ACSP · Analog Circuits And Signal Processing

Khaled Salah Mohamed

IP Cores Design from Specifications to Production

Modeling, Verification, Optimization, and Protection

Springer

Khaled Salah Mohamed

IP Cores Design from Specifications to Production

Modeling, Verification, Optimization, and Protection

Springer

Khaled Salah Mohamed
Emulation
Mentor Graphics
Heliopolis, Egypt

ISSN 1872-082X ISSN 2197-1854 (electronic)
Analog Circuits and Signal Processing
ISBN 978-3-319-22034-5 ISBN 978-3-319-22035-2 (eBook)
DOI 10.1007/978-3-319-22035-2

Library of Congress Control Number: 2015947256

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Preface

This book discusses the life cycle process of IP cores from specification to production which includes four major steps: (1) IP modeling, (2) IP verification, (3) IP optimization, and (4) IP protection. Moreover, the book presents most of the famous memory cores and controller IPs and analyzes the trade-off between them. In this book, we give an in-depth introduction to SoC buses and peripheral IPs. We explain their features and architectures in detail. Moreover, we provide a deep introduction to Verilog from both implementation and verification points of view. The book presents a simple methodology in building a reusable RTL verification environment using UVM. UVM is a culmination of well-known ideas and best practices. Moreover, it presents simple steps to verify an IP and build an efficient and smart verification environment. A SoC case study is presented to compare traditional verification with a UVM-based verification. Bug localization is a process of identifying specific locations or regions of source code that is buggy and needs to be modified to repair the defect. Bug localization can significantly reduce human effort and design cost. In this book, a novel automated coverage-based functional bug localization method for complex HDL designs is proposed, which significantly reduces debugging time. The proposed bug localization methodology takes information from regression suite as an input and produces a ranked list of suspicious parts of the code. We present an online RTL-level scan-chain methodology to reduce debugging time and effort for emulation. Run-time modifications of the values of any of the internal signals of the DUT during execution can be easily performed through the proposed online scan-chain methodology. A utility tool has been developed to help ease this process.

Heliopolis, Egypt Khaled Salah Mohamed

Contents

Chapter 1
Introduction

Technological progress enables more and more functionality to be integrated on a single chip. Figure 1.1 shows the most important milestones in Very-large-scale integration (VLSI), it is all about integration. In 1937, Shannon introduces the world to binary digital electronics. The first bipolar transistor was fabricated at Bell Labs in 1947 [1]. In 1960, the first MOSFET which contains one transistor was fabricated followed by the first integrated circuit (IC) which contains two transistors in 1961. The first DRAM cell was fabricated in 1968. One of the most important VLSI milestones was the fabrication of the first microprocessor which contains 100 transistors per chip in 1971. VLSI era started in 1980 by fabricating more than 200 K transistor per chip. FPGA was invented in 1985. System-on-chip (SoC) and intellectual property (IP) era started in 1995 by integration of more than 100 M transistor per chip. Recently in 2004, 3D integration era started. Table 1.1 summarizes the most important terms in VLSI. A SoC design is a "product creation process" which starts at identifying the end-user need and ends at delivering a product with enough functional satisfaction from the end user. A typical SoC contains hardware and software as depicted in Fig. 1.2. An example for the SoC architecture is shown in Fig. 1.3. Benefits of using SoC are reducing overall system cost, increasing performance, lowering power consumption, and reducing size. The advantages and disadvantages of SoC are summarized in Table 1.2 [2].

The early predecessor of a SoC was the Single Board Computer (SBC). All required logic was integrated on a single board (Fig. 1.4). When it became possible to integrate more logic into ICs, memory, and some peripherals were integrated into the microprocessor chip. The result is called "microcontroller." A single board computer with microcontrollers contains fewer chips and becomes cheaper. However, still additional logic and peripherals are necessary, since a microcontroller does not contain all required peripherals for most applications (Fig. 1.5). With the availability of programmable logic, the discrete logic ICs (costly and require board space and several extra wires) could disappear (Fig. 1.6). The FPGAs of today include microprocessor core, memories, and enough logic to include all kinds of peripherals (Fig. 1.7) [3].

K.S. Mohamed, *IP Cores Design from Specifications to Production*,
Analog Circuits and Signal Processing, DOI 10.1007/978-3-319-22035-2_1

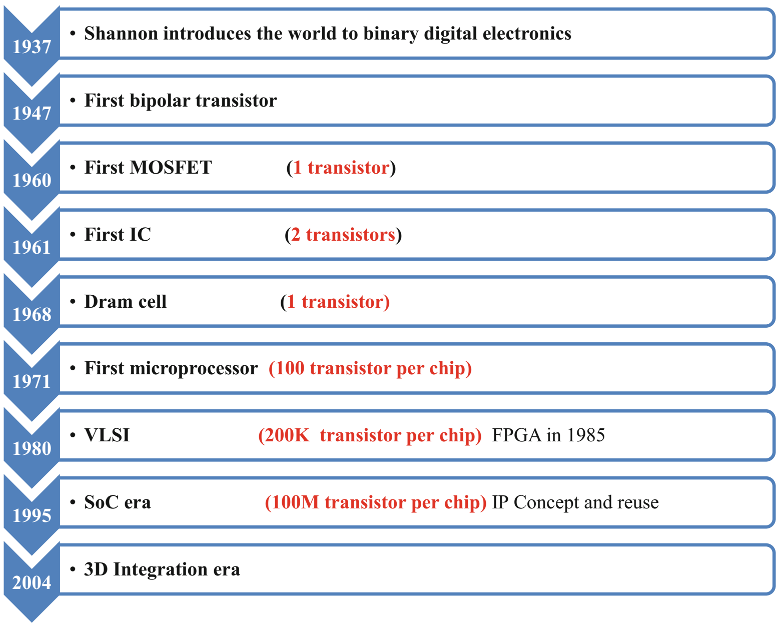

1937
• Shannon introduces the world to binary digital electronics
1947
• First bipolar transistor
1960
• First MOSFET (1 transistor)
1961
• First IC (2 transistors)
1968
• Dram cell (1 transistor)
1971
• First microprocessor (100 transistor per chip)
1980
• VLSI (200K transistor per chip) FPGA in 1985
1995
• SoC era (100M transistor per chip) IP Concept and reuse
2004
• 3D Integration era

Fig. 1.1 The most important milestones in VLSI: it is all about integration

Table 1.1 Important terms in VLSI

What is VLSI?	Integration improves the performance and reduces the cost
What is IC ?	The VLSI final product
What is SoC ?	It is a VLSI design style. Idea: combine several large blocks into one.
What is IP?	Predesigned component can be reused in different SoC. Protected through patents or copyrights
What is EDA tools?	Tools provide the design software used to create all of the world's electronic systems (VLSI, IC, IP, and SoC)

To conquer the complexity of SoC, predesigned components are used (IP reuse) [4]. Hardware IP cores have emerged as an integral part of modern SoC designs. IP cores are predesigned and preverified complex functional blocks. Based on their properties, IP cores can be distinguished into three types of cores: hard, firm, and soft as depicted in Table 1.3 [5, 6], where Soft-cores are architectural modules which are synthesizable and offer the highest degree of modification flexibility, Firm-cores are delivered as a mixture of RTL code and a technology-dependent netlist [7], and are synthesized with the rest of ASIC logic, and Hard-cores are mask and technology-dependent modules. Mapping of IP cores on VLSI design flow is shown in Fig. 1.8. IP core categories tradeoffs are summarized in Fig. 1.9.

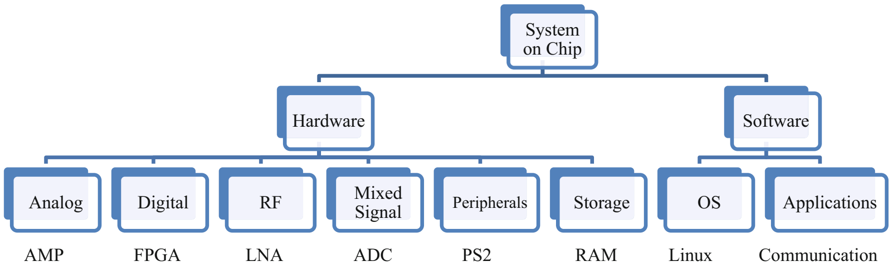


Fig. 1.2 SoC components: it contains hardware and software. Not all software fits on hardware, we have to check the compatibility

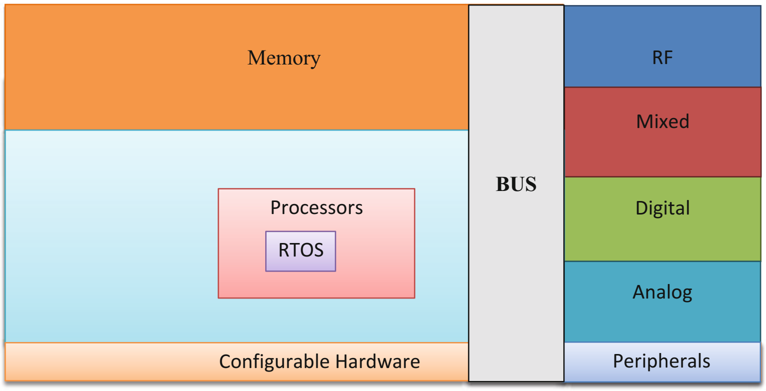


Fig. 1.3 An example of SoC architecture. Different components in single chip (same piece of Si). Many of the components have become standard IP

Table 1.2 Advantages and disadvantages of SOC

Advantages	Disadvantages
– Lower cost per gate	– Increased system complexity
– Lower power consumption	– Increased verification requirements
– Faster circuit operation	– HW/SW co-design
– More reliable implementation	– Integration of analog & RF IPs
– Smaller physical size/area	
– Greater design security	

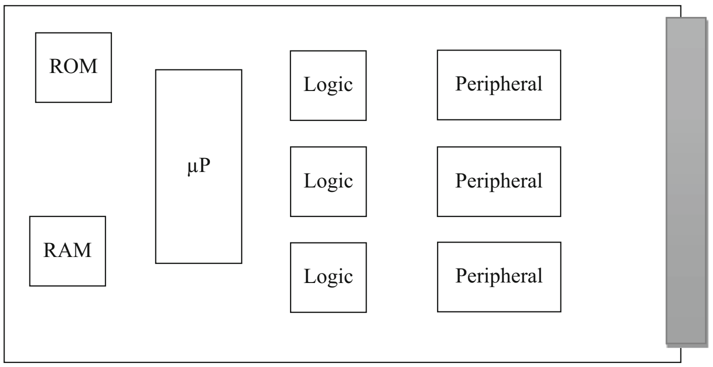


Fig. 1.4 Single board computer

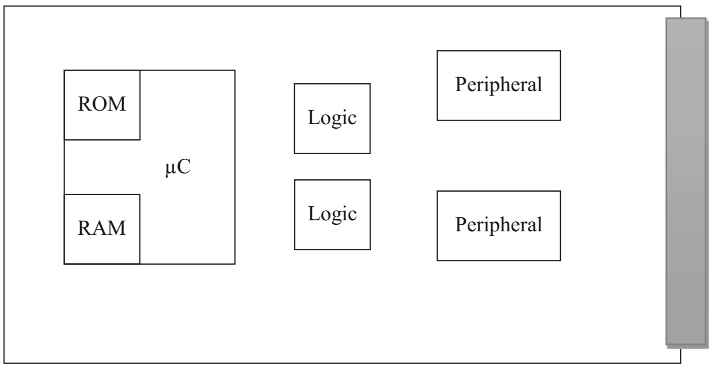


Fig. 1.5 Single board computer with microcontroller

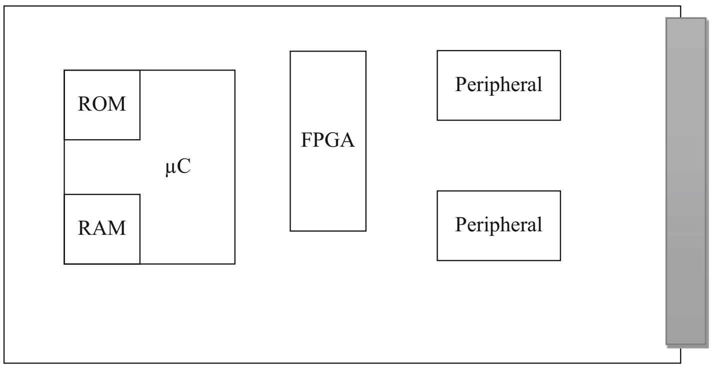


Fig. 1.6 Single board computer with microcontroller and programmable logic

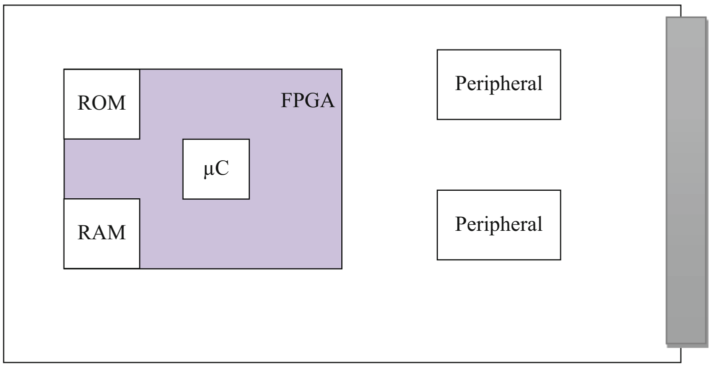


Fig. 1.7 Towards SoC structure

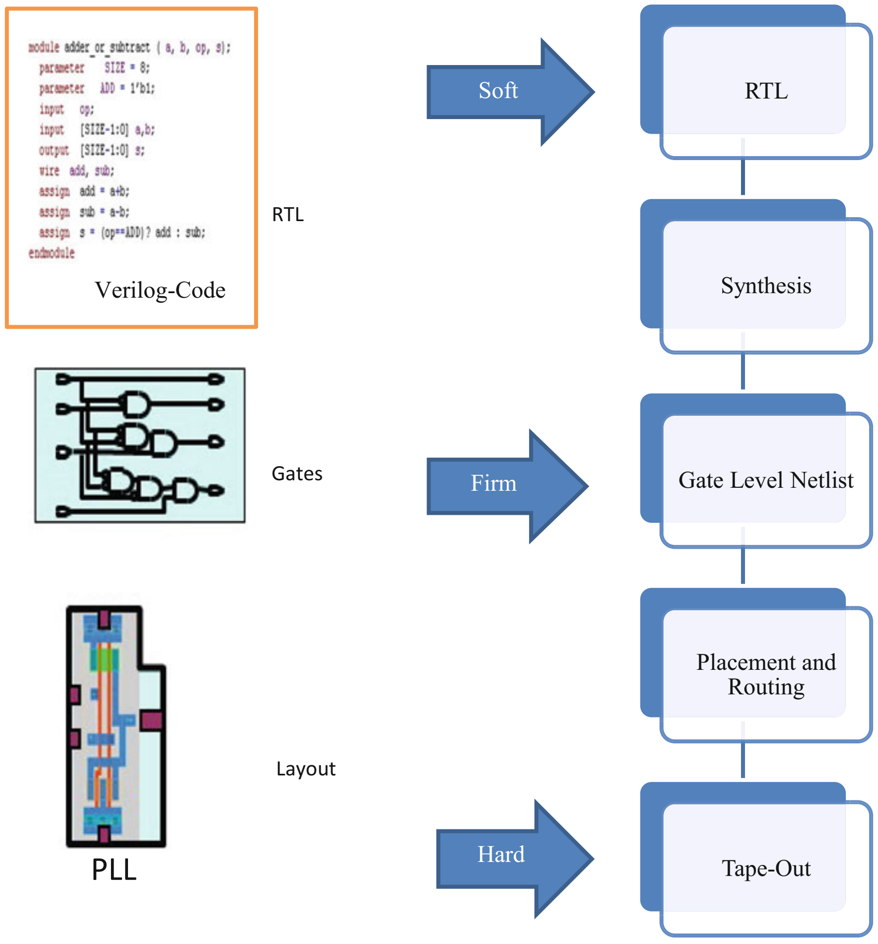
Table 1.3 Classification of hardware IP

IP	Representation	Technology	Optimization	Reuse	Changes
Soft	RTL (HDL)	Independent (Fabless level)	Low	Very high	Many
Firm	Gate level netlist	Independent	Medium	High	Some placement and routing
Hard	GDSII (layout)	Dependent (Fab level)	Very high	low	No

The main differences in design between IC and IP are that, in IC number of input/output (I/O), pins are limited, but in IP it is unlimited. Moreover, in IP we can parameterize IP Design, i.e., design all the functionality in hardware description language (HDL) code, but implement desired parts in the silicon (reusability). These differences are summarized in Table 1.4.

The IC design flow is shown in Fig. 1.10. The first step in IC design is design specification (what customer wants) then we convert the specification to behavioral description. The behavioral description is then converted to RTL description. Then we perform functional verification and if there are any bugs we fix it in the RTL and then do the verification again. If the functional verification is ok, we start synthesizing the RTL code and do the gate level verification. By this, the front-end design is done. The back-end design starts by placement and routing then post-layout verification, we may repeat it if there are any errors until we generate the mask and send it to the fab. After fabrication, chip testing is done.

There is a lot of SoC applications and corresponding IPs as shown in Table 1.5, where industry segments: including mobile communication, automotive, imaging, medical, and networking [8].



```
module adder_or_subtract ( a, b, op, s);
  parameter   SIZE = 8;
  parameter  ADD = 1'b1;
  input   op;
  input   [SIZE-1:0] a,b;
  output  [SIZE-1:0] s;
  wire  add, sub;
  assign  add = a+b;
  assign  sub = a-b;
  assign  s = (op==ADD)? add : sub;
endmodule
```

Fig. 1.8 IP cores in a typical VLSI design flow

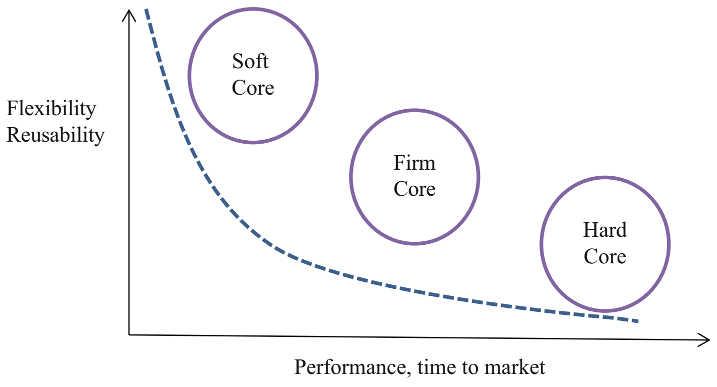


Fig. 1.9 IP cores categories tradeoffs [5]

Table 1.4 Differences between IP and IC

	IP	IC
I/O	Unlimited	Limited
Reusability/parameterization	✓	×

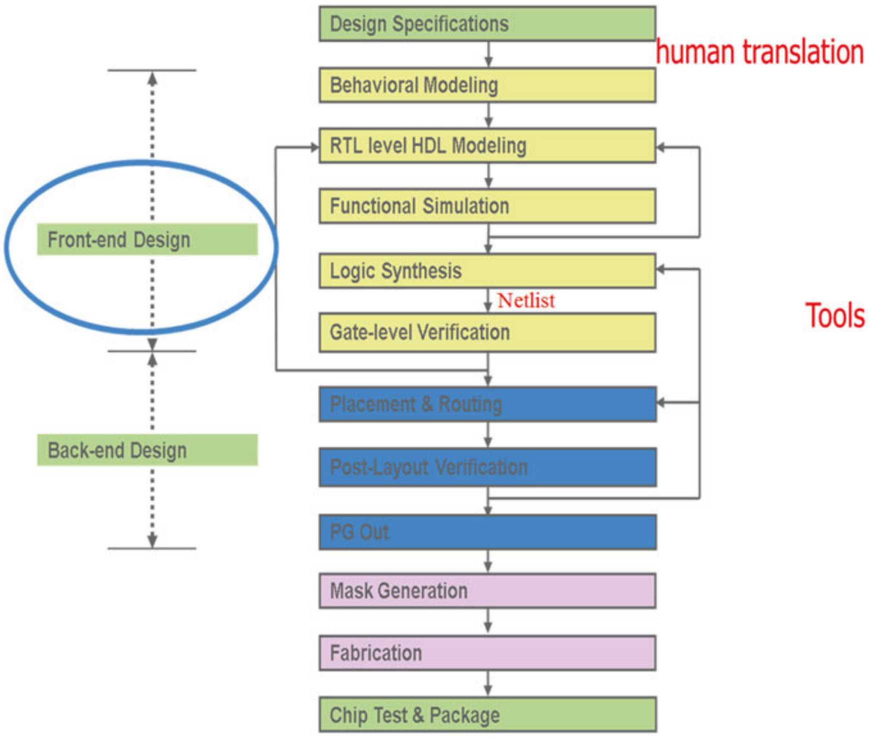


Fig. 1.10 A simplified high-level overview of IC design flow. PG stands for pattern generation

The complete picture for electronic systems is described in Figs. 1.11 and 1.12. For System with multiple SoCs, globally asynchronous locally synchronous (GALS) interconnect concept is used to simplify its design (Fig. 1.13). GALS aims at filling the gap between the purely synchronous and asynchronous domains [9].

IP cores life cycle process from specification to production includes four major steps: (1) IP Modeling, (2) IP verification, (3) IP optimization, (4) IP protection. These steps are elaborated in Fig. 1.14 [11].

IP life cycle is completed with the help of computer aided design (CAD)/electronic design automation (EDA) tools. EDA tools provide software to be used to create all of the world's electronic systems (VLSI, IC, IP, and SoC). The EDA tools play a vital rule in converting an IP specification to an IP product [10].

Table 1.5 SOC applications and IPS examples

Category	IP
Processors	ARM
DSP	MPEG4, Viterbi
I/Os	PCI, USB
Mixed signal	ADC, DAC, PLL
Multimedia	HDMI
Memories	DRAM controller, flash memory
SoC Buses	AHB
Miscellaneous	UART, Ethernet MAC

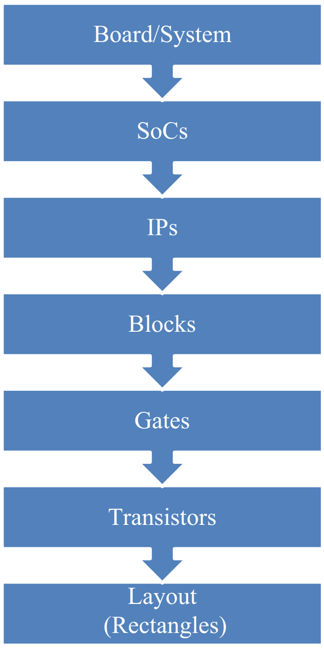


Fig. 1.11 Electronic systems level from board to transistors

Anatomy of EDA Tools: CAD + TCAD. TCAD tools are used for fabrication process, where it simulates the electrical characteristics of semiconductor devices. The EDA tools can be categorized according to the functionality:

1. Design entry (capture tools)
2. Synthesis tools
3. Simulation tools
4. IC physical design & layout tools
5. IC verification tools
6. PCB design & analysis tools

The most famous EDA companies are SYNOPOSYS, MENTOR GRAPHICS, and CADENCE.

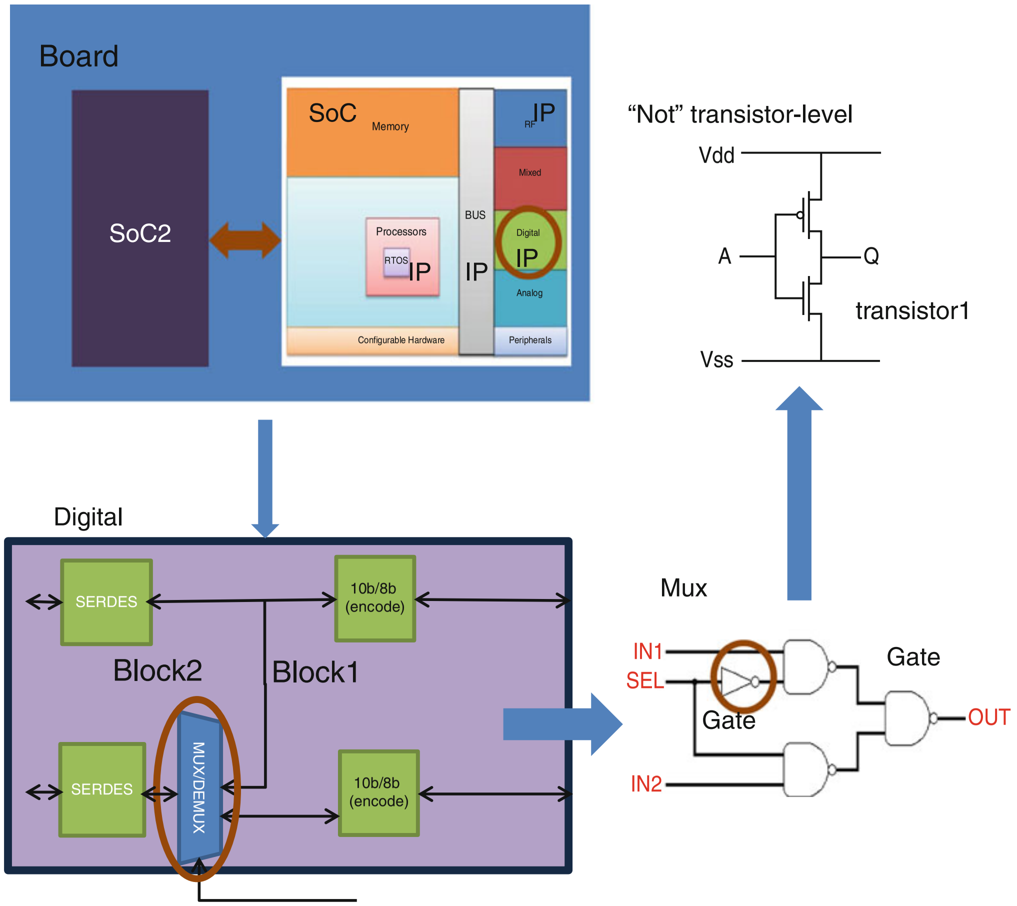


Fig. 1.12 Detailed electronic systems level, where a single board contains number of SoCs and each SoC consists of a number of IPs, these IPs consist of a number of blocks which consist of a number of gates. Gates consist of a number of transistors [8]

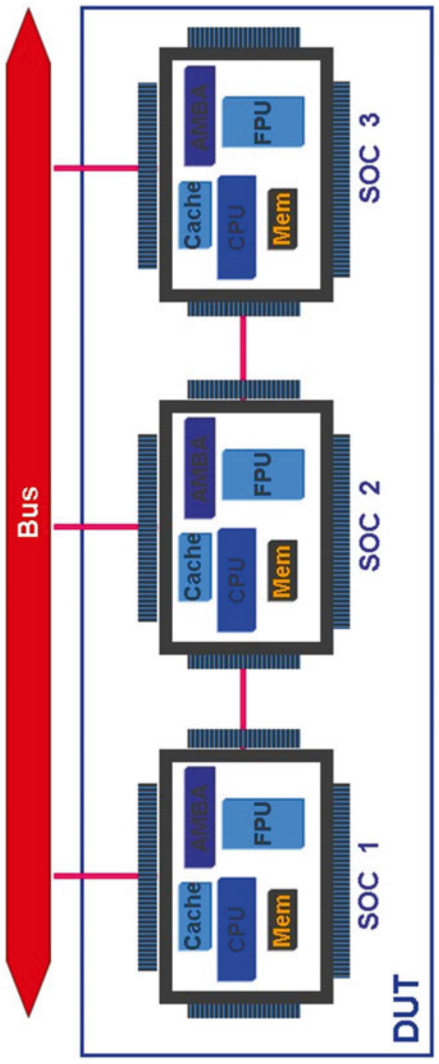


Fig. 1.13 System with multiple SOCs. Synchronous modules on a chip communication asynchronously [8]

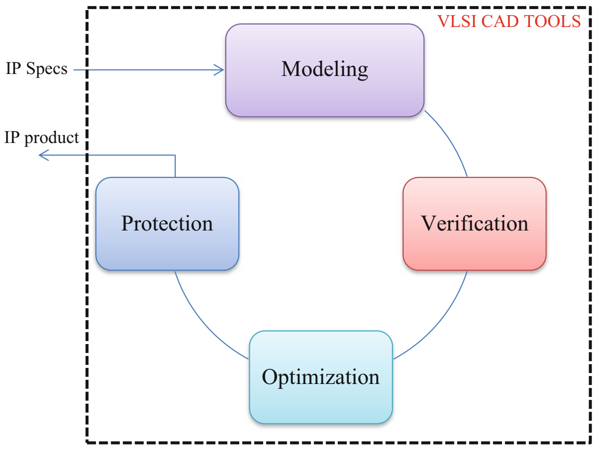


Fig. 1.14 IP core life cycle process: includes four major steps: (1) IP modeling, (2) IP verification, (3) IP optimization, (4) IP protection. Complete process from initial requirements through to finished product. These cycles or flow are done with the help of VLSI CAD tools

References

1. Lojek B (2007) History of semiconductor engineering. Springer, Heidelberg
2. Rajsuman R (2009) System-on-a-chip. Artech House, London
3. http://ce.sharif.edu/courses/88-89/1/ce757-1/resources/root/Slides/lec11.pdf. Accessed 2014
4. dic.csie.ncku.edu.tw/vlsi…/Introduction_to_SOC.pdf. Accessed 2014
5. Hoon Choi, Myung-Kyoon Yim, Jae-Young Lee, Byeong-Whee Yun, and Yun-Tae Lee (2006) Formal verification of a system-on-a-chip. ICCD 2000, Austin. pp 453–458
6. Xu J (2005) Obstacle-avoiding rectilinear minimum-delay Steiner tree construction towards IP-block-based SOC design. ISQED
7. Kong Weio Susanto (2003) A verification platform for a system on chip. University of Glasgow, Glasgow
8. Wolf M (2013) Computer as components: principles of embedded computing system design, 3rd edn. Morgan Kaufman, Burlington. ISBN 978-0-12-388436-7
9. Teehan P, Greenstreet M, Lemieux G (2007) A survey and taxonomy of GALS design styles. IEEE Des Test Comput 24(5):418–428
10. Mathaikutty DA, Shukla S (2009) Metamodeling-driven IP reuse for SoC integration and microprocessor design. Artech House, Norwood
11. Salah K, AbdElSalam M (2013) IP cores design from specifications to production. 25th International Conference on Microelectronics (ICM). IEEE, Beirut

Chapter 2
IP Cores Design from Specifications to Production: Modeling, Verification, Optimization, and Protection

2.1 Introduction

As stated earlier in the previous chapter, plug and play IP in SoC design is the recent trend in VLSI design (Fig. 2.1). IP cores life cycle process from specification to production includes four major steps: (1) IP modeling, (2) IP verification, (3) IP optimization, (4) IP protection. These steps are elaborated in Fig. 2.2. In the next sections, we will discuss each step in detail.

2.2 IP Modeling

To model an IP, we have four design modeling methodologies as depicted in Fig. 2.3 [1–6]:

1. FPGA-based Modeling: defined by fixed functionality and connectivity of hardware elements.
2. Processor-based Modeling: Processor running programs written using a predefined fixed set of instructions (ISA).
3. ASIC-based Modeling: Silicon-level Layout.
4. PCB-based Modeling: it uses standard ICs such as 74xx (TTL), 40xx (CMOS), it is not VLSI, it is just discrete components.

The comparison between theses typical hardware options is shown in Table 2.1. Choice of any option depends on application and requirements.

K.S. Mohamed, *IP Cores Design from Specifications to Production*,
Analog Circuits and Signal Processing, DOI 10.1007/978-3-319-22035-2_2

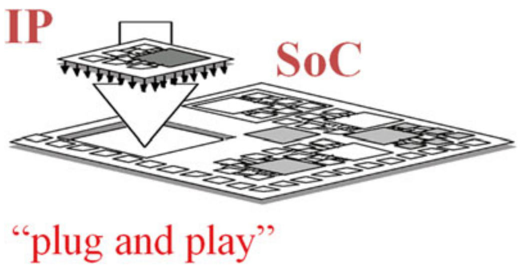


Fig. 2.1 Plug and play IP in SoC design

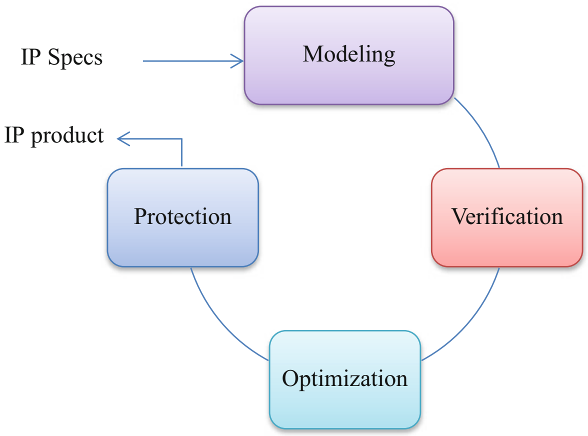


Fig. 2.2 IP core life cycle process: includes four major steps: (1) IP modeling, (2) IP verification, (3) IP optimization, (4) IP protection

Table 2.1 Comparison between different types of hardware

	Processor				
	GPP	DSP	ASIC	FPGA	PCB
Examples	μP, μC	MAC, FFT	–	–	–
Software/hardware	Software	Software	Hardware	Hardware	Hardware
Spatial/temporal	Temporal	Temporal	Spatial	Spatial	Spatial
Functionality	Programmable	Programmable	Fixed	Programmable	Fixed
Time-to-market	High	High	Low	High	Medium
Performance	Low	Medium	High	Med-high	Low
Cost	Low	Medium	High	Low	Low
Power	High	Medium	Low	Low-med	High
Memory bandwidth	Low	Low	High	High	Low
Companies	Intel-ARM	TI	TSMC	Xilinx-Altera-Actel	Valor
Design alternative	Digital	Digital	Digital analog	Digital	Digital analog
			RF mixed		RF mixed
Languages	C	C	–	Verilog	–
	Assembly			VHDL	

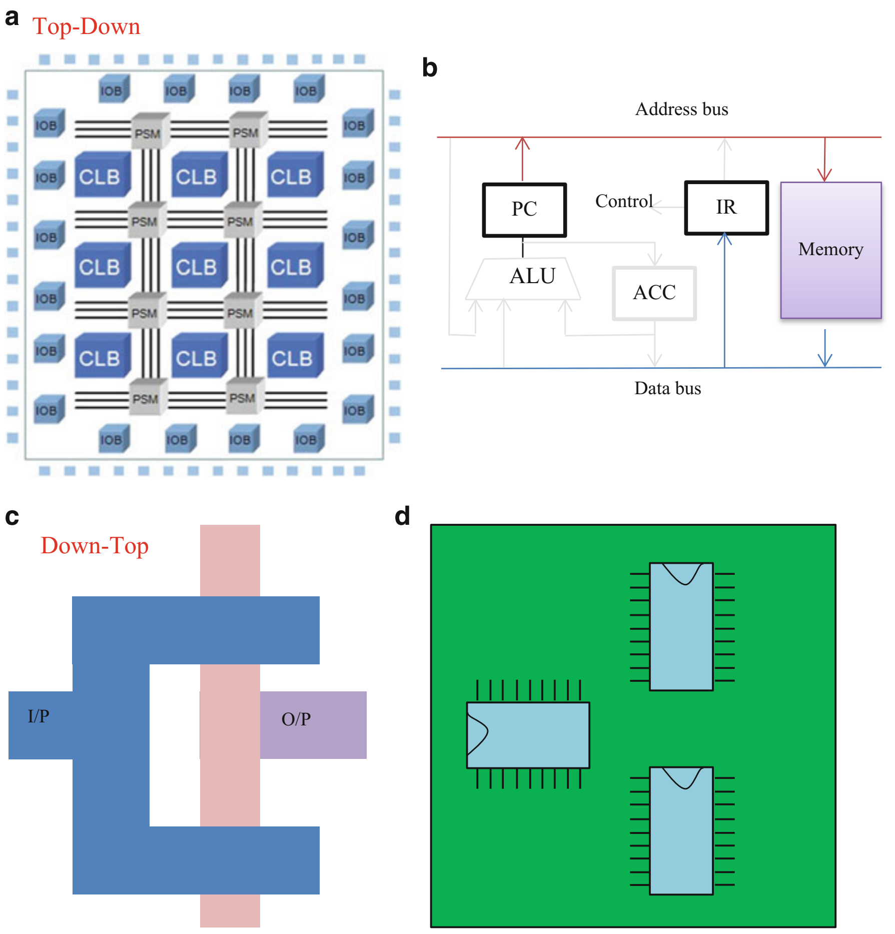


Fig. 2.3 (**a**) FPGA-based modeling, (**b**) processor-based modeling, (**c**) ASIC-based modeling, (**d**) PCB-based modeling

2.2.1 FPGA

FPGAs are programmable chips, compared to hard-wired chips, FPGAs can be customized as per needs of the user by programming. This convenience, coupled with the option of reprogramming in case of problems, makes the programmable chips very vital choice. Other benefits include instant turnaround, low starting cost, and low risk. FPGA means "The chip that flip-flops." An FPGA is like an electronic breadboard that is wired together by an automated synthesis tool. An example of a programmable function using FPGA is shown in Fig. 2.4. A 3-input lookup table (LUT) can implement any function of three inputs.

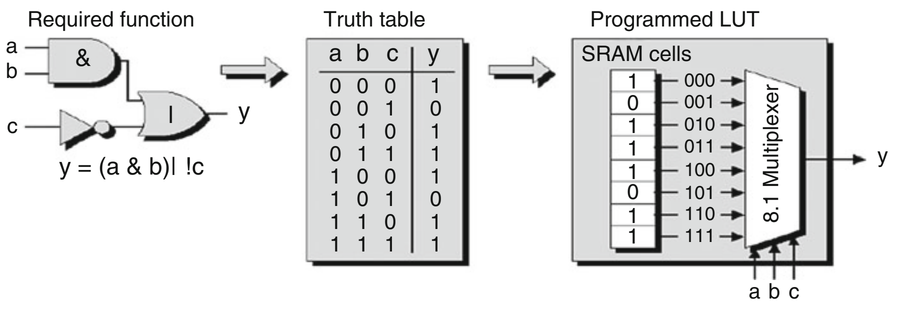


Fig. 2.4 Programmable function using LUT-based FPGA [7]

Referring to Fig. 2.3a, the general architecture of FPGA is shown where, CLB: Configurable Logic Block, IOB: Input/Output Block, and PSM: Programmable Switch Matrix. CLBs provide the functional elements for implementing the user's logic. IOBs provide the interface between the package pins and internal signal lines. Routing channels provide paths to interconnect the inputs and outputs of the CLBs and IOBs. An example for CLB and PSM architecture is shown in Fig. 2.5 [7–9]. The configurable block can be MUX not only LUT. MUX can implement any function, an example for implementing NOT and XOR function is shown in Figs. 2.6 and 2.7 respectively. Also an example for building a latch is shown in Fig. 2.8. FPGAs can be also classified according to their routing structure. The three most common structures are island-style, hierarchical, and row-based [10]. FPGAs are one-size fits all architectures.

FPGA is considered a top-down methodology (RTL to layout), this methodology makes design of complex systems more simpler as it focuses on functionality, reduce time-to-market as it shortens the design verification loop, and makes exploring different design options easier and cheaper for example (latency versus throughput).

As for modeling languages and the scope of using FPGA-based design, two levels for IP modeling are highlighted register-transfer level (RTL) and transaction level modeling (TLM) (Table 2.2).

RTL is the abstraction level between algorithm and logic gates. In RTL description, circuit is described in terms of registers (flip-flops or latches) and the data is transferred between them using logical operations (combinational logic, if needed). That is why the nomenclature: Register-Transfer Level (RTL). Y-chart is shown in Fig. 2.9.

TLM is a technique for describing a system by using function calls that define a set of transactions over a set of channels. TLM descriptions can be more abstract, and therefore simulate more quickly than the RTL. TLM separates computation from communication as depicted in Fig. 2.10.

Modeling at the transactional level has several advantages, not only for the IP provider (designers and verification engineers), but also for the users, which can evaluate the performances and the behavior of the IP very early in the design flow.

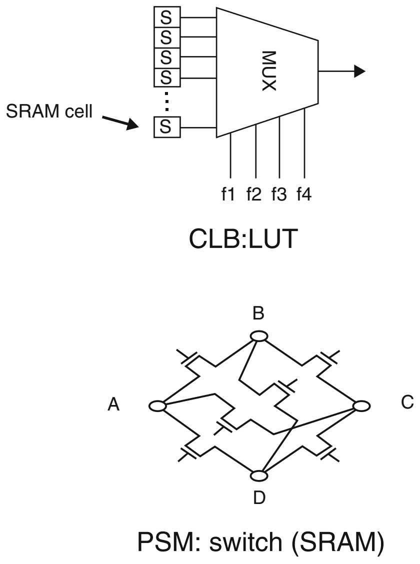


Fig. 2.5 CLB and PSM architecture example [7]

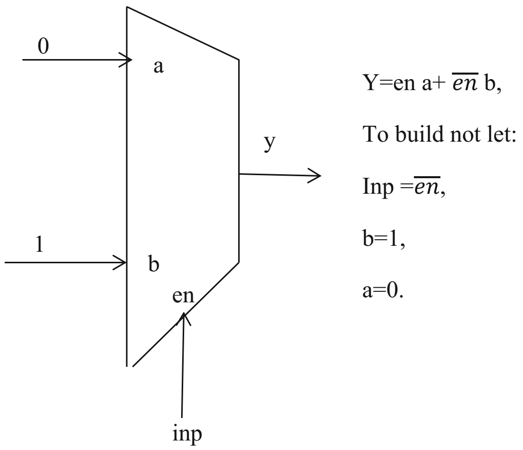


Fig. 2.6 Building NOT function from MUX

The different levels of abstraction and the different modeling languages are shown in Fig. 2.11 and Table 2.3.

System level modeling is widely employed at early stages of system development for simplifying design verification and architectural exploration. Raising the abstraction level results in a faster development of prototypes and the reduction of implementation details in system level design can increase the simulation speed and allow a more global view of the system. During the phase of RTL development, the system level design can serve as a reference model for RTL design and verification.

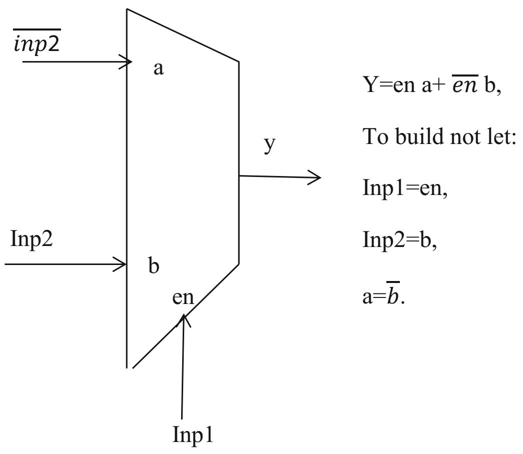


Fig. 2.7 Building XOR function from MUX

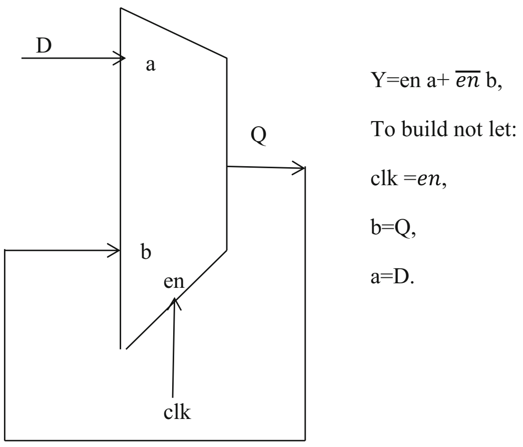


Fig. 2.8 Building LATCH function from MUX

Table 2.2 RTL and TLM comparison

	RTL	TLM
Simulation speed	⟶	
Abstraction level	⟶	
RTL synthesizable	Yes	No
Languages	Verilog, VHDL	Systemverilog, SystemC
Accuracy	⟵	

There are several high-level modeling languages like Systemverilog [11] and SystemC [12]. TLM does not contain a clock signal. TLMs use function calls for communication between different modules and events to trigger communication actions. It allows designers to implement high-level communication protocols for simulations up to faster than at register-transfer level (RTL). Thus encouraging the use of virtual platforms for fast simulation prior to the availability of the RTL code.

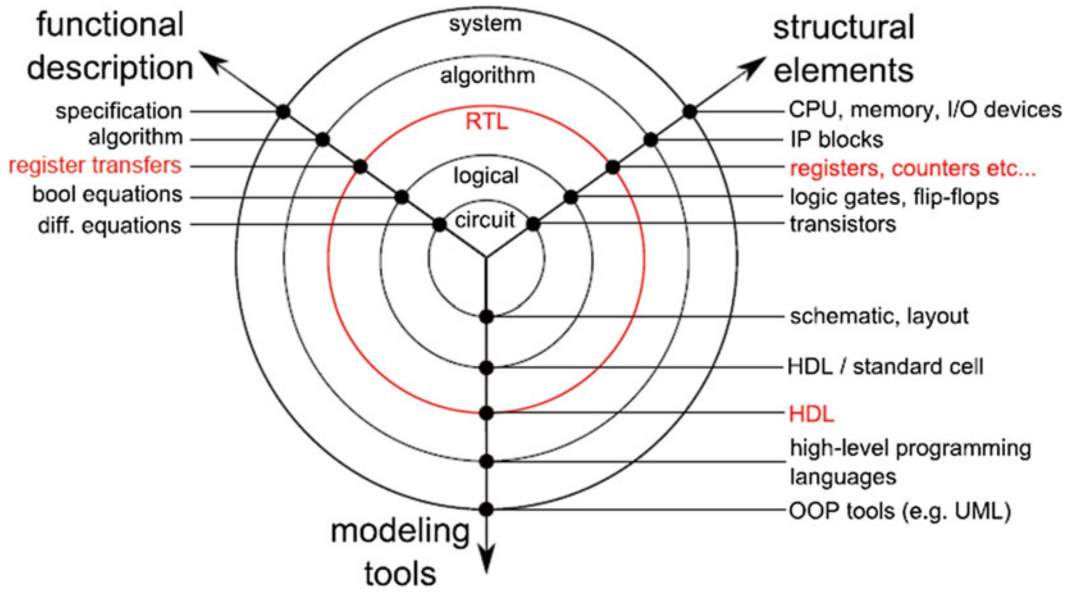


Fig. 2.9 Y-chart for RTL design representation: levels of abstraction (structural, behavioral, physical)

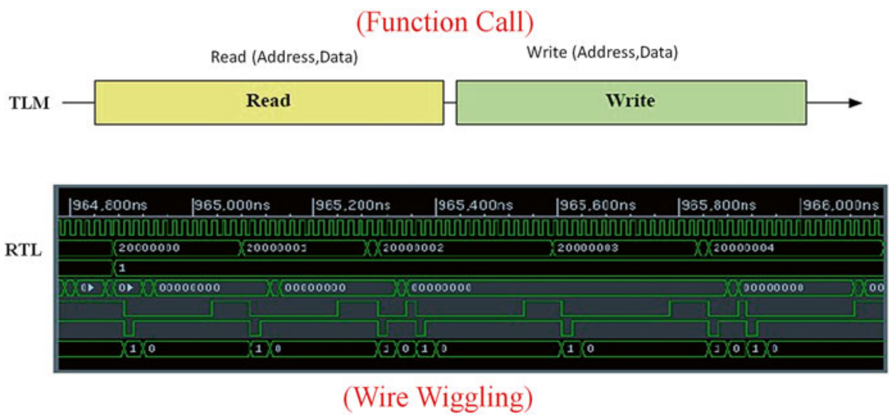


Fig. 2.10 TLM and RTL example, where TLM does not take into consideration the details, i.e., higher abstraction level. TLM replaces all pin-level events with a single function call. TLM speeds up verification

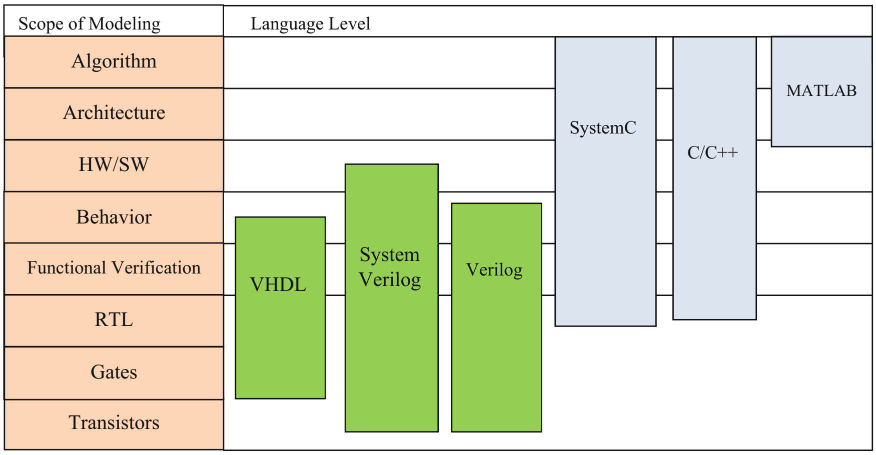


Fig. 2.11 Comparison between different modeling languages [4]

Table 2.3 The modeling languages comparison

	MATLAB	SystemC	Systemverilog	Verilog	VHDL
Requirements	Yes	YES	No	No	No
Architecture	Yes	Yes	No	No	No
HW/SW	No	Yes	No	No	No
Behavior	No	Yes	Yes	No	Yes
Functional verification	No	Yes	Yes	No	No
Testbench	No	Yes	Yes	Yes	Yes
RTL	No	Yes	Yes	Yes	Yes
Gates	No	No	Yes	Yes	Yes
Transistors	No	No	Yes	Yes	No

Systemverilog suffers from [13]:

1. It is closed source.
2. It is not software domain, i.e., does not support HW/SW co-verification.
3. Single core, no multi-core support.
4. Incomplete support for OOP, for example there is no const class method.
5. It does not support function overloading.
6. No automatic garbage collector.
7. DPI has a long runtime overhead.

SystemC suffers from:

1. Single core, no multi-core support.
2. No coverage support.
3. Transaction randomization is limited.

There is another family of languages called **scripting** languages like PERL [14], TCL [15], and Python [16]. Scripting languages are programming languages designed to make programming tasks easier, for example to run all the test cases automatically after every RTL change to make sure that it does not affect other test cases. Scripting languages are dynamic high-level languages with extensive standard library which enables rapid prototyping and experimentation.

There are advances in design methods such as using IP-XACT. IP-XACT is a standard written in an XML file format to describe hardware designs at a higher level [17, 18]. Also, it provides a standard for component design description exchange among heterogeneous platforms or among different designers working on different components or in other words, it helps in IP reuse.

The XML document is written using XML editors and it contains set of tags which represent a synthesizable hardware component such as registers and FIFO. IP-XACT documents the attributes of an IP component such as Interfaces, signals, parameters, memory, ports, and registers. An XML parser interprets the document and generates RTL code as XML is just plain text. The parsing process of an XML is relatively fast. Python is one of the languages used for parsing [19].

FPGA design flow comprises the following steps:

1. Convert specification to RTL code.
2. Synthesis the code which means converts the RTL code into generic Boolean netlist (gates, wires, registers).
3. Do mapping: map the generic Boolean gates into target technology (LUT or MUX CLB). The RTL can be mapped into FPGA or ASIC as depicted in Fig. 2.12.
4. Placement and routing.
5. Downloading: the file which is generated and downloaded to the FPGA is called bitstream file.

An example for a logic block is shown in Fig. 2.13. The placement process is described in Fig. 2.14 and the routing process is described in Fig. 2.15.

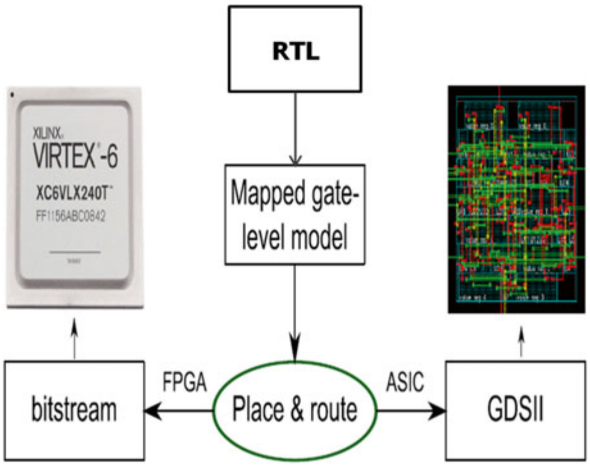


Fig. 2.12 RTL to FPGA or ASIC

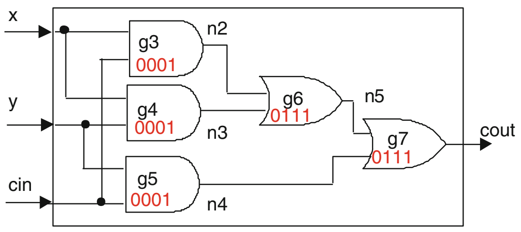


Fig. 2.13 Computed values for truth tables (two input only AND and OR gates logic network)

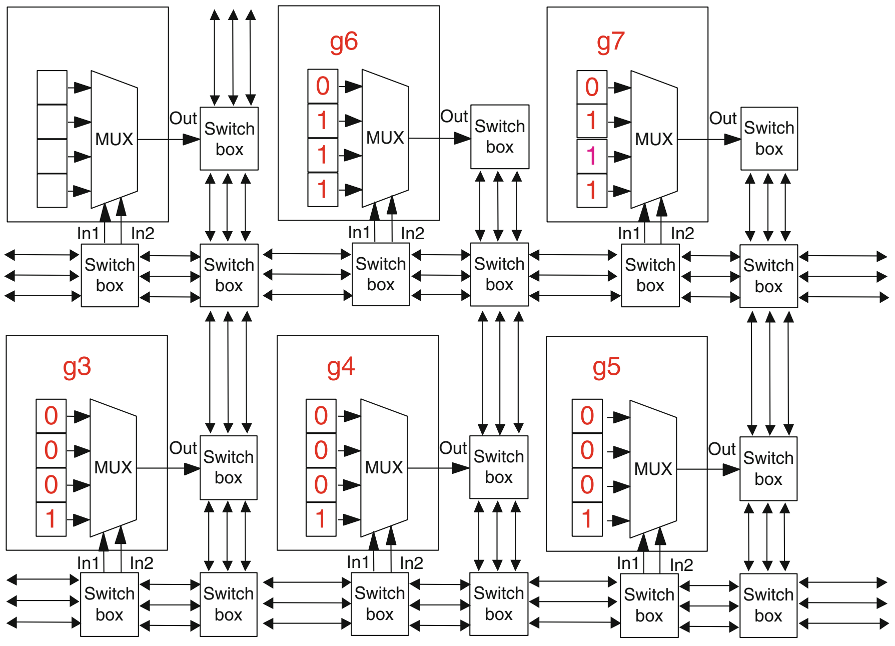


Fig. 2.14 FPGA placing

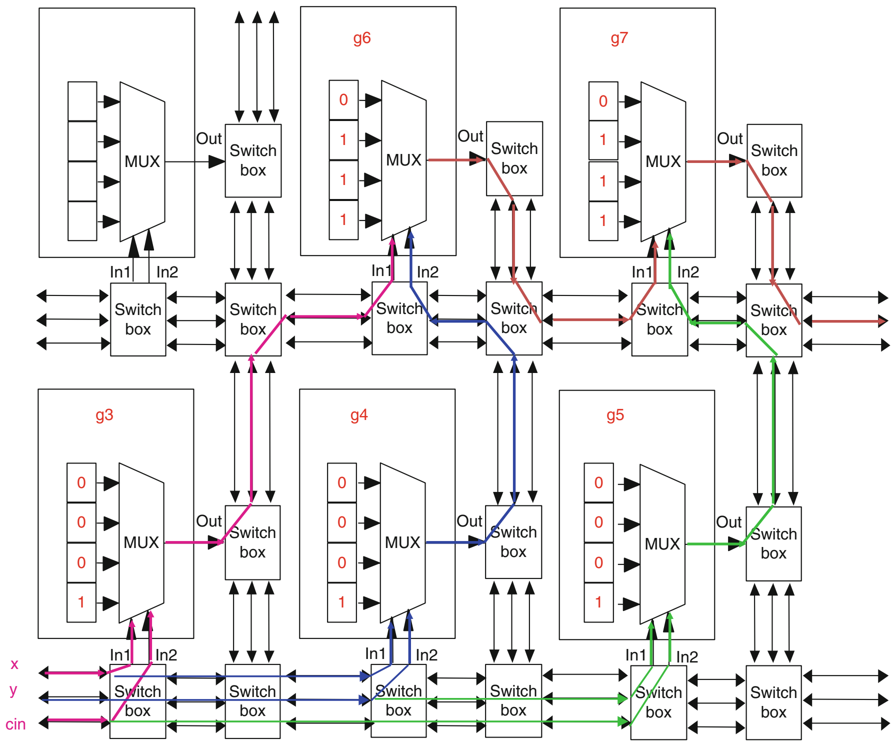


Fig. 2.15 FPGA routing

2.2.2 *Processor*

Referring to Fig. 2.3b, the general architecture for a very simple processor is shown, where PC: program counter, ACC: accumulator, ALU: arithmetic logic unit, IR: instruction register. The PC holds the address of next instruction to be executed, ACC holds the data to be processed, ALU performs operation on data, IR holds the current instruction code being executed. The operation can be summarized in the following steps (Fig. 2.16):

1. Instruction fetch: The value of PC is outputted on address bus, memory puts the corresponding instruction on data bus, where it is stored in the IR.
2. Instruction decode: The stored instruction is decoded to send control signals to ALU which increment the value of PC after pushing its value to the address bus.
3. Operand fetch: The IR provides the address of data where the memory outputs it to ACC or ALU.
4. Execute instruction: ALU is performing the processing and store the results in the ACC. The instruction types include: data transfer, data operation (arithmetic, logical), and program control such as interrupts.

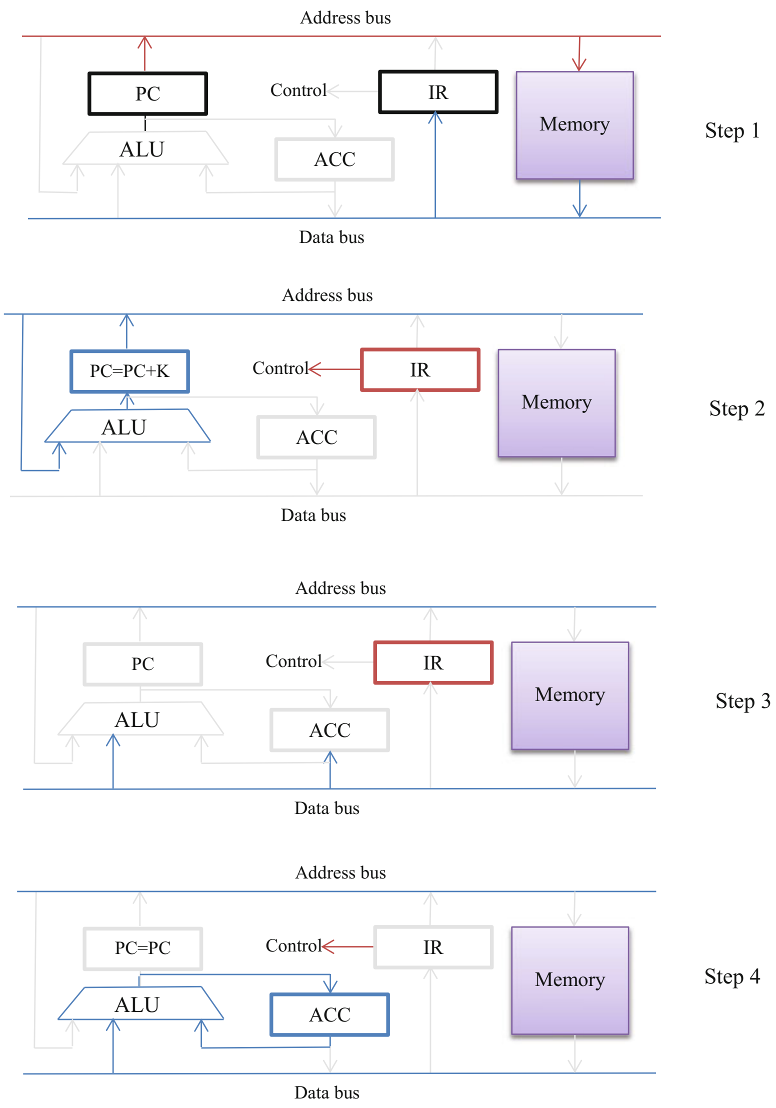


Fig. 2.16 A simple processor operation

Theses cycles are continuous and called fetch–decode–execute cycle. The processors can be programmed using high-level language such as C or mid-level language such as assembly [20]. Assembly is used for example in nuclear application because it is more accurate. At the end the compiler translates this language to the machine language which contains only ones and zeroes.

Instruction Set Architecture (ISA) describes a processor from the user's point of view and gives enough information to write correct programs. Examples of ISA are Intel ISA (8086, Pentium).

2.2.3 ASIC

Physical design converts a circuit description into a geometric description. This description is used to manufacture a chip. Geometric shapes which correspond to the patterns of metal, oxide, or semiconductor layers that make up the components of the integrated circuit. It is top view of the cross-sectional device [21].

Using ASIC design methodology, it is very hard to fix bugs and it needs long time through the fabrication process (Design, Layout, Prototype, Fabrication, and Testing). It requires expensive tools and requires a very expensive Fab. But, it provides superior performance [22]. In ASIC, the schematics is converted to stick diagram to find Euler path which determines the best way to put the devices in the substrate and then the stick diagram is converted to layout (Fig. 2.17). The layout can be analog, digital, or mixed signal. An example for a layout of a simple FET transistor is shown in Fig. 2.18. The layout has some design rules called design rule check (DRC) [23].

Since there are different semiconductor processes (with different set of rules and properties), the designer has to know the specifications for the one that is to be used. This information is stored in a set of files called Technology Files. The technology files contain information about:

- Layer definitions: Conductors, contacts, transistors.
- Design rules: minimum size, distance to objects.
- Display: Colors and patterns to use on the screen.
- Electrical properties: resistance, capacitance.

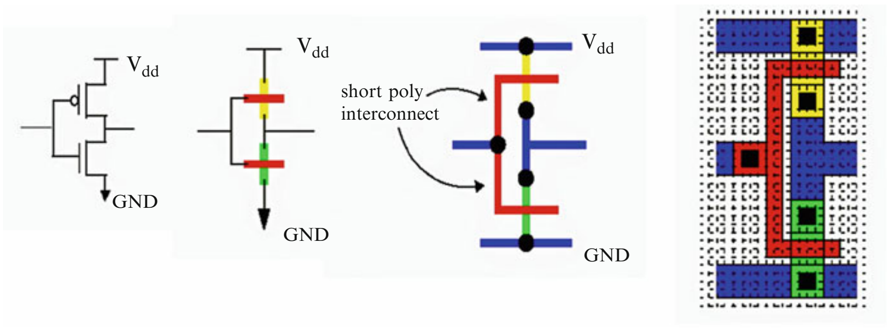
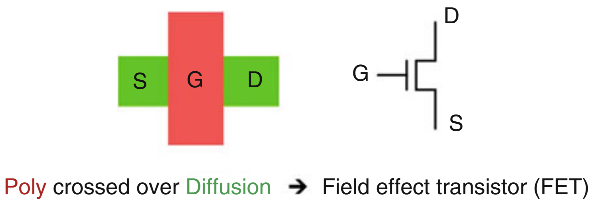


Fig. 2.17 Schematics to stick diagram to layout. A stick diagram is a symbolic layout: contains the basic topology of the circuit. It is always much faster to design layout on paper using stick diagram first before using the layout CAD tool [21]

Fig. 2.18 Layout of simple FET, where source and drain are interchangeable [21]



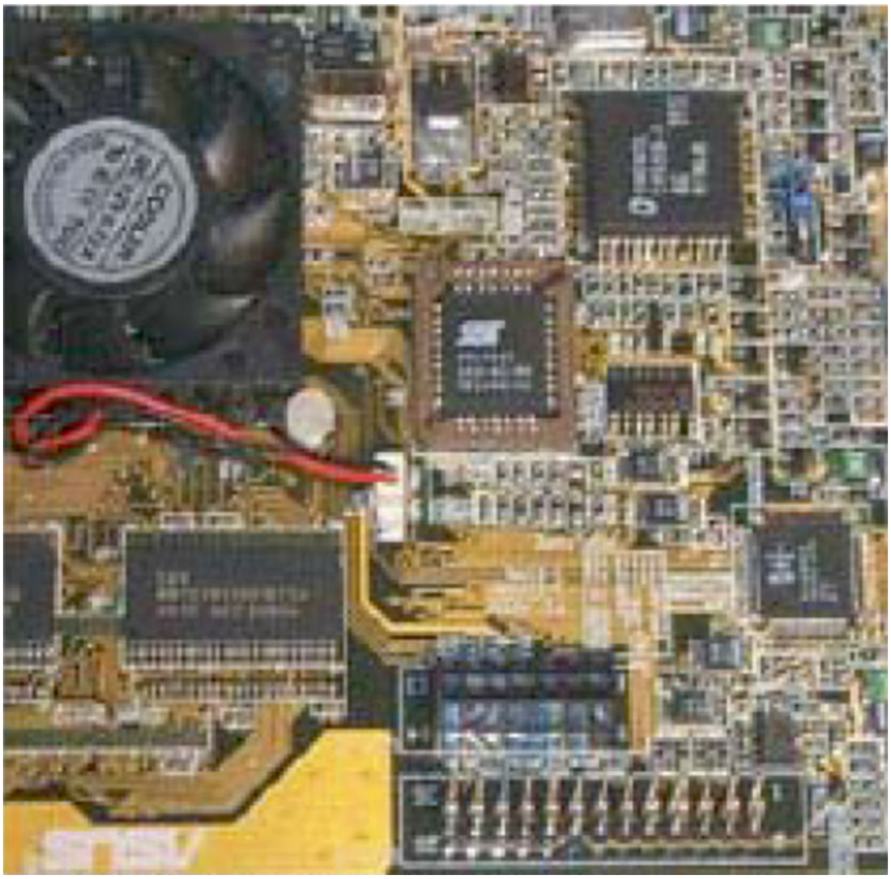


Fig. 2.19 Typical PCB: computer motherboard

The process features example:

- *p*-Type substrate
- *n*-Well
- $n+$ and $p+$ diffusion implants
- One layer of poly (gate material)
- Two layers of metal for interconnection (metal 1 and metal 2)
- Contact (metal 1 to poly or metal 1 to diffusion)
- Via (metal 1 to metal 2)

After finishing the layout, GDS-II file is sent to the fab to be fabricated. This stage is called “Tape out.”

2.2.4 PCB

Standard logic ICs provides fixed function devices which can be connected together on PCB to implement a system. Standard logic ICs has limited speed and limited number of pins. Standard ICs such as 74xx (TTL), 40xx (CMOS). Typical PCB is the computer motherboard as depicted in Fig. 2.19. PCBs are made of copper and dielectric. Copper is an excellent electrical conductor and it is inexpensive material. PCBs can be single-sided, double-sided, or multilayer boards [24].

For single-sided PCB, components are on one side and conductor pattern on the other side. Routing is very difficult.

For double-sided PCB, conductor patterns are on both sides of the board and we connect between the two layers through vias. Via is a hole in the PCB, filled or plated with metal and touches the conductor pattern on both sides. Since routing is on both sides, double-sided boards are more suitable for complex circuits than single-sided ones. It is always better to minimize the number of vias.

For multilayer PCB, these boards have one or more conductor patterns inside the board. Several double-sided boards are glued together with insulating layers in between. For interlayer connections, there is blind via to connects an inner layer to an outer layer and buried via to connects two inner layers. The layers are classified as: Signal layers, Ground plane, and Power plane. Power planes may have special restrictions such as wider track widths

2.3 IP Verification

Verification is a process used to demonstrate the functional correctness of a design (no bugs). The types of bugs are summarized in Fig. 2.20. It is called bugs because in 1942 using the computer to perform calculations, it gave the wrong results. To find out what was going wrong, they opened the computer and looked inside (remember, this was in the “good old days,” and an electromechanical computer was in use). And there they found a moth stuck inside the computer, which had caused the malfunction. The design/verification matrix is shown in Fig. 2.21.

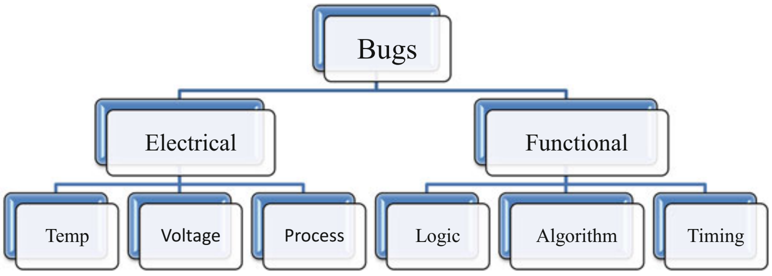


Fig. 2.20 Types of bugs

Fig. 2.21 Design/verification matrix: the cost of verification

	Bad Verification	Good Verification
Bad Design	Many Bugs Exists Bugs Not discovered Bad Reputation at customers	Many Bugs Exists Bugs discovered Time-to-market loss
Good Design	Few Bugs Exists Bugs Not discovered Bad Reputation at customers	Few Bugs Exists All bugs are discovered Customer Happy

2.3.1 FPGA-Based/Processor-Based IP Verification

To verify an IP, we have two options as depicted in Figs. 2.22 and 2.23:

1. Function-based verification
   (a) Simulation-based
   (b) Accelerator-based
   (c) Emulation-based
   (d) FPGA prototyping
2. Formal-based verification
   (a) Assertion-based

IPs functional verification is a key to reduce development cost and time-to-market. Simulation speed is a relevant issue for complex systems with multiple operational modes and configurations since in such cases a slow simulator may prevent the coverage of a sufficient number of test cases in the verification phase [25]. To boost the performance of simulation, a number of platforms have recently attracted interest as alternatives to software-based simulation: acceleration, emulation, and prototyping platforms. Advantages and disadvantages of each type is summarized in Table 2.4, where **simulation** is easy and low cost, but not fast enough for large IP designs. **FPGA prototyping** are fast, but has little debugging capability. **Accelerators** can improve the performance to an extent where, the DUT is mapped into hardware and the testbench is run on the workstation, if we use real host application SW and real OS SW to access the device is called **virtual accelerators**.

Emulation improves the accelerators performance, where the testbench and DUT are mapped into hardware; it also provides efficient debugging capabilities over the FPGA prototyping. The general architecture for the emulator is shown in Fig. 2.24, where many FPGAs are interconnected together for large gate capacity.

There is another mode of operation for the emulator called (in-circuit emulator) ICE, the difference between them can be interpreted by Fig. 2.23f, where in ICE part of the model is a real hardware.

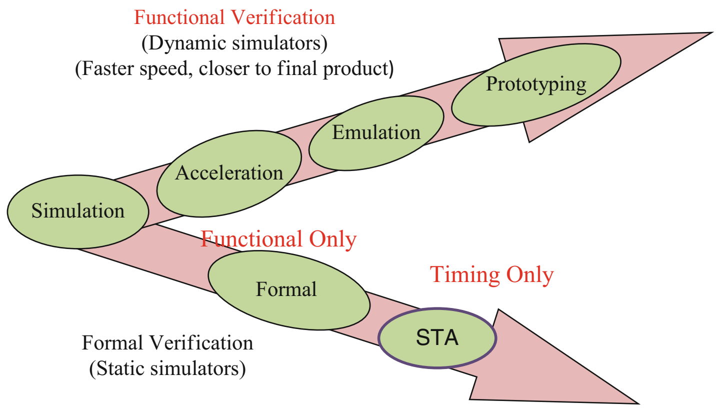


Fig. 2.22 IP cores verification options (platforms)

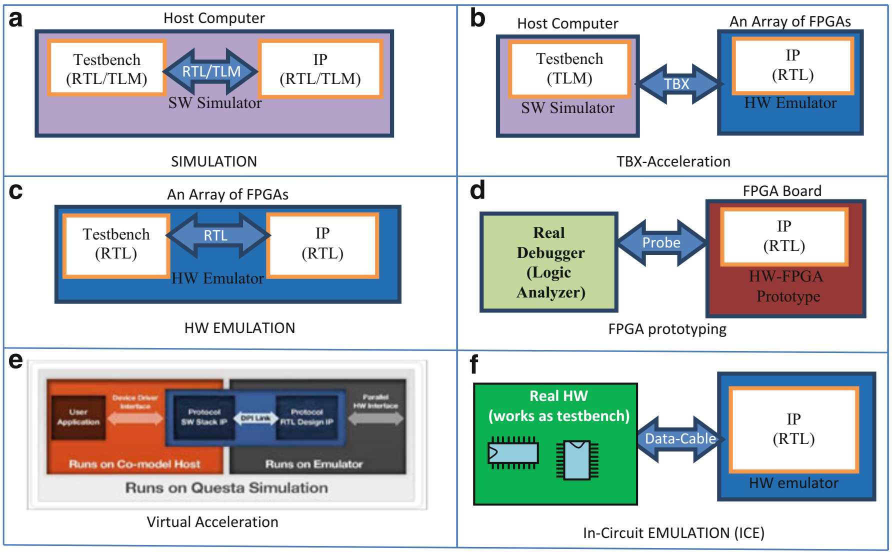


Fig. 2.23 Simulation, accelerators, emulation, FPGA prototyping platform comparison, the IP can be a host or peripheral. (**a**) Simulation, (**b**) TBX-acceleration, (**c**) HW emulation, (**d**) FPGA prototyping, (**e**) virtual acceleration, (**f**) in-circuit emulation (ICE)

The **formal verification** complements simulation-based RTL design verification by analyzing all possible behaviors of the design to detect any reachable error states using assertion-based verification (ABV) methodology and languages like SVA. This exhaustive analysis ensures that critical control blocks work correctly in all cases and locates design errors that may be missed in simulation. Moreover, it is a static simulator, that is why it takes less time in simulation than dynamic ones.

Table 2.4 Simulation, accelerators, emulation, FPGA prototyping comparison

		Simulation	Accelerators	Emulation	FPGA prototyping
Figure		Fig. 2.23a	Fig. 2.23b, e	Fig. 2.23c, f	Fig. 2.23d
Technology	Software	✓	✓	×	×
		Workstation	Workstation		
	Hardware	×	✓	✓	✓
			An array of FPGA's	An array of FPGA's	FPGA
Execution	DUT	Serially	In parallel	In parallel	In parallel
	Testbench	Serially	Serially	In parallel	–
Synthesizable		×	×	✓	–
Supported languages		Verilog	Verilog	Verilog	Verilog
		VHDL	VHDL	VHDL	VHDL
		SystemC	SystemC		
		Systemverilog	Systemverilog		
			C++		
Debugging		– Full RTL-level visibility	– Full RTL-level visibility	– Full RTL-level visibility	– Limited visibility (limited O/P)
		– Workstation	– Workstation	– Workstation	– Logic analyzer
				– Visibility for past time is limited to the depths of the emulator's trace memory	

Logic state	4	2	2	2
Speed	×	10^2×	10^4×	10^5×
	→			
Compilation time			But less than FPGA-based Prototyping as it does not make exhaustive optimization	
	→			
Frequency	→			
Cost	→			
Example	QUESTA	TBX	VELOCE	Xillinx

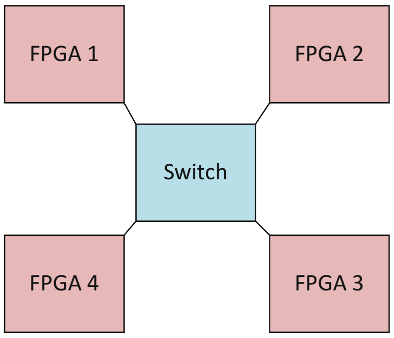


Fig. 2.24 The general architecture for the emulator, where Many FPGA's are interconnected together for large gate capacity

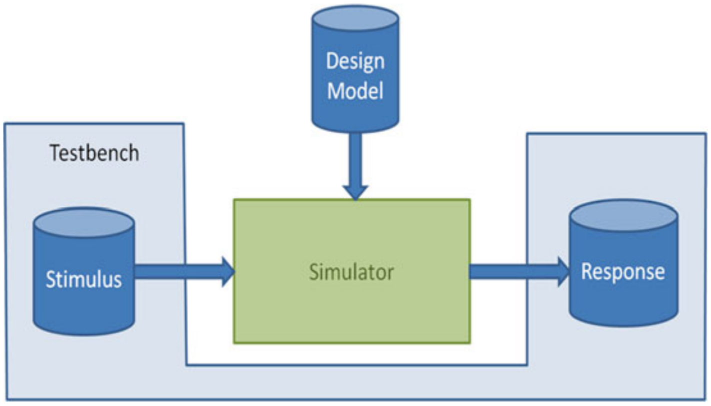


Fig. 2.25 Directed testing. Instantiates design under test (DUT), applies data to DUT, monitors the output

The verification methodologies can be classified into:

1. **Directed testing** (**traditional verification**):

   To ensure that the IP core is 100 % correct in its functionality and timing. Verification engineer sets goals and writes/generates directed tests for each item in Test Plan (Fig. 2.25). If the design is complex enough, it is impossible to cover all features with directed testbenches.
2. **UVM**:

   Reduce testbench development and testing as it supports all the building blocks required to build a test environment as depicted in Fig. 2.26, and it makes multi-master multi-slave testing easier. High-level verification languages and environments such as Systemverilog and e, as used in UVM, may be the state-of-the-art for writing test bench IP, but they are useless for developing models, transactors, and testbenches to run in FPGAs for emulation and prototyping. None of these languages are synthesizable. The component functionalities are as follows:

   - **Sequencer**: Transaction is an instruction from the sequence to the driver (through the sequencer) to exercise the DUT.
   - **Driver**: UVM component that converts a stream of transactions into pin wiggles.

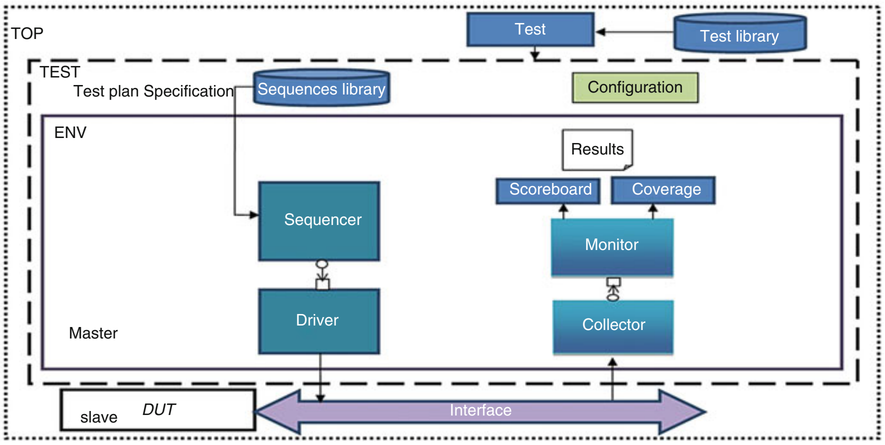


Fig. 2.26 UVM environment

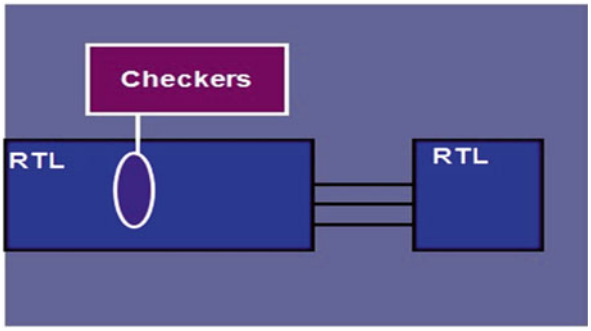


Fig. 2.27 Checkers (assertions)

- **Scoreboard**: Gets a copy of the transaction in the monitor through the Analysis port and use that transaction for analysis purposes.
- **Monitor**: UVM component that monitors the pins of the DUT.

3. **Checkers (assertions)**:

   An assertion is a statement about a specific functional characteristic or property that is expected to hold for a design. The assertion-based methodology is used to ensure the functionality of the IP, where it monitors the transactions on an interface and check for any invalid operation and outputs error and/or warning messing of bus protocol. Self-checking ensures proper DUT response (Fig. 2.27). Assertions enhance observability coverage, making it easier to spot the source of an error [26].

4. **Negative testing (error injection)**:

   Negative testing means "verify that the IP will produce an error report if it sees illegal traffic." The theory on which negative testing is based depending on the "Assertion-based" methodology [27]. The negative testbenches generate illegal traffic; the IP is supposed to recognize this traffic as illegal, and issues the trace error messages (Fig. 2.28).

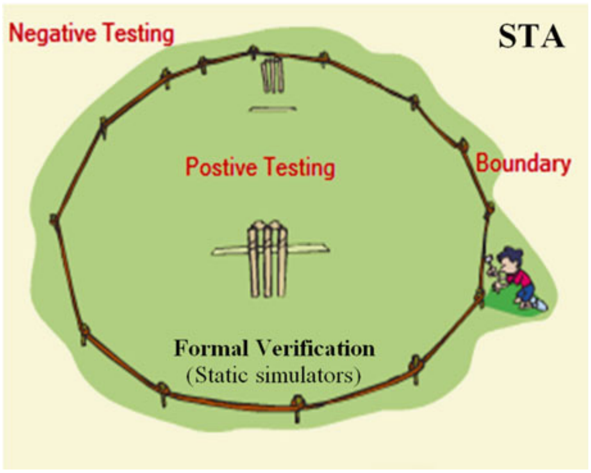


Fig. 2.28 Negative testing [27]

5. **Software-driven testing**:

   Software-driven testing adds a range of capabilities that promise to redraw the functional verification landscape. These include virtual host and peripheral models (called "virtual devices") and software debug technologies enabled by transaction-based, co-model channel technology. Virtual devices are an emerging technology, with products beginning to offer the same functionality as traditional In-Circuit (ICE) solutions, but without the need for additional cables and additional hardware units. Generally the function of virtual device architecture is to package a software stack running on the co-model host workstation with communication protocol IP running on Veloce using a TBX co-model link. This creates protocol solutions so customers can verify their IP at the device driver level and verify the DUT with realistic software, which is the device driver itself as depicted in Fig. 2.29.
6. **Coverage**:

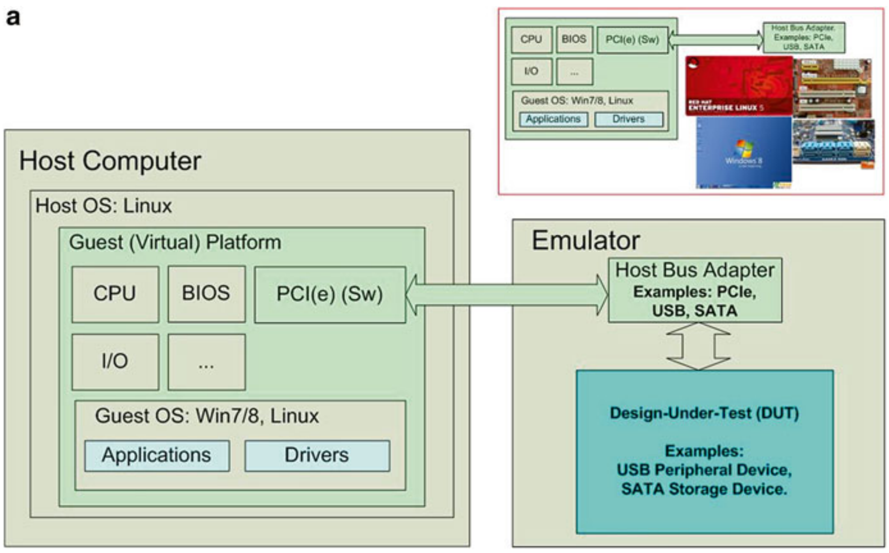
   The main purpose of coverage is to check whether the given property (functional coverage) or statement (code coverage) is covered during simulation/emulation. For example, is the sequence shown in (Fig. 2.30) ever followed by my FSM?
7. **Formal**:

   Input: HDL, post-synthesis gate-level netlist. It checks if the RTL description and the post-synthesis gate-level netlist have the same functionality. It is a static verification [28].
8. **STA**: **static timing analysis**

   Motivation: How can I ensure my design will work at the target frequency under all circumstances?

   How: By ensuring any timing path meets the timing requirements.

   Why: always fastest than a simulation!



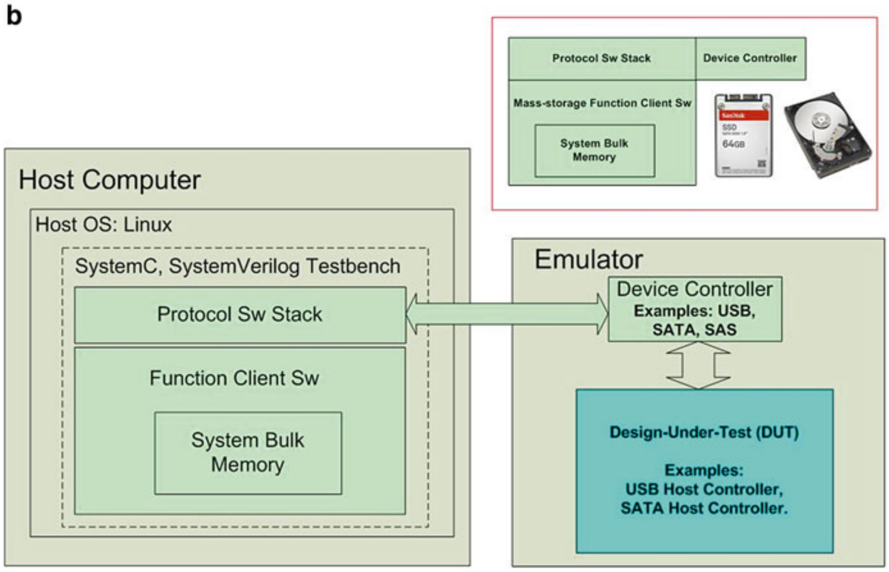
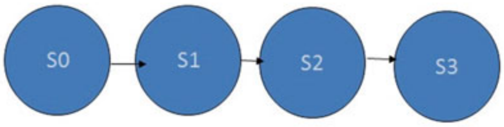


Fig. 2.29 A virtual device packages a software stack running on co-model workstation with communication protocol IP running on Veloce using a TBX co-model link, (**a**) host bus is running on emulator, (**b**) device controller is running on emulator

Fig. 2.30 Property coverage example



Concept: Check the data are available at the right time around the clock edge signal through static timing calculation.
Technique: Delay Calculation **R**, **C** = **f**(**Area**).
Hierarchical analysis is based on timing models for blocks
Notes: STA does not check functionality.

9. **Linting tools**
   Linting tools are widely used to check the HDL syntax before synthesizing it. The input to the linting tool is HDL source and the output is warning and error messages. Linting tools do not detect functional bugs. And they do not need stimulus [29]. They targets:

   - Unsynthesizable constructs.
   - Unintentional latches.
   - Unused declarations.
   - Driven and undriven signals.
   - Race conditions.
   - Incorrect usage of blocking and non-blocking assignments.
   - Incomplete assignments in subroutines.
   - Case statement style issues.
   - Out-of-range indexing.

2.3.2 ASIC-Based IP Verification

It is called physical verification and it includes [30]:

1. Design rule checking (**DRC**):
   DRC checks for if layout complies with foundry rules that is if the layout will be manufacturable. Typically this will have width check, density check, spacing checks, overlap checks, extension checks, etc.
2. Electrical rule check (**ERC**):
   Checks for no short contacts, no floating points, etc.
3. Layout vs. Schematics (**LVS**):
   LVS checks if the layout matches with the reference. In case of full-custom, the reference is spice netlist which is verified for functionality before getting into layout.
4. Post-layout simulation:
   Add the parasitics extracted to the model and resimulate it to make sure that its functionality is still ok.

2.3.3 PCB-Based IP Verification

After drawing the schematic of your circuit and verifying its functionality using any circuit simulator like spice, and after implementing it on PCB, you can verify it using these tips:

1. To perform the PCB verification test, compare the PCB with the layout. During this stage, you might also want to test the connectivity of each traces to ensure no broken traces by using the diode function in the multimeter especially those with buzzer sound. This will ease the verification process as once we hear the buzzer sound, you will know that the trace is connected from one end to another.
2. To check for shorts, look at any suspicious traces that are too close and test using diode function in the multimeter as well. This time, if your buzzer sounds, then you know there is an unwanted shorts [31].

2.4 IP Optimization

The optimization objective is to reduce area, delay, latency, and power and to increase performance and speed to meet the requirement.

2.4.1 FPGA-Based IP Optimization

To optimize an FPGA-based IP, we have three directions [5]:

1. Compilation time optimization.
2. Maximum frequency optimization.
3. Following some RTL design tips.

2.4.1.1 Compilation Time Optimization

Best practice design methodology

- Do not use long loops.
- Store large data in memory not in a register.
- Reduce the use of power "**" and the division "\", instead use log and shift right.
- Do not write long ternary statement "()? : () ? : () ? ...." This very Verilog-based designs.
- Use 2D memory instead of 1D memory as 2-D memory reduce the compile as it is mapped directly to the memory blocks not to the logic.

Use of the latest computer technology

- Parallel (distributed) compilation, use dual or more core feature.

Place-and-route algorithm improvements

- Improve the place-and-route algorithms in the CAD tool development.

2.4.1.2 Maximum Frequency Optimization

Best practice design methodology

1. Make long "Assign" in a clock statement (Pipelining). This is for Verilog-based designs. Note that removing clk cycle to improve latency is easier than inserting one to improve pipelining.
2. Initialization of all uninitialized registers.
3. Using of linting tools such as 0-IN from Mentor Graphics.
4. Make the design under test (DUT) works with posedge clock or negedge clock only, not a mix of them to avoid the half-cycle path. half-cycle path is a path where the data is launched by a flip-flop (FF) on posedge of a clock and captured by a FF on negedge, hence the time available is only half a cycle instead of full cycle where both FF are working on posedge.

2.4.1.3 Follow Some RTL Design Tips

1. **Partition a large memory into several small blocks**
   For example, Questa/Modelsim maximum limit is 2G addresses per memory, so you need to divide the memory if it is higher than 2G as depicted in Fig. 2.31.
2. **Clock gating**
   The concept of clock gating is shown in Fig. 2.32.
3. **Resetting**
   For proper operation we must reset all the registers into the reset process.
4. **FSM coding style**
   The explicit, naive style FSM is better than Mealy or Moore machines as these machines have two distinct disadvantages (Fig. 2.33): (1) they may end with long combinational paths as they don't have output registers. (2) Even worse, if the coding is not done properly latches could be introduced and there will be mismatches between simulation and emulation. So, we strongly recommend a state machine to use a naive style (Fig. 2.34). This way we will have registers for the states and the outputs. For granted this ends up with more

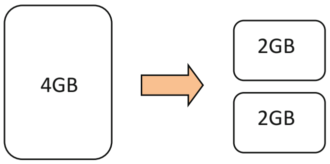
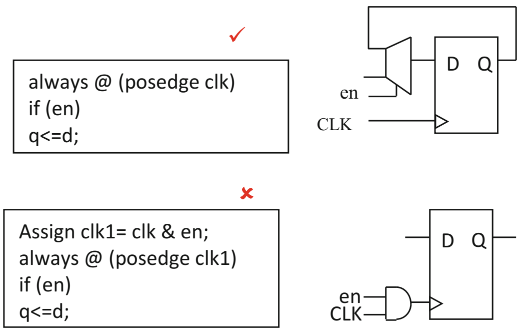


Fig. 2.31 Partition a large memory into several small blocks

Fig. 2.32 Clock gating



✓

```
always @ (posedge clk)
if (en)
q<=d;
```

×

```
Assign clk1= clk & en;
always @ (posedge clk1)
if (en)
q<=d;
```

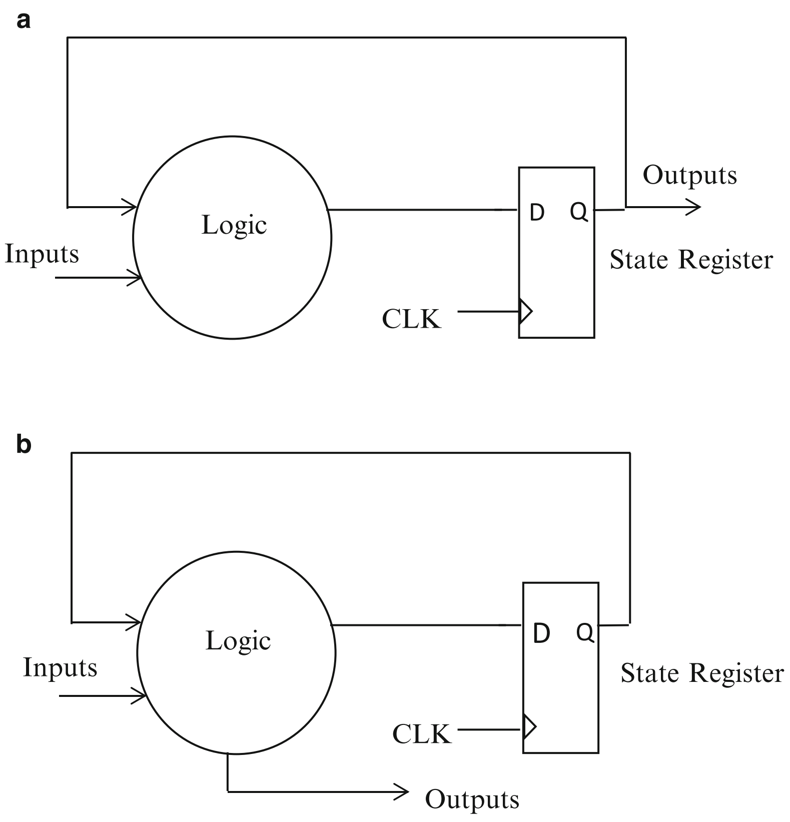


Fig. 2.33 Structures of (a) Moore type FSMs and (b) Mealy type FSMs

registers but it is much, much safer design and it makes it also run at higher frequency as the paths between registers are shorter [2].

Encoding of FSMs including different encoding styles, the most famous one is binary encoding. There is also gray encoding and one-hot encoding. Binary encoding implements very less logic. Also it used minimum number of FFs.

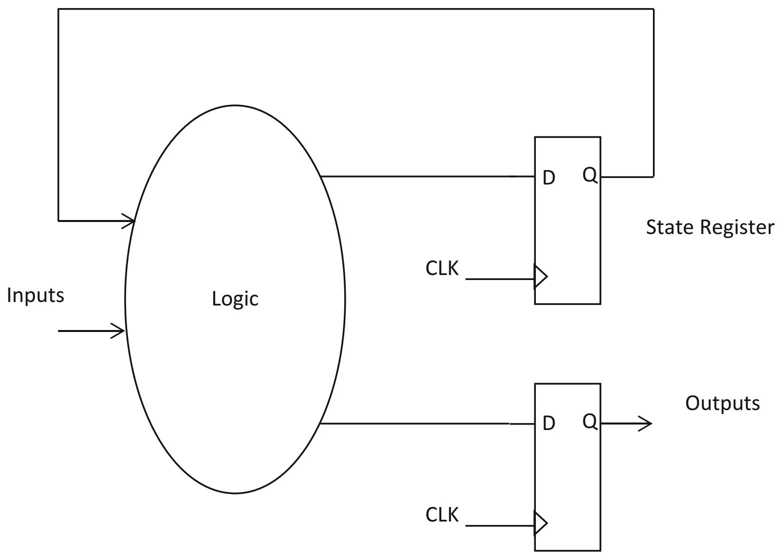


Fig. 2.34 Explicit naive style FSM

Possible state values for a 4 state binary state machine (00, 01, 10, 11). Gray encoding is especially useful when the outputs of the state bits are used asynchronously. This kind of state coding avoids intermediate logics. For example if a state wants to change its state from "01" to "10." In Gray coding between state transitions only one bit will change. Possible state values for a 4 state gray state machine (00, 01, 11, 10).

One-hot encoding uses one flip-flop for each state. For example if there are 10 states in logic then it will use 10 flip-flops. This type of encoding is fast because only one bit needed to check for each state. It implies complex logic and more area inside the chip due to more number of flip-flops. FPGAs are "Flip-flop rich," therefore one-hot state machine encoding is often a good approach. It also reduces hardware's logic switching rate. Possible state values for a 4 state one-hot state machine (0001, 0010, 0100, 1000), also an example of how to write the one-hot encoding FSM is shown in Table 2.5.

Choice of an encoding style is depending of the requirements and performance goals (Table 2.6). Here, one-hot Finite State Machine (FSM) encoding scheme is being adopted for HDL model. One-hot state machines are typically faster, where the logic complexity associated to each state gets decreased. For comparison between binary, gray, and one-hot encoding scheme, one sample state machine was taken with n states. Verilog code was developed using binary and one-hot encoding scheme and then was synthesized to evaluate performance and area. One-hot encoding is a preferred approach if the timing in the output path is critical. Conversion from Binary Encoding to Gray Encoding is shown in Fig. 2.35 [32].

5. **Parameterizing**

Use parameters as much as possible instead of hard-coded values, as it makes verification easier. Parameterization means design all features in HDL code and choose what you want to fabricate. Fixed IP versus parameterized IP is shown

Table 2.5 One-hot encoding verilog example

```
case (1'b1)
state [S0]:
if (in == 1)
next_state [S1] = 1'b1;
else
next_state [S2] = 1'b1;
state [S1]:
if (in == 1)
next_state [S0] = 1'b1;
else begin
next_state [S2] = 1'b1;
state [S2]:
next_state [S0] = 1'b1;
```

Table 2.6 Difference between different FSM encodings

Feature	Binary	Gray	One-hot
Number of flip-flops	#(flip flops) = $\log_2$(#states)	#(flip flops) = $\log_2$(#states)	#(flip flops) = #(states)
	Fewer	Fewer	
Speed	Slower	Slower, only one bit is changed in state transition	Faster
Critical path searching	Need more tracking to find critical path during STA	Need more tracking to find critical path during STA	Easy to find critical path during STA
Debug easiness	Tedious to debug	Tedious to debug	Easy to debug
Low power	Higher power	Suitable for low-power design because of low signal transitions	Higher power

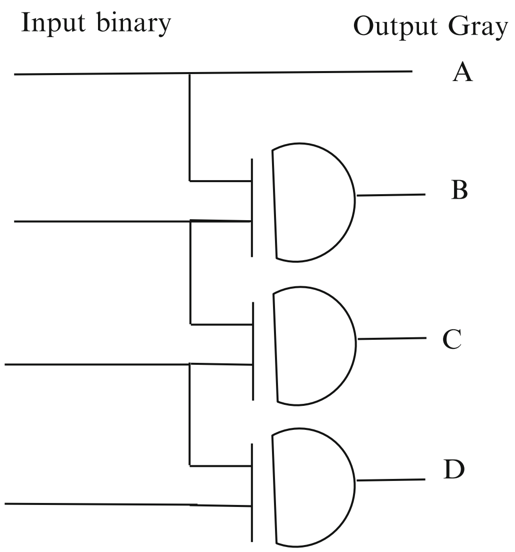


Fig. 2.35 Conversion from binary encoding to gray encoding

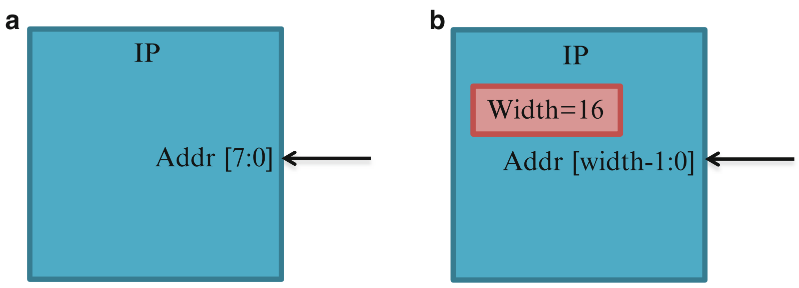
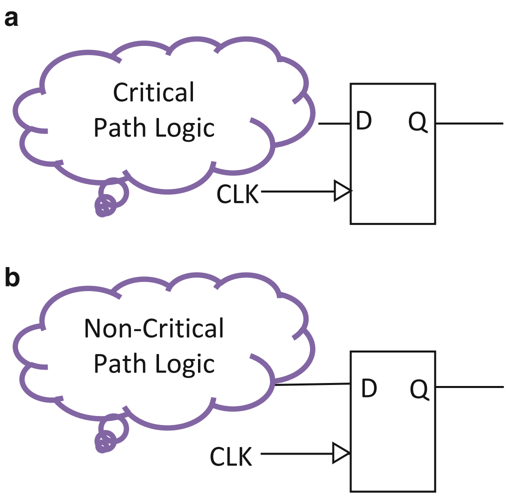


Fig. 2.36 (**a**) Fixed IP versus (**b**) parameterized IP

Fig. 2.37 (**a**) Speed optimization, (**b**) area optimization



in Fig. 2.36. The advantage of parameterization mechanisms over the use of constants/packages is that parameterization allows the same component to be used multiple times in a single design with different sets of parameters [33].

6. **Speed and area optimization**

   Keep critical path logic in a separate module, optimize the critical path logic for speed, and optimize the noncritical path logic for area (Fig. 2.37).

   Dynamic Partial Reconfiguration (DPR) is also used to optimize area usage. With DPR, it is possible to implement different circuits that are not needed at the same time, and that do not operate simultaneously, on the same FPGA area, resulting in considerable area savings as depicted in Fig. 2.38. This area is generally called the reconfigurable region (RR). Whenever the designer wants to change the implemented circuit, an amount of time is needed to rewrite the configuration memory at runtime and this is called the reconfiguration time [34–36]. The subsystem that performs the reconfiguration is called the reconfiguration manager and is generally implemented in software.

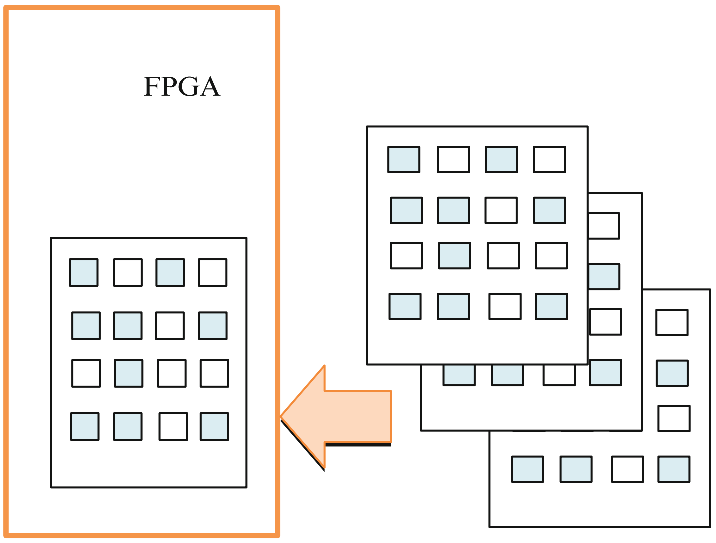


Fig. 2.38 DPR concept, implement different circuits that are not needed at the same time, and that do not operate simultaneously, on the same FPGA area, resulting in considerable area savings

The configuration memory of the reconfiguration region (RR) consists of SRAM memory cells that control the content of the lookup tables and the state of the routing switches. To implement a circuit in the RR, a configuration needs to be generated that contains the binary values that need to be written in the RR's memory cells. Figure 2.39 gives an example that describes the role of configuration memory [37–40].

In conventional DPR systems, a configuration bitstream is generated for every mode by implementing it separately in the RR, where every RR memory cell corresponds to a collection of binary values, one value for each mode. When these binary values are the same, this collection is called a static bit. If they are not the same, this collection is called a dynamic bit. Memory cells containing a static bit do not need to be rewritten during runtime.

The DPR design flow methodology framework comprises a set of steps, which are necessary to implement the proposed multi-mode memory controller's applications using DPR as described in Fig. 2.40.

(a) During the initial phase, the static modules and the partial reconfiguration modules (PRM) are described in HDL language.
(b) The PRMs are synthesized to generate the corresponding netlist for each module.
(c) Perform placement and routing and generation of the full and partial reconfiguration bitstream.
(d) Merges the full bitstream to generate a final downloadable bitstream.
(e) The final downloadable bitstream is copied onto the compact flash card and the card is plugged into the FPGA to bring up the design on the next power cycle.
(f) To switch between the different circuits, the reconfiguration manager writes the reconfigurable region with the appropriate bitstream configuration.

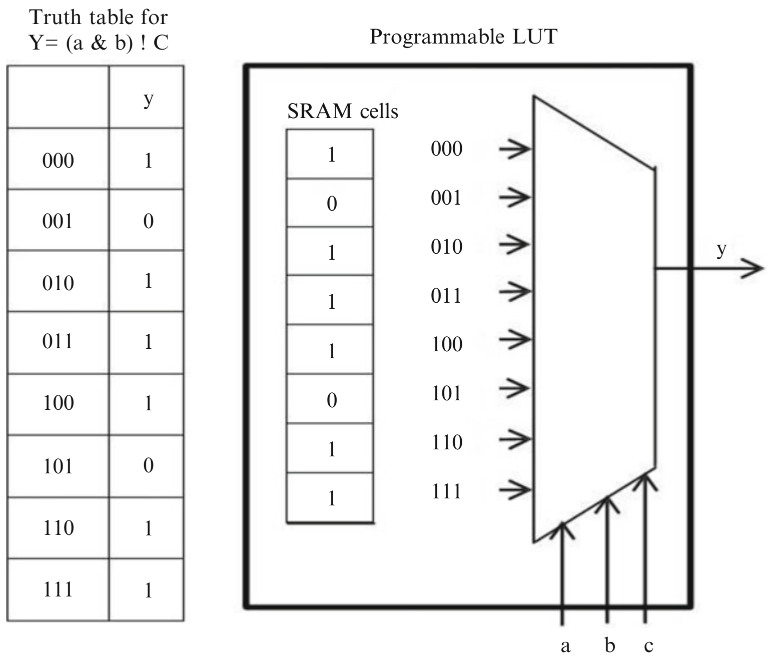


Fig. 2.39 An example describes the role of configuration memory [37]

7. Power optimization
    - Use gray-coding FSM.
    - Use line coding to reduce transitions (8b/10b encoder): reduce α (switching activity factor).
    - Increase data bus width to reduce transfer cycles: reduce α.

2.4.2 Processor-Based IP Optimization

A. **Best practice design methodology**
    1. Do not use long loops.
    2. Split logic circuits to shorten the critical path.
    3. Choose faster logic circuit architectures.

B. **Use of the latest computer technology**
    1. Parallel (distributed) compilation, use dual or more core feature.

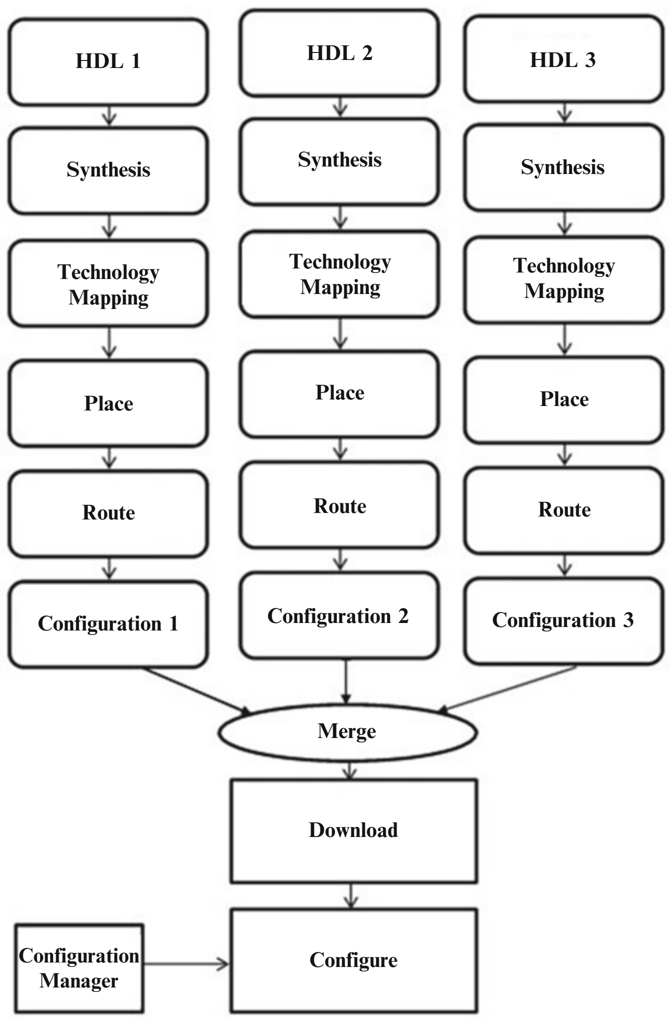


Fig. 2.40 DPR design flow methodology framework. It comprises a set of steps, which are necessary to implement the proposed multi-mode memory controller's applications using DPR

2.4.3 ASIC-Based IP Optimization

1. Keep *n*-devices near *n*-devices and *p*-devices near *p*-devices [1].
2. Keep *n*MOS near ground and *p*MOS near V_{dd}.
3. Layout of large transistor: large transistors can be viewed as number of parallel small transistors because as the gate width increases beyond certain limit, the efficiency of the transistors decreases as poly resistance increases.
4. Metal line bending: use 45° bending not 90° as the effective area of the current flow through 90° bending is reduced to 50 %.

5. Put guard rings around differential pairs, *n*-well, and *p*-well.
6. If we leave the differential pairs on the edges without dummies, they will see different surroundings and mechanical stress than the middle ones; with dummies we can avoid this.
7. Use interleaving between transistors so that if a fabrication error happened in a die, it does not affect the remaining transistors and the chip can remain working correctly.
8. Global signals should be routed on the top and bottom of layout blocks. Local signals should be routed through the center of layout blocks.

2.4.4 PCB-Based IP Optimization

1. Separate the digital and analog portions of the circuits (Fig. 2.41).
2. High frequency components should be placed near the connectors (Fig. 2.42).

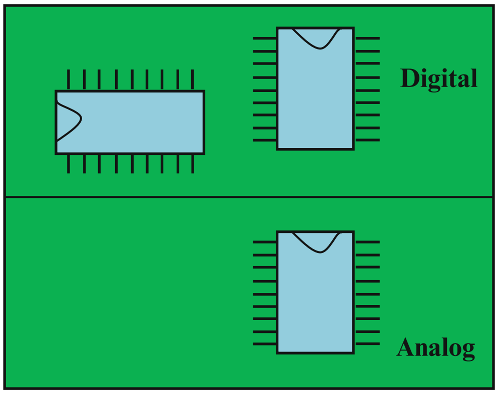


Fig. 2.41 Separate the digital and analog portions of the circuits

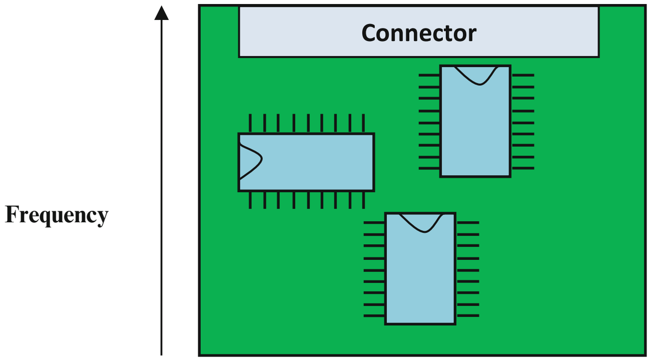


Fig. 2.42 High-frequency components should be placed near the connectors

2.5 IP Protection

Without IP protection, companies can lose revenue and market share.

2.5.1 FPGA-Based/Processor-Based IP Protection

IP vendors are facing major challenges to protect hardware IPs from IP-piracy as, unfortunately, recent trends in IP-piracy and reverse engineering efforts to produce counterfeit ICs have raised serious concerns in the IC design community. IP-piracy can take several forms, as illustrated by the following scenarios:

1. A chip design house buys an IP core from an IP vendor and makes an illegal copy or “clone” of the IP. The IC design house then sells it to another chip design house (after minor modifications) claiming the IP to be its own.
2. An untrusted fabrication house makes an illegal copy of the GDS-II database supplied by a chip design house and then illegally sells them as hard IP.
3. An untrusted foundry manufactures and sells counterfeit copies of the IC under a different brand name.
4. An adversary performs post-silicon reverse engineering on an IC to manufacture its illegal clone.

These scenarios demonstrate that all parties involved in the IC design flow are vulnerable to different forms of IP infringement which can result in loss of revenue and market share. Hence, there is a critical need of a piracy-proof design flow that equally benefits the IP vendor, the chip designer, as well as the system designer. A desirable characteristic of such a secure design flow is that it should be transparent to the end-user, i.e., it should not impose any constraint on the end-user with regard to its usage, cost, or performance.

To secure an IP, we need to obfuscate it then encrypt the contents before sending it to the customer. **Obfuscation** is a technique that transforms an application or a design into one that is functionally equivalent to the original but is significantly more difficult to reverse engineer. So, Obfuscation changes the name of all signals to numbers and characters combination. The second level is to encrypt the whole files [41, 42]. Although encryption is effective, code obfuscation is an effective enhancement that further deters code understanding for attackers [43]. Moreover, **Watermarking** can be used to protect Soft-IPs [44]. It includes modules duplication or module splitting.

2.5.2 ASIC-Based IP Protection

1. **Circuit camouflage**: let individual logic cells appear identical at each mask layer, when in fact subtle changes are present to differentiate logic functions. Changes are designed so that the reverse engineer is unable to automate cell recognition [45]. Figure 2.43 Shows an example of unprotected layout and Fig. 2.44 shows a protected one.

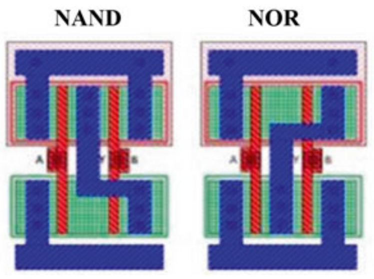


Fig. 2.43 Unprotected standard cell layouts the metal layers are different and hence it is easy to differentiate them by just looking at the top metal layer [45]

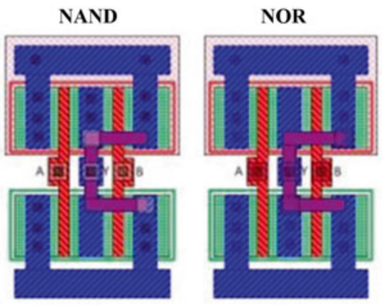


Fig. 2.44 Camouflaged standard cell layouts. The metal layers are identical and hence it is difficult to differentiate them by just looking at the top metal layer [45]

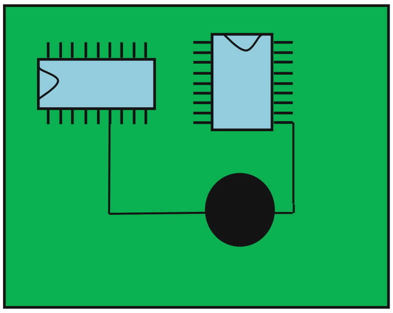


Fig. 2.45 Encapsulate the PCB into epoxy (*black blobs*)

2.5.3 *PCB-Based IP Protection*

1. Remove the markings from all the major ICs and mark them with in-house part numbers.
2. Encapsulate the PCB into epoxy (black blobs) as depicted in Fig. 2.45 [46].
3. Add a few fake layers for complexity.

2.6 Summary

This chapter discusses the IP cores life cycle process from specification to production which includes four major steps: (1) IP Modeling, (2) IP verification, (3) IP optimization, (4) IP protection. For IP modeling, four major methodologies are introduced which includes: FPGA-based modeling, processor-based modeling, ASIC-based modeling, and PCB-based modeling. For IP verification, different platforms are presented and analyzed such as simulation, acceleration, emulation, and prototyping. Moreover, different verification methodologies are introduced such as: UVM, direct testing, negative testing, software-driven testing, and formal testing. We presented different methods for IP optimization for the main design methodologies to improve area, speed, and power. For IP protection, we analyzed different strategies to perform protection not to make companies lose revenue and market share.

References

1. www.cs.clemson.edu/~mark/464/fab.pdf
2. Rafla NI, Davis, Brett LaVoy (2006) A study of finite state machine coding styles for implementation in FPGAs. 49th IEEE International Midwest Symposium on Circuits and Systems, San Juan
3. Roudier T, Moussa I, di Crescenzo P (2003) IP modelling and reuse for system level design. Published for DATE
4. http://www.esa.int/TEC/Microelectronics/SEM6Z0AMT7G_0.html
5. Simpson P, Jagtiani A (2007) How to achieve faster compile times in high-density FPGA. EE Times
6. Ricardo R, Marcelo L, Jochen J (2010) Design of systems on chip: design and test. Springer, Dordrecht
7. Clive M (ed) (2007) FPGAs: world class design. Newness, Burlington
8. Hauck S, DeHon A (2008) Reconfigurable computing: the theory and practice of FPGA-based computation. Morgan Kaufmann, Burlington
9. Maxfield CM (2004) The design warrior's guide to FPGAs. Newnes, Burlington
10. Betz V, Rose J, Marquardt A (1999) Architecture and CAD for deep-submicron FPGAs. Kluwer, Boston
11. Sutherland S, Davidmann S, Flake P (2003) Systemverilog for design: a guide to using systemverilog for hardware design and modeling. Kluwer, Norwell
12. Black D, Donovan J, Bunton B, Keist A (2010) SystemC: from the ground up, 2nd edn. Springer, New York. ISBN 978-0-387-69957-8
13. Goel P, Adhikari S (2014) Introduction to next generation verification language—Vlang. DVCON Conference and Exhibition, Munich
14. Schwartz RL, Phoenix T (2008) Learning PERL. O'Reilly Media, Sebastopol
15. www.ActiveState.com
16. Ucoluk G, Kalkan S (2007) Introduction to programming concepts with case studies in python. Springer, London
17. http://www.accellera.org/activities/committees/ip-xact
18. IEEE 1685-2009 IPXACT. Accessed 18 Feb 2010
19. Kulkarni R (2013) Automated RTL generator. M.Sc. Thesis, San Jose State University
20. Axelson J (1997) The microcontroller idea book. Lakeview Research, Madison
21. Sherwani NA (1999) Algorithms for VLSI physical design automation, 3rd edn. Kluwer, Boston
22. Sung Kyu L (2008) Practical problems in VLSI physical design automation. Springer, New York
23. Clein D (2000) CMOS IC layout concepts, methodologies, and tools. Butterworth–Heinemann, Newton
24. Coombs CF Jr (2001) Printed circuits handbook. McGraw-Hill, New York

25. Masahiro F, Indradeep G, Mukul P (2008) Verification techniques for system-level design. Morgan Kaufmann, San Francisco
26. Khan MA, Pittman RN, Forin A (2010) gNOSIS: a board-level debugging and verification tool. Proceedings of the IEEE Conference on ReConFigurable Computing and FPGAs (ReConFig), Microsoft Research, Redmond. pp 43–48
27. http://www.guru99.com/positive-vs-negative-testing.html
28. Pradhan DK, Harris IG (2009) Practical design verification. Cambridge University Press, Cambridge
29. Singh L, Drucker L, Khan N (2004) Advanced verification techniques: a SystemC based approach for successful. Kluwer, Boston
30. Scheffer L, Lavagno L, Martin G (2006) EDA for IC system design, verification, and testing. CRC, Boca Raton
31. https://zentronics.wordpress.com/tag/pcb-design-2/
32. Cassel M, Kastensmidt FL (2006) Evaluating one-hot encoding finite state machines for SEU reliability in SRAM-based FPGAs. Proceedings of 12th IEEE International On-Line Testing Symposium (IOLTS 2006), IEEE, Washington, p 6
33. http://www.arm.com/files/pdf/New_Whitepaper_Layout_Solving_Next_Generation_IP_Configurability.pdf. Accessed 2015
34. Bhuvaneswari K, Srinivasa Rao V (2013) Dynamic partial reconfiguration in low-cost FPGAs. Int J Sci Eng Res 4(9):1410–1413
35. Drahonovsky T, Rozkovec M, Novak O (2013) Relocation of reconfigurable modules on Xilinx FPGA. Proceedings of the 2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), IEEE, Karlovy Vary, pp 175–180
36. Partial reconfiguration user guide, Ug702 (v12.3) ed., Xilinx Corporation, October 2012
37. Partial reconfiguration of Xilinx FPGAs using ISE design suite, Xilinx Corporation, July 2012
38. Dunkley R (2012) Supporting a wide variety of communication protocols using partial dynamic reconfiguration. Proc IEEE Autotestcon 2012:120–125
39. Marques N, Rabah H, Dabellani E, Weber S (2011) Partially reconfigurable entropy encoder for multi standards video adaptation. 2011 IEEE 15th International Symposium on Consumer Electronics (ISCE), June 2011, pp 492–496
40. Wang Lie, Wu Fengyan (2009) Dynamic partial reconfiguration in FPGA. Third International Symposium on Intelligent Information Technology Application, IEEE Computer Society, Nanchang, pp 445–448
41. Chakraborty RS, Bhunia S (2008) Hardware protection and authentication through netlist level obfuscation. Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design, San Jose, 10–13 November 2008
42. Chakraborty RS, Bhunia S (2009) HARPOON: an obfuscation-based SoC design methodology for hardware protection. IEEE Trans Comput Aided Des Integr Circuits Syst 28(10):1493–1502
43. Kainth M, Krishnan L, Narayana C, Virupaksha SG, Tessier R (2015) Hardware-assisted code obfuscation for FPGA soft microprocessors. Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE, EDA Consortium, San Jose
44. Tehranipoor MM, Guin U, Forte D (2015) Counterfeit integrated circuits detection and avoidance. Springer, Cham
45. http://www.smi.tv/SMI_Circuit_Camo_Data_Sheet.pdf
46. http://www.eetimes.com/electronics-news/4212418/Standard-issued-for-PCB-IP-protection

Chapter 3
Analyzing the Trade-off Between Different Memory Cores and Controllers

3.1 Introduction

With the move to multicore computing, the demand for memory bandwidth grows with the number of cores. It is predicted that multicore computers will need 1 TBps of memory bandwidth. However, memory device scaling is facing increasing challenges due to the limited number of read and write cycles in flash memories and capacitor-scaling limitations for DRAM cells. Therefore, memory bottleneck is one of the main challenges in modern VLSI design. Microprocessors communicate with memory cores through memory controllers (Fig. 3.1). A detailed figure is shown in Fig. 3.2 [1–6].

Modern systems have complex memory hierarchies with diverse types of volatile and nonvolatile memories such as DRAM and flash. It is the task of the memory controller to manage these devices. To improve this communication as a solution for the memory bottleneck, the memory cores and memory controllers can be improved. The most famous existing memory cores–based solutions are to increase the amount of on-chip memory elements. However, this solution is expensive, and the most famous existing memory controllers–based solutions are to improve the controller architectures and scheduling algorithms.

Designing memory controllers is challenging in terms of performance, area, power consumption, and reliability. Since DRAM and NAND Flash scaling will be at risk as technology scales down to 20 nm, various technological innovations will be required to fulfill technological demands [7]. To address these challenges, different new memory cores architectures and protocols are analyzed in this chapter.

K.S. Mohamed, *IP Cores Design from Specifications to Production*,
Analog Circuits and Signal Processing, DOI 10.1007/978-3-319-22035-2_3

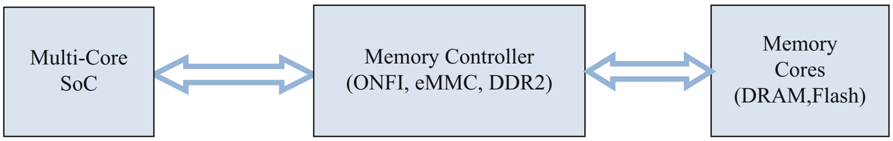
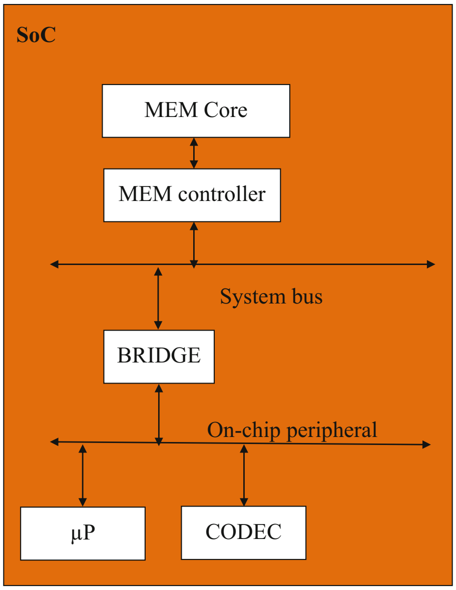

Multi-Core SoC
Memory Controller (ONFI, eMMC, DDR2)
Memory Cores (DRAM,Flash)

Fig. 3.1 Memory cores interfaces with microprocessors

Fig. 3.2 Memory cores interfaces with microprocessors through bridge



3.2 Memory Cores

Memory cores and most famous memory controllers are summarized in Fig. 3.3, where memories are classified into two main categories [8]:

1. **HDD**: Hard disk driver (HDD) utilizes ultrasophisticated magnetic recording and playback technologies. They are used as the primary data storage components in notebooks, desktops, servers, and dedicated storage systems.
2. **SSD**: Solid-state driver (SSD) is a data storage device that uses nonvolatile memory (ROM, EEPROM, and Flash) and volatile memory (SDRAM, DRAM) to store data.

Comparison between HDD and SDD are shown in Table 3.1, where SSD are showing better performance. HDD maximizes GB, not performance. In addition, the difference is shown in Fig. 3.4. Noting that, the flash-based memories are based on floating-gate technology as depicted in Fig. 3.5, how it works is shown in the following steps:

1. A large voltage difference between the drain and the source creates a large electric field between the drain and the source.

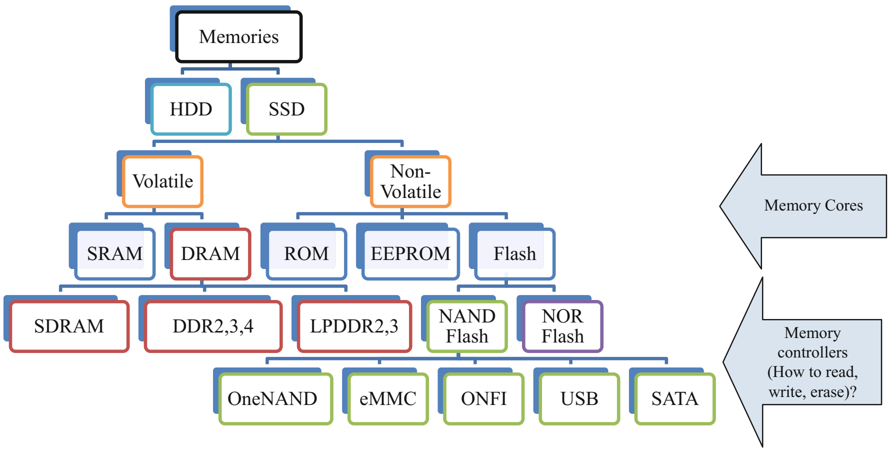


Fig. 3.3 Memory cores and memory interface, for example eMMC is NAND flash-based storage chip that features eMMC interface instead of the typical NAND flash or ONFI interface

Table 3.1 Comparison between SSD and HDD

	SSD	HDD
Capacity		✓
Performance	✓	
Reliability	✓	
Endurance	✓	
Power	✓	
Size	✓	
Weight	✓	
Shock	✓	
Temperature	✓	
Cost per bit		✓
Moving parts		✓

2. The electric field converts the previously nonconductive poly-Si material to a conductive channel, which allows electrons to flow between the source to the drain.
3. The electric field caused by a large gate voltage is used to bump electrons up from the channel onto the floating gate.
4. The number of electrons on the floating gate affects the threshold voltage of the cell (Vt). This effect is measured to determine the state of the cell.
5. The threshold voltage can be manipulated by the amount of charge put on the floating gate of the Flash cell.
6. Placing charge on the floating gate will increase the threshold voltage of the cell. When the threshold voltage is high enough, around 4.0 V, the cell will be read as programmed. No charge, or threshold voltage <4.0 V, will cause the cell to be sensed as erased.

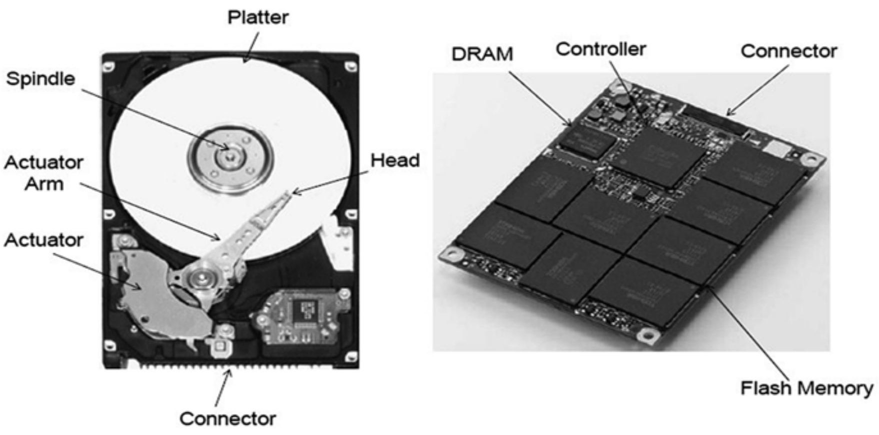


Fig. 3.4 Hard disk drive versus solid-state drive

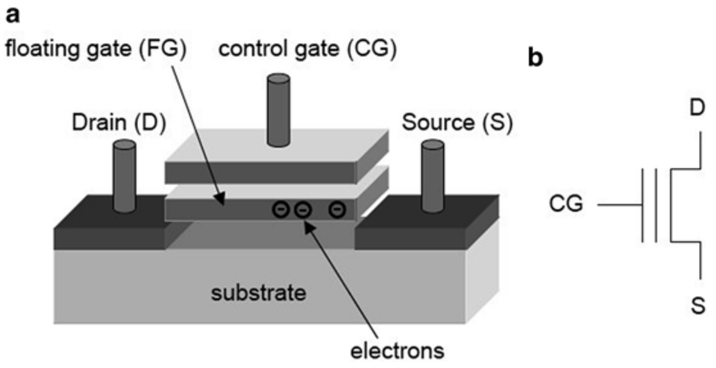


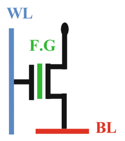
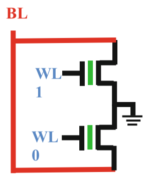
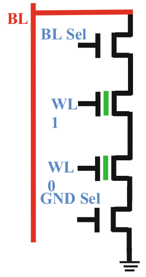
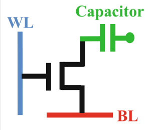
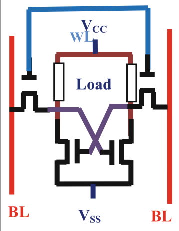
Fig. 3.5 (**a**) Floating-gate memory cell and (**b**) its schematic symbol

A comparison between different memories cores is shown in Table 3.2 [9]. The flash cell can be classified into (Fig. 3.6) [10]:

1. Multi-level cell NAND (**MLC**): stores four states per memory cell and enables two bits programmed/read per memory cell.
2. Single-level cell NAND (**SLC**): stores two states per memory cell and enables one bit programmed/read per memory cell.

A computer system contains a hierarchy of storage devices with different costs, capacities, and access times. With a memory hierarchy, a faster storage device at one level of the hierarchy acts as a staging area for a slower storage device at the next lower level. Software that is well written takes advantage of the hierarchy accessing the faster storage device at a particular level more frequently than the storage at the next level. Understanding the memory hierarchy will result in

Table 3.2 Comparison between different memory cores

	EEPROM	NOR flash	NAND flash	DRAM	SRAM
Cell structure					
Density average	128 Kb	128 Mb	2–16 Gb	1 Gb	128 Mb
Cost	Worst	Moderate	Cheap	Cheap	High
Nonvolatile	Yes	Yes	Yes	No	No
Software	Easy	Complex	Complex	Easy	Easy
Write bandwidth	Up to 30 KB/s	2 MB/s	10 MB/s	100 MB/s	500 MB/s
Read latency (ns)	200	100	50	80	30
Read performance	Asynchronous	166 MHz	40 MHz	400 MHz	600 MHz
Write performance	Asynchronous	Low	High	Medium	High
Endurance	105	105	104	Unlimited	Unlimited
Speed	Slow (1×)	Slow (100×)	Slow (500×)	Slow (5000×)	Fast (25,000×)
Sense amplifier	Not mandatory	Not mandatory	Not mandatory	Mandatory	Not mandatory
Refresh	Not required	Not required	Not required	Periodical	Not required
Manufacturer		Toshiba	Toshiba		
Cell structure	1 Floating-gate transistor	*n*-Transistors in parallel	*n*-Transistors in series	1 Transistor and 1 capacitor	6 Transistors

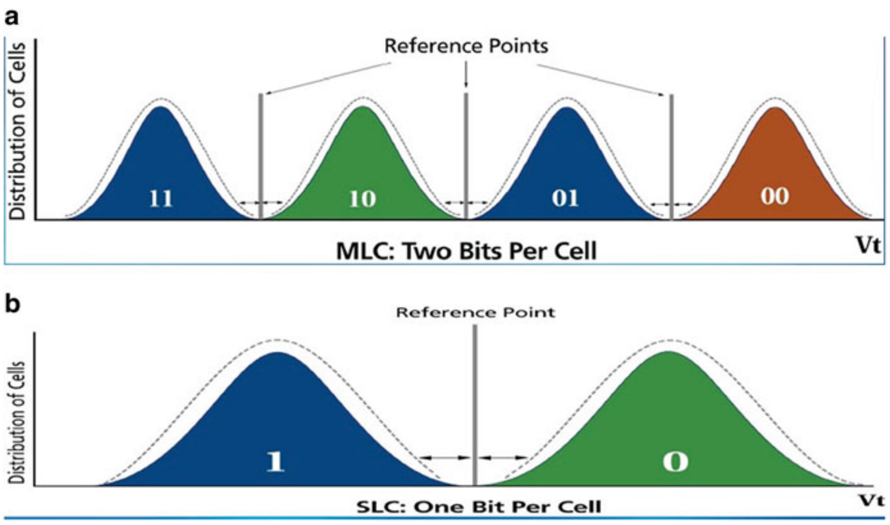


Fig. 3.6 (**a**) MLC, (**b**) SLC

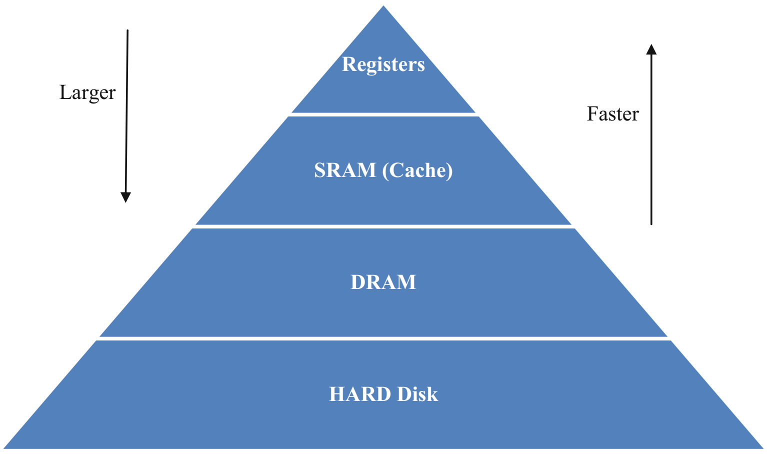


Fig. 3.7 An example of memory hierarchy

better performance of applications. The memory hierarchy can be summarized in Fig. 3.7. It starts with register file, SRAM, DRAM, then main memory or hard disk. Moreover, the comparison is shown in Table 3.3.

Table 3.3 Memory technology comparison

	Access delay	Cell area (μm^2)	Cells/mm^2 (Mb)
Register	<1 Cycle	0.7	1.5
SRAM	1 Cycle	0.4	2.5
DRAM	20–50 Cycle	0.04	15
Flash	Read: 50 cycles	0.02	50
	Write: 500 cycles		
Hard disk	5×10^6 Cycles	0.004	250

3.3 Why Standards?

SoC components (IPs) have an interface to the outside world consisting of a set of pins; it is responsible for sending/receiving addresses, data, and control. Number and functionality of pins must adhere to a specific interface standard. Standardization is important for seamless ***integration*** of SoC IPs—helps avoid integration mismatches [11]:

– E.g., 1—connecting IP with 32 data pins to a 16 bit data bus.
– E.g., 2—connecting IP supporting data bursts to a bus with no burst support.

It is also important because mismatches require development of "logic wrappers" at IP interfaces.

– To ensure correct data transfers.
– Time consuming to create, reduce performance, take up area.

Interface standards define a specific data transfer protocol to decide number and functionality of pins at IP interfaces and make it easy to connect diverse IPs quickly.

There are two categories of standards for SoC communication:

- **Standard bus architectures**
  – Define interface between IPs and bus architecture.
  – Define at least some specifics of bus architecture that implements data transfer protocol.
- **Socket-based bus interface standards**
  – Define interface between IPs and bus architecture.
  – Freedom w.r.t choice and implementation of bus architecture.

Ideally, designers want one standard to interconnect all IPs. In reality, several competing standards have emerged.

JEDEC: is an organization works as a Leading developer of standards for the solid-state industry [12].

3.4 Memory Controllers

There is a great variety of interfaces and protocols, which provide access to the internal memory cores in different ways to read, write, or erase. Referring to Fig. 3.3, examples of Flash-based Memory controllers are EMMC, OneNAND, and ONFI. Examples of DRAM-based memory controllers are DDRx, LPDDx.

The main aim of the memory controller is to provide the most suitable interface and protocol between the host and the memories and to efficiently handle data, maximizing transfer speed, data integrity and information retention (conservation of data with time). The main features are summarized in Table 3.4. If we compare the architecture of these different controllers, we realize that their architecture is common in many things. They mainly differ in the performance and the features. The following section will describe the most common memory controllers.

1. **eMMC**

   The eMMC is a managed memory capable of storing code and data. It is specifically designed for mobile devices. The eMMC is intended to offer the performance and features required by mobile devices while maintaining low power consumption. The eMMC device contains features that support high throughput for large data transfers and performance for small random data more commonly found in code usage. It also contains many security features. eMMC communication is based on an advanced 10-signal bus. An example of eMMC architecture is shown in Fig. 3.8 [13].
2. **OneNAND**

   Samsung's OneNAND meets the memory-hungry needs of next-generation devices by providing a single-chip flash that offers the ultrahigh density of NAND with the simplified interface neither of NOR at very attractive price points. OneNAND can achieve up to 108 MB/s read performance to optimize application

Table 3.4 Memory controller features

Features	Explanation
Topology	Point to point, or multi-master/multi-slave
Physical interface (#pins)	The physical interface with other circuits
Memory organization	The min unit for erase, write protection, read, write
Memory partitions	Single partition or multiple
Initialization process	How to start the memory controller operation? Negotiate different speeds, voltages and single/dual data rates, booting or/not
Command sets	To read, write, multiple read, multiple write, erase, write protection, partition, secure
Responses	How the card response to the host commands
Internal registers	Contains the initializations and the memory features
Data rate	The data can be DDR or SDR
Timing	The time between commands, responses, and data
Performance	Max clock
Reliability	ECC or not

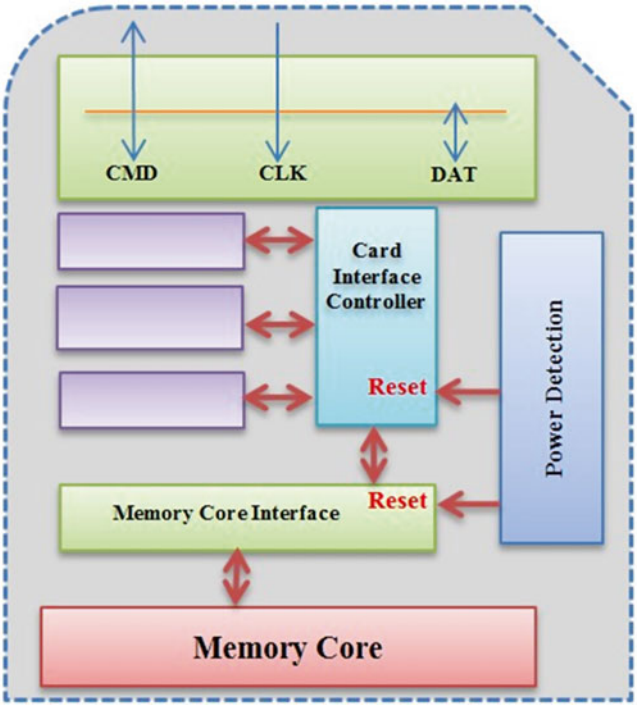


Fig. 3.8 eMMC architecture

functionality. It is available in densities from 256 Mb to 8 GB. With OneNAND, designers can use their existing chipset's NOR interface to communicate directly with the NAND flash memory, obviating the need for a separate NAND device. In addition, OneNAND's fast write-speed increases performance, which is extremely difficult to attain with NOR flash alone. OneNAND's compact size and range of features make it the ideal choice for: Handset, digital cameras, embedded solutions. An example of OneNAND architecture is shown in Fig. 3.9 [14].

3. **DDR3**

   The third generation of Dual Data Rate (DDR) Synchronous DRAM memory delivers significant performance and capacity improvements over older DDR2 memory. HP introduced DDR3 memory with the G6 and G7 ProLiant servers, coinciding with the transition to server architectures that use distributed memory and on-processor memory controllers. DDR3 continues to evolve in terms of speed and memory channel capacity, and the new HP ProLiant Gen8 servers fully support these improvements. An example of DDR3 architecture is shown in Fig. 3.10 [15].

4. **HMC**

   HMC uses 3D single packaging of 4 or 8 DRAM memory dies and one logic die collected together using through-silicon vias (TSV) and microbumps with smaller physical footprints. HMC exponentially is more power efficiency and

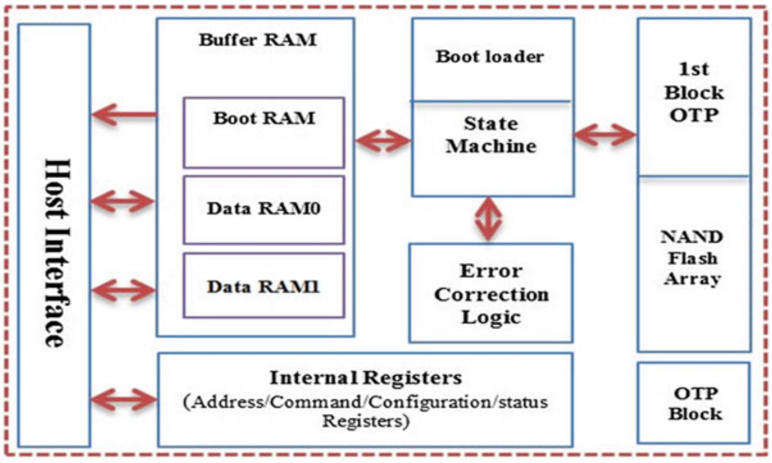


Fig. 3.9 OneNAND architecture

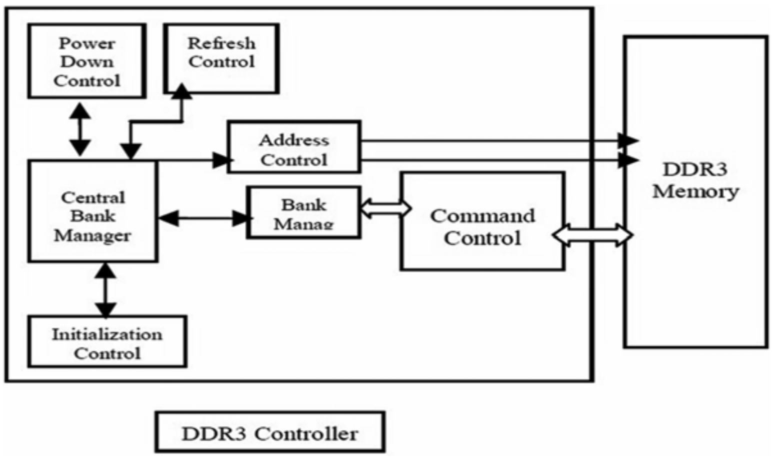


Fig. 3.10 DDR3 architecture

energy savings, utilizing 70 % less energy per bit than DDR3 DRAM technology. A single HMC can provide more than 15× the performance of DDR3 module, which increases bandwidth. HMC reduced latency with lower queue delays and higher bank availability. It can keep up with the advancements of CPUs and GPUs. HMC uses standard DRAM cells but its interface is incompatible with current DDR2 or DDR3 implementations. It has more data banks than classic DRAM of the same size. HMC memory controller is integrated into memory package as a separate logic die. The logic base manages multiple functions for HMC, like all HMC I/O, mode and configuration registers and data routing and buffering between I/O links and vault. A crossbar switch is an implementation example to connect the vaults with I/O links. The external I/O links consist

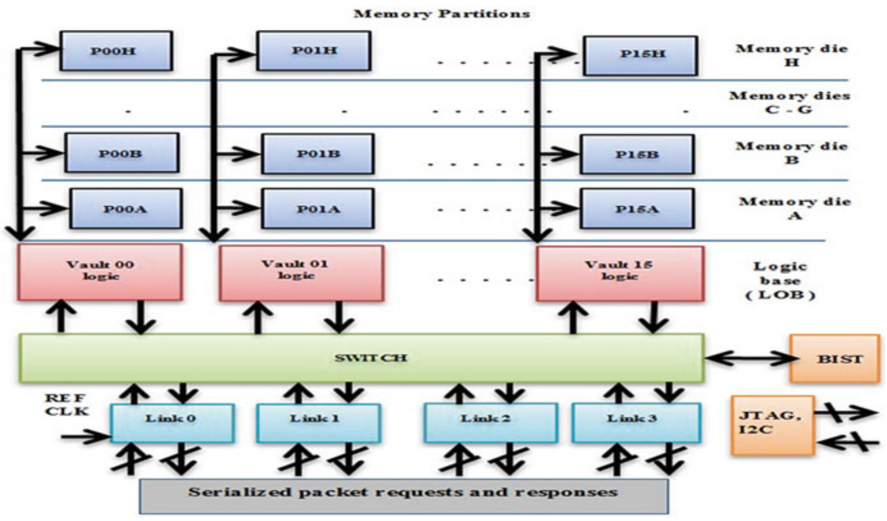


Fig. 3.11 HMC architecture

of multiple serialized 4 or 8 links, each link with a default of 16 input lanes and 16 output lanes for full width configuration, or 8 input lanes and 8 output lanes for half width configuration as shown in Fig. 3.11 [16].

5. **WideIO**

   WideIO mobile DRAM uses chip-level dimensional (3D) stacking with through-silicon vias (TSV) interconnects and memory chips directly stacked upon a system on a chip (SOC). WideIO DRAM major advantage over its predecessors (such as LPDDR DRAM) is that, it offers more bandwidth at lower power. WideIO is the first interface standard for 3D die stacks and offering a compelling bandwidth and power benefit. WideIO is particularly suited for applications requiring increased memory bandwidth UP to 17 GBps Such as 3D Gaming, HD video etc. WideIO will provide the ultimate in performance, energy efficiency and small size for smart phones, tablets, handheld gaming consoles, and other high-performance mobile devices. Given the ever-growing hunger for memory bandwidth and the need to reduce memory power in many applications; WideIO is the first standard for stackable WideIO DRAMs. This standard widens the conventional 32 bit DRAM interface to 512 bits. Memory diagram for WideIO is shown in Fig. 3.12 [17].

6. **ONFI**

   ONFI stands for Open NAND Flash Interface. Early NAND Flash devices from different manufacturers use similar interface but an open standard did not exist. As a result, subtle differences exist among devices from different vendors. ONFI standard aims to provide a common standard, so different device can be used interchangeably and sets the stage for future standard NAND Flash development as shown in Fig. 3.13. The lack of a standard caused serious design problems like host systems had to accommodate differences between vendors'

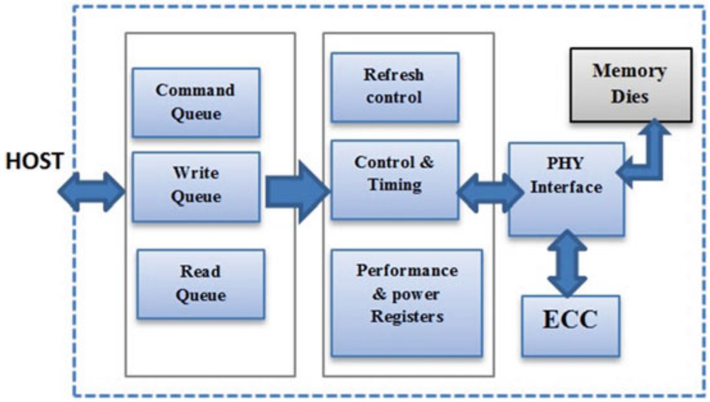


Fig. 3.12 WideIO architecture

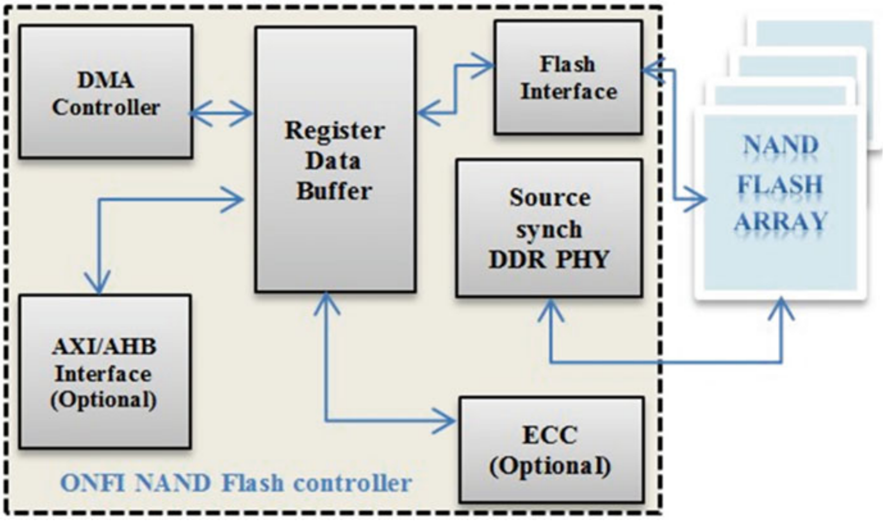


Fig. 3.13 ONFI architecture

devices and adapt to generational changes in parts from a single vendor. All of this made incorporating new or updated NAND Flash components extremely costly, often requiring extensive hardware, firmware, and/or software changes and additional testing which slowed time to market. ONFI works to solve all these issues by standardizing the NAND Flash interface-reducing vendor and generational incompatibilities and accelerating the adoption of new NAND products [18].

7. **UFS**

   UFS is most advanced specification for embedded and removable flash memory-based storage because it includes the feature set of eMMC specification as a subset. It also references several other standard specifications by MIPI

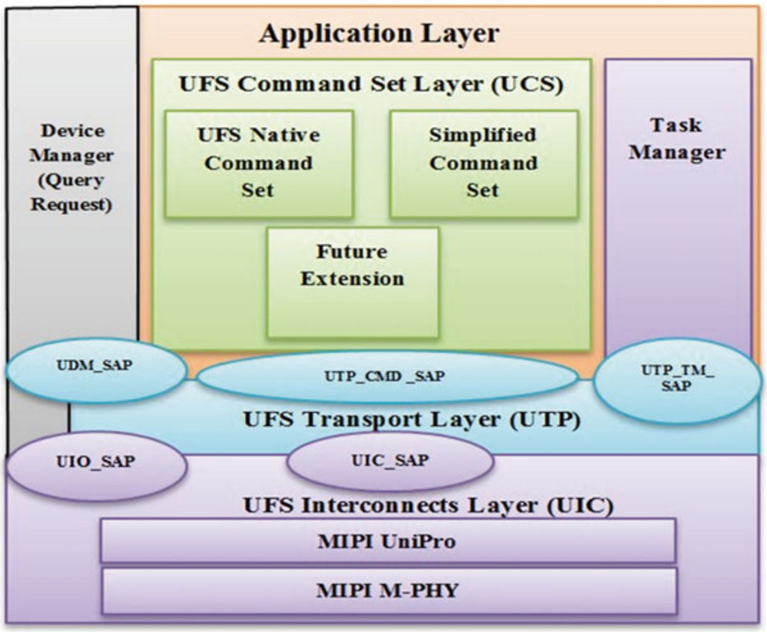


Fig. 3.14 UFS architecture

(M-PHY and UniPro specifications) and INCITS T10 (SBC, SPC, and SAM specifications) organizations. The UFS interface is a universal serial communication bus, based on MIPI M-PHY standard as physical layer for optimized performance and power. UFS references the INCITS T10 SAM model for ease of adoption. The UFS Top level Architecture Consists of three main layers as shown in Fig. 3.14. First layer is called application layer which consists of UFS command set layer (UCS) which handles normal commands, device manager which has two jobs which are device level operations such as sleep, and power-down management, and device-level configurations such as set of descriptors and handling query request. Task manager handles command queue control. UCS establishes the method of data exchange between host and device and also provides device management capability. Second layer is UFS transport protocol layer (UTP) which services the higher layers and its mission is to encapsulate the protocol into appropriate frame structure for the lower layer. Third layer is UFS interconnect layer (UIC) [19].

8. **HBM**

HBM (High-Bandwidth Memory) is a new type of DRAM-based memory chip with low power consumption, ultrawide communication lanes and a revolutionary new stacked configuration. HBM uses 128-bit wide channels. It can stack up to eight of them for a 1024-bit interface. The total bandwidth ranges from 128 to 256 GB/s. Each memory controller is independently timed and controlled. Future GPUs built with HBM might reach 1 TB/s of main memory bandwidth. HBM designed for high-performance GPU environments as it is cheaper than HMC [20].

3.5 Comparison Between Different Memory Controllers

There are completely different memory organizations which develop different protocols to enable the designer to pick up the most efficient and suitable one for his application.

For Flex-OneNAND, the building block unit is 4 KB page, which has main area and spare area. The 4 KB page is divided into eight sectors each of which is 512 bytes for main and 16 bytes for spare. ONFI has eight targets, each target has arbitrary multiple Logic units (LUNs). Each LUN consists of arbitrary number of blocks. Each block consists of number of pages. Each page consists of optional partial pages which are the smallest unit to program or read. LUN is minimum unit to execute command and report status. Block is the smallest erasable unit. eMMC is divided into write protect groups, each one consists of erase groups, and each erase group has write blocks with 512 bits for each. HMC is organized into vaults; each vault has 4 or 8 partitions according to the number of memory dies. One partition is multiple of 16 MB banks. Each four vaults called quadrant. WideIO consists of four memory dies which are called stack. Each die consists of four independent channels of 128 bidirectional data bits. Each channel has four Banks, each bank is 512 MB. The interface consists of 300 (microbump) pads per channel. UFS is consists of eight configurable Logic Units (LU) and four well-known logical units. LU is an externally independent addressable entity processes the commands and performs task management functions. Each LU can be configured as boot LU with maximum of two. The well-known logic units are: Boot which is virtual reference to the actual LU containing boot code, REPORT LUNs which provides the LU inventory, UFS device which provides UFS device level interaction (i.e., power management control), and RPMB supports RPMB function with its own independent processes and memory space.

HMC and WideIO are 3D protocols. The 3D design provides 15 % performance improvements due to eliminated pipeline stages and 15 % power saving due to eliminated repeaters and reduced wiring compared to 2D. The stacked security structure complicates attempts to reverse the circuitry.

The protocols support two main types of memory cells which are flash and DRAM. Flash memory cells have no power for storing data and hold a lot more data than DRAM but it is slower than DRAM. For flash type, SLC and MLC are both NAND-based nonvolatile memory technologies. MLC offers a larger capacity twice the density of SLC, but SLC provides an enhanced level of performance in the form of faster write speeds. The most powerful feature in Flex-OneNAND and ONFI is the combination between SLC and MLC.

Partitioning the memory array is playing a major role in specifying the functionality of each part of memory. Flex-OneNAND supports three memory partitions which are one-time programmable partition (OTP), first block OTP, and boot partition. eMMC is divided into two boot area partitions which are used to access and modifying boot data, one RPMB partition to store data in an authenticated and replay protected manner through HMAC-SHA algorithm which supports protection that requires passwords and keys for access, four general purpose partitions to store sensitive data or for other host usage models and enhanced user data area. Boot and RPMB partitions are read only programming, but general purpose area and enhanced user data area partitions are one-time programmable. In UFS, each LU can be differentiated over the others with many types during the system integration. The memory types are default type for regular memory characteristics, system code type for a logical unit that is rarely updated (e.g., system files or binary code executable files, …, etc.), Nonpersistent type is used for temporary information and enhanced memory type is left open in order to accomplish different needs and vendor-specific implementations.

Flex-OneNAND supports only three simple modes. Limited-based command mode which is used for booting operation. Register-based mode which is used for command execution. Idle mode is used when the device is waiting for host request. ONFI simply supports only two modes, active mode which is used for commands and operations execution and the other is idle mode which immediately entered after power on. eMMC cycle life time is divided into modes. First, eMMC optionally passes through boot mode, then passes through identification mode to validate operation voltage range and access mode, identifies the device and assigns a relative device address (RCA) on the bus and finally passes through data transfer mode executing any commands forwarded from the CPU. eMMC supports optional interrupt mode by specific command. Interrupt mode reduces the polling load for CPU hence the power consumption. HMC life cycle consists of multiple modes as initialization mode to prepare HMC for any request or data transfer, active mode where the HMC device is preparing to execute any request and transfer any data, sleep mode where it sets each link into lower power state by inverting its power state management pin from high to low. Then HMC enters down mode which is lower power state than sleep mode by disabling both serializer and deserializer circuitry and the link's PLLs. WideIO has five modes. First mode is idle mode in which the banks have been precharged. Precharge is to deactivate an open row in one or all banks. Banks cannot be used again after certain time. After precharging a bank in idle state requires an active command before any read or write commands forwarded to the bank. Second, active mode is to activate row of a given bank to read or write data. Power-down mode is supported for each channel circuit except for clock (CK) and clock enable (CKE), where they are gated off to reduce power consumption. The device enters power-down mode when CKE is low and exits when CKE is high. In deep power-down, all channels on that slice will exit deep power-down mode. The reset signal is used because reset signal is per memory die not per channel. UFS Device supports seven power modes which are controlled by the START STOP UNIT command and some attributes.

In order to minimize power consumption in a variety of operating environments, UFS supports four basic power modes which are Active, Sleep, idle, and power-Down. Also, it supports three transitional modes to facilitate the change from one mode to the next. UFS can support up to 16 active configurations. Each one has its own current profile. The host can choose from either predefined or user defined currents profiles to deliver the highest performance.

In Flex-OneNAND, after boot code is loaded, Boot buffer is always locked. For NAND Flash array protection, device has hardware and software write protection. Hardware write protection is implemented by executing a "Cold" or "Warm" reset.

Software write protection is implemented by specific commands. The write protect signal in ONFI disables Flash array program and erase operations. To allow eMMC to protect data against erase and write; the eMMC supports three levels of write protection commands such as permanent or temporary or power-on protection applying for the entire device or for specific segments. In WideIO, Input data mask (DM) is the input mask signal for write data. Input data is masked when DM is sampled high.

Flex-OneNAND supports 31 registers which are utilized by the device mainly for configuration of the device and status of the operations done by the device. In ONFI, parameter pages are used to describe NAND capabilities. Parameter page solves inconsistencies among devices by describing revision info, features, and organization timing. eMMC has six different registers with different sizes. These registers include configuration bytes and status bytes. The UFS software uses 37 registers that exist in the host side to control the device through HCI interface. HMC has 15 registers that consist of configuration registers and status registers with the same size of 32 bits.

Commands of these protocols indicate the major features. So in ONFI, the majority of commands are optional because all NAND Flash devices are not created equal, differences include architectural, performance, and command set, so ONFI helps to address many of these through optional commands and optional parameter pages. In eMMC, there are major 43 usable commands including read commands, write commands, erase commands, sleep command, and interrupt command. HMC uses 23 different commands concentrating on read and write commands only. The command or request is sent in shape of packet (multiple of 128 bits) associated with the data; the same as the response. Commands and responses are serialized and transmitted across the lanes of links. Every command and response contains header and tail which indicates important fields for example: address, command number, and CRC.

To know the echo of commands, there must be a response or status register to be checked. In Flex-OneNAND, response is checked from status registers after execution of command. ONFI Reads status and retrieves the status value for the last operation issued. In eMMC there are five responses vary from command to another by their included fields. eMMC includes some status bits like error switch bit. HMC has also a response packets and status register for CPU to check the situation of HMC. For WideIO, status register read (SRR) can only be issued after power up and initialization sequence are completed. SRR provides a method to read registers from WideIO DRAM. But, in UFS, UTP delivers commands, data and responses as standard packets over the UniPro network. The UFS transactions will be grouped into data structures called UFS protocol information unit (UPIU). There are UPIUs defined for commands, responses, and data in and data out. A response UPIU contains a command-specific operation status and other response information. This represents the status phase of the command.

The main comparison between the six memory controller architectures, which is based on the most important features that microelectronics designers are interested in, is summarized in Tables 3.5 and 3.6.

Table 3.5 Comparison between different memory controllers

	Flex-OneNAND	ONFI 3.0	eMMC v. 4.51	HMC	WideIO	UFS
Memory organization	1. Sector (528 B)	1. Partial pages	1. Write block	1. 8/16 Banks per vault	1. Banks	1. 8 Independent configurable Logic Units (LU)
	2. 4 KB page (8-sectors)	2. Pages	2. Erase group	2. Partition	2. Channels: 4 banks	2. 4 Well-known LUs (Boot, Device, RPMB, Report LUNs) (Fig. 3.20)
	3. 1024 Block (64 page SLC, 128 page MLC) (Fig. 3.15)	3. Blocks	3. Write protect group (Fig. 3.17)	3. Vault : 4 Partitions	3. Memory die: 4 channels	
		4. Logic units		4. Quadrant: 4 vaults (Fig. 3.18)	4. Stack: 4 memory dies (Fig. 3.19)	
		5. Targets (Fig. 3.16)				
Technology	2D	2D	2D	3D	3D	2D
Memory cells	Flash (convertible SLC and MLC)	Flash (SLC or MLC or both)	Flash (SLC)	DRAM	DRAM	Flash (SLC)
Memory partitions	1. OTP block	Only 1 partition	1. 2 Boot area partitions (×.128 KB)	Only 1 partition	Only 1 partition	1. Multiple user data partition
	2. 1st Block OTP		2. 1 RPMB partition (128 KB)			2. Boot partitions
	3. Partition information blocks (PI)		3. 4 General purpose partitions and enhanced user data areas (×.WPs)			3. RPMB partition
Modes of operations	1. Limited-command-based (for boot only)	1. Idle mode	1. Boot mode	1. Initialization	1. Idle	1. Active
	2. Register-based (Active)	2. Active mode	2. Identification Mode	2. Active	2. Active	2. Idle
	3. Idle		3. Interrupt mode	3. Sleep	3. Power down	3. Sleep
			4. Data transfer mode	4. Power down	4. Deep power down	4. Power down
						5. Pre active
						6. Pre sleep
						7. Pre power down

(continued)

Table 3.5 (continued)

	Flex-OneNAND	ONFI 3.0	eMMC v. 4.51	HMC	WideIO	UFS
Data protection	1. Write protection 2. Data protection during power down	Write protect pin	1. Permanent WP 2. Temporarily WP 3. Power-on WP	No Protection	Write data mask pin	1. Permanent WP 2. Power-on WP
Encryption	–	–	HMAC (RPMB)	Scrambler & de-scrambler	–	HMAC (RPMB)
Number of registers	31	1	6	15	8	37
Size of registers	16 Bits	768 Bytes parameter page definitions	Differ from register to another	32 Bits	19 Bits	32 Bits
Number of pins	39	48	13	29	48	14
Transmission type	Synch./Asynch.	Synch./Asynch.	Synchronous only	Synchronous only	Synchronous only	Synch./Asynch.
Number of commands	16	32 (9 Mandatory)	64 (21 Reserved)	23	32	27
Command length (bits)	16	8 or 16	48	×.128	4	128
Responses	Status registers checked	Status registers checked	5 Responses differ from command to another	Response (×.128 bits) & status registers	Status register	UPIU response (23 bytes)
Command/data bus	Command and data are sent on the same bus	Command and data are sent on the same bus Device may support 2 independent data buses	Command and data are sent on different buses	Command and data are sent on the same Links	Command and data are sent on different buses	Command sent on Upstream link Data sent on either up or Down stream link

Interface	1. CLK	1. CLK/ (write enable)	1. CLK	1. CLK	1. CLK	1. CLK
	2. CMD & data line	2. CMD enable	2. Reset	2. Reset	2. Command bus	2. Reset
	3. Interrupt	3. Address enable	3. 1-Bit CMD line (bidirectional)	3. 8/16 lanes data (I/O)	3. Address bus	3. Downstream/upstream lane input/output
	4. RDY	4. Data/CMD line	4. 8-Bits data lines (bidirectional)	4. JTAG	4. Data bus	4. Differential input/output true and complement signal pair
	5. Write enable	5. Data strobe		5. I^2C	5. Data mask	
	6. Address valid data	6. Ready/busy			6. Reset	
	7. Reset	7. Read enable/ (WR/RD)				
		8. Reset				
Interface type	Parallel	Parallel	Parallel	Serial	Parallel	Serial
Booting	Mandatory	No booting	Optional	No booting	No booting	Optional
Clock (MHz)	66/83	Up to 200	200	125/156.25/166.67	200	19.2/26/38.4/52
Speed	66/83 MB/s	400 MB/s	200 MB/s	10/12.5/15 Gb/s	200 MB/s	300 MB/s
Reliability	ECC	ECC	CRC	CRC / ECC	ECC	CRC
Data rate	SDR	SDR/DDR	SDR/DDR	DDR	SDR	DDR
Timing	One timing mode	5 Timing modes	One timing mode	One timing mode	One timing mode	One timing mode
Topology	Point to point	Point to point	Point to point	Point to multi	Point to point	Point to point
Bandwidth (Gb/s)	Not mention	Not mention	Not mention	160/200/240/320	12.8	3 Per lane
Power saving management	–	–	Sleep mode only	1. Sleep mode	1. Power-down mode	1. Sleep mode
				2. Down mode	2. Deep power-down mode	2. Power-down mode

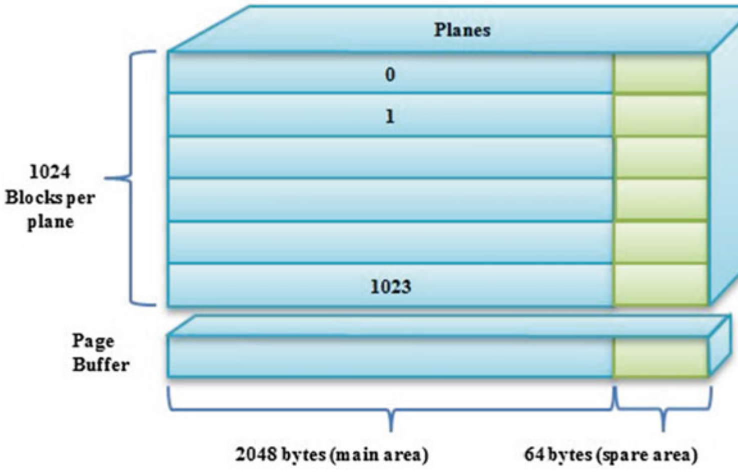


Fig. 3.15 Flex-OneNAND memory organization

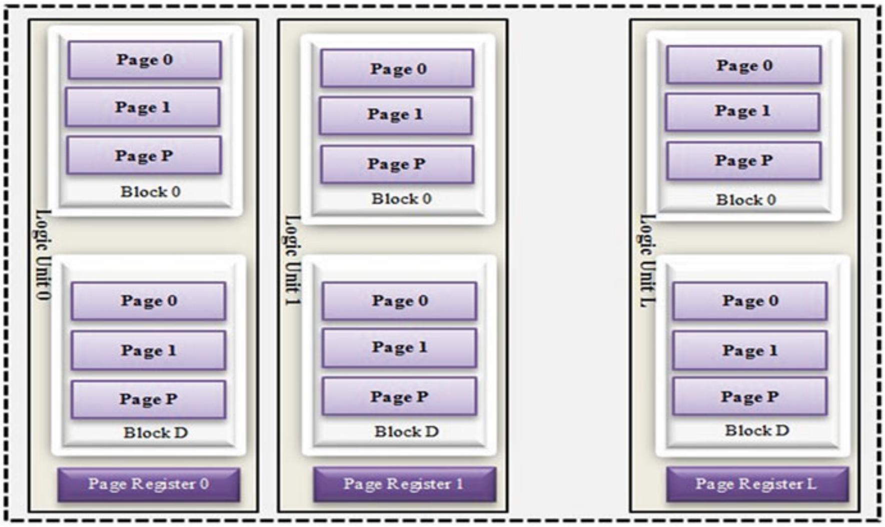


Fig. 3.16 ONFI memory organization

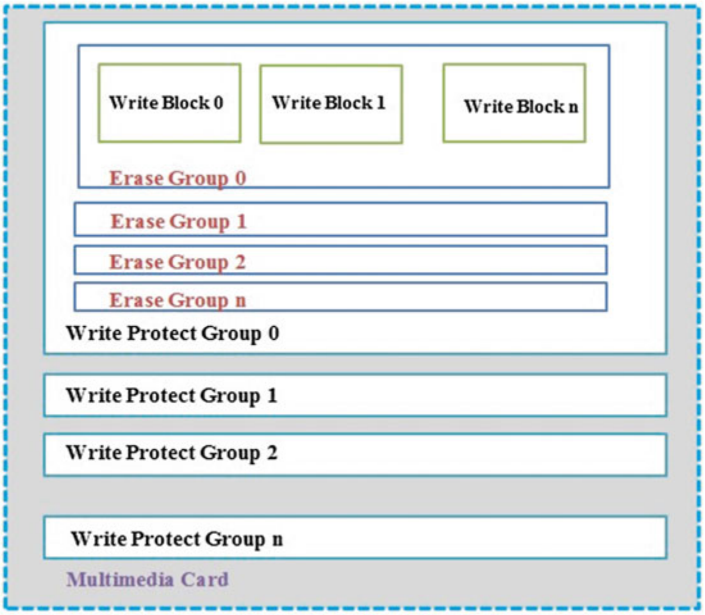


Fig. 3.17 eMMC Memory Organization

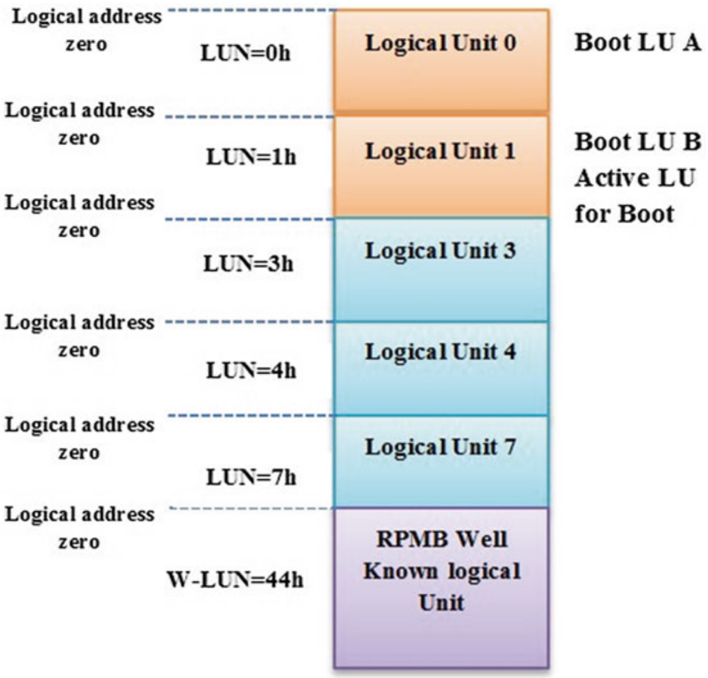


Fig. 3.18 UFS memory organization

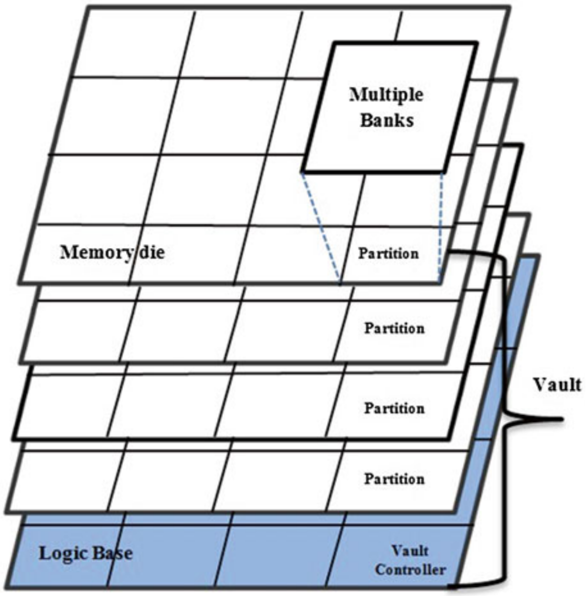


Fig. 3.19 HMC memory organization

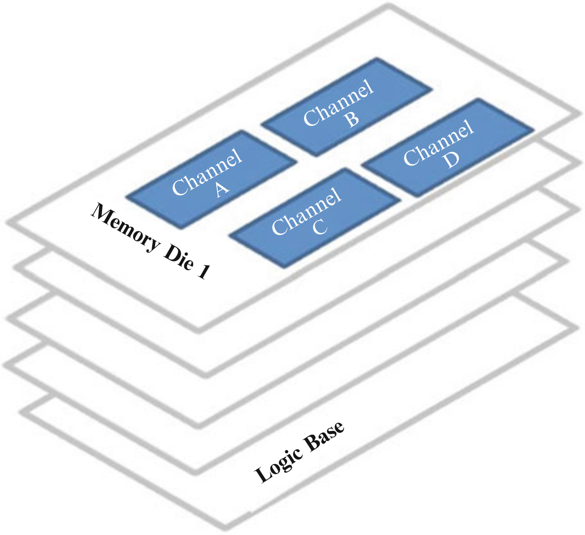


Fig. 3.20 WideIO memory organization

3.6 New Trends in SoC Memories

SRAM/DRAM is fast but has large leakage of power and is volatile. Floating-gate-based Flash is nonvolatile but exhibits low write speed and limited write endurance. Therefore, recent research focuses on hybrid memory structures to get the advantages of both. From the prospective of system level, 3D integration can be employed to integrate hybrid memory components with high density, where it can also reduce the distance between components to few micrometers instead of few centimeters. Emerging memory technologies are making steady progress towards product introductions, including phase-change memory (PCRAM), resistive memory (ReRAM), and magnetic memory (MRAM). The new trends in memories are summarized in Table 3.7. They provide higher density, lower latency, lower power per bit for both read and write operation, and high read/write/erase processing speed [21].

Memristor is built from titanium dioxide (TiO_2) and platinum (Pt) as depicted in Fig. 3.22. When the charge flows in one direction through a circuit, the resistance of the memristor increases. The resistance decreases when the charge flows in the opposite direction in the circuit. If the applied voltage is turned off, thus stopping the flow of charge, the memristor remembers the last resistance that it had. When the flow of charge is started again, the resistance of the circuit will be what it was when it was last active. Its main advantage is that program power is low and its main disadvantage is that platinum is expensive [22].

FeRAM replaces dielectric by ferroelectric material. Its performance is close to DRAMs and it does not need refreshing process [23].

Memory hierarchy requires new architecture and technology due to increasing demand of bandwidth and low power consumption. 3D Memory is an emerging memory technology, compared to existing memory interface (Fig. 3.23), TSV-based 3D technology provides better bandwidth and less power consumption. Lower power consumption is achieved by lower capacitance of TSV [24].

3.7 Summary

In this chapter, we present most famous memory cores and controllers and analyze the trade-off between them. The importance of standards is discussed. A descriptive comparison between various on-chip memory protocols is made. Comparing the architecture of these different controllers, it is realized that their architecture is common in many things. They mainly differ in the performance and the features. Moreover, we introduce new trends in SoC memories such as PCRAM, ReRAM, MRAM, and 3D memory.

Table 3.6 Comparison between the most common architecture and the most famous memory controller protocols

Features	Flex-OneNAND	ONFI	eMMC	HMC	WideIO	UFS
Read	✓	✓	✓	✓	✓	✓
Write	✓	✓	✓	✓	✓	✓
Write protection	✓	✓	✓		✓	✓
Erase	✓	✓	✓	✓	✓	✓
Background operations			✓			✓
High-priority interrupt			✓			✓
Context management			✓	✓		✓
Data tag mechanism			✓			✓
Power off notification			✓			✓
Hibernate						
Lock/unlock			✓			
Encryption			✓	✓		✓
Packed operations			✓			
Command queuing						✓
Retry				✓		
Partition			✓			✓
Copy-back	✓	✓				
Log						
Boot	✓		✓	✓		✓
Reset	✓	✓	✓	✓	✓	✓
Inquiry						✓
Power management						
Sleep			✓	✓		✓
Power down				✓	✓	✓
Deep power-down					✓	
Interrupt			✓			
Auto refresh					✓	
Precharge					✓	
Partial array self-refresh					✓	
Parallel operation		✓				✓

Table 3.7 New trends in SoC memories

CBRAM	Conductive bridge
ReRAM	Resistive
PCRAM	Phase change
FeRAM	Ferroelectric
ST-MRAM	Spin-torque magnet
Memristor	It is called the fourth element (change of flux with charge) as depicted in Fig. 3.21

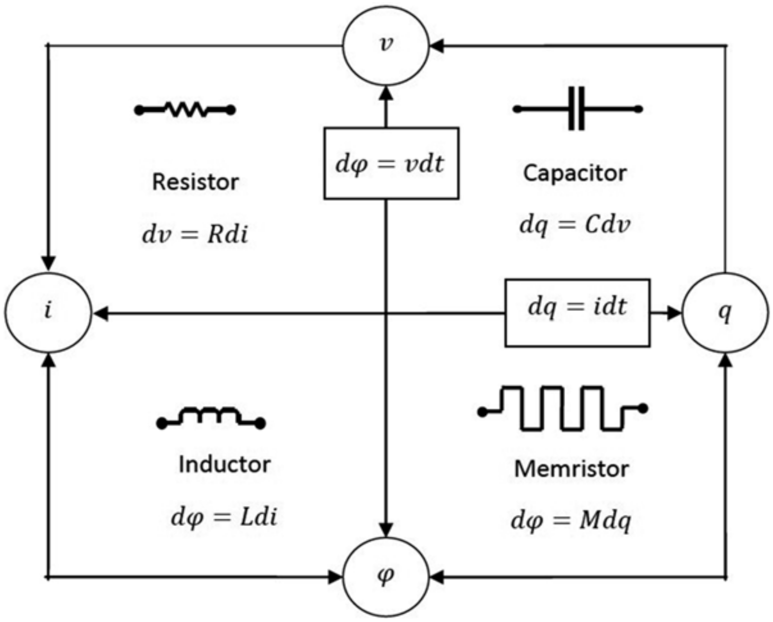


Fig. 3.21 The fourth element

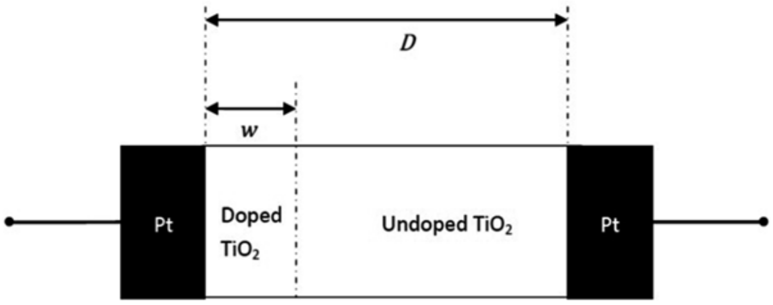


Fig. 3.22 Memristor structure

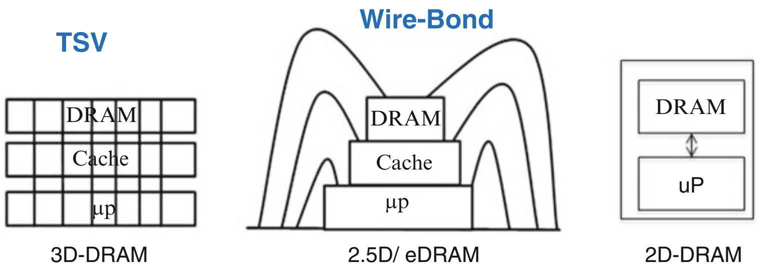


Fig. 3.23 3D DRAM as compared to 2D and 2.5 D DRAM

References

1. Akesson B, Huang P, Clermidy F, Dutoit D (2011) Memory controllers for high-performance and real-time MPSoCs. In: Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM, New York
2. Clermidy F, Darve F, Dutoit D (2011) 3D Embedded multi-core: some perspectives. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, Grenoble
3. Weis C, Wehn N, Igor L, Benini L (2011) Design space exploration for 3D-stacked DRAMs. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, Grenoble, pp 1–6
4. Min S, Nam E (2006) Current trends in flash memory technology. In: Asia and South Pacific Conference on Design Automation (ASPDAC), IEEE, Yokohama
5. Loi L, Benini L (2010) An efficient distributed memory interface for many-core platform with 3D stacked DRAM. In: Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE), Germany, pp 99-104
6. Zhang T, Wang K, Feng Y, Song X (2010) A customized design of DRAM controller for on-chip 3D DRAM stacking. In: IEEE Custom Integrated Circuits Conference (CICC), San Jose
7. Jacob B (2008) Memory systems cache, DRAM, disk. Morgan Kaufmann, Burlington
8. http://www.micron.com/~/media/Documents/Products/Presentation/WinHEC_Cooke.pdf
9. Zhang Y, Swanson S (2015) A study of application performance with non-volatile main memory. In: Proceedings of the 31st IEEE Conference on Massive Data Storage, IEEE
10. http://download.microsoft.com/download/d/f/6/df6accd5-4bf2-4984-8285-f4f23b7b1f37/winhec2007_micron_nand_flashmemory.doc
11. Pasricha S, Dutt N (2008) On-chip communication architectures: system on chip interconnects. Morgan Kaufmann, Burlington
12. https://www.jedec.org/
13. http://www.jedec.org/sites/default/files/docs/JESD84-B42.pdf
14. http://www.datasheetcatalog.org/datasheets2/12/1248447_1.pdf
15. http://www.jedec.org/standards-documents/docs/jesd-79-3d
16. Hybrid memory cube (2013) Technical Report Revision 1.0, HMC. www.hybridmemorycube.org. Accessed January 2013
17. Wide I/O single data rate, Technical Report Revision 1.0, Wide IO. Accessed December 2011
18. www.onfi.org
19. http://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs
20. https://www.jedec.org/standards-documents/docs/jesd235
21. Xie Y (2014) Emerging memory technologies. Springer, New York
22. Kavehei O, Iqbal A, Kim YS, Eshraghian K, AL-Sarawi SF, Abbott D (2010) The fourth element: characteristics, modelling, and electromagnetic theory of the memristor. Proc Roy Soc A Math Phys Eng Sci 466:2175–2202
23. Lacaze P-C, Lacroix J-C (2014) Non-volatile memories. Wiley, Hoboken
24. Kim C, Lee H-W, Song J (2014) High-bandwidth memory interface. Springer, New York

Chapter 4
SoC Buses and Peripherals: Features and Architectures

4.1 Introduction

Components connected on a Printed Circuit Board (PCB) or System-on-Board (SoB) can now be integrated onto single chip, hence the development of System-on-Chip (SoC) design as depicted in Fig. 4.1 [1]. SoC improves the bandwidth. The leveraged internal/on-chip bandwidth versus external/off-chip bandwidth as shown in Fig. 4.2.

SoC is not only a chip it is a system, where, **SoC** = Hardware + Software as depicted in Fig. 4.3.

The SoC Hardware includes:

– Embedded processor
– ASIC Logics and analog circuitry
– Embedded memory
– Peripherals

The SoC Software includes:

– OS/RTOS (Middleware, Device Drivers)
– Applications (C/C++, assembly)

One solution to the design productivity gap is to make ASIC designs more standardized by reusing segments of previously manufactured chips. These segments are known as "blocks," "macros," "cores," or "cells." The blocks can either be developed in-house or licensed from an IP company. **Cores** are the basic building blocks. The cores are communicating with each other through buses and with the outer world through peripherals [2–6].

K.S. Mohamed, *IP Cores Design from Specifications to Production*,
Analog Circuits and Signal Processing, DOI 10.1007/978-3-319-22035-2_4

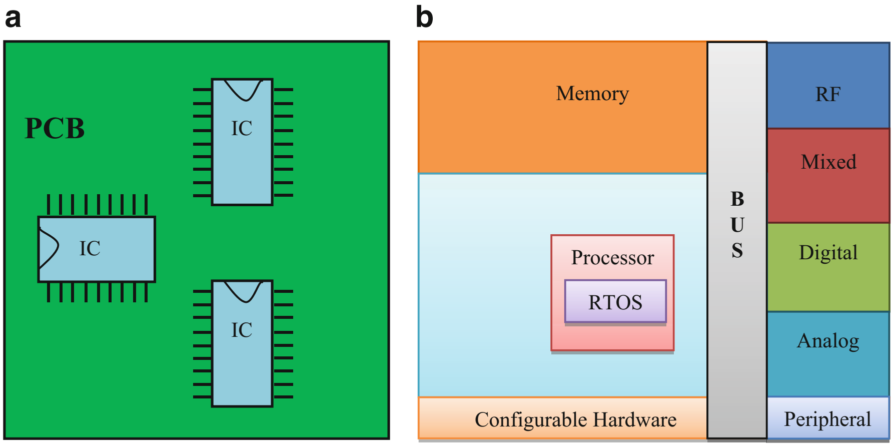


Fig. 4.1 (**a**) SoB versus (**b**) SoC

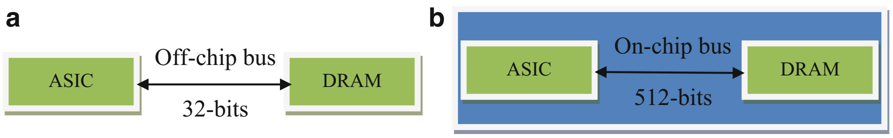


Fig. 4.2 (**a**) SoB bandwidth versus (**b**) SoC bandwidth

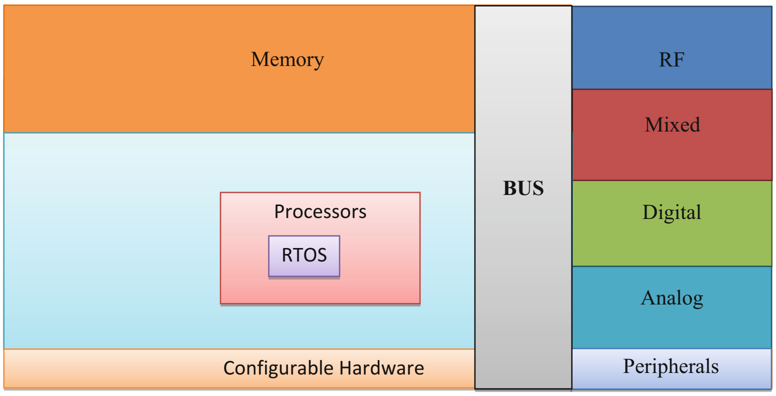


Fig. 4.3 An example of SoC architecture

4.2 SoC Buses and Peripherals Background

The SoC consists of buses and peripherals as depicted in Fig. 4.4, where buses are for communication between different blocks inside the chip and peripherals for communications with outer world. Buses are the simplest and most widely used SoC interconnection networks to connect between different IPs in the SoC [7].

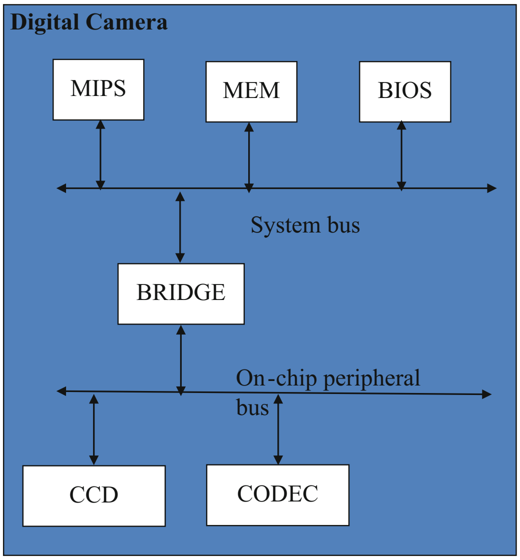


Fig. 4.4 SoC buses and peripherals

Table 4.1 Buses terminology

Bus terminology	Explanation
Master (or initiator)	IP component that initiates a read or write data transfer
Slave (or target)	IP component that does not initiate transfers and only responds to incoming transfer requests
Arbiter	Controls access to the shared bus
	Uses arbitration scheme to select master to grant access to bus
Decoder	Determines which component a transfer is intended for
Bridge	Connects two buses
	Acts as slave on one side and master on the other

Table 4.2 Bus signals

Signal	Explanation
Address	Carry address of destination for which transfer is initiated
	Can be shared or separate for read, write data
Data	Carry information between source and destination components
	Can be shared or separate for read, write data
	Choice of data width critical for application performance
Control	Requests and acknowledgements
	Specify more information about type of data transfer

The bus is a collection of signals (wires) to which one or more IP components (which need to communicate data with each other) are connected. Only one IP component can transfer data on the shared bus at any given time. The most important bus terminologies are summarized in Table 4.1. A bus typically consists of three types of signal lines summarized in Table 4.2.

To implement SoC buses we need standards to make it easy to connect diverse IPs quickly, where standards important for seamless ***integration*** of SoC IPs—helps avoid integration mismatches, where mismatches require development of "logic wrappers" at IP interfaces to ensure correct data transfers and it consumes time to be created, reduces performance, and takes up area.

- E.g., 1—connecting IP with 32 data pins to a 30 bit data bus.
- E.g., 2—connecting IP supporting data bursts to a bus with no burst support.

Two categories of standards for SoC communication are existing:

1. **Standard bus architectures**:
   - Define interface between IPs and bus architecture.
   - Define at least some specifics of bus architecture that implements data transfer protocol.
2. **Socket-based bus interface standards**:
   - Define interface between IPs and bus architecture.
   - Freedom w.r.t choice and implementation of bus architecture.

4.3 SoC Buses: Features and Architectures

The most famous features and architectures of SoCs are summarized in Table 4.3 and the details are below [1, 8, 9].

4.3.1 SoC Bus Topology

1. **Point to point**:
   - Only one master connected to one slave (Fig. 4.5).
   - Simple in design.
   - Optimal in terms of bandwidth, latency, and power.
   - If number of links increases, the area increases and faces routing problems.
2. **Unilevel shared bus**:
   - All masters and slaves share the same bus as depicted in Fig. 4.6.
3. **Hierarchical bus**:
   - Improves system throughput.
   - Multiple ongoing transfers on different buses as depicted in Fig. 4.7.
4. **Ring**:
   All masters and slaves are connected in a ring manner as depicted in Fig. 4.8.

Table 4.3 An overview of bus features and architectures and AHB and AXI as an example

	Topology (interconnect architectures) (bus)						Arbitration (Mux/Tri-state based)					Bus width			Transfers			Timing		Tx control		Tx type			Freq/ Data rate	Applications		
Name	Point to point	Ring	Unilevel shared bus	Hier-archal bus	Intercon-nection network (cross-bar switch)	NOC (router)	Static priority	TDMA	LOTTERY	Round-robin	Token-passing	Data bus width [bit]	Address bus width [bits]	Control bus width [bit]	Split transfer	Pipelined transfer	Burst transfer	Synchronous (SDR|DDR)	Asynchro-nous	Hand-shaking	Pre-amble	Point to point	Multi-cast	Broad-cast	Max freq	Multi-media	Stor-age	Others
AHB				✓			Application-specific					32	32		✓	✓	✓	✓		✓		✓			User defined			✓
AXI				✓			Application-specific					~1024	32			✓	✓	✓		✓		✓			User define			

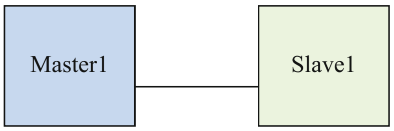


Fig. 4.5 Point to point

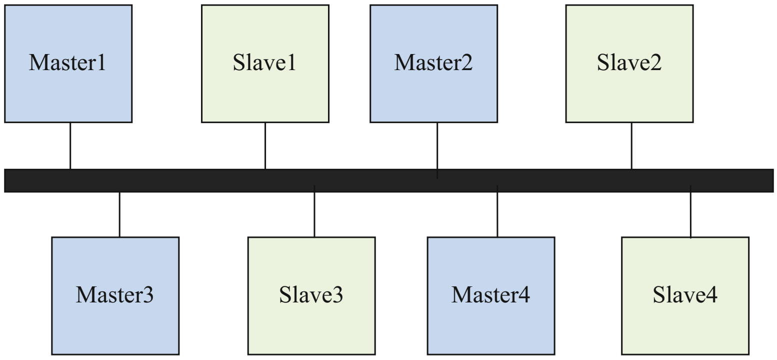


Fig. 4.6 Unilevel shared bus

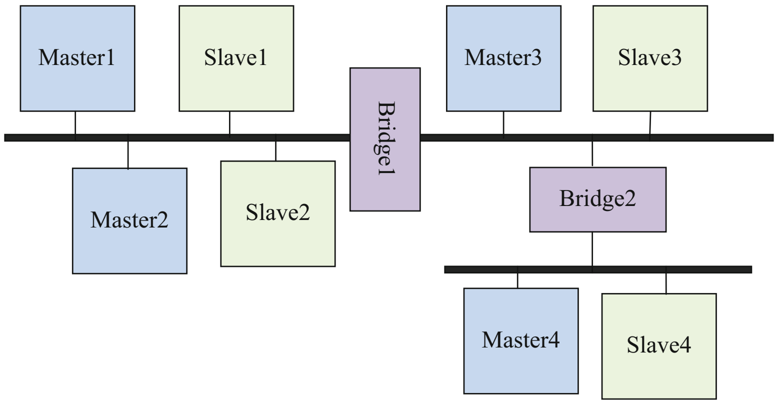


Fig. 4.7 Hierarchical bus

5. **Interconnection network** (**cross-bar switch**):
   - Every master/slave is connected to the remaining masters/slaves via point-to-point topology as depicted in Fig. 4.9.
6. **NOC** (**router**):
   - Each on-chip component connected by an intelligent switch to particular communication wires as depicted in Fig. 4.10.
   - Improvement over standard bus-based interconnections for SoC architectures in terms of throughput and bandwidth [10].

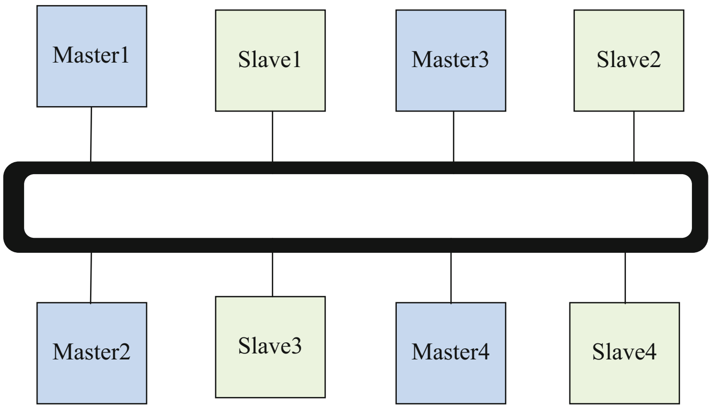


Fig. 4.8 Ring topology

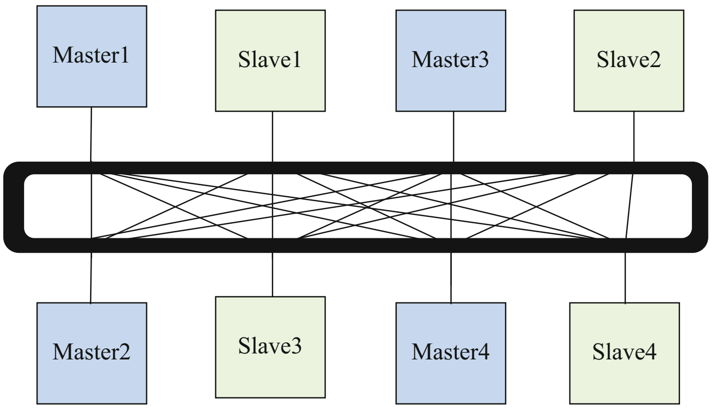


Fig. 4.9 Cross-bar switch (no collision), it sends request to the required slave only

4.3.2 Arbitration (Mux/Tri-State-Based)

The arbitration is Tri-state topology (Fig. 4.11) or mux-based topology (Fig. 4.12) to avoid collision.

The arbitration algorithms are as follows and they are summarized in Table 4.4:

1. **Static priority**

- Masters assigned static priorities.
- Higher priority master request always serviced first.
- Can be preemptive or nonpreemptive.
- May lead to starvation of low-priority masters.

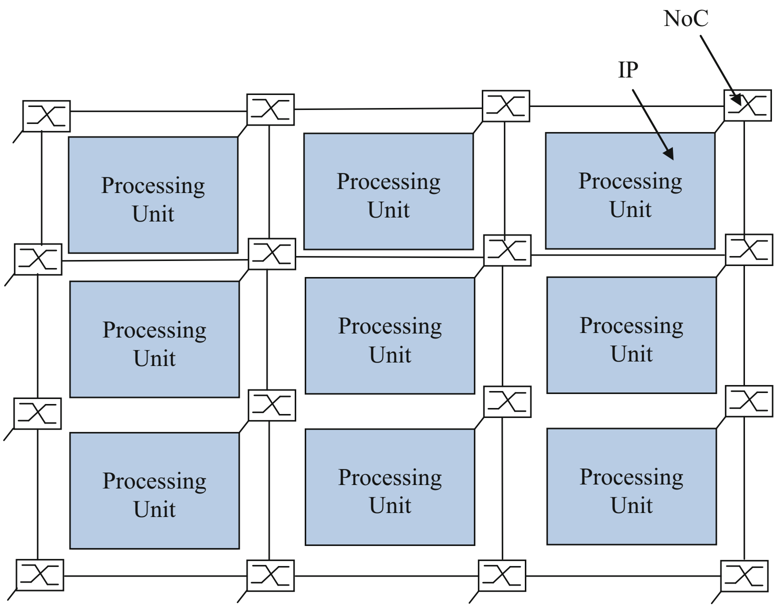


Fig. 4.10 NoC (smarter), select the best path

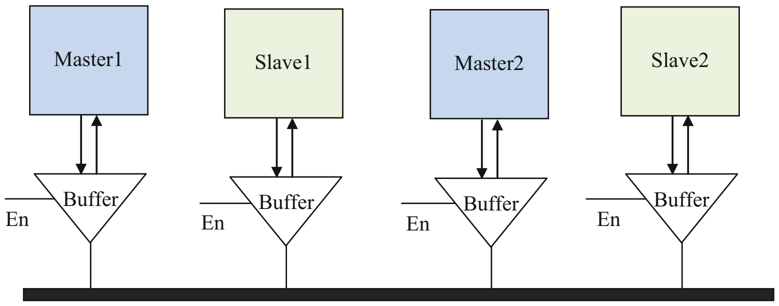


Fig. 4.11 Tri-state arbitration topology

2. **TDMA**

- Uses time division multiple access.
- Assign slots to masters based on BW requirements.
- If a master does not have anything to read/write during its time slots, this leads to low performance.
- Choice of time slot length and number is critical.

3. **LOTTERY (random)**

- Randomly select master to grant bus access.

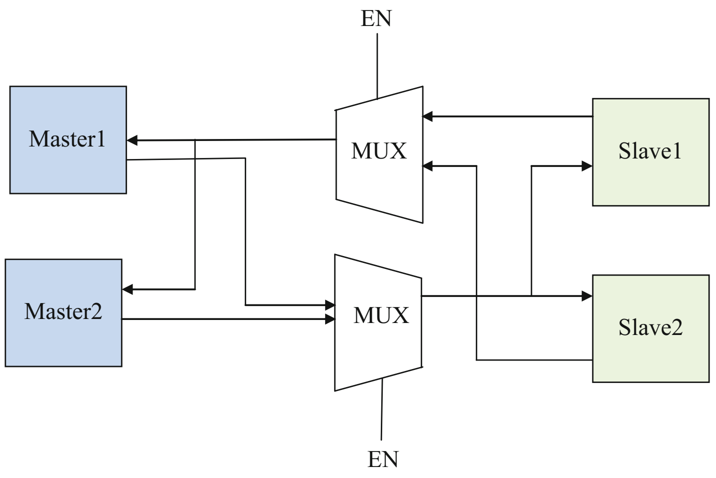


Fig. 4.12 Mux-based arbitration topology

Table 4.4 The arbitration algorithms comparison

Scheme	Description	Advantages	Disadvantages
Static priority	Masters assigned static priorities	Simple	It may lead to starvation of low-priority masters
	Higher priority master request always serviced first		
	It can be preemptive (task can be interrupted) or nonpreemptive (task cannot be interrupted)		
TDMA	Assign slots to masters based on BW requirements	No starvation	If a master does not have anything to read/ write during its time slots, leads to low performance
			Choice of time slot length and number is critical
LOTTERY (Random)	Randomly select master to grant bus access	Simple	Depends on probability
			Starvation
Round-robin	Masters allowed to access bus in a round-robin manner	No starvation—every master guaranteed bus access	High latency for critical data streams
	TDMA but If a master does not have anything to read/write during its time slots the grant moves to another master and so on		
Token-passing	Each master waits for a special token to have a control of the bus, after finishing its operation, it releases the token	Simple	Starvation
			Inefficient if masters have vastly different data injection rates

4. **Round-robin**

   Tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called preemptive scheduling. E.g., round-robin in nonpreemptive scheduling, a running task is executed till completion. It cannot be interrupted. E.g., First In First Out.

   - Masters allowed to access bus in a round-robin manner.
   - No starvation—every master guaranteed bus access.
   - Inefficient if masters have vastly different data injection rates.
   - High latency for critical data streams.

5. **Token-passing**

   - Each master wait for a special token to have a control of the bus, after finishing its operation, it releases the token.

4.3.3 *Transfers*

1. **Nonpipelined transfer**

   - Simplest transfer mode.
   - First request for access to bus from arbiter.
   - On being granted access, set address and control signals.
   - Send/receive data in subsequent cycles.

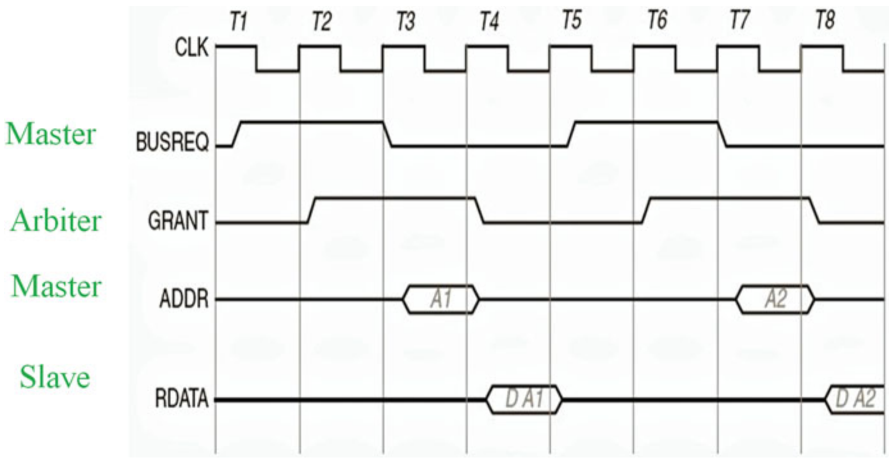


Fig. 4.13 Nonpipelined transfer [11]

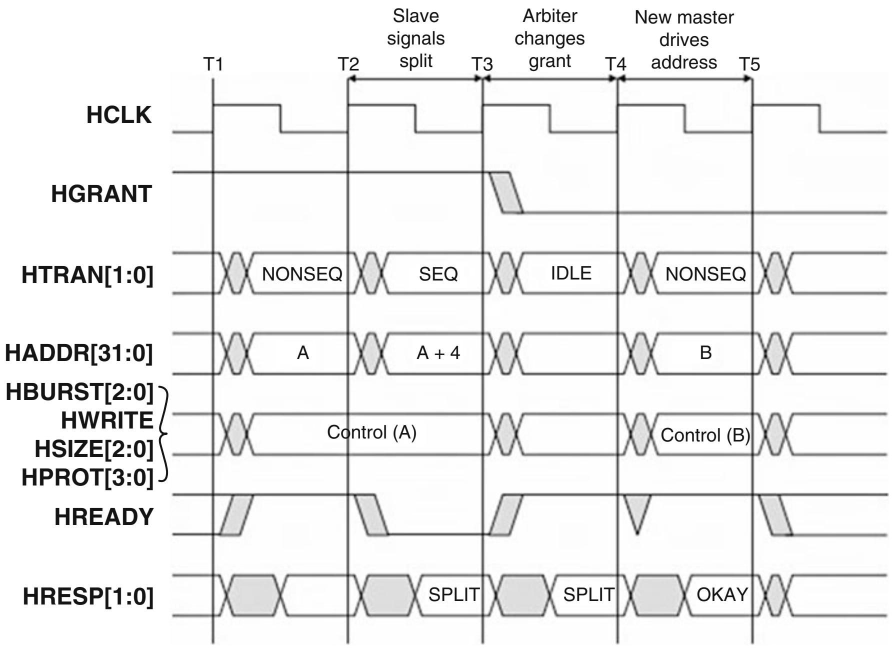


Fig. 4.14 Split transfer [11]

- The operation is summarized in Fig. 4.13. It should receive data of address A1, before sending data of address A2.

2. **Split transfer**

   - If slaves take a long time to read/write data, it can prevent other masters from using the bus. Split transfers improve performance by “splitting” a transaction. Master sends read request to slave. Slave relinquishes control of bus as it prepares data. Arbiter can grant bus access to another waiting master. Allows utilizing otherwise idle cycles on the bus. When slave is ready, it requests bus access from arbiter. On being granted access, it sends data to master (Fig. 4.14).

3. **Pipelined transfer**

   - Overlap address and data phases.
   - Only works if separate address and data buses are present.
   - The operation is summarized in Fig. 4.15, It can send address A2, before receiving data of address A1.

4. **Burst transfer**

   - Send multiple data items, with only a single arbitration for entire transaction.
   - Master must indicate to arbiter it intends to perform burst transfer.

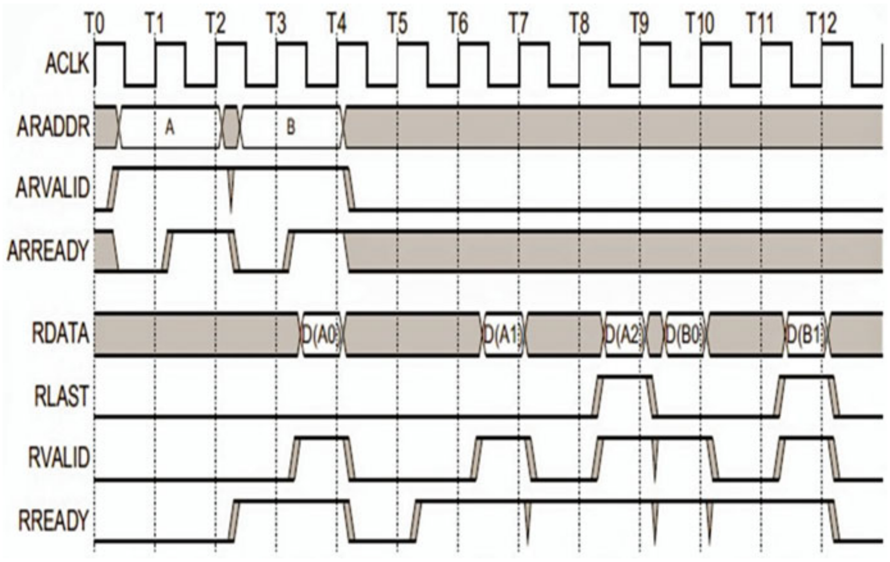


Fig. 4.15 Pipelined transfer [11]

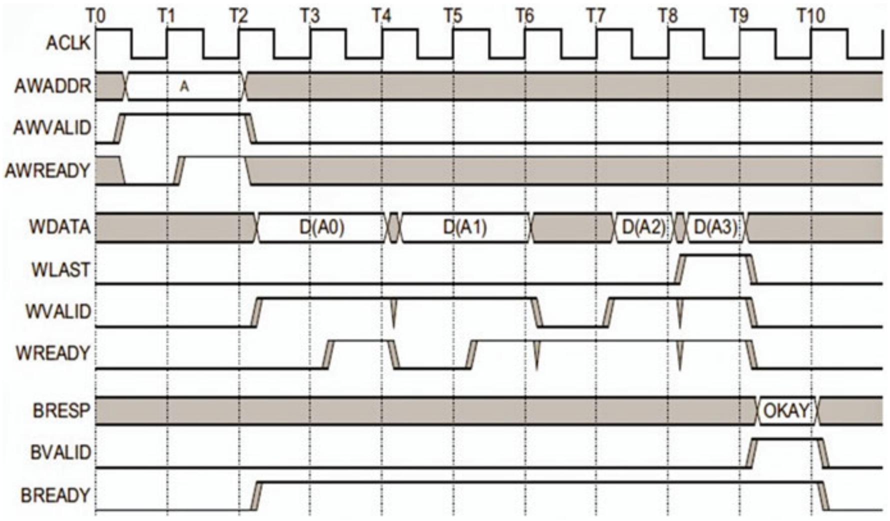


Fig. 4.16 Burst transfer [11]

- Saves time spent requesting for arbitration.

The operation is summarized in Fig. 4.16.

4.3.4 *Timing*

1. **Synchronous**
   - Includes a clock in control lines.
   - Fixed protocol for communication that is relative to clock.
   - Involves very little logic and can run very fast.
   - Require frequency converters across frequency domains.
   - It suffers from clock skew.
   - An example is shown in Fig. 4.17 [12].
2. **Asynchronous**

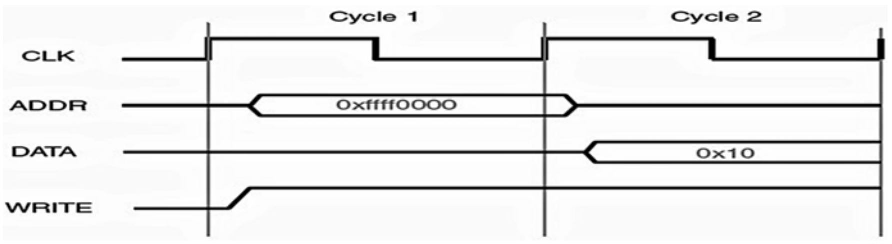


Fig. 4.17 Synchronous timing [12]

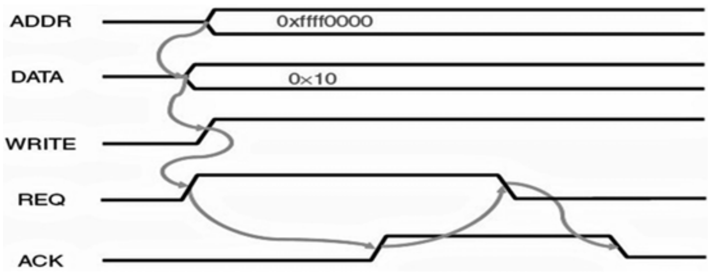


Fig. 4.18 Asynchronous timing [12]

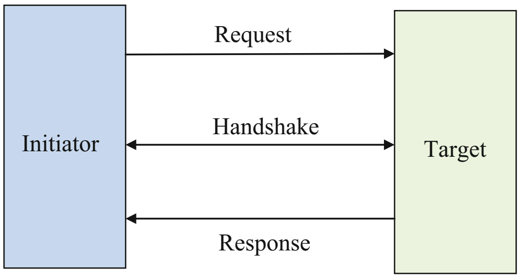


Fig. 4.19 Handshaking Tx control

- Not clocked (data is transmitted and received without accompanying of a clock).
- Requires a handshaking protocol.
- Performance not as good as that of synchronous bus.
- No need for frequency converters, but does need extra lines (pins).
- Does not suffer from clock skew like the synchronous bus.
- An example is shown in Fig. 4.18.

4.3.5 *Tx Control*

Tx control means: "someone is about to transmit data."

1. **Handshaking**
    - It is based on request/response method as depicted in Fig. 4.19.
2. **Preamble**
    - The role of the preamble is to define a specific series of transmission criteria that is understood to mean "someone is about to transmit data." It is a constant pattern.
    - It is a constant pattern or at beginning the bus is high, when it goes low it means I will start communication. It is like a flag.
    - Example is shown in Table 4.5.

4.3.6 *Tx Type*

Table 4.5 Preamble example

Start of data block pattern	1011
Start of frame pattern	0101

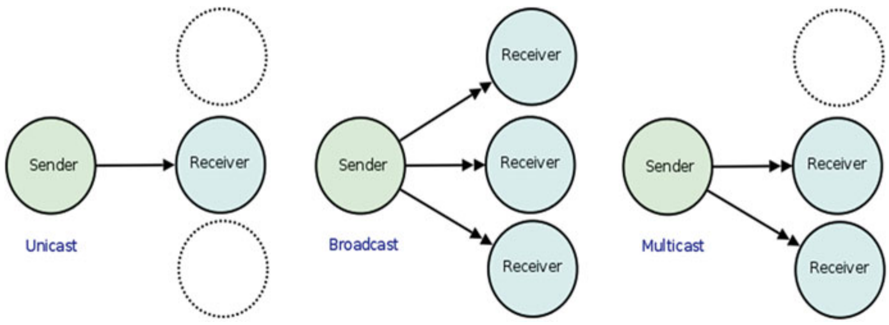


Fig. 4.20 Tx types

Tx types are summarized in Fig. 4.20.

1. **Point to point** (**unicast**)
   - Data is sent from one point to another point.
2. **Multicast**
   - Data is sent from one point to all other points.
3. **Broadcast**
   - Data sent from one or more points to a set of other points.

4.4 Bus Architecture Examples

In this section, we will discuss and define some common IC bus architectures currently in use and on the market

4.4.1 I2C Bus

I2C eliminates the need for address decoders and glue logic, and it reduces space requirements, which keeps designs simple and flexible. It also supports simple constructions and enables easy upgrades. I2C buses are popular in the marketplace for low-speed peripheral devices such as radios, televisions, and personal digital assistants (PDA).

I2C has a physical layout of two bidirectional wires, Serial Data Line (SDA), and Serial Clock Line (SCL), which transmit information between devices. Each device connected to the bus has a unique address assigned to it and can operate in receive and/or transmit mode with a designation as a master or slave. I2C offers the possibility of having multiple masters; however, only one master can transmit data over the bus at a time [13].

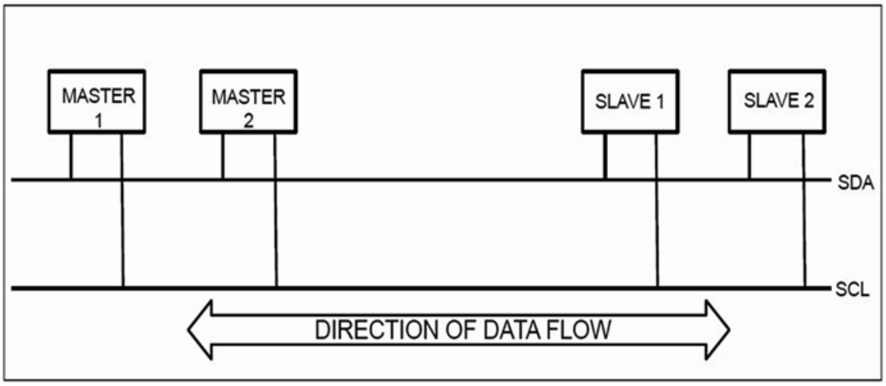


Fig. 4.21 I2C bus topology [13]

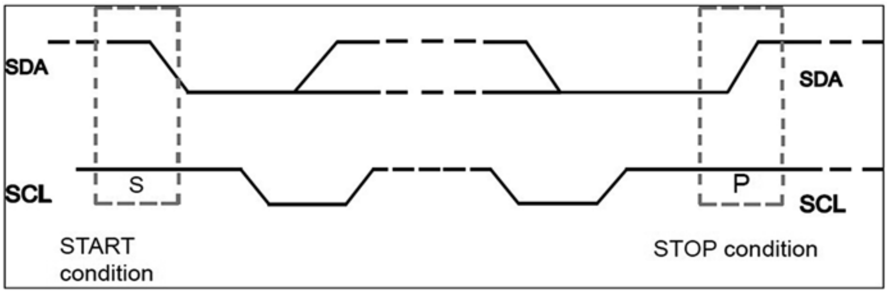


Fig. 4.22 I2C START and STOP conditions [13]

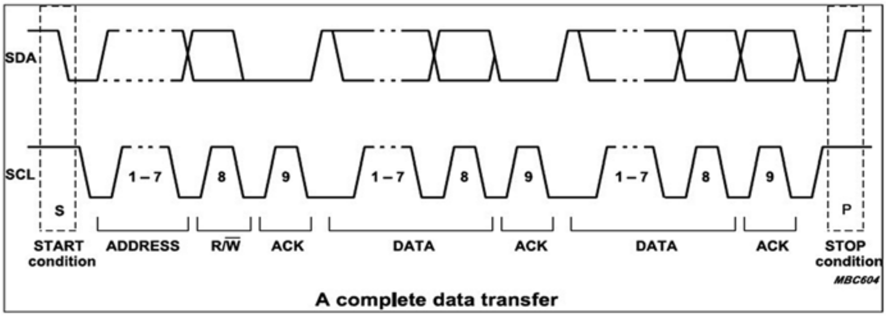


Fig. 4.23 I2C byte format [13]

Figure 4.21 exhibits the topology of I2C. Figure 4.22 depicts high and low states that initiate and terminate transmissions on the bus. I2C requires each byte of data to be eight bits in length before it is placed on the SDA line. Figure 4.23 depicts an I2C sequence.

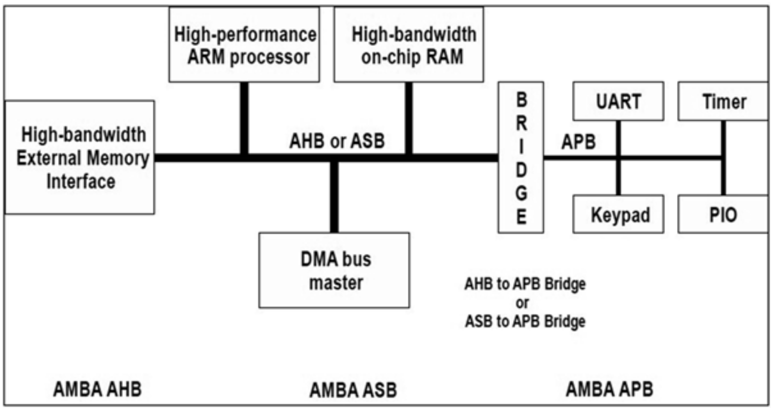

High-performance ARM processor
High-bandwidth on-chip RAM
High-bandwidth External Memory Interface
AHB or ASB
BRIDGE
UART
Timer
APB
Keypad
PIO
DMA bus master
AHB to APB Bridge or ASB to APB Bridge
AMBA AHB
AMBA ASB
AMBA APB

Fig. 4.24 AMBA architecture [14]

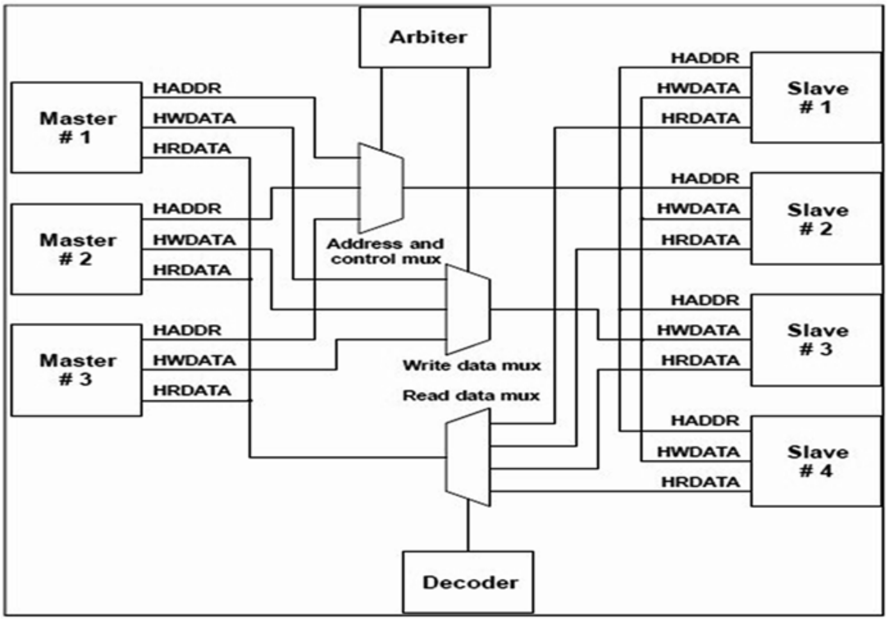


Fig. 4.25 AHB interconnection [14]

4.4.2 *Advanced Microcontroller Bus Architecture (AMBA)*

AMBA is unique in that is it has many distinctly different specifications, versions, bus types, etc. The first is the Advanced High-Performance Bus (AHB), which is used as the backbone for high-performance systems and supports connections between processors, on-chip communications, and off-chip communications. The second type is the Advanced System Bus (ASB), which is a less complex alternative to AHB. The third is the Advanced Peripheral Bus (APB), which is optimized for minimal power consumption and is used for interfacing peripheral devices that do not require high performance or high bandwidth. Figure 4.24 depicts the standard AMBA topology [14].

Figure 4.25 shows a standard AHB interconnection for a standard bus sequence. A typical operational scenario of AHB would involve a master requesting access to a slave to perform a write operation. The arbiter will receive the request signal and determine whether the requesting master device has permission to access the slave device and whether the slave is available (i.e., not performing another operation). Assuming the master device has the appropriate access and the slave device is free from use, the arbiter then transfers the address and control signals to the slave device. The control signals provide the information, direction, and width of the transfer and indicate whether a burst transfer is required. During the transfer, the slave shows the status using response signals (i.e., OKAY, ERROR, RETRY, and SPLIT) [15].

ASB is similar to AHB except that it cannot perform SPLIT transactions. Its bus protocol can be used with a central multiplexor interconnection scheme. Using the interconnection scheme, the bus master will send address and control signals to indicate the desired operation to the central arbiter. The central arbiter reviews the bus master's address and control signals and determines whether the bus master has the appropriate access to the desired slave device (i.e., the master may have read access, but no write access). Data read and response signals from the multiplexor require a central decoder, which will select the appropriate signals from the slave device.

Similar to I2C, the APB is designed for minimal power consumption and reduced complexity. APBs interface with low power, low bandwidth, and low-performance peripherals. The bridge interface between APB and ASB/AHB is the only bus master for APB, but is a slave device on the high-performance ASB/AHB. An APB slave has a simple and flexible interface. Its exact implementation details depend on individual design requirements. Typical operations of an APB slave connected to an ASB bus are read-and-write transfers; however, an APB slave interfacing with an AHB performs the same operations as an APB slave connected to an ASB, but also can perform back-to-back transfers and utilize multiplexing data bus implementations. Multiplexing supports combining read-and-write data buses into a single bus in which read-and-write operations never occur simultaneously.

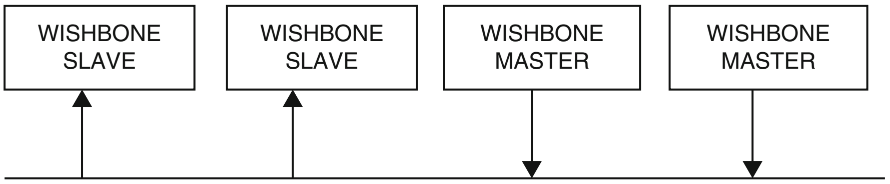


Fig. 4.26 Shared bus interconnection [16]

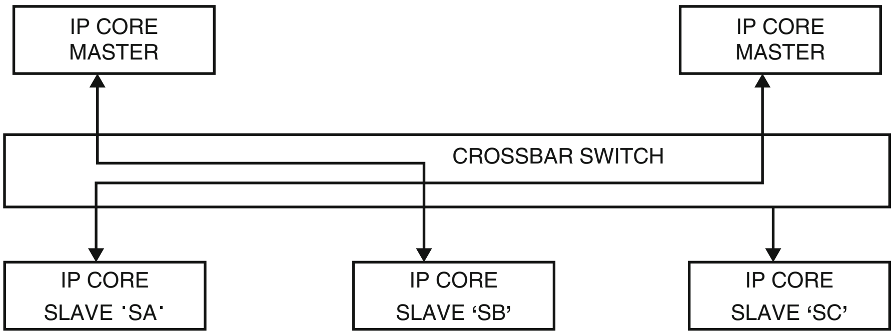


Fig. 4.27 Cross-bar switch interconnection [16]

4.4.3 Wishbone

Wishbone is a SoC bus for portable IP cores and offers perhaps the greatest flexibility in design methodology with semiconductor IP cores. Wishbone is a product of OpenCores, which is an open-source hardware community for professionals and hardware design enthusiasts. Similar to AMBA, the purpose of Wishbone is to ease the integration of SoC components through design reuse. There are three common architectures associated with Wishbone: Shared Bus (Fig. 4.26), Pipeline, and Crossbar (Fig. 4.27) [16].

Designers will choose a shared bus interconnection when there are two or more masters that need to be connected to one or more slaves. The master initiates a bus cycle to a target slave, and then the target slave participates in one or more bus cycles with the master. An arbiter determines when a master may gain access to the shared bus. An arbiter acts like a traffic cop and dictates how shared resources can be accessed.

A crossbar connects two or more masters so that each can access two or more slaves. In this configuration, a master initiates an addressable bus cycle to a target slave. An arbiter determines when each master may gain access to that slave.

The simplest topology is a pipelined topology, in which data is processed in a sequential manner. Data flow architecture exploits parallelism, which speeds up execution time.

4.5 Summary

In this chapter, we introduce a deep introduction for SoC buses and peripherals. We explain in detail their features and architectures. Different SoC bus topologies are discussed such as point to point, unilevel shared bus, hierarchical bus, ring, crossbar bus, NoC. The arbitration algorithms are explored. Moreover, SoC buses examples are explained in detail. We give a methodology for extraction of any SoC bus features from its standard. The different features include topology, arbitration, bus width, transfers, timing, transmission control and type.

References

1. http://eecs.wsu.edu/~pande/Journal_Papers/Paper_IEEE_Proceedings.pdf
2. Stallings W (2003) Computer architecture and organization, 6th edn, Eastern economy edition. PHI Learning, New Delhi
3. Tanenbaum AS (2000) Structured computer organization, 4th edn. Prentice Hall, Upper Saddle River
4. Hamacher VC, Vranesic Z, Zaky S (1996) Computer organization, 4th edn. McGraw Hill, New York
5. Mano MM (2001) Computer system architecture. Prentice Hall, New Delhi
6. Hayes JP (2002) Computer architecture and organization, 3rd edn. McGraw Hill, New York
7. Mitic M, Stojčev M (2006) An overview of on-chip buses. Ser Elec Energy 19(3):405–428
8. http://www.engr.colostate.edu/~sudeep/teaching/ppt/lec06_communication1.pdf. (Accessed 2015)
9. http://facta.junis.ni.ac.rs/eae/fu2k63/stojcev.pdf. (Accessed 2015)
10. Ahmed AB (2015) High-throughput architecture and routing algorithms towards the design of reliable mesh-based many-core network-on-chip systems. Ph.D. dissertation, The University of Aizu
11. Pasricha S, Dutt N (2008) On chip communication architecture: system on chip interconnect, Computer architecture. Morgan Kaufmann, Burlington
12. Murdocca M, Heuring V (2007) Computer architecture and organization
13. Paret D (1997) The I2C bus: from theory to practice. Wiley, Chichester
14. ARM (1999) AMBA specification revision 2.0
15. Patil RP, Sangamkar PV (2015) A review of system-on-chip bus protocols. Int J Adv Res Electr Electron Instrum Eng 4(1):271–281
16. OpenCores (2011) Wishbone B4: WISHBONE system-on-chip (SOC) interconnection architecture for portable IP cores, 4th ed. cdn.opencores.org/downloads/wbspec_b4.pdf. Accessed 5 November 2011

Chapter 5
Verilog for Implementation and Verification

5.1 Introduction

Hardware Description Language (HDL) is widely used as it is easier to explore different design options (e.g., throughput vs. latency), reduce design time and cost significantly, allows larger designs, can reuse design to target different technologies as it is technology-independent language.

The HDL description can be synthesized into a gate-level description of a chosen technology. Two popular HDLs in the IC design: VHDL which is similar to Ada/Pascal in software programming and case insensitive, Verilog which is similar to C language, case sensitive (CLOCK, clock, and Clock are different), a bit easier to learn. The differences are shown in Table 5.1. Disadvantages of HDL are that quality of synthesis varies from tool to tool and synthesis is not standard [1].

Verilog hardware language is used to simulate RTL. Verilog and C bear a lot of similarities in both syntaxes and semantics. Of course, Verilog incorporates features specifically designed for hardware modeling. For instance, Verilog can directly manipulate vectors and support a larger set of bit-level operations such as concatenation and reduction. Such disparities can be handled by adding new functions in C. The most important difference, however, is that Verilog allows two types of assignments, blocking and nonblocking, while C only has blocking assignments. A blocking assignment has to finish before its next statement can be executed, but a nonblocking statement allows its succeeding statements to run concurrently [2]. Figure 5.1 shows a comparison between software and hardware from execution-time point of view [3–10]. Verilog is hardware language not a programming language like C.

Verilog can be used for design and for verification. When trying to write Verilog you should think hardware not software. The main difference from software is time notation, Bit/vector data type, and parallelism.

Poorly written HDL code will either be synthesizable, functionally incorrect, or lead to poor performance/area/power results.

K.S. Mohamed, *IP Cores Design from Specifications to Production*,
Analog Circuits and Signal Processing, DOI 10.1007/978-3-319-22035-2_5

Table 5.1 Differences between VHDL and verilog

VHDL	Verilog
Like ADA	Like C
Verbose	Concise
Harder to learn	Easier to learn
Not case sensitive	Case sensitive
Better in high-level behavior modeling	Doesn't have the ability to define new data types
Level of abstraction (3)	Level of abstraction (4), includes switch level

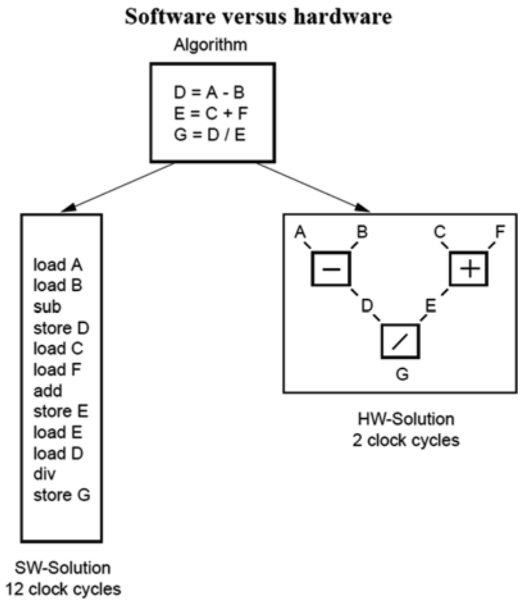
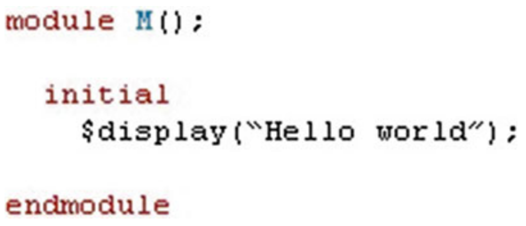


Fig. 5.1 Software versus hardware [3]

5.2 Verilog for Implementation

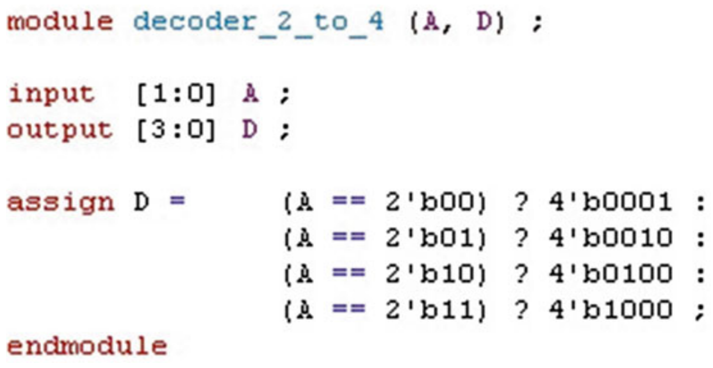
5.2.1 Introduction

A complete Verilog description consists of a module in which the interface signals are declared and the functionality of the component is described. Verilog provides constructs and mechanisms for describing the structure of components which may be constructed from simpler subsystems. Verilog also provides some high-level description language constructs (e.g., loops, conditionals) to model complex behavior easily. Finally, the underlying timing model in Verilog supports both the concurrency and delay observed in digital electronic systems.



```
module M();

  initial
    $display("Hello world");

endmodule
```

Fig. 5.2 Verilog “hello world” example. It starts with the keyword module followed by the name of the module. The keyword “initial” marks the beginning of the operation of the component. The keyword “endmodule” marks the end of the module



```
module decoder_2_to_4 (A, D) ;

input  [1:0] A ;
output [3:0] D ;

assign D =      (A == 2'b00) ? 4'b0001 :
                (A == 2'b01) ? 4'b0010 :
                (A == 2'b10) ? 4'b0100 :
                (A == 2'b11) ? 4'b1000 ;
endmodule
```

Fig. 5.3 Declaration example

In Verilog, a circuit is a module. Module encapsulates structural and functional details. To model any IP using Verilog, you should follow the following steps:

1. Declare a module (Fig. 5.2 shows “hello world” example).
2. Declare the ports type (connectivity).
   (a) Input
   (b) Output
   (c) Inout (bidirectional)
3. Declare the ports size (connectivity).
   (a) **Scalar** (single bit) input A;
   (b) **Vector** (multiple bits) input [0:4] A;
   (c) **Array** input A [0:4];
   (d) **Memory** input [7:0] A [0:7]; // multidimensional arrays are not allowed.
4. Declare the module contents.

   An example for declaration is shown in Fig. 5.3.

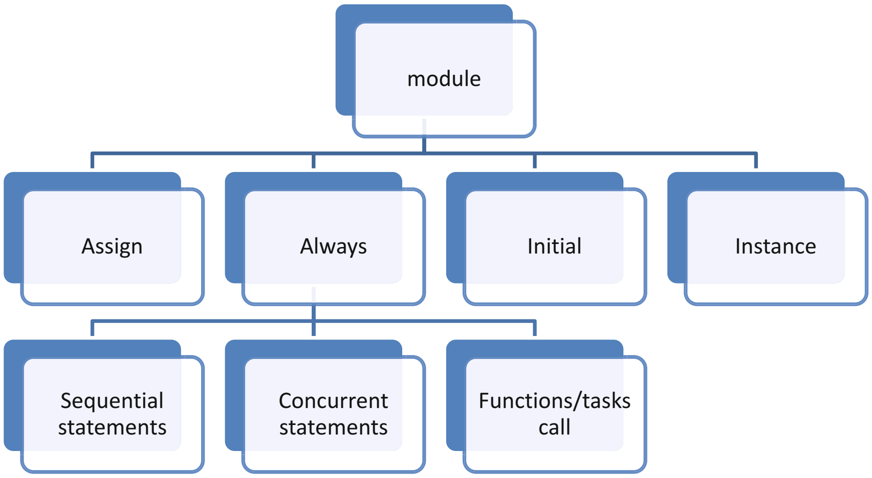
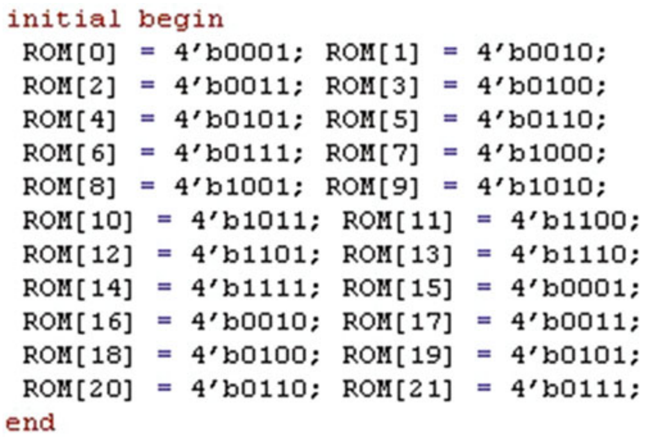


Fig. 5.4 Verilog hierarchy. Putting it all together



```
initial begin
 ROM[0]  = 4'b0001; ROM[1]  = 4'b0010;
 ROM[2]  = 4'b0011; ROM[3]  = 4'b0100;
 ROM[4]  = 4'b0101; ROM[5]  = 4'b0110;
 ROM[6]  = 4'b0111; ROM[7]  = 4'b1000;
 ROM[8]  = 4'b1001; ROM[9]  = 4'b1010;
 ROM[10]  = 4'b1011; ROM[11]  = 4'b1100;
 ROM[12]  = 4'b1101; ROM[13]  = 4'b1110;
 ROM[14]  = 4'b1111; ROM[15]  = 4'b0001;
 ROM[16]  = 4'b0010; ROM[17]  = 4'b0011;
 ROM[18]  = 4'b0100; ROM[19]  = 4'b0101;
 ROM[20]  = 4'b0110; ROM[21]  = 4'b0111;
end
```

Fig. 5.5 Initial block usage example

RTL description usually consists of a hierarchy of concurrently (order-independent) running processes (e.g., always, initial blocks, and assign statements), each with arbitrary internal behavior. At the register transfer level, circuit behaviors are represented as a set of interacting processes running concurrently. The minimal unit of parallel execution in Verilog is a process. The verilog hierarchy is shown in Fig. 5.4, where it captures the main features of a complete Verilog model.

Initial block is executed only once, at the beginning of the simulation, and it is useful for verification, for example, to initialize ROM (Fig. 5.5).

5.2.2 *Data Representation*

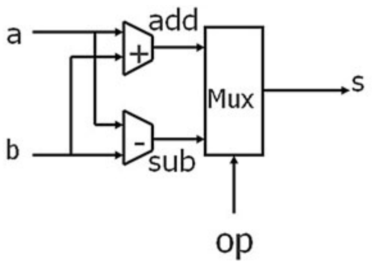
Verilog data types are shown in Table 5.2. Verilog supports built-in data type not user-defined data types. To define an internal signal which is not input nor output we use “wire” for combinational circuits as depicted in Fig. 5.6 or we use “reg” for asynchronous sequential circuits as depicted in Fig. 5.7 or for synchronous sequential circuits as depicted in Fig. 5.8. Note that if the circuit contains sequential and combinational logic, we should separate them. Assign for combinational logic and always for sequential logic.

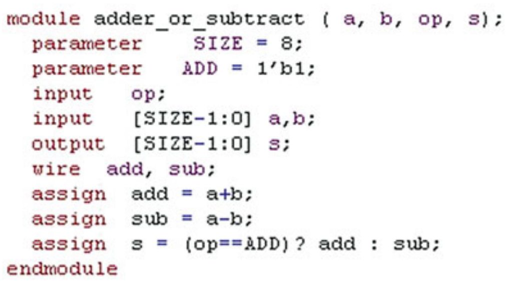
Table 5.2 Verilog datatypes

Data type	Description
Reg	Store data
Wire	Physical connection
Tri1	A net in verilog that pull-up the output if it is not driven
Tri0	A net in verilog that pull-down the output if it is not driven
Parameter	To ease configuration. If not overwritten, they keep their default value
Localparam	Like parameters, but cannot be modified hierarchically during the instantiation

Array	reg [7:0] ram [0:7];
	// to reset it use for loop
	for (i = 0; i < 8; i++) begin ram[i] <= 0; end

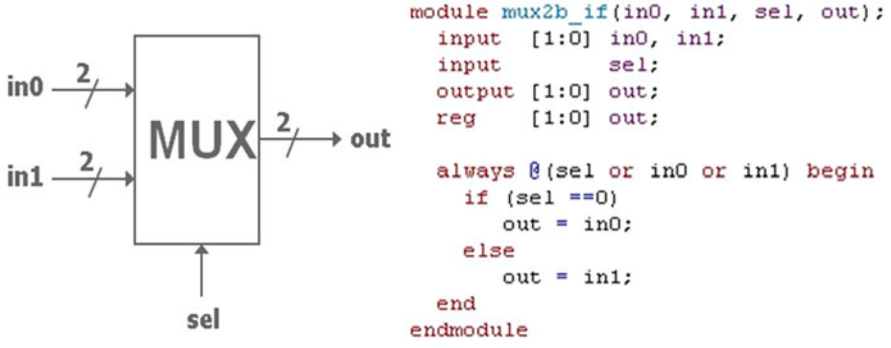
Preprocessor directive	‘define CMD0 4’b0100 like a global parameter ‘define INRANGE(x) ((x) > 2 && (x) < 5) // parameterized macro
ifdef	Used to enable or disable some features
Enum	enum integer {step1 = 0, step2 = 1, tep3 = 2} state; Make debugging easy using waveforms
‘include	‘include “timing.vh”





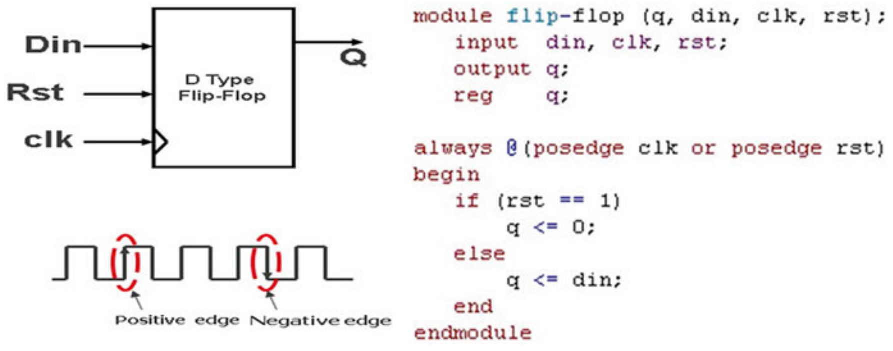
```
module adder_or_subtract ( a, b, op, s);
  parameter    SIZE = 8;
  parameter   ADD = 1'b1;
  input    op;
  input    [SIZE-1:0] a,b;
  output   [SIZE-1:0] s;
  wire  add, sub;
  assign  add = a+b;
  assign  sub = a-b;
  assign  s = (op==ADD)? add : sub;
endmodule
```

Fig. 5.6 Wire usage in verilog



```
module mux2b_if(in0, in1, sel, out);
  input   [1:0] in0, in1;
  input         sel;
  output  [1:0] out;
  reg     [1:0] out;

  always @(sel or in0 or in1) begin
    if (sel ==0)
       out = in0;
    else
       out = in1;
  end
endmodule
```

Fig. 5.7 Reg usage in verilog for asynchronous sequential circuits. Always is triggered when it has finished executing and one of the events in the sensitivity list happens. Use always @(*) instead of writing the whole sensitivity list



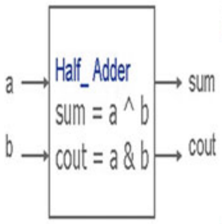
```
module flip-flop (q, din, clk, rst);
   input  din, clk, rst;
   output q;
   reg    q;

always @(posedge clk or posedge rst)
begin
   if (rst == 1)
      q <= 0;
   else
      q <= din;
   end
endmodule
```

Fig. 5.8 Reg usage in verilog for synchronous sequential circuits

5.2.3 *Verilog Coding Style*

Verilog is one language, but it contains many coding Styles. Verilog description can be structural or behavioral. Behavior means what does it do? (Boolean Expressions or FSM). Structure means what is it composed of? (Blocks, gates). An example to show the difference between the behavioral and structural implementation is shown in Fig. 5.9.

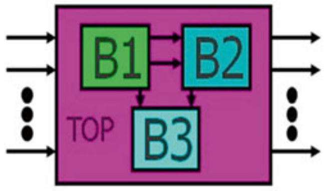
For complex design we partition the modules into submodules as depicted in Fig. 5.10 and use generate statement to reduce the manual coding effort (Fig. 5.11), generate statement is written parallel to always not inside it. Another example which is useful to show the importance of generate statement is n-stage FIR filter design.



```
//Structural
module half_add1 (a, b, sum, cout);
   input   a, b;
   output sum, cout;
   wire     cout_bar;

   xor (sum, a, b);
   nand (cout_bar, a, b);
   not (cout, cout_bar);
endmodule
```

```
//Behavioral
module half_adder2(a, b, sum, cout);
   input a, b;
   output sum, cout;
     assign cout = a & b;
     assign sum  = a ^ b;
endmodule
```

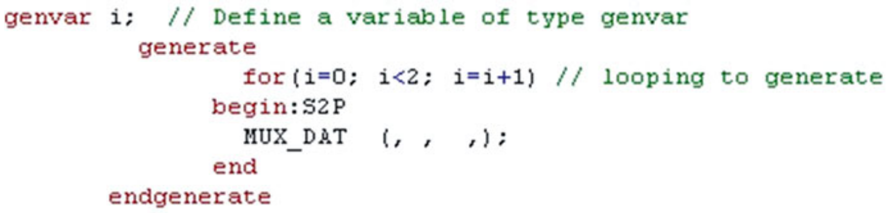
Fig. 5.9 Structural versus behavioral implementation



```
module B1(a, b, c);
   ………..
endmodule
module B2(a, b, c, d);
   ………..
endmodule
module B3(x, y, z);
   ………..
endmodule
```

```
module top (q, w, e, f);
…..
B1 b1(q, w, a, b, c);
B2 b2(a, b, d, e);
B3 b3(c, d, f);
…..
endmodule
```

Fig. 5.10 Submodules example



```
genvar i;  // Define a variable of type genvar
      generate
           for(i=0; i<2; i=i+1) // looping to generate
         begin:S2P
           MUX_DAT  (, ,  ,);
         end
     endgenerate
```

Fig. 5.11 Generate statement example

5.2.4 *Verilog Operators and Control Constructs*

Verilog HDL operators are summarized in Table 5.3.

The fundamental control constructs are shown in Fig. 5.12. If statement is used only in always block (Fig. 5.13). Same for "case" statement (Fig. 5.14). The iteration examples are shown in Fig. 5.15.

Tasks and functions are used in HDL languages. Data is passed to the task or function, processing is done, and the result is returned to the main procedure. Functions are very much similar to tasks, with very little difference, e.g., a function cannot drive more than one output and, also, it cannot contain delays. The differences between tasks and functions are summarized in Table 5.4.

Table 5.3 Verilog HDL operators

+	Binary addition
–	Binary subtraction
&	Bit-wise AND
|	Bit-wise OR
^	Bit-wise XOR
~	Bit-wise NOT
==	Equality assign s = (op == ADD) ? a + b : a–b;
>	Greater than
<	Smaller than
{}	Concatenation assign s = {a, b};
? :	Conditional
!	logical NOT
&&	Logical AND
||	Logical OR
!=	Logical inequality
<<	Shift left
>>	Shift right

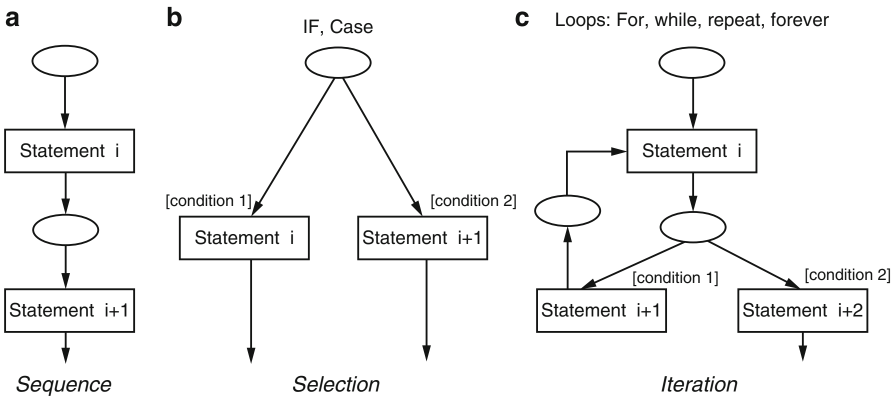
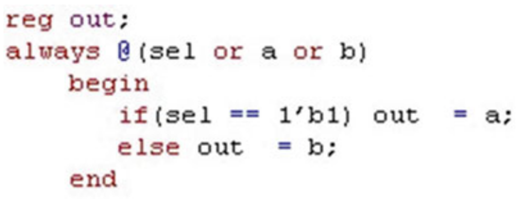
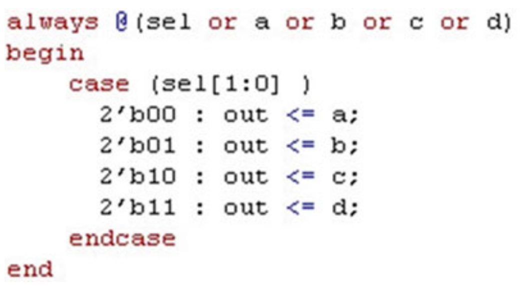


Fig. 5.12 Verilog fundamental control construct

Fig. 5.13 If statement example

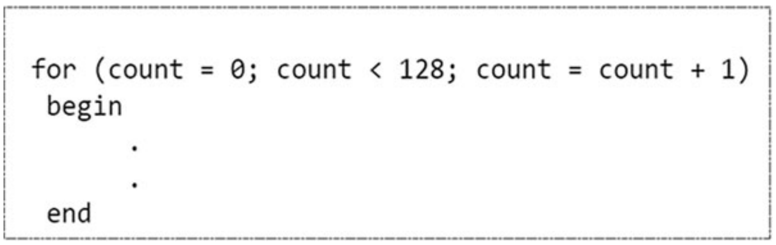


```
reg out;
always @(sel or a or b)
    begin
        if(sel == 1'b1) out  = a;
        else out  = b;
    end
```

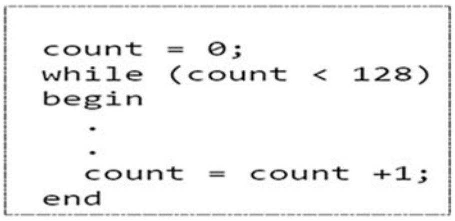


```
always @(sel or a or b or c or d)
begin
    case (sel[1:0] )
      2'b00 : out <= a;
      2'b01 : out <= b;
      2'b10 : out <= c;
      2'b11 : out <= d;
    endcase
end
```

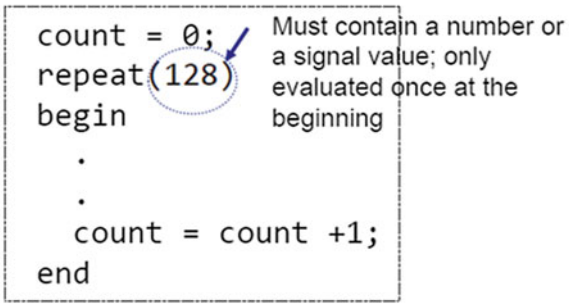
Fig. 5.14 Case statement example, if 2’b00 and 2’b01 are in the same state, we use “,” to separate between them



```
for (count = 0; count < 128; count = count + 1)
 begin
       .
       .
 end
```



```
count = 0;
while (count < 128)
begin
   .
   .
   count = count +1;
end
```



```
count = 0;
repeat(128)
begin
   .
   .
   count = count +1;
end
```

Must contain a number or a signal value; only evaluated once at the beginning

Fig. 5.15 Iteration statements (loops): for, while, repeat

Table 5.4 Differences between functions and tasks

Functions	Tasks
• Can call just another function (not task)	• Can enable other tasks and functions
• Execute in 0 simulation time	• May execute in nonzero simulation time
• No timing control statements allowed	• May contain any timing control statements
• At least one input	• May have arbitrary input, output, or inouts
• Return only a single value	• Do not return any value
• Are defined in a module	
• Do not contain initial or always statements	
• Are called from initial or always statements or other tasks or functions	

task convert;
input [7:0] temp_in;
output [7:0] temp_out;
begin
temp_out = (9/5) *(temp_in + 32);
end
endtask
	function myfunction;
	input a, b, c, d;
	begin
	myfunction = ((a + b) + (c–d));
	end
	endfunction

5.2.5 *Verilog Design Issues*

Race condition happens when two different processes try to write the same signal during the same time step. To avoid it, don't write the same signal in different processes, unless you really know what you do (you know that the two processes will never write the signal in the same time step) and do not make assignments to the same signal in more than one always statement or continuous assign statement. Also, to avoid race condition, always use nonblocking assignments (<=) for sequential circuits and blocking (=) assignments for combinational.

For clock, avoid combinational feedback clock, internally generated clocks, and avoid mixed cock edges. For Resetting, asynchronous RST is preferred, avoid internally generated resets, and for proper operation, all the registers should be resetted into the reset process. Non-Synthesizable Verilog Statements are described in Table 5.5.

5.2.6 *Verilog Template and Reusable Code Tips*

A Verilog template is suggested in Fig. 5.16. The design should start with defining declarations, then module declarations, then parameters declarations, then inputs/outputs declarations, then wire declarations, then registers declarations, then wire assignments, then sequential logics, and then instances declarations.

If you want to write a Verilog reusable code, you may follow the following tips [11]:

1. Don't write code that isn't needed.
2. Don't duplicate code.

Table 5.5 Constructs not supported in synthesis

#	Constructs not supported in synthesis
1	"Hierarchical name reference not supported"
	card.resp_gen.device_reg
2	Time:
	# 580ns
3	Assign on reg not allowed (but it is ok for wire)
	reg [15:0] block_cnt = 2
8	"Mixed blocking and nonblocking assignment is not supported."
	X = 1;
	X <= 1;
9	Real datatype
10	Initial statement
11	Repeat, while, forever statements
12	Division and modulus operators for variables
13	Nonfixed size for loops

```
// define declarations ===========================================================

// Module declarations ==========================================================

// Parameters declarations ======================================================

// Inputs/Outputs declarations ==================================================

// Wire declarations ============================================================

// Register declarations ========================================================

// Wire assignments=============================================================

// Sequential logic =============================================================

// Instances=====================================================================
```

Fig. 5.16 Verilog template

3. Naming conventions: use meaningful names for modules, ports, regs, and wires.
4. Make a task/function do just one thing.
5. Try to reduce coupling.
6. Make your code more modular.
7. Comment, **in detail**, everything that seems like it might be confusing when you come back to the code next time.

Table 5.6 The main building blocks in any digital system

Task	Hardware examples
Arithmetic	+, −, *, %, >>, 2's compliment, CORDIC, ALU
Multiplexing	Arbitration
Comparison	Comparator
Storage	RAM (random access), FIFO (non random access)
Counter	Counter
Communication	Channel encoding, scrambler
Error detection and correction	ECC, CRC
Randomization	LFSR
Encryption	DES
Synchronization	Clocking

8. Include a header mentioning.
   (a) Filename
   (b) Author
   (c) Date
   (d) Time
   (e) Abstract
9. Use indentation.
10. Before a code can been reusable, it has to be usable.

5.2.7 *Main Digital System Building Blocks*

The main building blocks in any digital system can be summarized in Table 5.6. These building blocks can be used to implement or architect any IP.

5.3 Verilog for Verification

How DUV responses can be displayed and checked or monitored. Verilog simulation environments provide two kinds of display of simulation results:

- Graphical (waveforms editors): suitable for small design as you can check by eye or by using system tasks such as $display, $strobe, $monitor. These system tasks are summarized in Table 5.7.
- Text-based: writing or reading to/from a file, suitable for large designs like video streaming.

To check the DUT behavior, we simply drive the inputs and monitor the outputs as depicted in Fig. 5.17. In some cases, the verification should wait a response from the DUT before it can send the next trigger (DUT outputting status indicators to testbench). Verilog can test both combinational (Fig. 5.18) and sequential circuits (Fig. 5.19).

Table 5.7 Verilog system tasks

System task	Description
$display	Display strings, expression, or values to standard output
$monitor	Same as display but displays when any of the values change
$stop	Suspend simulation, put in interactive mode
$finish	Stop simulation altogether

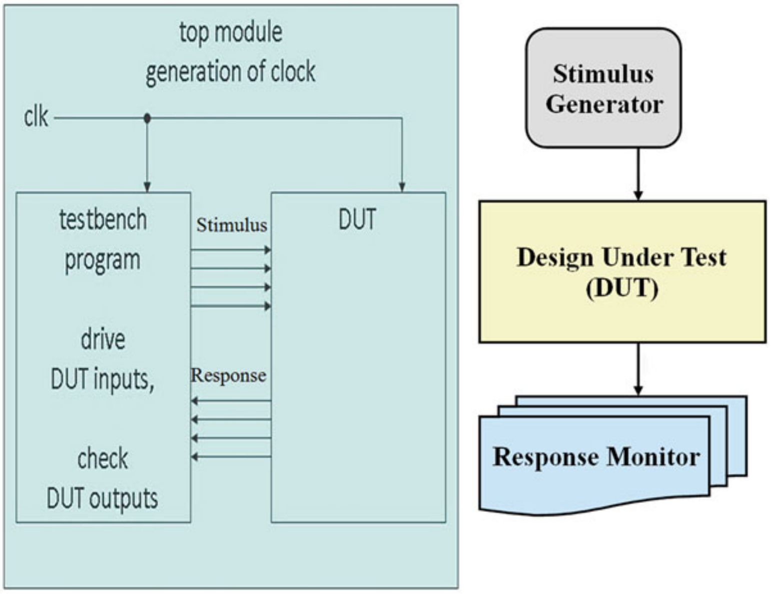
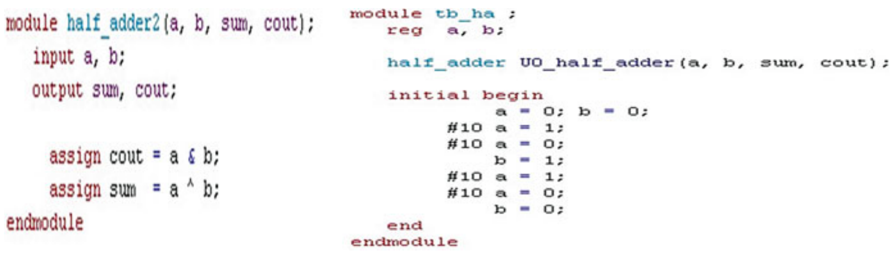


Fig. 5.17 Testbench structure



```
module half_adder2(a, b, sum, cout);
  input a, b;
  output sum, cout;

    assign cout = a & b;
    assign sum  = a ^ b;
endmodule
```

```
module tb_ha ;
    reg  a, b;

    half_adder U0_half_adder(a, b, sum, cout);

    initial begin
                a = 0; b = 0;
            #10 a = 1;
            #10 a = 0;
                b = 1;
            #10 a = 1;
            #10 a = 0;
                b = 0;
    end
endmodule
```

Fig. 5.18 Verification example of half adder. # Means delay

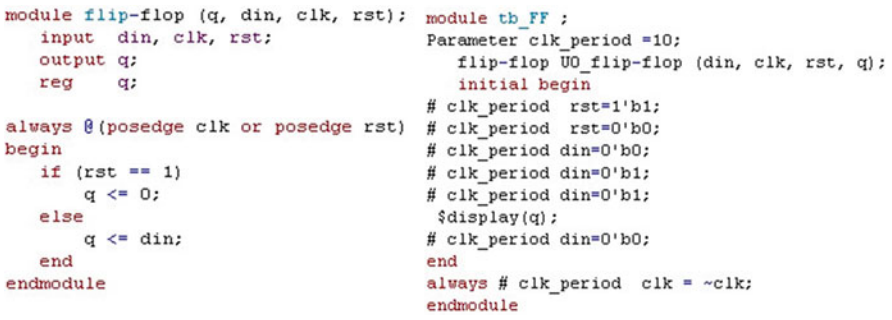
To write testbench, it is important to have the design specifications of the DUT. Specifications need to be understood clearly and test-plan should be done accordingly. The test-plan documents the testbench architecture and the test scenarios in detail.

To reduce the verification time, we can call C code inside Verilog as depicted in Fig. 5.20. Verilog PLI (Programming Language Interface) is a mechanism to invoke C or C++ functions from Verilog code. Use these Functions in Verilog code

(Mostly Verilog Testbench). Compile C++ to generate shared libs Based on *simulator*, pass the C/C++ function details to simulator during compile process of Verilog Code [12].

We can also call VHDL-code inside Verilog to reduce verification time, if you have a preexisting VHDL-code (Fig. 5.21). To instantiate a VHDL module inside a Verilog design, make sure the two files are in the same directory and that they have been added to the project for compilation.

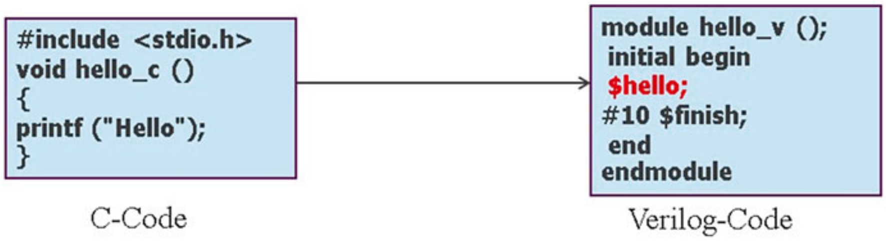
For Text-based verification, writing or reading to/from a file example is shown below (Fig. 5.22):



```
module flip-flop (q, din, clk, rst);
   input  din, clk, rst;
   output q;
   reg    q;

always @(posedge clk or posedge rst)
begin
   if (rst == 1)
       q <= 0;
   else
       q <= din;
   end
endmodule
```

```
module tb_FF ;
Parameter clk_period =10;
   flip-flop U0_flip-flop (din, clk, rst, q);
   initial begin
# clk_period  rst=1'b1;
# clk_period  rst=0'b0;
# clk_period din=0'b0;
# clk_period din=0'b1;
# clk_period din=0'b1;
 $display(q);
# clk_period din=0'b0;
end
always # clk_period  clk = ~clk;
endmodule
```

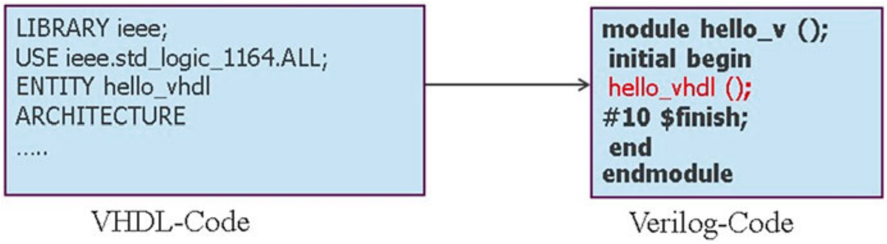
Fig. 5.19 Verification example of flip-flop



```
#include <stdio.h>
void hello_c ()
{
printf ("Hello");
}
```

```
module hello_v ();
 initial begin
 $hello;
#10 $finish;
 end
endmodule
```

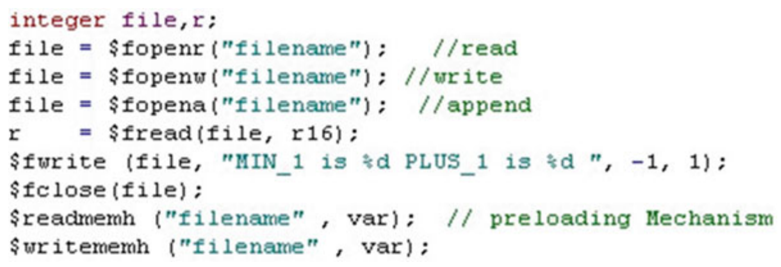
Fig. 5.20 Call C code inside verilog



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY hello_vhdl
ARCHITECTURE
.....
```

```
module hello_v ();
 initial begin
 hello_vhdl ();
#10 $finish;
 end
endmodule
```

Fig. 5.21 Call VHDL-code inside verilog



```
integer file,r;
file = $fopenr("filename");   //read
file = $fopenw("filename"); //write
file = $fopena("filename");  //append
r    = $fread(file, r16);
$fwrite (file, "MIN_1 is %d PLUS_1 is %d ", -1, 1);
$fclose(file);
$readmemh ("filename" , var);  // preloading Mechanism
$writememh ("filename" , var);
```

Fig. 5.22 Writing or reading to/from a file example

5.4 Logic Simulators

Logic simulation is one of the most intensively studied problems in the field of electronic design automation. Existing sequential logic simulators virtually fall into two categories, oblivious simulation and event-driven simulation.

1. The **oblivious** (cycle-based) simulation takes a straightforward approach in which all logic elements are evaluated at every simulation step, no matter they undergo logic transitions or not [13].
2. **Event-driven** simulation was proposed to improve the efficiency of oblivious simulation. An event-driven simulator only evaluates logic modules whose input ports receive new values. Due to its higher efficiency, event-driven simulation has become the workhorse of virtually all commercial and research logic simulators.

From an implementation point of view, a logic simulator could be either interpretive or compiled.

1. **Interpretive** maps the simulated circuit into an internal representation. The response to input patterns can then be evaluated on the representation.
2. **Compiled** translates the circuit into machine code for direct execution. The underlying idea is to take advantage of the similarity between logic operations and CPU instructions.

Parallel logic simulation has attracted considerable research efforts in the past 40 years for its strong potential. An intuitive approach is to use multiple processors to evaluate simultaneously happened events in parallel. However, it has been proved that such parallelism is not sufficient to maintain a decent speed-up due to the following two reasons: (1) generally only a small percentage (e.g., ~1 %) of all circuit elements have active events, and (2) not all elements with active events can be handled simultaneously because the logic dependency actually implies a partial ordering in which the events have to be processed.

Many parallel simulation protocols have been proposed to extract a higher level of inherent parallelism. Basically, these protocols can be classified into two categories, conservative and optimistic.

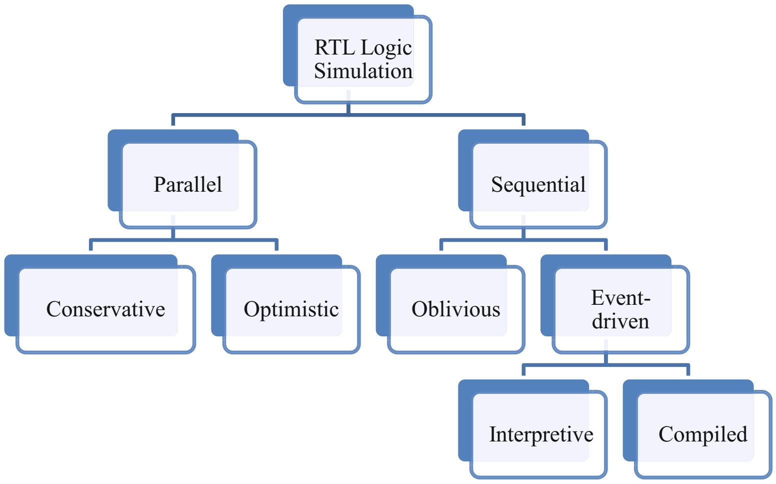


Fig. 5.23 Logic simulation classifications

1. The **conservative** protocol enforces the causal relation during simulation in the sense that events happened earlier are always simulated ahead of later events.
2. The **optimistic** protocol allows the causal relation to be temporarily violated for higher parallelism. However, a roll-back is necessary if a later evaluation invalidates earlier simulation results. Figure 5.23 summarizes these types of logic simulators.

Questa™ is a CPU-based sequential simulator; there is a **GPU-based** parallel simulator for acceleration [14]. A GPU includes a number of multiprocessors which communicate through a small shared memory bank. Questa platform is shown in Fig. 5.24 and their detailed usages are shown in the next subsections [15].

5.4.1 Questa Simulation

The **Questa Simulator** combines high performance and capacity simulation with unified advanced debug capabilities for the most complete native support of Verilog, SystemVerilog, VHDL, SystemC, PSL, and **UPF** (power aware).

The Questa Advanced Simulator is the core simulation and debug engine of the Questa Verification Platform; the comprehensive advanced verification platform capable of reducing the risk of validating complex FPGA and SoC designs.

Questa spans the levels of abstraction required for complex SoC and FPGA design and verification from TLM (Transaction Level Modeling) through RTL, gates, and transistors and has superior support of multiple verification methodologies including Assertion-Based Verification (ABV), the Open Verification Methodology (**OVM**), and the Universal Verification Methodology (UVM) to increase testbench productivity, automation, and reusability.

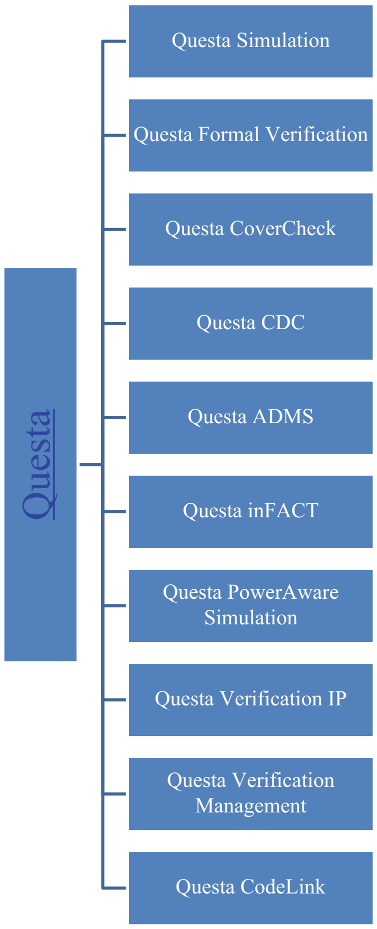


Fig. 5.24 Questa platforms

The Questa Advanced Simulator achieves industry-leading performance and capacity through very aggressive global compile and **simulation optimization algorithms** of SystemVerilog and VHDL, improving SystemVerilog and mixed VHDL/SystemVerilog RTL simulation performance by up to 10×.

Questa also supports very fast time-to-next simulation and effective library management while maintaining high performance with unique capabilities to preoptimize and define debug visibility on a block-by-block basis enabling dramatic regression throughput improvements of up to 3× when running a large suite of tests.

To increase simulation performance for large designs with long simulation times, Questa also has a multi-core option. Questa **Multi-Core** takes advantage of modern compute systems by partitioning the design to run in parallel on multiple CPUs or computers using either automatic or manually driven partitions.

To achieve even greater performance, Questa supports **TBX**; the highest performance Transaction Level link to the Veloce platform enabling a 100× increase in performance with debug visibility and a common testbench.

5.4.2 Questa Formal Verification

It complements simulation-based RTL design verification. The **Questa Formal Verification** tool complements simulation-based RTL design verification by analyzing all possible behaviors of the design to detect any reachable error states. This exhaustive analysis ensures that critical control blocks work correctly in all cases and locates design errors that may be missed in simulation.

Questa Formal Verification can be used as soon as the design is complete to debug blocks before integration, and to find potential errors long before simulation test environments are available. Sharing a common language front end with the Questa Simulator and leveraging the integration with the Unified Coverage Database (UCDB), Questa Formal Verification is the perfect tool to accelerate bug detection, error correction, and **coverage** closure.

Questa Formal Verification analyzes the behavior of the design to identify all design states that are reachable from the initial state. This analysis allows Questa Formal Verification to explore the whole state space in a breadth-first manner, in contrast to the depth-first approach used in simulation.

Questa Formal Verification is therefore able to discover any design errors that can occur, without needing specific stimulus to detect the bug. This ensures that the verified design is bug-free in all legal input scenarios. At the same time, this approach inherently identifies unreachable coverage points, which helps accelerate coverage closure.

Questa Formal Verification provides easy-to-use automatic checking for many common design errors. With Questa Formal Verification, designers can easily check out new code to look for functional issues such as floating or **multiply-driven buses**, combinational loops, arithmetic errors, and **initialization problems**. Finding and fixing these errors before integrating new code into the design avoids injecting difficult-to-find bugs into the larger system, and accelerates downstream verification. Since these checks are based on exact analysis of the reachable state space, the errors detected are real errors, not the noisy results that are often generated by simple **lint checkers**.

Questa Formal Verification also supports general **assertion-based formal verification** to ensure that the design meets its specific functional requirements. With support for PSL, SVA, and OVL, including multiclocked assertions, Questa Formal Verification easily verifies even very large designs with many assertions. Its multiple high-capacity formal engines cooperate to complete verification faster. Questa Formal Verification is integrated with the Questa Simulator for easy debug of assertion failures.

5.4.3 Questa CoverCheck

Questa CoverCheck reads code coverage results from simulation in the Unified Coverage Database (UCDB) and then leverages AutoCheck technology to do various useful verification tasks with regard to the coverage holes. The most obvious: prove that the code can be safely ignored. That is, the tool might mathematically prove that no stimulus could ever activate the code in question. In such cases, waivers are automatically generated to refine the code coverage results.

Secondly, the tool can also identify segments of code that, though difficult to reach, might someday be exercised in silicon. In such cases, CoverCheck helps point the way **to testbench enhancements** to better reach these parts of the design. Finally, CoverCheck flags code coverage items that are difficult to reach by formal techniques and haven't been hit in simulation, and thus provides a valuable measure of verification complexity.

Automates code coverage closure—achieve 100 % coverage with automatic formal reachability analysis.

Improved fidelity of code coverage results—eliminate code that is never meant to be exercised.

Mode-sensitive analysis—tune the code coverage reporting considering only the relevant modes of operation.

Guide testbench enhancement—waveforms show how uncovered but formally reachable coverage bins can be hit in simulation.

5.4.4 Questa CDC

It stands for Questa Clock-Domain Crossing (CDC) Verification. It Performs clock-domain crossing verification with Questa CDC is straightforward. The CDC compiler analyzes the RTL code, identifies all clocks and clock-domain crossings, and offers a rich, intuitive debugging environment to resolve all types of CDC issues. Once these issues are resolved, it automatically generates a set of protocol assertions and metastability models that are linked in to the simulation of the RTL code

Questa CDC addresses a number of critical verification issues that simply cannot be dealt with by simulation-based verification techniques.

An RTL or gate-level simulation of a design that has **multiple clock domains** does not accurately capture the timing related to the transfer of data between clock domains. As a consequence, simulation does not accurately predict silicon behavior, and critical bugs may escape the verification process. The Questa CDC Verification solution solves this problem. It is also used for metastability check.

5.4.5 Questa ADMS

It is used for Complex Analog/Mixed-Signal System-on-Chip Designs. Questa ADMS gives designers a comprehensive environment for verifying complex analog/mixed-signal System-on-Chip designs. ADMS combines four high-performance simulation engines in one efficient tool: Eldo® for general purpose analog simulations, Questa® for digital simulations, ADiT™ for fast transistor-level simulations and Eldo RF for modulated steady state simulation.

Universally accepts IP written in any of the standard design languages for easy migration. Builds on previous design investments through its design flow integration with Mentor Graphics Design Architect IC and Cadence Analog Design Environment.

ADMS integrates into the Cadence Virtuoso Analog Design Environment with the same look and feel as any simulator inside the environment, but gives designers the advantage of ADMS analysis, commands, and options. An enhanced symbol library providing specific Eldo devices is compatible with the Cadence library.

ADMS is the simulation engine underlying Mentor Graphics HyperLynx Analog for functional verification of complete printed circuit boards. A single schematic supports both PCB layout and functional analysis. HyperLynx Analog combines with HyperLynx Signal Integrity to extract parasitic PCB trace models for comprehensive board-level functional analysis.

5.4.6 Questa inFACT

It is an intelligent Testbench Automation. Recently announced, intelligent software-driven verification ("iSDV") has been added to the Questa inFact functionality to automatically generate embedded **C test programs** for both single-core and multi-core SoC design verification. iSDV bridges the gap between IP block and full system level verification by applying intelligent testbench automation to hardware/software verification at the system level. While writing directed tests in C to verify single-core designs at the system level was challenging, today's multi-core multi-threaded designs has made this process virtually impossible. Questa iSDV automates this process.

Questa® inFact is the industry's most advanced testbench automation solution. It targets as much functionality **as traditional constrained random testing**, but achieves coverage goals 10–100× faster.

5.4.7 Questa Power Aware Simulation

It verifies active power management. The **Questa® Power Aware Simulator** enables design teams to verify the architecture and behavior of active **power managemen**t planned for the implementation, but starting much earlier in the design process.

Verification of active power management at the **RTL** stage makes it possible to explore alternative power management approaches long before implementation begins, to achieve the greatest power reduction at the least cost.

Verification of active power management in the post-synthesis **Gate-Level** netlist stages makes it possible to ensure that synthesis and manual transformations have correctly preserved the active power management architecture and its behavior.

How Questa Power Aware Simulation Works

- Given a description of power intent expressed in the industry-standard Unified Power Format (UPF), the Questa Power Aware Simulator.
- Partitions the HDL design into power domains.
- Adds isolation, level-shifting, and retention cells.
- Integrates the power supply network into the design to power each domain
- The augmented HDL design can then be simulated with full control over the power state of each domain, for accurate modeling of the effects of active power management on the design's functionality.

5.4.8 Questa Verification IP

Verification IP (VIP) improves quality and reduces schedule times by building Mentor's protocol and methodology expertise into a library of reusable components that support many industry-standard interfaces. This frees up engineering resources from having to spend time developing BFMs, verification components, or VIP themselves, enabling them to focus on the unique and high-value aspects of their design.

VIP integrates seamlessly into advanced verification environments, **including testbenches built using UVM**, OVM, Verilog, VHDL, and SystemC. It is the industry's only VIP with a native SystemVerilog UVM and OVM architecture across all protocols, ensuring maximum productivity and flexibility. Transaction-level score boarding, analysis, and debug. Synthesizable memory models for use with simulation acceleration and emulation.

5.4.9 Questa Verification Management

Questa's verification management capabilities are **built upon the Unified Coverage Database** (UCDB). The UCDB captures any source of coverage data generated by verification tools and processes; Questa and ModelSim use this format natively to store code coverage, functionality coverage, and assertion data in all supported languages. It is used for Test-**plan tracking**.

Projects are tracked in spreadsheets or documents created by a range of applications, from Microsoft Excel and Word to OpenOffice Calc and Write. So it's critical that a verification management tool be open to a range of file formats, a basic feature of Questa, which is built on the premise that a user should be able to use any capture tool to record and manage the plan.

5.4.10 Questa CodeLink

It is software-driven hardware verification. Questa$^{®}$ Codelink is the industry's leading software-driven hardware verification solution. It makes every verification engineer an instant "CPU expert" by providing 100 % accurate processor views for system level testing.

Everything is fully synchronized and easily viewed, including logic simulation waveforms, **processor states**, source code, internal memory, registers, **stacks**, and output. Questa Codelink then presents only the important information needed to quickly debug software-driven tests.

As a result, companies using Questa Codelink have been able to **reduce their system level debugging time from months to days**. Complex simulation failures that used to require extensive analysis of multiple files and databases, can now be diagnosed within one robust multi-viewing debugging environment called Questa Codelink.

5.5 Summary

In this chapter, we introduce a deep introduction for Verilog for both implementation and verification point of view. The chapter used design examples for showing ways in which Verilog could be used in a design for both implementation and verification. This chapter did not cover all of Verilog, but only some important topics. Moreover, a survey on the current existing logic simulators is presented.

References

1. Mehler R (2015) Digital integrated circuit design using verilog and systemverilog. Elsevier, Oxford
2. Williams JM (2014) Digital VLSI design with verilog. Springer, Cham
3. Dubey R (2007) Introduction to embedded system design using field programmable gate arrays. Springer, London
4. Kilts S (2007) Advanced FPGA design architecture, implementation, and optimization. Wiley, Hoboken
5. Chu PP (2008) FPGA prototyping by verilog examples Xilinx SpartanTM-3 version. Wiley, Hoboken
6. Ciletti MD (2003) Advanced digital design with the verilog HDL. Prentice Hall, Upper Saddle River
7. Ciletti IMD (2003) Starter's guide to verilog 2001. Prentice Hall, Upper Saddle River
8. Ashenden PJ (2008) Digital design: an embedded systems approach using verilog. Morgan Kaufmann, Burlington
9. Lilja DJ, Sapatnekar SS (2005) Designing digital computer systems with verilog. Cambridge University Press, New York
10. Navabi Z (2005) Digital design and implementation with field programmable devices. Kluwer, Boston

11. http://www.verilogcourseteam.com
12. http://www.asic-world.com/verilog/pli2.html
13. Yuan J, Pixley C, Aziz A (2006) Constraint-based verification. Springer, New York
14. Qian H, Deng Y (2011) Accelerating RTL simulation with GPUs. In: 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), IEEE, San Jose, pp 687–693
15. http://www.mentor.com/products/fv/questa-verification-platform

Chapter 6
New Trends in SoC Verification: UVM, Bug Localization, Scan-C0068ain-Based Methodology, GA-Based Test Generation

6.1 Part I: UVM

6.1.1 Introduction

Now, SystemVerilog (SV)/UVM gradually dominate the verification landscape. SV does not support MACROS and the language alone was insufficient to enable widespread adoption of the best-practice verification techniques that inspired its development that is why we need UVM [1, 2]. UVM is a methodology for SoC functional verification that uses TLM standard for communication between blocks and SystemVerilog for its languages, or in other words, it uses SV for creating components and TLM for interconnects between components.

Methodology is a systematic way of doing things with a rich set of standard rules and guidelines. Methodology provides the necessary infrastructure to build a robust, reliable, and complete verification environment. Methodology shrinks verification efforts with its predefined libraries. It makes life easier by preventing the designer from making mistakes or poor decisions. It also helps make sure that whatever you do will mesh nicely with what others do (reusability). Methodology is basically a set of base class library which we can use to build our testbenches.

UVM main goals are: reusability to reduce time to market and it is targeted to verify systems from small to large concept (Fig. 6.1), speed verification: it helps the designers to find more bugs earlier in the design process, so it provides practical and efficient SoC verification flow by reusing IP testcase and testbench, standardization: vendor independent, dynamic not static like traditional testing (Table 6.1), randomization, and automation [3, 4]. UVM makes multi-master multi-slave systems verification easier as it separates tests from testbench.

Table 6.2 summarizes the companies, simulators, and versions related to UVM [5, 6], it is noted that UVM is supported by all major simulator vendors, which is not the case with earlier methodologies [7]. Various IPs are connected to and controlled through a bus, so the functional verification uses BFM (bus functional model).

K.S. Mohamed, *IP Cores Design from Specifications to Production*,
Analog Circuits and Signal Processing, DOI 10.1007/978-3-319-22035-2_6

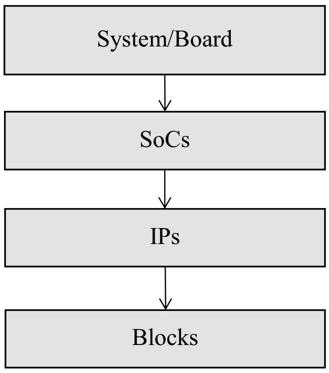


Fig. 6.1 Levels of verification: UVM verifies systems from small to large concept. SoC is a collection of IPs

Table 6.1 Comparison between UVM and traditional testing

	Traditional testing	UVM
Stimulus structure	Procedural code	Constrained random variable
Type	Static	Dynamic
Reusability	Nonreusable	Reusable (customization)
Scalability	Nonscalable	Scalable
Test redundancy	None	Yes
Simulation overhead	None	10~40 % to solve constraints
Controllability	Coarse-grained	Fine-grained (smoother)
Observability	lower	Higher (assertion, coverage)
Maintainability	Hard	Easy

Table 6.2 Companies, simulators, and versions related to UVM

Companies	$\text{UVM} = \begin{cases} \text{OVM, AVM (Mentor)} \\ \text{URM (Cadence)} \\ \text{VMM (synopsys)} \end{cases}$
Simulators	UVM supports all simulators {Questa, IUS, and VCS}
Releases	$\text{UVM} = \begin{cases} \text{UVM1.0} \rightarrow \text{Released} \\ \text{UVM1.1(a,b,c,d)} \rightarrow \text{Released} \\ \text{UVM1.2} \rightarrow \text{Released} \end{cases}$

The rest of this chapter is organized as follows. In Sect. 6.1.2, SystemVerilog features are proposed. In Sect. 6.1.3, TLM features are proposed. In Sect. 6.1.4, UVM features are introduced. Summary is given in Sect. 6.1.5.

6.1.2 *SystemVerilog*

Initially, Verilog is used for verification. But, for complex design, developing a verification environment in Verilog is tedious process and consumes a lot of time. So, SystemVerilog is used to create verification environment which reduces effort to develop testbench. SystemVerilog is an extensive set of enhancements to Verilog and it is called hardware description verification language (HDVL), the important features of it are summarized in Fig. 6.2. SystemVerilog supports constrained random stimulus generation and coverage analysis, and object-oriented programming (OOP) structure which contributes to transaction-level verification and providing the reusability of verification. Object-oriented programming can greatly enhance the reusability of testbench components [8–11]. It has **C-like** control constructs such as foreach, and **VHDL-like** package and import features. In this section, we discuss the main features of SV, where OOP is introduced in Sect. 6.1.2.1, easy call of C programs (direct programming interface) is introduced in Sect. 6.1.2.2, constrained randomization is introduced in Sect. 6.1.2.3, functional coverage is introduced in Sect. 6.1.2.4, assertion is introduced in Sect. 6.1.2.5, other constructs such as: interface, modport, clocking, fork_join, and always are introduced in Sect. 6.1.2.6, and new data types are introduced in Sect. 6.1.2.7.

6.1.2.1 Object-Oriented Programming

Object-oriented programming is used for code reusability (inheritance), where object = entity (hold the data) + method (operate on the data). It is packing data and function in one structure, moving functions inside data structure is for consistency. Comparison between instantiation of class in SystemVerilog and instantiation of module in Verilog is shown in Table 6.3. Moreover, comparison between procedural code and OOP is shown in Table 6.4. The main features of OOP are summarized in Table 6.5. The OOP in SV has some restrictions as it supports only single inheritance [11].

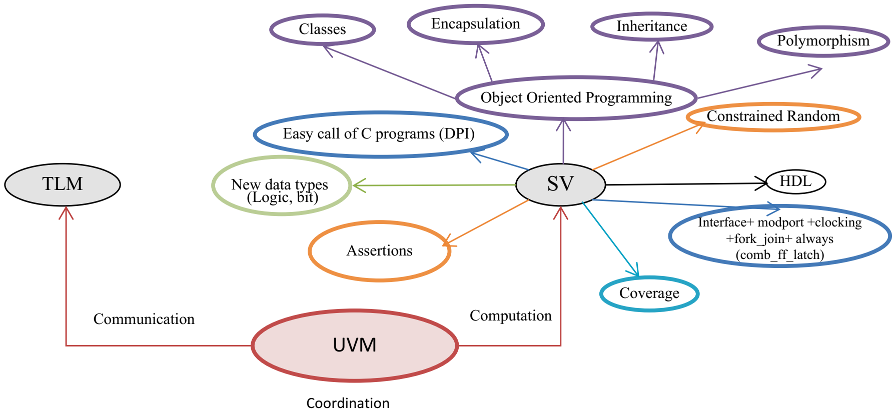

Classes
Encapsulation
Inheritance
Polymorphism
Object Oriented Programming
Constrained Random
Easy call of C programs (DPI)
TLM
New data types (Logic, bit)
SV
HDL
Interface+ modport +clocking +fork_join+ always (comb_ff_latch)
Assertions
Coverage
Communication
Computation
UVM
Coordination

Fig. 6.2 UVM consists of TLM and SV

Table 6.3 Comparison between instantiation of class in SystemVerilog and instantiation of module in verilog

Instantiation of class in SystemVerilog	Instantiation of module in Verilog
Dynamic @ run time, parameterized class	Static

Table 6.4 Comparison between procedural code and OOP by example

Procedural code	OOP

Struct driver { wire A,B}
Void init {};
Void send_data {};
Begin
Driver driver;
Init ();
Send_data ();
end
	Struct driver{
	wire A,B;
	void init {};
	Void send_data {};}
	Begin
	Driver driver;
	Driver.Init ();
	Driver.Send_data ();
	end

Table 6.5 Main features of OOP

Class	Defines set of properties and behavior of object, and it is a data type
Object	Is an instant of the class and defined inside program/module
Inheritance	"**Extends**" for code reusability
Encapsulation	Bind data and method together for consistency
Polymorphism	It means to have many forms. Bind data and method at run time. "**Virtual**" keyword

6.1.2.2 Easy Call of C Programs (Direct Programming Interface)

In Verilog, calling C programs is called PLI and it is complicated, In SV it is called direct programming interface (DPI) and it makes C program calls easier [11]. SV functions can be called in C using export and C functions can be called in SV using import.

6.1.2.3 Constrained Randomization

Constrained random verification applies stimuli to the device under test (DUT) that are solutions of constraints. These solutions are determined by a constraint solver. Thereby, the generated stimulus is much more likely to hit corner cases which make discovering unexpected bugs easier. Randomizing the stimulus also makes reaching the verification coverage easier. We put some constraints on that stimulus in order to generate legal or interesting scenarios. Make sure that there is no conflict or contradict between constraints. Constraints are like control knobs. **Weighted constraints** are very important to hit boundary values. In a nutshell, constrained random should be an intelligent process. You can disable constrains using **constrain_mode** (0) method.

6.1.2.4 Functional Coverage

Functional coverage is a user-defined metric that measures how many percentages of the verification objectives are met by the testplan [2]. Quality of verification depends upon the quality of testplan. Actually, coverage answers the question "did we do enough randomization?" For coverage closure, we may need to write direct testing, enhance stimulus generator, or randomize seeds {**vsim –sv_seed**}.

6.1.2.5 Assertion

Assertion acts as constraints that determine and define legal and expected behavior when blocks interact with each other [2]. Complex protocol checks are often implemented using SystemVerilog Assertions. Assertions could be tool independent: used with both static and dynamic tools. SV has two types of assertions: immediate (clock-independent) and concurrent (clock-dependent) [9]. Assertion improves observability and debug ability.

6.1.2.6 Other Constructs: Interface + Modport + Clocking + Fork_Join (Any None) + Always (comb_ff_latch)

One of the problems of direct DUT signal access is that driver and monitor are dependent on signal name of DUT, and duplicate efforts. So, using interface as a signal-map makes it easy to add or remove wire, reduce errors which occur during model connections, remove redundancy in wires (Fig. 6.3). Modport: for direction which is input/output/inout. Clocking block is highly recommended usage in test-bench to avoid race conditions. Fork-join acts like simply begin–end but inside fork-join all statements are taken as concurrent. Classic fork-join is a "join all" construct. That's if you fork two threads, then both of them need to finish for the join to end. With join_none, one can spawn threads and continue, this is useful in launching multiple input data streams for example.

To assist synthesis, there are some extra keywords. The always_comb, always_latch, and always_ff keywords identify the intent of the process, so that a synthesis tool can detect user errors [6], i.e., the synthesis compiler can tell us when we have the wrong type of logic in our RTL models.

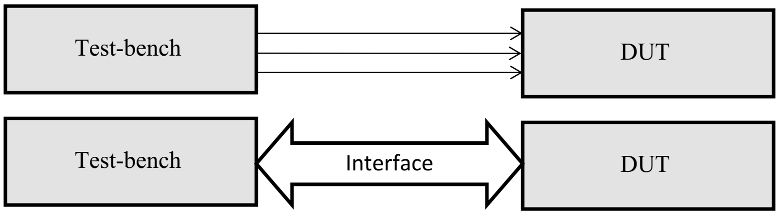


Fig. 6.3 Interface versus conventional connections

6.1.2.7 New Data Types

Bit (2-valued) and logic (4-valued) are new data types introduced by SV to allow continuous assignments to logic variables. Using a 2-valued data type will speed up simulation of the code. We no longer need to worry about when to declare module ports or local signals as wire or reg. With SV, we can declare all module ports and local signals as logic, and software tools will correctly infer nets or variables for you [10]. SV also offers dynamic and associative array and queue.

6.1.3 TLM

Transaction-Level Modeling (TLM) provides abstraction level description for the IP which means lack of details (Fig. 6.4). Advantages: simulation speed increases, observation of traffic is easier, debugging on TLM level is easier than debugging on RTL. Disadvantages: accuracy decreases. TLM separates communication from computation and it is unidirectional put/get interface that works as a bridge to enable UVM verifies multilanguages like SystemC. TLM is a library built on top of SystemC which itself is a class library of C++. It encapsulates the communication between different modules to separate communication from computation. Translation of TLM2.0 from SystemC to SystemVerilog is needed, because it is written at the beginning in SystemC. Connect () method using TLM analysis port is the most famous method for TLM in UVM. We have three types for TLM communications: port, export, and analysis_port.

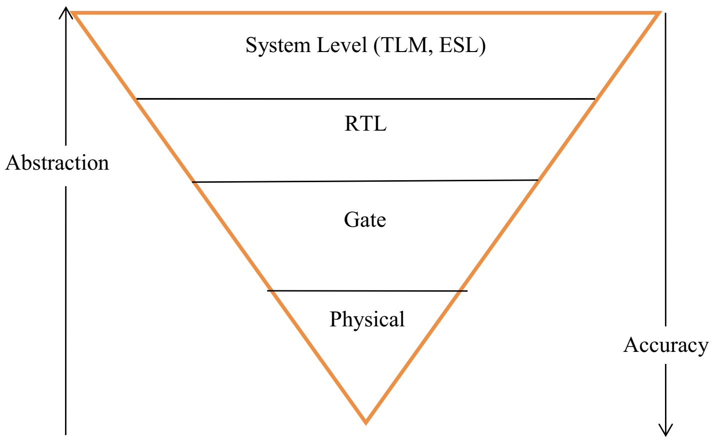


Fig. 6.4 Abstraction level versus accuracy, ESL is electronic system level

6.1.4 UVM

In this section we discuss the main features of UVM, where UVM infrastructure is introduced in Sect. 6.1.2.1, Steps to verify an IP using UVM is introduced in Sect. 6.1.2.2, and Drawbacks of UVM is introduced in Sect. 6.1.2.3, Opportunities for UVM are discussed in Sect. 6.1.2.4. A case study is introduced in Sect. 6.1.2.5.

6.1.4.1 UVM Infrastructure

UVM testbench is composed of reusable verification component, which consists of a complete set of elements for stimulating, checking, and collecting coverage information for a specific protocol or design. These verification components are applied to the DUT to verify it [12]. The testbench should be layered to break the problem into manageable pieces to help in controlling the complexity.

The UVM main infrastructure, components, and all the terminology related to UVM is introduced and summarized in Table 6.6, and the general architecture is shown in Fig. 6.5. Master sequencer generates the data and it is sent to the DUT through the driver. The data received by the slave are feed back to the scoreboard via collector for comparison then here the sent and received data item are compared in the scoreboard. The monitor samples the stimulus and responses. The configuration parameters are used to configure these components. All these components can be reused. The driver, monitor, and responder are called **transactors**/translators/adaptors.

Table 6.6 UVM infrastructure description

Component	Description
Interface	For communication between classes and modules
Transaction	Representation of arbitrary activity in a device which has attributes and bounded by time
Driver = BFM	Apply stimulus to DUT (protocol specific). Also, Convert TLM to RTL (pin wiggles). BFM = bus functional model
	Think in the driver as a **normal testbench**
Monitor	Monitor traffic, collect coverage, and send them to the various analysis ports such as coverage and scoreboard
	It looks like duplication of **driver**, but without triggering DUT wires (passive)
Collector = receiver = responder	Detects signal level activity, convert RTL to TLM and send it to monitor
Sequencer = producer = generator	Execution of traffic, coordinate what to do. Running different streams without the need to change the component instantiation. It is like **arbitration** logic
Sequences = **scenarios**	Generate stimulus. Protocol dependent and consists of multiple of sequence items. It is generated from test class
Sequence item	Low level representation like address, data. A transaction object from the sequencer that stimulates the driver

(continued)

Table 6.6 (continued)

Component	Description
Virtual interface	Inside driver to connect to RTL, like pointer to enable configuration at runtime. It is a reference to the actual interface
Sequence library	Different sequences used by sequencer
TLM port	To connect between sequencer and driver
Agent = component = module = UVC	Instantiate, configure subcomponents like {driver, sequencer, monitor, collector}. Agent for TX, agent for RX
Agent type	Tx, Rx, Master, Slave, Arbiter
Virtual sequencer	Coordinate traffic between different UVCs, does not have a sequence item. It is protocol independent. It starts sequences on sequencer. Virtual sequences mean that sequences are calling other sequences
Scoreboard/checker	Self-checking mechanism. Check that the design is doing what we expect. Need abstract reference model which can be MATLAB or Python. **Golden** model and RTL must be developed by different teams, errors might be in both. Compare (received, expected). It is a TLM-based checking. It is preferred to separate protocol checking from data checking for reusability. We can build the **assertions** inside the scoreboard. Scoreboard checks that if the DUT and the reference model have the same stimulus, they should have the same response
Functional coverage	For completeness as it measures important behavior, covers operation, dimension (as buffer size). Did we exercise the whole testplan? To stress the device if not. We need to know what all the tests have accomplished and this is done by storing the data in a database and merging it all together. So, basically we should implement a regression environment for functional coverage measurement. Regressions are the continuous running of the tests preciously defined in the testplan [13]
	Illegal bins should be analyzed to check if any test case is out of the design specifications
Code coverage	Did we exercise the whole code?
Testbench	Contains all subcomponents, connections
Test	Call testbench, configure traffic, and can be {directed, random constrained, intelligent: driven random constrained to remove redundancy}. Coverage-driven testing ->continue randomization until coverage = 100 %
Configuration	To change the behavior of an already instantiated component to provide flexibility. Such as #slaves, #masters
	It provides configuration information to all parts of TB. Configuration database is like parameters in Verilog
Factory	For class override at runtime, this helps making modifications to an existing testbench. Create () method
Phases	Synchronization of UVM components. UVM components have different phases that operate in a particular sequence:
	Build (new ())->connect (TLM 2.0)->end of elaboration (Config) ->strt_sim->run->extract->check->report
	*elaboration = @compile time
	*on the fly = @ run time
	Build and connect are **functions** as they consume zero time. Run is **task** as it consumes some time

(continued)

Table 6.6 (continued)

Component	Description
Class library	The UVM comes with a bunch of classes which are used to implement the verification environment
	UVM packages:
	1. UVM_components (structural)
	2. UVM_objects (configuration)
	3. UVM_transaction (stimulus)
Objections	Any component that is busy should raise an objection to ending the test, and then drop the objection when it is finished
	For example, you can raise objection until coverage is 100 % (get_coverage ( )) and then drop the objection
UVM register layer	Mechanism to setup and access DUT internal registers and memory. It extends from UVM_reg. **IP-XACT** format is very useful for this feature.
Verification Plan	It is a roadmap that summarizes test function points according to IP specification (Failing to plan = planning to fail). It should be smart testplan which effectively and efficiently tests the DUT
Macros	Macro is a construct that enables user to extend the language. Macros implement some useful methods in classes as it can be used for shorthand notation of complex implement. They are optional, but recommended. The most common ones are:
	'**UVM_component_utils**—This macro registers the new class type. It's usually used when deriving new classes like a new agent, driver, monitor, and so on
	'**UVM_object_utils**—This macro registers the objects like sequences
	'**UVM_field_int**—This macro registers a variable in the UVM factory and implements some functions like copy (), compare (), and print ()
	'**UVM_object_param_utils**—This macro registers the parameterized objects
	'**UVM_component_param_utils**—This macro registers the parameterized components
	'**UVM_info**—This is a very useful macro to print messages from the UVM environment during simulation time
	'**UVM_fatal**—This is a very useful macro to print fatal error messages from the UVM environment during simulation time
	'**UVM_error**—This is a very useful macro to print error messages from the UVM environment during simulation time
	'**UVM_warning**—This is a very useful macro to print warning messages from the UVM environment during simulation time
+Plusarg = command line processing	Some of the famous UVM + plusarg are:
	+UVM_TESTNAME
	+UVM_VERBOSITY
	+UVM_TIMEOUT

The UVM library defines a set of base classes and utilities that facilitate the design of scalable, reusable verification environments as depicted in Fig. 6.6. The basic building blocks for all environments are components and the transactions they use to communicate which are called objects [7, 12, 14].

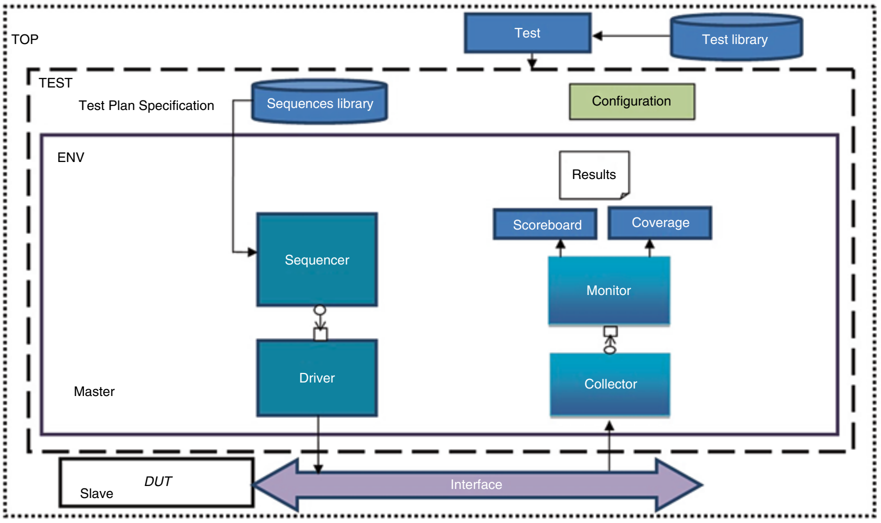


Fig. 6.5 UVM architecture and skeleton: the big picture

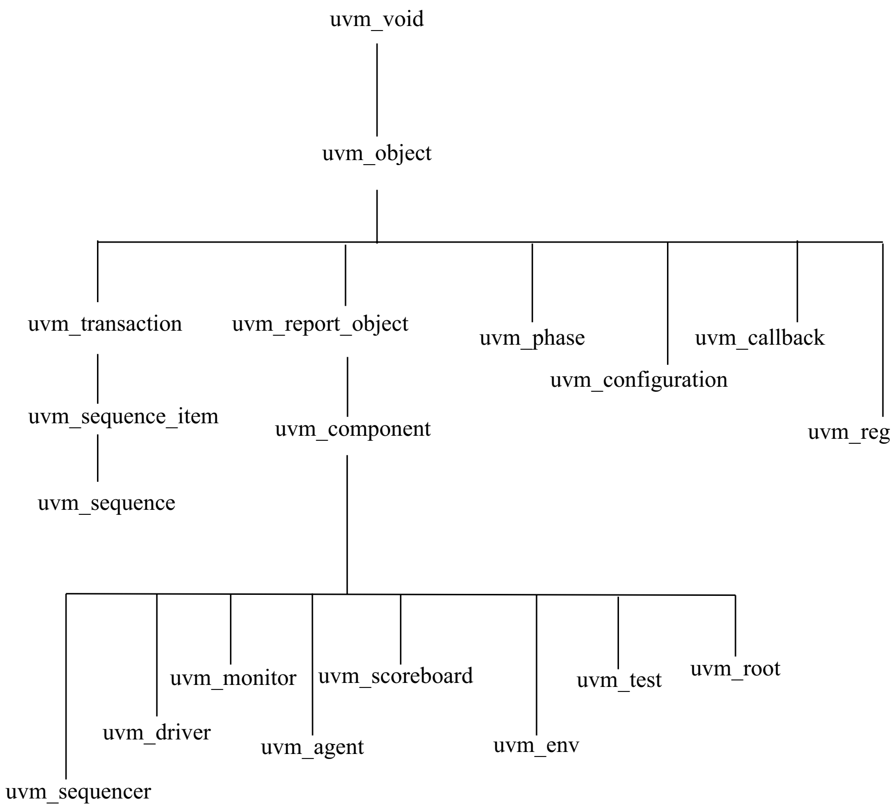


Fig. 6.6 Partial UVM class tree (UVM_pkg), we can inherit from any class

UVM_Void

The UVM_void class is the base class for all UVM classes. It is an abstract class with no data members or functions. It allows for generic containers of objects to be created. It works similar to a void pointer in the C programming language.

UVM_Object

All components and transactions derive from UVM_object, which defines an interface of core class-based operations: create, copy, compare, print, and record. It also defines interfaces for instance identification (name, type name, unique id, etc.) and the random seeding.

UVM_Component

The UVM_component class is the root base class for all UVM components. Components are objects that exist throughout simulation. Every component is uniquely addressable using hierarchical path name.

UVM_Transaction

The UVM_transaction is the root base class for UVM transactions. It extends UVM_object to include timing and recording interface. Simple transactions can derive directly from UVM_transaction.

UVM_Root

The UVM_root class is special UVM_component that serves as the top level component for all UVM components, provides phasing control for all UVM components, and other global services. UVM_TOP is a singleton of it.

UVM_Callback

The UVM_callback class is the base class for user-defined callback classes. We define an application-specific callback class that extends from this class. In that, we will define one or more virtual methods, called a callback interface that represent the hooks available for user override.

6.1.4.2 Steps to Verify an IP Smartly Using UVM

The steps to verify an IP smartly using UVM can be summarized as follows:

1. Understand the specification: implement the DUT.
2. Prepare verification plan: feature extraction, specifies how design will be verified, constrained random coverage driven, written in excel sheet, link it to coverpoint in coverage code written in SystemVerilog. You should expose your DUT to stress testing.
3. Build verification environment in the following order: interface, configuration, scoreboard, and monitors, generate sequences based on verification plan, Env Class + simple testcase and simulate it. Debug from the generated UVM report summary.
4. Measuring coverage progress against the testplan, run regressions, and add testcases for coverage holes. For closing **coverage** you start to run with multiple seeds, but sometimes certain scenarios can never be covered by the randomness and we need a directed test case.
5. Error handling and debugging: when you find a bug, before debugging it ask yourself the following questions: Is this mistake somewhere else also? What next bug is hidden behind this one? What should I do to prevent bugs like this? Then, you can start debugging using waveforms, tracing, or logging. Use built-in **watchdog** timer class to handle testcase hanging.
6. When all tests in the testplan have been tested and no bugs were found, then the verification task is over.

6.1.4.3 Drawbacks of UVM

Synthesis tool for SV is limited. This is a major drawback which is restricting designers to accept SV as a design language. Also, there are limitations for using UVM with emulators. Moreover, UVM is very complicated, so it does not make sense with small projects. Besides, there are challenges in using UVM at SoC Level. Also, debugging Macros is difficult. UVM provides no links between testbenches and code running in the embedded processors.

6.1.4.4 Opportunities for UVM

UVM methodology can be enhanced to offer a flexible framework for the virtual prototyping of multidiscipline testbenches that supports both digital and Analog Mixed-Signal (AMS) at the architectural level [15]. The extension of UVM for mixed-signal verification of analog models is reported in literature [15]. Moreover, UVM is a promising solution in verifying 3D-SoC which has many IPs and heterogeneous systems.

Table 6.7 Comparison between direct testing and UVM

WISHBONE metric	Direct testing	UVM
# Tests to reach 100 % coverage	30	120
Regression time (h)	3	0.25
Benefits	–	12× faster

6.1.4.5 A Case Study: WISHBONE

A SoC case study is presented to illustrate the pros and cons of the UVM and to compare traditional verification with UVM-based verification. WISHBONE SoC interconnect architecture for portable IP cores are used as a case study [16]. The results can be shown in Table 6.7, where the UVM-based approach improves the coverage time by 12 times.

6.1.5 Summary

This chapter presents an overview on building a reusable RTL verification environment using the UVM verification methodology. UVM is a culmination of well-known ideas and best practices. This chapter also presents a survey on the features of UVM. It presents its pros, cons, challenges, and opportunities. Moreover, it presents simple steps to verify an IP and build an efficient and smart verification environment. A SoC case study was presented to compare traditional verification with UVM-based verification.

6.2 Part II: RTL Bug Localization

6.2.1 Introduction

In VLSI, design flow functional verification is a required process to ensure that the implementation of the design is in accordance with the specification. Due to the increasing design complexity of VLSI circuits, the cost of verification and debugging has significantly increased.

According to ITRS [17], Verification process is now considered a bottleneck as it consumes up to 60 % of the design cost.

Verification tools check the correctness of a design against its specification. Register Transfer Level (RTL) is still the dominant description level for the hardware design.

There are two types of bugs: (1) electrical bugs caused by interaction between the design and physical effects such as cross-talk, supply noise, temperature, process variation, and signal integrity. (2) Design or functional bugs at RTL which are classified into three major classes: logic bugs, algorithmic bugs, and timing/synchronization bugs [18].

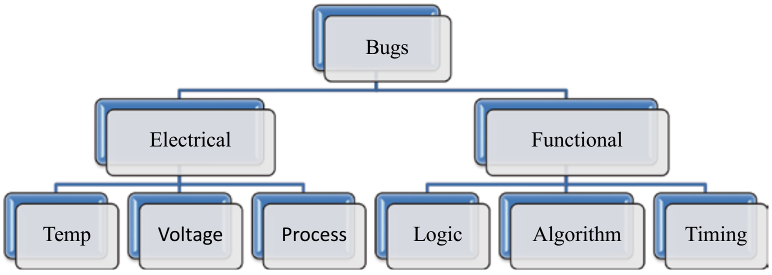


Fig. 6.7 There are two types of bugs: (1) electrical bugs caused by interaction between the design and physical effects such as cross-talk, supply noise, temperature, process variation, and signal integrity. (2) Design or functional bugs at RTL which are classified into three major classes: logic bugs, algorithmic bugs, and timing/synchronization bugs

The logic bug is characterized by erroneous logic in combinational circuits. A logic bug occurs because the designer formed an erroneous logic block.

The algorithmic bug covers major design bugs related to the algorithmic implementation of the design. These design bugs exhibit algorithmic deviations from the design specification and they usually require major modifications to be fixed.

The timing bug is associated with the timing correctness of the implementation, where a signal needed to be latched a cycle earlier or a cycle later in order to keep the timing of signals correct in the design. These types of bugs are summarized in Fig. 6.7.

In order to keep the production costs low, it is required to detect bugs as soon as possible. This chapter targets localization of functional errors.

While there are a lot of verification methodologies for error detection in RTL design, there is fewer work for debugging the error which includes localization and correction stages. Moreover, most of the related works are concentrating on gate-level error localization [19–21], and are applied to small designs.

For gate-level bug localization, there are basically two approaches: symbolic and simulation-based. Symbolic approaches are accurate but suffer from combinatorial explosion, whereas simulation-based approaches, although scalable with design size, require numerous test vectors for sufficient accuracy. A SAT-based automated bug localization is used for gate-level [22, 23].

Other work is focusing on formal methods and failed properties which are not suitable for large designs [24, 25].

Here, we are focusing on the RTL-level and large designs. Detecting and locating the source of erroneous behavior in large and complex RTL design is challenging. In this chapter, we present a novel approach for bug localization methodology to address this challenge using information from regression suit results about failed and passed testcases and number of statements executed by each test. The idea is inherited from software domain [26–28]. We present a proof of concept for this idea using Verilog-based case studies.

This chapter is organized as follows: In Sect. 6.2.2, the proposed methodology for bug localization error is presented and discussed. Moreover, the experimental results are analyzed. In Sect. 6.2.3, summary is given.

6.2.2 RTL Bug Localization

In this section, proposed methodology is given in Sect. 6.2.2.1. Results are discussed in Sect. 6.2.2.2.

6.2.2.1 Proposed Methodology

Given a set of statements (S) for which an HDL design exhibits an incorrect behavior, the objective of design debugging is to find the highly candidate statement that may be responsible for this incorrect behavior. The failing and passing testcases are used to find the bug location. If a statement is executed by more than two failing testcases, so this statement is more likely to have the bug. So, run the complete regression suite until the coverage is 100 %, then extract the needed information about the passed and failed testcases and obtain a list of design statements executed by each test.

An example to show how our proposed method works is shown in Fig. 6.8, where we assume that our DUT has only one bug due to only one incorrect statement and we have ten testcases to test its behavior.

From Table 6.8, the left columns shows how each RTL statement is executed by each testcase either it is failing or passing. An entry 1 indicates that the statement is executed by the corresponding test case and an entry 0 means it is not executed. The most right column shows the execution result with an entry 1 for a failed testcase and an entry 0 for a passing testcase. If a statement is executed by a successful test case, its likelihood of containing a bug is reduced.

The suspiciousness of each statement = the number of failed tests that execute it—the number of successful tests that execute it. But, this way cannot distinguish a statement executed by one successful and one failed test from another statement executed by 10 successful and 10 failed tests.

So, we will use weighted probability to indicate that more successful executions imply less likely to contain the bug. So the suspiciousness of each statement = the number of failed tests that execute it/the number of successful tests. And we will choose the maximum value to start with, i.e., the large rank. The proposed methodology for automated bug localization is shown in Fig. 6.9.

6.2.2.2 Experimental Results

Experimental results show that our method can detect errors in large designs up to several thousand lines of RTL code in few minutes with high accuracy compared to time consumed in hours using manual bug localization. Here, we only localize the error not correcting it. Other experiments are done on more bugs to observe the effectiveness of our methodology. We insert errors into some other parts of the code for the complex RTL design then we applied our methodology to locate the error. Table 6.9 reveals some results, where it is clear that our methodology reduces the time needed to localize the bug significantly.

Fig. 6.8 A case study: a behavior Verilog code for a part of complex design contains a bug in s10. The design is more than 5000 lines of RTL code

```
/*************************************************
 * Verilog code for a part of design contains a bug *
 *************************************************/
 always @ (negedge CLK or negedge RST_N)
 begin
  if (~RST_N)                          //s0
    begin
        rd_cnt <= 16'h0000;            //s1
       cnt8_1 <= 3'b000;               //s2
   end
  else if (~incr_rd_user_addr)         //s3
  begin
        rd_cnt <= 16'h0000;            //s4
        cnt8_1 <= 3'b000;              //s5
  end
  else if (incr_rd_user_addr)          //s6
  begin
     cnt8_1 <= cnt8_1 + 3'b001;        //s7
  if (BUS_WIDTH == 3'h2)               //s8
   begin
     cnt8_1 <= 3'h0;                   //s9
    rd_cnt <= rd_cnt + 2;              //s10
   end
  else if (BUS_WIDTH == 3'h0)          //s11
    begin
           cnt8_1 <= 3'h0;             //s12
            rd_cnt <= rd_cnt + 1;      //s13
      end
  end
end
```

Table 6.8 A motivational example to describe the proposed methodology

	Statements															Test status (T)
Test suite	S0	S1	S2	S3	S4	S5	S6	S6	S7	S8	S9	S10	S11	S12	S13	P(0)/F(1)
Test 1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
Test 2	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0
Test 3	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0
Test 4	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0
Test 5	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0
Test 6	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1
Test 7	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1
Test 8	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0
Test 9	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1
Test 10	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Failed tests per statements	4	4	4	4	4	4	4	4	4	4	4	4	1	1	1	
Passed tests per statements	6	6	6	6	6	6	5	5	3	0	0	0	5	5	3	
Suspicious per statement	−2	−2	−2	−2	−2	−2	−1	−1	1	4	4	4	−4	−4	−2	
Weighted Suspicious per statement	0.6	0.6	0.6	0.6	0.6	0.6	0.8	0.8	1.3	Inf	Inf	Inf	0.2	0.2	0.3	

Multiply each statement (S) column with the (T) column to obtain the number of failed tests that execute each statement. Multiply statement (S) which contains ones with the (T) column which contains zeros to obtain the number of successful tests that execute each statement. the suspiciousness of each statement = the number of failed tests that execute it/the number of successful tests. “0” Between test number and Sx means statement not executed

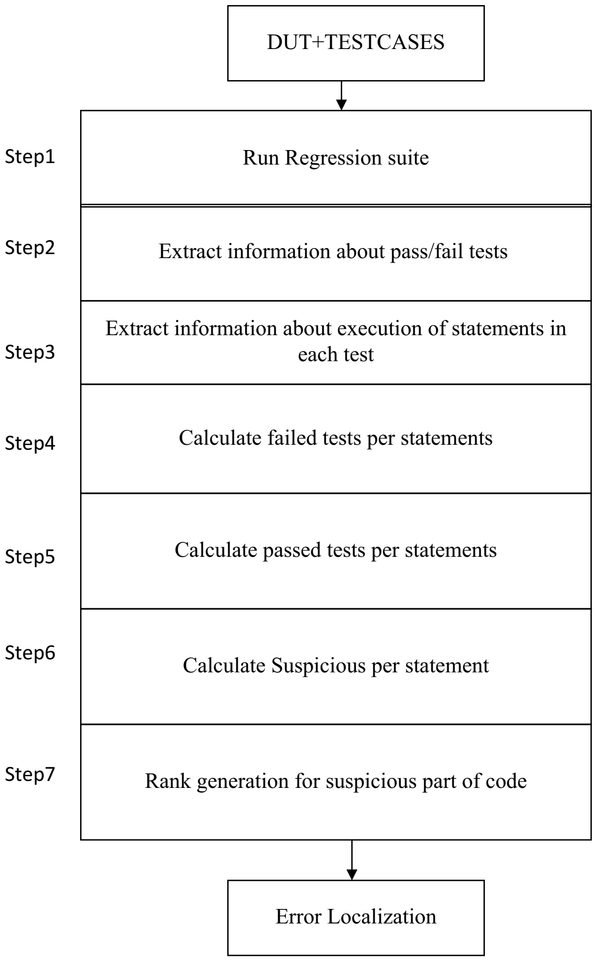


Fig. 6.9 The proposed methodology for automated bug localization
DUT+TESTCASES
Step1
Run Regression suite
Step2
Extract information about pass/fail tests
Step3
Extract information about execution of statements in each test
Step4
Calculate failed tests per statements
Step5
Calculate passed tests per statements
Step6
Calculate Suspicious per statement
Step7
Rank generation for suspicious part of code
Error Localization

If the testcase fails in the regression although it passes alone, we should merge it with the previous testcase to create only one testcase as the previous testcase does not reset a certain variable which caused the followed testcase to fail. The effectiveness of this methodology varies for different designs, bugs, and testcases. Here, we assume that we have a rich and correct testcases.

6.2.3 Summary

Bug localization is a process of identifying the specific locations or regions of source code that is buggy and needs to be modified to repair the defect. Bug localization can significantly reduce human effort and design cost.

Table 6.9 Bug localization time using the proposed methodology versus manual debug in a complex design, which contains more than 5000 lines of RTL code

	Time			
Bug ID	The proposed methodology (min)	Manual debug (h)	Wrong behavior	Correct behavior
Bug 1	3	1	x1 = x2 + x3 + x4 + x5	x1 = x2−x3 + x4 + x5
Bug 2	4	2	If (y1)	If (~y1)
Bug 3	7	2	If (~y2)	If (y2)
Bug 4	5	2	cnt = cnt + 2;	cnt = cnt + 3;
Bug 5	2	1	if (~btst_card_en & ~ strt_cmd_data_dly)	if (~btst_card_en & strt_cmd_data_dly)
Bug 6	3	3	TST_DATA <= 8'h00;	TST_DATA <= 8'h01;
Bug 7	10	2	MFSM_BUS <=MFSM_BUS_REG;	MFSM_BUS <=MFSM_BUS_REG/2;
Bug 8	6	2	if (CMD6_ARG [31] ==1)	if (CMD6_ARG [31] ==0)
Bug 9	7	1	current_state <= Data	current_state <= Rcv;
Bug 10	4	3	MFSM_OUT_ENABLE <= 4'hf;	MFSM_OUT_ENABLE <= 4'he;
Bug 11	2	1	If (OP_MODE ==2)	If (OP_MODE ==1)
Bug 12	9	1	MFSM_STRT_DATA_P2S <= 1'b1;	MFSM_STRT_DATA_P2S <= 1'b0;
Bug 13	4	2	bus_width_prev <= MFSM _WIDTH;	bus_width_prev <= MFSM _WIDTH/2;
Bug 14	5	1	MFSM_BUS_WIDTH <= 4'h0;	MFSM_BUS_WIDTH <= 4'h1;
Bug 15	2	1	MFSM_LEN <= 32'h0;	MFSM_LEN <= 32'h200;
Bug 16	3	1	strt_cmd_data_dly <= 1'b0;	strt_cmd_data_dly <= 1'b1;

(continued)

Table 6.9 (continued)

	Time			
Bug ID	The proposed methodology (min)	Manual debug (h)	Wrong behavior	Correct behavior
Bug 17	2	2	If ((1'b1 <<WRITE_BL_LEN) + 1'b1))	If ((1'b1 <<WRITE_BL_LEN)- 1'b1))
Bug 18	2	1	If ((blk_dis == 1'h1)	If ((blk_dis == 1'h0)
Bug 19	5	1	crc_dis <=(cnt_crc==16-NC)? 1'h1:1'h1;	crc_dis <=(cnt_crc==16-NC)? 1'h0:1'h1;
Bug 20	3	2	if (blk_no1 != blk_count)	if (blk_no1 == blk_count)
Bug 21	2	1	W_OR_R <= 0 ;	W_OR_R <= 1 ;
Bug 22	1	1	If (blk_len_cmd16 < blk_len)	If (blk_len_cmd16 > blk_len)
Bug 23	3	2	cnt4 <= cnt4 + 1;	cnt4 <= cnt4–1;
Bug 24	1	3	else if (~incr_rd_user_addr)	else if (incr_rd_user_addr)
Bug 25	5	1	Else (WRITE_BLK_MISALIGN)	Else (~ WRITE_BLK_MISALIGN)
Bug 26	1	1	erase_start_addr <(ERASE_SIZE)*512	erase_start_addr <(ERASE_SIZE + 1)*512
Bug 27	2	2	data_cnt_cmd25 <= 32'h0;	data_cnt_cmd25 <= 32'h1;
Bug 28	4	1	data_cnt_cmd25_en <= 1'b0;	data_cnt_cmd25_en <= 1'b1;
Bug 29	1	2	TST_DATA <= 8'h00;	TST_DATA <= 8'h80;

In this chapter, a novel automated coverage-based functional bug localization method for complex HDL designs is proposed which significantly reduces debugging time. The proposed bug localization methodology takes information from regression suite as an input and produces a ranked list of suspicious part of code. Our methodology is a promising solution to reduce required time to localize RTL bugs significantly.

6.3 Part III: RTL Scan-Chain

6.3.1 Introduction

Simulation-based verification scheme of large sophisticated intellectual property (IPs) is considered a time consuming process. Mainly, there are two famous methods to help accelerate simulation process and reduce verification time: hardware acceleration, and hardware RTL emulation. The RTL hardware accelerator solutions are based on using application-specific ASICs, each contains special-application processors and memories [29–32]. The RTL hardware emulators are based on using FPGAs, where the design is synthesized into a gate-level netlist. However, most hardware emulator does not provide easy debugging capability at runtime. In this chapter, a scan-chain scheme is proposed to reduce debugging time. Runtime changes of the values of the signals of the IP during execution-time can be done through the proposed scan-chain methodology.

The proposed method provides internal glue-block which automatically replaces any signal with a mux and extra input, so that at run time if we enable this method we can replace any internal signal by a forced one.

The rest of this chapter is organized as follows. In Sect. 6.3.2, the proposed RTL-level scan-chain methodology is presented. Summary is given in Sect. 6.3.3.

6.3.2 The Proposed RTL-Level Scan-Chain Methodology

RTL simulation provides system-on-chip (SoC) verification with full debugging capabilities, but its disadvantages are the low-speed simulation for complicated RTL design. By using FPGA-based RTL emulation, we can have high-speed simulation. But, it is not easy to debug it because it has poor-capabilities visibility. Other solutions provide full debug capabilities such as RTL emulators, but the offline debugging method needs to recompile the whole design, which slows the verification process. In this chapter, we propose an online RTL-level scan-chain-based methodology for accelerating IP emulation debugging time at Runtime. This method provides internal glue-block which automatically replaces any signal with a mux and extra input, so that at runtime if we enable this method we can replace any internal signal by a forced one. Our experiment shows that, the area overhead is neglected compared to the gained performance benefits. The conventional emulation flow versus the proposed scan-chain based emulation flow is shown in Figs. 6.10 and 6.11 respectively.

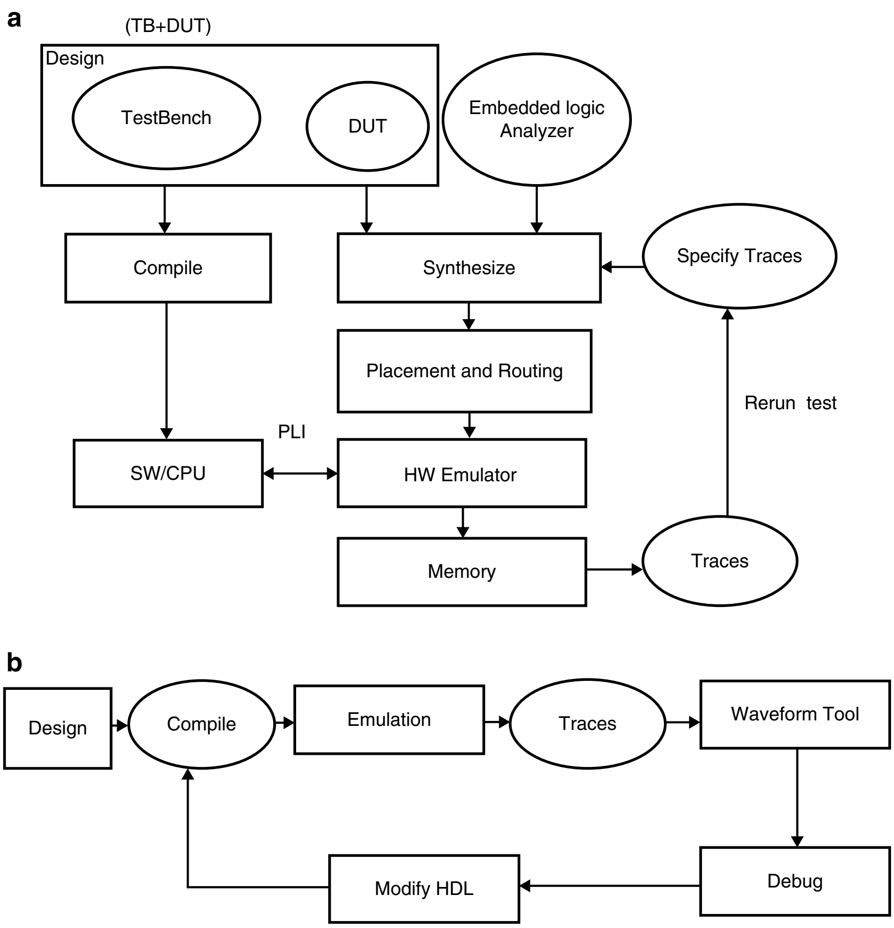


Fig. 6.10 Conventional emulation flow (offline debug) (**a**) detailed, (**b**) simplified [32]

To illustrate the proposed method, we assume the example shown in Fig. 6.12a, where: $out \leq (A+B) \times C$; where C is predetermined value that we want to change it in runtime, we compile the design and run emulation. If we want to change value of C, we have to recompile the whole design. Sometimes, it takes very large time depending on the complexity of the design. So, here we propose to use the online RTL-level scan-chain methodology to be able to change the value of C at run time without recompiling the whole design which accelerates the emulation debugging time. We will create a utility tool that instantiates glue logic and a mux with each "reg" definition in the VERILOG file, the glue logic is a null connection which puts the input into the output as depicted in Fig. 6.12b. So, the designer can change the value at runtime. It will be automatically auto-generated for all the registers defined in the design. Our experiment shows that, the area overhead is neglected compared to the gained performance benefits.

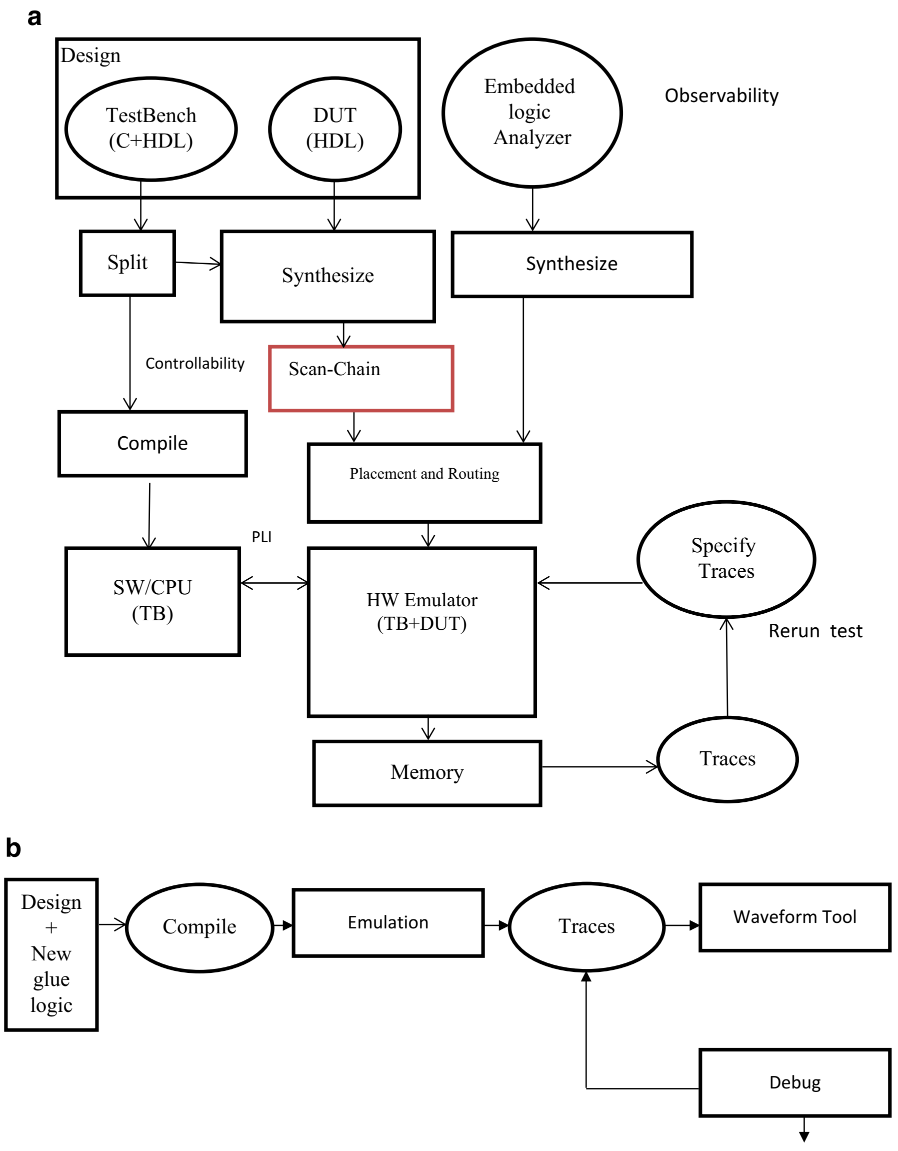


Fig. 6.11 Proposed emulation flow (online flow), synthesizable testbench methodology, scan-chain methodology, (**a**) detailed, (**b**) simplified

6.3.3 Summary

An online RTL-level scan-chain methodology is proposed to reduce debugging time and effort for emulation. Runtime modifications of the values of any of the internal signals of the DUT during execution can be easily performed through the proposed

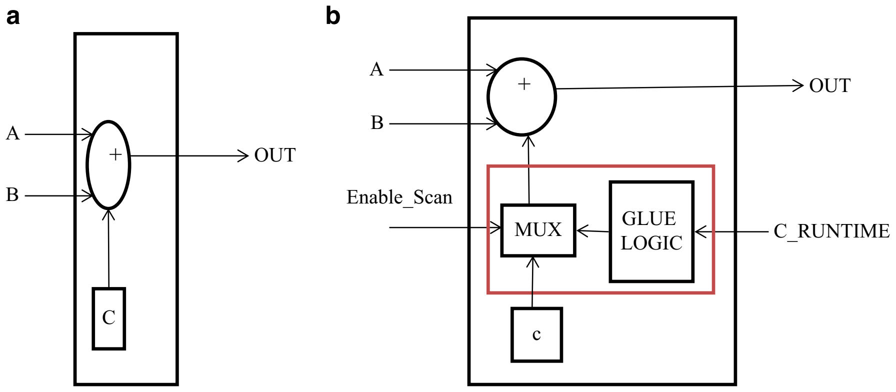


Fig. 6.12 (**a**) Normal design example, (**b**) proposed scan-chain methodology for the design example in (**a**)

online scan-chain methodology. A utility tool was developed to help ease this process. Our experiment shows that the area overhead is neglected compared to the gained performance benefits. But, IP design requires more compilation time.

6.4 Part IV: Automatic Test Generation Based on Genetic Algorithms

6.4.1 Introduction

Verification is the bottleneck in the SoC life cycle. Moreover, the coverage space is very huge. Code coverage cannot cover the functional coverage. The efficiency of the verification is proportional to achieving the coverage goals in less simulation time.

$$\eta_{\text{verification}} \propto \frac{\text{Coverage goals}}{\text{Simulation time}} \tag{6.1}$$

The verification process problems will be considered as an optimization problem. GA is used to solve it. Genetic Algorithms (GA) are the heuristic (experience-based) search and time-efficient learning and optimization techniques that mimic the process of natural evolution based on Darwinian Paradigm (Fig. 6.13). Thus genetic algorithms implement the optimization strategies by simulating evolution of species through natural selection. The nature to computer mapping is shown in Table 6.10, where each cell of a living thing contains chromosomes (strings of DNA), each chromosome contains a set of genes (blocks of DNA), and each gene determines some aspect of the organism (like eye color). In other words, parameters of the solution (genes) are concatenated to form a string (chromosome). In a chromosome, each gene controls a particular characteristic of the individual. The population evolves towards the optimal solution (Fig. 6.14). Evolution based on

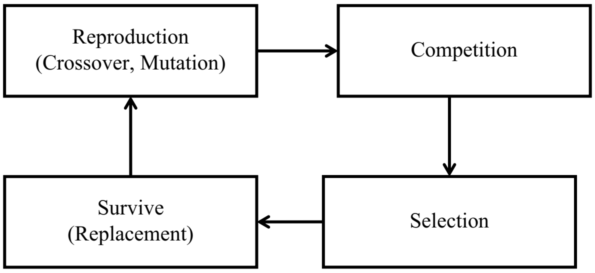


Fig. 6.13 Darwinian paradigm

Table 6.10 The nature to computer mapping

Nature	Computer
Population	Set of solutions
Individual	Solution to a problem
Fitness	Quality of a solution
Chromosome	Encoding for a solution
Gene	Part of the encoding solution
Reproduction	Crossover

"survival of the fittest." Genetic algorithms are well suited for hard problems where little is known about the underlying search space. So, it is considered a robust search and optimization mechanism. The genetic algorithm used in this work consists of the following steps or operations [33–38], and can be seen in Fig. 6.15:

1. Initialization and encoding:
   The GA starts with the creation of random strings, which represent each member in the population.
2. Evaluation (Fitness):
   The fitness used as a measure to reflect the degree of goodness of the individual, is calculated for each individual in the population.
3. Selection
   In the selection process, individuals are chosen from the current population to enter a mating pool devoted to the creation of new individuals for the next generation such that the chance of a given individual to be selected to mate is proportional to its relative fitness. This means that best individuals receive more copies in subsequent generations so that their desirable traits may be passed onto their offspring. This step ensures that the overall quality of the population increases from one generation to the next.
4. Crossover:
   Crossover provides the means by which valuable information is shared among the population. It combines the features of two parent individuals to form two children individuals that may have new patterns compared to those of their parents and plays a central role in Gas. The crossover operator takes two chromosomes and interchanges part of their genetic information to produce two new chromosomes.
5. Mutation:
   Mutation is often introduced to guard against premature convergence. Generally, over a period of several generations, the gene pool tends to become more and

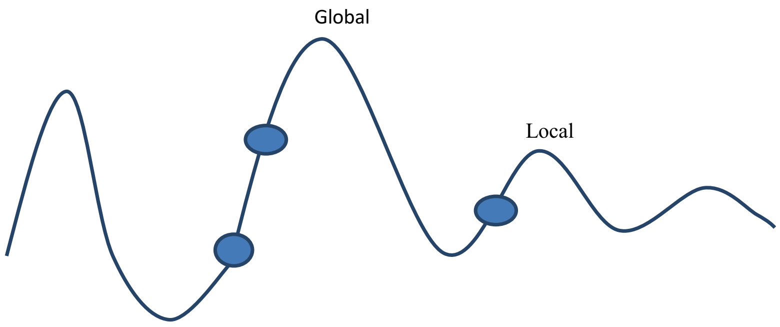


Fig. 6.14 GA searches the optimal solution in the entire search space. We chose random solutions and move around it, until we reach global optimal not local one

more homogeneous. The purpose of mutation is to introduce occasional perturbations to the parameters to maintain genetic diversity within the population.

6. Replacement:
   After generating the offspring's population through the application of the genetic operators to the parents "population, the parents" population is totally replaced by the offspring's population. This is known as no overlapping, generational, replacement. This completes the "life cycle" of the population.
7. Termination
   The GA is terminated when some convergence criterion is met. Possible convergence criteria are: the fitness of the best individual so far found exceeds a threshold value; the maximum number of generations is reached. An example for the parameter used in GA is shown in Table 6.11.

   Many different test data generation methods like random test data generator have been proposed in the literature [33–35].

In this chapter, artificial intelligence algorithms, such as genetic algorithm, are proposed as a novel method for test generation.

6.4.2 Proposed Methodology

The verification process problems will be considered as an optimization problem. GA is used to solve it. The methodology is as follows: generate stimulus based on the feedback from previously generated stimulus to cover areas which were not explored by previously applied tests. During each stimulus cycle, coverage results are collected and sent as an input to the genetic algorithm and used as a guideline for next stimulus. The next stimulus will be more effective compared to randomly generated one (Fig. 6.16). The fitness function here is chosen to maximize the functional coverage percentage, where:

$$\text{Fitness} = \text{Functional coverage ratio} \tag{6.2}$$

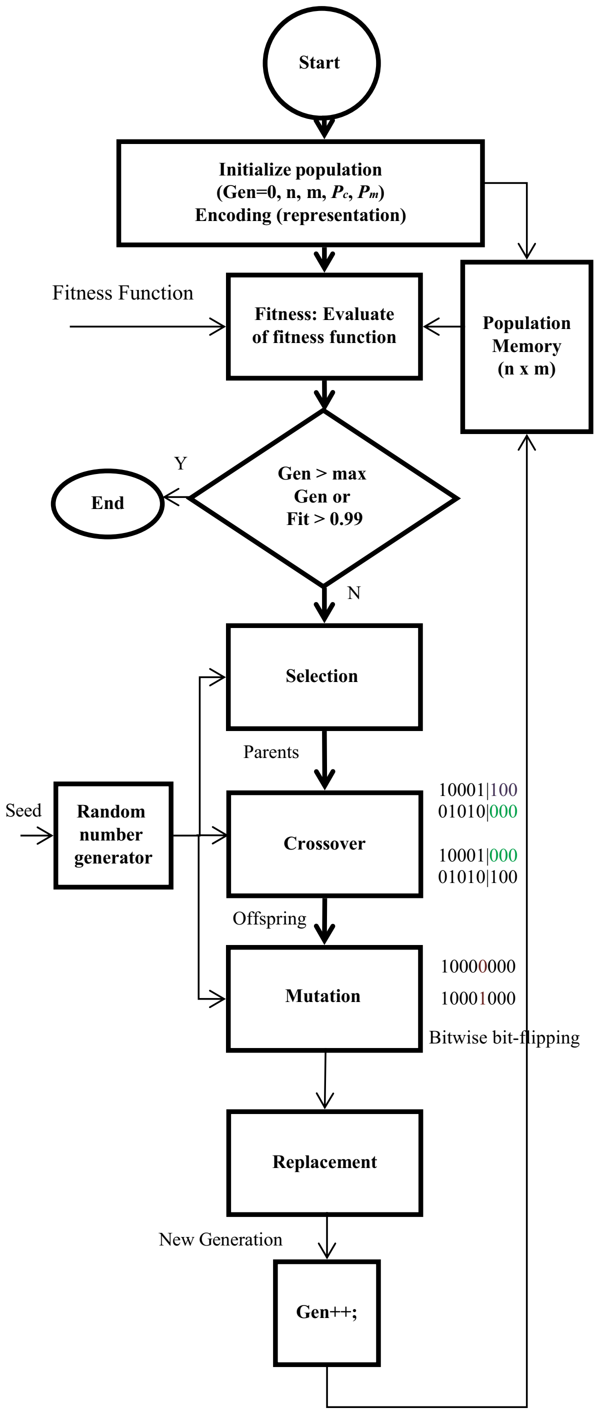


Fig. 6.15 Genetic algorithm chart: A GA typically operates iteratively through a simple cycle of stages: (1) creation of a population of strings, (2) evaluation of each string, (3) selection of the best strings, and (4) genetic manipulation to create a new population of strings. The fitness function is problem-dependent. The used encoding is binary encoding

Table 6.11 Parameters used by the GA, the parameters are not fixed and may be changed according to the situation

Name	Symbol	Value (type)
Number of generations	gen	200
Population size	n	50
Chromosome length	m	80 bits
Crossover probability	P_c	0.9
Mutation probability	P_m	0.01
Type of selection	–	Normal geometric, rank-based selection, Roulette wheel
Type of crossover	–	Arithmetic, multipoint
Type of mutation	–	Nonuniform, flip
Termination method	–	Maximum generation, fitness >0.99

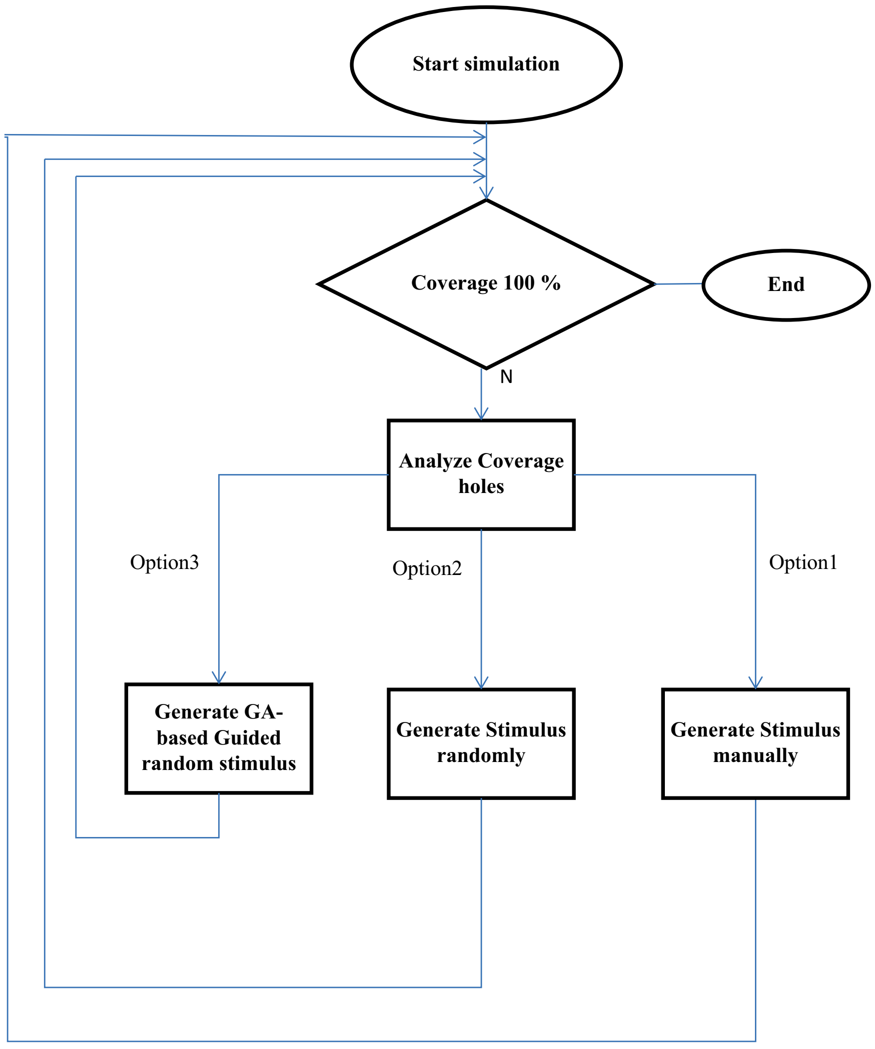

Start simulation
Coverage 100 %
End
N
Analyze Coverage holes
Option3
Option2
Option1
Generate GA-based Guided random stimulus
Generate Stimulus randomly
Generate Stimulus manually

Fig. 6.16 The proposed GA methodology to speedup coverage closure. Using genetic algorithms, there is no test redundancy

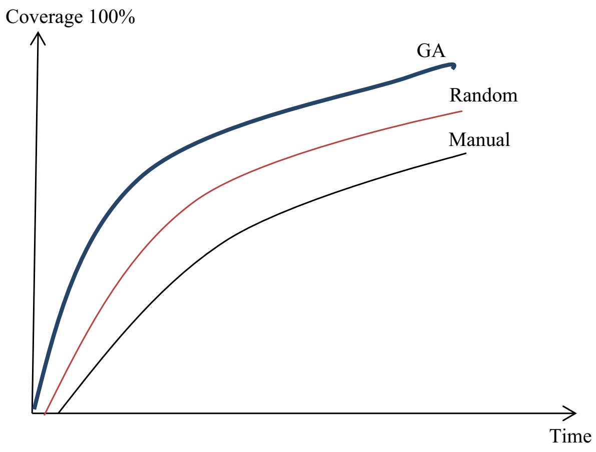


Fig. 6.17 The GA performance

Table 6.12 GA-based test generation results to get 100 % coverage

Method		Random testing		Our GA approach	
Design	# Scenarios (100 % coverage)	# Stimulus	Runtime (s)	# Stimulus	Runtime (s)
#1	4	120	3	100	2
#2	16	200	4	150	2.6
#3	6	130	3.2	90	1
#4	12	180	3.5	110	1.3
#5	8	190	3.7	120	1.5
#6	10	195	3.8	124	2.1
#7	6	130	3	120	2.2
#8	18	210	4	155	2.6
#9	8	180	3.7	96	1.6
#10	14	190	3.5	114	1.5
#11	10	170	3.2	111	1.7
#12	12	215	3.2	144	2.4

Simulation results show that:

1. Coverage holes can be hit automatically with less effort and less time (Fig. 6.17).
2. Computational resources should be low.

The results for some designs are reported in Table 6.12, where it is clear that using GA, we can reach 100 % coverage in less time with less number of stimulus.

6.4.3 Summary

The main challenge in using constraint random testing (CRT) is that manual analysis for the coverage report is needed to find the untested scenarios and modify the testcases to achieve 100 % coverage. We need to replace the manual effort by an automatic method or a tool that will be able to extract the coverage report, identify the untested scenarios, add new constraints, and iterate this process until 100 % coverage is attained. In other words, we need an automated technique to automate the feedback from coverage report analysis to test generation process. In this chapter, the implementation of this automatic feedback loop is presented. The verification environment is created using universal verification methodology (UVM) for reusability. The automatic feedback loop is based on artificial intelligence technique called genetic algorithm (GA). This technique accelerates coverage-driven functional verification and achieves coverage closure rapidly by covering uncovered scenarios in the coverage report (coverage holes).

References

1. Bromley J (2013) If systemverilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language. In: 2013 Forum on Specification & Design Languages (FDL), IEEE, Paris
2. Oliveira FS, Haedicke F, Drechsler R, Kuznik C, Le HM, Ecker W, Mueller W, Große D, Esen V (2012) The system verification methodology for advanced TLM verification. In: Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM, New York, pp 313–322
3. Zhaohui H, Pierres A, Shiqing H, Fang C, Royannez P, See EP, Hoon YL (2012) Practical and efficient SOC verification flow by reusing IP testcase and testbench. In: 2012 International SoC Design Conference (ISOCC), IEEE, Jeju Island, pp 175–178
4. Raghuvanshi S, Singh V (2014) Review on universal verification methodology (UVM) concepts for functional verification. Int J Electr Electron Data Commun 2(3):101–107
5. Young-Nam Yun (2011) Beyond UVM for practical SoC verification. In: International SoC design conference (ISOCC), IEEE, Jeju, pp 158–162
6. Sutherland S, Mills D (2013) Synthesizing systemverilog: busting the myth that systemverilog is only for verification. SNUG Silicon Valley 2013. http://www.sutherland-hdl.com/papers/2013-SNUG-SV_Synthesizable-SystemVerilog_paper.pdf
7. Vaidya B, Pithadiya N (2013) An introduction to universal verification methodology. J Inf Knowl Res Electron Commun Eng 2(02):420–424
8. Spear C, Tumbush G (2012) Systemverilog for verification—a guide to learning the testbench language features, 2nd edn. Springer, New York
9. Sohofi H, Navabi Z (2014) Assertion-based verification for system-level designs. In: Proceedings of 15th International Symposium on Quality Electronic Design (ISQED), IEEE, Santa Clara, pp 582–588
10. Sutherland S, Mills D (2014) Can my synthesis compiler do that? What ASIC and FPGA synthesis compilers support in the systemverilog-2012 standard. In: Presented at DVCon-2014, San Jose
11. Oh Y-J, Song G-Y (2014) System-level verification platform using systemverilog layered testbench & systemC OOP. Int J Control Autom Syst 7(2):221–230

12. Vijayan U, Anjo CA, Vignesh Raja B, Arun Kumar N (2013) Development of basic template environment for functional verification of VLSI design using UVM. Int J Emerg Technol Adv Eng 3(12):214–216
13. Wile B, Goss JC, Roesne W (2005) Comprehensive functional verification the complete industry cycle. Elsevier, San Francisco
14. Accellera (2011) Universal verification methodology (UVM) 1.1 user's guide. Cedence Design System, San Jose
15. Vörtler T, Klotz T, Einwich K, et al. (2014) Enriching UVM in systemC with AMS extensions for randomization ad functional coverage. In: Conference—Design and Verification Conference & Exhibition Europe (DVCon Europe), Munich
16. www.opencores.org
17. http://www.itrs.net/
18. Constantinides K, Mutlu O, Austin TM (2008) Online design bug detection: RTL analysis, flexible mechanisms, and evaluation. In: International Symposium on Microarchitecture (MICRO), IEEE, Lake Como, pp 282–293
19. Park SB, Mitra S (2009) IFRA: Post-silicon bug localization in processors. In: Proceedings of IEEE International High Level Design Validation and Test Workshop, 2007. HLVDT 2007, IEEE, Irvine, pp 154–159
20. Chang K, Wagner I, Bertacco V, Markov I (2007) Automatic error diagnosis and correction for RTL designs. In: Proceedings of IEEE International High Level Design Validation and Test Workshop, 2007. HLVDT 2007, IEEE, Irvine, pp 65–72
21. Mirzaeian S, Zheng F, Cheng K (2008) RTL error diagnosis using a word-level SAT-solver. In: International Test Conference (ITC), IEEE, Santa Clara, pp 1–8
22. Brummayer R, Biere A (2009) Boolector: an efficient SMT solver for bit-vectors and arrays. In: Proceedings of 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, York, pp 174–177
23. Safarpour S, Veneris A (2009) Automated design debugging with abstraction and refinement. IEEE Trans Comput Aided Des Integr Circuits Syst 28(10):1597–1608
24. Peischl B, Wotawa F (2006) Automated source-level error localization in hardware designs. IEEE Des Test Comput 23(1):8–19
25. Matsumoto T, Ono S, Fujita M (2012) An efficient method to localize and correct bugs in high-level designs using counterexamples and potential dependence. In: 20th IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC), IEEE, Santa Cruz
26. Wong WE, Debroy V, Choi B (2010) A family of code coverage-based heuristics for effective fault localization. J Syst Softw 83(2):188–208
27. Wong WE, Wei T (2008) A crosstab-based statistical method for effective fault localization. In: Proceedings of the First International Conference on Software Testing, Verification and Validation (ICST), Lillehammer, pp 42–51
28. Jones JA, Harrold MJ (2005) Empirical evaluation of the Tarantula automatic fault-localization technique. In: Proceedings of International Conference on Automated Software Engineering, New York, pp 273–283
29. Rau J, Chien C, Ma J (2005) Reconfigurable multiple scan-chains for reducing test application time of SOCs. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp 5846–5849
30. Mavroidis I, Papaefstathiou I (2009) Accelerating emulation and providing full chip observability and controllability. IEEE Des Test Comput 26(6):84–94
31. Mavroidis I, Papaefstathiou I (2007) Efficient testbench code synthesis for a hardware emulator system. Design, Automation & Test in Europe Conference & Exhibition (DATE 2007), Nice, pp 1–6
32. Banerjee S, Gupta T (2012) Efficient online RTL debugging methodology for logic emulation systems. In: 25th International Conference on VLSI Design, IEEE, Hyderabad, pp 298–303

33. Yingpan Wu, Lixin Yu, Wei Zhuang and Jianyong Wang (2009) A coverage-driven constraint random-based functional verification method of pipeline unit. ACIS International Conference on Computer and Information Science, IEEE, Shanghai, pp 1049–1054
34. Benjamin M, Geist D, Hartman A, Wolfsthal Y, Mas G, Smeets R (1999) A study in coverage-driven test generation. Proceedings of 36th Issue Design Automation Conference, IEEE, New Orleans, pp 970–975
35. Fine S, Ziv A (2003) Coverage directed test generation for functional verification using Bayesian networks. In: Proceedings of Design Automation Conference, IEEE, pp 286–291, 2–6 June 2003
36. Abo-Hammour ZS, Alsmadi OMK, Al-Smadi AM (2011) Frequency-based model order reduction via genetic algorithm approach. In: 7th International Workshop on Systems, Signal Processing and their Applications (WOSSPA), IEEE, Tipaza, pp 91–94
37. Yun I, Carastro LA, Poddar R, Brooke MA, May GS, Hyun K-S, Pyun KE (2000) Extraction of passive device model parameters using genetic algorithms. ETRI J 22(1):38–46
38. Thirugnanam K, Reena E, Singh M, Kumar P (2014) Mathematical modeling of Li-ion battery using genetic algorithm approach for V2G applications. IEEE Trans Energy Convers 29(2): 332–343

Chapter 7
Conclusions

This book discusses the IP cores life cycle process from specification to production which includes four major steps: (1) IP Modeling, (2) IP verification, (3) IP optimization, (4) IP protection. For IP modeling, four major methodologies are introduced which includes: FPGA-based modeling, processor-based modeling, ASIC-based modeling, and PCB-based modeling. For IP verification, different platforms are presented and analyzed such as simulation, acceleration, emulation, and prototyping. Moreover, different verification methodologies are introduced such as: UVM, direct testing, negative testing, software-driven testing, and formal testing. We presented different methods for IP optimization for the main design methodologies to improve area, speed, and power. For IP protection, we analyzed different strategies to perform protection not to make companies lose revenue and market share.

In this book, we present most famous memory cores and controllers and analyze the trade-off between them. A descriptive comparison between various on-chip memory protocols is made. Comparing the architecture of these different controllers, it is realized that their architecture is common in many things. They mainly differ in the performance and the features. Moreover, we introduce new trends in SoC memories such as PCRAM, ReRAM, MRAM, and 3D memory.

Moreover, in this book, we introduce a deep introduction for SoC buses and peripherals. We explain in detail their features and architectures. Moreover, SoC buses examples are explained in detail. Different SoC bus topologies are discussed such as point to point, unilevel shared bus, hierarchical bus, ring, cross-bar bus, NoC. The arbitration algorithms are explored. We give a methodology for extraction of any SoC bus features from its standard. The different features include topology, arbitration, bus width, transfers, timing, transmission control, and type.

In this book, we introduce a deep introduction for Verilog for both implementation and verification point of view. The chapter used design examples for showing ways in which Verilog could be used in a design for both implementation and verification. This chapter did not cover all of Verilog, but only some important topics. Moreover, a survey on the current existing logic simulators is presented.

K.S. Mohamed, *IP Cores Design from Specifications to Production*,
Analog Circuits and Signal Processing, DOI 10.1007/978-3-319-22035-2_7

This book presents an overview on building a reusable RTL verification environment using the UVM verification methodology. UVM is a culmination of well-known ideas and best practices. This book also presents a survey on the features of UVM. It presents its pros, cons, challenges, and opportunities. Moreover, it presents simple steps to verify an IP and build an efficient and smart verification environment. A SoC case study was presented to compare traditional verification with UVM-based verification.

Bug localization is a process of identifying the specific locations or regions of source code that is buggy and needs to be modified to repair the defect. Bug localization can significantly reduce human effort and design cost.

In this book, a novel automated coverage-based functional bug localization method for complex HDL designs is proposed which significantly reduces debugging time. The proposed bug localization methodology takes information from regression suite as an input and produces a ranked list of suspicious part of code. Our methodology is a promising solution to reduce required time to localize bugs significantly.

An online RTL-level scan-chain methodology is proposed to reduce debugging time and effort for emulation. Runtime modifications of the values of any of the internal signals of the DUT during execution can be easily performed through the proposed online scan-chain methodology. A utility tool was developed to help ease this process. Our experiment shows that, the area overhead is neglected compared to the gained performance benefits. But, IP design requires more compilation time.

The main challenge in using constraint random testing (CRT) is that manual analysis for the coverage report is needed to find the untested scenarios and modify the test cases to achieve 100 % coverage. We need to replace the manual effort by an automatic method or a tool that will be able to extract the coverage report, identify the untested scenarios, add new constraints, and iterate this process until 100 % coverage is attained. In other words, we need an automated technique to automate the feedback from coverage report analysis to test generation process. In this chapter, the implementation of this automatic feedback loop is presented. The verification environment is created using universal verification methodology (UVM) for reusability. The automatic feedback loop is based on artificial intelligence technique called genetic algorithm (GA). This technique accelerates coverage-driven functional verification and achieves coverage closure rapidly by covering uncovered scenarios in the coverage report (coverage holes).