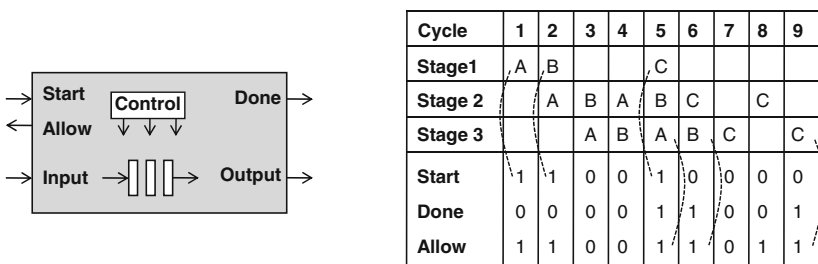| Cycle	1	2	3	4	5	6	7	8	9
Stage1	A	B			C				
Stage 2		A	B	A	B	C		C	
Stage 3			A	B	A	B	C		C
Start	1	1	0	0	1	0	0	0	0
Done	0	0	0	0	1	1	0	0	1
Allow	1	1	0	0	1	1	0	1	1

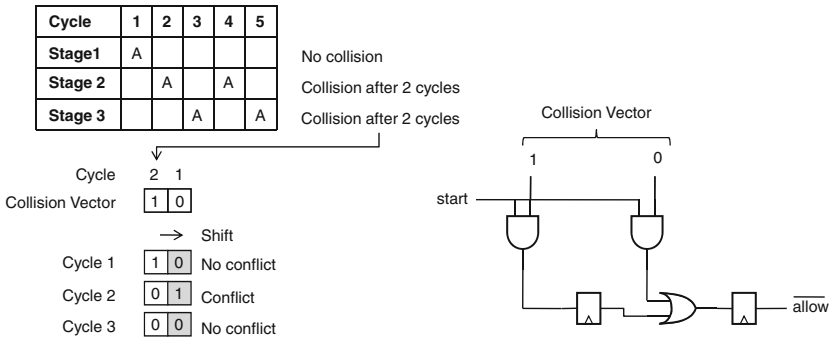**Fig. 10.13** Control handshakes for a nonlinear pipeline

Fig. 10.14 Generation of the `done` signal

forbidden latency. A *collision vector* is a bit-vector where forbidden latencies are marked by means of a bit. The index of this bit corresponds to the forbidden latency. For the reservation table shown, the collision vector equals `10`, since the only forbidden latency is two clock cycles. The `allow` signal can now be generated using the collision vector and a shift register, as shown on the right of Fig. 10.14. Each time a new instance of a pipeline instruction starts, a copy of the collision vector is added to the shift register. The last bit of this shift register indicates if the current clock cycle coincides with a forbidden latency. Hence, this bit is the inverse of the `allow` bit.

10.4 Programmer's Model = Control Design + Data Design

The previous two sections highlighted two aspects that affect control shell design. *Data design* is concerned with moving data from software to the encapsulated hardware module and back. *Control design* is concerned with generating control signals for the encapsulated hardware module.

In this section, we consider the impact of these design decisions on the software driver. The software view of a hardware module is defined as the *programmer's model*. This includes a collection of the memory areas used by the custom hardware module, and a definition of the commands (or instructions) understood by the module.

10.4.1 Address Map

The *address map* reflects the organization of software-readable and software-writable storage elements of the hardware module, as seen from software. The address map is part of the design of a memory-mapped coprocessor, and its

design should consider the viewpoint of the software designer rather than the hardware designer. Here are some of the considerations that affect the design of the address map.

- To a software designer, *read* and *write* operations commonly refer to the *same* memory location. For a hardware designer, it is easy to route read and write operations to different registers, since read strobes and write strobes are available on a bus as separate registers. This practice should be avoided because it goes against the expectations of the software designer. A given memory-mapped address should always affect the same hardware registers.
- In the same spirit, a hardware designer can create memory-mapped registers that are read-only, write-only or read-write registers. By default all memory-mapped registers should be read/write. This matches the expectations of the software designer. *Read/write* memory-mapped registers also allow a software designer to conveniently implement bit-masking operations (such as flipping a single bit in a memory-mapped register). In some cases, *read-only* registers are justified, such as for example to implement registers that reflect hardware status information or sampled-data signals. However, there are very few cases that justify a *write-only* register.
- In software, read/write operations always handle aligned data. For example, extracting bits number 5–12 out of a 32-bit word is more complicated than extracting the second byte of the same word. While a hardware designer may have a tendency to make everything as compact as possible, this practice may result in an address map that is very hard to handle for a software designer. Hence, the address map should respect the alignment of the processor.

10.4.2 Instruction Set

The *instruction set* of custom-hardware module defines how software can control the module. The design of a good instruction-set is a hard problem; it requires the codesigner to make the proper trade-off between flexibility and efficiency. Instructions that trigger complex activities in the hardware module may be very efficient, but they are difficult to use and understand for a software designer. The design of an instruction-set strongly depends on the function of the custom-hardware module, and therefore very few generic guidelines can be given.

- One can distinguish *three classes* of instructions: one-time commands, on–off commands, and configurations. One-time commands trigger a single activity in the hardware module (which may take multiple clock cycles to complete). Pipeline control commands, such as discussed in Sect. 10.3.1, fall in the same category. On–Off commands come in pairs, and they control a continuous activity on the hardware module. Finally, configurations provide a parameter to an algorithm. They affect the general behavior of the hardware module. Making the

proper choice between these is important in order to minimize the amount of control interaction between the software driver and the hardware module.

- *Synchronization* between software and hardware is typically implemented at multiple levels of abstraction. At the lowest level, the hardware/software interfaces will ensure that data items are transferred correctly from hardware to software and vice versa. However, additional synchronization may be needed at the algorithm level. For example, a hardware module with a data-dependent execution time could indicate completion to the driver software through a status flag. In this case, a status flag can support this additional layer of synchronization.
- Another synchronization problem is present when multiple software users share a single hardware module. In this case, the different users all see the same hardware registers, which may be undesirable. This synchronization issue can be handled in several ways. Coprocessor usage could be serialised (allowing only a single user at a time), or else a context switch can be implemented in the hardware module.
- Finally, *reset design* must be carefully considered. An example of flawed reset design is when a hardware module can only be initialized by means of full system reset. It makes sense to define one or several instructions for the hardware module to handle module initialization and reset.

10.5 Example: AES Encryption Coprocessor

Here is an example of command design for an encryption engine. As command design and control shell design as a whole are closely linked, we will describe both of them at the same time. The coprocessor implements the Advanced Encryption Standard, a block cipher with a block size of 128 bit, and a key size of 128 bit. In its most simple form, the processor implements the following function template.

```
void encrypt(unsigned plaintext[4],
             unsigned key[4],
             unsigned ciphertext[4]);
```

In this function call, `plaintext` and `key` are input arguments, and `ciphertext` is an output argument. Each argument is 128-bit, and is mapped into an array of 4 integers on a 32-bit microprocessor. In addition, the update rate of the ports is different. For most practical cases, the `key` argument can be thought of as a parameter, while the `plaintext` and `ciphertext` are arguments.

Figure 10.15 shows the hardware module which must be matched to the `encrypt` function call. The module has two data inputs, `text_in` and `text_out`. The timing diagram on the right of the figure shows how the block operates. The encryption starts with a high level in `ld` when the `done` output is high. A key and a plaintext are sampled from the 128-bit input ports, and several clock cycles later, a corresponding ciphertext is generated, and the `done` pin is raised again.
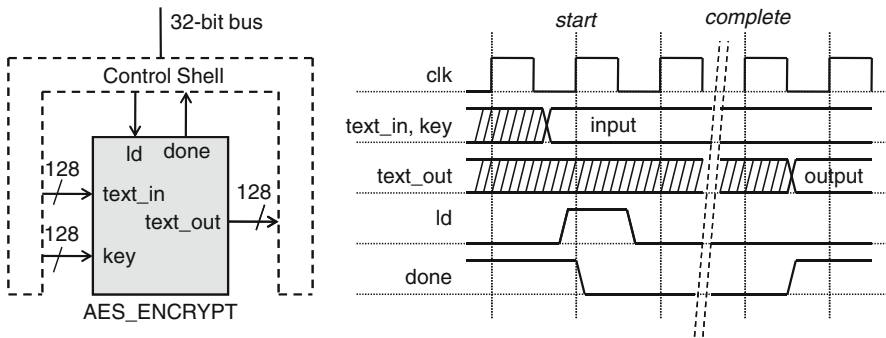
Fig. 10.15 AES encryption module interface

10.5.1 Control Shell Operation

The control shell is operated through a 32-bit bus. We have to design a command set to operate the hardware module according to the interface protocol of Fig. 10.15. We will need to choose a command set which will allow to read and write data to the coprocessor, and to start the encryption. Furthermore, since the key is a parameter and the plaintext input is an argument, it will be useful to have a separate command to change the key and to change the data input. The 128-bit data words need to be transferred in chunks of 32-bit. We'll solve this using a multiplexing technique.

Figure 10.16 shows the datapath of the control shell. Three 32-bit control shell ports are included: `data_in`. `data_out`, and `decode`. The 128-bit operands are assembled using a 96-bit working register in combination with a data input port and a data output port. The control port steers the update of the working register and the control pins of the encryption module.

10.5.2 Programmer's Model

We now consider the software view of the control shell, and define the instruction set of the coprocessor. Table 10.3 shows the address map. The three memory-mapped registers are mapped onto two addresses; one for data, and one for control.

An *instruction* for the AES coprocessor is the combination of a value written to the control register in combination with a value written to (or read from) the data register. Table 10.4 describes the command set of the coprocessor. These commands have the following meaning.

- `INIT` is used to initialize the coprocessor.
- `SHIFT` is used to shift data into the working register of the coprocessor. The argument of this command is `DATA`.
- `KEY` is used to copy the working register of the coprocessor to the key register. The argument of this command is `DATA`.
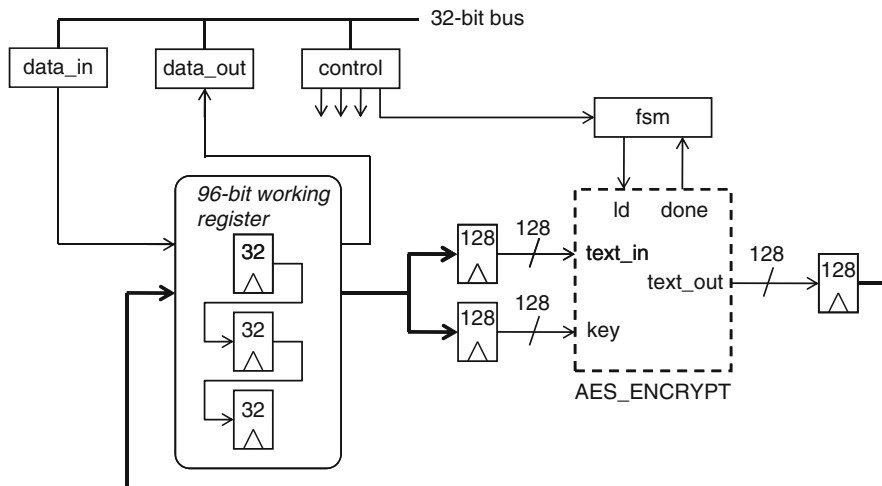
Fig. 10.16 AES encryption control shell datapath

Table 10.3 Command set for AES encryption coprocessor

Offset	Write	Read
0x0	data_in	data_out
0x4	control	

Table 10.4 Command set for AES encryption coprocessor

Command	control	data_in	data_out
INIT	ins_rst	X	0
SHIFT DATA	ins_load	DATA	0
KEY DATA	ins_key	DATA	0
PTEXT DATA	ins_text	DATA	0
ENCRYPT	ins_crypt	X	DONE
CTEXT *DATA	ins_textout	X	DATA
READ *DATA	ins_read	X	DATA
SYNC	ins_idle	X	0

- PTEXT is used to copy the working register of the coprocessor to the plaintext input register. The argument of this command is DATA.
- ENCRYPT is to initiate the encryption operation on the coprocessor. The encryption for this AES module completes in 10 clock cycles. The completion status of the encryption is available in the data output register.
- CTEXT copies the cipher output register of the coprocessor to the working register. This command returns a result in *DATA.
- READ is used to shift data out of the working register of the coprocessor. This command returns a result in *DATA.
- SYNC is used as part of the high-level synchronization protocol used by the coprocessor. This synchronization needs to make sure that the command written to the control register is consistent with the value on the data_in or data_out

ports. This works as follows. Each instruction for the coprocessor is a sequence of two values at the control port, SYNC followed by an active command. To transfer data to the coprocessor, the data_in port needs to be updated between the SYNC command and the active command. To retrieve data from the coprocessor, the data_out port needs to be read after the active command in the sequence SYNC, active command.

Each high level function call in C can now be converted into a sequence of coprocessor commands. The following example illustrates the sequence of commands required to load a key and a plaintext block onto the coprocessor, perform encryption, and retrieve the ciphertext. This command sequence will be generated, for example, through software using memory-mapped write and read operations to control, data_in and data_out.

```
// Command Sequence for Encryption
// Input:  plaintext[0..3]   4 words of plaintext
//         key[0..3]         4 words of key
// Output: ciphertext[0..4]  4 words of ciphertext
  SYNC
  SHIFT plaintext[0]
  SYNC
  SHIFT plaintext[1]
  SYNC
  SHIFT plaintext[2]
  SYNC
  PTEXT plaintext[3]

  SYNC
  SHIFT key[0]
  SYNC
  SHIFT key[1]
  SYNC
  SHIFT key[3]
  SYNC
  KEY key[4]

  ENCRYPT
  wait until (*DATA == 1)

  SYNC
  CTEXT ciphertext[0]
  SYNC
  READ ciphertext[1]
  SYNC
  READ ciphertext[2]
  SYNC
  READ cuphertext[3]
```

Finally, the command set for a coprocessor also needs to choose an *encoding*. As discussed earlier with microcoded engines and microprocessors, command encoding can be very complex and it can contain multiple fields. In this case, we implement a very simple encoding scheme for control which consists of a single field. Table 10.5 illustrates this encoding.

Table 10.5 Command encoding for AES encryption coprocessor

Command	control	Encoding
SYNC	ins_idle	0
INIT	ins_rst	1
SHIFT DATA	ins_load	2
KEY DATA	ins_key	3
PTEXT DATA	ins_text	4
ENCRYPT	ins_crypt	5
CTEXT *DATA	ins_textout	6
READ *DATA	ins_read	7

10.5.3 Software Driver Design

Listing 10.1 shows a software driver for the AES encryption processor. We're making the assumption that we are using a memory-mapped hardware/software interface. The three memory-mapped registers are defined on lines 3–5, and we assume that their pointer value has been initialized properly. The command encoding is captured in an enum statement on line 1.

The coprocessor API uses two functions, set_key and do_encrypt. The set_key function transfers four words of an 128-bit key using the protocol described earlier. First, control is set to ins_idle and the data_in argument is loaded. Next, the actual command is given (lines 11–13). Using ins_load, the first three words of the key are shifted into the working register. Using ins_key, all 128 bits of the key register are programmed.

The do_encrypt function shows a similar sequence for loading the plaintext. The encryption command, which takes no arguments, is shown on lines 27–28. The encryption can take many clock cycles, depending on the architecture of the encryption coprocessor.

Finally, in lines 33–37, the do_encrypt function retrieves the result from the coprocessor. This works again using an interleaved ins_idle/command sequence. First, control is set to ins_idle. Next, the actual command is given and the output argument is retrieved. Using ins_textout, the working register is initialized with the output encryption result. Using ins_read, this result is gradually shifted out of the working register.

Figure 10.17 illustrates how the software driver interacts with the hardware. The clk signal in this diagram is the hardware clock, which can be unrelated to the clock of the microprocessor. The signals control, data_in, and data_out are ports of the control shell. They are controlled by software, and their value can change asynchronously from the hardware. The interleaved idle/active sequence on the control port enables the hardware to select a single clock cycle when the data_in must have a known value, when to start the encryption, and when the data_out must be updated. In the next section, we discuss an RTL implementation for this control shell.

Listing 10.1 A C driver for an AES memory-mapped coprocessor

```
1   enum {ins_idle, ins_rst, ins_load, ins_key,
2         ins_text, ins_crypt, ins_textout, ins_read};
3   volatile unsigned int *control;   // memory-mapped register for controle
4   volatile unsigned int *data_in;   // memory-mapped register for data_in
5   volatile unsigned int *date_out;  // memory-mapped register for data_out
6
7   void set_key(unsigned key[4]) {
8     unsigned i;
9
10    for (i=0; i < 4; i++) {
11      *control = ins_idle;
12      *data_in = key[i];
13      *control = (i == 3) ? ins_key : ins_load;
14    }
15  }
16
17  void do_encrypt(unsigned plaintext[4],
18                  unsigned ciphertext[4]) {
19    unsigned i;
20
21    for (i=0; i < 4; i++) {
22      *control = ins_idle;
23      *data_in = plaintext[i];
24      *control = (i == 3) ? ins_text : ins_load;
25    }
26
27    *control = ins_idle;
28    *control = ins_crypt;
29    while (*data_out == 0)  ;  // wait for encryption to complete
30
31    for (i=0; i < 4; i++) {
32      *control = ins_idle;
33      *control = (i == 0) ? ins_textout : ins_read;
34      ciphertext[i] = *data_out;
35    }
36
37  }
```

10.5.4 Control Shell Design

Listing 10.2 shows a GEZEL implementation of a control shell for the AES coprocessor. The AES hardware module is instantiated on line 13, and it is controlled through three ports: control, data_in, data_out. Several registers (key, text_in, text_out) surround the aes module following an arrangement as shown in Fig. 10.16.

The easiest way to understand the operation of this design is to start with the FSM description on line 45–71. The overall operation of the decoder FSM is an infinite loop that accepts a command from software, and then executes that command. Each state performs a specific step in the command execution.
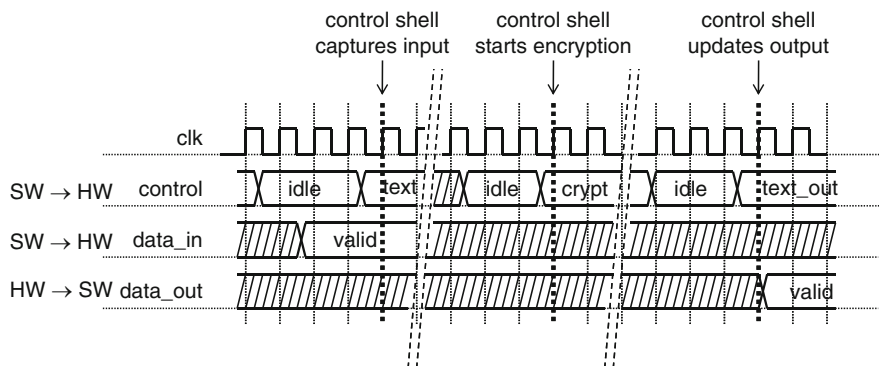
Fig. 10.17 AES control shell operation

- In state s1, the FSM tests insreg, which holds the latest value of the control port, against each possible command. Obviously, it must follow the command encoding chosen earlier for the C driver program. Depending on the value of the command, the FSM will transition to state s2, s3, s5, s6.
- State s2 performs the second half of the command handshake protocol, and waits for ins_idle before going back to s1 for the next command. State s2 is used for the SYNC command.
- State s3 is entered after the ENCRYPT command. This state waits for the encryption to complete. Thus, the control shell is unable to accept new instructions while the coprocessor is operational. The software can detect command completion by reading and testing the value of the data_out memory-mapped register. During the ENCRYPT command, the register will reflect the value of the done flag of the coprocessor.
- State s5 is entered when the first word of the output is read back in software.
- State s6 is entered when the next three words of the output are read back in software.

The datapath of the control shell (Listing 10.2 lines 15–43) implements the register transfers that map the control shell ports to the AES module ports and back.

Listing 10.2 A control shell for the AES coprocessor
```
1   dp aes_decoder(in  control  : ns( 8);
2                  in  data_in   : ns(32);
3                  out data_out : ns(32)) {
4
5     reg rst, ld, done            : ns(  1);
6     reg key, text_in, text_out  : ns(128);
7     sig sigdone                  : ns(  1);
8     sig sigtext_out              : ns(128);
9     reg wrkreg0, wrkreg1, wrkreg2 : ns( 32);
10    reg insreg                   : ns(  8);
11    reg dinreg                   : ns( 32);
12
13    use aes_top(rst, ld, sigdone, key, text_in, sigtext_out);
```

```
14
15    always       { insreg   = control;
16                   dinreg   = data_in;
17                   done     = sigdone;
18                   text_out = sigdone ? sigtext_out : text_out; }
19    sfg dout_d   { data_out = done;                       }
20    sfg dout_t   { data_out = text_out[127:96];           }
21    sfg dout_w   { data_out = wrkreg2;                     }
22    sfg aes_idle { rst = 0;  ld = 0;                       }
23    sfg aes_rst  { rst = 1;  ld = 0;                       }
24    sfg aes_ld   { rst = 0;  ld = 1;                       }
25    sfg putword  { wrkreg0  = dinreg;
26                   wrkreg1  = wrkreg0;
27                   wrkreg2  = wrkreg1;                     }
28    sfg setkey   { key      = (wrkreg2 << 96) |
29                              (wrkreg1 << 64) |
30                              (wrkreg0 << 32) |
31                               dinreg;                     }
32    sfg settext  { text_in  = (wrkreg2 << 96) |
33                              (wrkreg1 << 64) |
34                              (wrkreg0 << 32) |
35                               dinreg;                     }
36    sfg gettext  { data_out = text_out[127:96];
37                   wrkreg2  = text_out[95:64];
38                   wrkreg1  = text_out[63:32];
39                   wrkreg0  = text_out[31:0];     }
40    sfg shiftw   { wrkreg2  = wrkreg1;
41                   wrkreg1  = wrkreg0;                     }
42    sfg getword  { data_out = wrkreg2;                     }
43  }
44
45  fsm faes_decoder(aes_decoder) {
46    initial s0;
47    state s1, s2, s3, s4, s5, s6;
48    @s0 (aes_idle, dout_0)                                      -> s1;
49    @s1 if (insreg == 1)     then (aes_rst,  dout_0)            -> s2;
50        else if (insreg == 2) then (aes_idle, putword, dout_0) -> s2;
51        else if (insreg == 3) then (aes_idle, setkey,  dout_0) -> s2;
52        else if (insreg == 4) then (aes_idle, settext, dout_0) -> s2;
53        else if (insreg == 5) then (aes_ld,   dout_d)          -> s3;
54        else if (insreg == 6) then (aes_idle, gettext)         -> s5;
55        else if (insreg == 7) then (aes_idle, getword)         -> s6;
56                              else (aes_idle, dout_0)          -> s1;
57    // SYNC
58    @s2 if (insreg == 0)     then (aes_idle, dout_0)           -> s1;
59                             else (aes_idle, dout_0)           -> s2;
60    // ENCRYPT
61    @s3 if (done == 1)       then (aes_idle, dout_d)           -> s4;
62                             else (aes_idle, dout_d)           -> s3;
63    @s4 if (insreg == 0)     then (aes_idle, dout_d)           -> s1;
64                             else (aes_idle, dout_d)           -> s4;
65    // CTEXT
66    @s5 if (insreg == 0)     then (aes_idle, dout_0)           -> s1;
67                             else (aes_idle, dout_t)           -> s5;
68    // READ
69    @s6 if (insreg == 0)     then (aes_idle, shiftw, dout_0)  -> s1;
70                             else (aes_idle, dout_w)           -> s6;
71  }
```
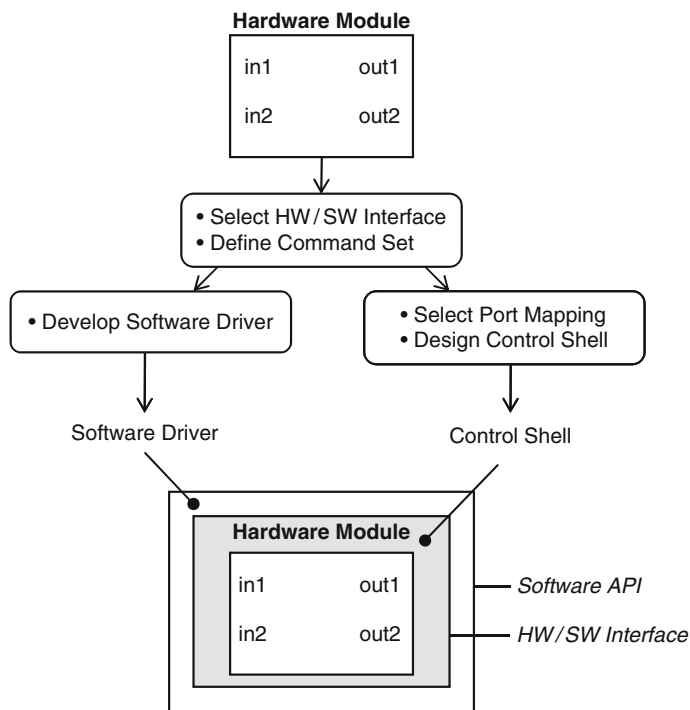
Fig. 10.18 Development of a control shell

Figure 10.18 shows the design process that we have followed so far. We started from a custom hardware module and integrated that into software. This requires the selection of a hardware/software interface, and the definition of a command set. Once these are defined, the integration of hardware and software follow two independent paths. For software, we created a software driver that provides a smooth transition from a high-level API to the hardware/software interface. For hardware, we encapsulated the hardware module into a control shell that connects directly onto the hardware/software interface.

Of course, while Fig. 10.18 shows how to integrate a hardware module, it does not address the performance of the resulting hardware/software interface. As we will discuss next, the design of a control shell often has critical impact on the performance of the overall design.

10.5.5 *System Performance Evaluation*

Control shell design has substantial impact on the overall system performance of a design. We will evaluate the performance of the AES coprocessor following the scheme of Fig. 10.19. We compare an all-software, optimized AES implementation
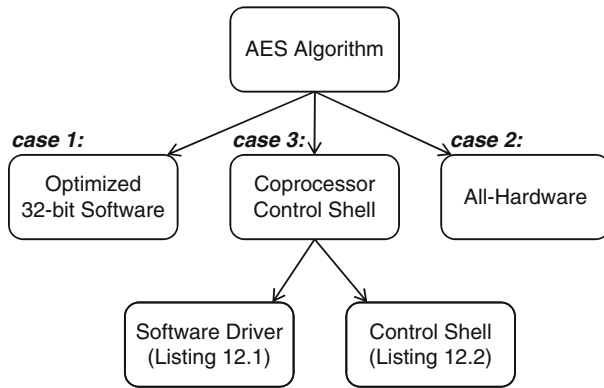
Fig. 10.19 AES coprocessor performance evaluation

with an all-hardware standalone AES implementation. We also compare these results against the performance of an integrated coprocessor using the software driver and control shell developed earlier. The experiments were done using GEZEL and the Simit-ARM intruction set simulator. A C version of the AES algorithm (derived from the OpenSSH library) takes around 3,600 cycles per encryption on an ARM. On the other hand, a full-hardware implementation of AES which requires one clock cycle per round takes 11 clock cycles. The speedup of the hardware design is now defined as:

$$S = \frac{\text{cycles}_{\text{software}}}{\text{cycles}_{\text{hardware}}} \times \frac{T_{\text{clock,software}}}{T_{\text{clock,hardware}}}. \tag{10.1}$$

If we assume that the microprocessor and the custom-hardware run at the same clock frequency, the speedup S is around 327 times for the full hardware implementation.

Next, we also compare the design of the hardware AES integrated onto the control shell discussed earlier. In this case, we use the GEZEL cosimulator to obtain combine hardware and software simulations and obtain overall performance. The software includes the software driver of Listing 10.1. The hardware includes the hardware control shell of Listing 10.2 and the custom hardware module. The system cycle count is around 334 cycles per encryption. This is still a factor of 10.9 faster than the all-software implementation, but is it also a factor of 30.2 *slower* than the all-hardware implementation.

Table 10.6 shows the performance summary of the AES design. The analysis of the speedup factors shows that a hardware/software interface can easily become a bottleneck. In this case, each encryption requires 12 words to be transferred: a key, a plaintext word, and a ciphertext result. There are many optimization possibilities, at the algorithmic level as well as at the architecture level. At the algorithmic level, we can obtain a 30% performance improvement by programming the key only once in the coprocessor and reusing it as much as possible. At the architecture level, we can select a faster, more efficient hardware/software interface: using buffer memories,

Table 10.6 Performance
of 100 AES encryptions on
different platforms

Implementation	Cycle count	Speedup over software
AES software (32-bit)	362,702	1.0
AES custom hardware	1,100	329.7
AES coprocessor	33,381	10.9

burst transfer mode, and so on. In addition, we can also evaluate how to increase the overlap between communication and computation.

10.6 Summary

In this chapter we discussed design techniques to encapsulate hardware modules onto a predefined hardware/software interface. These techniques are collected under the single term *control shell design*. A control shell implements the connection mechanisms between low-level software and a hardware module, including the transfer of operands to and from the hardware module, and the generation of control signals for the hardware module. This requires, in general, the addition of data input/output buffers, and the addition of a controller. Design of a good control shell is not easy; as demonstrated through the example of an encryption processor, the hardware/software interface can add significant overhead, and even become a bottleneck in the overall system performance. Therefore, optimizing the communication between hardware and software is an important objective. One optimization technique is to improve the overlap between hardware/software communication and computation. This can be achieved by means of block-level pipelining and/or internal pipelining. Both forms of pipelining provide improved system-level performance, at the expense of additional hardware and increased complexity in system control. Therefore, we also discussed control techniques for pipelined hardware modules.

10.7 Further Reading

Similar to hardware/software interface design, the implementation of coprocessor control shells is an ad-hoc design process for which few systematic development has been done.

The classic work on optimization of pipelined architectures is by Kogge Kogge (1981), and its ideas on scheduling of pipelined architectures are still relevant.

The Advanced Encryption Standard was published by NIST in Federal Information Processing Standard 197 (FIPS 197), which can be consulted online NIST (2001). A detailed description of an AES coprocessor that follows the principles of control shell design is given by Hodjat in Hodjat and Verbauwhede (2004).

10.8 Problems

10.1. The Motorola DSP56000 processor is a pipelined processor. One of its assembly instructions look as follows.

```
MPY  x0, Y0, A     X: (R0)+, X0   Y: (R4)+, Y0
```

There are no comments in the above line – everything on that line is part of the instruction! This instruction multiplies register X0 with Y0 and places the product in the A accumulator. At the same time, the value of register X0 is updated with the memory location pointed to by register R0, and register Y0 is updated with the memory location pointed to by register R4. Does the Motorola DSP56000 use time-stationary control or does it use data-stationary control?

10.2. Listing 10.3 is a design of a control shell for a median module, which evaluates the median of three values. Study the operation of the coprocessor by studying the GEZEL code listing, and answer the questions later.

Listing 10.3 A control shell for a median module
```
dp median(in  a, b, c : ns(32);
          out q       : ns(32)) {
  sig q1, q2 : ns(32);
  always {
   q1 = (a > b) ? a : b;
   q2 = (a > b) ? b : a;
   q  = (c > q1) ? q1 : (c < q2) ? q2 : c;
  }
}

ipblock myarm {
  iptype "armsystem";
  ipparm "exec=median_driver";
}

ipblock b_datain(out data : ns(32)) {
  iptype "armsystemsource";
  ipparm "core=myarm";
  ipparm "address=0x80000000";
}

ipblock b_dataout(in data : ns(32)) {
  iptype "armsystemsink";
  ipparm "core=myarm";
  ipparm "address=0x80000004";
}

dp medianshell {
  reg a1, a2, a3 : ns(32);
  sig q : ns(32);
  use median(a1, a2, a3, q);

  sig v_in, v_out : ns(32);
  use b_datain(v_in);
```

```
  use b_dataout(v_out);

  use myarm;

  reg old_v : ns(32);
  always {
    old_v = v_in;
    a1    = ((old_v == 0) & (v_in > 0)) ? v_in : a1;
    a2    = ((old_v == 0) & (v_in > 0)) ? a1   : a2;
    a3    = ((old_v == 0) & (v_in > 0)) ? a2   : a3;
    v_out = q;
  }
}

system S {
  medianshell;
}
```

(a) How many data-input and data-output ports does the coprocessor have?
(b) Is the `median` module communication-constrained or computation-constrained with respect to this hardware/software interface?
(c) Describe how software should operate the coprocessor (write values to it, and retrieve the result from it).
(d) Write a C program that uses this coprocessor to evaluate the median value of the numbers 36, 99, and 58.

10.3. Listing 10.4 is a hardware implementation of a vector generator module. Given a set of input coordinates (`ix`, `iy`), the module will generate all integer coordinates lying on the straight line between (`0,0`) and (`ix`, `iy`). The algorithm implemented by the module, the Bresenham algorithm, is used to draw lines on raster-scan displays. The listing in 10.4 assumes that (`ix`, `iy`) lies in the first quadrant, so that `ix` and `iy` are always positive. The timing diagram on the right of Fig. 10.20 shows how to operate the module. New target coordinates can be entered using the `ld` control input. Loading new coordinates also resets the output coordinates to (`0,0`). After that, the `next` control input can be used to retrieve new output points from the vector. The ouput will be refreshed on the second clock edge after `next` is high.

(a) Design a control shell for this module implemented as a coprocessor with a single 32-bit input port, and a single 32-bit output port. Optimize your design to take advantage of the fact that the coordinates of the module are 12-bit. Assume a memory-mapped interface. The design of the control shell includes: definition of the coprocessor instruction set, design of the control shell hardware, and design of sample driver software.
(b) How would you modify the design of (a) when the coprocessor needs to be connected through an 8-bit bus? Describe the required modifications to the instruction-set, the control-shell hardware, and the sample driver software.
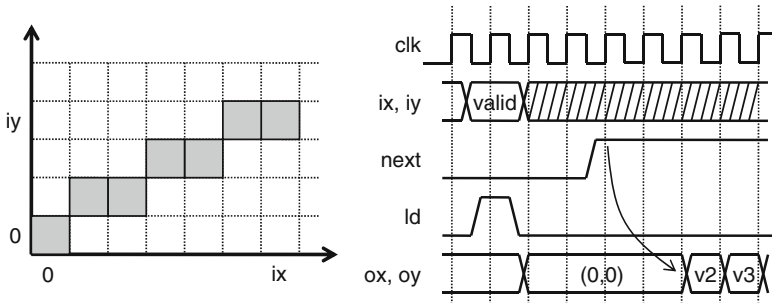
Fig. 10.20 Bresenham vector generation module

Listing 10.4 A Vector Generator

```
dp bresen(in  ix, iy    : ns(12);
          in  ld, next  : ns( 1);
          out ox, oy    : ns(12)) {
  reg x, y                : ns(12);  // current position
  sig nx, ny              : ns(12);
  reg e                   : tc(12);
  reg eol                 : ns(1);
  reg rix, riy            : ns(12);  // current target
  reg einc1, einc2        : tc(12);
  reg xinc, yinc          : ns(1);
  reg ldr, nextr          : ns(1);

  always {
    ldr   = ld;
    nextr = next;
    rix   = ld ? ix : rix;
    riy   = ld ? iy : riy;
    ox    = x;
    oy    = y;
  }

  sfg init {
    einc1 = (rix > riy) ? (riy - rix) : (rix - riy);
    einc2 = (rix > riy) ? riy          : rix;
    xinc  = (rix > riy) ? 1 : 0;
    yinc  = (rix > riy) ? 0 : 1;
    e     = (rix > riy) ? 2 * riy - rix : 2 * rix - riy;
    x     = 0;
    y     = 0;
  }

  sfg loop {
    nx    = (e >= 0) ? x + 1     : x + xinc;
    ny    = (e >= 0) ? y + 1     : y + yinc;
    e     = (e >= 0) ? e + einc1 : e + einc2;
    x     = nx;
    y     = ny;
```

```
    eol   = ((nx == rix) & (ny == riy));
  }

  sfg idle { }
}

fsm f_bresen(bresen) {
  initial s0;
  state s1, s2;
  @s0 (init)                       -> s1;
  @s1 if (ldr) then (init)       -> s1;
      else if (eol)   then (idle) -> s2;
      else if (nextr) then (loop) -> s1;
      else (idle)                 -> s1;

  @s2 if (ldr) then (init)       -> s1;
      else (idle)                 -> s2;
}

// testbench
dp test_bresen {
  reg ix, iy   : ns(12);
  reg ld, next : ns(1);
  sig ox, oy   : ns(12);
  use bresen(ix, iy, ld, next, ox, oy);

  always    { $display("<",$cycle, ">", $dec, " ox ", ox, " oy ",
             oy); }
  sfg init { ld = 1; next = 0; ix = 11; iy = 7; }
  sfg step { ld = 0; next = 1; ix =  0; iy = 0; }
}
fsm ctl_test_bresen(test_bresen) {
  initial s0;
  state s1;
  @s0 (init) -> s1;
  @s1 (step) -> s1;
}

system S {
  test_bresen;
}
```

10.4. Figure 10.21 shows a nonlinear pipeline architecture with three stages. The shaded blocks labeled 1, 2, and 3 represent combinational logic. Pipeline stage 2 iterates two times over each data item entered. As a result, this architecture can only process one data item every clock cycle.

 (a) Find the forbidden latencies for this pipeline.

Cycle	1	2	3	4
Stage1	A			
Stage 2		A	A	
Stage 3				A
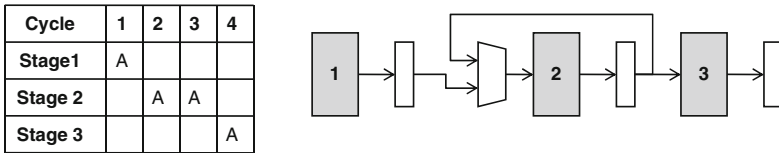
Fig. 10.21 A nonlinear pipeline for problem 12.4

(b) Can this pipeline accept a new data item *every other* clock cycle? If not, how could you modify this architecture so that this becomes possible? *Every other* cycle means that the pipeline should be able to accept new inputs at regular intervals.

Part IV
Applications

The final part of the book describes two complete applications of hardware/software codesign. These examples are given with adequate application background, so that the system-level specification and the design refinement process is clear. The examples are fully implemented as prototypes in Field Programmable Gate Arrays (FPGA), and the reader has access to the full source code of these designs. The first application is a co-processor of the Trivium stream-cipher algorithm. This coprocessor is used for streaming-encryption applications. The second application is a co-processor for the evaluation of digital rotation functions (CORDIC).

Chapter 11
Trivium Crypto-Coprocessor

Abstract Stream ciphers are complex state machines that generate an infinite stream of pseudo-random bits starting from a single key. These bits can be used as a *keystream* in encryption and decryption operations. In this chapter we'll discuss the implementation of such a stream cipher algorithm, called Trivium, as a co-processor. The co-processor is attached to a host processor. The software on that host processor initializes the Trivium coprocessor, and retrieves a very long (infinite) keystream. We consider different types of host processors, including an 8-bit 8051 microcontroller, a 32-bit StrongARM RISC, and a 32-bit Microblaze processor. We will consider the impact of different types of hardware–software interfaces on the performance of the overall design. We will also investigate the path to implementation on an FPGA.

11.1 The Trivium Stream Cipher Algorithm

The Trivium stream cipher algorithm was proposed by Christophe De Canniere and Bart Preneel in 2006 in the context of the eSTREAM project, a European effort that ran from 2004 to 2008 to develop a new stream ciphers. In September 2008, Trivium was selected as a part of the official eSTREAM portfolio, together with six other stream cipher algorithms. The algorithm is remarkably simple, yet to this date it remains unbroken in practice. We will clarify further what it means to *break* a stream cipher. In this section, we discuss the concept of a stream cipher, and the details of the Trivium algorithm.

11.1.1 Stream Ciphers

Let us first clarify how a stream cipher works and how it is different from a block cipher such as AES (which was discussed in Chap. 10). The left of Fig. 11.1 illustrates the difference between a stream cipher and a block cipher. A stream cipher
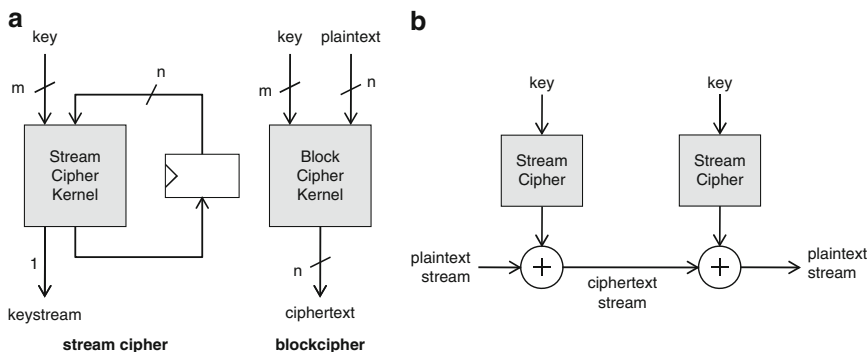
Fig. 11.1 (**a**) Stream cipher and Block cipher (**b**) Stream cipher encryption/decryption

is a state machine with an internal state register of n bits. The stream cipher kernel will initialize the state register based on a key, and it will update the state register while producing the keystream.

In contrast to a stream cipher, a block cipher is a state-less function that combines an m bit key with a block of n bits of plaintext. Because there is no state, the encryption of one block of plaintext bits is independent of the encryption of the previous block of bits. Of course, many hardware implementations of block ciphers *do* include registers. However, these registers are an effect of sequentializing the block cipher algorithm; it is perfectly feasible to implement block ciphers without any register.

The cryptographic properties of the stream cipher are based on the highly non-linear functions used for state register initialization and state register update. These nonlinear functions ensure that the keystream cannot be predicted even after a very large number of keystream bits has been observed. *Breaking* a stream cipher means that one has found a way to predict the output of the stream cipher, or even better, one has found a way to reveal the contents of the state register. For a state register of n bits, the stream cipher can be in 2^n possible states, so the total length of the key stream can be no more than $n.2^n$ bits. Practical stream ciphers have an n between 80 and several hundred.

A stream cipher by itself does not produce ciphertext, but only a stream of keybits. The right of Fig. 11.1 illustrates how one can perform encryption and decryption with a stream cipher. The keystream is combined (xor-ed) with a stream of plaintext bits to obtain a stream of ciphertext bits. Using an identical stream cipher that produces the same keystream, the stream of ciphertext bits can be converted back to plaintext using a second xor operation.

A stream cipher algorithm produces, conceptually, a stream of bits. When the message to encrypt is not formatted as a stream of bits, but as a stream of bigger entities, such as words, the stream cipher will need to produce a stream of words instead. On a RISC processor, for example, it makes sense to represent a stream as a sequence of 32-bit words. Therefore, depending on the computer architecture,
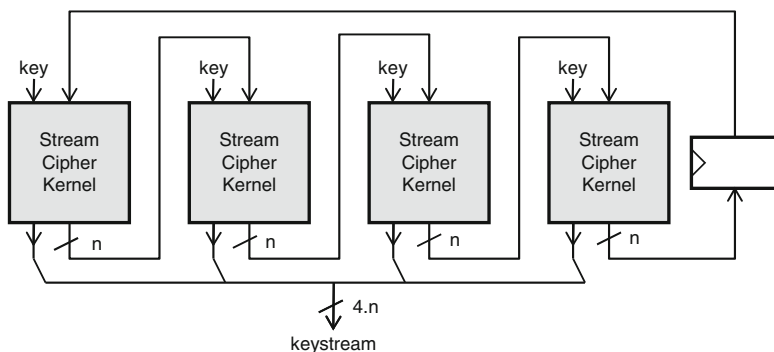
Fig. 11.2 Parallel stream cipher implementation

we would have a key-stream formatted as single bits, as bytes, as 32-bit words, and so on. One way to obtain a wider keystream is to run the stream cipher kernel at high speed and perform a serial-to-parallel conversion of the output. An alternative is illustrated in Fig. 11.2: the stream cipher can be easily parallelized to produce multiple keystream bits per clock cycle. This is especially useful when the stream cipher kernel is a very simple function, as is the case with Trivium.

11.1.2 Trivium

Trivium is a stream cipher with a state register of 288 bits. The state register is initialized based on an 80-bit key and an 80-bit initial value (IV). After initialization, Trivium produces a stream of keybits. The specification of Trivium is shown in Listing 11.1. Each iteration of the loop, a single output bit z is generated, and the state register s is updated. The addition and multiplication (+ and .) are taken over

Listing 11.1 Trivium round

```
state s[1..288];
loop
  t1          = s[66] + s[93];
  t2          = s[162] + s]177];
  t3          = s[243] + s[288];
  z           = t1 + t2 + t3;
  t1          = t1 + s[91].s[92] + s[171];
  t2          = t2 + s[175].s[176] + s[264];
  t3          = t3 + s[286].s[287] + s[69];
  s[1..93]    = t3 || s[1..92];
  s[94..177]  = t1 || s[94..176];
  s[178..288] = t2 || s[178..287];
end loop
```
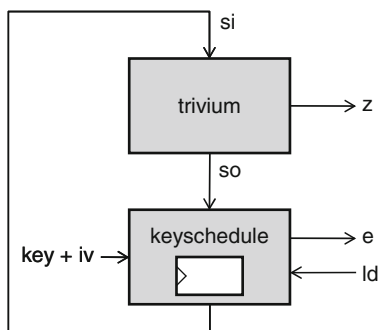
Listing 11.2 Trivium initialization

```
state s[1..287];
s[1..92]      = K || 0;
s[94..177]    = IV || 0;
s[178 .. 288] = 7;
loop for (4 * 288)
  t1          = s[66] + s[93] + s[91].s[92] + s[171];
  t2          = s[162] + s]177] + s[175].s[176] + s[264];
  t3          = s[243] + s[288] + s[286].s[287] + s[69];
  s[1..93]    = t3 || s[1..92];
  s[94..177]  = t1 || s[94..176];
  s[178..288] = t2 || s[178..287];
end loop
```

Fig. 11.3 Hardware mapping of Trivium

GF(2). They can be implemented with exclusive-or and bitwise-and, respectively. The double-bar operation ($||$) denotes concatenation.

The initialization of the state register proceeds as follows. The 80-bit key K and the 80-bit initial value IV are loaded into the state register, and the state register is updated 4 times 288 without producing keybits. After that, the state register is ready to produce keystream bits. This is illustrated in the pseudocode of Listing 11.2.

These listings confirm that, from a computational perspective, Trivium is a very simple algorithm. A single state register update requires nine single-bit xor operations and three single-bit and operations. We need two additional single-bit xor operations to produce the output bit z.

11.1.3 Hardware Mapping of Trivium

A straightforward hardware mapping of the Trivium algorithm requires 288 registers, 11 XOR gates, and 3 AND gates. Clearly, the largest cost of this algorithm is in the storage. Figure 11.3 shows how Trivium is partitioned into hardware modules.

- The trivium module calculates the next state. We will use the term Trivium *kernel* to indicate the loop body of Listing 11.1, without the state register update.

- The `keyschedule` module manages state register initialization and update. The `keyschedule` module has a single control input `ld` to initiate the state register initialization processor. In addition, `keyschedule` has a single status bit `e` that indicates when the initialization has completed, and thus when the output keystream `z` is valid. This partitioning between keyschedule and trivium kernel was chosen with loop unrolling in mind (Fig. 11.2).

Based on this partitioning and the Trivium specification given earlier, it is straightforward to create a GEZEL description of Trivium. Listing 11.3 shows the implementation of a 1 bit per cycle Trivium. The control signals in the `key-schedule` module are generated based on a counter which is initialized after a pulse on the `ld` control input.

To create a bit-parallel keystream, we need to modify the code as follows. First, we need to instantiate the `trivium` module multiple times, and chain the state input and output ports together as shown in Fig. 11.2. Second, we need to adjust the key schedule, because the initialization phase will take less than four times 288 clock cycles. As an example, Listing 11.4 shows how to unroll Trivium eight times, thus obtain a stream cipher that generates one byte of keystream per clock cycle. In this case, the initialization completes 8 times faster, after 143 clock cycles (line 33).

What is the limiting factor when unrolling Trivium? First, notice that unrolling the algorithm will not increase the critical path of the Trivium kernel operations as long as they affect different state register bits. Thus, as long as the state register bits read are different from the state register bits written, then all the kernel operations are independent. Next, observe that a single Trivium round consists of three circular shift registers, as shown in Fig. 11.4. The length of each shift register is indicated inside of the shaded boxes. To find how far we can unroll this structure, we look
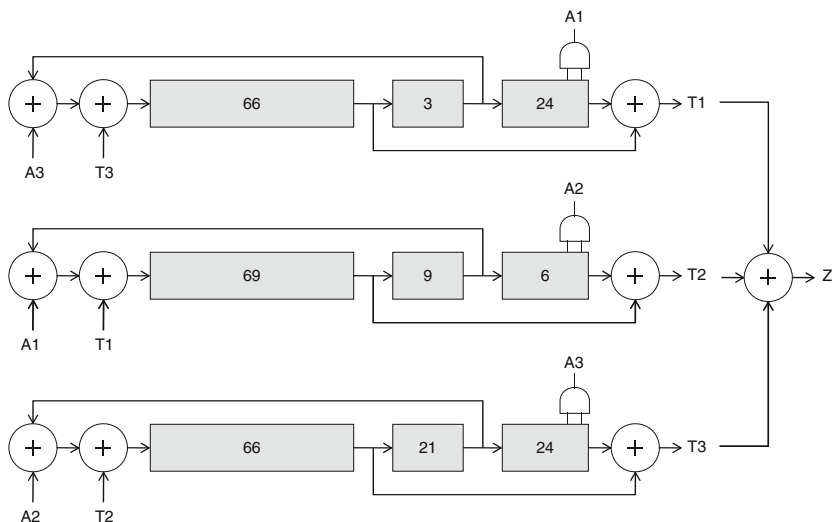

Fig. 11.4 Trivium round structure

Listing 11.3 1-bit-per-cycle Trivium

```
1   dp trivium(in  si : ns(288);  // state input
2              out so : ns(288);  // state output
3              out z  : ns(1)) {  // crypto bit out
4     sig  t1,  t2,  t3 : ns(  1);
5     sig t11, t22, t33 : ns(  1);
6     sig saa           : ns( 93);
7     sig sbb           : ns( 84);
8     sig scc           : ns(111);
9     always {
10      t1  = si[ 65] ^ si[ 92];
11      t2  = si[161] ^ si[176];
12      t3  = si[242] ^ si[287];
13      z   = t1 ^ t2 ^ t3;
14      t11 = t1 ^ (si[ 90] & si[ 91]) ^ si[170];
15      t22 = t2 ^ (si[174] & si[175]) ^ si[263];
16      t33 = t3 ^ (si[285] & si[286]) ^ si[ 68];
17      saa = si[  0: 92] # t33;
18      sbb = si[ 93:176] # t11;
19      scc = si[177:287] # t22;
20      so  = scc # sbb # saa;
21    }
22  }
23
24  dp keyschedule(in  ld : ns(1);     // reload key & iv
25                 in  iv : ns(80);    // initialization vector
26                 in key : ns(80);    // key
27                 out  e : ns(1);     // output valid
28                 in  si : ns(288);   // state input
29                 out so : ns(288)) { // state output
30    reg s        : ns(288);          // state register
31    reg cnt      : ns(11);           // initialization counter
32    sig saa      : ns( 93);
33    sig sbb      : ns( 84);
34    sig scc      : ns(111);
35    sig cte      : ns(111);
36    always {
37      saa = ld ? key       : si[  0: 92];
38      sbb = ld ? iv        : si[ 93:176];
39      cte = 7;
40      scc = ld ? (cte << 108) : si[177:287];
41      s   = scc # sbb # saa;
42      so  = s;
43      cnt = ld ? 1152 : (cnt ? cnt - 1 : cnt);        1152 = 4 * 288
44      e   = (cnt ? 0 : 1);
45    }
46  }
```

Listing 11.4 1-byte-per-cycle Trivium

```
1  dp trivium(in  si : ns(288);  // state input
2             out so : ns(288);  // state output
3             out z  : ns(1)) {  // crypto bit out
4    // ...
5  }
6  dp trivium2 : trivium
7  dp trivium3 : trivium
8  dp trivium4 : trivium
9  dp trivium5 : trivium
10 dp trivium6 : trivium
11 dp trivium7 : trivium
12 dp trivium8 : trivium
13
14 dp keyschedule(in  ld : ns(1);     // reload key & iv
15                in  iv : ns(80);    // initialization vector
16                in key : ns(80);    // key
17                out  e : ns(1);     // output valid
18                in  si : ns(288);   // state input
19                out so : ns(288)) { // state output
20   reg s         : ns(288);             // state register
21   reg cnt       : ns(11);              // initialization counter
22   sig saa       : ns( 93);
23   sig sbb       : ns( 84);
24   sig scc       : ns(111);
25   sig cte       : ns(111);
26   always {
27     saa  = ld ? key        : si[  0: 92];
28     sbb  = ld ? iv         : si[ 93:176];
29     cte  = 7;
30     scc  = ld ? (cte << 108) : si[177:287];
31     s    = scc # sbb # saa;
32     so   = s;
33     cnt  = ld ? 143 : (cnt ? cnt - 1 : cnt);      143 = 4 * 288 / 8 - 1
34     e    = (cnt ? 0 : 1);
35   }
36 }
37
38 dp triviumtop(in  ld : ns(1);     // reload key & iv
39               in  iv : ns(80);    // initialization vector
40               in key : ns(80);    // key
41               out  z : ns(8);     // encrypted output
42               out  e : ns(1)) {   // output valid
43   sig si, so0, so1, so2, so3, so4, so5, so6, so7 : ns(288);
44   sig z0, z1, z2, z3, z4, z5, z6, z7 : ns(1);
45   use keyschedule(ld, iv, key, e, si, so0);
46   use trivium    (so0, so1, z0);
47   use trivium2   (so1, so2, z1);
48   use trivium3   (so2, so3, z2);
49   use trivium4   (so3, so4, z3);
50   use trivium5   (so4, so5, z4);
51   use trivium6   (so5, so6, z5);
52   use trivium7   (so6, so7, z6);
53   use trivium8   (so7, si,  z7);
54   always {
55     z = z0 # z1 # z2 # z3 # z4 # z5 # z6 # z7;
56   }
57 }
```

for the *smallest* feedback loop. This loop is located in the upper circular shift register, and spans 69 bits. Therefore, we can unroll Trivium at least 69 times before the critical path will increase beyond a single AND-gate and two XOR gates. In practice, this means that Trivium can be easily adjusted to generate a key-stream of double-words (64 bits). After that, the critical path will increase each 69 bits. Thus, a 192 bit-parallel Trivium will be twice as slow as a 64 bit-parallel Trivium, and a 256 bit-parallel Trivium will be roughly three times as slow.

11.1.4 A Hardware Testbench for Trivium

For completeness, we also show a hardware testbench for the Trivium kernel in Listing 11.5. In this testbench, the key value is programmed to 0x80 and the IV to 0x0. After loading the key (lines 12–15), the testbench waits until the e-flag indicates the keystream is ready (lines 29–30). Next, each output byte is printed on the output (lines 19–22). The first 160 cycles of the simulation produce the following output.

```
> fdlsim trivium8.fdl 160
147 11001100 cc
148 11001110 ce
149 01110101 75
150 01111011 7b
151 10011001 99
152 10111101 bd
153 01111001 79
154 00100000 20
155 10011010 9a
156 00100011 23
157 01011010 5a
158 10001000 88
159 00010010 12
```

The key stream bytes produced by Trivium consists of the bytes 0xcc, 0xce, 0x75, 0x7b, 0x99, and so on. The bits in each byte are read left to right (from most significant to least significant). In the next sections, we will integrate this module as a coprocessor next to a processor.

11.2 Trivium for 8-bit Platforms

Our first coprocessor design will attach the Trivium stream cipher hardware to an 8-bit microcontroller. We will make use of an 8051 microcontroller. Like many other microcontrollers, it has several general-purpose digital input–output ports, which can be used to create hardware–software interfaces. Thus, we will be building a *port-mapped* control shell for the Trivium coprocessor. The 8051 microcontroller

Listing 11.5 Testbench for a 1-byte-per-cycle Trivium

```
1   // testbench
2   dp triviumtest {
3     sig ld       : ns(1);
4     sig iv, key  : ns(80);
5     sig e        : ns(1);
6     reg re       : ns(1);
7     sig z        : ns(8);
8     reg rz       : ns(8);
9     sig bs       : ns(8);
10    use triviumtop(ld, iv, key, z, e);
11    always    { rz  = z;
12                re  = e;    }
13    sfg init0 { iv  = 0;
14                key = 0x80;
15                ld  = 1;
16              }
17    sfg idle  { ld = 0; }
18    sfg bstuf { ld = 0;
19              }
20    sfg show  { ld = 0;
21                bs = rz;
22                $display($cycle, " ", $bin, bs, $hex, " ", bs);
23              }
24  }
25  fsm ft(triviumtest) {
26    initial s0;
27    state s10, s1, s2;
28    @s0   (init0)                    -> s10;
29    @s10  (init0)                    -> s1;
30    @s1   if (re) then (bstuf)       -> s2;
31               else (idle)          -> s1;
32    @s2   (show)                     -> s2;
33  }
```

also has an external memory bus (XBUS), which supports a memory space of 64K. Such external memory busses are rather uncommon for microcontrollers. However, we will demonstrate the use of such a memory-bus in our design as well.

11.2.1 Overall Design of the 8051 Coprocessor

Figure 11.5 illustrates the overall design. The coprocessor is controlled through three 8-bit ports (P0, P1, P2). They are used to transfer operands, instructions, and to retrieve the coprocessor status, respectively. The Trivium hardware will dump the resulting keystream into a dual-port RAM module, and the contents of the keystream can be retrieved by the 8051 through the XBUS.

The system works as follows. First, the 8051 programs a key and an initialization vector into the Trivium coprocessor. Next, the 8051 commands the Trivium
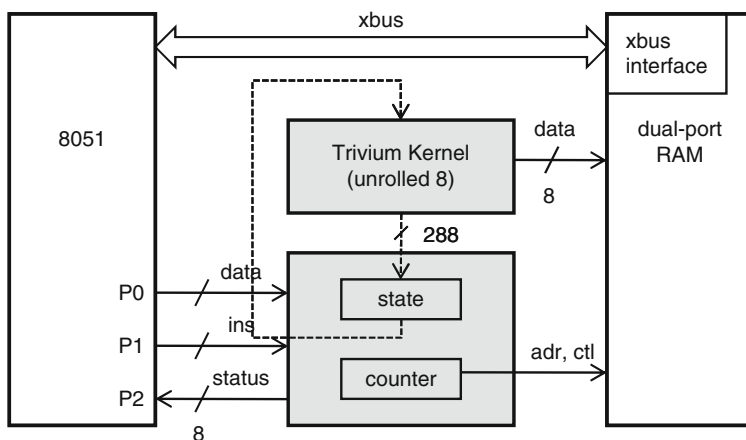
Fig. 11.5 Trivium coprocessor integration on a 8051

coprocessor to generate N keybytes, which will be stored in the shared RAM on the XBUS. Finally, the 8051 can retrieve the keybytes from the RAM. Note that the retrieval of the keybytes from RAM is only shown as an example; depending on the actual application, the keystream may be used for a different purpose. The essential part of this example is the control of the coprocessor from within the microcontroller.

To design the control shell, we will need to develop a command set for the Trivium coprocessor. As the 8-bit ports of the 8051 do not include strobes, we will make use of a similar handshake procedure as was used earlier in Chap. 10: a simple *idle* instruction will help us to determine the exact clock cycle when a command becomes valid. The command set for the coprocessor is shown in Table 11.1. All of the commands except one complete within a single clock cycle. The last command, ins_enc, takes up to 256 clock cycles to complete. The status port of the 8051 is used to indicate when the the encryption phase has completed. Figure 11.6 illustrates the command sequence for the generation of 10 bytes of keystream. Note that the status port becomes zero when the keystream generation is complete.

11.2.2 Hardware Platform of the 8051 Coprocessor

We will now capture the hardware platform of Fig. 11.5 as a GEZEL program. Listing 11.6 shows the complete platform apart from the Trivium kernel (which was discussed in Sect. 11.1.3). The first part of the Listing captures all the 8051-specific interfaces. The Trivium coprocessor will be connected on top of these interfaces.

- Line 1–6: The 8051 core my8051 will read in an executable called trivium.ihx. The executable is in Intel Hex Format, a common format for microcontroller binaries. The period of the core is 1, meaning that the clock

Table 11.1 Command set for trivium coprocessor

Value at P0 (instruction)	Value at P1 (data)	Value at P2 (status)	Meaning
ins_idle	do not care	do not care	Idle Instruction
ins_key0	Key Byte 0	do not care	Program key byte
ins_key1	Key Byte 1	do not care	Program key byte
ins_key2	Key Byte 2	do not care	Program key byte
ins_key3	Key Byte 3	do not care	Program key byte
ins_key4	Key Byte 4	do not care	Program key byte
ins_key5	Key Byte 5	do not care	Program key byte
ins_key6	Key Byte 6	do not care	Program key byte
ins_key7	Key Byte 7	do not care	Program key byte
ins_key8	Key Byte 8	do not care	Program key byte
ins_key9	Key Byte 9	do not care	Program key byte
ins_iv0	IV Byte 0	do not care	Program IV byte
ins_iv1	IV Byte 1	do not care	Program IV byte
ins_iv2	IV Byte 2	do not care	Program IV byte
ins_iv3	IV Byte 3	do not care	Program IV byte
ins_iv4	IV Byte 4	do not care	Program IV byte
ins_iv5	IV Byte 5	do not care	Program IV byte
ins_iv6	IV Byte 6	do not care	Program IV byte
ins_iv7	IV Byte 7	do not care	Program IV byte
ins_iv8	IV Byte 8	do not care	Program IV byte
ins_iv9	IV Byte 9	do not care	Program IV byte
ins_init	do not care	do not care	Initializes state register
ins_enc	rounds	isready	Encrypts rounds
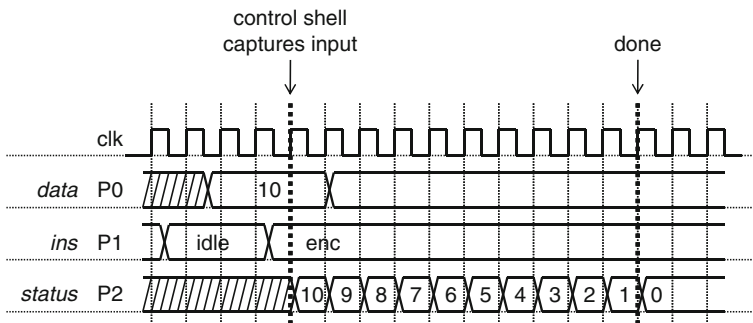
Fig. 11.6 Command sequence for encryption

frequency of the 8051 core is the same as the hardware clock frequency. A traditional 8051 architecture uses 12 clock cycles per instruction. Thus, a `period` of 1 means that there will be a single instruction executing each 12 clock cycles.

- Line 7–21: Three I/O ports of the 8051 are defined as P0, P1, and P2. A port is configured either as input or as output by choosing its type to be

       `i8051systemsource` (e.g., Line 8,13) or else `i8051systemsink` (e.g., Line 18).

- Line 22–30: A dual-port, shared-memory RAM module attached to the XBUS is modeled using an `ipblock`. The module allows to specify the starting address (`xbus`, Line 28) as well as the amount of memory locations (`xrange`, Line 29).

The `triviumitf` module integrates the Trivium hardware kernel (Line 44) on top of the hardware/software interfaces. Several registers are used to manage this module, including a Trivium state register `tstate`, a round counter `cnt`, and a ram address counter `ramcnt` (Line 50–53).

The key and initialization vector are programmed into the state register through a sequence of chained multiplexers (Line 56–82). This works as follows. First consider the update of `tstate` on Line 82. If the counter value `cnt` is nonzero, `tstate` will copy the value `so`, which is the output of the Trivium kernel. If the counter value `cnt` is zero, `tstate` will instead copy the value of `init`, which is defined through Line 56–78. Thus, by loading a nonzero value into `cnt` (Line 80–81), the Trivium kernel performs active encryption rounds.

Now, when the count value is zero, the state register can be reinitialized with a chosen key and initialization vector. Each particular command in the range `0x1` to `0x14` will replace a single byte of the key or the initialization vector (Line 56–76). The `init` command will pad `0b111` into the most significant bits of the state register (Line 78).

Finally, the RAM control logic is shown on Line 86–89. Whenever the count value is nonzero, the ram address starts incrementing and the ram interface carries a write command.

Listing 11.6 Hardware platform for the 8051 coprocessor

```
1   ipblock my8051 {                                          8051 core
2     iptype "i8051system";
3     ipparm "exec=trivium.ihx";
4     ipparm "verbose=1";
5     ipparm "period=1";
6   }
7   ipblock my8051_data(out data : ns(8)) {                   8051 interfaces
8     iptype "i8051systemsource";
9     ipparm "core=my8051";
10    ipparm "port=P0";
11  }
12  ipblock my8051_ins(out data : ns(8)) {
13    iptype "i8051systemsource";
14    ipparm "core=my8051";
15    ipparm "port=P1";
16  }
17  ipblock my8051_status(in data : ns(8)) {
18    iptype "i8051systemsink";
19    ipparm "core=my8051";
20    ipparm "port=P2";
21  }
```

```
22   ipblock my8051_xram(in  idata   : ns(8);
23                       out odata   : ns(8);
24                       in  address : ns(8);
25                       in  wr      : ns(1)) {
26     iptype "i8051buffer";
27     ipparm "core=my8051";
28     ipparm "xbus=0x4000";
29     ipparm "xrange=0x100"; // 256 locations at address 0x4000
30   }
31
32   dp triviumitf {                                          Trivium control shell
33     sig updata, upins, upstatus : ns(8);
34     use my8051_data   (updata  );
35     use my8051_ins    (upins   );
36     use my8051_status(upstatus);
37
38     sig ramadr, ramidata, ramodata : ns(8);
39     sig wr : ns(1);
40     use my8051_xram(ramidata, ramodata, ramadr, wr);
41
42     sig si, so : ns(288);
43     sig z : ns(8);
44     use trivium80(si, so, z);                              Trivium kernel
45
46     sig k0, k1, k2, k3, k4, k5, k6, k7, k8, k9 : ns(288);
47     sig v0, v1, v2, v3, v4, v5, v6, v7, v8, v9 : ns(288);
48     sig init : ns(288);
49
50     reg tstate : ns(288);
51     sig newcnt : ns(8);
52     reg cnt    : ns(8);
53     reg ramcnt : ns(8);
54
55     always {
56       k0 = (upins == 0x1) ? tstate[287:  8] # updata : tstate;
57       k1 = (upins == 0x2) ? k0[287: 16] # updata # k0[ 7: 0] : k0;
58       k2 = (upins == 0x3) ? k1[287: 24] # updata # k1[15: 0] : k1;
59       k3 = (upins == 0x4) ? k2[287: 32] # updata # k2[23: 0] : k2;
60       k4 = (upins == 0x5) ? k3[287: 40] # updata # k3[31: 0] : k3;
61       k5 = (upins == 0x6) ? k4[287: 48] # updata # k4[39: 0] : k4;
62       k6 = (upins == 0x7) ? k5[287: 56] # updata # k5[47: 0] : k5;
63       k7 = (upins == 0x8) ? k6[287: 64] # updata # k6[55: 0] : k6;
64       k8 = (upins == 0x9) ? k7[287: 72] # updata # k7[63: 0] : k7;
65       k9 = (upins == 0xA) ? k8[287: 80] # updata # k8[71: 0] : k8;
66
67       v0 = (upins == 0xB)  ? k9[287:101] # updata # k9[ 92: 0] : k9;
68       v1 = (upins == 0xC)  ? v0[287:109] # updata # v0[100: 0] : v0;
69       v2 = (upins == 0xD)  ? v1[287:117] # updata # v1[108: 0] : v1;
70       v3 = (upins == 0xE)  ? v2[287:125] # updata # v2[116: 0] : v2;
71       v4 = (upins == 0xF)  ? v3[287:133] # updata # v3[124: 0] : v3;
72       v5 = (upins == 0x10) ? v4[287:141] # updata # v4[132: 0] : v4;
73       v6 = (upins == 0x11) ? v5[287:149] # updata # v5[140: 0] : v5;
74       v7 = (upins == 0x12) ? v6[287:157] # updata # v6[148: 0] : v6;
75       v8 = (upins == 0x13) ? v7[287:165] # updata # v7[156: 0] : v7;
76       v9 = (upins == 0x14) ? v8[287:173] # updata # v8[164: 0] : v8;
```

```
77
78      init = (upins == 015) ? 0b111 # v9[284:0] : v9;
79
80      newcnt    = (upins == 0x16) ? updata : 0;
81      cnt       = (cnt) ? cnt - 1 : newcnt;
82      tstate    = (cnt) ? so : init;
83      si        = tstate;
84      upstatus = cnt;
85
86      ramcnt    = (cnt) ? ramcnt + 1 : 0;
87      ramadr    = ramcnt;
88      wr        = (cnt) ? 1 : 0;
89      ramidata = z;
90    }
91  }
92
93  system S {
94    my8051;
95    triviumitf;
96  }
```

11.2.3   Software Driver for 8051

The software driver for the above coprocessor is shown in Listing 11.7. This C
code is written for the 8051 and can be compiled with SDCC, the Small Devices C
Compiler (http://sdcc.sourceforge.net). This compiler allows directly
using symbolic names, such as the names of the I/O ports P0, P1, and P2.

The program demonstrates the loading of a key and initialization vector (Line
21–43), the execution of the key schedule (Line 46–50), and the generation of a
keystream of 250 bytes (Line 53–56). Note that the software driver does not strictly
follow the interleaving of active commands with ins_idle. However, this code
will work fine for the hardware model from Listing 11.6.

As discussed before, the key scheduling of Trivium is similar to the normal op-
eration of Trivium. Key scheduling involves running Trivium for a fixed number of
rounds while discarding the keystream. Hence, the key scheduling part of the driver
software is, apart from the number of rounds, identical to the encryption part.

Finally, Line 64 illustrates how to terminate the simulation. By writing the value
0x55 into port P3, the simulation will halt. This is an artificial construct. Indeed,
the software on a real microcontroller will run indefinitely.

Listing 11.7 8051 Software driver for the Trivium coprocessor
```
1  #include <8051.h>
2
3  enum {ins_idle, ins_key0, ins_key1,
4        ins_key2, ins_key3, ins_key4, ins_key5,
5        ins_key6, ins_key7, ins_key8, ins_key9,
6        ins_iv0,  ins_iv1,  ins_iv2,  ins_iv3,
```

```
7           ins_iv4,  ins_iv5,  ins_iv6,  ins_iv7,
8           ins_iv8,  ins_iv9,  ins_init, ins_enc};
9
10  void terminate() {
11    // special command to stop simulator
12    P3 = 0x55;
13  }
14
15  void main() {
16    volatile xdata unsigned char *shared =
17      (volatile xdata unsigned char *) 0x4000;
18    unsigned i;
19
20    // program key, iv
21    P1 = ins_key0; P0 = 0x80;
22    P1 = ins_key1; P0 = 0x00;
23    P1 = ins_key2; P0 = 0x00;
24    P1 = ins_key3; P0 = 0x00;
25    P1 = ins_key4; P0 = 0x00;
26    P1 = ins_key5; P0 = 0x00;
27    P1 = ins_key6; P0 = 0x00;
28    P1 = ins_key7; P0 = 0x00;
29    P1 = ins_key8; P0 = 0x00;
30    P1 = ins_key9; P0 = 0x00;
31    P1 = ins_iv0;  P0 = 0x00;
32    P1 = ins_iv1;  P0 = 0x00;
33    P1 = ins_iv2;  P0 = 0x00;
34    P1 = ins_iv3;  P0 = 0x00;
35    P1 = ins_iv4;  P0 = 0x00;
36    P1 = ins_iv5;  P0 = 0x00;
37    P1 = ins_iv6;  P0 = 0x00;
38    P1 = ins_iv7;  P0 = 0x00;
39    P1 = ins_iv8;  P0 = 0x00;
40    P1 = ins_iv9;  P0 = 0x00;
41
42    // prepare for key schedule
43    P1 = ins_init;
44
45    // execute key schedule
46    P0 = 143; P1 = ins_enc;
47    P1 = ins_idle;
48
49    // wait until done
50    while (P2) ;
51
52    // produce 250 stream bytes
53    P0 = 250; P1 = ins_enc;
54    P1 = ins_idle;
55
56    while (P2) ;  // wait until done
57
58    // read out shared ram and send to port P0, P1
59    for (i=0; i< 8; i++) {
60      P0 = i;
```

```
61      P1 = shared[i];
62    }
63
64    terminate();
65  }
```

We can now compile the software driver and execute the simulation. The following commands illustrate the output generated by the program. Note that the 8051 microcontroller does not support standard I/O in the traditional sense: it is not possible to use `printf` statements without additional I/O hardware and appropriate software libraries. The instruction-set simulator deals with this limitation by printing the value of all ports each time a new value is written into them. Hence, the four columns below correspond to the value of `P0`, `P1`, `P2`, and `P3`, respectively. We annotated the tool output to clarify the meaning of the sequence of values.

```
> sdcc trivium.c
> gplatform tstream.fdl
i8051system: loading executable [trivium.ihx]
0xFF     0x01     0x00     0xFF

0x80     0x01     0x00     0xFF                                    Program Key

0x80     0x02     0x00     0xFF
0x00     0x02     0x00     0xFF
0x00     0x03     0x00     0xFF
0x00     0x04     0x00     0xFF
0x00     0x05     0x00     0xFF
0x00     0x06     0x00     0xFF
0x00     0x07     0x00     0xFF
0x00     0x08     0x00     0xFF
0x00     0x09     0x00     0xFF
0x00     0x0A     0x00     0xFF

0x00     0x0B     0x00     0xFF                                    Program IV

0x00     0x0C     0x00     0xFF
0x00     0x0D     0x00     0xFF
0x00     0x0E     0x00     0xFF
0x00     0x0F     0x00     0xFF
0x00     0x10     0x00     0xFF
0x00     0x11     0x00     0xFF
0x00     0x12     0x00     0xFF
0x00     0x13     0x00     0xFF
0x00     0x14     0x00     0xFF
0x00     0x15     0x00     0xFF
0x8F     0x15     0x00     0xFF
0x8F     0x16     0x00     0xFF

0x8F     0x00     0x7A     0xFF                                    Run key schedule

0xFA     0x00     0x00     0xFF
0xFA     0x16     0x00     0xFF

0xFA     0x00     0xE5     0xFF                                    Produce 250 bytes

0x00     0x00     0x00     0xFF

0x00     0xCB     0x00     0xFF                                    First output byte

0x01     0xCB     0x00     0xFF
```

```
0x01    0xCC    0x00    0xFF                          Second output byte
0x02    0xCC    0x00    0xFF
0x02    0xCE    0x00    0xFF                          Third output byte
0x03    0xCE    0x00    0xFF
0x03    0x75    0x00    0xFF
0x04    0x75    0x00    0xFF
0x04    0x7B    0x00    0xFF
0x05    0x7B    0x00    0xFF
0x05    0x99    0x00    0xFF
0x06    0x99    0x00    0xFF
0x06    0xBD    0x00    0xFF
0x07    0xBD    0x00    0xFF
0x07    0x79    0x00    0xFF
0x07    0x79    0x00    0x55                          Terminate
Total Cycles: 13332
```

The last line of output shows 13,232 cycles, which is a long time when we realize that a single key stream byte can be produced by the hardware within a single clock cycle. How hard is it to determine intermediate time-stamps on the execution of this program? While some instruction-set simulators provide direct support for this, we will need to develop a small amount of support code to answer this question. We will introduce an additional coprocessor command which, when observed by the `triviumitf` module, will display the current cycle count. This is a debug-only command, similar to the `terminate` call in the 8051 software.

The modifications for such a command to the code are minimal. In the C code, we add a function to call when we would like to see the current cycle count.
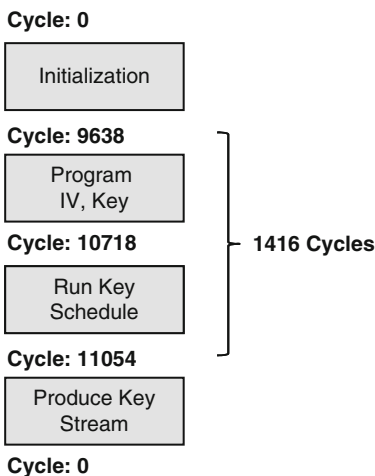
```
void showcycle() {
  P1 = 0x20; P1 = 0x0;
}
```

In the GEZEL code, we extend the `triviumitf` with a small FSM to execute the new command.

```
dp triviumitf {
  reg rupins : ns(8);
  ...
  always {
    ...
    rupins    = upins;
  }
  sfg show {
    $display("Cycle: ", $cycle);
  }
  sfg idle { }
}
fsm f_triviumitf(triviumitf) {
  initial s0;
  state s1;

  @s0 if (rupins == 0x20) then (show) -> s1;
                          else (idle) -> s0;
```

Fig. 11.7 Performance
measurement of Trivium
coprocessor

Cycle: 0

Initialization

Cycle: 9638

Program
IV, Key

Cycle: 10718

Run Key
Schedule

Cycle: 11054

Produce Key
Stream

Cycle: 0

1416 Cycles

```
  @s1 if (rupins == 0x00) then (idle) -> s0;
                          else (idle) -> s1;
}
```

Each time `showcycle()` executes, the current cycle count will be printed by
GEZEL. This particular way of measuring performance has a small overhead (88
cycles per call to `showcycle()`). We add the command in the C code at the fol-
lowing places.

- In the `main` function, just before programming the first key byte.
- In the `main` function, just before starting the key schedule.
- In the `main` function, just before starting the key stream.

Figure 11.7 illustrates the resulting cycle counts obtained from the simulation.
The output shows that most time is spent in startup (initialization of the micro-
controller), and that the software-hardware interaction, as expected, is expensive in
cycle-cost. For example, programming a new key and re-running the key schedule
costs 1416 cycles, almost ten times as long as what is really needed by the hard-
ware (143 cycles). This stresses once more the importance of carefully considering
hardware–software interactions during the design.

11.3 Trivium for 32-bit Platforms

Our second Trivium coprocessor integrates the algorithm on a 32-bit StrongARM
processor. We will compare two integration strategies: a memory-mapped interface
and a custom-instruction interface. Both scenario's are supported through library

modules in the GEZEL kernel. The hardware kernel follows the same ideas as before. By unrolling a trivium kernel 32 times, we obtain a module that produces 32 bits of keystream material per clock cycle. After loading the key and initialization vector, the key schedule of such a module has to execute for $4 * 288/32 = 36$ clock cycles before the first word of the keystream is available.

11.3.1 Hardware Platform Using Memory-mapped Interfaces

Figure 11.8 shows the control shell design for a Trivium kernel integrated to a 32-bit memory-mapped interface. There are four memory-mapped registers involved: din, dout, control, and status. In this case, the key stream is directly read by the processor. The Trivium kernel follows the design we discussed earlier in Sect. 11.1.3. There is one additional control input, go, which is used to control the update of the state register. Instead of having a free-running Trivium kernel, the update of the state register will be strictly controlled by software, so that the entire keystream is captured using read operations from a memory-mapped interface.

As with other memory-mapped interfaces, our first task is to design a control shell to drive the Trivium kernel. We start with the command set. The command set must be able to load a key, an initialization vector, run the key schedule, and retrieve a single word from the key stream. Figure 11.9 illustrates the command set for this coprocessor.

The control memory-mapped register has a dual purpose. It transfers an instruction opcode as well as a parameter. The parameter indicates the part of the key or initial value which is being transferred. The parameter is 0, 1, or 2, because 3 words are sufficient to cover the 80 bits from the stream cipher key or the stream cipher initial value. The ins_idle instruction has the same purpose as before: it
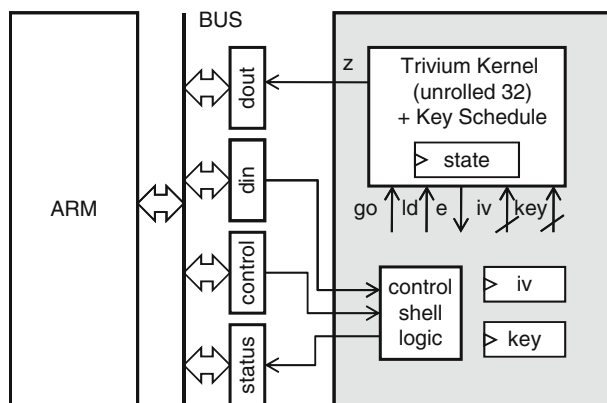

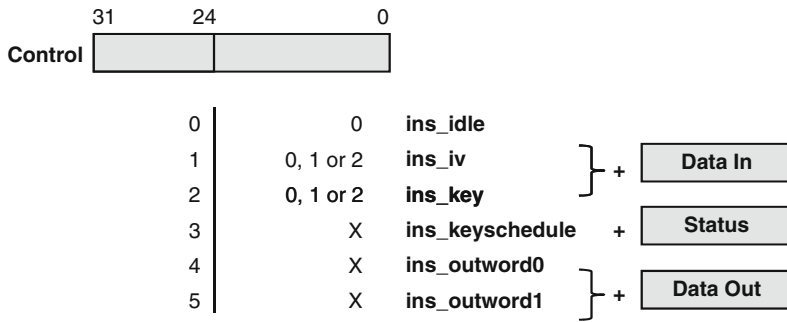**Fig. 11.8** Memory-mapped integration of Trivium on a 32-bit processor

Fig. 11.9 Command set for a memory-mapped Trivium coprocessor

is used to synchronize the transfer of data operands with instructions. There are two commands to retrieve keystream bits from the coprocessor: ins_outword0 and ins_outword1. Both of these transfer a single word from the stream cipher dout, and they are used alternately in order to avoid sending dummy ins_idle to the coprocessor.

Listing 11.8 Hardware platform for the StrongARM coprocessor

```
1   ipblock myarm {                                          ARM Core
2     iptype "armsystem";
3     ipparm "exec=trivium";
4     ipparm "period=1";
5   }
6   ipblock armdout(in data : ns(32)) {                      ARM interfaces
7     iptype "armsystemsink";
8     ipparm "core=myarm";
9     ipparm "address=0x80000000";
10  }
11  ipblock armdin(out data : ns(32)) {
12    iptype "armsystemsource";
13    ipparm "core=myarm";
14    ipparm "address=0x80000004";
15  }
16  ipblock armstatus(in data : ns(32)) {
17    iptype "armsystemsink";
18    ipparm "core=myarm";
19    ipparm "address=0x80000008";
20  }
21  ipblock armcontrol(out data : ns(32)) {
22    iptype "armsystemsource";
23    ipparm "core=myarm";
24    ipparm "address=0x8000000C";
25  }
26
```

```
27  dp triviumitf(in  din    : ns(32);                        ┌──────────────────────┐
28               out dout    : ns(32);                        │ Trivium control shell│
29               in  ctl     : ns(32);                        └──────────────────────┘
30               out status  : ns(32)) {
31    sig ld      : ns(1);
32    sig go      : ns(1);
33    sig iv, key : ns(80);
34    sig e       : ns(1);
35    sig z       : ns(32);

36    use triviumtop(ld, go, iv, key, z, e);                  ┌──────────────────┐
                                                              │ Trivium kernel   │
37    reg ivr, keyr       : ns(80);                           └──────────────────┘
38    sig ivr0, ivr1, ivr2 : ns(32);
39    sig key0, key1, key2 : ns(32);
40    reg oldread : ns(3);
41
42    always {
43      iv  = ivr;
44      key = keyr;
45
46      // program new IV
47      ivr0= ((ctl[24:26] == 0x1) & (ctl[0:1] == 0x0)) ? din : ivr[31: 0];
48      ivr1= ((ctl[24:26] == 0x1) & (ctl[0:1] == 0x1)) ? din : ivr[63:32];
49      ivr2= ((ctl[24:26] == 0x1) & (ctl[0:1] == 0x2)) ? din : ivr[79:64];
50      ivr = ivr2 # ivr1 # ivr0;
51
52      // program new KEY
53      key0= ((ctl[24:26] == 0x2) & (ctl[0:1] == 0x0)) ? din : keyr[31: 0];
54      key1= ((ctl[24:26] == 0x2) & (ctl[0:1] == 0x1)) ? din : keyr[63:32];
55      key2= ((ctl[24:26] == 0x2) & (ctl[0:1] == 0x2)) ? din : keyr[79:64];
56      keyr= key2 # key1 # key0;
57
58      // control start
59      ld  = ((ctl[24:26] == 0x3) ? 1 : 0);
60
61      // read status
62      status = e;
63
64      // read output data
65      dout= z;
66
67      // trivium control
68      oldread = ((ctl[24:26]));
69      go  = ((ctl[24:26] == 0x4) & (oldread ==0x5)) |
70            ((ctl[24:26] == 0x5) & (oldread ==0x4)) |
71            ((ctl[24:26] == 0x3) & (oldread ==0x0));
72    }
73  }
74
75  dp triviumsystem {
76    sig din, dout, ctl, status : ns(32);
77    use myarm;
78    use triviumitf(din, dout, ctl, status);
79    use armdin(din);
```

```
80    use armdout(dout);
81    use armstatus(status);
82    use armcontrol(ctl);
83  }
84  system S {
85    triviumsystem;
86  }
```

Listing 11.8 shows the design of the control shell module. The design of the
Trivium kernel is not shown in this listing, although a very similar design can be
found in Listing 11.4. The first part of Listing 11.8, Line 1–25, shows the memory-
mapped interface to the ARM core. This includes instantiation of the core (Line
1–5), and four memory-mapped registers (Line 6–25). The bulk of the code, Line
25–73, contains the control shell for the Trivium kernel. The kernel is instantiated on
Line 36. The registers for key and initial value, defined on Line 33, are programmed
from software through a series of simple decode steps (Line 46–56). The encoding
used by the control memory mapped register corresponds to Fig. 11.9.

The control pins of the Trivium kernel (ld, go) are programmed by means of
simple decoding steps as well (Line 59, 67–71). Note that the go pin is driven by a
pulse of a single clock cycle, rather than a level programmed from software. This is
done by detecting the exact cycle when the value of the control memory mapped
interface changes. Note that the overall design of this control shell is quite simple,
and does not require complex control or a finite state machine. Finally, the system
integration consists of interconnecting the control shell and the memory-mapped
interfaces (Line 75–86).

11.3.2 Software Driver Using Memory-mapped Interfaces

A software driver for the memory-mapped Trivium coprocessor is shown in Listing
11.9. The driver programs the initial value and key, runs the key schedule, and next
receives 512 words of keystream. The state update of the Trivium coprocessor is
controlled by alternately writing 4 and 5 to the command field of the control
memory-mapped interface. This is done during the key schedule (Line 28–32) as
well as during the keystream generation (Line 34–39).

The code also contains an external system call getcyclecount(). This is
a simulator-specific call, in this case specific to SimIt-ARM, to return the current
cycle count of the simulation. By inserting such calls in the driver code (in this case,
on Line 27, 33, 40), we can obtain the execution time of selected phases of the
keystream generation.

To execute the system simulation, we compile the software driver, and run the
GEZEL hardware module and the software executable in gplatform. The simula-
tion output shows the expected keystream bytes: 0xcc, 0xce,0x75, ... The output

Listing 11.9 StrongARM software driver for the memory-mapped Trivium coprocessor

```
1  extern unsigned long long getcyclecount();
2
3  int main() {
4    volatile unsigned int *data   = (unsigned int *) 0x80000004;
5    volatile unsigned int *ctl    = (unsigned int *) 0x8000000C;
6    volatile unsigned int *output = (unsigned int *) 0x80000000;
7    volatile unsigned int *status = (unsigned int *) 0x80000008;
8
9    int i;
10   unsigned int stream[512];
11   unsigned long long c0, c1, c2;
12
13   // program iv
14   *ctl  = (1 << 24);        *data = 0;    // word 0
15   *ctl  = (1 << 24) | 0x1;                // word 1
16   *ctl  = (1 << 24) | 0x2;                // word 2
17
18   // program key
19   *ctl  = (2 << 24);        *data = 0x80; // word 0
20   *ctl  = (2 << 24) | 0x1; *data = 0;     // word 1
21   *ctl  = (2 << 24) | 0x2;                // word 2
22
23   // run the key schedule
24   *ctl  = 0;
25   *ctl  = (3 << 24);  // start pulse
26
27   c0 = getcyclecount();
28   while (! *status) {
29     *ctl = (4 << 24);
30     if (*status) break;
31     *ctl = (5 << 24);
32   }
33   c1 = getcyclecount();
34   for (i=0; i<256; i++) {
35     stream[2*i] = *output;
36     *ctl = (4 << 24);
37     stream[2*i+1] = *output;
38     *ctl = (5 << 24);
39   }
40   c2 = getcyclecount();
41
42   for (i=0; i<16; i++) {
43     printf("%8x ", stream[i]);
44     if (!((i+1) % 8))
45       printf("\n");
46   }
47   printf("key schedule cycles:",
48          " %lld  stream cycles: %lld\n",
49          c1 - c0, c2 - c1);
50
51   return 0;
52 }
```

also shows that the key schedule completed in 435 cycles, and that 512 words of
keystream were generated in 10,524 cycles.

```
>arm-linux-gcc -static trivium.c cycle.s -o trivium
>gplatform trivium32.fdl
core myarm
armsystem: loading executable [trivium]
armsystemsink: set address 2147483648
armsystemsink: set address 2147483656
ccce757b ccce757b 99bd7920 9a235a88 1251fc9f aff0a655 7ec8ee4e
bfd42128
86dae608 806ea7eb 58aec102 16fa88f4 c5c3aa3e b1bcc9f2 bb440b3f
c4349c9f
key schedule cycles: 435   stream cycles: 10524
Total Cycles: 269540
```

We now analyze the performance results for this design. As the Trivium kernel
used in this design is unrolled 32 times (and thus can produce a new word every
clock cycle), 512 words in 10,524 clock cycles is not a stellar result. Each word
requires around 20 clock cycles. This includes synchronization of software and
hardware, transfer of a result, writing that result into memory, and managing the
loop counter and address generation (lines 34–39 in Listing 11.9). However, an-
other way to phrase the performance question is: how much better is this result
compared to an optimized full-software implementation? To answer this question,
we can port an available, optimized implementation to the StrongARM and make
a similar profiling. We used the implementation developed by Trivium's authors,
C. De Canniere, in this profiling experiment, and found that this implementation
takes 3,810 cycles for key schedule and 48,815 cycles for generating 512 words.
Thus, each word of the keystream requires close to 100 clock cycles on the ARM.
Therefore, we conclude that the hardware coprocessor is still five times faster
compared to an optimized software implementation, although the hardware co-
processor has an overhead factor of 20 times compared to a standalone hardware
implementation.

As we wrote the hardware from scratch, one may wonder if it would not have
been easier to try to *port* the Trivium software implementation into hardware. In
practice, this may be hard to do, because the optimizations one does for software
are very different than the optimizations one does for hardware. As an example,
Listing 11.10 shows part of the software-optimized Trivium implementation of
De Canniere. This implementation was written with 64-bit execution in mind.
Clearly, the efficient translation of this code into hardware is quite difficult, since
the specification does not have the same clarity compared to the algorithm defini-
tion we discussed at the start of the chapter.

This completes our discussion of the memory-mapped Trivium coprocessor de-
sign. In the next section, we consider a third type of hardware/software interface
for the Trivium kernel: the mapping of Trivium into custom instructions on a 32-bit
processor.

Listing 11.10 Optimized software implementation for Trivium

```
// Support Macro's
#define U32TO8_LITTLE(p, v) (((u32*)(p))[0] = U32TO32_LITTLE(v))
#define U8TO32_LITTLE(p) U32TO32_LITTLE((((u32*)(p)))[0])
#define U32TO32_LITTLE(v) (v)

#define Z(w) (U32TO8_LITTLE(output + 4 * i, \
              U8TO32_LITTLE(input + 4 * i) ^ w))
#define S(a, n) (s##a##n)
#define T(a) (t##a)

#define S00(a, b) ((S(a, 1)<<( 32-(b))))
#define S32(a, b) ((S(a, 2)<<( 64-(b)))|(S(a, 1)>>((b)-32)))
#define S64(a, b) ((S(a, 3)<<( 96-(b)))|(S(a, 2)>>((b)-64)))
#define S96(a, b) ((S(a, 4)<<(128-(b)))|(S(a, 3)>>((b)-96)))

#define UPDATE()                                                    \
  do {                                                              \
    T(1) = S64(1,  66) ^ S64(1,  93);                               \
    T(2) = S64(2,  69) ^ S64(2,  84);                               \
    T(3) = S64(3,  66) ^ S96(3, 111);                               \
                                                                    \
    Z(T(1) ^ T(2) ^ T(3));                                          \
                                                                    \
    T(1) ^= (S64(1,  91) & S64(1,  92)) ^ S64(2,  78);              \
    T(2) ^= (S64(2,  82) & S64(2,  83)) ^ S64(3,  87);              \
    T(3) ^= (S96(3, 109) & S96(3, 110)) ^ S64(1,  69);              \
  } while (0)

#define ROTATE()                                                    \
  do {                                                              \
    S(1, 3) = S(1, 2); S(1, 2) = S(1, 1); S(1, 1) = T(3);     \
    S(2, 3) = S(2, 2); S(2, 2) = S(2, 1); S(2, 1) = T(1);     \
    S(3, 4) = S(3, 3); S(3, 3) = S(3, 2); S(3, 2) = S(3, 1); \
    S(3, 1) = T(2);                                                 \
  } while (0)


// ...

// This is the Trivium keystream generation loop

  for (i = 0; i < msglen / 4; ++i)
    {
      u32 t1, t2, t3;

      UPDATE();
      ROTATE();
    }
```

11.3.3 Hardware Platform Using a Custom-Instruction Interface

The integration of a Trivium coprocessor as a custom datapath in a processor requires a processor that supports custom-instruction extensions. As discussed in Chap. 9, this has a strong impact on the tools that come with the processor. In this example, we will make use of the custom-instruction interface of the StrongARM processor discussed in Sect. 9.5.1. Figure 11.10 shows the design of a Trivium Kernel integrated into two custom-instruction interfaces, an `OP3X1` and an `OP2X2`. The former is an instruction that takes three 32-bit operands and produces a single 32-bit result. The latter is an instruction that takes two 32-bit operands and produces two 32-bit results.

During normal operation, the trivium state is fed through two Trivium kernels which each provide 32-bit of keystream. These two words form the results of an `OP2x2` instruction. The same `OP2x2` instruction also controls the update of the Trivium state. This way, each custom `OP2x2` instruction advances the Trivium
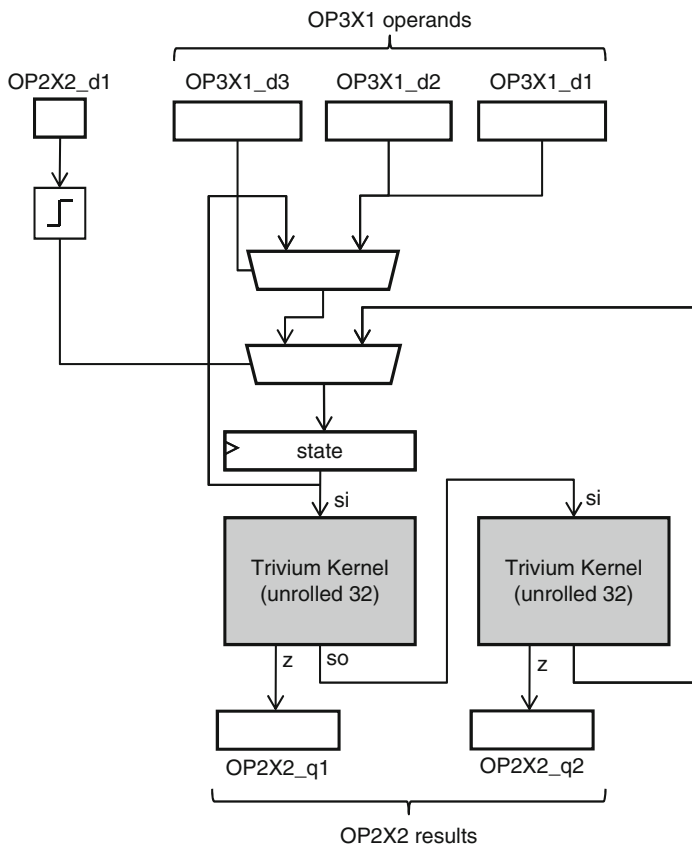
Fig. 11.10 Custom-instruction integration of Trivium on a 32-bit processor

algorithm for 1 step, producing 64 bits of keystream. When the Trivium algorithm is
not advancing, the state register can be reprogrammed by means of OP3x1 instruc-
tions. The third operand of OP3x1 selects which part of the 288-bit state register
will be modified. The first and second operands contain 64 bit of state register data.
The result of the OP3x1 instruction is ignored.

Thus, both programming and keystream retrieval can be done using a bandwidth
of 64 bits, which is larger than the memory-mapped interface. Hence, we can expect
a speedup over the previous implementation. Listing 11.11 shows a GEZEL listing
for this design. As before, we have left out the Trivium kernel which is similar to
the one used in Listing 11.4.

The interface with the ARM is captured on line 1–16, and this is followed by
the Trivium control shell on line 18–57. The Trivium state register is represented
as nine registers of 32 bit rather then a single 288-bit register. Two 32-bit Trivium
kernels are instantiated on line 33 and 34. The state register update is controlled
by the adv control flag, as well as the value of the third operand of the OP3X1
instruction (line 37–45). The output of the Trivium kernels is fed into the result of
the OP2X2 instruction (line 51–52). Finally, the adv flag is created by detecting an
edge in the OP2x2 operand (line 54–55). In practice, this means that two calls to
OP2X2 are needed to advance Trivium one step.

Listing 11.11 Integration of Trivium as two custom-instructions on a 32-bit processor

```
1   ipblock myarm {                                                    ARM core
2     iptype "armsystem";
3     ipparm "exec=trivium";
4   }
5   ipblock armsfu1(out d1, d2 : ns(32);                          ARM interfaces
6                   in  q1, q2 : ns(32)) {
7     iptype "armsfu2x2";
8     ipparm "core = myarm";
9     ipparm "device = 0";
10  }
11  ipblock armsfu2(out d1, d2, d3 : ns(32);
12                  in  q1         : ns(32)) {
13    iptype "armsfu3x1";
14    ipparm "core = myarm";
15    ipparm "device = 0";
16  }
17
18  dp triviumsfu {                                           Trivium control shell
19    sig o2x2_d1, o2x2_d2, o2x2_q1, o2x2_q2 : ns(32);
20    sig o3x1_d1, o3x1_d2, o3x1_d3, o3x1_q1 : ns(32);
21    use armsfu1( o2x2_d1, o2x2_d2, o2x2_q1, o2x2_q2);
22    use armsfu2( o3x1_d1, o3x1_d2, o3x1_d3, o3x1_q1);
23    use myarm;
24
25    reg w1, w2, w3, w4 : ns(32);
26    reg w5, w6, w7, w8 : ns(32);
27    reg w9             : ns(32);
28    reg tick           : ns(1);
```

```
29    sig adv            : ns(1);
30    sig si0, si1       : ns(288);
31    sig so0, so1       : ns(288);
32    sig z0,  z1        : ns(32);
33    use trivium320(si0, so0, z0);                          Trivium kernel
34    use trivium321(si1, so1, z1);                          Trivium kernel
35
36    always {
37      w1 = adv ? so1[  0: 31] : ((o3x1_d3 == 1) ? o3x1_d1 : w1);
38      w2 = adv ? so1[ 32: 63] : ((o3x1_d3 == 1) ? o3x1_d2 : w2);
39      w3 = adv ? so1[ 64: 95] : ((o3x1_d3 == 2) ? o3x1_d1 : w3);
40      w4 = adv ? so1[ 96:127] : ((o3x1_d3 == 2) ? o3x1_d2 : w4);
41      w5 = adv ? so1[128:159] : ((o3x1_d3 == 3) ? o3x1_d1 : w5);
42      w6 = adv ? so1[160:191] : ((o3x1_d3 == 3) ? o3x1_d2 : w6);
43      w7 = adv ? so1[192:223] : ((o3x1_d3 == 4) ? o3x1_d1 : w7);
44      w8 = adv ? so1[224:255] : ((o3x1_d3 == 4) ? o3x1_d2 : w8);
45      w9 = adv ? so1[256:287] : ((o3x1_d3 == 5) ? o3x1_d1 : w9);
46      o3x1_q1 = 0;
47
48      si0 = w9 # w8 # w7 # w6 # w5 # w4 # w3 # w2 # w1;
49      si1 = so0;
50
51      o2x2_q1 = z0;
52      o2x2_q2 = z1;
53
54      tick = o2x2_d1[0];
55      adv  = (tick != o2x2_d1[0]);
56    }
57 }
58
59 system S {
60    triviumsfu;
61 }
```

11.3.4 Software Driver for a Custom-Instruction Interface

Listing 11.12 shows a software driver for the Trivium custom-instruction processor
that generates a keystream of 512 words in memory. The driver starts by loading
key and data (line 25–30), running the key schedule (line 34–37), and retrieving
the keystream (line 41–48). At the same time, the getcyclecount system call is
used to determine the performance of the key schedule and the keystream genera-
tion part.

Listing 11.12 Custom-instruction software driver for the Trivium ASIP
```
1  #include <stdio.h>
2  #define OP2x2_1(D1,D2,S1,S2) \
3      asm volatile ("smullnv %0, %1, %2, %3": \
4          "=&r"(D1),"=&r"(D2): \
5          "r"(S1),"r"(S2));
6
7  #define OP3x1_1(D1, S1, S2, S3) \
```

```
 8      asm volatile ("mlanv %0, %1, %2, %3": \
 9          "=&r"(D1): "r"(S1), "r"(S2), "r"(S3)); \
10
11   extern unsigned long long getcyclecount();
12
13   int main() {
14     int z1, z2, i;
15     unsigned int stream[512];
16     unsigned long long c0, c1, c2;
17
18     int key1 = 0x80;
19     int key2 = 0xe0000000;
20
21     // clear 'tick'
22     OP2x2_1(z1, z2, 0, 0);
23
24     // load key = 80 and IV = 0
25     OP3x1_1(z1,key1, 0, 1);
26     OP3x1_1(z1,   0, 0, 2);
27     OP3x1_1(z1,   0, 0, 3);
28     OP3x1_1(z1,   0, 0, 4);
29     OP3x1_1(z1,key2, 0, 5);
30     OP3x1_1(z1,   0, 0, 0);
31
32     // run key schedule
33     c0 = getcyclecount();
34     for (i=0; i<9; i++) {
35       OP2x2_1(z1, z2, 1, 0);
36       OP2x2_1(z1, z2, 0, 0);
37     }
38     c1 = getcyclecount();
39
40     // run keystream
41     for (i=0; i<128; i++) {
42       OP2x2_1(z1, z2, 1, 0);
43       stream[4*i]   = z1;
44       stream[4*i+1] = z2;
45       OP2x2_1(z1, z2, 0, 0);
46       stream[4*i+2] = z1;
47       stream[4*i+3] = z2;
48     }
49     c2 = getcyclecount();
50
51     for (i=0; i<16; i++) {
52       printf("%8x ", stream[i]);
53       if (!((i+1) % 8))
54         printf("\n");
55     }
56     printf("key schedule cycles:",
57            "%lld  stream cycles: %lld\n",
58            c1 - c0, c2 - c1);
59
60     return 0;
61   }
```

The algorithm can be compiled with the ARM cross-compiler and simulated on top of GEZEL `gplatform`. This results in the following output.

```
>arm-linux-gcc -static trivium.c cycle.s -o trivium
>gplatform triviumsfu.fdl
core myarm
armsystem: loading executable [trivium]
ccce757b 99bd7920 9a235a88 1251fc9f aff0a655 7ec8ee4e bfd42128
86dae608
806ea7eb 58aec102 16fa88f4 c5c3aa3e b1bcc9f2 bb440b3f c4349c9f
be0a7e3c
key schedule cycles: 289  stream cycles: 8862
Total Cycles: 42688
```

We can verify that, as before, the correct keystream is generated. The cycle count of the algorithm is significantly smaller than before: the key schedule completes in 289 cycles, and the keystream is generated within 8,862 cycles. This implies that each word of keystream required around 17 cycles. If we turn on the `O3` flag while compiling the driver code, we obtain 67 and 1,425 clock cycles for key schedule and keystream, respectively, implying that each word of the keystream requires less than three cycles! Hence, we conclude that for this design, an ASIP interface is significantly more efficient than a memory-mapped interface.

11.4 Summary

In this chapter, we designed a stream cipher coprocessor for three different hosts: a small 8-bit microcontroller, a 32-bit SoC processor, and a 32-bit ASIP. In each of these cases, we created a control shell to match the coprocessor to the available hardware–software interface. The stream cipher algorithm was easy to scale over different word-lengths by simply unrolling the algorithm. The performance evaluation results of all these implementations are captured in Table 11.2. These results demonstrate two points. First, it is not easy to achieve the performance of raw hardware. All of the coprocessors are limited by their hardware/software interface or the

Table 11.2 Performance evaluation of trivium coprocessors on multiple platforms

Platform	Hardware	8051	Hardware	StrongARM	Unit
Interface	Native	Port-mapped	Native	Memory mapped	
Wordlength	8	8	32	32	bit
Key schedule	144	1416	36	435	cycles
Key stream	4	6.7	1	20.5	cycles/word
Platform	StrongARM	StrongARM	StrongARM		Unit
Interface	SW	ASIP	ASIP (-O3)		
Wordlength	32	64	64		bit
Key schedule	3810	289	67		cycles
Key stream	95	17	3		cycles/word

speed of software on the host, not by the computational limits of the hardware copro-
cessors. Second, the wide variation of performance results underline the importance
of a carefully designed control shell, and a careful consideration of the application
when selecting a hardware/software interface.

11.5 Further Reading

The standard reference of cryptographic algorithms is by Menezes, van Oorschot,
and Vanstone Menezes et al. (2001). Of course, cryptography is a fast-moving
field. The algorithm described in this section was developed for the eStream
Project ECRYPT (2008) in 2005. The Trivium specifications are by De Canniere
De Canniere and Preneel (2005). The Trivium webpage in the eStream website
describes several other hardware implementations of Trivium.

11.6 Problems

11.1. Design a control shell for the Trivium algorithm on top of a *Fast Simplex Link*
interface. Please refer to Sect. 9.4.2 for a description of the FSL timing and the FSL
protocol. Assume the following interface for your module.

```
dp trivium_fsl(in  idata   : ns(32);  // input slave interface
               in  exists  : ns(1);
               out read    : ns(1);
               out odata   : ns(32);  // output master interface
               in  full    : ns(1);
               out write   : ns(1))
```

11.2. Consider a simple linear feedback shift register, defined by the following
polynomial: $g(x) = x^{35} + x^2 + 0$. A possible hardware implementation of this
LFSR is shown in Fig. 11.11. This polynomial is *primitive*, which implies that the
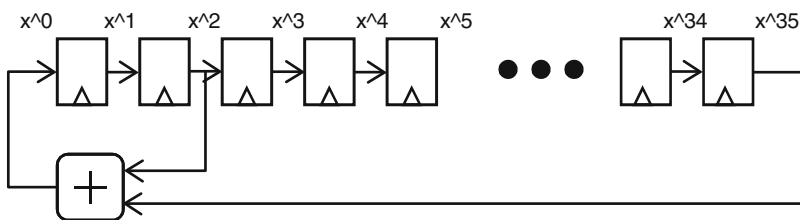
Fig. 11.11 LFSR for $g(x) = x^{35} + x^2 + 0$

LFSR will generate a so-called *m-sequence*: for a given initialization of the registers, the structure will cycle through all possible $2^{35} - 1$ states before returning to the same state.

(a) Write an optimized software implementation of an LFSR generator that calculates the first 1,024 states starting from the initialization $x^{32} = x^{33} = x^{34} = x^{35} = 1$ and all other bits 0. For each state you need to store only the first 32 bits.

(b) Write an optimized standalone hardware implementation of an LFSR generator that calculates the first 1,024 states starting from the initialization $x^{32} = x^{33} = x^{34} = x^{35} = 1$ and all other bits 0. You do not need to store the first 32 bits, but can feed them directly to an output port.

(c) Design a control shell for the module you have designed under (b), and use a memory-mapped interface to capture and store the first 1,024 states of the LFSR. You only need to capture the first 32 bits of each state. Compare the resulting performance to the solution of (a).

(d) Design a control shell for the module you have designed under (b), and use a custom-instruction interface to capture and store the first 1,024 states of the LFSR. You only need to capture the first 32 bits of each state. Compare the resulting performance to the solution of (a).

Chapter 12
CORDIC Coprocessor

Abstract The Coordinate Rotation Digital Computer Algorithm (CORDIC for short) is a well known algorithm to perform rotations using simple, integer arithmetic. The algorithm implements a conversion between rectangular (X, Y) coordinates and polar (r, θ) coordinates. In this chapter, we discuss the design of a coprocessor that implements the CORDIC algorithm. We will use a Fast-Simplex-Link (FSL) interface. We also discuss a prototype implementation of the design on a Spartan 3E Starter Kit, and show how to resolve the communication bottleneck occuring from an inefficient hardware/software interface.

12.1 The Coordinate Rotation Digital Computer Algorithm

In this section we introduce the CORDIC algorithm, including a reference implementation in C.

12.1.1 The Algorithm

The CORDIC algorithm calculates the rotation of a two-dimensional vector x_0, y_0 over an arbitrary angle α. Figure 12.1a describes the problem of coordinate rotation. Given (x_0, y_0) and a rotation angle α, the coordinates (x_T, y_T) are given by:

$$\begin{bmatrix} x_T \\ y_T \end{bmatrix} = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}. \tag{12.1}$$

This rotation can be written in terms of a single function $\tan \alpha$ by using

$$\cos \alpha = \frac{1}{\sqrt{1 + \tan^2 \alpha}} \tag{12.2}$$

$$\sin \alpha = \frac{\tan \alpha}{\sqrt{1 + \tan^2 \alpha}}. \tag{12.3}$$
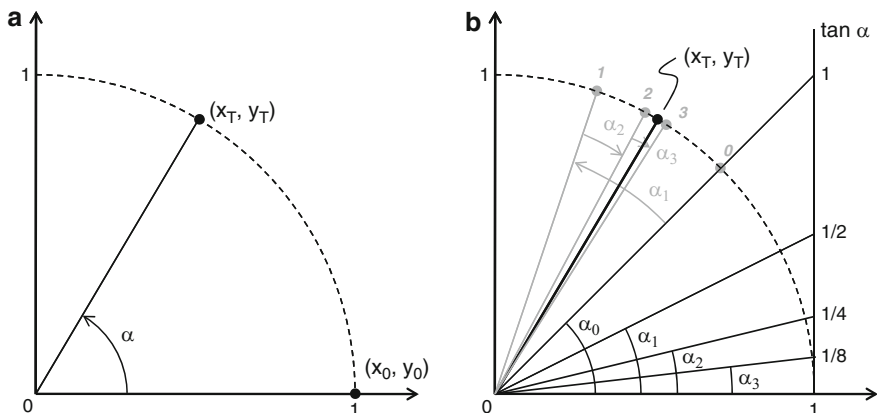
Fig. 12.1 (**a**) Coordinate Rotation over α (**b**) Decomposition of the rotation angle $\alpha = \alpha_0 + \alpha_1 - \alpha_2 - \alpha_3$

The resulting coordinate rotation now becomes

$$\begin{bmatrix} x_T \\ y_T \end{bmatrix} = \frac{1}{\sqrt{1 + \tan^2 \alpha}} \begin{bmatrix} 1 & -\tan \alpha \\ \tan \alpha & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}. \tag{12.4}$$

The clever part of the CORDIC algorithm is that the rotation over the angle α can be expressed in terms of rotations over smaller angles. The CORDIC algorithm chooses a decomposition in angles whose tangent is a power of 2, as illustrated in Fig. 12.1b. Thus, we choose a set of angles α_i so that

$$\tan \alpha_i = \frac{1}{2^i}. \tag{12.5}$$

From the figure, we can see that α can be reasonably approximated as $\alpha_0 + \alpha_1 - \alpha_2 - \alpha_3$. Because of the particular property of these angles, Formula 12.4 becomes easy to evaluate: (x_{i+1}, y_{i+1}) can be found using addition, subtraction, and shifting of (x_i, y_i). For example, suppose that we want to rotate *clockwise* over α_i, then

$$x_{i+1} = K_i \left\{ x_i + \frac{y_i}{2^i} \right\} \tag{12.6}$$

$$y_{i+1} = K_i \left\{ \frac{-x_i}{2^i} + y_i \right\}. \tag{12.7}$$

If we want to rotate *counter-clockwise* over α_i, then we use instead

$$x_{i+1} = K_i \left\{ x_i - \frac{y_i}{2^i} \right\} \tag{12.8}$$

$$y_{i+1} = K_i \left\{ \frac{x_i}{2^i} + y_i \right\}. \tag{12.9}$$

In these formulas, K_i is a constant that can be precalculated:

$$K_i = \frac{1}{\sqrt{1 + 2^{-2i}}}. \tag{12.10}$$

We can now approximate an arbitrary, but unknown, angle β by means of a binary-search process as follows. We precalculate the set of angles $\alpha_i = \arctan 2^{-i}$ and store them in a lookup table. Assume our current approximation of β is β_i. If $\beta_i > \beta$, we rotate clockwise and $\beta_{i+1} = \beta_i - \alpha_i$. If $\beta_i < \beta$, we rotate counterclockwise and $\beta_{i+1} = \beta_i + \alpha_i$. We continue this process iteratively until $\beta_n \simeq \beta$. We gain around 1 bit of precision per iteration. For example, after 20 iterations, the precision on the angle is around one part in one million (six significant digits).

12.1.2 Reference Implementation in C

A distinctive property of the CORDIC algorithm is that it can execute using only additions, subtractions, and shift operations. It maps well to integer arithmetic, even though the numbers being handled are still fractional. We will discuss a CORDIC implementation in C that uses scaled `int` types.

Fractional arithmetic can be implemented using integer numbers, by scaling each number by a power of two. The resulting representation is called a `<M,N>` *fixed point representation*. `M` represents the integer wordlength, and `N` the fractional wordlength. For example, a `<32,28>` fixed-point number has a wordlength of 32 bits, and has 28 fractional bits. Thus, the weight of the least significant bit is 2^{-28}. Fixed-point numbers only change the relative weight of the bits in a binary number. They work just like integers in all other respects – you can add, subtract, compare, and shift them. So, a 32-bit unsigned number with value 8834773 will, as a `<32,28>` number, have the value $8834773/2^{28} = 0.3291209...$

Listing 12.1 shows a fixed-point version of a 32-bit CORDIC algorithm, using `<32,28>` fixed point arithmetic. The CORDIC is evaluated using 20 iterations, which means that it can approximate angles with a precision of $\arctan 2^{-20}$, or around one-millionth of a radian. At the start, the program defines a few relevant constants.

- `PI`, the well-known mathematical constant, equals $\pi * 2^{28}$ or 843314856.
- `K_CONST` is the product of the twenty first K_i according to (12.10). This constant factor needs to be evaluated once.
- `angles[]` is an array of constants that holds the angles α_i defined by (12.5). For example, the first element is 210828714, which is a `<32,28>` number corresponding to $atan(1) = 0.78540$.

The cordic function, on lines 13–34, first initalizes the angle accumulator `current`, and the initial vector `(X,Y)`. Next, it goes through 20 iterations where

Listing 12.1 Reference implementation of a fixed-point CORDIC algorithm

```
1   #include <stdio.h>
2   #define K_CONST 163008218 /* 0.60725293510314      */
3   #define PI 843314856      /* 3.141593.. in <32,28> */
4   typedef int fixed;        /* <32,28>               */
5
6   static const int angles[] = {
7     210828714,  124459457,   65760959,   33381289,
8      16755421,    8385878,    4193962,    2097109,
9       1048570,     524287,     262143,     131071,
10        65535,      32767,      16383,       8191,
11         4095,       2047,       1024,        511 };
12
13  void cordic(int target, int *rX, int *rY) {
14    fixed X, Y, T, current;
15    unsigned step;
16    X        = K_CONST;
17    Y        = 0;
18    current = 0;
19    for(step=0; step < 20; step++) {
20      if (target > current) {
21        T              =  X - (Y >> step);
22        Y              = (X >> step) + Y;
23        X              = T;
24        current    += angles[step];
25      } else {
26        T              = X + (Y >> step);
27        Y              = -(X >> step) + Y;
28        X              = T;
29        current    -= angles[step];
30      }
31    }
32    *rX = X;
33    *rY = Y;
34  }
35
36  int main(void) {
37    fixed X, Y, target;
38    fixed accsw, accfsl;
39
40    target = PI / 17;
41    cordic(target, &X, &Y);
42
43    printf("Target %d: (X,Y) = (%d,%d)\n", target, X, Y);
44    return(0);
45  }
```

the angle accumulator is compared with the target angle, and where the vector is rotated clockwise or counterclockwise.

The main function, on lines 36–45, demonstrates the operation of the function with a simple testcase, a rotation of (1, 0) over $\pi/17$. We can compile and run this

program on a PC, or for the SimIT-ARM simulator, and it generates the following output.

```
Target 49606756: (X,Y) = (263864846, 49324815)
```

Indeed, after scaling everything by 2^{28}, we can verify that for the target $\pi/17$, (X,Y) equals $(0.98297, 0.18375)$, or $(cos(\pi/17), sin(\pi/17))$.

To evaluate the performance of this function on an embedded processor, a similar technique as in Sect. 11.3.2 can be used. Measurement of the execution time for `cordic` on Simit-ARM yields 485 cycles (-O3 compiler optimization) per call. In the next section, we will develop a hardware implementation of the CORDIC algorithm. Next, we will integrate this hardware design as a coprocessor to the software.

12.2 A Hardware Coprocessor for CORDIC

We'll develop a hardware implementation of the CORDIC design presented in the previous section. The objective is to create a coprocessor, and the first step is to create a hardware kernel to implement CORDIC. Next, we convert the kernel into a coprocessor design. As before, selecting the right hardware/software interface among all that are available, is a crucial design decision. In this case, we intend to map the design onto an FPGA, and the selection is constrained by what is available in the FPGA design environment. We will be making use of the Fast Simplex Link interface discussed in Sect. 9.4.2.

12.2.1 A CORDIC Kernel in Hardware

Listing 12.2 illustrates a hardware CORDIC kernel. In anticipation of using the FSL-based interface, the input/output protocol of the algorithm uses two-way handshake interfaces. The input uses a slave interface, while the output implements a master interface. The computational part of the algorithm is in `sfg iterate`, lines 30–38. This `iterate` instruction is very close to the inner-loop of the `cordic` function in Listing 12.1, including the use of a lookup table `angles` to store the rotation angles. Note that while GEZEL supports lookup tables, it does not support read/write arrays.

Listing 12.2 A Standalone hardware implementation of the CORDIC algorithm
```
1  dp cordic_fsmd    (in  rdata   : tc(32);  // interface to slave
2                     in  exists  : ns(1);
3                     out read    : ns(1);
4                     out wdata   : tc(32);  // interface to master
5                     in  full    : ns(1);
6                     out write   : ns(1)) {
7    lookup angles : tc(32) = {
8      210828714,    124459457,    65760959,    33381289,
```

```
 9      16755421,      8385878,      4193962,      2097109,
10       1048570,       524287,       262143,       131071,
11         65535,        32767,        16383,         8191,
12          4095,         2047,         1024,          511 };
13    reg X, Y, target, current: tc(32);
14    reg step                 : ns( 5);
15    reg done, rexists, rfull : ns( 1);
16    sig cmp : ns(1);
17    always        { rexists = exists;
18                    rfull   = full; }
19    sfg dowrite   { write   = 1; }
20    sfg dontwrite { write   = 0;
21                    wdata   = 0; }
22    sfg doread    { read    = 1; }
23    sfg dontread  { read    = 0; }
24    sfg capture   { step    = 0;
25                    done    = 0;
26                    current = 0;
27                    X       = 163008218;  // K
28                    Y       = 0;
29                    target  = rdata;      }
30    sfg iterate   { step    = step + 1;
31                    done    = (step == 19);
32                    cmp     = (target > current);
33                    X       = cmp ? X - (Y >> step):
34                                    X + (Y >> step);
35                    Y       = cmp ? Y + (X >> step):
36                                    Y - (X >> step);
37                    current = cmp ? current + angles(step):
38                                    current - angles(step); }
39    sfg writeX    { wdata = X; }
40    sfg writeY    { wdata = Y; }
41
42  }
43  fsm fsm_cordic_fsmd(cordic_fsmd) {
44    initial s0;
45    state s1, s2, s22;
46    state c1;
47
48    // wait for SW to write slave
49    @s0 if (rexists) then (capture , doread, dontwrite)   -> c1;
50                     else (dontread, dontwrite)           -> s0;
51
52    // calculate result
53    @c1 if (done)    then (dontread, dontwrite)           -> s1;
54                     else (iterate, dontread, dontwrite)  -> c1;
55
56    // after read op completes, do a write to the master
57    @s1 if (rfull)   then (dontread, dontwrite)           -> s1;
58                     else (dowrite , writeX, dontread )   -> s2;
59    @s2 if (rfull)   then (dontread, dontwrite)           -> s2;
60                     else (dowrite , writeY, dontread )   -> s0;
61  }
```
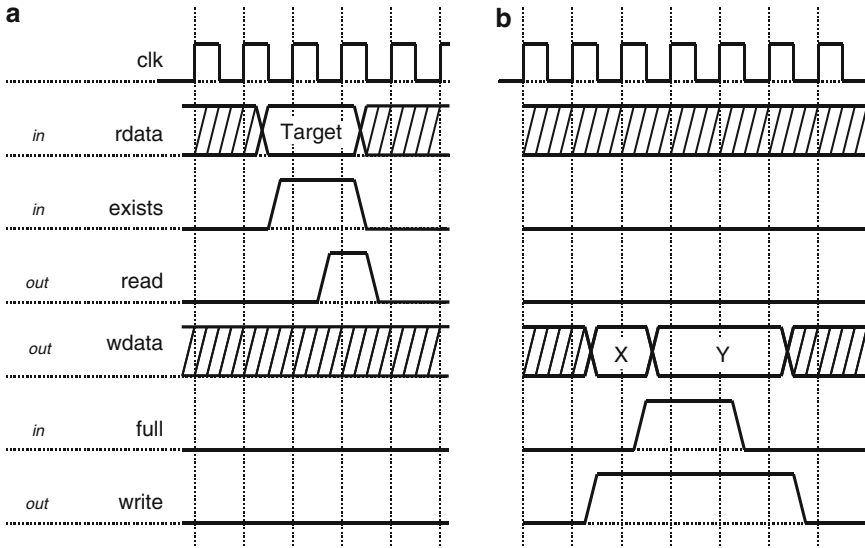
Fig. 12.2 (**a**) Input slave handshake (**b**) Output master handshake

The remaining datapath instructions in Listing 12.2 support the implementation of the input/output operations of the algorithm, and they are most easily understood by studying the controller description on Lines 43–61. The four states of the finite state machine correspond to the following activities:

- State s0: Reading the target angle.
- State c1: Perform the rotation.
- State s1: Produce output X.
- State s2: Produce output Y.

Figure 12.2 shows a sample input operation and a sample output operation. The cordic coprocessor goes through an infinite loop consisting of the operations: read target, calculate, write X, and write Y. Each time it needs a new target angle, the coprocessor will wait for exists to be raised. The coprocessor will acknowledge the request through read and grab a target angle. Next, the coprocessor proceeds to evaluate the output coordinates X and Y. When they are available, the write output is raised and, as long as the full input remains low, X and Y are passed to the output in a single clock cycle. In Fig. 12.2, four clock cycles are required for the complete output operation because the full input is raised for two clock cycles.

The hardware testbench for this design is left as an exercise for the reader (See Problem 12.1).

12.2.2 A Control Shell for Fast-Simplex-Link Coprocessors

We will now integrate the hardware kernel into a control shell with FSL interfaces. The FSL interface is natively supported only on the MicroBlaze processor (which is currently not included in the GEZEL simulation kernel). Therefore, GEZEL emulates FSL interfaces through memory-mapped operations on the ARM, and an `ipblock` with the outline of a real Fast Simplex Link interface. Figure 12.3 demonstrates this approach. The cordic kernel is encapsulated in a module, `fslcordic`, which defines the proper FSL interface. The FSL interface is driven from an ARM simulator, which drives the value of the interface signals through memory-mapped read and write operations.

Listing 12.3 shows the hardware platform of the complete design. The FSL interface is on lines 6–24. The pinout of this interface follows the specifications of the Xilinx FSL interface, although this GEZEL model does not use all features of the interface and will not use all signals on the interface pinout. In particular, the Xilinx FSL supports asynchronous operation and control information in conjunction with data, while GEZEL sticks to synchronous operation and data-only FSL transfers. The operation of the FSL interface is emulated with read and write operations on memory addresses in the ARM simulation model (lines 19–23). The control shell module, `fslcordic`, is very simple because the outline of the cordic kernel is already conform with the FSL interface. In fact, the cordic kernel can be directly instantiated (line 40) and wired to the FSL ports (lines 43–53). Finally, the top-level module interconnects the system simulation.
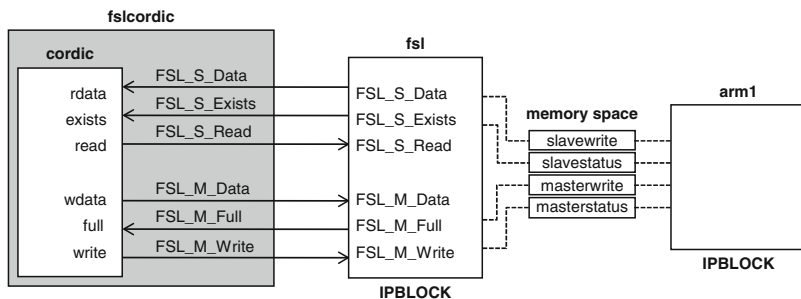
Fig. 12.3 Hierarchy of the GEZEL model in Listing 12.3

Listing 12.3 The CORDIC coprocessor attached toa fast simplex link

```
1  ipblock arm1 {
2    iptype "armsystem";
3    ipparm "exec = cordic_fixp";
4  }
5
6  ipblock fsl(in  FSL_S_Clk   : ns(1);   // notused
7             in  FSL_S_Read  : ns(1);   // hshk for slave side
8             out FSL_S_Data  : ns(32);  // data for slave side
```

```
 9               out FSL_S_Control : ns(1);   // control for slave side
10               out FSL_S_Exists  : ns(1);   // hshk for slave side
11               in  FSL_M_Clk     : ns(1);   // notused
12               in  FSL_M_Write   : ns(1);   // hshk for master side
13               in  FSL_M_Data    : ns(32);  // data for master side
14               in  FSL_M_Control : ns(1);   // control for master side
15               out FSL_M_Full    : ns(1)) { // hshk for master side
16     iptype "xilinx_fsl";
17     ipparm "core=arm1";                    // strongarm core
18
19     ipparm "slavewrite  = 0x80000000"; // write slave data
20     ipparm "slavestatus = 0x80000004"; // poll slave status
21
22     ipparm "masterread  = 0x80000008"; // read master data
23     ipparm "masterstatus= 0x8000000C"; // poll master status
24   }
25
26   dp fslcordic( out FSL_S_Clk     : ns(1);   // notused
27                 out FSL_S_Read    : ns(1);   // hshk for slave side
28                 in  FSL_S_Data    : ns(32);  // data for slave side
29                 in  FSL_S_Control : ns(1);   // control for slave // side
30                 in  FSL_S_Exists  : ns(1);   // hshk for slave side
31                 out FSL_M_Clk     : ns(1);   // notused
32                 out FSL_M_Write   : ns(1);   // hshk for master side
33                 out FSL_M_Data    : ns(32);  // data for master side
34                 out FSL_M_Control : ns(1);   // control for master side
35                 in  FSL_M_Full    : ns(1)) { // hshk for master side
36     sig rdata, wdata      : ns(32);
37     sig write, read       : ns(1);
38     sig exists, full      : ns(1);
39
40     use cordic_fsmd   (rdata, exists, read,
41                        wdata, full, write);
42
43     always {
44       rdata         = FSL_S_Data;
45       exists        = FSL_S_Exists;
46       FSL_S_Read    = read;
47
48       FSL_M_Data    = wdata;
49       FSL_M_Control = 0;
50       FSL_M_Write   = write;
51       full          = FSL_M_Full;
52
53       FSL_S_Clk     = 0;
54       FSL_M_Clk     = 0;
55     }
56   }
57
58   dp top {
59     sig FSL_Clk, FSL_Rst, FSL_S_Clk, FSL_M_Clk : ns( 1);
60     sig FSL_S_Read, FSL_S_Control, FSL_S_Exists : ns( 1);
61     sig FSL_M_Write, FSL_M_Control, FSL_M_Full : ns( 1);
62     sig FSL_S_Data, FSL_M_Data : ns(32);
```

```
63
64    use arm1;
65
66    use fslcordic (FSL_S_Clk, FSL_S_Read, FSL_S_Data, FSL_S_Control,
67                   FSL_S_Exists, FSL_M_Clk, FSL_M_Write, FSL_M_Data,
68                   FSL_M_Control, FSL_M_Full);
69
70    use fsl (FSL_S_Clk, FSL_S_Read, FSL_S_Data, FSL_S_Control,
71             FSL_S_Exists, FSL_M_Clk, FSL_M_Write, FSL_M_Data,
72             FSL_M_Control, FSL_M_Full);
73  }
74
75  system S {
76    top;
77  }
```

In order to verify the design, we also need a software driver. Listing 12.4 shows an example driver to compare the reference software implementation (Listing 12.1) with the FSL coprocessor. The driver evaluates 4096 rotations from 0 to $\frac{\pi}{2}$ and accumulates the coordinates. As the CORDIC design is in fixed point, the results of the hardware coprocessor must be exactly the same as the results from the software reference implementation. Of particular interest in the software driver is the emulation of the FSL interface signals through memory-mapped operations. The driver first transfers a token to the coprocessor (lines 8–9), and then reads two coordinates from the coprocessor (lines 11–15).

Listing 12.4 Driver for the CORDIC coprocessor on the emulated FSL interface
```
1   void cordic_driver(int target, int *rX, int *rY) {
2     volatile unsigned int *wchannel_data   = (int *) 0x80000000;
3     volatile unsigned int *wchannel_status = (int *) 0x80000004;
4     volatile unsigned int *rchannel_data   = (int *) 0x80000008;
5     volatile unsigned int *rchannel_status = (int *) 0x8000000C;
6     int i;
7
8     while (*wchannel_status == 1) ;
9     *wchannel_data = target;
10
11    while (*rchannel_status != 1) ;
12    *rX = *rchannel_data;
13
14    while (*rchannel_status != 1) ;
15    *rY = *rchannel_data;
16  }
17
18  // Reference implementation
19  extern void cordic(int target, int *rX, int *rY);
20
21  int main(void) {
22    fixed X, Y, target;
```

```
23    fixed accsw, accfsl;
24
25    accsw = 0;
26    for (target = 0; target < PI/2; target += (1 << (UNIT - 12))) {
27      cordic(target, &X, &Y);
28      accsw += (X + Y);
29    }
30
31    accfsl = 0;
32    for (target = 0; target < PI/2; target += (1 << (UNIT - 12))) {
33      cordic_driver(target, &X, &Y);
34      accfsl += (X + Y);
35    }
36
37    printf("Checksum SW %x FSL %x\n", accsw, accfsl);
38    return(0);
39  }
```

The cosimulation of this model, consisting of the hardware design in Listing 12.3 and the software design in Listing 12.4, confirms that the reference implementation and the hardware coprocessor behave identically. This functional verification is the most important feature of this model. The cosimulation model is less relevant for performance evaluation. As the behavior of the FSL interface is emulated, the performance of the implementation may still change once we move to a processor with a native FSL interface. In the next section, we will port this coprocessor to the FPGA for a detailed performance analysis.

```
> /usr/local/arm/bin/arm-linux-gcc -static -O3 cordic_fixp.c
  -o cordic_fixp
> gplatform fsl.fdl
core arm1
armsystem: loading executable [cordic_fixp]
Checksum SW 4ae1ee FSL 4ae1ee
Total Cycles: 3467162
```

12.3   An FPGA Prototype of the CORDIC Coprocessor

Our next step is to map the complete system – processor and coprocessor – to an FPGA prototype. The prototyping environment we are using contains the following components.

- Spartan-3E Starter Kit, including a Spartan 3ES500 Xilinx FPGA and various peripherals. We will be making use of one peripheral besides the FPGA: a 64 MByte DDR SDRAM Module.
- FPGA Design Software, consisting of Xilinx Platform Studio (EDK 9.2) and associated hardware synthesis tools (ISE 9.2.04).
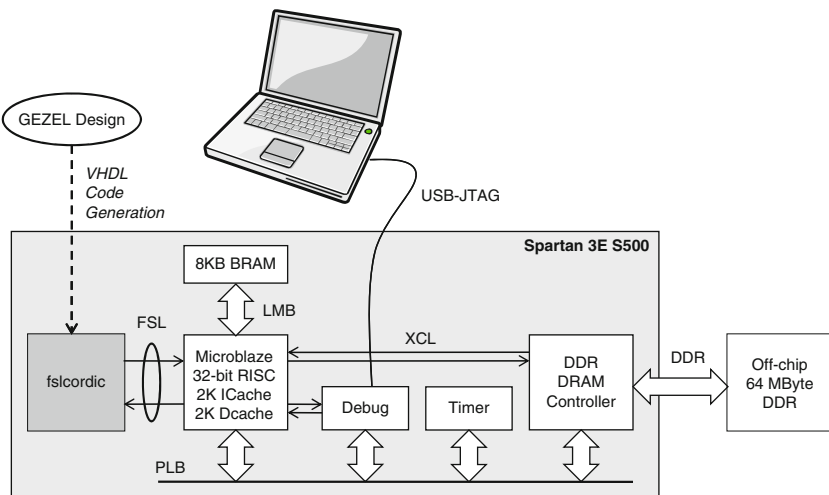
Fig. 12.4 FPGA prototype

Figure 12.4 shows the system architecture of the CORDIC design. The coprocessor connects through a Fast Simplex Link (FSL) to a Microblaze processor. The processor includes an instruction-cache and a data-cache of two 2 KByte. Besides the processor, several other components are included on the platform: an 8 KByte local memory, a debug unit, a timer, and a Dual Data Rate (DDR) DRAM Controller. The interconnect architecture of the system is quite sophisticated, if we consider how these components interact.

- The 8 KByte local memory is intended as a local store for the Microblaze, for example to hold the stack or the heap segment of the embedded software. The local memory uses a dedicated Local Memory Bus (LMB) so that local memory can be accessed with a fixed latency (2 cycles).
- The DDR Controller provides access to a large off-chip 64 MByte DRAM. The DDR Controller has two connections to the Microblaze processor. The first uses the common Processor Local Bus (PLB), and is used for control operations on the DDR Controller. The second uses the Xilinx Cache Link (XCL), a fast point-to-point bus similar to FSL, and is used for data transfer from the DDR Controller to the cache memories.
- The debug unit allows the system to be controlled using debug software on a laptop. The debug unit has two connections to the Microblaze processor: a PLB connection for control operations on the debug unit, and a dedicated connection to the debug port on the Microblaze processor. The debug port provides access to all internal processor registers, and it supports low-level control operations such as single-stepping the processor.

- The timer unit is used to obtain accurate performance estimations, by counting the number of elapsed cycles between two positions in the program. The timer unit is controlled through the PLB.

The platform design and implementation flow relies on Xilinx Platform Studio (XPS) and will not be discussed in detail here. We will clarify the implementation path from GEZEL to FPGA using XPS. Once we have a working GEZEL system simulation, we can convert GEZEL code into synthesizable VHDL. The code generator is called `fdlvhd`, and a sample run of the tool on Listing 12.3 is as follows.

```
> fdlvhd -c FSL_Clk FSL_Rst fsl.fdl
Pre-processing System ...
Output VHDL source ...
---------------------------
Generate file: arm1.vhd
Generate file: fsl.vhd
Generate file: cordic_fsmd.vhd
Generate file: fslcordic.vhd
Generate file: top.vhd
Generate file: system.vhd
Generate file: std_logic_arithext.vhd
```

The code generator creates a separate file for each module in the system. In addition, one extra library file is generated (`std_logic_arithext.vhd`), which is needed for synthesis of GEZEL-generated VHDL code. The code generator also uses a command line parameter, `-c FSL_Clk FSL_Rst`, which enables a user to choose the name of the clock and reset signal on the top-level module. This makes the resulting code pin-compatible with VHDL modules expected by XPS. If we consider the system hierarchy of Fig. 12.3 once more, we conclude that not all of the generated code is of interest. Only `cordic` and `fslcordic` constitute the actual coprocessor. The other modules are "simulation stubs." Note that GEZEL does not generate VHDL code for an ARM or and `fsl` interface; the `ipblock` modules in Listing 12.3 translate to black-box views. The transfer of the code to XPS relies on the standard design flow to create new peripherals. This makes it possible to port GEZEL to XPS without writing a single line of VHDL.

A detailed discussion of the synthesis results for the platform is beyond the scope of this example. However, it is useful to make a brief summary of the results, in particular because it illustrates the relative hardware cost of a coprocessor in a system platform such as in Fig. 12.4. All components of the platform run at 50 MHz. We partition the implementation cost of the system into logic cells (lookup-tables for FPGA), flip-flops, and BlockRAM cells. The entire system requires 4842 logic cells, 3411 flip-flops and 14 BlockRAM cells. Figure 12.5 shows the relative resource cost for each of the major components in the platform. There are several important conclusions to make from this figure. First, the processor occupies around one quarter of the resources on the platform. The most expensive component in terms of resources is the DDR controller. The coprocessor cost is relatively minor, although still half of the Microblaze in logic area.

Logic Cells (LUT)   Flip-Flop   Block RAM

M = Microblaze, **D** = DDR Controller, **C** = CORDIC, **T** = Timer, **R** = Local RAM
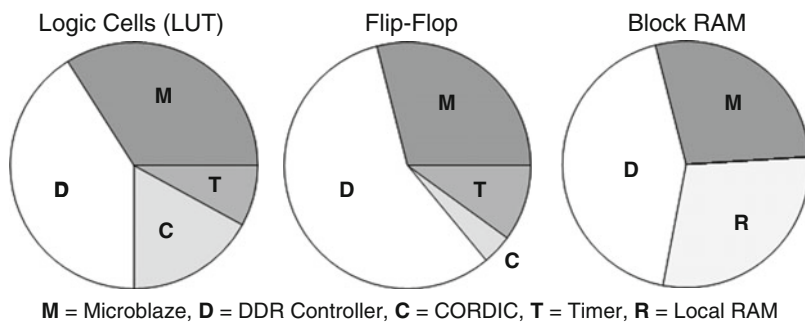
Fig. 12.5 Relative resource cost of platform components for Fig. 12.4

12.4 Handling Large Amounts of Rotations

In this section, we investgate the performance of the FPGA prototype of the
CORDIC coprocessor, and we consider performance optimizations on the overall
system throughput.

A coprocessor is not very useful if you use it only once. Therefore, we will con-
sider a scenario that includes a large number of rotations. We use a large table of
target angles, stored in off-chip memory. The objective is to convert this table into an
equivalent table of (X, Y) coordinates, also stored in off-chip memory. Refering to
Fig. 12.4, the table will be stored in the 64 MByte DDR memory, and the elements
of that table need to be fetched by the Microblaze and processed in the attached
CORDIC coprocessor. The outputs of the CORDIC need to be written back to the
64 MByte DDR memory.

Let's start with a simple software driver for this coprocessor. Listing 12.5 il-
lustrates a Microblaze program that performs CORDIC transformations on an
array of 8192 elements. The coprocessor driver, `cordic_driver`, is implemented
on lines 9–16. The Microblaze processor uses dedicated instructions to write to,
and read from the FSL interface: (`putfsl`) and (`getfsl`). One function call
to `cordic_driver` will complete a single CORDIC rotation. While the copro-
cessor is active, the Microblaze processor will stall and wait for the result. As
discussed earlier, each rotation takes 20 clock cycles in the hardware implemen-
tation. The function `compare`, lines 17–45, compares the performance of 8192
cordic rotations in software vs. 8192 cordic rotations in hardware. Performance mea-
surements are obtained from a timer module: `XTmrCtr_Start`, `XTmrCtr_Stop`,
and `XTmrCtr_GetValue` will start, stop and query the timer module, respectively.
The resulting rotations are also accumulated (in `accsw` and `accfsl`) as a simple
checksum to verify that the software and the hardware obtain the same result.

We compile Listing 12.5 while allocating all sections in off-chip memory. The
compilation command line selects medium optimization (`-O2`), as well as several
options specific to the Microblaze processor hardware. This includes a hardware
integer multiplier (`-mno-xl-soft-mul`) and a hardware pattern comparator

Listing 12.5 Driver for the CORDIC coprocessor on Microblaze

```
1   #include "fsl.h"
2   #include "xparameters.h"
3   #include "xtmrctr.h"
4   #define N 8196
5   int arrayT[N];
6   int arrayX[N];
7   int arrayY[N];
8
9   void cordic_driver(int target, int *rX, int *rY) {
10    int r;
11    putfslx(target,0,FSL_ATOMIC);
12    getfslx(r,0,FSL_ATOMIC);
13    *rX = r;
14    getfslx(r,0,FSL_ATOMIC);
15    *rY = r;
16  }
17
18  int compare(void) {
19    unsigned i;
20    int accsw = 0, accfsl = 0;
21    int timesw, timefsl;
22    XTmrCtr T;
23
24    XTmrCtr_Start(&T, 0);
25    for (i=0; i<N; i++) {
26      cordic(arrayT[i], &arrayX[i], &arrayY[i]);
27      accsw += (arrayX[i] + arrayY[i]);
28    }
29    XTmrCtr_Stop(&T, 0);
30    timesw = XTmrCtr_GetValue(&T, 0);
31
32    XTmrCtr_Start(&T, 0);
33    for (i=0; i<N; i++) {
34      cordic_driver(arrayT[i], &arrayX[i], &arrayY[i]);
35      accfsl += (arrayX[i] + arrayY[i]);
36    }
37    XTmrCtr_Stop(&T, 0);
38    timefsl = XTmrCtr_GetValue(&T, 0);
39
40    xil_printf("Checksum SW %x FSL %x\n", accsw, accfsl);
41    xil_printf("Cycles   SW %d FSL %d\n", timesw, timefsl);
42
43    return(0);
44  }
```

(-mxl-pattern-compare). The compiler command line also shows the use of
a *linker script*, which allows the allocation of sections to specific regions of memory
(See Sect. 6.3).

```
> mb-gcc -O2 \
      cordiclarge.c     \
```

```
        -o executable.elf \
        -mno-xl-soft-mul  \
        -mxl-pattern-compare \
        -mcpu=v7.00.b          \
        -Wl,-T -Wl,cordiclarge_linker_script.ld \
        -g \
        -I./microblaze_0/include/ \
        -L./microblaze_0/lib/
> mb-size executable.elf
   text    data     bss     dec     hex filename
   7032     416  100440  107888   1a570 executable.elf
```

The actual sizes of the program sections are shown using the `mb-size` command. Recall that `text` contains instructions, `data` contains initialized data, and `bss` contains uninitialized data. The large `bss` section is occupied by 3 arrays of 8192 elements each, which require 98304 bytes. The remainder of that section is required for other global data, such as global variables in the C library.

The resulting performance of the program is shown in scenario 1 and 2 of Table 12.1. The software CORDIC requires 358 million cycles, while the hardware-accelerated cordic requires 4.8 million cycles, giving a speedup of 74.5 times. While this is an excellent improvement, it also involves significant overhead. Indeed, 8192 CORDIC rotations take 1,63,840 cycles in the FSL coprocessor. Over the total runtime of 4.8 million cycles, the coprocessor thus has only 3% utilization! Considering the program in Listing 12.5, this is also a peak utilization, because the coprocessor is called in a tight loop with virtually no other software activities. Clearly, there is still another bottleneck in the system.

That bottleneck is the off-chip memory, in combination with the PLB memory bus leading to the microprocessor. As all program segments are stored in off-chip memory, the microblaze will fetch not only all CORDIC data elements, but also all instructions from the off-chip memory. Worse, the cache memory on a microblaze is not enabled by default until instructed so by software, so that the program does not benefit from on-chip memory at all.

There are two possible solutions: local on-chip memories and cache memory. We will show that the effect of both of them is similar.

Table 12.1 Performance evaluation over 8192 CORDIC rotations

Scenario	CORDIC	Cache	text segment	data segment	bss segment	Performance (cycles)	Speedup
1	SW	no	DDR	DDR	DDR	358024365	1
2	FSL	no	DDR	DDR	DDR	4801716	74.5
3	SW	no	On-chip	On-chip	DDR	16409224	21.8
4	FSL	no	On-chip	On-chip	DDR	1173651	305
5	SW	yes	DDR	DDR	DDR	16057950	22.3
6	FSL	yes	DDR	DDR	DDR	594655	602
7	FSL (prefetch)	yes	DDR	DDR	DDR	405840	882
8	FSL (prefetch)	yes/8	On-chip	On-chip	DDR	387744	923

- To enable the use of the on-chip memory (Fig. 12.4), we need to modify the linker script and re-allocate sections to on-chip memory. In this case, we need to move the `text` segments as well as the constant `data` segment to on-chip memory. In addition, we can also allocate the stack and heap to on-chip memory, which will ensure that local variables and dynamic variables will remain on-chip.
- To enable the use of cache memory, we need to include

```
microblaze_enable_icache();
microblaze_enable_dcache();
```

at the start of the program. The data and instruction cache of a microblaze is a direct-mapped, 4-word-per-line cache architecture.

The result of each of these two optimizations is shown in scenario 3, 4, 5, and 6 in Table 12.1. For the software implementations, the use of on-chip local memory, and the use of a cache each provide a speedup of approximately 22 times. For the hardware-accelerated implementations, the use of on-chip local memory provides a speedup of 305 times, while the use of a cache provides a speedup of 602 times. These results confirm that off-chip memory clearly was a major bottleneck in system performance. In general the effect of adding a cache is larger than the effect of moving the text segment/local data into on-chip memory. This is because of two reasons: (a) the cache improves memory-access time, and (b) the cache improves the *off-chip* memory-access time. Indeed, the "XCL" connections, shown in Fig. 12.4, enable burst-access to the off-chip memory, while the same burst-access effect cannot be achieved through the "PLB" connection.

We note also that the impact of cache on the hardware coprocessor is much more dramatic (600 times speedup instead of 300 times speedup) than its impact on the software CORDIC (22.3 speedup instead of 21.8 speedup). This can be understood by looking at the absolute performance numbers. For the case of software, the cache provides an advantage of 3.5 million cycles over local-memory (scenario 3 vs. scenario 5). For the case of hardware, the cache provides an advantage of only 500,000 cycles over local memory (scenario 4 vs. scenario 6). However, the hardware-accelerated system is already heavily optimized, and hence very sensitive to inefficiencies.

How can we improve this design even further? By close inspection of the loop that drives the FSL coprocessor, we find that the memory accesses and the coprocessor execution are strictly sequential. This is in particular the case for memory-writes, since the write-through cache of the Microblaze forces all of them to be an off-chip access. Indeed, the code first accesses `arrayT`, then runs the coprocessor through `putfsl` and `getfsl`, and finally writes back the results into `arrayX` and `arrayY`. This is illustrated in Fig. 12.6a.

```
for (i=0; i<N; i++) {
  cordic_driver(arrayT[i], &arrayX[i], &arrayY[i]);
  accfsl += (arrayX[i] + arrayY[i]);
}
```

The key optimization is to exploit parallelism between the memory accesses and the coprocessor execution. Specifically, instead of waiting for the result of the
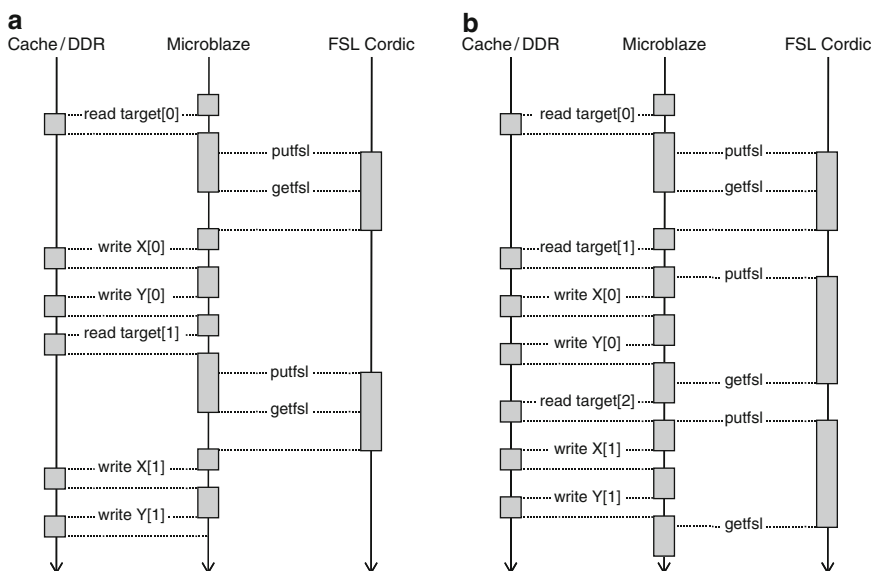
Fig. 12.6 (**a**) Sequential memory access and coprocessor execution (**b**) Folded memory access and coprocessor execution

coprocessor (using `getfsl`), the Microblaze processor may use that time to read from/ write to memory. A solution that uses this overlap for memory writes is shown in Fig. 12.6b. An equivalent C code fragment that achieves this behavior looks as follows.

```
cordic_put(arrayT[0]);
for (i=1; i<N; i++) {
  cordic_get(&tmpX, &tmpY);
  cordic_put(arrayT[i]);
  arrayX[i-1]  = tmpX;
  arrayY[i-1]  = tmpY;
  accfsl       += (tmpX + tmpY);
}
cordic_get(&arrayX[N-1], &arrayY[N-1]);
accfsl += ( arrayX[N-1] + arrayY[N-1]);
```

The effect of this optimization is illustrated in scenario 7 of Table 12.1. An additional 2,00,000 clock cycles are gained, and the overall execution time is around 4,00,000 clock cycles, or a speedup of 882. At this point, the coprocessor utilization has increased to 41%, which is an improvement of more than 10 times over the original case.

Further improvements are still possible. For example, we know that the accesses to `arrayT` are strictly sequential. Hence, it makes sense to increase the line size of the cache as much as possible (the line size is the number of consecutive elements that are read after a cache miss). In addition, we can use an on-chip memory for

the program, but reserve all the cache memory for data accesses. The result of these optimizations is an additional 18,000 cycles, as shown in Table 12.1, scenario 8. The overall speedup is now 923 times, and the coprocessor utilization is 42%. As long as the utilization is not 100%, the system-level bottleneck is *not* in hardware but rather between the microblaze and the off-chip memory. The next step in the optimization is to investigate the assembly code of the loop, and to profile the behavior of the loop in detail.

In conclusion, this section has demonstrated that the design of a hardware module is only the first step in an efficient hardware/software codesign. System integration is the next, and often more complicated, step.

12.5 Summary

In this chapter we discussed the implementation of the Coordinate Rotation Digital Computer (CORDIC) algorithm as a hardware coprocessor. The CORDIC algorithm rotates, iteratively, a vector (X, Y) over a given angle α. The algorithm uses only integer operations, which makes it very well suited for embedded system implementation. We discussed a coprocessor design based on the Fast Simple Link coprocessor interface, and we created a simulation model of the CORDIC in GEZEL, first as a standalone module, and next as a coprocessor module. After functional verification of the coprocessor at high abstraction level, we ported the design to an FPGA prototype using a Spartan 3E chip. The resulting embedded system architecture consists roughly of one-third microprocessor, one-third memory-controller, and one-third peripherals (with the coprocessor being included in the peripherals). Early implementation results showed that the coprocessor provided a speedup of 74 over the CORDIC software implementation. Through careful tuning, in particular by optimizing off-chip accesses, that speedup can be further improved to 923 times.

12.6 Further Reading

The CORDIC algorithm is 50 years old, and was developed for "real-time airborne computation," in other words, for a military application. The original CORDIC proposal, by Volder, is a good example of a paper that truly stands the test of time Volder (1959). More recently, Maharatna has provided a comprehensive overview Maharatna et al. (2009). Valls discusses CORDIC applications in digital radio receivers Valls et al. (2006).

Table 12.2 Test cases
for problem 17.1

Angle	cos(angle)	sin(angle)
0	1	0
$\pi/6$	$\sqrt{3}/2$	$1/2$
$\pi/4$	$1/\sqrt{2}$	$1/\sqrt{2}$
$\pi/3$	$1/2$	$\sqrt{3}/2$
$\pi/2$	0	1

12.7 Problems

12.1. Design a GEZEL testbench for the standalone CORDIC design shown in
Listing 12.2. Verify the sine and cosine values shown in Table 12.2.

12.2. The CORDIC algorithm in this chapter is working in the so-called rotation
mode. In rotation mode, the CORDIC iterations aim to drive the value of the angle
accumulator to zero (refer to Listing 12.1). CORDIC can also be used in vector
mode. In this mode, the CORDIC rotations aim to drive the value of the Y coordinate
to zero. In this case, the input of the algorithm consists of the vector (x_0, y_0), and
the angle accumulator is initialized to zero.

(a) Show that, in the vector mode, the final values of (x_T, y_T) are given by:

$$\begin{bmatrix} x_T \\ y_T \end{bmatrix} = \begin{bmatrix} K.\sqrt{(x_0{}^2 + y_0{}^2)} \\ 0 \end{bmatrix} \qquad (12.11)$$

with K a similar magnitude constant as used in the rotation mode.

(b) Show that, in the vector mode, the final value of the angle accumulator is
given by:

$$\alpha = \arctan\left(\frac{y_0}{x_0}\right) \qquad (12.12)$$

(c) Adjust the hardware design in Listing 12.2 so that it implements CORDIC in
vector mode. Verify your design with some of the tuples shown in Table 12.2.

12.3. Develop a CORDIC coprocessor in rotation mode using the custom-
instruction interface discussed in Sect. 11.3.3. The recommended approach is to
build a coprocessor that does a single CORDIC iteration per custom-instruction
call. Hence, you will need an OP2X2 instruction for each iteration. You will also
need a mechanism to program the rotation angle. For example, the software driver
could look like:

```
int target, X, Y;
unsigned i;

// argument 1: target angle
// argument 2: 10 iterations
OP2x2_1(target, 10, 0, 0);

for (i=0; i<10; i++)
  OP2x2_1(X, Y, X, Y);
```

References

Aeroflex G (2009) Leon-3/grlib intellectual property cores. Tech. rep., http://www.gaisler.com

Appel AW (1997) Modern Compiler Implementation in C: Basic Techniques. Cambridge University Press

Atmel (2008) At91sam7l128 preliminary. Tech. rep., http://www.atmel.com/dyn/products/product_card.asp?part_id=4293

Berry G (2000) The foundations of esterel. In: Proof, Language, and Interaction, pp 425–454

Bogdanov A, Knudsen L, Leander G, Paar C, Poschmann A, Robshaw M, Seurin Y, Vikkelsoe C (2007) Present: An ultra-lightweight block cipher. In: Proc. Cryptographic Hardware and Embedded Systems 2007, pp 450–466

Buck JT (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. Dissertation, UCB/ERL 93/63, UC Berkeley, CA

Claasen T (1999) High speed: not the only way to exploit the intrinsic computational power of silicon. In: Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International, pp 22–25

Claasen T (2006) An industry perspective on current and future state of the art in system-on-chip (soc) technology. Proceedings of the IEEE 94(6):1121–1137

Committee T (1995) Tool interface standard executable and linkable format (elf) specification, version 1.2. Tech. rep., http://refspecs.freestandards.org/elf/elf.pdf

Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK (1991) Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst 13(4):451–490

Davio M, Deschamps JP, Thayse A (1983) Digital Systems with Algorithm Implementation. John Wiley & Sons, Inc., New York, NY, USA

De Canniere C, Preneel B (2005) Trivium specifications. Tech. rep., ESAT/SCD-COSIC, K.U.Leuven, http://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf

Dennis J (2007) A dataflow retrospective - how it all began. http://csg.csail.mit.edu/Dataflow/talks/DennisTalk.pdf

D'Errico J, Qin W (2006) Constructing portable compiled instruction-set simulators: an adl-driven approach. In: DATE '06: Proceedings of the conference on Design, automation and test in Europe, pp 112–117

Dijkstra EW (2009) The E.W. Dijkstra Archive. Tech. rep., http://www.cs.utexas.edu/users/EWD/

ECRYPT (2008) The estream project. Tech. rep., http://www.ecrypt.eu.org/stream/technical.html

Edwards SA (2006) The challenges of synthesizing hardware from c-like languages. IEEE Design & Test of Computers 23(5):375–386

Eker J, Janneck J, Lee E, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y (2003) Taming heterogeneity - the ptolemy approach. Proceedings of the IEEE 91(1):127–144

Gajski DD, Abdi S, Gerstlauere A, Schirner G (2009) Embedded System Design: Modeling, Synthesis, Verification. Springer

Ganesan P, Venugopalan R, Peddabachagari P, Dean A, Mueller F, Sichitiu M (2003) Analyzing and modeling encryption overhead for sensor network nodes. In: WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, ACM, New York, NY, USA, pp 151–159, DOI http://doi.acm.org/10.1145/941350.941372

Good T, Benaissa M (2007) Hardware results for selected stream cipher candidates. Tech. rep., eSTREAM project, http://www.ecrypt.eu.org/stream/hw.html

Gupta S, Gupta R, Dutt N, Nicolau A (2004) SPARK:A Parallelizing Approach to the High-Level Synthesis of Digital Circuits. Springer

Harel D (1987) Statecharts: A visual formulation for complex systems. Sci Comput Program 8(3):231–274

Hennessy JL, Patterson DA (2006) Computer Architecture: A Quantitative Approach, 4th Edition. Morgan Kaufmann

Hillis WD, Steele GL Jr (1986) Data parallel algorithms. Commun ACM 29(12):1170–1183

Hodjat A, Verbauwhede I (2004) High-throughput programmable cryptocoprocessor. IEEE Micro 24(3):34–45

Hoe JC (2000) Operation-centric hardware description and synthesis. PhD thesis, MIT

IBM (2009) Coreconnect bus architecture. Tech. rep., https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture

Inc X (2009) Xilinx embedded development toolkit. Tech. rep., http://www.xilinx.com/support/documentation/dt_edk.htm

Ivanov A, De Micheli G (2005) Guest editors' introduction: The network-on-chip paradigm in practice and research. Design & Test of Computers, IEEE 22(5):399–403

Kaps JP (2008) Chai-tea, cryptographic hardware implementations of xtea. In: INDOCRYPT, pp 363–375

Karlof C, Sastry N, Wagner D (2004) Tinysec: a link layer security architecture for wireless sensor networks. In: SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems, ACM, New York, NY, USA, pp 162–175, DOI http://doi.acm.org/10.1145/1031495.1031515

Kastner R, Kaplan A, Sarrafzadeh M (2003) Synthesis Techniques and Optimizations for Reconfigurable Systems. Kluwer Academic Publishers

Keutzer K, Newton A, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 19(12):1523–1543

Kogge PM (1981) The Architecture of Pipelined Computers. McGraw-Kill

Leander G, Paar C, Poschmann A, Schramm K (2007) New lightweight des variants. In: Fast Software Encryption, Lecture Notes on Computer Science, vol 4593, pp 196–200

Lee EA, Messerschmitt DG (1987) Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans Computers 36(1):24–35

Leupers R, Ienne P (2006) Customizable Embedded Processors: Design Technologies and Applications. Morgan Kaufmann Publishers Inc.

Lynch M (1993) Micro-programmed State Machine Design, CRC Press, 1993

Ltd A (2009a) The amba system architecture. Tech. rep., http://www.arm.com/products/solutions/AMBAHomePage.html

Ltd A (2009b) Arm infocenter. Tech. rep., http://infocenter.arm.com/help/index.jsp

Madsen J, Steensgaard-Madsen J, Christensen L (2002) A sophomore course in codesign. Computer 35(11):108–110, DOI http://dx.doi.org/10.1109/MC.2002.1046983

Maharatna K, Valls J, Juang TB, Sridharan K, Meher P (2009) 50 years of cordic: Algorithms, architectures, and applications. Circuits and Systems I: Regular Papers, IEEE Transactions on 56(9):1893–1907

McKee S (2004) Reflections on the memory wall. In: Conf. Computing Frontiers, pp 162–168

Meiser G, Eisenbarth T, Lemke-Rust K, Paar C (2007) Software implementation of estream profile i ciphers on embedded 8-bit avr microcontrollers. Tech. rep., eSTREAM project, http://www.ecrypt.eu.org/stream/sw.html

Menezes A, van Oorschot P, Vanstone S (2001) Handbook of Applied Cryptography. CRC Press

Micheli GD, Benini L (2006) Networks on Chips: Technology and Tools (Systems on Silicon). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Micheli GD, Wolf W, Ernst R (2001) Readings in Hardware/Software Co-Design. Morgan Kaufmann Publishers Inc.

Moderchai BA (2006) Principles of Concurrent and Distributed Programming, 2nd Edition. Addison Wesley

Muchnick SS (1997) Advanced Compiler Design and Implementation. Morgan Kaufmann

NIST (2001) Federal information processing standards publication 197: Announcing the advanced encryption standard (aes). Tech. rep., http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

Parhi KK, Messerschmitt DG (1989) Fully-static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. In: ICPP (1), pp 209–216

Pasricha S, Dutt N (2008) On-Chip Communication Architectures: System on Chip Interconnect. Morgan Kaufmann

Potop-Butucaru D, Edwards SA, Berry G (2007) Compiling Esterel. Springer

Qin W (2004) Modeling and description of embedded processors for the development of software tools. PhD thesis, Princeton University

Qin W, Malik S (2003) Flexible and formal modeling of microprocessors with application to retargetable simulation. In: DATE '03: Proceedings of the conference on Design, Automation and Test in Europe, p 10556

Rabaey JM (2009) Low Power Design Essentials. Springer

Rowen C (2004) Engineering the Complex SOC:Fast, Flexible Design with Configurable Processors. Prentice Hall

Saleh R, Wilton S, Mirabbasi S, Hu A, Greenstreet M, Lemieux G, Pande P, Grecu C, Ivanov A (2006) System-on-chip: Reuse and integration. Proceedings of the IEEE 94(6):1050–1069

Satoh A, Morioka S (2003) Hardware-focused performance comparison for the standard block ciphers aes, camellia, and triple-des. In: ISC, no. 2851 in Lecture Notes on Computer Science, pp 252–266

Schaumont P, Shukla SK, Verbauwhede I (2006) Design with race-free hardware semantics. In: DATE, pp 571–576

Smotherman M (2009) A brief history of microprogramming. Tech. rep., Clemson University, http://www.cs.clemson.edu/mark/uprog.html

Stanford Graphics Lab (2003) Brook language. Http://graphics.stanford.edu/projects/brookgpu/lang.html

Talla D, Hung CY, Talluri R, Brill F, Smith D, Brier D, Xiong B, Huynh D (2004) Anatomy of a portable digital mediaprocessor. Micro, IEEE 24(2):32–39

Taubenfeld G (2006) Synchronization Algorithms and Concurrent Programming. Pearson/Prentice Hall

Thies W (2008) Language and compiler support for stream programs. PhD thesis, MIT, http://groups.csail.mit.edu/cag/streamit/shtml/documentation.shtml

Vahid F (2003) The softening of hardware. Computer 36(4):27–34

Vahid F (2007a) Digital Design. John Wiler and Sons Publishers

Vahid F (2007b) It's time to stop calling circuits "hardware". Computer 40(9):106–108

Vahid F (2009) Dalton project. Tech. rep., http://www.cs.ucr.edu/dalton/

Valls J, Sansaloni T, Perez-Pascual A, Torres V, Almenar V (2006) The use of cordic in software defined radios: a tutorial. Communications Magazine, IEEE 44(9):46–50

Verbauwhede I, Scheers C, Rabaey J (1994) Memory estimation for high level synthesis. In: Proceedings of the Design Automation Conference, pp 143–148

Volder JE (1959) The cordic trigonometric computing technique. Electronic Computers, IEEE Transactions on EC-8(3):330–334

Wolf W (2003) A decade of hardware/software codesign. Computer 36(4):38–43

Wulf W, McKee S (1995) Hitting the memory wall: Implications of the obvious. In: ACM SIGARCH Computer Architecture News, 23, http://www.cs.virginia.edu/papers/Hitting_Memory_Wall-wulf94.pdf

Xilinx I (2009) Picoblaze for extended spartan-3a family, virtex-4, virtex-ii, and virtex-ii pro fpgas.
    Tech. rep., http://www.xilinx.com/products/ipcenter/picoblaze-S3-V2-Pro.htm
Yaghmour K, Masters J, Ben-Yossef G, Gerum P (2008) Building Embedded Linux Systems, 2nd
    Edition. O'Reilly Media, Inc.

Index

8051
    in GEZEL, 221

address decoding, 233
admissible schedule, 40
Advanced Encryption Standard, 319
Ahmdahl's law, 26
Application Binary Interface, 191
ASIC, 18
ASIP, 12, 18
assembler, 166

bandwidth
    off-chip, 211
    on-chip, 211
big-endian, 179
bit-parallel processor, 307
bit-serial processor, 307
block cipher, 337
blocking, 267
boolean data flow, 46
bus
    alignment, 240
    burst transfer, 242
    clock cycle, 235
    locking, 248
    multi-layer, 250
    naming convention, 235
    overlapping arbitration, 246
    pipelined transfer, 242
    split transfer, 242
    timeout, 238
    timing diagram, 235
    topology, 250
    transfer sizing, 239
    wait state, 238
bus arbiter, 233, 245

bus bridge, 206
bus interface, 231
bus master, 206, 233
bus slave, 206, 233

cache
    set-associative, 195
cast
    in GEZEL, 98
CFG, *see* control flow graph
ciphertext, 337
circular queue, 50
code inlining, 60
communication-constrained coprocessor, 306
compiler, 166
computation-constrained coprocessor, 306
computational efficiency
    actual, 210
    intrinsic, 209
concurrency, 25
Connection Machine, 26
continuous-time model, 23
control edge, 72
control edge implementation, 74
control flow graph, 73, 75
    construction from C, 76
    for control statements, 76
    of euclid's algorithm, 76
control flow modeling, 46
control hierarchy, 311
control path, 76
control processing, 22
control shell, 231, 303
    index-multiplexing, 309
    time-multiplexing, 309
control shell port, 305
control store, 136
    control store address register, 136