UNIT 2

The Hardware Side: An Introduction, The Core Level, Representing Information, Understanding Numbers, Addresses, Instructions, Registers-A First Look, Embedded Systems-An Instruction Set View, Embedded Systems-A Register View, Register View of a Microprocessor The Hardware Side: Storage Elements and Finite-State Machines (2 hour) The concepts of State and Time, The State Diagram, Finite State Machines- A Theoretical Model.

An Introduction

As we know the **essential elements** for an embedded system are hardware, software and firmware and each will bring out strengths and weaknesses of the embedded system. Here we study about the high level structure and components of the embedded systems are hardware and computing core of an embedded application. That core is usually manifest as a microprocessor, microcomputer, and microcontroller.

In today's high-tech and changing world, we can put together a **working hierarchy** of hardware components. At the top, we find VLSI (Very Large-Scale Integrated) circuits comprising significant pieces of functionality: microprocessors, microcontrollers, FPGAs (Field Programmable Gate Arrays), (C)PLDs ((Complex) Programmable Logic Devices), and ASICs (Application Specific Integrated Circuits). Perhaps we could include memories as well. At the next level down, we find MSI (Medium-Scale Integrated) circuits, which bring smaller, yet complete, pieces of functionality. Going down one more step, we have SSI (Small-Scale Integrated) circuits. At the very bottom, we have the electrical signals we use to represent our data and control information and the other signals that come into our system as noise or other unwanted signals.

Throughout our studies on the hardware side, we will utilize the **Verilog modelling language** to enable us to test, confirm, and demonstrate the viability of our designs prior to committing to hardware. The language will enable us to work at various levels of detail at the **top or behavioral level, we can confirm high-level functionality**, and at the **lower level or structural level, we can confirm details of timing, scheduling, and control**. Facility at both levels is essential today.

Our study of the hardware side of embedded systems begins with a high-level view of the computing core of the system. We will expand and refine that view to include a detailed discussion of the hardware (and its interaction with the software) both inside and outside of that core. Figure 2.1 illustrates the sequence we follow.

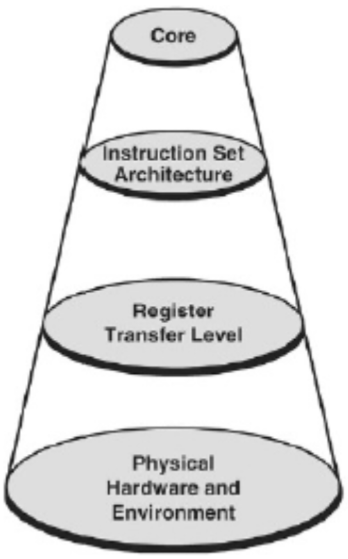

Core
Instruction Set Architecture
Register Transfer Level
Physical Hardware and Environment

Figure 2.1 Exploring Embedded Systems

The **computing core is the central hardware component** in any modern embedded application. It often appears as a microprocessor, microcomputer, or microcontroller. Occasionally, it may appear as a custom-designed VLSI circuit or FPGA. It interacts with and utilizes the remaining components of the system to implement the required application. Such actions are **under the control of a set of software and firmware instructions**. Information and data come into the system from the surrounding environment and from the application. These data are processed according to the software instructions into signals that are sent back out of the system to the application. The software and firmware instructions, as well as signals coming into or going out of the system, are stored in memory.

The Core Level

A model comprising four **major functional blocks** (input, output, memory, and datapath and control) depicting the embedded hardware core and the high level signal flow. These are illustrated in Figure 2.2.

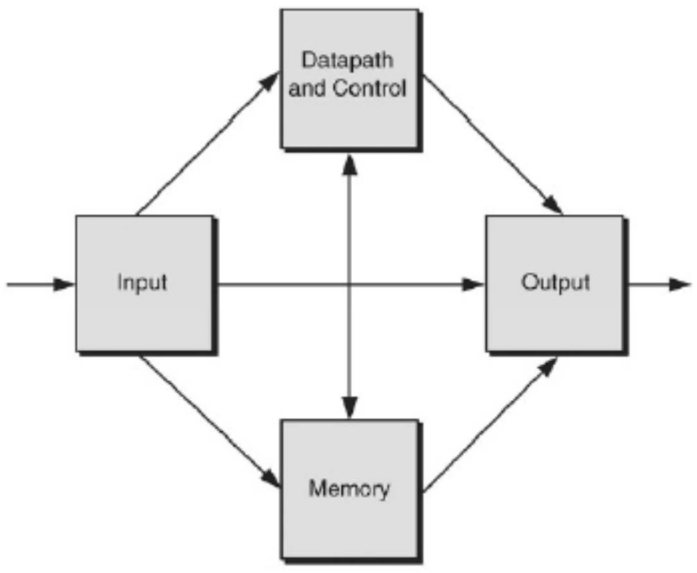


Figure 2.2 Four Major blocks of an embedded hardware

Figure 2.2 Four Major blocks of an embedded hardware, the **memory block serves to hold collections of program instructions** that we call software and firmware as well as to provide short-term storage for input data, output data, and intermediate results of computations. Data as well as other kinds of signals come into the system from the external world through the input block. Once inside of the system, they may be directed to any number of destinations.

The **output block** provides the means to send data or other signals back to the outside world.

The **datapath and control block, more commonly known as the CPU** or central processing unit, coordinates the activities of the system as well as performs computations and data manipulation operations necessary to executing the application.

For the execution, the CPU fetches instructions from memory, interprets them, and then performs the task indicated by the instruction. In doing so, it may retrieve additional data from memory or from the input block. Often, it will also produce information that is sent out of the system.

We move signals into, out of, or throughout the system on paths called **busses**. In their most common implementation, busses are simply collections of wires that are carrying related electrical signals from one place to another.. Signals flowing on the wires making up the busses are classified into three major categories: **address, data and control**.

• The data are the key signals that are being moved around;

• The address signals identify where the data is coming from and where it is going to.

• The control signals specify and coordinate how the data is transported.

The width of a bus, that is, the number of signals or bits that it can carry simultaneously, provides **an indirect measure of how quickly information can be moved**. Transferring 64 bits of data on a bus that is 32 bits wide requires two transfers to move the data. In contrast, a bus that is only 8 bits wide will require eight transfers. Figure 2.3 illustrates moving such a set of data over an 8-bit bus from a source module to a destination module. In the model, each transfer requires one time unit; the process begins at time t0 and completes at time t7. Time increases to the left.

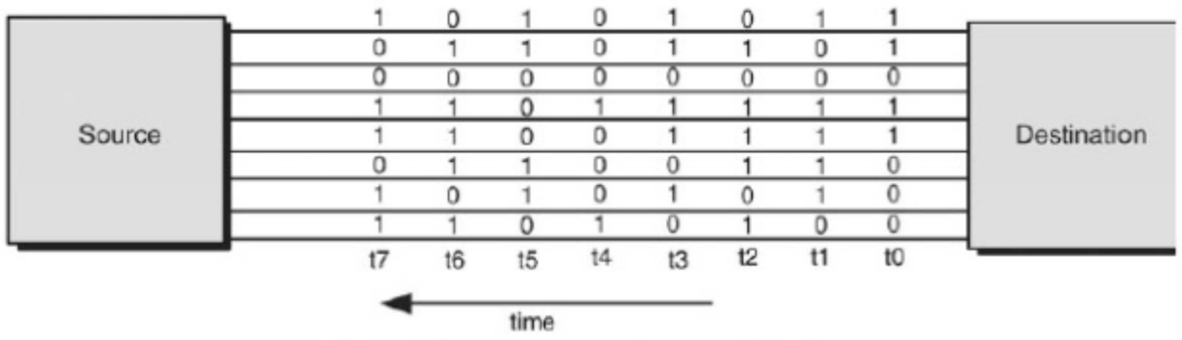


Figure 2.3 Data movement over an 8-bit bus

The source of the transfer is the array of eight bit values; the destination is perhaps a display. In Figure 2.4, we refine the high-level functional diagram to illustrate a typical bus configuration comprising the address, data, and control lines.

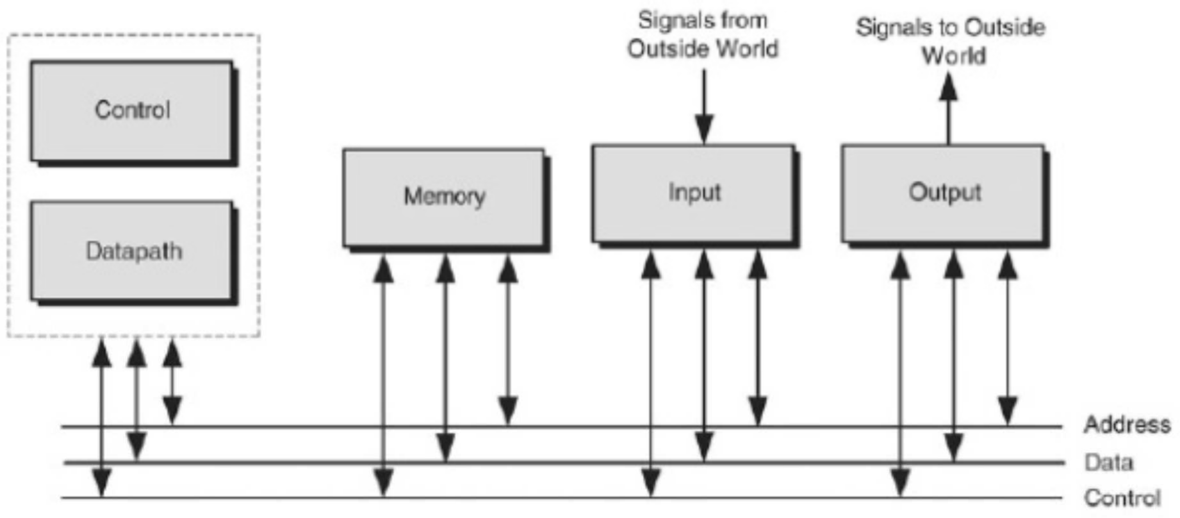


Figure 2.4 A typical bus structure comprising address, data and control signals

None of the busses is required to have the same number of lines. To avoid cluttering a drawing by including all of the signals or conducting paths that make up a bus, we will often label the bus width using the annotation / bus width as illustrated in Figure 2.5 In this example, the address bus is shown to have 18 signals, the data bus 16, and the control bus 7.

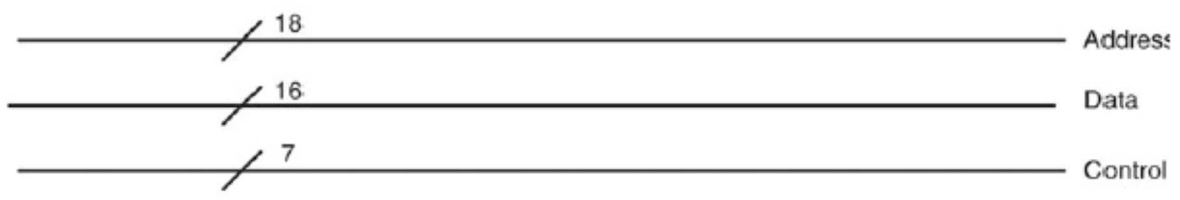


Figure 2.5 Identifying the number of signals in a bus

The microprocessor

A microprocessor is a device that will **receive inputs and process the data** and gives the output. It is often simply referred to as a CPU or datapath. Microprocessors **differ in complexity,** power consumption, and cost. Today microprocessors range from devices with only a few thousand transistors at cost of a dollar or less to units with 5 to 10 million transistors at a cost of several thousand dollars.

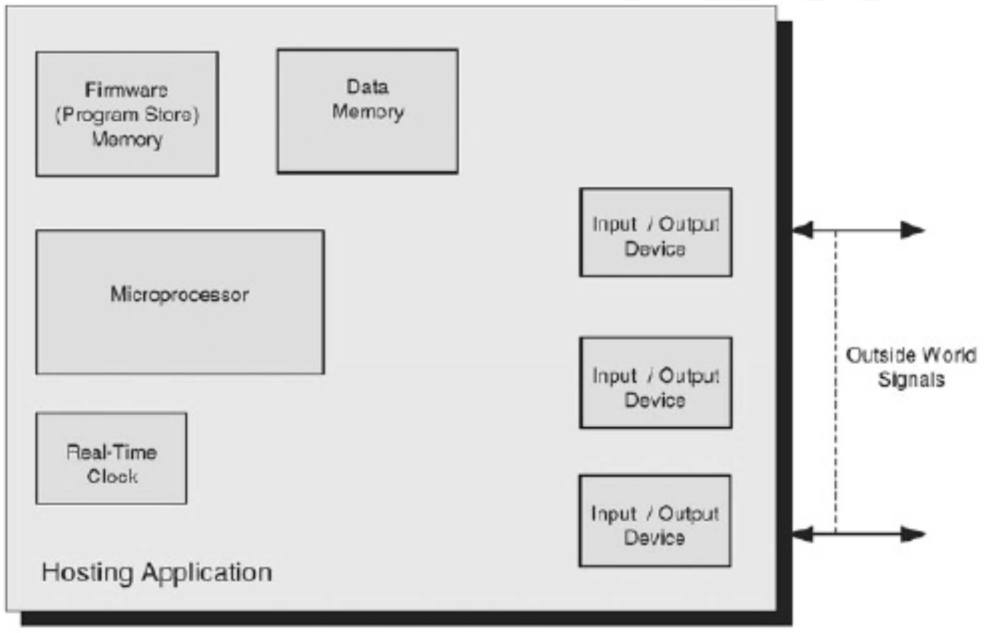


Figure 2.6 A block diagram for a µp based system

The figure 2.6 shows the microprocessor based embedded system, it clearly shows that the **microprocessor solely cannot act as an embedded system** because it needs a support from other peripherals so different blocks such as ROM memory to store the firmware, RAM memory to store the data, Real Time Clock to strictly follow the time constraints and input and output peripherals.

The microcomputer

The microcomputer is a **complete computer system that uses a microprocessor as a CPU**, typically a microcomputer will also utilize numerous other large scale integrated circuits to provide necessary functionality. The complexity of the microcomputer varies from simple units tha ate implemented on a chip along with the small amount of on-chip memory and elementary I/O system to the complex that will have a microprocessor with a wide array of powerful peripheral circuitry.

The Microcontroller

The block diagram of the microcontroller based embedded system is shown in figure 2.7 below. Basically microcontroller it receives the input from the external world and does some process on the data and the **output of the microcontroller is used to control some circuitry** which is connected in the periphery of the system.

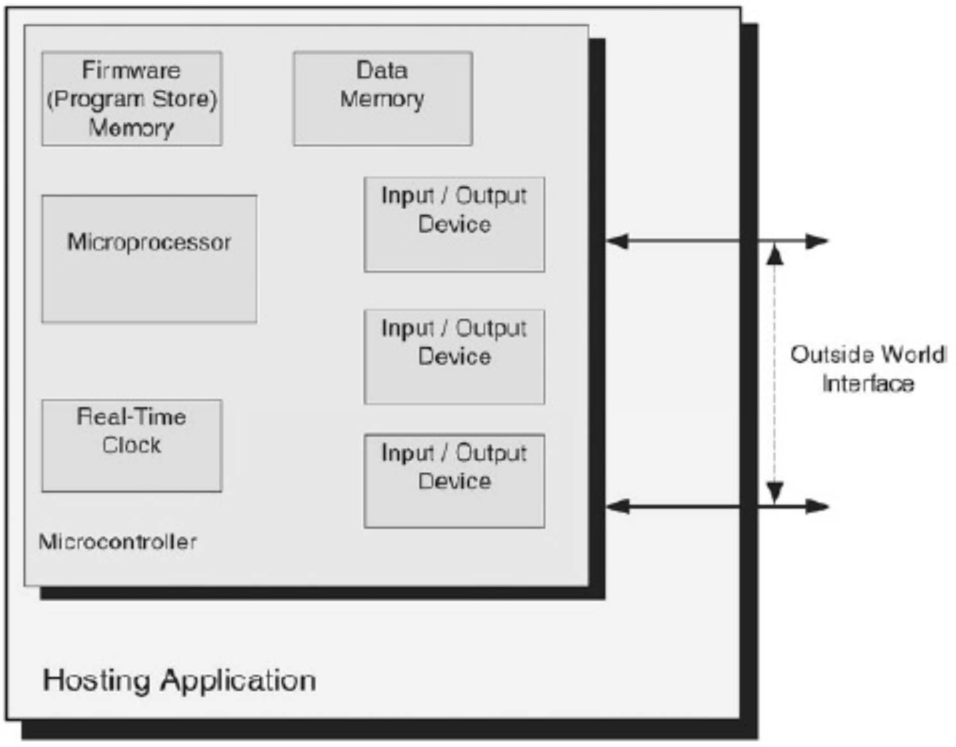


Figure 2.7 A block diagram for microcontroller based system

The microcontroller **combines microprocessor and rich collection of peripherals** such as timers, data converters, serial and parallel communications, a direct memory access into a single integrated circuit. If the designer feels that the inbuilt memory is not sufficient he can add an external memory to satisfy the requirements.

The Digital Signal Processor

It is a **single purpose microprocessor** most widely used in embedded applications. The DSP is typically used in conjunction with a general purpose processor **to perform specialized tasks** such as image, speech, audio, or video processing and its block diagram is shown in figure 2.8. The input to the DSP basically and analog signal from the external world is received through input block and it will **converted into digital signal** using analog to digital signal for the internal use. After the signal processing the digital signal will be **converted into analog signal** using a digital to analog converter and that signal will be given to the external world.

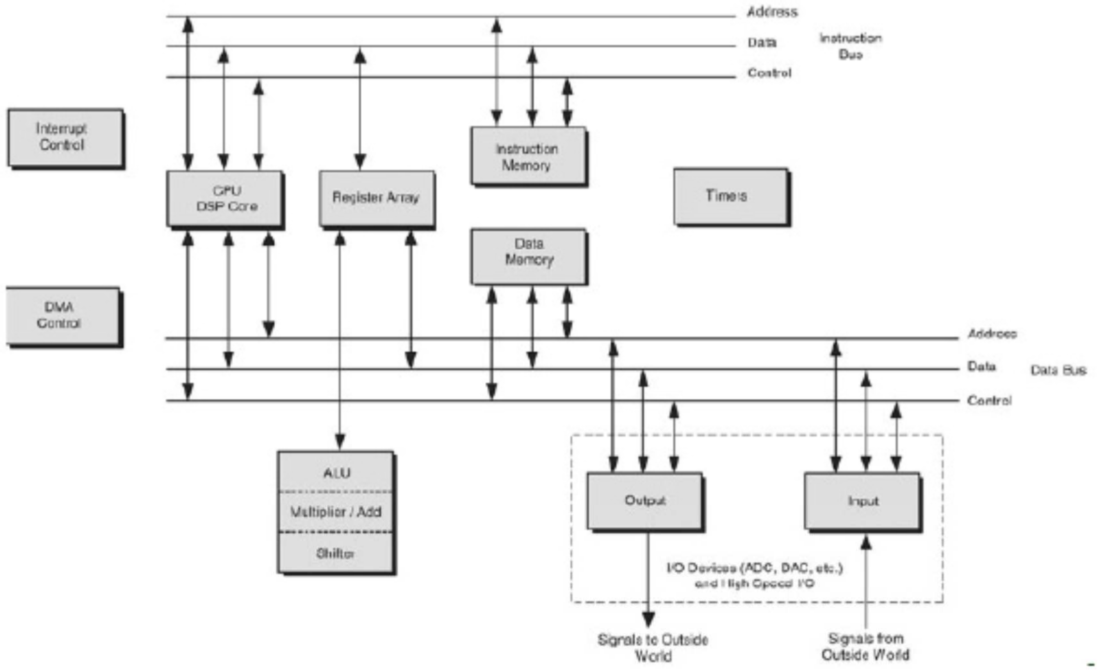


Figure 2.8 A block diagram for a Digital Signal Processor

One of the **major strengths of the DSP** is its ability to perform basic arithmetic computations such as multiply, add and shift at the rate of millions of operations per second.

To support high speed arithmetic, multiply-accumulate unit which will perform multiply and add to the accumulator is performed in a single operation, which is useful in matrix operations is incorporated in DSP.

To facilitate high speed signal processing, the DSP device is architecture as a Harvard rather than the Von Neumann, and has multiple computational units, large number of registers, and wide high bandwidth data busses.

Representing information

A **data is the very important phenomenon** in the embedded systems because throughout the system whether it is a processing, or controlling or manipulation these will be done only on data, in addition to this the data only act as input and output to the systems.

Let us study about how we can represent these kinds of information and with such representation, what limitations we will encounter.

Word Size

The computing engine is classified based on the size of the data it is handling; this is commonly known as word size. Generally a word size in a computing engine refers to the size of an integer. We will assume for the remainder of this study that **we are working with a microprocessor that is using a word size of 32 bits. Such a device is called as 32-bit machine and it has 4 bytes of data**.

The following representations can be used for representing information as shown in figure 2.9 below.

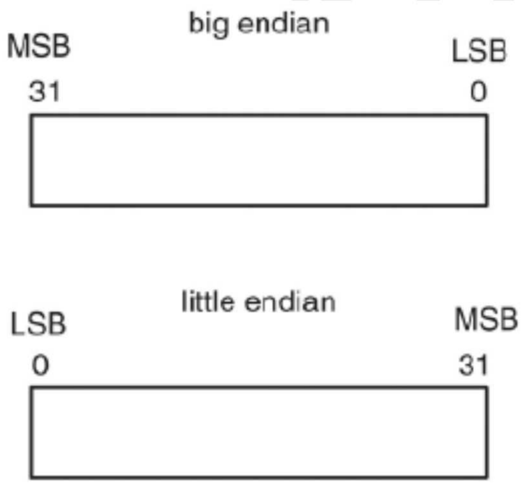


Figure 2.9 big endian Vs Little endian notation

In the above shown figure 2.9 the top structure shows the big endian format, and the below structure show the little endian format, different microprocessors, operating systems, and networks interpret such words in different ways.

In this subject **we will assume a big endian interpretation** unless it is mentioned.

The interpretation of the order of the bits is **just that, an interpretation**. There is nothing inherent in the construction of a number that dictates which end has the MSB or LSB.

Understanding Numbers

In the last section we discussed about the data called word size, here we will study the effects of finite word size on resolution, accuracy, errors and the propagation of errors in different operations.

In an embedded system, the integers and floating point numbers are normally represented as binary values and are stored either in memory or in registers. **The expressive power of any number is dependent on the number of bits in the number**.

Resolution

The following table shows if the four bit word is used to hold a number, what is it corresponding expressive power. If the bits are interpreted as expressing an unsigned integer, the integer may range from 0 to 15; the resolution is 20.

Interpretation	Expressive Power
Integer	0–15
Real	
xxx.x	0–7.5
xx.xx	0–2.75
x.xxx	0–1.6875

Interpreting the number with two bits devoted to the fractional component provides two digits of resolution. That is we can express and resolve a binary number to 2^{-2}.

Example : to represent the number 2.3 using a 4-bit binary number with 2 bits of resolution, the best that one can write is either (10.10) 2.5 or (10.01) 2.25. The error in the expression will be either +0.2 or -0.05.

Because word size limits one's ability to express number, eventually we are going to have to either round or truncate a number inorder to be able to store it in internal memory. Among this which provides the greater accuracy, and which will give the best representation of a measured value?

Let's consider a real number N, following either truncation or rounding of the original number to fit the microprocessor's word size, the number will have a fractional component of n bits. The value of the LSB is 2-n. whether we round or truncate, the resulting number will have an error. The error following the operation is computed as

$$E_R = N_{rounded} - N$$

$$E_T = N_{truncate} - N$$

The range of the errors of the truncation and rounding is as follows

Truncation

$$-2^{-n} < E_T \le 0$$

Rounding

$$-\tfrac{1}{2}2^{-n} < E_R \le \tfrac{1}{2}2^{-n}$$

Propagation of errors

To represent the data we use a big endian or little endian methods and the data should be represented in the available bits. If it does not fits in a available bits then we do truncation and rounding but both results will have an error and this error will propagate throughout the embedded system.

ADDITION

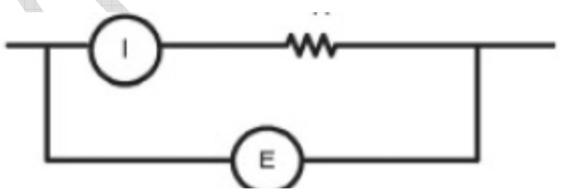
We can express the numbers with an error as

$$N_{1E} = N_1 + E_1$$
$$N_{2E} = N_2 + E_2$$
$$N_{1E} + N_{2E} = (N_1 + E_1) + (N_2 + E_2)$$
$$= N_1 + N_2 + E_1 + E_2$$

Observe that the resulting error is the algebraic sum of the original errors.

This error is not only propagating through the basic mathematic calculations. The below example shows that the electronic circuit will also propagate the error.

Consider the conventional measuring instrument and its measurement details.



$$E = 100\ \text{VDC} \pm 1\%$$
$$I = 10\ \text{A} \pm 1\%$$
$$R = 10\ \Omega \pm 1\%$$

As we know the power can be calculated in three ways

Method 1

$$\begin{aligned} EI &= (100V \pm 1\%) \bullet (10A \pm 1\%) \\ &= ((1000 \pm 10 \bullet 1\%) \pm ((100 \bullet 1\%) \pm (1\% \bullet 1\%))) \\ &= (1000 \pm 1.1) \end{aligned}$$

$$EI \Rightarrow 998.9 \rightarrow 1001.1$$

Method 2

$$\begin{aligned} I^2R &= (10A \pm 1\%) \bullet (10A \pm 1\%) \bullet (10\Omega \pm 1\%) \\ &= (100 \pm (20 \bullet 1\%) \pm (1\% \bullet 1\%)) \bullet (10 \pm 1\%) \\ &= (100 \pm 0.2) \bullet (10 \pm 1\%) \\ &= ((1000 \pm 2) \pm ((100 \bullet 1\%) \pm (0.2 \bullet 1\%))) \\ &= (1000 \pm 3) \end{aligned}$$

$$I^2R \Rightarrow 997 \rightarrow 1003$$

Method 3

$$\begin{aligned} \frac{E^2}{R} &= \frac{(100V \pm 1\%) \bullet (100V \pm 1\%)}{(10\Omega \pm 1\%)} \\ &= \frac{(10000 \pm 2) \bullet (1\% \pm 1\%)}{(10 \pm 1\%)} \end{aligned}$$

$$\frac{E^2}{R} \Rightarrow 908.9 \rightarrow 1111.3$$

The above three results not only yields different answers but differing in magnitude. So **the designer of embedded system should know the cause of the error, rectification of the error and impact of the error as well. If the errors are taken care carefully the safety of the embedded system will be affected**.

Addresses

The data in an embedded system will be stored in the memory; memory is digital hardware and is used to store multiple bits. These data can be accessed by using addresses. In microprocessor **addresses has unique binary pattern**. The number of addresses in a memory depends on the memory locations. The microprocessor does not support negative addresses.

For a word size of 32 bits, the addresses will range (in hex) from 00000000 to FFFFFFFF. Thus, with 32 bits, we have up to 4,294,967,296 unique combinations and therefore that same number of possible addresses.

As data, the addresses also can be represented by using big endian and little endian methods. **The memory is not only used to store the data, it is also used to store the address** of the required data the following examples shows the same.

```
int myVar = 10;
int* myVarPtr = &myVar;  // take the address of myVar
                         // assign it to the pointer variable myVarPtr
```

In the above code we are interested in storing number 10 in the memory location 3000 using pointer function.

When this program executes system will allocate some address 5000 to store the address 3000 of the data. Where the number 10 has to be stored.

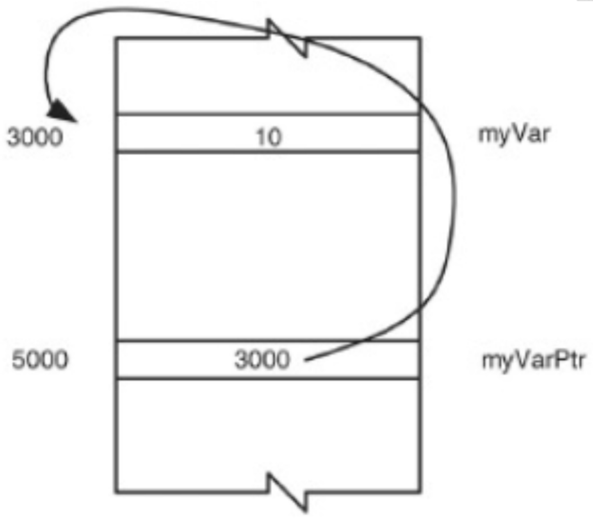


Figure 2.10: using the value of one variable to hold the address of another variable

Instructions

Instructions are used to **direct the hardware of the microprocessor to perform a series of actions**, such as Arithmetic and logic, assignment, reading the data, moving the data in and through the microprocessor.

An instruction consists of operator and each operator requires entities to operate on called **operands**. The no of instructions that an instruction operate on at any time is called **arity** of the operation.

Look at the following example code

X= Y+Z;

Here we have two operations, the addition operation and the assignment operation. First, the addition operation: that operation is performed on two operands, y and z. The addition operator is said to be a binary operator—its arity is two. Now, the assignment operator: the operation is performed by giving x the value returned by the addition operation. It also is performed on two operands, the result of the addition and the operand x—its arity is two as well.

In C and C++, both operators are referred to as binary operators. Operators taking only a single operand have an arity of one and are referred to as unary operators. With one exception, all operators in C and C++ are either unary or binary.

Let’s look at several common C or C++ instructions

1. x = y;

This is an assignment operation, here the value of y is assigned to the target x. to do this we require two operands, thus this operator is called binary operator. Such instruction is called as two operand or two address instruction.

2. z= x+y;

In this instruction two operations are taking place, addition and assignment. Both are binary. If we ignore the transient intermediate result, we see that for we have three operands, x and y are sources and z is a destination. Such an instruction is designated a three-operand or three-address instruction

3. x=x+y;

If we ignore the transient intermediate result as before, we see that for this code we have two operands, x and y are sources and x is also a destination. Such an instruction is designated a two-operand or two-address instruction.

4. ++x or x++

The operation of the above code is pre increment and post increment, the operand required here is one. Such an instruction is designated as a one-operand or one-address instruction.

The previous code fragments have illustrated three classes of instructions we might find in the system software or firmware. These classes are the one, two, or three operand instruction.

Let’s now see how we can interpret the bits in a 32-bit word to reflect such instructions. Any such interpretation will have to support the ability to express both operands and operations as seen in Figure 2.11

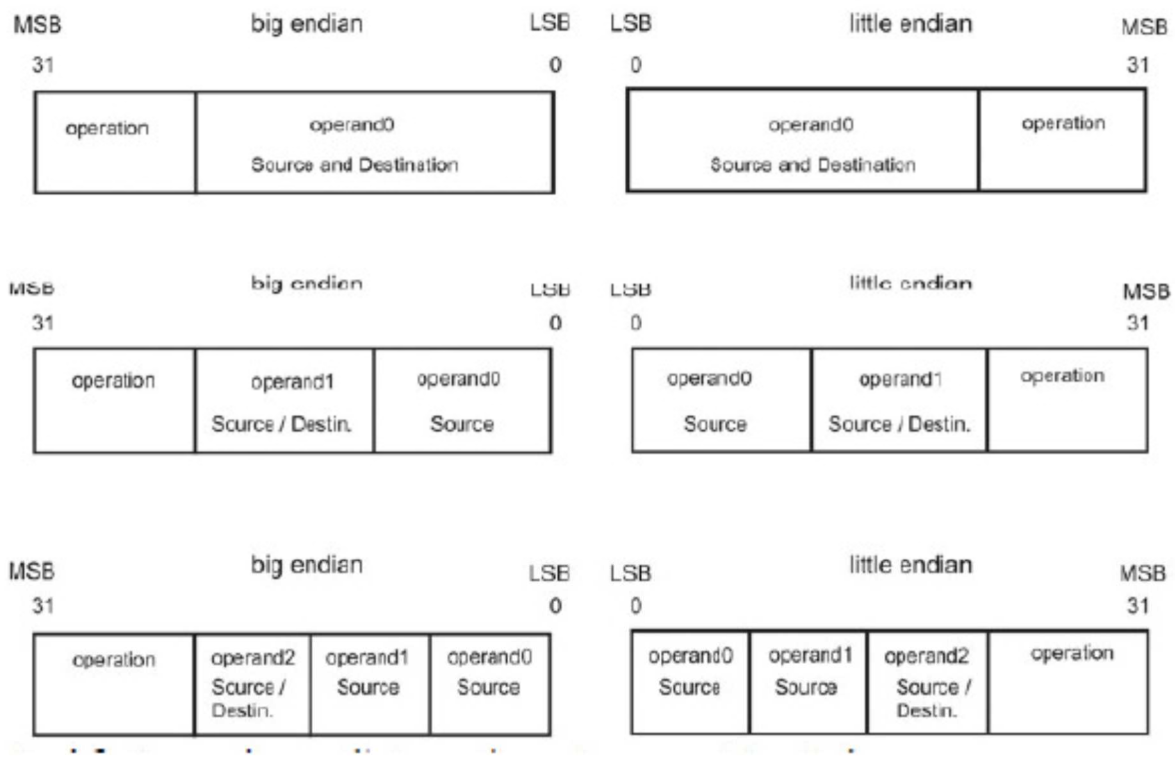


Figure 2.11 Expressing instructions

Register- A first look

The instruction which is used to operate microprocessor is stored in a memory. If the word size of the operand is more it cannot be stored in a memory then the **alternative is to store the operand in the registers** collection of short term memory which is large enough to hold an operands. In prior to execution the data has to be moved in out of memory during fetch and decode operations of the microprocessor.

Some microprocessors employ few register and this processor is called complex instruction set computer **(CISC)**. Microprocessors that has large number of register is called reduced instruction set computer **(RISC)**.

Let’s assume a hypothetical microprocessor with 144 instructions. To permit each instruction to be uniquely identified, we will have to specify that the op-code contains 8 bits since 27 < 144 < 28. Let’s further assume that the microprocessor is designed to include 256 registers. To permit each register to be uniquely identified will also require 8 bits.

Our earlier diagram for the various instruction formats can now be modified to reflect the new interpretation of the operand fields as illustrated in Figure 2.12.

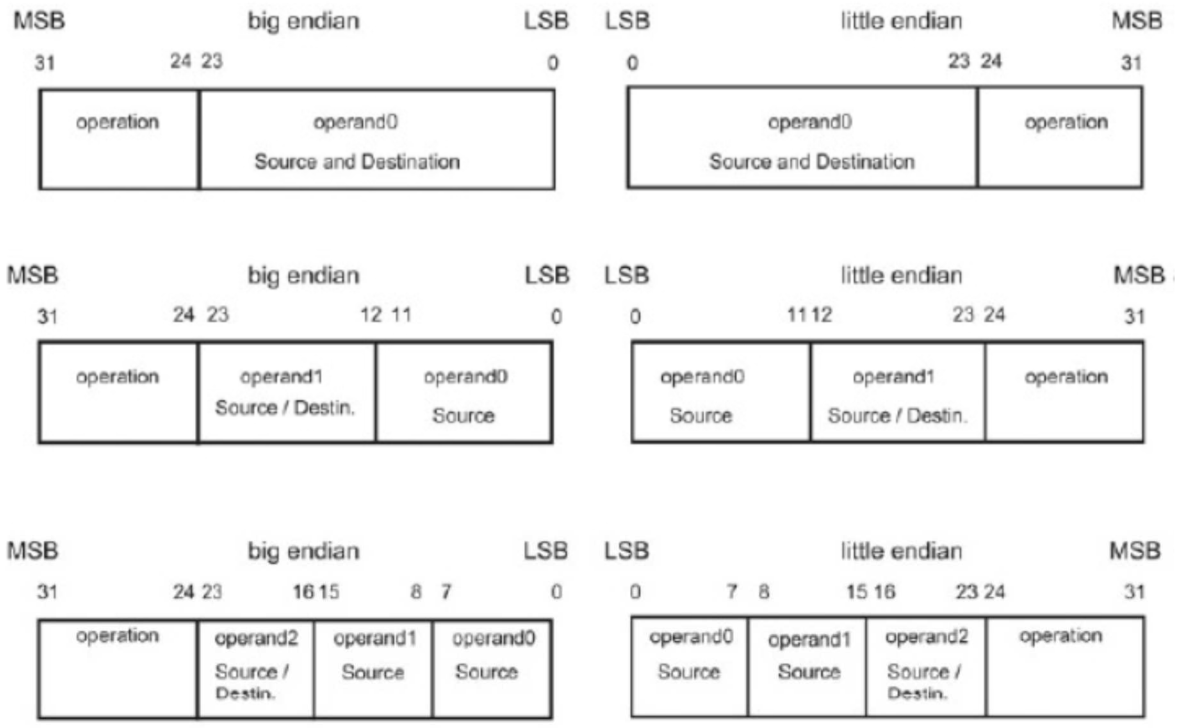


Figure 2.12 Expressing Instructions

The Figure 2.13 summarizes the big endian interpretations of a word in a microprocessor system. The little endian interpretations follow naturally.

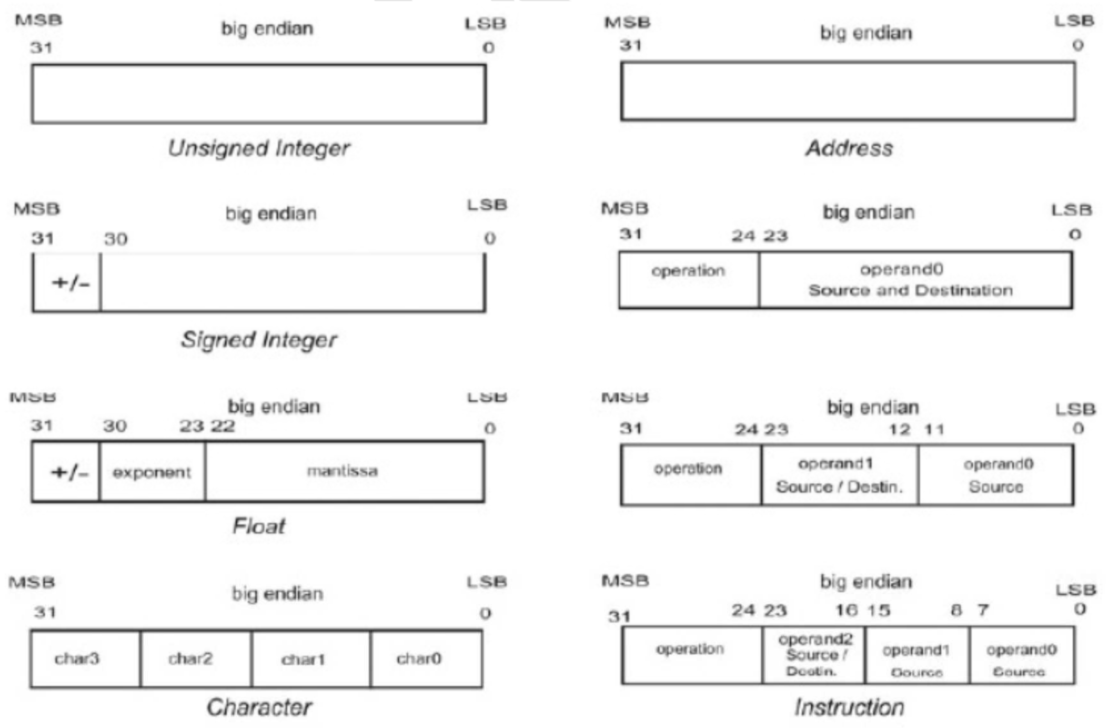


Figure 2.13 : Possible Interpretations of a Set of Bits as Big Endian representation. Little Endian follows similarly

Embedded systems – an instruction set view

The computation engine can be used by developing firmware and executing the same. This **firmware can be developed using high level language, assembly language and combination of assembly and high level language**.

A firmware using assembly language can be developed with reference to the instruction set that is provided with the microprocessor. An assembler is a tool that converts assembly language to machine language in which the firmware is developed with opcodes.

Instruction set- Instruction types

The instruction set specifies the **basic operations** supported by the machine. The objectives of the operations are

• To transfer or store data

• To operate on data

• To make decisions on the data values

The instructions in a computation engine are classified into

• Data transfer,

• Flow of control,

• Arithmetic and Logic.

Data Transfer Instructions

The data transfer instructions are **responsible for moving data inside the processor, bringing the data in, and sending data out.** These instructions consist of source and destination. The source and destination may be a register, memory, or an input or output port. The operations involved in data transfer instructions are illustrated below.

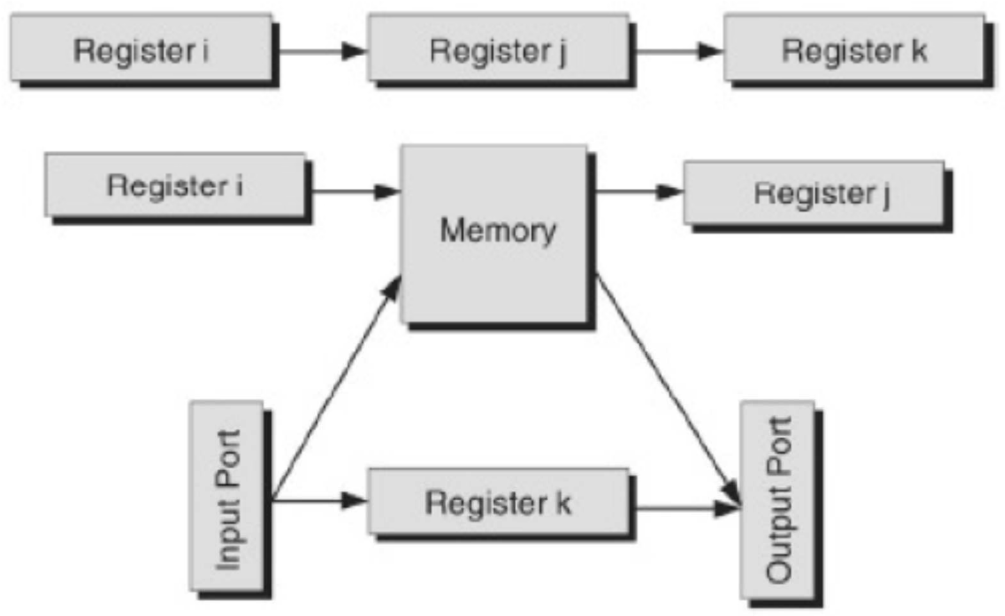
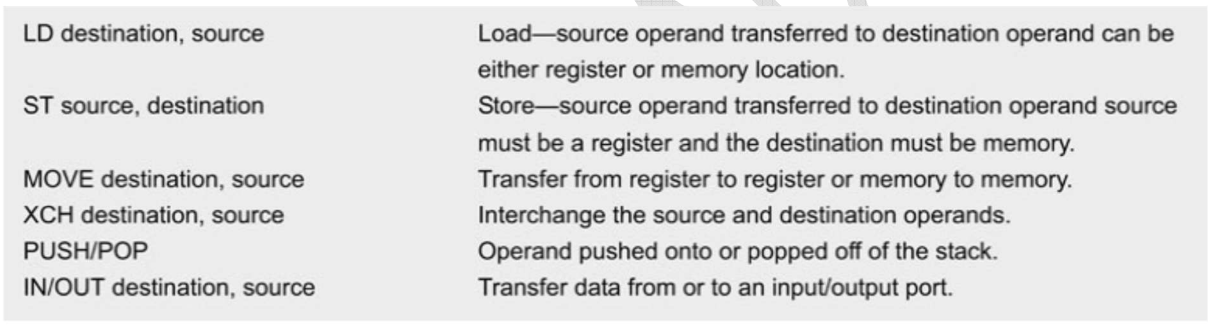


Figure 2.14 data transfer operations

Some of the common instructions used in data transfers are listed below.



LD destination, source	Load—source operand transferred to destination operand can be either register or memory location.
ST source, destination	Store—source operand transferred to destination operand source must be a register and the destination must be memory.
MOVE destination, source	Transfer from register to register or memory to memory.
XCH destination, source	Interchange the source and destination operands.
PUSH/POP	Operand pushed onto or popped off of the stack.
IN/OUT destination, source	Transfer data from or to an input/output port.

Figure 2.15 Data Transfer Instructions

Addressing modes

• Method to specify the location of data

A portion of each operand field is designated as a specification to the hardware as to how to interpret or use the information in the remaining bits of the associated address field. That specification is called the **address mode for the operand**.

The address that is ultimately used to select the operand is called the **effective address**. Addressing modes are included in an instruction in order to offer the designer greater flexibility in accessing data and controlling the flow of the program as it executes. However, some of the address variations can impact flow through a program as well as the execution time for individual instructions. Each is identified by a unique binary bit pattern that is interpreted. The drawings in Figure 1.18 refine our earlier expression of each instruction format to reflect the inclusion of the address mode information.

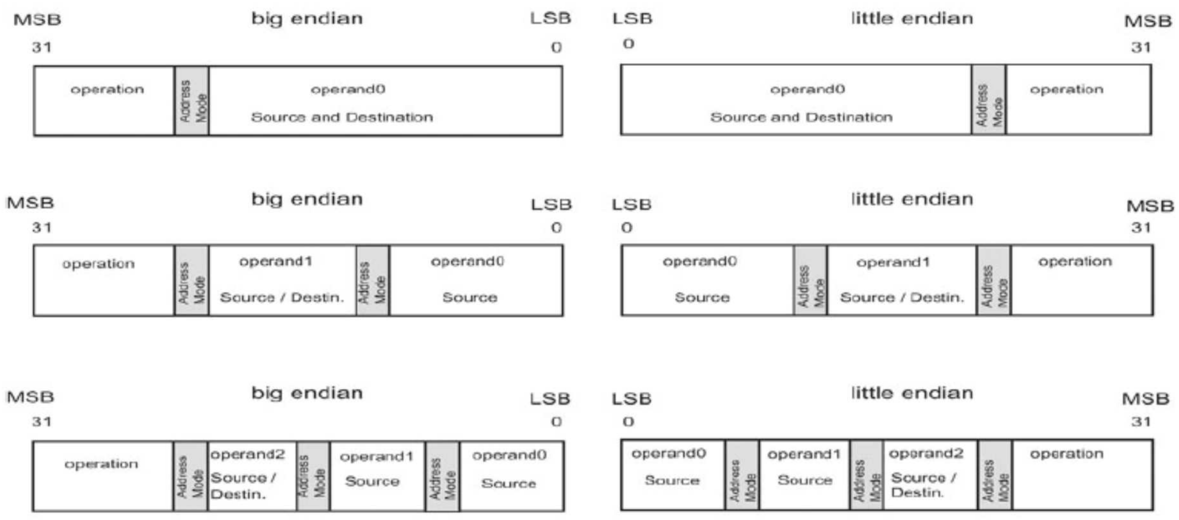


Figure 2.16 enhanced instruction types

Some of the commonly used addressing modes include

• Immediate

• Direct and indirect

• Register Direct and Register indirect

• Indexed

• Program Counter Relative

Immediate Mode

An immediate mode instruction uses one of the **operand fields to hold the value** of the operand rather than a reference to it, as shown in Figure 2.17.

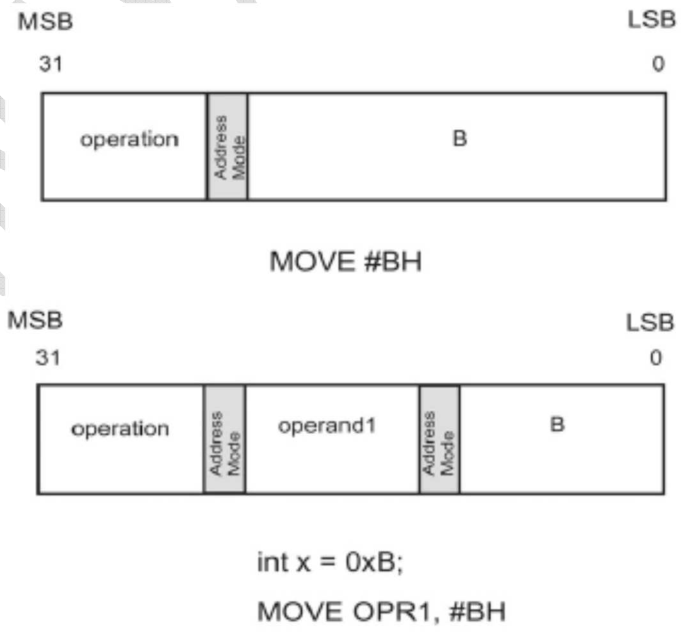


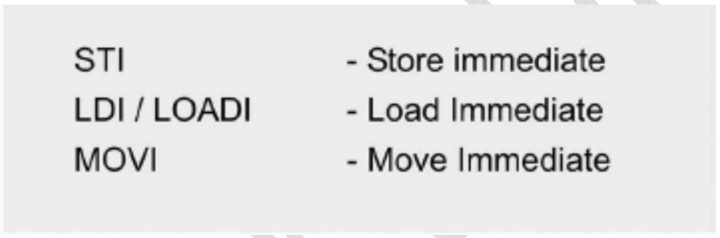
Figure 2.17: Immediate mode instruction formats

The major advantage of such an instruction is that the **number of memory accesses is reduced**. Fetching the instruction retrieves the operand at the same time; there is no need for an additional access. Such a scheme works well if the value of the immediate operand is small. The immediate instruction might appear as a one- or two-operand instruction as illustrated in Figure 2.17.

The one-operand version contains only the immediate value. Without an explicit destination, the target must be implied. Typically, that is the accumulator in the arithmetic and logic unit (ALU).

The two-operand version illustrates the operation at both the C or C++ level and the assembly language level. In the former case, the variable y is declared and initialized to the hex value 0xB. Such an expression is compiled into an assembly language statement of the kind shown.

On some processors, the instruction mnemonic designates that the operation is to use an immediate operand. In such cases, the instruction may be written as illustrated in Figure 2.18.



STI	- Store immediate
LDI / LOADI	- Load Immediate
MOVI	- Move Immediate

Figure 2.18 Variations on the immediate mode instruction

Direct and Indirect Modes

When using the direct and indirect addressing modes, **we are working with operand ddresses** rather than operand values. In both cases, the first level of address information is contained in the instruction. The difference between the two modes is that, in the direct mode, the contents of the operand field in the instruction are the address of the desired operand, whereas in the indirect case, the field contains the address of the address of the operand.

With either mode, the major disadvantage is the **additional memory accesses necessary to retrieve an operand**

In the figure 2.19 two different data transfer operations are shown.

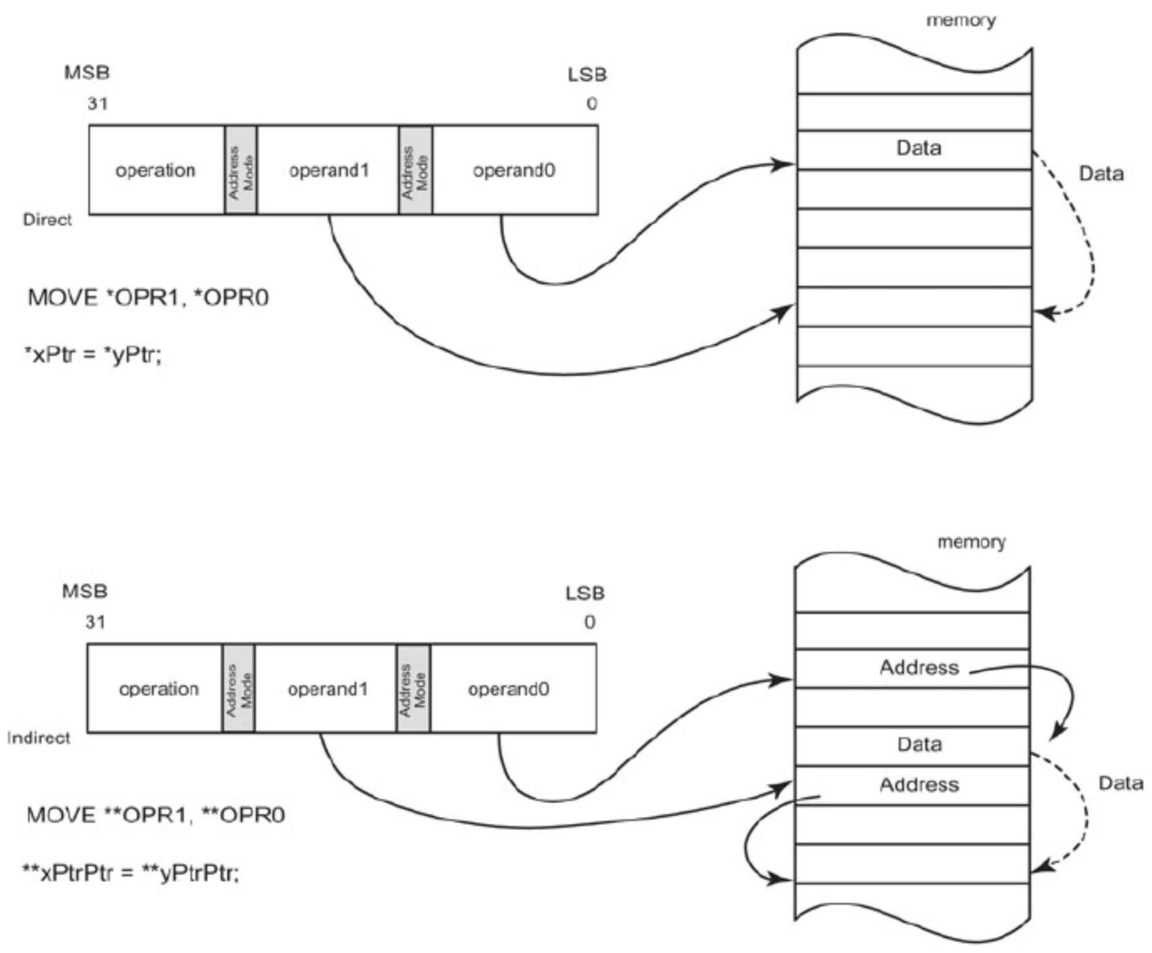


Figure 2.19 Direct an Indirect Instruction Formats

For the direct operation, at the C/C++ level, the value pointed to by one variable, yPtr, is assigned to a second variable pointed to by xPtr. At the assembly language level, the MOVE instruction directs that the contents referenced by operand1 be copied to the location referenced by operand0.

For the indirect operation, at the C/C++ level, the value of one variable, stored in memory and pointed to by the pointer variable yPtrPtr, is assigned to a second variable pointed to by a second pointer variable, xPtrPtr. At the assembly language level, the MOVE instruction now directs that the contents of one memory location serve as the address in memory of the operand that is to be assigned to the location in memory identified by the second operand.

The double ** symbols preceding the operands in the indirect access mode indicate that two levels of indirection are necessary to reach the final operand in memory

Register Direct and Register indirect Modes

In the direct addressing mode the **register contains the value of an operand**. In the indirect addressing the **register contains the address of an operand**. The register indirect allows us to use the pointer functions in high level language.

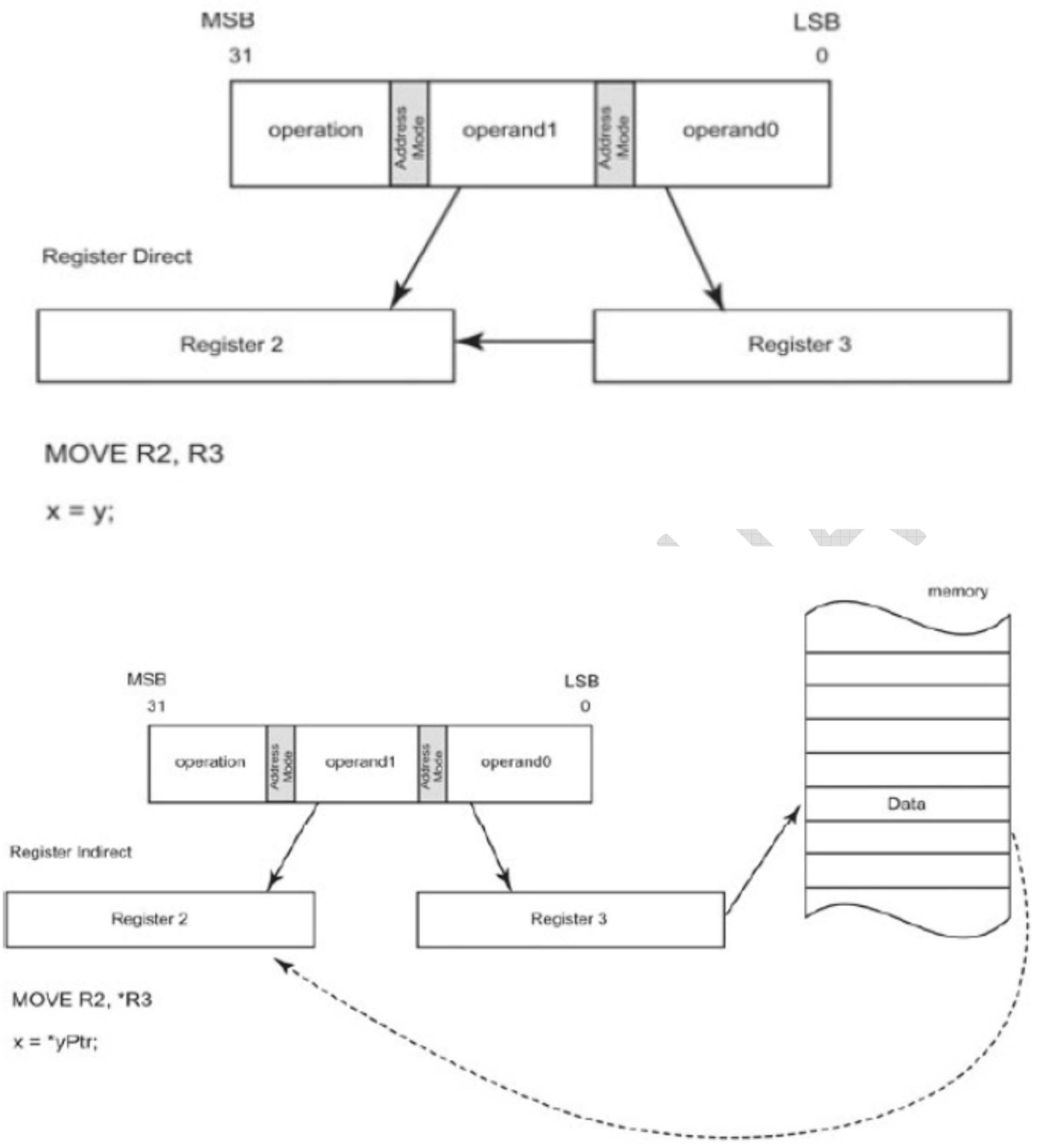


Figure 2.20 Register Indirect data transfer operations

In figure 2.20 two different data transfer operations are shown. For the register direct operation, at the C/C++ level, the value of one variable, y, is assigned to a second variable, x. At the assembly language level, we assume that the values for x and y have previously been stored in registers R2 and R3, respectively. The MOVE instruction directs that the contents of R3 be copied to R2.

For the register indirect operation, at the C/C++ level, the value of one variable, stored in memory and pointed to by the pointer variable yPtr, is assigned to a second variable, x. At the assembly language level, once again we assume that the values for x and yPtr have been previously stored in registers R2 and R3, respectively. The MOVE instruction now directs that the contents of R3 serve as an address into memory; the value of the variable at that address is to be retrieved and to be copied into R2.

The * preceding the second operand in the indirect instruction indicates that the assembler is to set the indirect addressing mode for the instruction. The major disadvantage of indirect addressing is that an additional memory access is necessary to retrieve the operand's value. In contrast, when utilizing direct addressing, the value of the operand is found in the register.

Indexed Mode

The indexed or displacement addressing mode is used **to access container type data structures such as arrays**. The effective address is computed as the **sum of a base address and the contents of the indexing register.**

Note that during execution neither the base address nor the index values are changed.

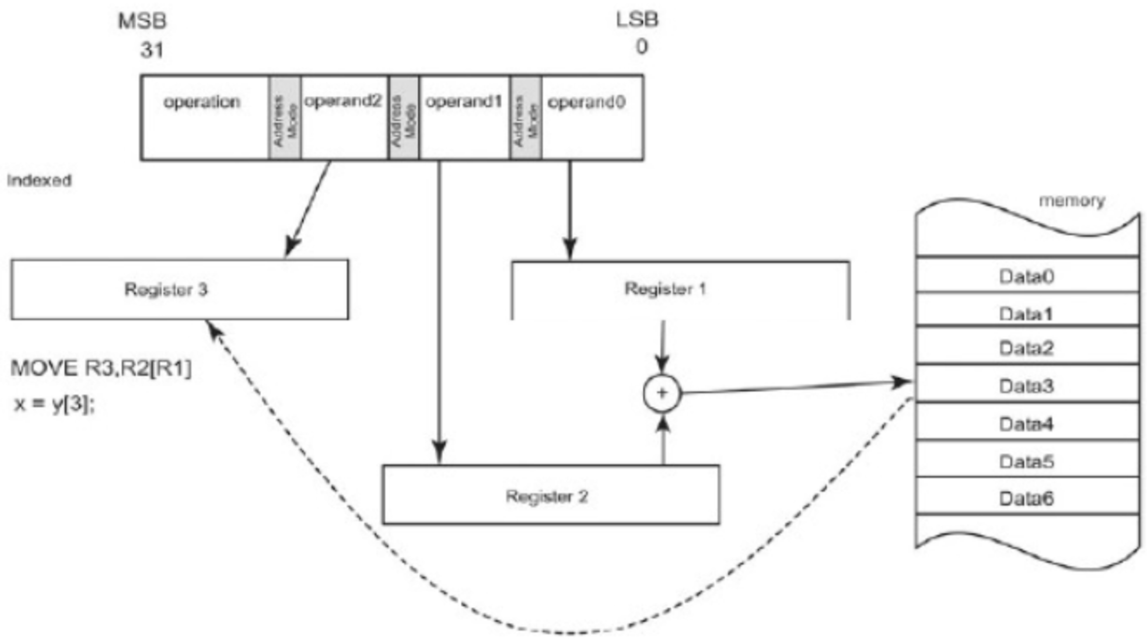


Figure 2.21 Indexed Mode Data Transfer Operations

In the figure 2.21, starting at the C/C++ level, we have an array variable named y and an integer variable x. The variable x is to be assigned the value contained in the fourth element of the array.

At the assembler level, the C/C++ fragment gets translated into a three-operand instruction. The base register, R2, will hold the starting address of the container in this case, the address of the 0th element of the array named y. The value of the variable y contains the address of the variable Data0, the start of the array. Register R1 will serve as the index register—that is, provide the offset. At the assembly level, we assume that the register R1 has already been initialized to the value 3, the offset into the container.

When the instruction is executed, the contents of R1 are added to the contents of R2, giving an address in memory. The value of the data stored in memory at the computed address is retrieved and written into register R3.

The major disadvantage of indexed addressing is the time burden associated with computing the address of the operand and then retrieving the value from memory. Indexing adds a greater burden to system performance than does indirect addressing.

Program counter relative mode

Program counter contains the address of the next instruction to be executed. The program counter relative mode is similar to the indexed addressing mode with **few important differences.**

• The value in the program counter serves as the base address.

• The program counter is assigned the value of the computed effective address.

• The offset that is added to the program counter is a signed number.

Thus, the PC contents following the addition of the offset may refer to an address that is higher (the offset was positive) or lower (the offset was negative) than the original value and its mechanism is shown in figure below.

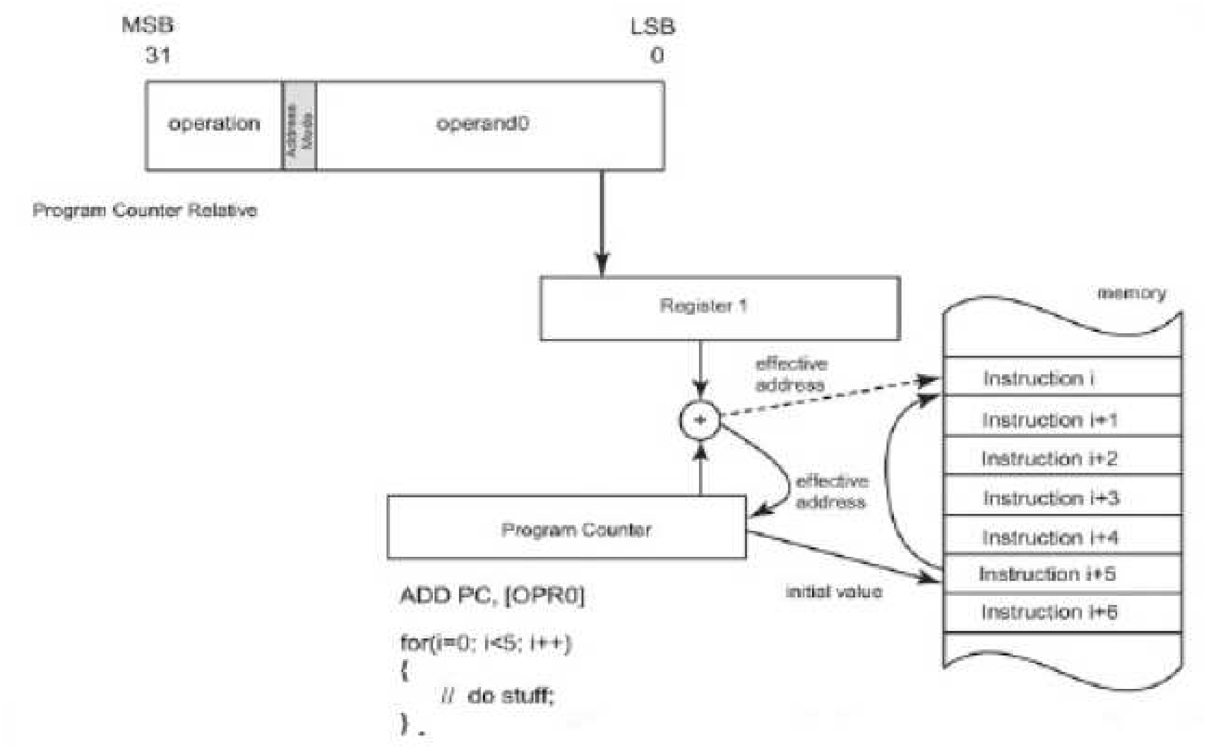


Figure 2.22 Program counter relative Operations

For this instruction, operand0 is serving as the index register and is holding a value that has already been stored in it. The effective address is computed by adding the contents of the register identified by operand0 (R1 in this case) to the contents of the program counter. The program counter contents are then updated to the new value and now refer to the instruction at the computed address.

In the figure 2.22. The C/C++ code fragment illustrates a simple for loop. Following the execution of the body of the loop, the flow must move back to the top of the loop and test the loop variable once again. A negative offset would have to be added to the contents of the PC to effect that movement.

The disadvantage of the PC relative mode is there can be potential degradation of system performance.

Execution flow

The execution flow or control flow of the each instruction is

• sequential

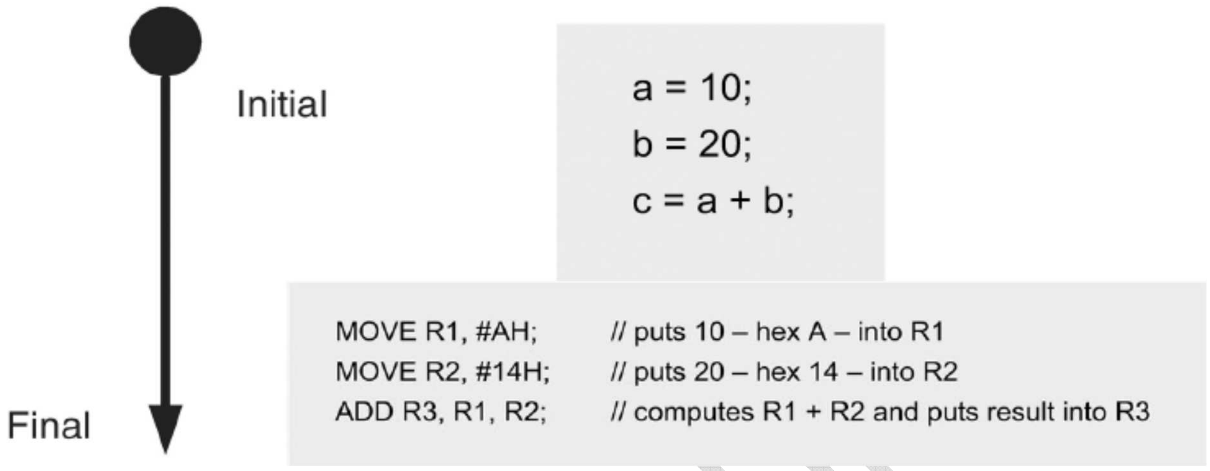
• branch

• loop

• procedure or function call

Sequential flow

Sequential control flow describes the **fundamental movement through a program**. Each instruction contained in the program is executed in sequence, one after another. In the figure 2.23 the sequential flow is shown for both C/C++ and assembler.



```
a = 10;
b = 20;
c = a + b;
```

```
MOVE R1, #AH;      // puts 10 – hex A – into R1
MOVE R2, #14H;     // puts 20 – hex 14 – into R2
ADD R3, R1, R2;    // computes R1 + R2 and puts result into R3
```

Figure 2.23 Sequential Flow

Branch

A branching construct **terminates a sequential flow** of control with a decision point. At such a point, one of several alternate paths for continued execution is taken based on the outcome of a test on some condition. Graphically, such a construct is seen in Figure 2.24.

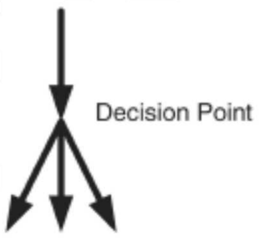
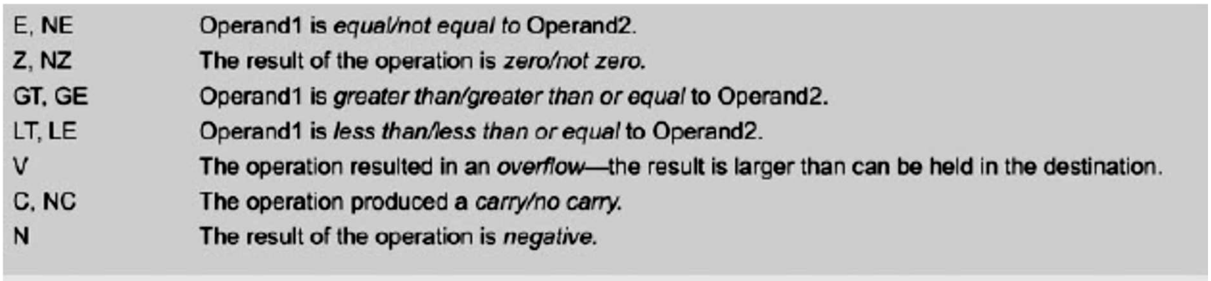


Figure 2.24 the branch construct

The branch construct is used to implement an, if else, switch, or case statement. The branch may be executed unconditionally, in which case the contents of the PC are replaced by the effective address specified by the operand. Alternately, the branch may be taken conditionally based on the side effects of operations performed on data or on several different kinds of comparisons between two variables such as equality, a greater than or less than relationship, a carry from an arithmetic operation, or a variable being equal to or not equal to zero.

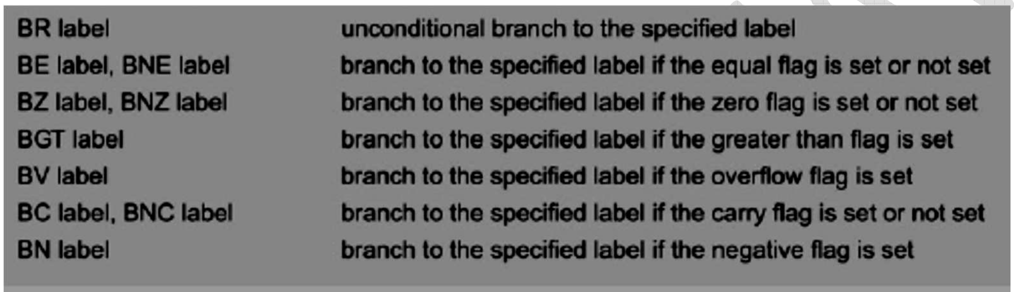
The conditional information is temporarily held as a collection of bits in a flag register or condition code register. The state of each bit in the register is evaluated and potentially changed following the execution of every instruction. Some of the typical conditional codes are listed in the figure 2.25.



E, NE	Operand1 is *equal/not equal to* Operand2.
Z, NZ	The result of the operation is *zero/not zero.*
GT, GE	Operand1 is *greater than/greater than or equal* to Operand2.
LT, LE	Operand1 is *less than/less than or equal* to Operand2.
V	The operation resulted in an *overflow*—the result is larger than can be held in the destination.
C, NC	The operation produced a *carry/no carry.*
N	The result of the operation is *negative.*

Figure 2.25 typical conditional codes

Some of the typical branching instructions are listed in the figure 2.26

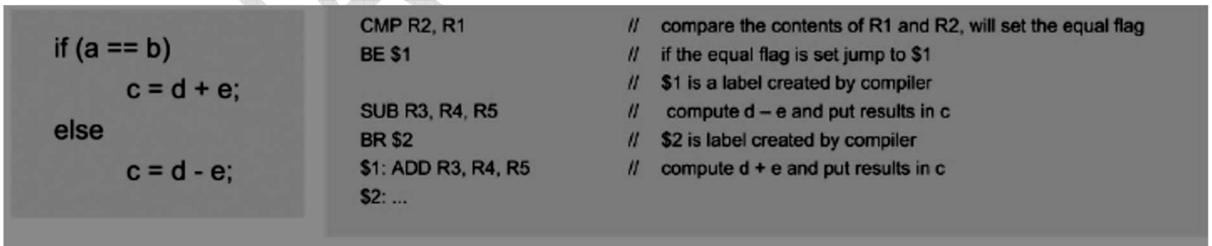


BR label	unconditional branch to the specified label
BE label, BNE label	branch to the specified label if the equal flag is set or not set
BZ label, BNZ label	branch to the specified label if the zero flag is set or not set
BGT label	branch to the specified label if the greater than flag is set
BV label	branch to the specified label if the overflow flag is set
BC label, BNC label	branch to the specified label if the carry flag is set or not set
BN label	branch to the specified label if the negative flag is set

Figure 2.26 typical branching instructions

If-else construct

In the C code in Figure 2.27



```
if (a == b)
        c = d + e;
else
        c = d - e;
```

```
CMP R2, R1          // compare the contents of R1 and R2, will set the equal flag
BE $1               // if the equal flag is set jump to $1
                    // $1 is a label created by compiler
SUB R3, R4, R5      //  compute d – e and put results in c
BR $2               // $2 is label created by compiler
$1: ADD R3, R4, R5  // compute d + e and put results in c
$2: ...
```

Figure 2.27 if-else construct

The two variables are compared. If they are equal, one arithmetic operation is performed; otherwise a second one is executed. The code fragment in Figure 1.32 illustrates the construct in assembler. We assume that the variables a–e have been placed into registers R1–R5. The compiler will create labels $1 and $2 if the original source was written in a high level language or by the designer if the original source was assembler code.

Loop

The loop construct permits the designer to repeatedly execute a set of instructions either forever or until some condition is met. As Figure 2.28 illustrates, the decision to evaluate the body of the loop can be made before the loop is entered (entry condition loop) or after the body of the loop is evaluated (exit condition loop). In the former case, the code may not be executed, whereas in the latter, the code is executed at least once. The loop type of construct is seen in the do, repeat, while, or for statements.

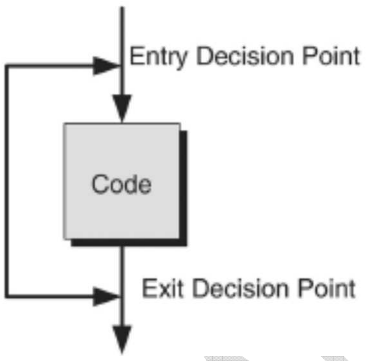
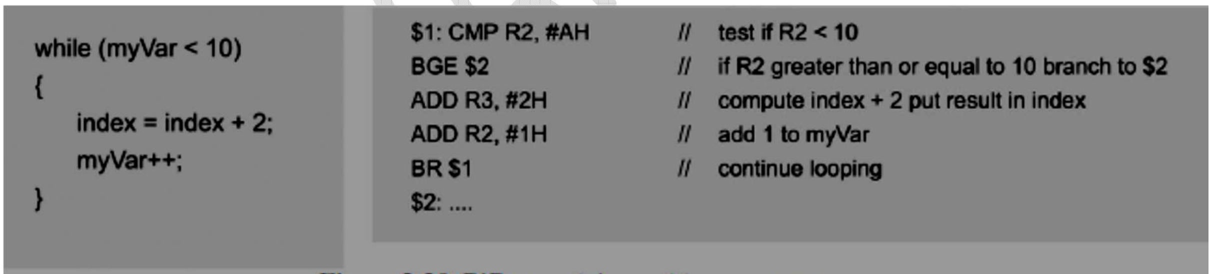


Figure 2.28 the looping construct

The following C/C++ and assembler code fragments in Figure 2.29 illustrate a while loop construct. The body of the loop is continually evaluated as long as the loop variable is less than a specified value. This code fragment implements an entry condition loop. Assume that the variables myVar and index have been placed in R2 and R3, respectively.



```
while (myVar < 10)
{
      index = index + 2;
      myVar++;
}
```

```
$1: CMP R2, #AH        //   test if R2 < 10
BGE $2                 //   if R2 greater than or equal to 10 branch to $2
ADD R3, #2H            //   compute index + 2 put result in index
ADD R2, #1H            //   add 1 to myVar
BR $1                  //   continue looping
$2: ....
```

Figure 2.29 C/C++ and Assembler constructs

Stack

The stack is a data structure that occupies an area in memory. It has finite size and supports several operations. Its structure is similar to an array except that, unlike an array, data can be entered or removed at only one location called the top. The top of the stack is equivalent to the 0th index in an array. When a new piece of data is entered, everything below is pushed down.

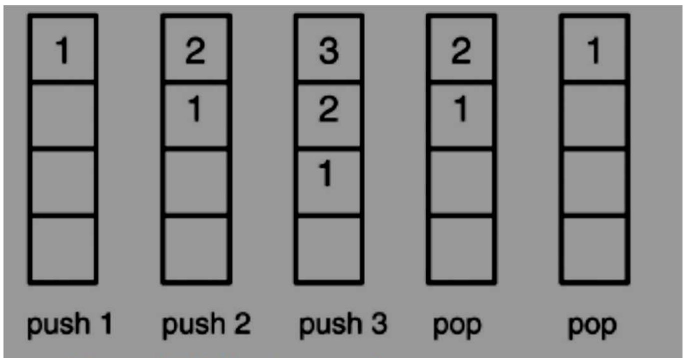


Figure 2.30 Stack operations

Figure 2.30 illustrates a model for the operations for several pieces of data. **Data entry is called a push and data removal is called a pop**. In reality, such a model is impractical because of the time burden in moving every piece of data each time a new entry is made. A more practical implementation adds or removes data at the open end of the structure.

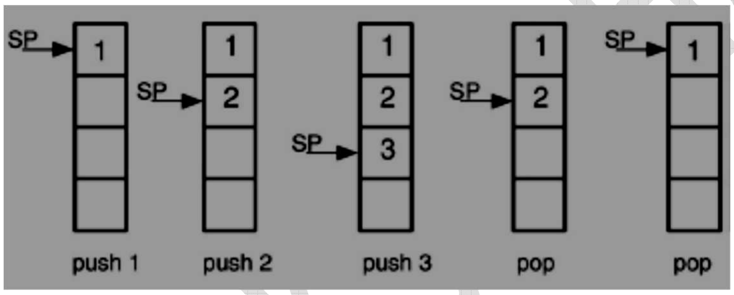


Figure 2.31: Managing the stack pointer

The memory address reflecting the current top of the stack is remembered and modified after each addition or removal. Such an address is called a **stack pointer**. Figure 2.31presents a modified version of the previous diagram and illustrates how the stack pointer is properly managed.

Push

The push operation puts something onto the top of the stack where it is held for later use. Mechanically, the push operation increments the address that is held by the stack pointer to refer to the next empty spot (the new top of the stack) and then writes the data to be stored into the address in memory designated by that address. As we see in Figure 2.31, for ease of implementation, the address contained in the stack pointer is typically incremented from a lower memory address to higher memory address.

Pop

The pop operation takes something off the top of the stack by first retrieving the value in the memory location designated by the stack pointer and then decrementing the address that is held by the stack pointer to refer to the next lower address (the new top of the stack). The retrieved value is returned as the result of the pop operation.

Procedure or Function call

The procedure or function is basically a **subroutine in high level language**. When the function is invoked from the main program, it suspend the execution of main program and change the control of execution to the subroutines and starts executing the program in this context and **return to the main program and resume its execution**. This process is illustrated in the figure 2.32.

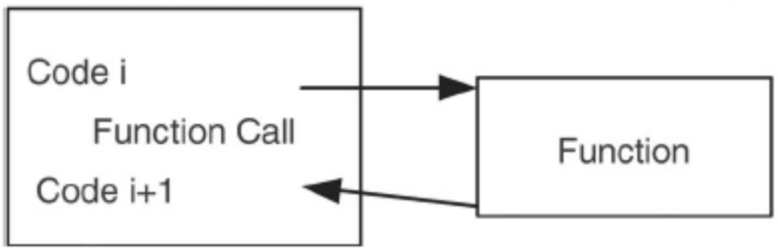
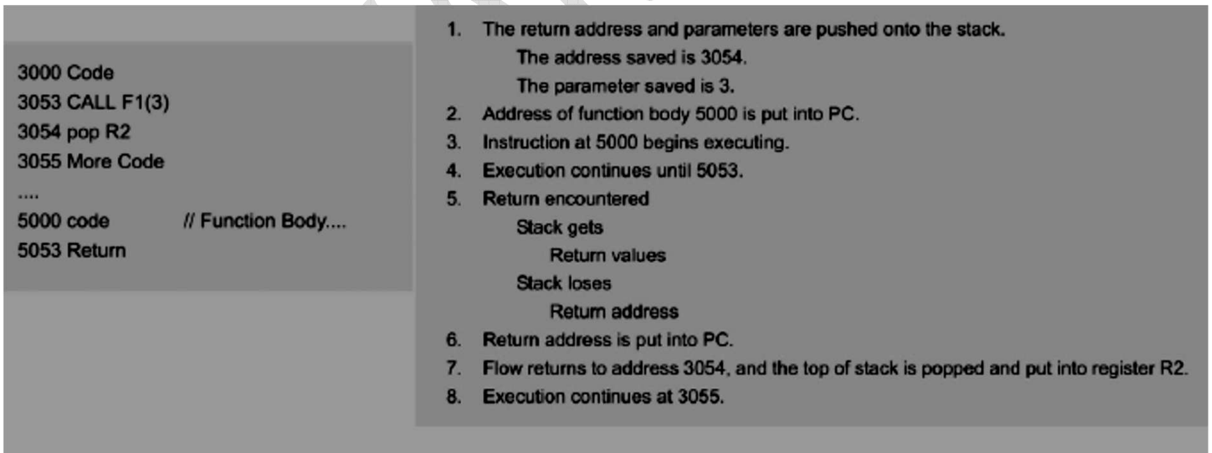


Figure 2.32 the function or procedure call

The process involving in the function and procedure call is. From a high-level point of view, code execution proceeds in a sequential manner until the function call is encountered. Flow of control switches to the function, the code comprising the function body is executed, and flow returns to the original context as seen in Figure 2.33.



```
3000 Code
3053 CALL F1(3)
3054 pop R2
3055 More Code
....
5000 code          // Function Body....
5053 Return
```

1. The return address and parameters are pushed onto the stack.
   The address saved is 3054.
   The parameter saved is 3.
2. Address of function body 5000 is put into PC.
3. Instruction at 5000 begins executing.
4. Execution continues until 5053.
5. Return encountered
   Stack gets
     Return values
   Stack loses
     Return address
6. Return address is put into PC.
7. Flow returns to address 3054, and the top of stack is popped and put into register R2.
8. Execution continues at 3055.

Figure 2.33 Function call construct and flow of control

Arithmetic and Logic

Arithmetic and logical operations are essential elements in affecting what the processor is to do. Such operations are executed by any of several hardware components comprising the ALU (arithmetic and logic unit). Figure 2.34 presents a block diagram for a possible functional ALU architecture.

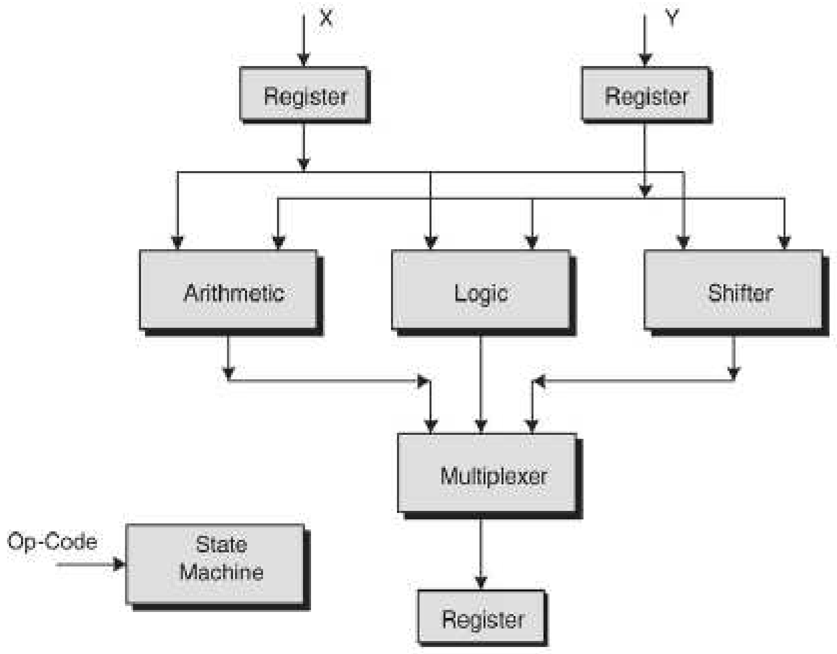


Figure 2.34 An ALU block diagram

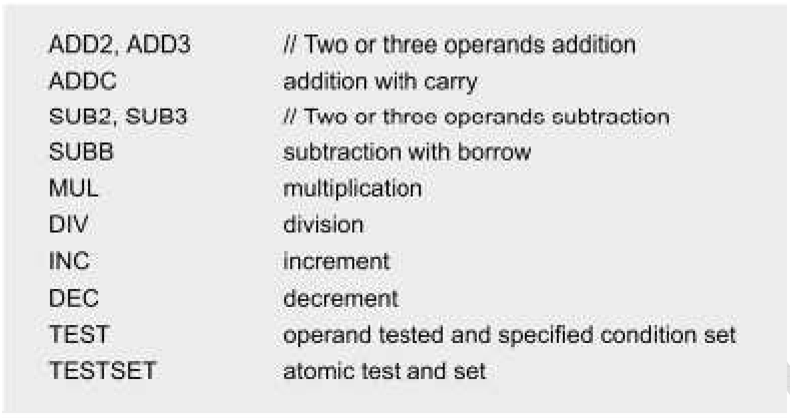
Data is brought into the ALU and held in local registers. The op-code is decoded, the appropriate operation is performed on the selected operand(s), and the result is placed in another local register.

Arithmetic

Typically, the processor will support the four basic arithmetic functions: add, subtract, multiply, and divide. Simpler processors will only implement the first two, relegating the last two to a software implementation by the designer. The add and subtraction operations maybe supported in two versions, with and without carry and borrow.

The last two versions are intended to support double-precision operations. Such an operation is performed in two steps: the first computation holds any carry (borrow) and then utilizes that value as a carry in (borrow in) to the second step. Most such operations are implemented to support integer-only computations. If floating point mathematics is supported, a separate floating point hardware unit may be included. In addition to the four basic functions, the processor may also implement hardware increment and decrement operations.

The typical arithmetic instructions supported by microprocessor is listed in figure 2.35

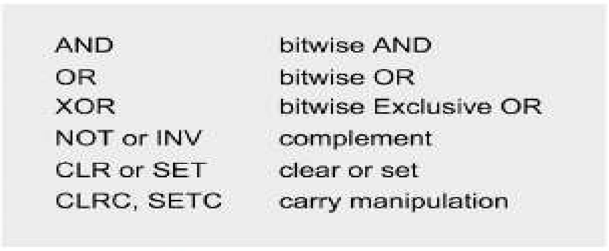


ADD2, ADD3	// Two or three operands addition
ADDC	addition with carry
SUB2, SUB3	// Two or three operands subtraction
SUBB	subtraction with borrow
MUL	multiplication
DIV	division
INC	increment
DEC	decrement
TEST	operand tested and specified condition set
TESTSET	atomic test and set

Figure 2.35 typical arithmetic instructions

Logical operations

Logical operations perform **traditional binary operations on collections of bits or words**. Such operations are particularly useful in embedded applications where bit manipulation is common. Such operations are discussed in detail in our studies of the software side of embedded systems. Typical operations included in the set of logical instructions are illustrated in Figure 2.36



AND	bitwise AND
OR	bitwise OR
XOR	bitwise Exclusive OR
NOT or INV	complement
CLR or SET	clear or set
CLRC, SETC	carry manipulation

figure 2.36 Typical logical instructions

Shift operations

Shift operations typically perform several **different kinds of shifts on collections of bits or words. The major differences concern how the boundary values on either side of the shift are managed**. Typically, three kinds of shift are supported: logical, arithmetic, and rotate. Any of the shifts may be implemented as a shift to the left or to the right.

A logical shift enters a 0 into the position emptied by the shift; the bit on the end is discarded. An arithmetic shift to the right propagates (and preserves) the sign bit into the vacated position; a shift to the left enters 0's on the right-hand side and overwrites the sign bit. The rotate shift circulates the end bit into the vacated bit position on the right- or left-hand side based on a shift to the left or to the right.

Typical shift operations supported by a computation device are listed in the figure 2.37.

SHR operand, count	logical shift right
SHL operand, count	logical shift left
SHRA operand, count	arithmetic shift right
SHLA operand, count	arithmetic shift left
ROR operand, count	rotate right
ROL operand, count	rotate left

Figure 2.37 typical shift operations

Embedded Systems – A register view

As we know the instruction sets are used to operate the microprocessor hardware. The microprocessor hardware consists of control and data path blocks as shown in figure 2.38.

In this figure the data path is responsible for performing required operations in the microprocessor, it consists of different types of registers to store the value temporarily. The control unit control and manage the operations

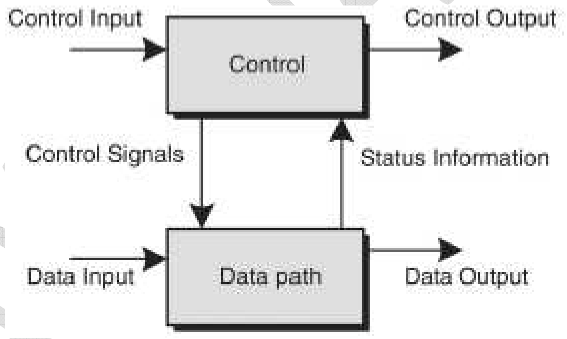


Figure 2.38 a control and data path block diagram

The Basic Register

A register is a sequential digital circuit which is capable of storing single or multiple bits. Here let's take different abstract view of a register shown in figure 2.39 and 2.40.

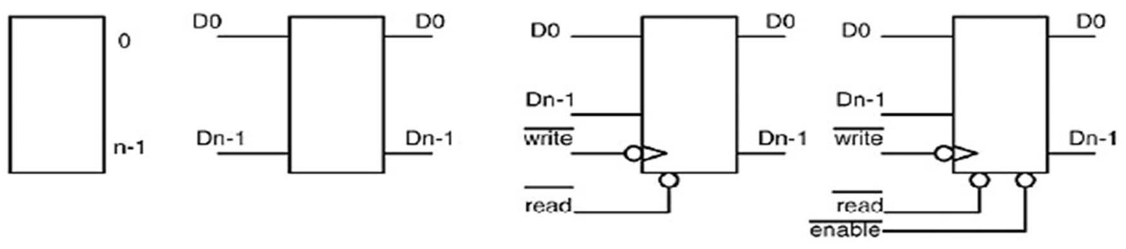


Figure 2.39 the register at several levels of abstractions – Parallel data entry

From the left the first diagram gives the high level abstraction as the size of the register, then the second diagram shows the no of inputs and outputs to the register, the third digram includes two active low inputs to write into and read from registers. The last diagram illustrates the enabling capability of the register operation.

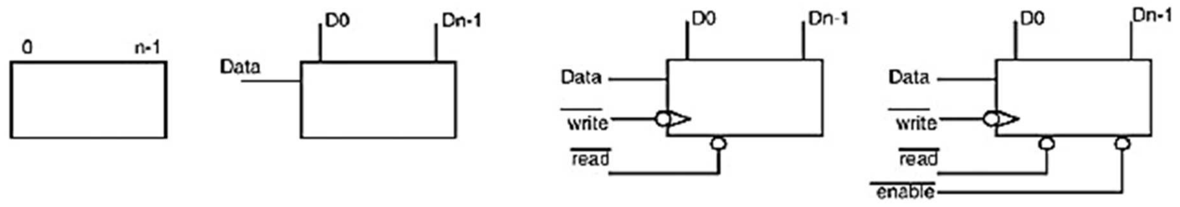


Figure 2.40 the register at several levels of abstractions – serial data entry

Similar to the parallel registers the serial data entry registers are having different abstract levels from the left to right.
Registers support basically two operations

- read
- write

Write to a register

A parallel write operation begins when the data is placed onto the inputs of the register. Following a delay to allow the data to settle on the bus, the write signal is asserted. For a serial write operation, a write signal must accompany each data bit that is entered. In the drawings shown in Figure 2.41, the write signal is asserted low which is typical. Following each write operation, the contents of the register are changed to reflect the new values of the input data.

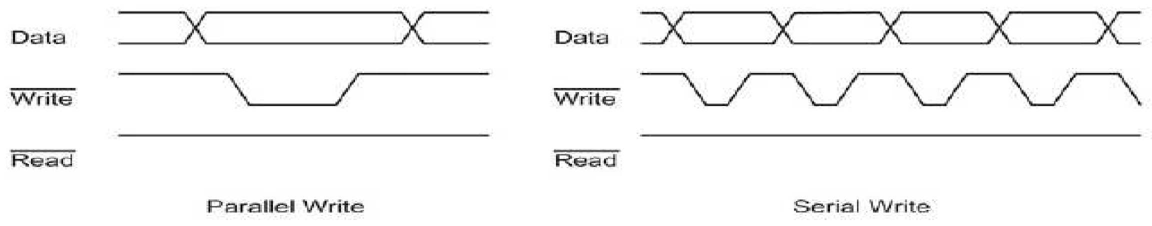


Figure 2.41 writing into a register

Read from a register

The read operation is executed as shown in Figure 2.42. The read signal is issued; following some delay, the data appears on the register output. In this illustration, the read signal is shown as asserted low.

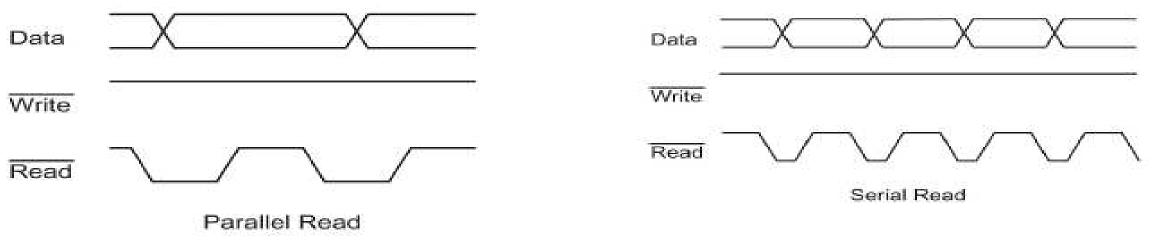


Figure 2.42 Parallel read and serial read

Register View of a Microprocessor

Here let us examine the datapath and control unit for a simple microprocessor at RTL level.

The datapath

Figure 2.43 depicts the architecture of the datapath and the memory interface for a simple microprocessor at the register transfer level.

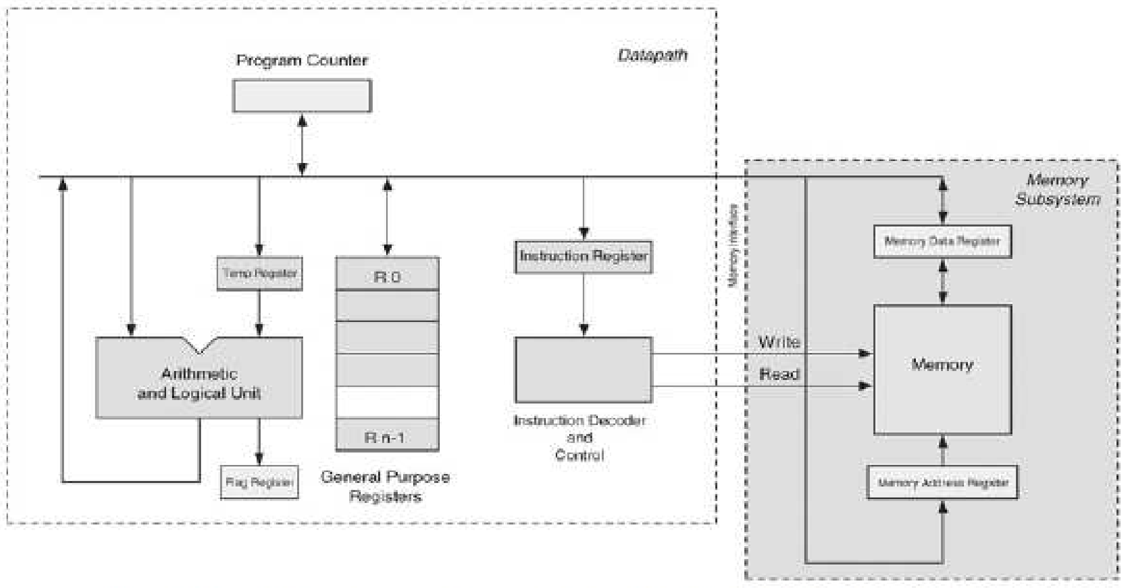


Figure 2.43 RTN Model for a Microprocessor Datapath and Memory Interface

The above figure shows the different registers used in a microprocessor with memory interface. Initially the program counter will point the address of the instructions to be fetched from memory through memory address register. From the memory the data is fetched and given to the memory data register the same is given to the instruction register and instruction decoder will decode to find the purpose of the instruction again the memory is contacted for the required data to execute the instructions. Finally the execution is performed by the Arithmetic and Logic unit and the temporary results are stored in the general purpose registers.

The control

The control of the microprocessor datapath consists of four fundamental operations defined as the instruction cycle. These steps are identified in Figure 2.44, and are further described according to state diagram in 2.45.

Fetch	Fetch instruction
Decode	Decode current instruction
Execute	Execute current instruction
Next	Compute address of next instruction

Figure 2.44 Steps in the Instruction Cycle

Fetch

The fetch operation retrieves an instruction from memory. That instruction is identified by its address, which is the contents of the program counter, PC. Thus, at the ISA level, the fetch operation is written as

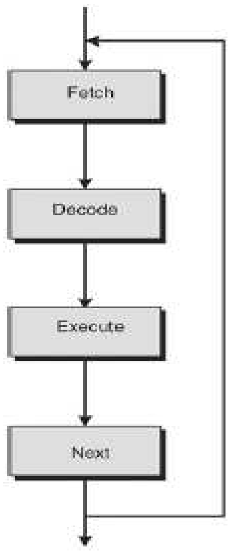


Figure 2.45 The instruction cycle

MOVE IR, *PC;

Move the memory word identified by the address contained in the program counter into the instruction register The first step in the fetch operation places the contents of the program counter (which identifies the address of the next instruction) into the Memory Address Register (MAR). A Read command is issued to the memory, which retrieves the instruction stored in the addressed location and places it into the Memory Data Register (MDR). The contents of the MDR are then transferred to the Instruction Register (IR). At the RTL level, the fetch decomposes into the sequence of steps given in Figure 2.46.

The second step in this sequence executes a READ operation from the specified memory location. The underlying hardware will generate the read control signal and manage the underlying timing.

```
MAR←PC;                  // PC enabled out to bus, MAR captures value
MDR ← Memory[MAR];       // contents of specified memory location placed into MDR
IR ← MDR;                // MDR enabled out to bus, IR captures value
```

Figure 2.46 Components of the fetch Instruction

Decode

The decode step is performed when the op-code field in the instruction is extracted from the instruction and decoded by the Instruction Decoder. That information is forwarded to the control logic, which will initiate the execute portion of the instruction cycle.

Execute

Based on the value contained in the op-code field, the control logic performs the sequence of steps necessary to execute the instruction. Two examples are given in Figure 2.47. Store the contents of a register in a named location in memory. Add the contents of a register to a piece of data stored in memory and place the result back into memory, but at a different location.

```
// C Instruction
        *xPtr = y;
// ISA Level Instruction
        ST *R1, R2;
// RTL Level Instructions
        MAR ← R1;               // R1 enabled out to bus, MAR captures value
        MDR ← Memory[MAR];      // contents of specified memory location placed into MDR
        R2 ← MDR;               // MDR enabled out to bus, R2 captures value
```

```
// C Instruction
        *zPtr = x + *yPtr;
// ISA Level Instruction
        ADD *R3, R1, *R2;
// RTL Level Instructions
// Assume that R2 and R3 already contain the desired addresses in memory
        TR0 ← R1;               // R1 enabled out to bus, TR0 captures value
        MAR ← R2;               // R2 enabled out to bus, MAR captures value
        MDR← Memory[MAR];       // contents of specified memory location placed into MDR
        TR1← TR0 + MDR;         // MDR enabled out to bus, ALU adds TR0 and MDR
                                // places result in TR1
        MAR ← R3;               // R3 enabled out to bus, MAR captures value
        MDR ← TR1;              // TR1 enabled out to bus, MDR captures value
        Memory[MAR] ← MDR;      // contents of MDR placed into specified memory location
```

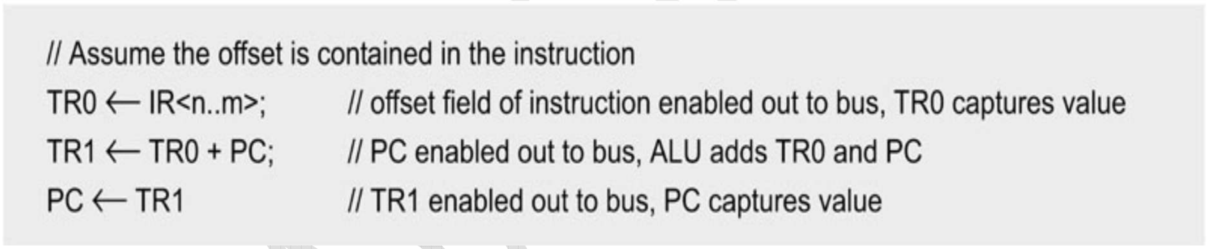
Figure 2.47 An Execute Sequence

Next

The address of the next instruction to be executed is dependent on the type of instruction to be executed and, potentially, on the state of the condition flags as modified by the recently completed instruction. At the end of the day, most reduce to algebraically adding a value to the PC. For short jumps, the displacement may be contained in one of the operand fields of the instruction; for longer jumps, the value may be contained in the memory location following the instruction. Thus, at the ISA level, the several versions of the next operation are written as

ADD PC, offset;

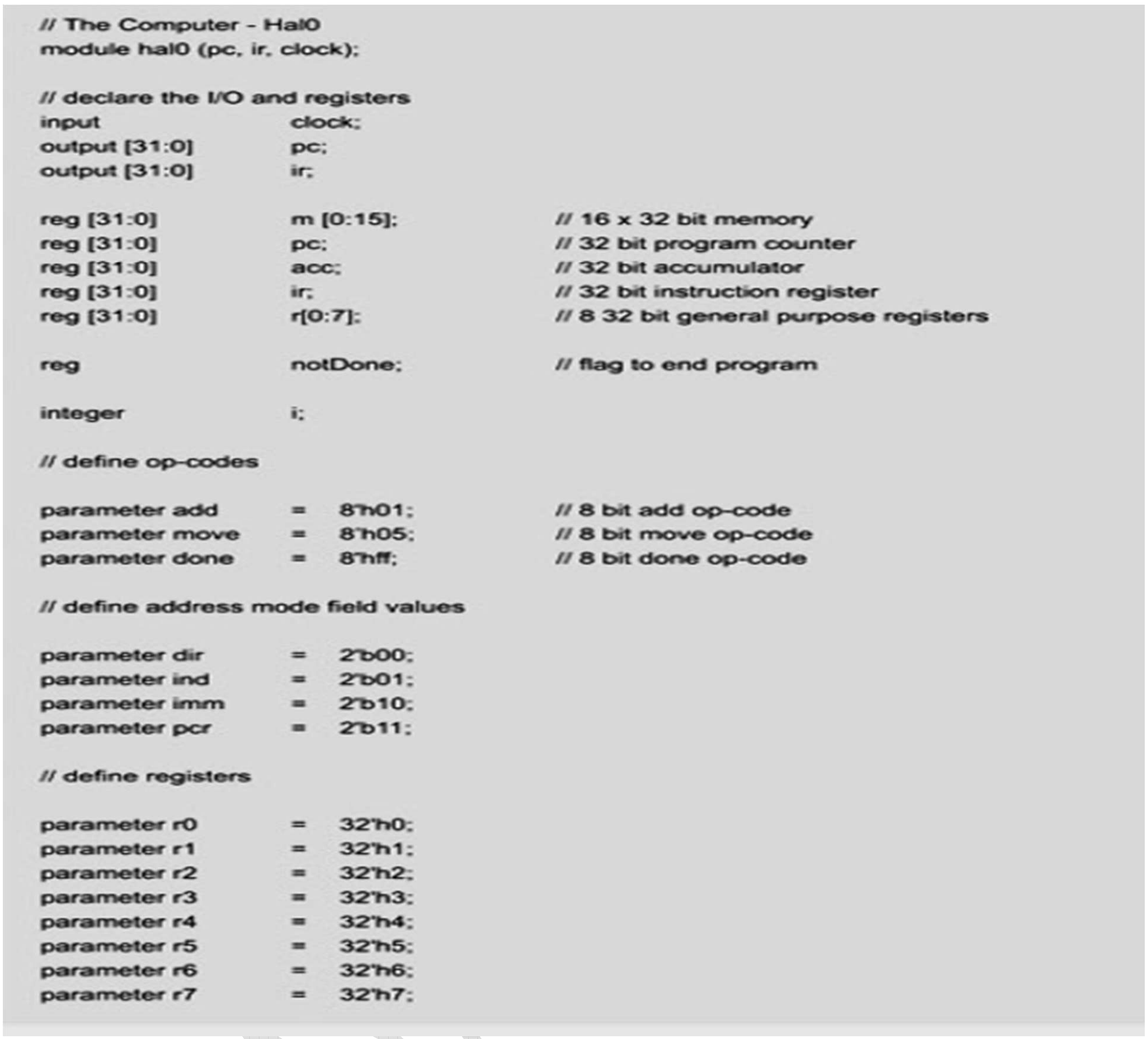
Algebraically modifying the PC is best accomplished by using one of the arithmetic functions in the ALU. The operation begins when the control logic places the desired offset into the ALU's temporary register. Next, the output of the PC is directed to the other ALU input. The ADD instruction is executed, and the result is entered into the PC. Placing a specific value into the PC can be done directly by the control logic since the target address is generally contained in the instruction. At the RTL level, the next step decomposes into the sequence of steps given in Figure 2.48.



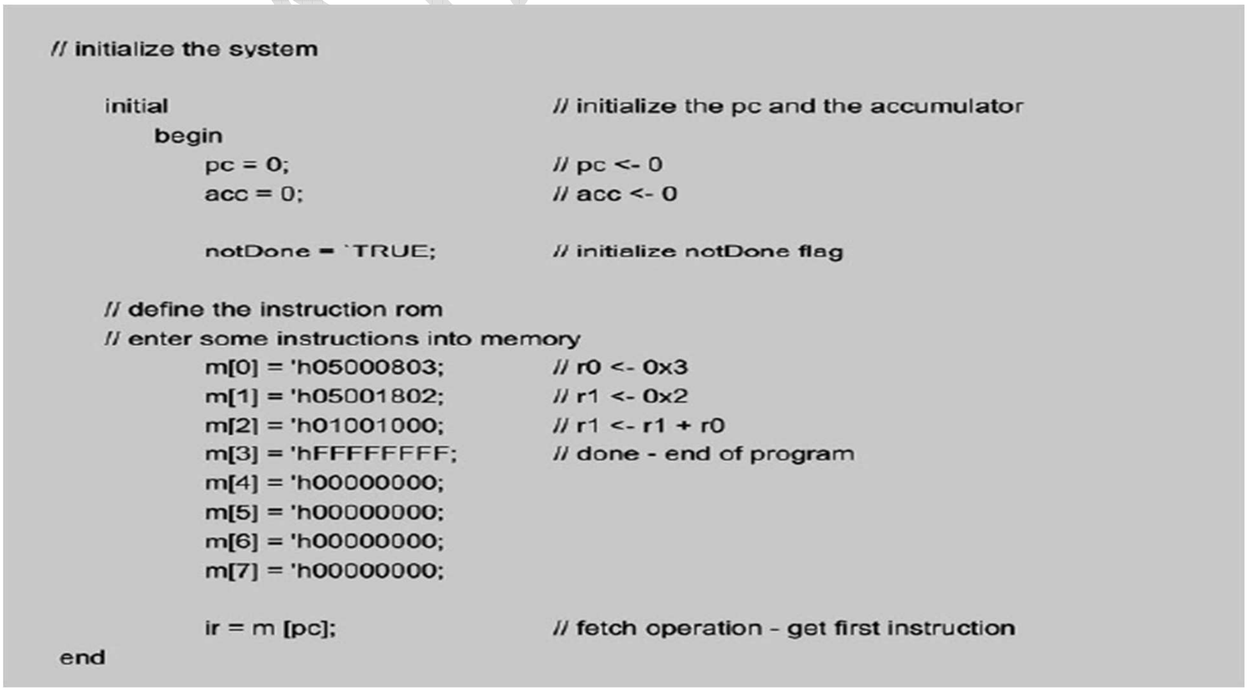
```
// Assume the offset is contained in the instruction
TR0 ← IR<n..m>;        // offset field of instruction enabled out to bus, TR0 captures value
TR1 ← TR0 + PC;        // PC enabled out to bus, ALU adds TR0 and PC
PC ← TR1               // TR1 enabled out to bus, PC captures value
```

Figure 2.48 The Next Sequence

The Verilog program in Figure 2.49 implements a behavioural model of a portion of the datapath and control for the simple CPU presented at the start of this section. The number of registers has been reduced, only two instructions are implemented, and the address mode field supports four different modes. Nonetheless, the architecture implements a working system

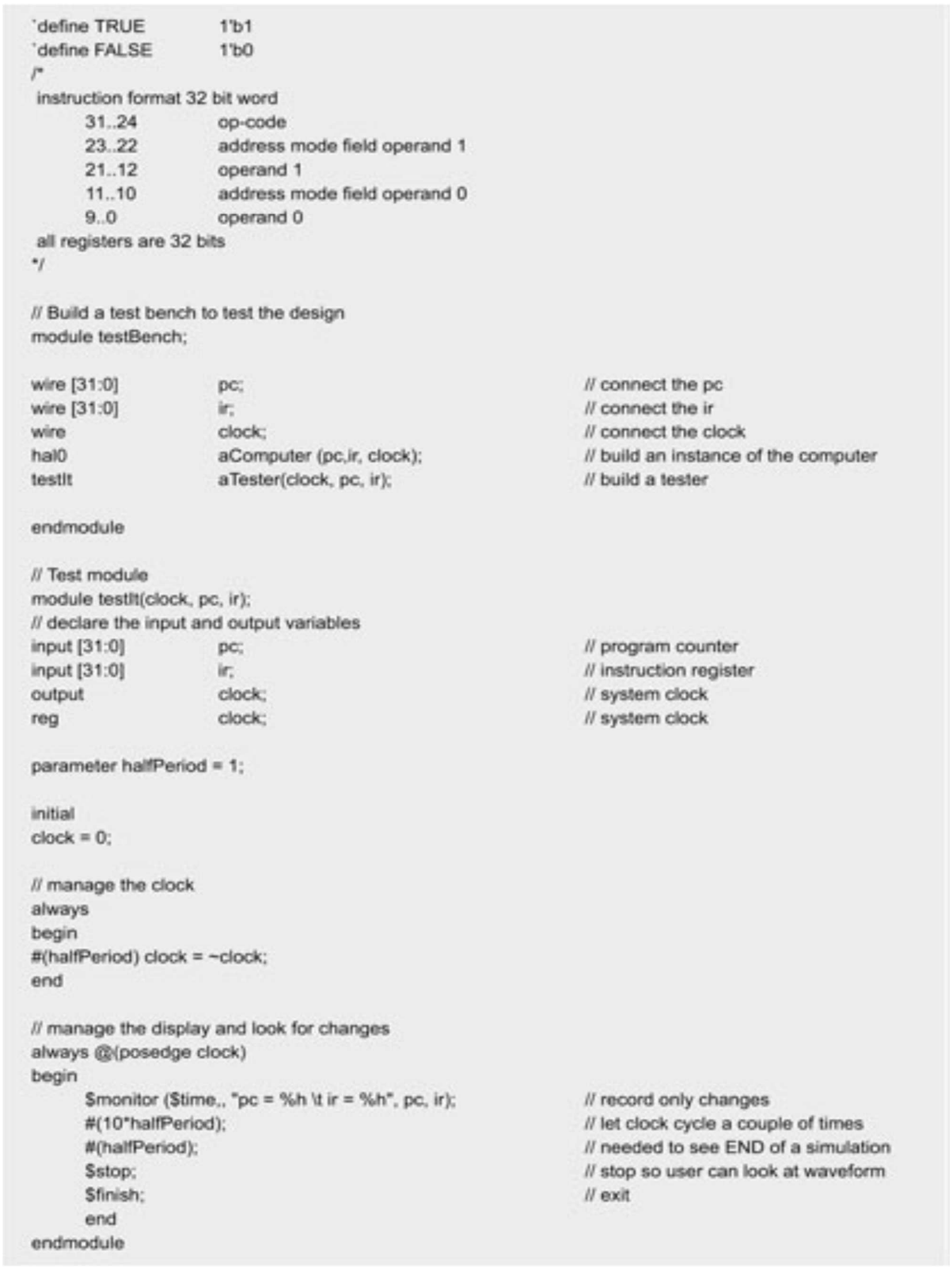


```
// The Computer - Hal0
module hal0 (pc, ir, clock);

// declare the I/O and registers
input                  clock;
output [31:0]          pc;
output [31:0]          ir;

reg [31:0]             m [0:15];          // 16 x 32 bit memory
reg [31:0]             pc;                // 32 bit program counter
reg [31:0]             acc;               // 32 bit accumulator
reg [31:0]             ir;                // 32 bit instruction register
reg [31:0]             r[0:7];            // 8 32 bit general purpose registers

reg                    notDone;           // flag to end program

integer                i;

// define op-codes

parameter add      =   8'h01;             // 8 bit add op-code
parameter move     =   8'h05;             // 8 bit move op-code
parameter done     =   8'hff;             // 8 bit done op-code

// define address mode field values

parameter dir      =   2'b00;
parameter ind      =   2'b01;
parameter imm      =   2'b10;
parameter pcr      =   2'b11;

// define registers

parameter r0       =   32'h0;
parameter r1       =   32'h1;
parameter r2       =   32'h2;
parameter r3       =   32'h3;
parameter r4       =   32'h4;
parameter r5       =   32'h5;
parameter r6       =   32'h6;
parameter r7       =   32'h7;
```



```
// initialize the system

    initial                             // initialize the pc and the accumulator
        begin
            pc = 0;                     // pc <- 0
            acc = 0;                    // acc <- 0

            notDone = `TRUE;            // initialize notDone flag

    // define the instruction rom
    // enter some instructions into memory
            m[0] = 'h05000803;          // r0 <- 0x3
            m[1] = 'h05001802;          // r1 <- 0x2
            m[2] = 'h01001000;          // r1 <- r1 + r0
            m[3] = 'hFFFFFFFF;          // done - end of program
            m[4] = 'h00000000;
            m[5] = 'h00000000;
            m[6] = 'h00000000;
            m[7] = 'h00000000;

            ir = m [pc];                // fetch operation - get first instruction
 end
```

```
// run the program
      always
        while (notDone == `TRUE)
        begin
          @(posedge clock)                          // control system timing
          case(ir[31:24])                           // decode operation
            move:                                   // move op-code
            begin
              $display("\nMove");                   // annotate execution

              case(ir[11:10])                       // check address mode
                dir:    $display("\ndirect");
                ind:    $display("\nindirect");
                imm:                                // implement immediate mode move
                begin
                  $display("\nimmediate");          // execute operation
                  r[ (ir[21:12]) ] = ir[9:0];       // rx <- aValue

                  $display("\n register value %h", r[ (ir[21:12]) ]);
                end

                pcr: $display("\n pc relative");
              endcase
            end
            add:                                    // add op-code
            begin
              $display("add\n");
              case(ir[11:10])
                dir:                                // register direct
                begin
                  $display("\ndirect");             // execute operation

                  // rx <- rx + ry
                  r[ (ir[21:12]) ] = r[ (ir[21:12]) ] + r[ (ir[11:0]) ];
                  $display("\n register value %h", r[ (ir[21:12]) ]);
                end

                ind:   $display("\nindirect");
                imm: $display("\nindirect");
                pcr:   $display("\npc relative");
              endcase
            end
            default:  $display("illegal op-code trap\n");  // identify illegal op-codes
          endcase
          pc = pc+1;                                // next operation compute next address
          ir = m [pc];                              // fetch operation
          if ( ir[31:24] == done )                  // check for end of program
            notDone =`FALSE;                        // done
        end
```



```
`define TRUE          1'b1
`define FALSE         1'b0
/*
 instruction format 32 bit word
        31..24        op-code
        23..22        address mode field operand 1
        21..12        operand 1
        11..10        address mode field operand 0
        9..0          operand 0
 all registers are 32 bits
*/

// Build a test bench to test the design
module testBench;

wire [31:0]           pc;                               // connect the pc
wire [31:0]           ir;                               // connect the ir
wire                  clock;                            // connect the clock
hal0                  aComputer (pc,ir, clock);         // build an instance of the computer
testIt                aTester(clock, pc, ir);           // build a tester

endmodule

// Test module
module testIt(clock, pc, ir);
// declare the input and output variables
input [31:0]          pc;                               // program counter
input [31:0]          ir;                               // instruction register
output                clock;                            // system clock
reg                   clock;                            // system clock

parameter halfPeriod = 1;

initial
clock = 0;

// manage the clock
always
begin
#(halfPeriod) clock = ~clock;
end

// manage the display and look for changes
always @(posedge clock)
begin
        $monitor ($time,, "pc = %h \t ir = %h", pc, ir);  // record only changes
        #(10*halfPeriod);                                 // let clock cycle a couple of times
        #(halfPeriod);                                    // needed to see END of a simulation
        $stop;                                            // stop so user can look at waveform
        $finish;                                          // exit
        end
endmodule
```

Figure 2.49 the simple CPU

The Hardware Side: Storage Elements and Finite-State Machines

The hardware in the microprocessor is the digital circuits, the digital circuits are classified into two namely combinational and sequential logic circuits.

In combinational logic circuits output depends on the primary state only. Examples of such circuits are adder, multiplexor, decoder and encoder

In sequential logic circuit the output of the circuit depends upon the present input and past outputs. Examples of such circuits are latch, flipflop, counters and registers further sequential circuits are classified into synchronous sequential and asynchronous sequential circuit. To design such a complex systems we use finite state machine.

Finite State Machine (FSM) used to describe the behaviour of digital circuits using finite no of states.

The concepts of time and state

In combinational circuit ignoring delay output is a function of input and no time constraints.

In sequential circuits output depends on what time clock reaches the circuit?, what time we get output?, what time input reaches the circuit. Hence the time is the integrated behaviour of sequential circuits.

State

In general state represents to the condition things are in, what they are like at a particular time? In this context the **state represents output of a circuit**, there may be one or more states but each state must be represented uniquely. To represent the output we use several variables this set of variables is called state variables The state of the circuit will change depends on the input and state changes with respect to time is called behaviour of the system.

State Diagram

State diagram is a **graphical representation** of behaviour of the circuit, it us very useful in complex logic design. We show the transition between two states using a labelled directed line or arrow called an arc (or edge in graph theoretic language) as illustrated in Figure 2.50. Because the line has a direction, the state diagram is referred to as a **directed graph**. The head or point of the **arrow** identifies the final state, and the tail or back of the arrow identifies the initial state. Special arcs, such as the one labelled initial, reflect an external, overriding asynchronous event, such as a reset, which places the system into a designated state, here state a Because the graph can contain cycles, it is further qualified as a **cyclic graph**. The label on each identifies what caused the change and the output(s) of system, if appropriate.

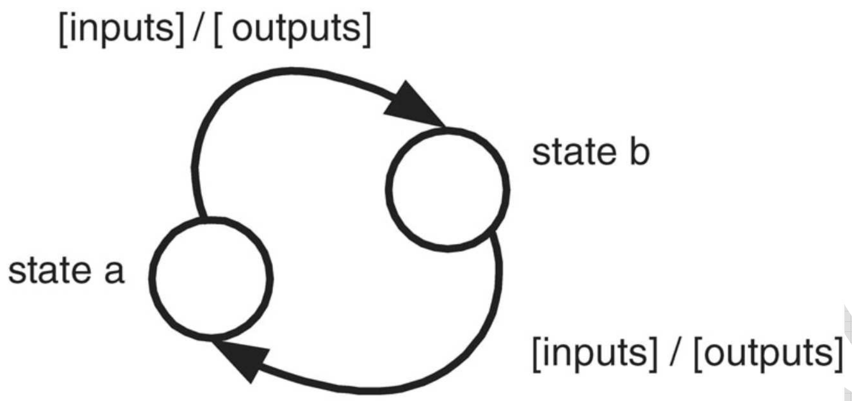
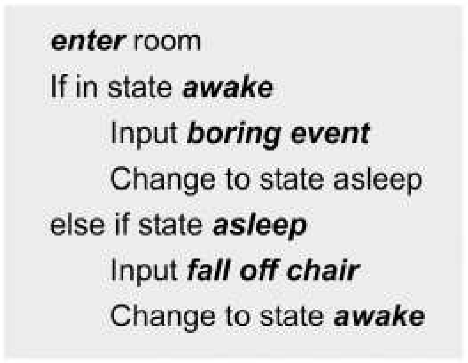


Figure 2.50 Transitions between States in a Digital Memory Device

Example describe an evening's entertainment. Draw state diagram graphically to expresses the same behaviour that is described textually in Figure 2.51.



```
enter room
If in state awake
    Input boring event
    Change to state asleep
else if state asleep
    Input fall off chair
    Change to state awake
```

Figure 2.51 Textual Description

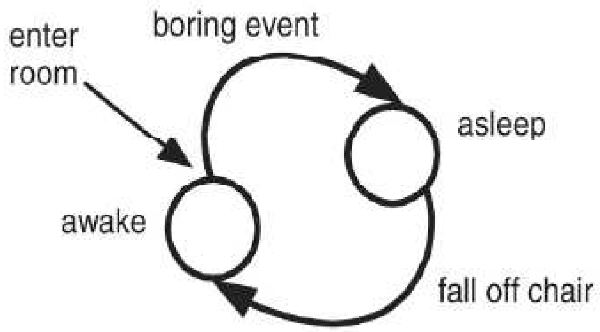


Figure 2.52 State diagram

Finite State Machine – A Theoretical Model

Here we discuss different abstraction levels of Simple finite-state machines, the first view is as shown in Figure 2.53 have no inputs other than a clock and have only primitive outputs (we generally don't show the clock). Such machines are referred to as **autonomous clocks**.

As we move to more complex designs, we will introduce **inputs** as well as more sophisticated outputs. A high-level block diagram for a finite-state machine begins with the diagram in Figure 2.54.

The **outputs** shown in the diagram may be the values of the state variables (as they will be in counting-type designs), combinations of the **state variables, or combinations of the state variables and the inputs.** Refining the level of detail the block diagram for the state machine appears as shown in Figure 2.55

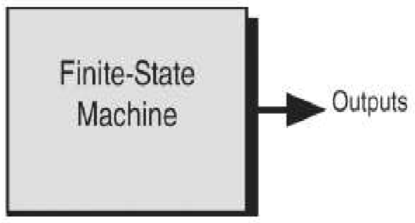


Figure 2.53 An Autonomous Clock

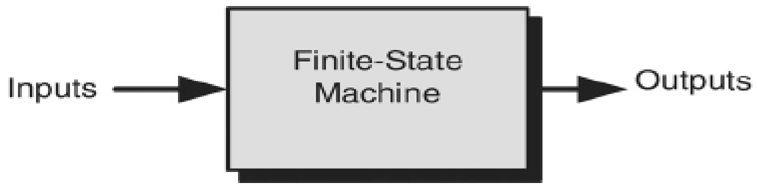


figure 2.54 A High-Level B D for a FSM

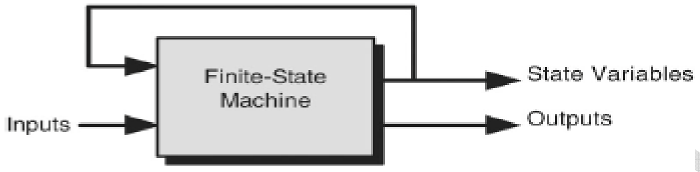


Figure 2.55 A High-Level Block Diagram for a Finite-State Machine

We see first that the signals out of the finite-state machine decompose into two sets: state variables and outputs. Observe that the **state variables are fed back as inputs** to the system. The diagram illustrates the essence of the strength of the machine. It has the ability to recognize the state that it is in and then to react based on the values of the state variables and (potentially) to the inputs to the system. The **decision as to which state to go to next** is determined from the current input and the state that the machine is currently in. The present state of a finite-state machine inherently encodes the history of the path taken to get there.

If we continue increasing the level of detail in the model, we now include the **storage elements** comprising the machine and the combinational logic that implements the output functionality and the input equations to the storage elements. The block diagram now becomes that in Figure 2.56.

The model has n inputs, m outputs, and p state variables. A memory device is associated with each state variable, and each state variable is associated with a memory device. At this point no particular type of memory device is specified

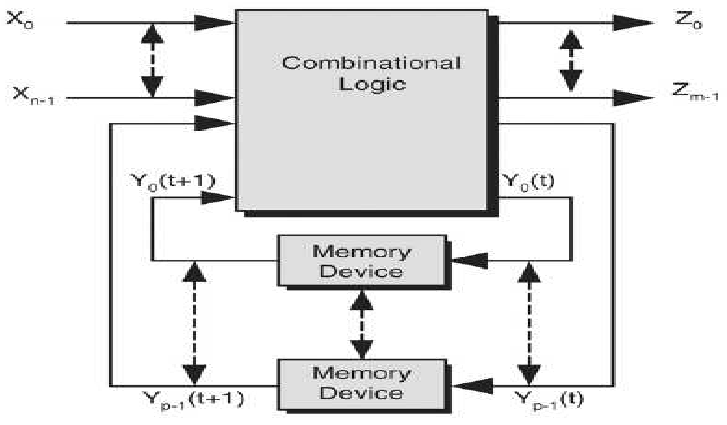


Figure 2.56 A High-Level Block Diagram for a Finite State Machine

We specify the set of variables Xi to represent the n inputs to the system; Zj to represent the m outputs from the system; and Yk to represent the p internal state variables. We define our finite-state machine as a quintuple (function of five).

M = (I, O, S, λ,δ)

I - Finite nonempty set or vector of inputs

O - Finite nonempty set or vector of outputs

S - Finite nonempty set or vector of states

δ - Mapping **I x S -> S**

λ 1 - Mapping **I x S -> O** - Mealy Machine

λ 2 - Mapping **S - O** ->Moore Machine

There are two types of Mealy and Moore machines.

Mealy machine - λ 1 The output is a function of the present state and inputs

Moore machine - λ 2 The output function of the present state only.

Extensions to the basic diagram, such as we find in hardware/software co-design tools or the Unified Modelling Language (UML) state charts, support a rich set of system modelling capabilities.