**Chap-05 Interfacing and Communication**

## I/O Fundamentals

Peripheral devices that are under the direct control of the computer are said to be connected online.
Input –output devices attached to the computer are also called peripherals.

The input–output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user. A computer serves no useful purpose without the ability to receive information from an outside source and to transmit results in a meaningful form.

Input Devices are: Keyboard, mouse, Scanner, Joystick etc.
Output Devices: Monitor, Printer, Speaker, and Headphone Faxes etc.

## Important Monitor Facts

A monitor, no matter the type, connects to either an HDMI, DVI, or VGA port on the computer. Before investing in a new monitor, make sure that both devices support the same type of connection.

Monitors are not typically user serviceable. For your safety, it's not usually wise to open and work on a monitor.

## Monitor Description

Monitors are display devices external to the computer case and connect via a cable to a port on the video card or motherboard. Even though the monitor sits outside the main computer housing, it is an essential part of the complete system.

Monitors come in two major types - LCD or CRT. CRT monitors look much like old-fashioned televisions and are very deep in size. LCD monitors are much thinner, use less energy, and provide a greater graphics quality.

LCD monitors have completely obsoleted CRT monitors due to their higher quality, smaller "footprint" on the desk, and decreasing price.

Most monitors are in a widescreen format and range in size from 17" to 24" or more. This size is a diagonal measurement from one corner of the screen to the other.

Monitors are built-in as part of the computer system in laptops, tablets, netbooks, and all-in-one desktop machines.

## CRT vs. LCD Monitors

The primary advantage that CRT monitors held over LCDs was their color rendering. The contrast ratios and depths of colors displayed were much greater with CRT monitors than LCDs. While this still holds true in most cases, many strides have been made in LCDs such that this difference is not as great as it once was. Many graphic designers still use the very expensive large CRT monitors in their work because of the color advantages. Of course, this color ability does degrade over time as the phosphors in the tube break down.

The other advantage that CRT monitors held over LCD screens is the ability to easily scale to various resolutions. This is referred to as multisync by the industry. By adjusting the electron beam in the tube, the screen can easily be adjusted downward to lower resolutions while keeping the picture clarity intact.

While these two items may play an important role for CRT monitors, there are disadvantages as well. The biggest of these are the size and weight of the tubes. An equivalent sized LCD monitor is upwards of 80% smaller in size and weight compared to a CRT tube. The larger the screen, the bigger the size difference. The other major drawback deals with the power consumption. The energy needed for the electron beam means that the monitors consumer and generate a lot more heat than the LCD monitors.

## Definition of High Definition

High Definition is largely a fluid term, taking the shape of a lot of different containers, with the only real meaning being something with a significantly increased picture quality and clarity based on the resolution of the monitor in question. For this reason, High Definition is, in reality, a synonym for high resolution.

High resolution means more pixels in your screen, which leads to a remarkably more clear picture. There are some standards now that allow a more concrete resolution of what it means to have an "official" HD display monitor, even for your PC.

The following are the standard definitions for HD video, which is able to be displayed on monitors of slightly varying native resolutions, some being standard for computer screens, other for TV screens, but they are to a large degree interchangeable because they all work to display these resolutions of video:

- **1280x720 - aka 720p**
- **1920x1080 - aka 1080i**
- **1920x1080 Progressive - aka 1080p**

The "p" and "i" after the resolution denotes either Progressive or Interlaced scanning, respectively. Progressive has been proven to the the faster, clearer picture, less prone to blurring, and therefore has won the title of the "standardized" best possible resolution for HD broadcasting at 1080p.

**Buffer**

A buffer is a data area shared by hardware devices or program processes that operate at different speeds or with different sets of priorities. The buffer allows each device or process to operate without being held up by the other. In order for a buffer to be effective, the size of the buffer and the algorithms for moving data into and out of the buffer need to be considered by the buffer designer. Like a cache, a buffer is a "midpoint holding place" but exists not so much to accelerate the speed of an activity as to support the coordination of separate activities.

This term is used both in programming and in hardware. In programming, buffering sometimes implies the need to screen data from its final intended place so that it can be edited or otherwise

The input–output organization of a computer is a function of the size of the computer and the devices connected to it. The difference between a small and a large system is mostly dependent on the amount of hardware the computer has available for communicating with peripheral units and the number of peripherals connected to the system. Since each peripheral behaves differently from any other, it would be prohibitive to dwell on the detailed interconnections needed between the computer and each peripheral. Certain techniques common to most peripherals are presented in this chapter.

processed before being moved to a regular file or database.

**Input-Output Interface**

-It provides a method of transferring information between internal storage and external I/O devices.

- The data transfer rate of peripherals is usually slower than the transfer rate of the CPU.

- The operating modes of peripherals are different from each other.

**I/O Bus Interface Modules**

- I/O bus consists of data lines, address lines, and control lines.

-The I/O bus from the processor is attached to all peripheral interfaces.

-When the interface detects its own address, it activates the path between the bus lines and the device that it controls.

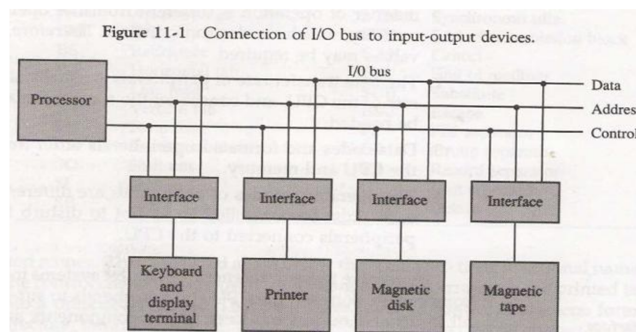


Figure 11-1 Connection of I/O bus to input-output devices.

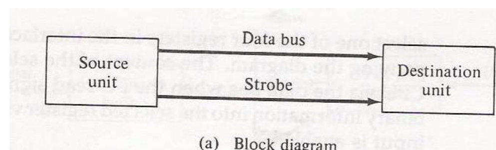
- The interface selected respond to the function code and proceeds to execute it, the function code is referred as I/O command.

**Handshaking, Strobe**

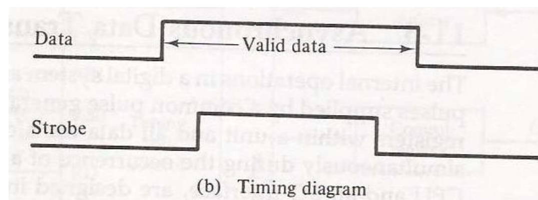
Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. One way of achieving this is by means of a *strobe* pulse supplied by one of the units to indicate to the other unit when the transfer has to occur. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as *handshaking*.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two independent units. In the general case we consider the transmitting unit as the source and the receiving unit as the destination. For example, the CPU is the source unit during an output or a write transfer and it is the destination unit during an input and read transfer.

-In many computers the strobe tube is controlled by the clock pulse in the CPU.
-Time out- if the data is not transferred during predetermined time.



(a) Block diagram

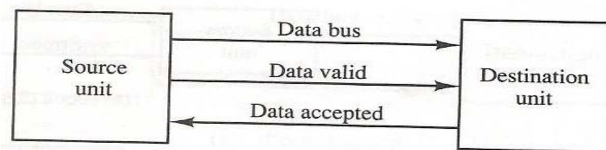


(b) Timing diagram

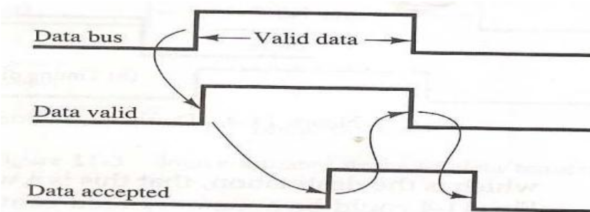
Figure 11-3 Source-initiated strobe for data transfer.

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer. The basic principle of the two-wire handshaking method of data transfer is as follows. One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there are valid data in the bus. The other control line is in the other direction from the destination to the source. It is used by the destination unit to inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.

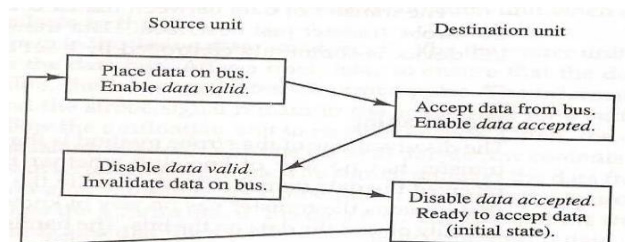
- The two handshaking lines are data valid, which is generated by the source unit and data accepted, are generated by the destination unit.
- The source unit initiated the transfer by placing the data on the bus and enabling its data valid signal, which validates the data on the bus.
- The source unit then disable sits data valid signals, which invalidates the data on the bus.




(a) Block diagram




(b) Timing diagram




(c) Sequence of events

**Modes of Transfer**

**-Data** transfer between central computer and I/O devices may be handled in a variety of modes.

- Programmed I/O
- Interrupt I/O
- Direct memory Access

- Programmed I/O operations are the result of I/O instructions written in the computer program.

-Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made.

- in a **program I/O** method, the CPU stays in a program loop until the I/Unit indicates that it is ready for data transfer.

- This is a time consuming process, it keeps the processor busy.

- it can be avoided by using an interrupt facility and special command to inform the interface to issue an interrupt request signal when the data are available from the device. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

Transfer of data under programmed I/O is between CPU and peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory.

**Interrupt Structures**

A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to requests which, if delayed or interrupted, could have serious consequences. Devices with high-speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from one unit to another. In principle, there are two methods for accomplishing this. One is called *vectored interrupt* and the other, *nonvectored interrupt*. In a nonvectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the *interrupt vector*. In some

the interrupt vector is the first address of the I/O service routine.

- The device with the highest priority is placed in the first position, followed by lower priority devices up to the device with lowest priority, which is placed last.
- If any device has its interrupt signal in the low level state, the interrupt lines goes in the low level state and enables the interrupt input in the CPU.

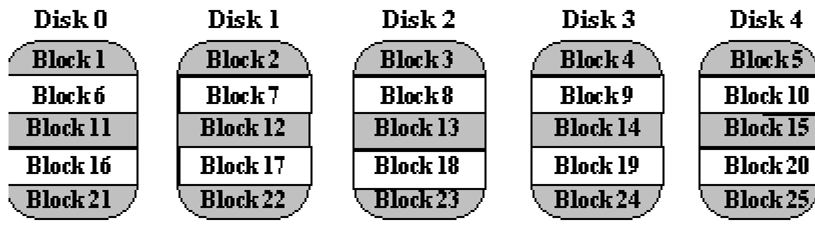
When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU. This is equivalent to a negative-logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its *PI* (priority in) input. The acknowledge signal passes on to the next device through the *PO* (priority out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the *PO* output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.

**RAID Architecture**

## Non-Redundant (RAID Level 0)

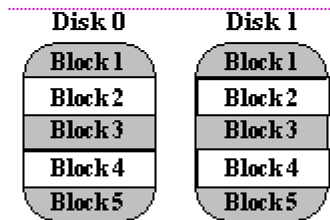
A non-redundant disk array, or RAID level 0, has the lowest cost of any RAID organization because it does not employ redundancy at all. This scheme offers the best performance since it never needs to update redundant information. Surprisingly, it does not have the best performance. Redundancy schemes that duplicate data, such as mirroring, can perform better on reads by selectively scheduling requests on the disk with the shortest expected seek and rotational delays. Without, redundancy, any single disk failure will result in data-loss. Non-redundant disk arrays are widely used in super-computing environments where performance and capacity, rather than reliability, are the primary concerns.

Sequential blocks of data are written across multiple disks in stripes, as follows:




**Mirrored (RAID Level 1)**

The traditional solution, called mirroring or shadowing, uses twice as many disks as a non-redundant disk array. whenever data is written to a disk the same data is also written to a redundant disk, so that there are always two copies of the information. When data is read, it can be retrieved from the disk with the shorter queuing, seek and rotational delays. If a disk fails, the other copy is used to service requests. Mirroring is frequently used in database applications where availability and transaction time are more important than storage efficiency.




**Memory-Style (RAID Level 2)**

Memory systems have provided recovery from failed components with much less cost than mirroring by using Hamming codes. Hamming codes contain parity for distinct overlapping subsets of components. In one version of this scheme, four disks require three redundant disks, one less than mirroring. Since the number of redundant disks is proportional to the log of the total number of the disks on the system, storage efficiency increases as the number of data disks increases.

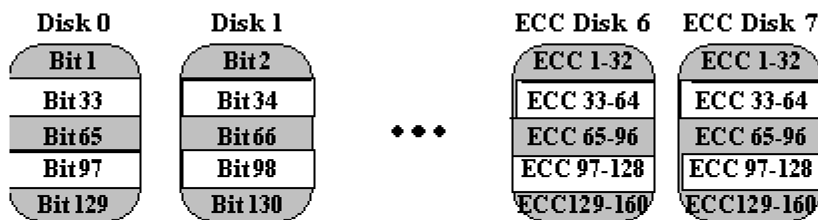
If a single component fails, several of the parity components will have inconsistent values, and the failed component is the one held in common by each incorrect subset. The lost information is recovered by reading the other components in a subset, including the parity component, and setting the missing bit to 0 or 1 to create proper parity value for that subset. Thus, multiple redundant disks are needed to identify the failed disk, but only one is needed to recover the lost information.

In you are unaware of parity, you can think of the redundant disk as having the sum of all data in the other disks. When a disk fails, you can subtract all the data on the good disks form the parity

disk; the remaining information must be the missing information. Parity is simply this sum modulo 2.

A RAID 2 system would normally have as many data disks as the word size of the computer, typically 32. In addition, RAID 2 requires the use of extra disks to store an error-correcting code for redundancy. With 32 data disks, a RAID 2 system would require 7 additional disks for a Hamming-code ECC. Such an array of 39 disks was the subject of a U.S. patent granted to Unisys Corporation in 1988, but no commercial product was ever released.

For a number of reasons, including the fact that modern disk drives contain their own internal ECC, RAID 2 is not a practical disk array scheme.




## Chap-06 Functional Organization

**Implementation of simple data paths**

### 4.1. The Central Processor - Control and Dataflow

Recall that, in Section 3, we designed an ALU based on (a) building blocks such as multiplexers for selecting an operation to produce ALU output, (b) carry look ahead adders to reduce the complexity and (in practice) the critical path length of arithmetic operations, and (c) components such as coprocessors to perform costly operations such as floating point arithmetic. We also showed that computer arithmetic suffers from errors due to finite precision, lack of associatively, and limitations of protocols such as the IEEE 754 floating point standard

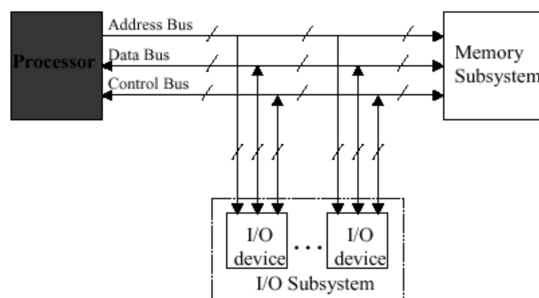



**Figure 4.1.** Schematic diagram of a modern von Neumann processor, where the CPU is denoted by a shaded box.

## 4.2. Datapath Design and Implementation

The data path is the "brawn" of a processor, since it implements the fetch-decode-execute cycle. The general discipline for data path design is to (1) determine the instruction classes and formats in the ISA, (2) design data path components and interconnections for each instruction class or format, and (3) compose the data path segments designed in Step 2) to yield a composite data path.

Simple data path components include *memory* (stores the current instruction), *PC* or program counter (stores the address of current instruction), and *ALU* (executes current instruction). The interconnection of these simple components to form a basic data path is illustrated in Figure 4.5. Note that the register file is written to by the output of the ALU. As in Section 4.1, the register file shown in Figure 4.6 is clocked by the RegWrite signal.

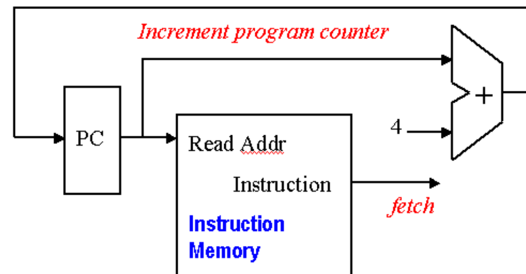



**Figure 4.5.** Schematic high-level diagram of MIPS datapath from an implementation perspective

Implementation of the datapath for I- and J-format instructions requires two more components - a *data memory* and a *sign extender*, illustrated in Figure 4.6. The data memory stores ALU results and operands, including instructions, and has two enabling inputs (MemWrite and MemRead) that cannot both be active (have a logical high value) at the same time. The data memory accepts an address and either accepts data (WriteData port if MemWrite is enabled) or outputs data (ReadData port if MemRead is enabled), at the indicated address. The sign extender adds 16 leading digits to a 16-bit word with most significant bit *b*, to product a 32-bit word. In particular, the additional 16 digits have the same value as *b*, thus implementing sign extension in twos complement representation.

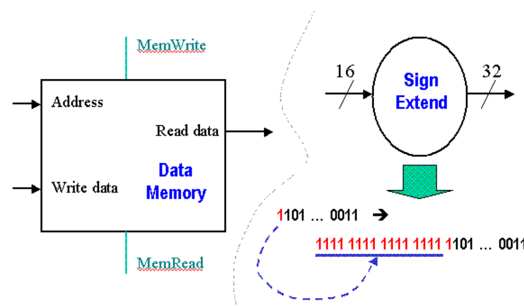



**Figure 4.6.** Schematic diagram of Data Memory and Sign Extender

### 4.2.2. Load/Store Datapath

The load/store datapath uses instructions such as lw $t1, **offset**($t2), where *offset* denotes a memory address offset applied to the base address in register $t2. The lw instruction reads from memory and writes into register $t1. The sw instruction reads from register $t1 and writes into memory. In order to compute the memory address, the MIPS ISA specification says that we have to sign-extend the 16-bit offset to a 32-bit signed value. This is done using the sign extender shown in Figure 4.6.

The load/store datapath is illustrated in Figure 4.8, and performs the following actions in the order given:

1. *Register Access* takes input from the register file, to implement the instruction, data, or address *fetch* step of the fetch-decode-execute cycle.
2. *Memory Address Calculation* decodes the base address and offset, combining them to produce the actual memory address. This step uses the sign extender and ALU.
3. *Read/Write from Memory* takes data or instructions from the data memory, and implements the first part of the *execute* step of the fetch/decode/execute cycle.
4. *Write into Register File* puts data or instructions into the data memory, implementing the second part of the *execute* step of the fetch/decode/execute cycle.

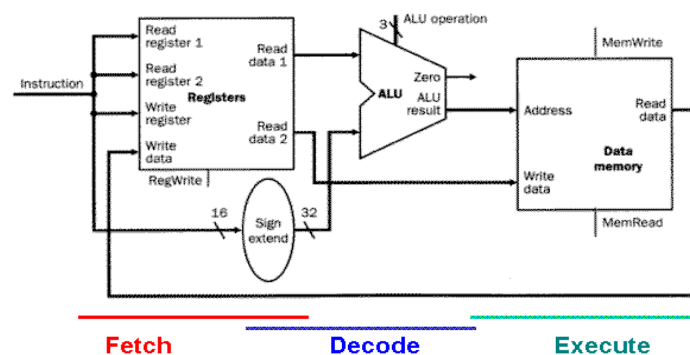



**Figure 4.8.** Schematic diagram of the Load/Store instruction datapath. Note that the *execute* step also includes writing of data back to the register file, which is not shown in the figure, for simplicity

### 4.2.3. Branch/Jump Datapath

The branch datapath (jump is an unconditional branch) uses instructions such as `beq $t1, $t2,` **`offset`**, where *offset* is a 16-bit offset for computing the branch target address via PC-relative addressing. The `beq` instruction reads from registers `$t1` and `$t2`, then compares the data obtained from these registers to see if they are equal. If equal, the branch is taken. Otherwise, the branch is not taken.

By *taking the branch*, the ISA specification means that the ALU adds a sign-extended offset to the program counter (PC). The offset is shifted left 2 bits to allow for word alignment (since $2^2$ =

4, and words are comprised of 4 bytes). Thus, to jump to the target address, the lower 26 bits of the PC are replaced with the lower 26 bits of the instruction shifted left 2 bits.

The branch instruction datapath is illustrated in Figure 4.9, and performs the following actions in the order given:

1. *Register Access* takes input from the register file, to implement the *instruction fetch* or *data fetch* step of the fetch-decode-execute cycle.
2. *Calculate Branch Target* - Concurrent with ALU #1's evaluation of the branch condition, ALU #2 calculates the branch target address, to be ready for the branch if it is taken. This completes the *decode* step of the fetch-decode-execute cycle.
3. *Evaluate Branch Condition and Jump to BTA or PC+4* uses ALU #1 in Figure 4.9, to determine whether or not the branch should be taken. Jump to BTA or PC+4 uses control logic hardware to transfer control to the instruction referenced by the branch target address. This effectively changes the PC to the branch target address, and completes the *execute* step of the fetch-decode-execute cycle.

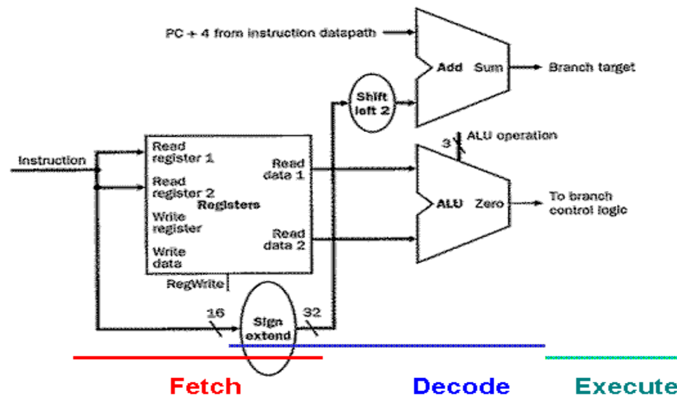



**Figure 4.9.** Schematic diagram of the Branch instruction datapath. Note that, unlike the Load/Store datapath, the *execute* step does not include writing of results back to the register file

The branch datapath takes operand #1 (the offset) from the instruction input to the register file, then sign-extends the offset. The sign-extended offset and the program counter (incremented by 4 bytes to reference the next instruction after the branch instruction) are combined by ALU #1 to yield the branch target address. The operands for the branch condition to evaluate are concurrently obtained from the register file via the ReadData ports, and are input to ALU #2, which outputs a one or zero value to the branch control logic.

MIPS has the special feature of a *delayed branch*, that is, instruction $I_b$ which follows the branch is always fetched, decoded, and prepared for execution. If the branch condition is false, a normal branch occurs. If the branch condition is true, then $I_b$ is executed. One wonders why this extra work is performed - the answer is that delayed branch improves the efficiency of pipeline execution, as we shall see in Section 5. Also, the use of branch-not-taken (where $I_b$ is executed) is sometimes the common case.

## Control Unit

### Hardwired Control Unit

Figure 2 is a block diagram showing the internal organization of a hard-wired control unit for our simple computer. Input to the controller consists of the 4-bit opcode of the instruction currently contained in the Instruction Register and the negative flag from the accumulator. The controller's output is a set of 16 control signals that go out to the various registers and to the memory of the computer, in addition to a HLT signal that is activated whenever the leading bit of the op-code is one. The controller is composed of the following functional units: A ring counter, an instruction decoder, and a control matrix.

The ring counter provides a sequence of six consecutive active signals that cycle continuously. Synchronized by the system clock, the ring counter first activates its T0 line, then its T1 line, and so forth. After T5 is active, the sequence begins again with T0. Figure 3 shows how the ring counter might be organized internally.

The instruction decoder takes its four-bit input from the op-code field of the instruction register and activates one and only one of its 8 output lines. Each line corresponds to one of the instructions in the computer's instruction set. Figure 4 shows the internal organization of this decoder.

The most important part of the hard-wired controller is the control matrix. It receives input from the ring counter and the instruction decoder and provides the proper sequence of control signals. Figure 5 is a diagram of how the control matrix for our simple machine might be wired. To understand how this diagram was obtained, we must look carefully at the machine's instruction set The various control signals are placed horizontally along the top of the table. Entries into the table consist of the moments (ring counter pulses T0, T1, T2, T3, T4, or T5) at which each control signal must be active in order to have the instruction executed. This table is prepared very easily by reading off the information for each instruction given in Table 1. For example, the Fetch operation has the EP and LM control signals active at ring count 1, and ED, LI, and IPC active at ring count 2. Therefore the first row (Fetch) of Table 2 has T0 entered below EP and LM, T1 below R, and T2 below IP, ED, and LI.

Once Table 2 has been prepared, the logic required for each control signal is easily obtained. For each an AND operation is performed between any active ring counter (Ti) signals that were entered into the signal's column and the corresponding instruction contained in the far left-hand column. If a column has more than one entry, the output of the ANDs are ORed together to produce the final control signal. For example, the LM column has the following entries: T0 (Fetch), T3 associated with the LDA instruction, and T3 associated with the STA instruction. Therefore, the logic for this signal is:

LM = T0 + T3*LDA + T3*STA

This means that control signal LM will be activated whenever any of the following conditions is satisfied: (1) ring pulse T0 (first step of an instruction fetch) is active, or (2) an LDA instruction

is in the IR and the ring counter is issuing pulse 3, or (3) and STA instruction is in the IR and the ring counter is issuing pulse 3.

The entries in the JN (Jump Negative) row of this table require some further explanation. The LP and EI signals are active during T3 for this instruction if and only if the accumulator's negative flag has been set. Therefore the entries that appear above these signals for the JN instruction are T3*NF, meaning that the state of the negative flag must be ANDed in for the LP and EI control signals.

These equations have been read from Table 2, as explained above. The circuit diagram of the control matrix (Figure 5) is constructed directly from these equations.

It should be noticed that the HLT line from the instruction decoder does not enter the control matrix, Instead this signal goes directly to circuitry (not shown) that will stop the clock and thus terminate execution.

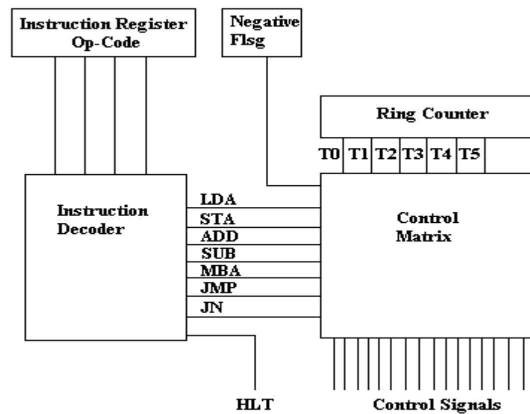



**Figure 5.** Hardwired Control unit

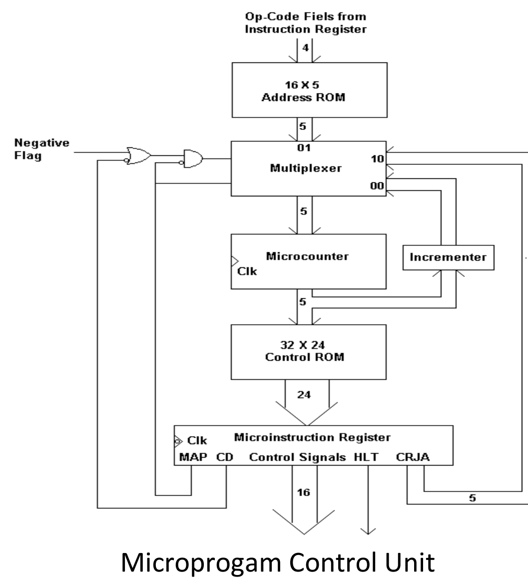
**Micro Program Control Unit**

As we have seen, the controller causes instructions to be executed by issuing a specific set of control signals at each beat of the system clock. Each set of control signals issued causes one basic operation (micro-operation), such as a register transfer, to occur within the data path section of the computer. In the case of a hard-wired control unit the control matrix is responsible for sending out the required sequence of signals.

An alternative way of generating the control signals is that of micro-programmed control. In order to understand this method it is convenient to think of sets of control signals that cause specific micro-operations to occur as being "microinstructions" that could be stored in a memory. Each bit of a microinstruction might correspond to one control signal. If the bit is set it means that the control signal will be active; if cleared the signal will be inactive. Sequences of microinstructions could be stored in an internal "control" memory. Execution of a machine language instruction could then be caused by fetching the proper sequence of microinstructions

from the control memory and sending them out to the data path section of the computer. A sequence of microinstructions that implements an instruction on the external computer is known as a micro-routine. The instruction set of the computer is thus determined by the set of micro-routines, the "microprogram," stored in the controller's memory. The control unit of a microprogram-controlled computer is essentially a computer within a computer.

Figure 7 is a block diagram of a micro-programmed control unit that may be used to implement the instruction set of the computer we described above. The heart of the controller is the control 32 X 24 ROM memory in which upt to 32 24-bit long microinstructions can be stored. Each is composed of two main fields: a 16-bit wide control signal field and an 8-bit wide next-address field. Each bit in the control signal field corresponds to one of the control signals discussed above. The next-address field contains bits that determine the address of the next microinstruction to be fetched from the control ROM. We shall see the details of how these bits work shortly. Words selected from the control ROM feed the microinstruction register. This 24-bit wide register is analogous to the outer machine's instruction register. Specifically, the leading 16 bits (the control-signal field) of the microinstruction register are connected to the control-signal lines that go to the various components of the external machine's data path section.



Microprogam Control Unit

**Pipelining**

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments. The name "pipeline" implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments

The pipeline organization is being shown by an example:
Suppose we want to perform the combined multiply, and add with a stream of numbers

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2. $R1$ through $R5$ are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

| | |
|---|---|
| $R1 \leftarrow A_i, \quad R2 \leftarrow B_i$ | Input $A_i$ and $B_i$ |
| $R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$ | Multiply and input $C_i$ |
| $R5 \leftarrow R3 + R4$ | Add $C_i$ to product |

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1. The first clock pulse transfers $A_1$ and $B_1$ into $R1$ and

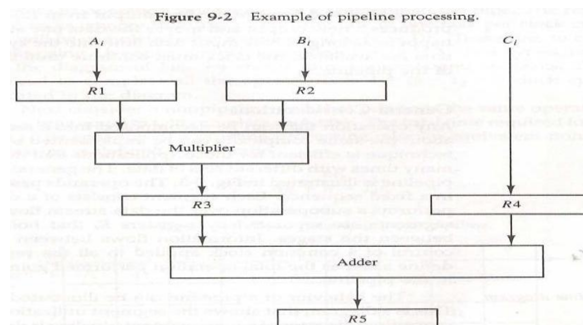


**TABLE 9-1** Content of Registers in Pipeline Example

| Clock Pulse Number | Segment 1 | | Segment 2 | | Segment 3 |
|---|---|---|---|---|---|
| | $R1$ | $R2$ | $R3$ | $R4$ | $R5$ |
| 1 | $A_1$ | $B_1$ | — | — | — |
| 2 | $A_2$ | $B_2$ | $A_1 * B_1$ | $C_1$ | — |
| 3 | $A_3$ | $B_3$ | $A_2 * B_2$ | $C_2$ | $A_1 * B_1 + C_1$ |
| 4 | $A_4$ | $B_4$ | $A_3 * B_3$ | $C_3$ | $A_2 * B_2 + C_2$ |
| 5 | $A_5$ | $B_5$ | $A_4 * B_4$ | $C_4$ | $A_3 * B_3 + C_3$ |
| 6 | $A_6$ | $B_6$ | $A_5 * B_5$ | $C_5$ | $A_4 * B_4 + C_4$ |
| 7 | $A_7$ | $B_7$ | $A_6 * B_6$ | $C_6$ | $A_5 * B_5 + C_5$ |
| 8 | — | — | $A_7 * B_7$ | $C_7$ | $A_6 * B_6 + C_6$ |
| 9 | — | — | — | — | $A_7 * B_7 + C_7$ |

$R2$. The second clock pulse transfers the product of $R1$ and $R2$ into $R3$ and $C_1$ into $R4$. The same clock pulse transfers $A_2$ and $B_2$ into $R1$ and $R2$. The third clock pulse operates on all three segments simultaneously. It places $A_3$ and $B_3$ into $R1$ and $R2$, transfers the product of $R1$ and $R2$ into $R3$, transfers $C_2$ into $R4$, and places the sum of $R3$ and $R4$ into $R5$. It takes three clock pulses to fill up the pipe and retrieve the first output from $R5$. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

**Instruction Pipelining**

Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline. The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer. This is a type of unit that forms a queue rather than a stack. Whenever the execution unit is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis. Thus an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment. The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions. Whenever there is space in the FIFO buffer, the control unit initiates the next instruction fetch phase. The buffer acts as a queue from which control then extracts the instructions for the execution unit.

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following sequence of steps.

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory. Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. In this way, an instruction word and a data word can be read simultaneously from two different modules.

**Problems in Instruction Pipelining**

Several difficulties prevent instruction pipelining from being as simple as the above description suggests. The principal problems are:

TIMING VARIATIONS: Not all stages take the same amount of time. This means that the speed gain of a pipeline will be determined by its slowest stage. This problem is particularly acute in instruction processing, since different instructions have different operand requirements and sometimes vastly different processing time. Moreover, synchronization mechanisms are required to ensure that data is passed from stage to stage only when both stages are ready.

DATA HAZARDS: When several instructions are in partial execution, a problem arises if they reference the same data. We must ensure that a later instruction does not attempt to access data sooner than a preceding instruction, if this will lead to incorrect results. For example, instruction N+1 must not be permitted to fetch an operand that is yet to be stored into by instruction N.

BRANCHING: In order to fetch the "next" instruction, we must know which one is required. If the present instruction is a conditional branch, the next instruction may not be known until the current one is processed.

INTERRUPTS: Interrupts insert unplanned "extra" instructions into the instruction stream. The interrupt must take effect between instructions, that is, when one instruction has completed and the next has not yet begun. With pipelining, the next instruction has usually begun before the current one has completed

## Some Solutions

Possible solutions to the problems described above include the following strategies:

*Timing Variations*

To maximize the speed gain, stages must first be chosen to be as uniform as possible in timing requirements. However, a timing mechanism is needed. A synchronous method could be used, in which a stage is assumed to be complete in a definite number of clock cycles. However, asynchronous techniques are generally more efficient. A flag bit or signal line is passed forward to the next stage indicating when valid data is available. A signal must also be passed back from the next stage when the data has been accepted.

In all cases there must be a buffer register between stages to hold the data; sometimes this buffer is expanded to a memory which can hold several data items. Each stage must take care not to accept input data until it is valid, and not to produce output data until there is room in its output buffer.

*Data Hazards*

To guard against data hazards it is necessary for each stage to be aware of the operands in use by stages further down the pipeline. The type of use must also be known, since two successive reads do not conflict and should not be cause to slow the pipeline. Only when writing is involved is there a possible conflict.

The pipeline is typically equipped with a small associative check memory which can store the address and operation type (read or write) for each instruction currently in the pipe. The concept of "address" must be extended to identify registers as well. Each instruction can affect only a small number of operands, but indirect effects of addressing must not be neglected.

As each instruction prepares to enter the pipe, its operand addresses are compared with those already stored. If there is a conflict, the instruction (and usually those behind it) must wait. When

there is no conflict, the instruction enters the pipe and its operands addresses are stored in the check memory. When the instruction completes, these addresses are removed. The memory must be associative to handle the high-speed lookups required.

### *Branching*

The problem in branching is that the pipeline may be slowed down by a branch instruction because we do not know which branch to follow. In the absence of any special help in this area, it would be necessary to delay processing of further instructions until the branch destination is resolved. Since branches are extremely frequent, this delay would be unacceptable.

One solution which is widely used, especially in RISC architectures, is **deferred branching**. In this method, the instruction set is designed so that after a conditional branch instruction, the next instruction in sequence is always executed, and then the branch is taken. Thus every branch must be followed by one instruction which logically precedes it and is to be executed in all cases. This gives the pipeline some breathing room. If necessary this instruction can be a no-op, but frequent use of no-ops would destroy the speed benefit.

Use of this technique requires a coding method which is confusing for programmers but not too difficult for compiler code generators.

Most other techniques involve some type of **speculative execution**, in which instructions are processed which are not known with certainty to be correct. It must be possible to discard or "back out" from the results of this execution if necessary.

The usual solution is to follow the "obvious" branch, that is, the next sequential instruction, taking care to perform no irreversible action. Operands may be fetched and processed, but no results may be stored until the branch is decoded. If the choice was wrong, it can be abandoned and the alternate branch can be processed.

This method works reasonably well if the obvious branch is usually right. When coding for such pipelined CPU's, care should be taken to code branches (especially error transfers) so that the "straight through" path is the one usually taken. Of course, unnecessary branching should be avoided.

Another possibility is to restructure programs so that fewer branches are present, such as by "unrolling" certain types of loops. This can be done by optimizing compilers or, in some cases, by the hardware itself.

A widely-used strategy in much current architecture is some type of **branch prediction.** This may be based on information provided by the compiler or on statistics collected by the hardware. The goal in any case is to make the best guess as to whether or not a particular branch will be taken, and to use this guess to continue the pipeline.

A more costly solution occasionally used is to split the pipeline and begin processing **both** branches. This idea is receiving new attention in some of the newest processors.

## Instruction Level Parallelism

Computer designers and computer architects have been striving to improve uniprocessor computer performance since the first computer was designed. The most significant advances in uniprocessor performance have come from exploiting advances in implementation technology. Architectural innovations have also played a part, and one of the most significant of these over The last decade has been the rediscovery of RISC architectures. Now that RISC architectures have gained acceptance both in scientific and marketing circles, computer architects have been thinking of new ways to improve uniprocessor performance. Many of these proposals such as VLIW, superscalar, and even relatively old ideas such as vector processing try to improve computer performance by exploiting instruction-level parallelism. They take advantage of this parallelism by issuing more than one instruction per cycle explicitly (as in VLIW or superscalar machines) or implicitly (as in vector machines).

The amount of instruction-level parallelism varies widely depending on the type of code being executed. When we consider uniprocessor performance improvements due to exploitation of instruction-level parallelism, it is important to keep in mind the type of application environment. If the applications are dominated by highly parallel code (e.g., weather forecasting), any of a number of different parallel computers (e.g., vector, MIMD) would improve application performance. However, if the dominant applications have little instruction-level parallelism (e.g., compilers, editors, event-driven simulators, lisp interpreters), the performance improvements will be much smaller

## TYPES OF PARALLELISM

There are three main types of parallelism.

### Instruction Level Parallelism

Instruction level parallelism (ILP) takes advantage of sequences of instructions that require different functional units (such as the load unit, ALU, FP multiplier, etc). Different architectures approach this in different ways, but the idea is to have these non-dependent instructions executing simultaneously to keep the functional units busy as often as possible.

### Data Level Parallelism

Data level parallelism (DLP) is more of a special case than instruction level parallelism. DLP to the act of performing the same operation on multiple datum simultaneously. A classic example of DLP is performing an operation on an image in which processing each pixel is independent from the ones around it (such as brightening). This type of image processing lends itself well to having multiple pixels modified simultaneously using the same modification function. Other types of operations that allow the exploitation of DLP are matrix, array, and vector processing.

### Thread Level Parallelism

Thread level parallelism (TLP) is the act of running multiple flows of execution of a single process simultaneously. TLP is most often found in applications that need to run

independent, unrelated tasks (such as computing, memory accesses, and IO) simultaneously. These types of applications are often found on machines that have a high workload, such as web servers. TLP is a popular ground for current research due to the rising popularity of multi-core and multi- processor systems, which allow for different threads to truly execute in parallel.

**INSTRUCTION LEVEL PARALLELISM**

**DEFINITION**

Abbreviated as ILP, Instruction- Level Parallelism is a measurement of the number of operations that can be performed simultaneously in a computer program. Microprocessors exploit ILP by executing multiple instructions from a single program in a single cycle.

**EXPLANTION**

Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. Consider the following program:

1. e = a + b
2. f = c + d
3. g = e * f

Operation 3 depends on the results of operations 1 and 2, so it cannot be calculated until both of them are completed. However, operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously. If we assume that each operation can be completed in one unit of time then these three instructions can be completed in a total of two units of time, giving an ILP of 3/2.

A goal of compiler and processor designers is to identify and take advantage of as much ILP as possible. Ordinary programs are typically written under a sequential execution model where instructions execute one after the other and in the order specified by the programmer. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

How much ILP exists in programs is very application specific. In certain fields, such as graphics and scientific computing the amount can be very large. However, workloads such as cryptography exhibit much less parallelism.