Chapter-1
Embedded System

1.0 Introduction to Embedded System

1.1 Embedded System Overview

An embedded system is nearly any computing system other than a desktop computer. An embedded system is dedicated system which performs the desired function upon power up repeatedly.
Embedded systems are found in a variety of common electronic devices such as

a. Consumer Electronics
- Cell phone, pagers digital cameras, calculators, portable video games, personal digital assistance and so on

b. Home Appliance
- Micro-oven, answering machine, home security system, washing machine, lightening system etc.

c. Office Automation
- Fax machines, copiers, printers and scanners.

d. Business Equipments
- Cash registers, check-in, alarm system, card reader, product scanner, automated telling machine etc.

e. Automobiles
- Cruise control, driver assistance system, parking assistance sytem, anti-lock brakes and active suspensions etc.

1.2 General Characteristics of Embedded System

1. Single-functioned

An embedded system usually executes a specific program repeatedly. They are usually based on philosophy of super loop concept. For example a washing machine is continuously monitoring the water level, detergent level, cloth condition; on contrary desktop computers executes variety of programs like word-processing, video games and many other application programs.

2. Tightly constrained

An embedded system has some constraints which are especially tight. It should be cheap, fit into a single chip, must perform fast, must be reliable and predictable and also should consume less power to extend battery life.

3. Real Time

An embedded system must compute results in real time without any delay. In hard-real time, any delay in deadline could result in catastrophe. For e.g. air-Bag system, landing of airplane. Whereas in Soft-real time any delay in deadline could degrade the system quality of service. For e.g. on-line reservation system, live video streaming etc.

4. Reactive

Most of the embedded sytems must continually react to changes in the system environment and must compute certain results in real time without delay.

For example a car's cruise controller contimually monitors and reacts to speed and brake sensors. It must compute acceleration or deacceleration amount repeatedly within a limited time, a delayed computation could result in failure to maintain the control of the car.

2.0 Classification of Embedded system

1. **Small Scale Embedded System**
   These systems are designed with a single 8-or 16 bit microcontrollers, they have little hardware and software complexities and involve board-level design. They may even be battery operated. When developing embedded software for these, an editor assembler and cross assembler specific to the microcontroller or processor is used.

2. **Medium-Scale Embedded System**
   These systems are usually designed with a single or few 16- or 32 bit microcontroller or DSPs. They have both hardware and software complexities. For complex software design, RTOS, source code engineering tool, Simulator, Debugger and Integrated Development Environment (IDE) are used to solve these problems.
   Software tools also provide the solution to the hardware complexities.

3. **Sophisticated Embedded System**
   Sophisticated embedded systems have enormous hardware and software complexities and may need scalable processor or configurable processor and programmable logic arrays. They are used for cutting edge application that needs hardware and software co-design and integration in the final system; however they are constrained by the processing speed available in their hardware unit.
   Certain software functions such as encryption and deciphering algorithms, discrete cosine transformation and inverse transformation algorithms, TCP/IP protocols stacking and network driver functions are implemented in the hardware to obtain additional speed by saving time.
   Some of the functions of the hardware resources in the system are also implemented by the software. Development tools for these systems may not be readily available at a reasonable cost or may not be available at all.
   In some case a compiler or retarget able compiler might have to be developed for these.

2.1 Essential Components of Embedded System

1. Processor in an Embedded System
   A processor is an important unit in the embedded system hardware. A micro-controller is an integrated chip that has the processor, memory and several other hardware units in it; these form the microcomputer part of the embedded system. An embedded processor is a processor with special feature that allow it to be embedded into a system. A digital signal processor (DSPs) is a processor meant for applications that process digital signals.
2. Commonly used microprocessors, microcontrollers and DSPs in the small, medium and large scale embedded systems.
3. A recently introduced technology that additionally incorporates the application-specific system processor in the embedded system.
4. Multiple processors in a system

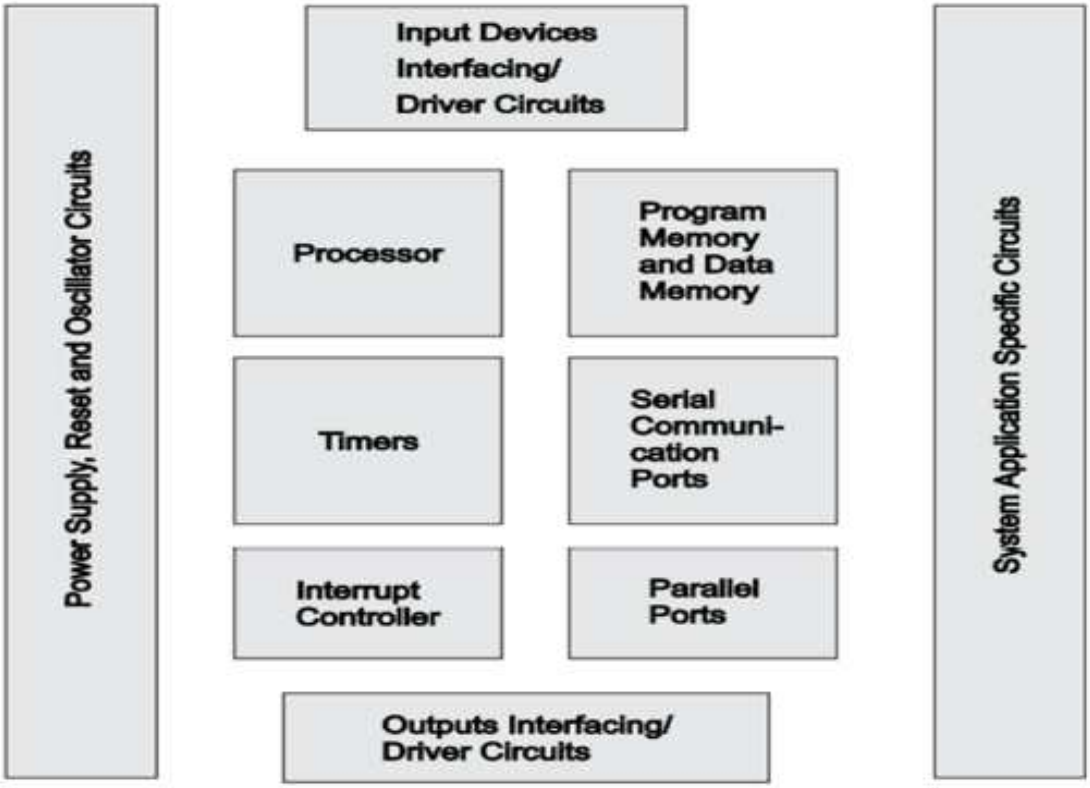


Figure: Hardware Units of Embedded System

3.0 Overview of Processors in Embedded system

Processor technology relates to the architecture of the computation engine used to implement a system's desired functionality.

1. **General purpose processor**

   The designer of a general-purpose processor builds a programmable device that is suitable for a variety of applications to maximize the number of device sold.

   One feature of such processor is a program memory; the designer of such processor doesn't know what program will run on the processor, so the program cannot be built into the digital circuit.

   Another feature is general data path,--the data path must be general enough to handle a variety of computations, so such a data path typically has a large register file and one or more general-purpose arithmetic-logic unit.

   Thus embedded system designer simply uses a general purpose processor to carry out required functionality by programming it. Benefits of such processor are time to market and NRE (non-recurring engineering cost) and high flexibility.

   However, there are also some design-metric drawbacks. Unit cost may be relatively high for large quantities. Performance may be slow for certain applications. Size and power may be large due to unnecessary processor hardware. Pentium is most popular general purpose processor.

Examples:

- Microprocessors
  - Intel: 8085\8086, 80186,80286
  - Motorola: 6800,6809,G3,G4,G5
- Microcontrollers
  - Intel: 8032,8051,8052
  - AVR: ATMEGA 328
- Embedded processors
  - ARM 7/9/11, CORTEX-M Intel i960
- Digital Signal Processors
  - PAC,TMS320XX series, Zed-board

2. Application Specific processor

An application-specific instruction-set processor (ASIP) is optimized for a particular class of applications having common characteristics, such as embedded control, digital-signal processing or telecommunication. The designer of such processor can optimize the datapath for the application class, perhaps adding special functional unit for common operations and eliminating other infrequently used units.

ASIP in an embedded system can provide benefits of flexibility while still achieving good performance, power and size.

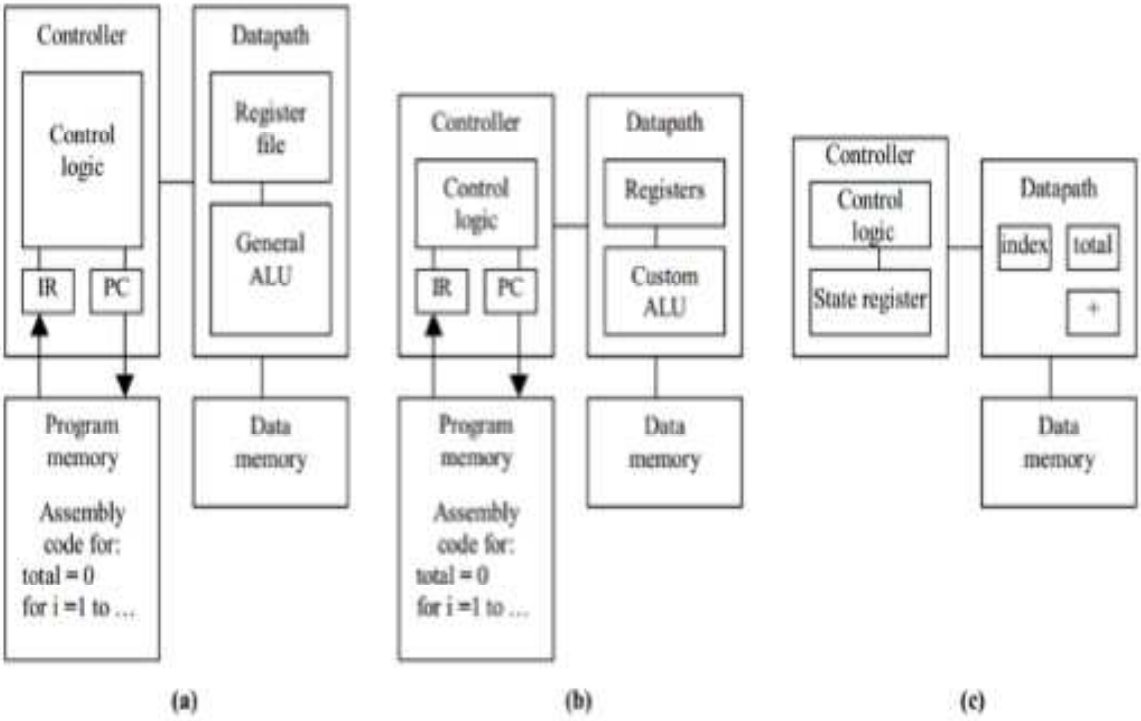


Figure: implementing desired functionality on different processor types
a) general-purpose b) application specific c) single-purpose

3. **Single purpose processor**

   A single purpose processor is a digital circuit designed to execute one program. It contains only the components to execute a single program. Using a single-purpose processor in an embedded system results in several design-metric as well as have some drawbacks.

   Performance may be fast, size and power may be small and unit cost may be low for large quantities, while design time is high as well as flexibility low.

3.1 Hardware units in an embedded system

A microcontroller is a small computer on a single integrated circuit containing a processor core, memory and programmable input/output peripherals. Program memory in the form of NOR flash or OTP ROM is also often includes on chip as well as a typically small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the micro-processor used in personal computers or other general applications.

3.2 Exemplary application of each type of embedded system

Embedded systems have very diversified applications. A few selected application area of embedded system are Telecom, smart cards, Missile and satellite, computer Networking, Digital Consumer Electronics, and Automotive.

Figure below show some applications of embedded systems:

I. **Telecom**

- Mobile computing
- Wireless
- Networking

II. **Smart card**

- Banking
- Telephone
- Security

III. **Missiles and satellites**

- Defense
- Aerospace
- Communication

IV. **Computer Networking systems and peripherals**

- Networking system
- Image processing
- Printers
- Network cards
- Monitors and display

V. **Automotive**

- Motor control system
- Cruise control
- Engine/body safety
- Car Entertainment etc

3.3 Common Design Metrics

The embedded-system designer must construct an implementation that fulfill the desired functionality as well as simultaneously optimizes numerous design metrics.

Commonly used design metrics includes

- **NRE Cost(nonrecurring engineering cost):**
  One time cost to research, developed design and test a new product. Once the system is designed any number of units can be manufactured without incurring any additional cost.
- **Unit Cost**
  The monetary cost of manufacturing each copy of the system, excluding NRE cost.
- **Size**
  The physical space required by the system, often measured in bytes for software and gates or transistors for hardware.
- **Performance**
  The execution time of the system
- **Power**
  Amount of power consumed by the system
- **Flexibility**
  Ability to change the functionality of the system without incurring heavy NRE cost.
- **Time-to-prototype**
  Time required to develop a working miniature model of the system
- **Time-to-market**
  Time required to develop a system so it can be released and sold to the customer
- **Maintainability**
  The ability to modify the system after its initial realese, especially by designer who did not originally design the system
- **Safety**
  The probability that the system will not cause harm

UNIT 2

2.0 Hardware and software Design Issue

2.1 Introduction

A processor is a digital circuit designed to perform computation task. A processor consists of data path capable of storing and manipulating data and a controller is capable of moving data through the data path. A general purpose processor is designed such that it can carry out a wide variety of computation task, which are described by a set of instructions.

In contrast, a single purpose processor is designed specifically to carry out a particular computation task. While some tasks are so common that we can purchase standard single-purpose processor to implement those tasks, other is unique to a particular embedded system. Such custom tasks may be best implemented using custom single-purpose processor that we design.

An embedded system designer may obtain several benefits by choosing to use a custom single-purpose processor rather than a general-purpose processor to implement a computation task.

In this unit, we start with basic of combinational and sequential design and describe the method of converting programs to custom-single purpose processor.

2.1.1 Combinational and sequential logic

Transistors and logic gates

A transistor is the basic electrical component in digital systems. Combinations of transistors form more abstract components called logic gates, which designer use when building digital systems.

A transistor acts as simple on/off switch. Today's digital design is dominated by CMOS technology. Thus all the transistors are manufactured using CMOS technology.

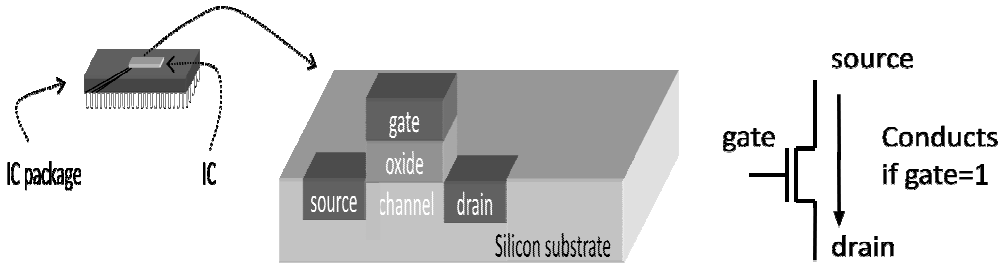


Figure: A simplified view of a CMOS transistor on silicon

In digital electronics high voltage i.e. +3v or +5v is referred to logic 1 and low voltage typically ground voltage is referred as logic 0.

When logic 1 is applied to the gate of the transistor currents flow from source to drain. Similarly when logic 0 is applied transistor doesn't conduct.

CMOS transistor implementation

Using CMOS transistors we can implement some of the basic logic. We have two type of CMOS transistors. a)Nmos b)Pmos transistors.

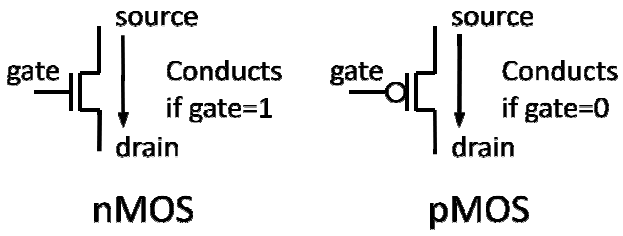
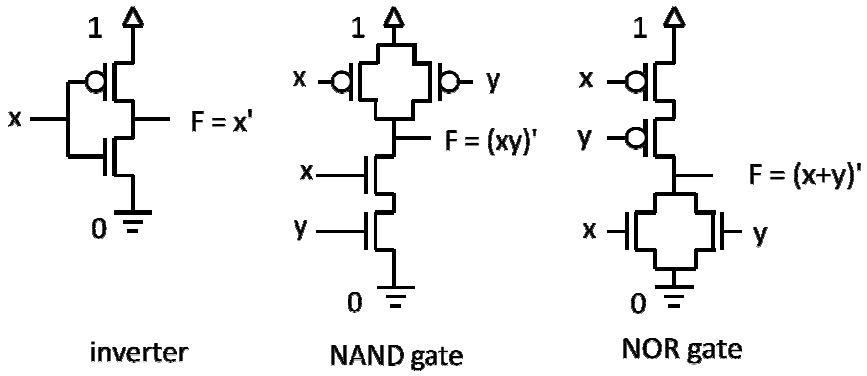


Figure: nMOS transistor pMOS transistor

Basic gates using CMOS transistors



- When the input X is logic 0, the top transistor conducts and the bottom transistor doesn't conducts so, logic 1 appears at the output F.
- NAND gate is complement of AND gate. We can implement NAND gate by using configuration as shown
- If both X&Y are logic 1 it doesn't conducts so the output is complement of (X.Y)
- If any of the input is zero the output is logic high.

Basic Logic Gates:

Digital system designer usually works at the abstraction level of logic gates rather than transistors. Each gate is represented symbolically and with Boolean equations along with truth table as show below.

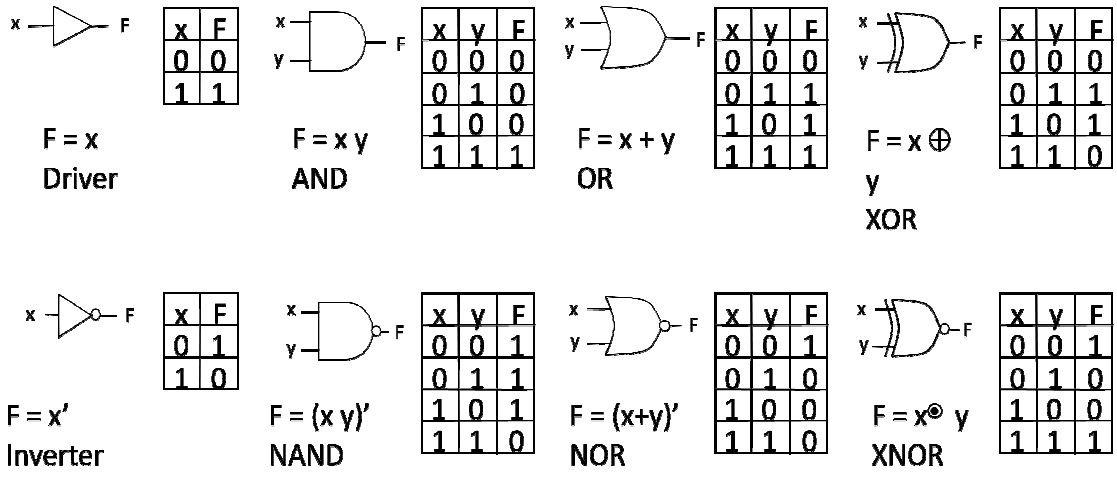


Figure: Basic logic gates

2.1.1 Basic combinational logic Design:

A combinational circuit is a digital circuit whose output is purely a function of its present inputs. Such a circuit has no memory of past inputs.

2.1.2 Combinational circuit design procedure

The design procedures for combinational logic circuits start with the problem specification and compromise the following steps.

i. Determine the required number of inputs and outputs from the specifications.
ii. Derive the truth table for each of the outputs based on their relationship to the input.
iii. Simplify the Boolean expression for each output. Use Karnaugh map.
iv. Draw a logic diagram that represents the simplified Boolean expression. Verify the design by analyzing or simulating the circuits

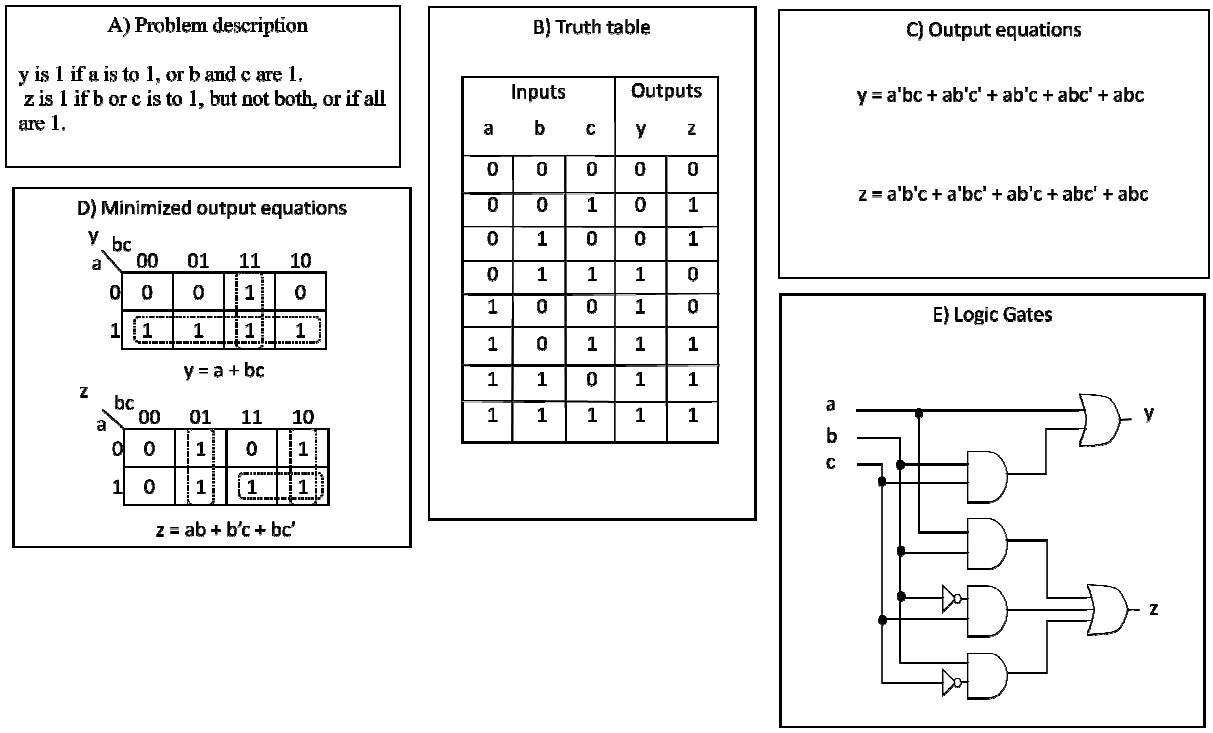


Figure: Combinational logic design (a) problem description (b) truth table (c) output equation (d) minimized output equations (f) final circuit

2.1.3 RT-Level combinational components.

All combinational circuits can be designed in the manner as shown above but larger circuits would be very complex to design. For e.g. a circuit with 16 inputs would have 2^{16} or 64k rows in its truth table. One way to reduce the complexity is to use combinational components. Such combinational components are often called register-transfer or RT components.

1. Multiplexer

The multiplexer is a combinational logic circuit designed to switch one of several input lines through a single common output line by the application of a control signal.
If there are m data inputs, then there are $\log_2(m)$ select lines S.

2. Decoder

The binary decoder is another combinational logic circuit. The decoder transforms a set of digital input signals into an equivalent decimal code at its output. Some decoders have an additional input pin labeled “Enable” that controls the output from the device. This extra input allows the decoder output to be turned “ON” or “OFF” as required. These types of binary decoders are commonly used as “memory address decoders”.

3. Adder

An adder adds two n-bit binary inputs A and B generating n-bit output sum along with an output of 1-bit carry.

4. Comparator

A comparator compares two n-bits binary inputs A and B, generating outputs that indicates whether A is less than, equal to or greater than B.

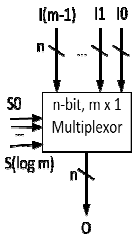
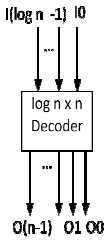
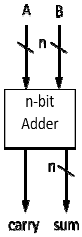
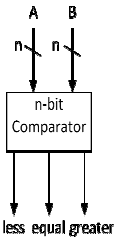
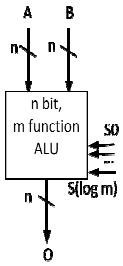
5. Arithmetic and Logic Unit

An arithmetic and logical unit (ALU) can perform a variety of arithmetic and logic function on its n-bit inputs A and B. The select line ‘S’ choose the current function, if there are possible functions, then there must be at least log2(m) select lines. Commonly functions includes addition, subtraction, AND and OR.

6. Shifter

Another common RT-level component is a shifter. An n-bit input I can be shifted left or right and then output to an output 0.

For e.g. a 4-bit shifter with an input 1010 would output 0101 when shifting right one position.

				
O = I0 if S=0..00 I1 if S=0..01 ... I(m-1) if S=1..11	O0 =1 if I=0..00 O1 =1 if I=0..01 ... O(n-1) =1 if I=1..11	sum = A+B (first n bits) carry = (n+1)’th bit of A+B	less = 1 if A<B equal =1 if A=B greater=1 if A>B	O = A *op* B *op* determined by S.
	With enable input e → all O’s are 0 if e=0	With carry-in input Ci→ sum = A + B + Ci		May have status outputs carry, zero, etc.

2.2.0 Sequential logic

Unlike combinational logic circuit that changes state depending upon the actual signal being applied to their inputs at that time, sequential logic circuit have some form of inherent "Memory" built in to as they are able to take into account their previous state as well as those actually present.

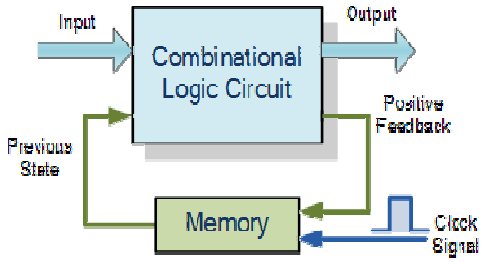
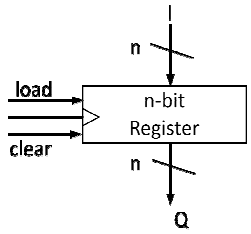
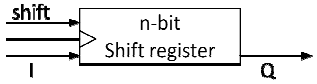
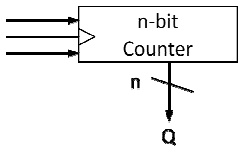


Figure: sequential Circuit

In other words, the output state of a "sequential logic circuit" is a function of the following three states.

- Present input
- Past input and/or
- Past output

Thus sequential logic circuit remember these conditions and stay fixed in their current state until the next clock signal changes one of the states, giving sequential logic circuit "Memory".

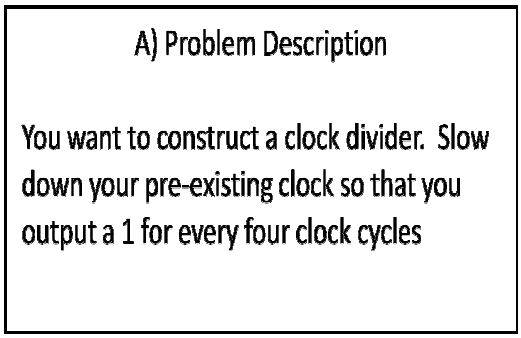
		
Q = 0 if clear=1, I if load=1 and clock=1, Q(previous) otherwise.	Q = lsb - Content shifted - I stored in msb	Q = 0 if clear=1, Q(prev)+1 if count=1 and clock=1.

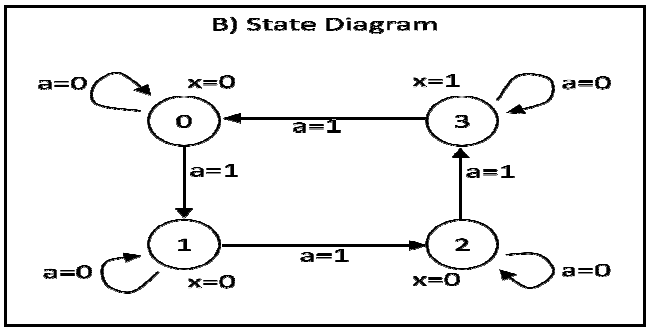
2.2.1 Sequential logic design

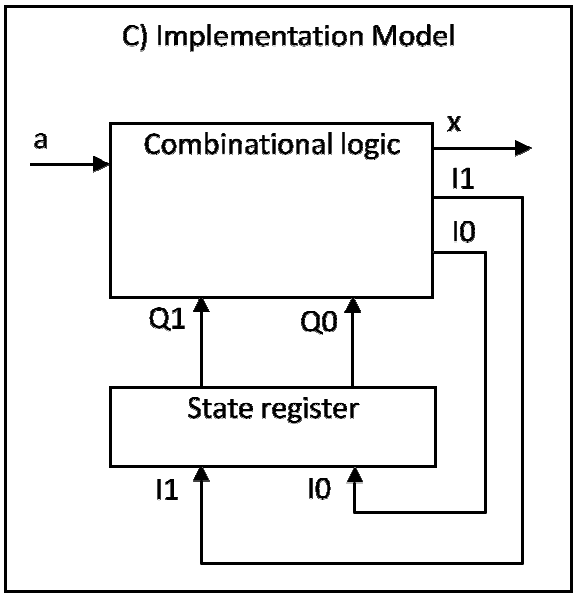
Sequential logic design can be achieved using a straight forward technique. Some of the design procedures are as follows.

- Derive circuit state diagram for design specifications.
- Create state table
- Create circuit excitation table
- Construct K-maps for
    - Flip-flops inputs
    - Primary inputs
- Obtain minimized SOP equation

- Draw logic diagram
- Simulate to verify design and debug as needed
- Perform circuit analysis and logic optimization







D) State Table (Moore-type)

Inputs			Outputs		
Q1	Q0	a	I1	I0	x
0	0	0	0	0	0
0	0	1	0	1	
0	1	0	0	1	0
0	1	1	1	0	
1	0	0	1	0	0
1	0	1	1	1	
1	1	0	1	1	1
1	1	1	0	0	

Figure: Sequential logic design a) Problem description b) state diagram c) implementation model d) State table

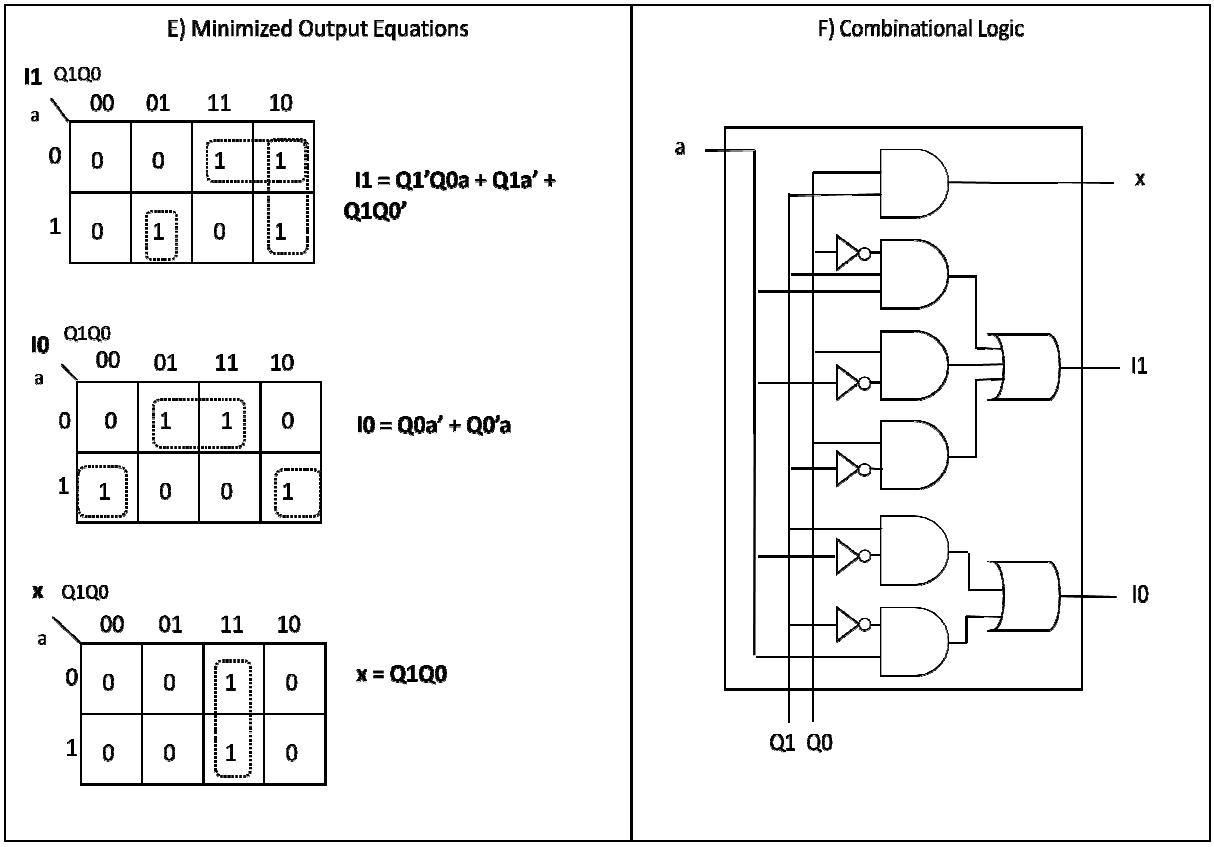


Figure: Sequential logic design e) minimized output equation f) combinational logic

2.3.0 Custom single-purpose processor basic model

Processor is the heart of the embedded system. A single purpose processor is designed to perform a single but general task. A basic processor consists of a controller and a datapath as shown in figure below.

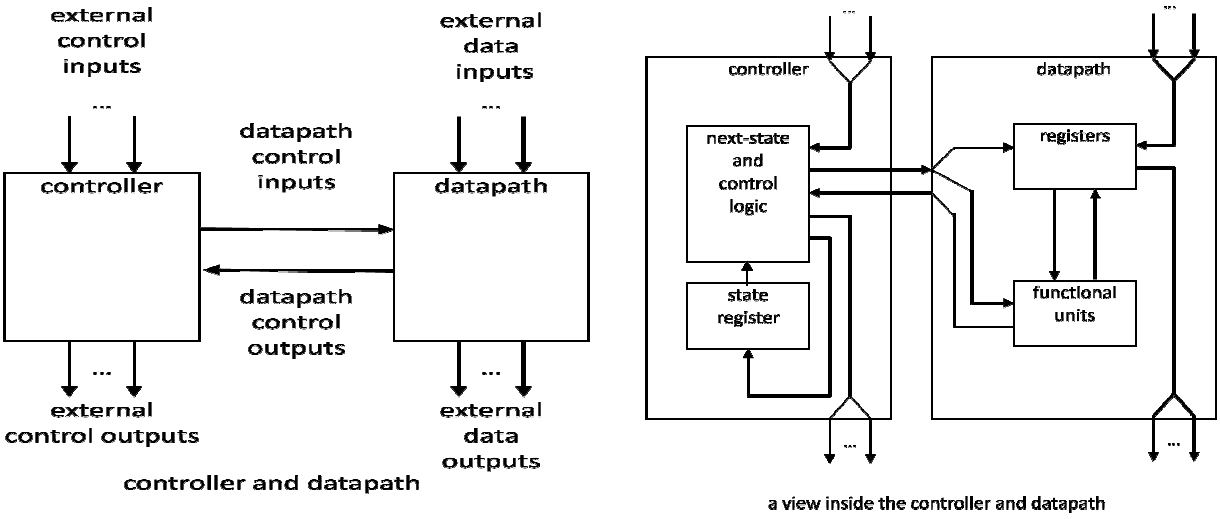


Figure: A basic processor a) controller and data path b) a view inside the controller and data path

2.3.1 Data path:

The data path stores and manipulates a system data. Examples of data in an embedded system include binary number generated by sensors.

Data path consists of registers, ports, functional units and the interconnection between these modules. Thus data path can be configured or designed to read data from particular register, feed them through certain function units and store the result back to a register.

2.3.2 Controller:

Controller is capable of moving data through the datapath. It sets the datapath control i/o such as registers loads, multiplexer selects signals of register units, functional unit and connection unit to obtain the desired configuration; controller is capable of monitoring external control inputs as well as datapath control outputs.

Both controller and datapath can be designed using combinational and sequential logic.

Design a custom single purpose processor that computes the GCD/HCF of two numbers eg 2 inputs 8 and 4, the output should be 4

a. Black- box design.

Lets create a black-box diagram of the desired system having x_i and y_i as data input and d_o as data output.

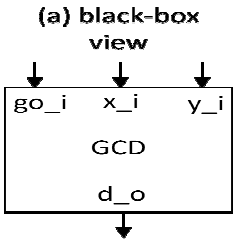


Figure: Black-box view

b. System functionality

The output should represent the GCD of the inputs. If two numbers are 8 and 4, the output should be 4.

```
0:      int x,y;
1:      while (1){
2:              while(!go_i);
3:                      x=x_i;
4:                      y=y_i;
5:                              while(x!=y){
6:                                      if(x<y)
7:                                              y=y-x;
                                        else
8:                                              x=x-y;
                                                }
9:                                      d_o=x;
                }
```

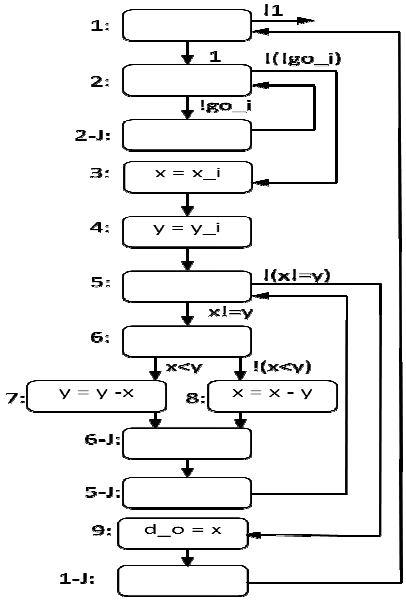


Figure: State diagram

2.4.0 Steps to design custom single purpose processor

1. To begin building a custom single purpose processor we need to convert a program's statement into a complex state diagram. (This complex state diagram here means finite state machine with data) FSMD.
    - In complex state diagram, the states and arc(transitional condition) may include arithmetic expression and those expressions may use external inputs and outputs as well as variables.
    - Complex state diagrams looks like a sequential programs in which statements have been changed into states.
    - We classify statements into assignment, loop and branch

a. Assignment Statement:

- For an assignment statement, create a state with that assignment as its action.
- Add an arc form this state to the state for the next statement.

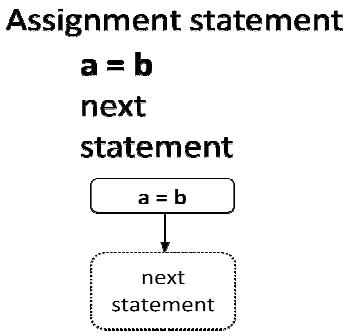


Figure: assignment template

b. Loop statement

- For a loop statement, create a condition statement (C) and a join state (J) both with no action
- Add arc with the loop's condition from the condition state to the first statement in the loop body.
- Again add a second arc with the complement of the loop's condition from the condition state to the next statement after the loop body.
- Don't forget to add an arc from join state back to condition state.

```
While (cond){
        Loop-body-statement;
}
Next statement;
```

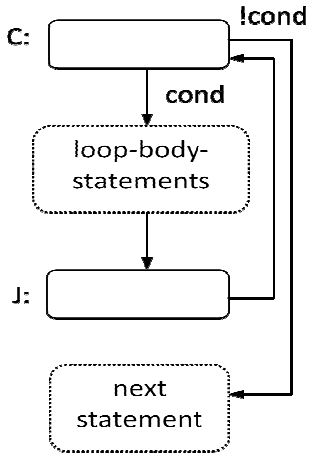


Figure: loop template

c. Branch statements

- For branch statement, we create a condition state C and a join state J, both with no actions.
- Add an arc with the first branch's condition from the condition state to the branch first statement.
- Similarly add another arc with the complement of the first branch condition ANDed with the second branches condition from the condition state to the branch's first statement.
- Same procedure is repeated for each branch.
- Finally connect the arc leaving the last statement of each branch to the join state and add an arc from this state to the next state.

```
If(C1)
   C1 statement;
else if(C2)
          C2 statement;
else
     other statements;
next statement
```

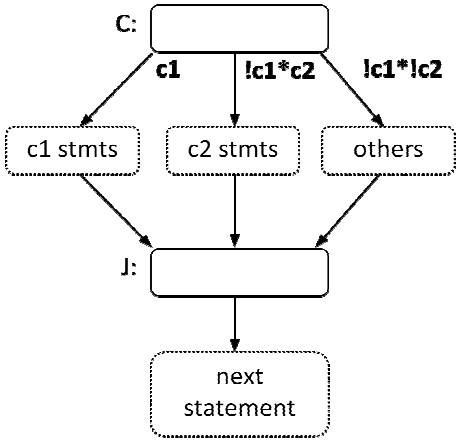


Figure: Branch template

2. Once the complex state diagram is obtained, functionality is divided between controller and data path.

- → The datapath should only consist of an interconnection of the combinational and sequential blocks
- → The controller should consists of a basic state diagram i.e. one containing only Boolean action and conditions.

a) Constructing the data path

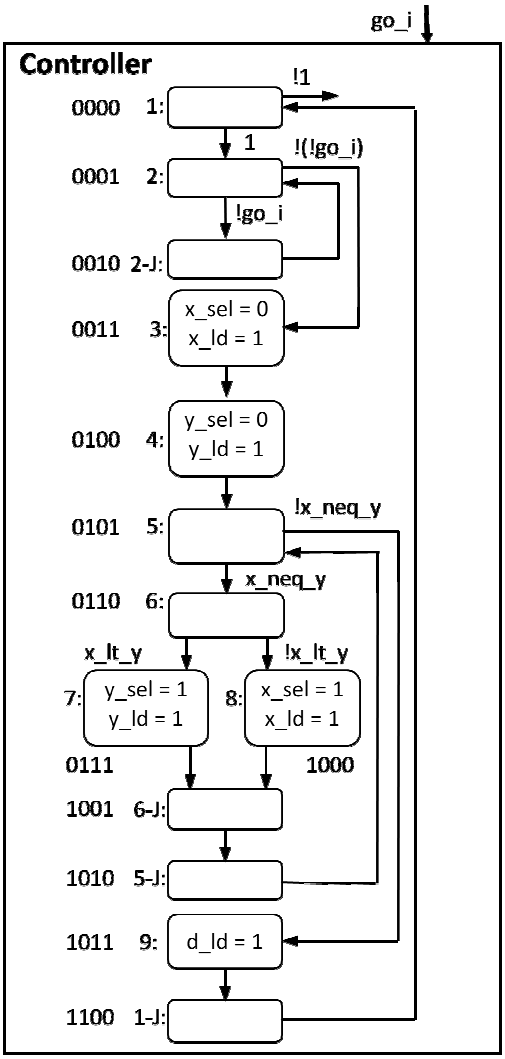
- → Create a register, for any declared variable
- → Create a functional unit for each arithmetic operation in the state diagram.
- → Then connect the ports, registers and functional units together.
- → For each logic and arithmetic operation connect sources to an input of the operations corresponding functional blocks.
- → When more than one source is connected to a register, add an approximately sized multiplexer.
- → Finally create a unique identifier for each datapath components, control inputs and outputs.

b) Constructing the controller:

- → The state diagram for the controller has the same characteristics as the complex state diagram (FSMD) but the complex actions have been replaced by the Boolean ones.
- → Replace every variable "write" by action that set the select signal of multiplexer in front of the variable register.

→ Replace every logical operation in a condition by the corresponding functional unit control output.

→ Complete the controller design using a basic state diagram (FSM) using the standard sequential design process.



`

Figure: controller

2.5.0 GCD

Step 1:

- Black box design

  Same as previous

Step 2:

- Functional Algorithm

Step 3:

- Data path design

i. Create a register for any declared variable. Here our declared variables are (x) and (y)

ii. Any output port is an input variable so create a register (d) and connect it to the output port.

ii. Create functional units for each arithmetic operation

For our example, we have two subtractors one comparator for less than and one comparison for inequality. In total two subtractors and two comparators.

iii. Connect the ports, registers and functional units

→ When more than one source is present add an approximately sized multiplexed.

iv. Create a unique identifier for each control input and output of the data path components.

e.g.

x_sel = select x
y_sel = select y
x_neq_y = x not equal y
y_lt_y = x less than y
d_ld = load d
x_ld = load x
y_ld = load

Complex State diagram

Since, we have complete datapath; we modify our FSMD into FSM representing our controller

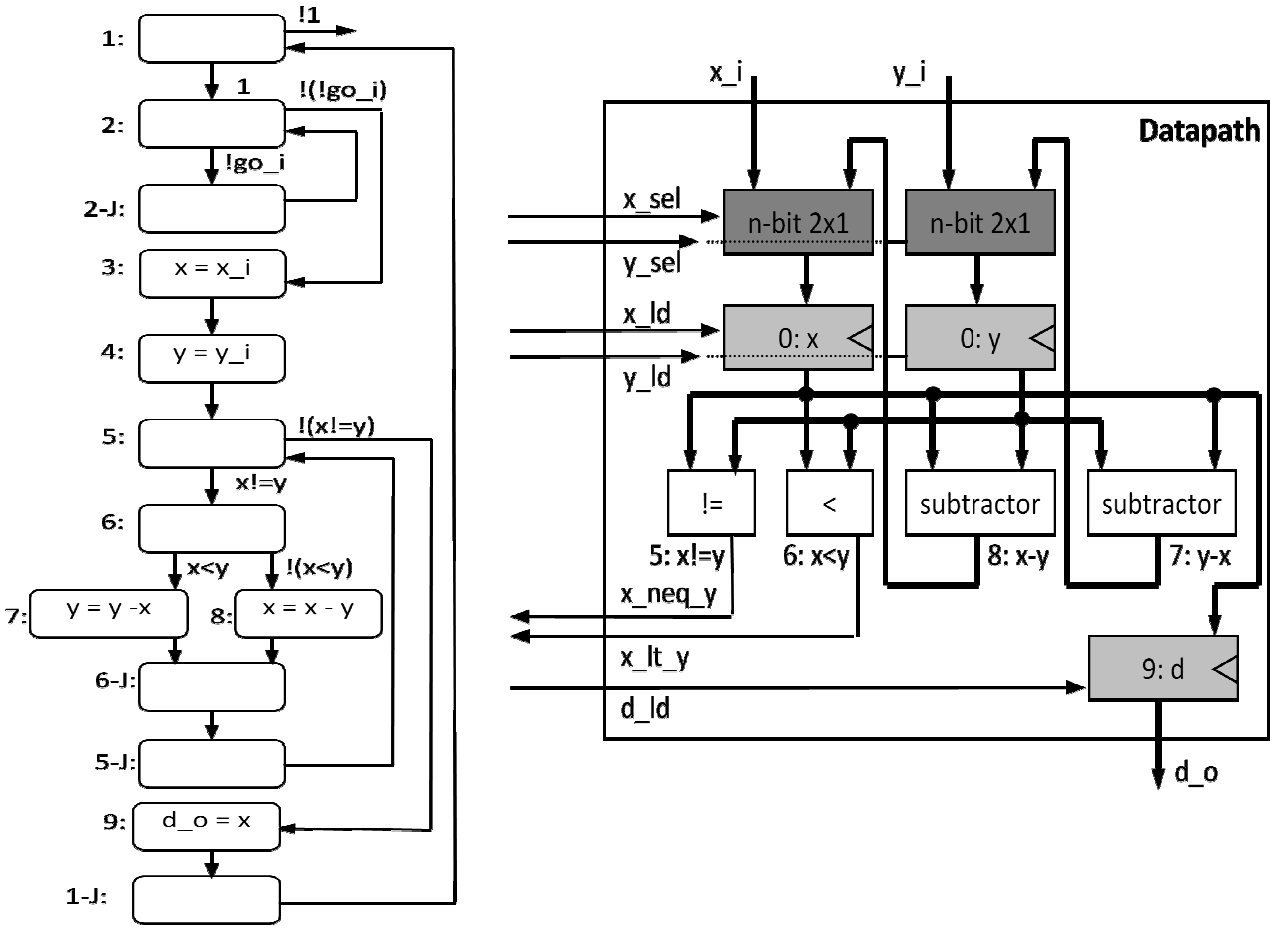


Figure: FSMD and Data path

2. Controller Design

→ Modify FSMD into FSM

→ FSM has the same states and transitions as the FSMD. However, complex actions and conditions are replaced by Boolean ones.

- → Replace every variable “write” by action that select signals of the mulitplexers infornt of the variable’s register.
- → Only then assert the load signal to the register.
- → Replace every logical operation in a condition by a corresponding functional unit control output.

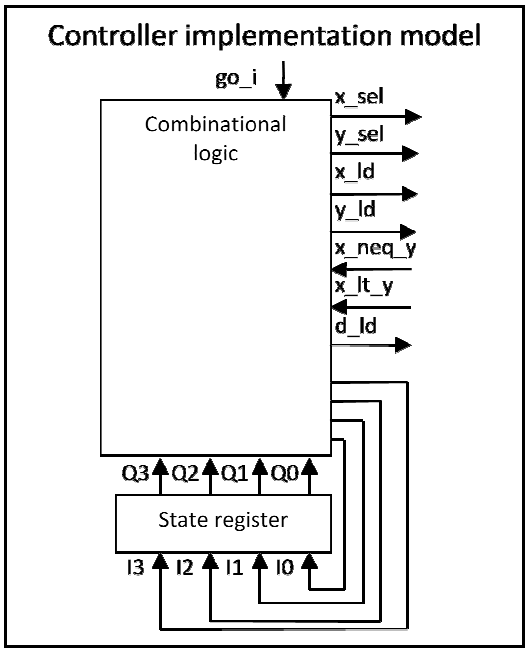


Figure: Controller model

State table for controller

Since there are 7 inputs, there will be 27 (128) rows in state table. We have reduced the rows in the state table by using * for some input combinations.

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

Figure: State table for GCD example

Completing the GCD custom single-purpose processor design.

1. We finished the datapath.
2. We have a state table for the next state and control logic.
   - → All that is left is combinational logic design

2.6 Optimizing custom single purpose processor

In the previous GCD example, we ignored several simplification opportunities.

- → FSM had several states that obviously do nothing and could have been removed.
- → Data path has two addresses whereas one would have been sufficient and so on.

In a typical real custom design task, we typically wish to optimize every possible parameter. Optimization is the task of making design metric value the best possible.

Some of the optimization opportunities are:

- **Original program**
- **FSMD**
- **Datapath**
- **FSM for controller**

1. Optimizing the original program

Let us analyze the program attributes and look for areas of possible improvement

a) Number of computations
b) Size of variable
c) Time and space complexity
d) Operation used
   - → Like multiplication and division are expensive so avoid them.

Original program

Let's re-write our original algorithm and try to develop an alternate algorithm that is more efficient.

Original program

```
0:      int x,y;
1:      while (1){
2:              while(!go_i);
3:                      x=x_i;
4:                      y=y_i;
5:                              while(x!=y){
6:                                      if(x<y)
7:                                              y=y-x;
                                        else
8:                                              x=x-y;
                                                }
9:                                      d_o=x;
                }
```

Let us see, how many iteration does it takes to find GCD of (42,8)

(42,8)→(34,8) → (26,8)→(18,8) → (10,8)→(2,8) →(2,6)→(2,4) →(2,2)

Thus, we can see it takes 9 iteration to complete. Now, let us refine our algorithm by using the next approach as shown below.

Optimized Program

```
 0: int  x, y, r;
 1: while (1) {
 2:   while (!go_i);
      // x must be the larger number
 3:   if (x_i >= y_i)  {
 4:     x=x_i;
 5:     y=y_i;
      }
 6:   else {
 7:     x=y_i;
 8:     y=x_i;
      }
 9:   while (y != 0)  {
10:     r = x % y;
11:     x = y;
12:     y = r;
      }
13:   d_o = x;
    }
```

Again let's see how many iteration does it takes to find GCD of (42,8)

(42,8)→(8,2) → (2,0)

Thus this algorithm is far more efficient in terms of time it takes for computation of result. Thus the choice of algorithm can have the biggest impact on the efficiency of the design processor.

Note:

We can replace the subtraction operations with modulo operation in order to speed up the program.

Optimizing the FSMD

Some of the areas of possible improvement are

A. Merge state
   - States with constants on transitions can be eliminated, transition taken is already known
   - States with independent operations can be merged.

B. Separate states
   - States which requires complex operation can be broken into smaller states to reduce hardware. E.g. if we have statement as

     A=b*c*d*e

     This statement requires three multipliers as multipliers are expensive. Breaking above operations into smaller operations can reduce number of multipliers.

     t1=b*c

     t2=d*e

     A=t1*t2

     This statement requires only one multiplier as the same multiplier cab be shared

C. Scheduling
   - The timing of output operation could be changed. We can generate GCD output in fewer clock cycles. However changing the timing is not always acceptable.

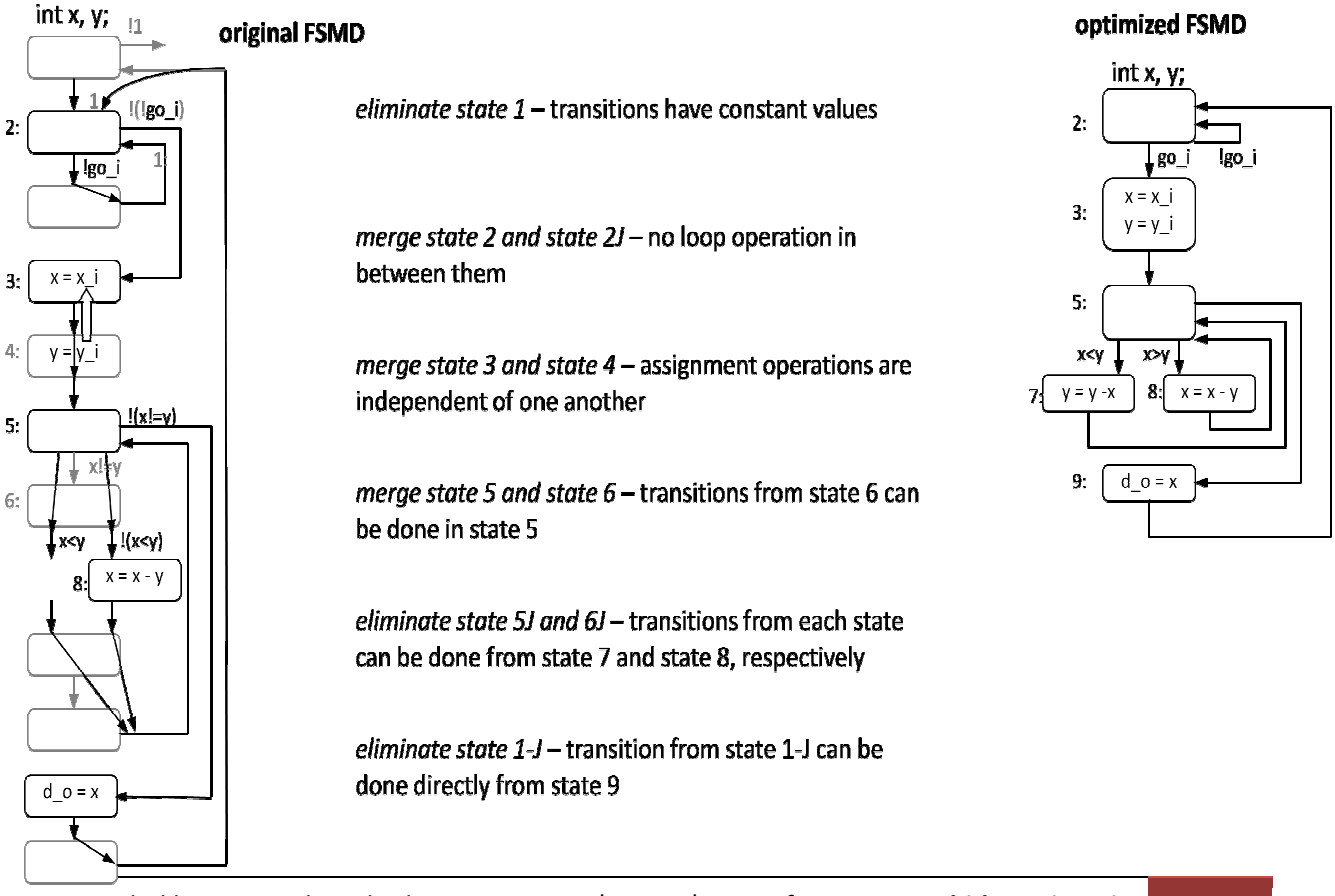


Figure: Optimizing the FSMD for the GCD example a) original FSMD and optimization, and b) optimized FSMD

Optimizing Datapath

A. Sharing of functional units
   A unique functional unit for every arithmetic operation is not always necessary. Many arithmetic operations can share a single functional unit, if that functional units supports those equations, and those operations occurs in different states.
   E.g. instead of using two subtractors, we could have used only one subtractor and used multiplexer to choose whether operation should be (X-Y) or(Y-X).
B. ALU supports a variety of operations; it can be shared among operations occurring in different states.

Optimizing FSM of controller

A. State Encoding

- State encoding is the task of assigning a unique bit pattern to each state in FSM.
- Any assignment in which the encoding are unique will work, but size of state register as well as size of the combination logic may differ for different encoding.
  For example:

A	**00**	**A**	**11**
B	**01**	**B**	**10**
C	**10**	**C**	**01**
D	**11**	**D**	**10**

For FSM with n states where n is a power of two, there are n! Possible encoding. We can order the states and assign binary encoding starting with 00,…..00 for first state 00,…01 for second state and so on. Thus there are n! Possible ways for n states.
CAD tools helps in searching for the optimum state encoding.

B. State Minimization

1. It is the task of merging equivalent states into a single state.

Note:

Two states are said to be equivalent if for all possible input combinations the two states generates the same outputs and transistors to the next same state. Such states can be merged.

3.0 Software Design Issue

3.1.0 Introduction

A general purpose processor is a programmable digital system intended to solve computation problems in a large variety of applications.

The unit cost of the processor may be very low, because the manufacturer spreads NRE over large number of units. Motorola sold half a billion 68HC05 microcontroller in 1996 alone.

Since, processor manufacturer can spread NRE cost over large number of units; the manufacturer can afford to invest large NRE cost into the processor design, without significantly increasing the unit cost.

3.1.1 Basic Architecture

A general purpose processor consists of data path and a control unit, tightly linked with a memory unit.

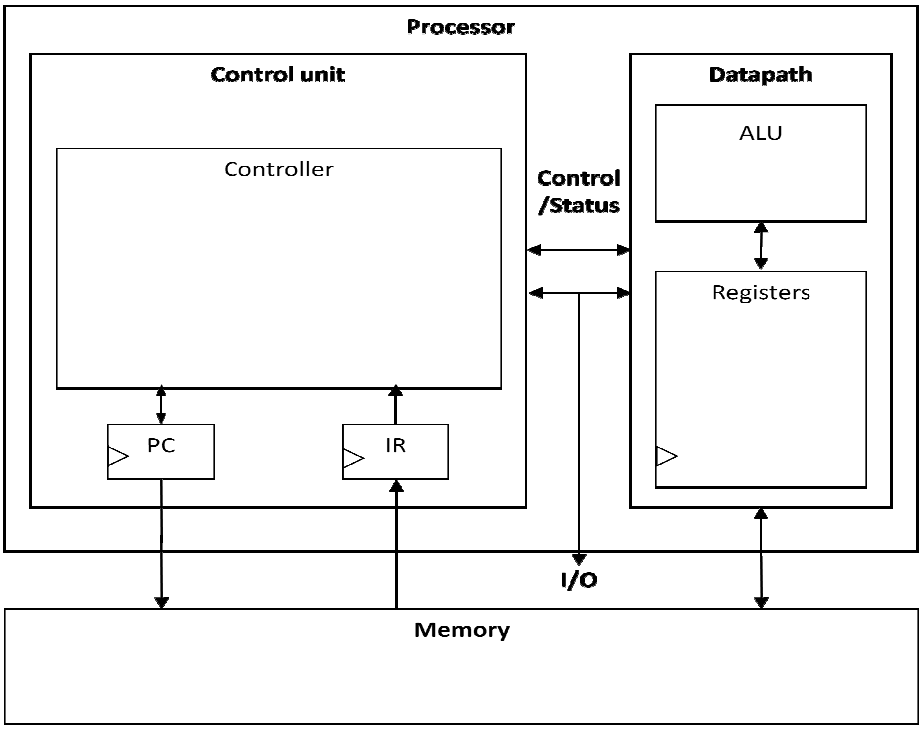


Figure: General Purpose Processor basic architecture.

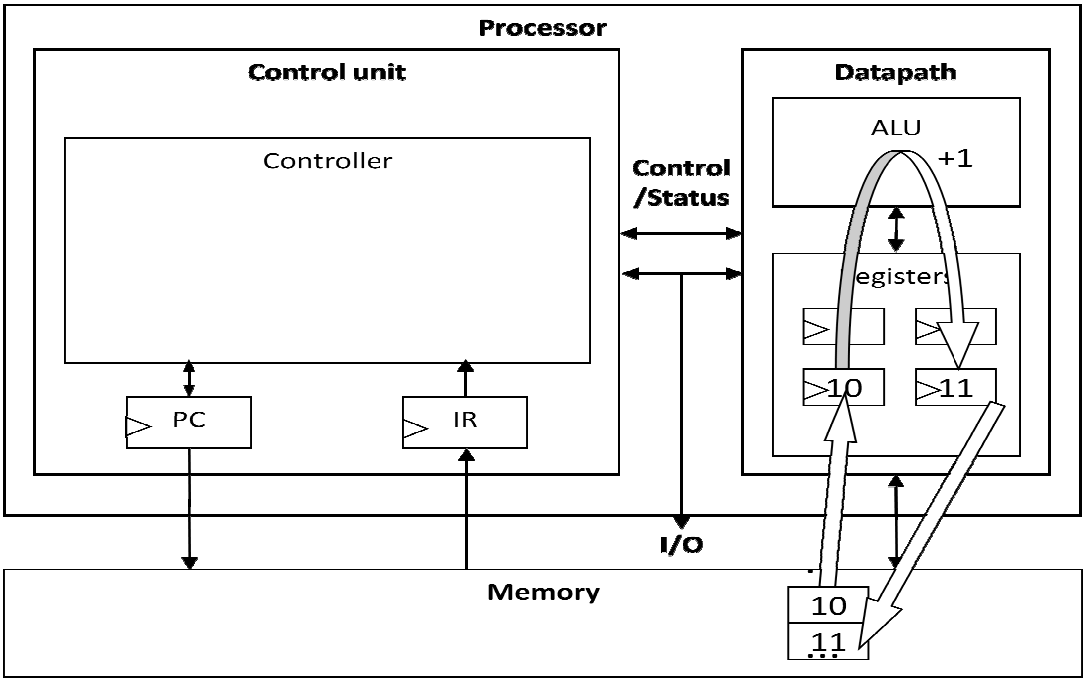
3.1.2 Data path

It consists of circuitry for transforming data and for storing temporary data. The datapath contains an arithmetic-logic unit (ALU) capable of transforming data through operation like addition, subtraction, logical AND, logical OR inverting, shifting etc.

Storing of temporary data is done by using registers.

a) Data is brought in from memory but not yet used by ALU.
b) Data coming from ALU will be needed for later ALU operation or may be sent back to memory.
c) Data that must be moved from one memory location to another.

Internal data bus carries data within the data path, while the external data bus carries data to and from the data memory.



Processor Size:

- → A processor size is measured by bit-width of the data path components.
  An N-bit processor may have N-bit-wide register, an N-bit-wide ALU, an N-bit-wide internal bus over which data is brought in and out of the data path.
- → Common processor sizes includes 4-bit, 8-bit 16-bit ,32-bit and 64-bit.
- → The 8051/8052 microcontroller is an 8-bit processor

3.1.3 Control Unit

Control unit consists of circuitry for retrieving program instructions and for moving data to/from and through the data path according to those instructions.

The control unit consists of program counter (PC) that holds the address of the next instruction to fetch an instruction register (IR) to hold the fetched instruction.

The controller determines the next value of the program counter. For a non-branch instruction, the controller's increments the program counters. For a branch instruction, the controller looks at the datapath of the status signal and the IR to determine the appropriate next address.

The program counter bit-width represents the processor address size. The address size determines the number of directly accessible location, called the address space.

It the address size is (M), the address space will be 2M e.g. a 16-bit pc can directly address 2^{16}=65536 memory location.

For each instruction, the controller typically sequence through the following

a. Fetch stage
b. Decode stage
c. Fetch operand stage
d. Execute stage
e. Store stage

Fetch stage

- → Get the instruction into IR

Decode Stage

- → Determine what the instruction means

Fetch operands

→ Move data from memory into appropriate data path registers.

Execute

→ Feeding appropriate data path registers

Store

→ Write data from register to memory.

Clock Cycle:

The longest time required for data to travel from one register to another register. The path through which the data travels and that results in longest time is called critical path.

Inverse of clock cycle is clock frequency, measured in cycles per second e.g. if clock cycles =10ns, then clock frequency =100 MHz for 8051/52 microcontroller, clock frequency is 11.0593 MHz.

3.1.4 Memory:

Registers acts as temporary or short term storage entities. While memory services as processor medium and long-term storage requirements.

We can classify storage information as program and data. Program information consists of the sequence of instruction that cause the processor to carry out the desired system functionality. Data information represents the value being i/p, 0/p and transformed by the program. Both program and data can be stored together or separately.

Princeton Architecture (Von-Neumann)

→ In von-Neumann architecture both program and data share the same memory space.

→ Since single bus for both data and instruction architecture is much simpler.

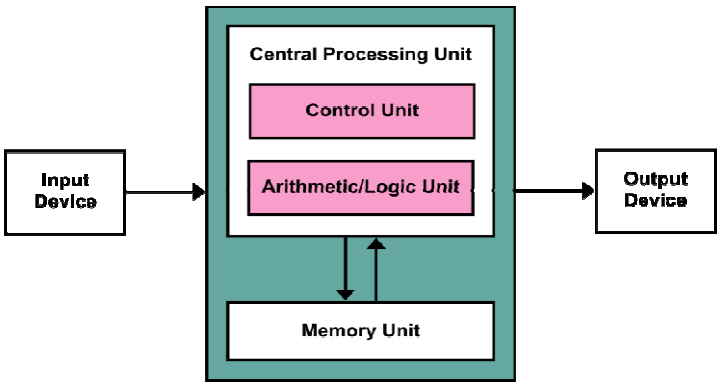


Figure: Von-Neumann architecture

Harvard Architecture

→ Program memory space is kept distinct from data memory space.

→ Harvard architecture can perform both data instruction fetch simultaneously.

→ 8051/52 microcontroller has Harvard architecture.

→

Memory can be further divided into RAM and ROM. An embedded system uses ROM for program memory, because an embedded system program does not change. In embedded system usually constant data are stored in ROM, however data that changes are stored in RAM.

Memory can be on-chip or off-chip. On-chip resides on the same IC as the processor, while off-chip resides on a different IC.

The processor can usually access on-chip memory much faster than off-chip memory, usually in one clock cycle but finite IC capacity leads to limited on-chip memory.
Off chip memory are quiet slow, to reduce the time needed to access memory, a local copy of a portion of memory called cache memory may be used. Cache memory resides on-chip and often uses very expensive static RAM which is much faster than Dynamic RAM.

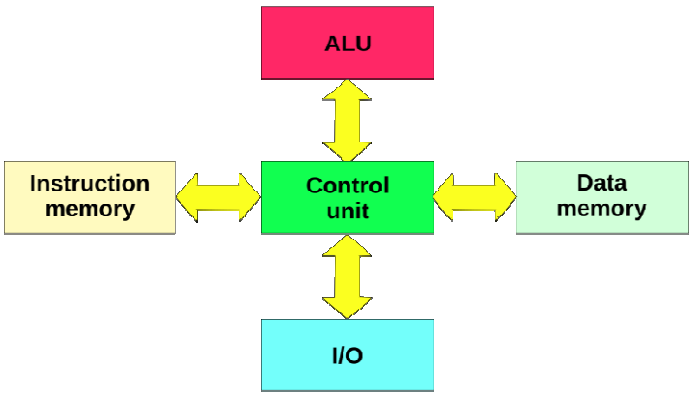


Figure: Harvard Processor

3.1.5 Principle of Cache Memory

- → At a particular time, a processor access a particular memory location, then the processor is likely to access the neighboring memory location in the near future.
- → When we first access a location in memory, we copy that location and some of its neighbor in to cache, and then access the local copy form the cache.
- → When we access another location, we first check a cache-table for memory location to see if a copy of location resides in cache. If the copy resides in cache table, it is called cache hit. I cache does not exist we called it as cache miss.

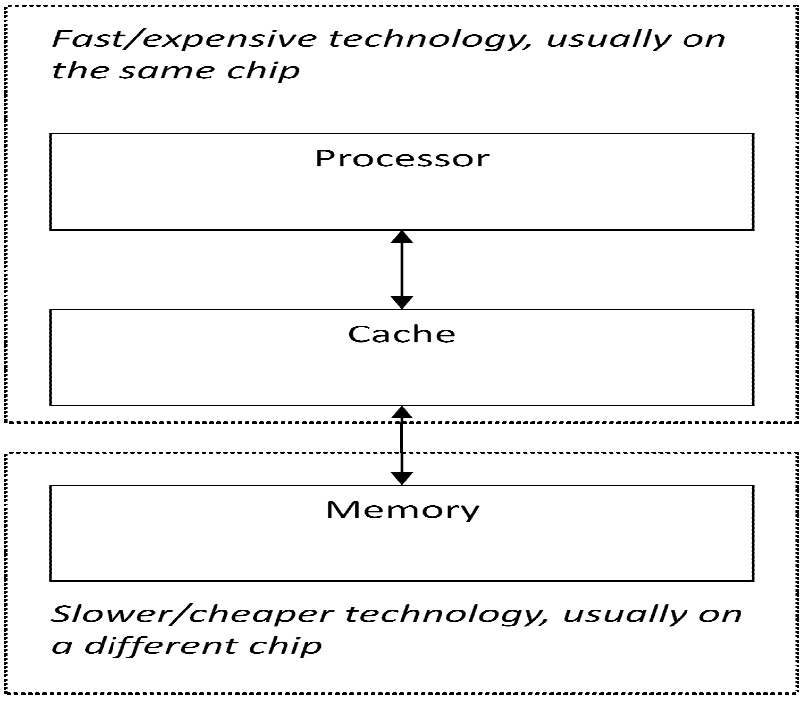


Figure: Cache Memory

3.1.6 Operation

3.1.6.1 Instruction Execution

Microprocessor instruction execution consists of following basic stages

1. **Fetch instruction:**
   - → The task of reading the next instruction from memory into the instruction register
   - → Program counter always points to next instruction and instruction register always holds the fetched instruction.

2. **Decode Instruction:**
   - → The task of determining what operation the instruction in the instruction register represents(e.g. add, mov etc)

3. **Fetch operand:**
   - → The task of moving the instruction's operand's data into appropritate register.

4. **Execute operation:**
   - → The task of feeding the appropriate register through the ALU and back into an appropriate register.

5. **Store result**
   - → The task of writing a register into memory.

3.1.7 Pipelining

Instruction pipelining is a technique that implements a form of parallelism called instruction level parallelism within a single processor, thus allows faster CPU throughput. Rather than processing each instruction sequentially, each instruction is split up into a sequence of steps, so different instruction can be executed parallel and can be processed concurrently.
Once, the pipe-line architecture is full we get concurrent execution of operations.

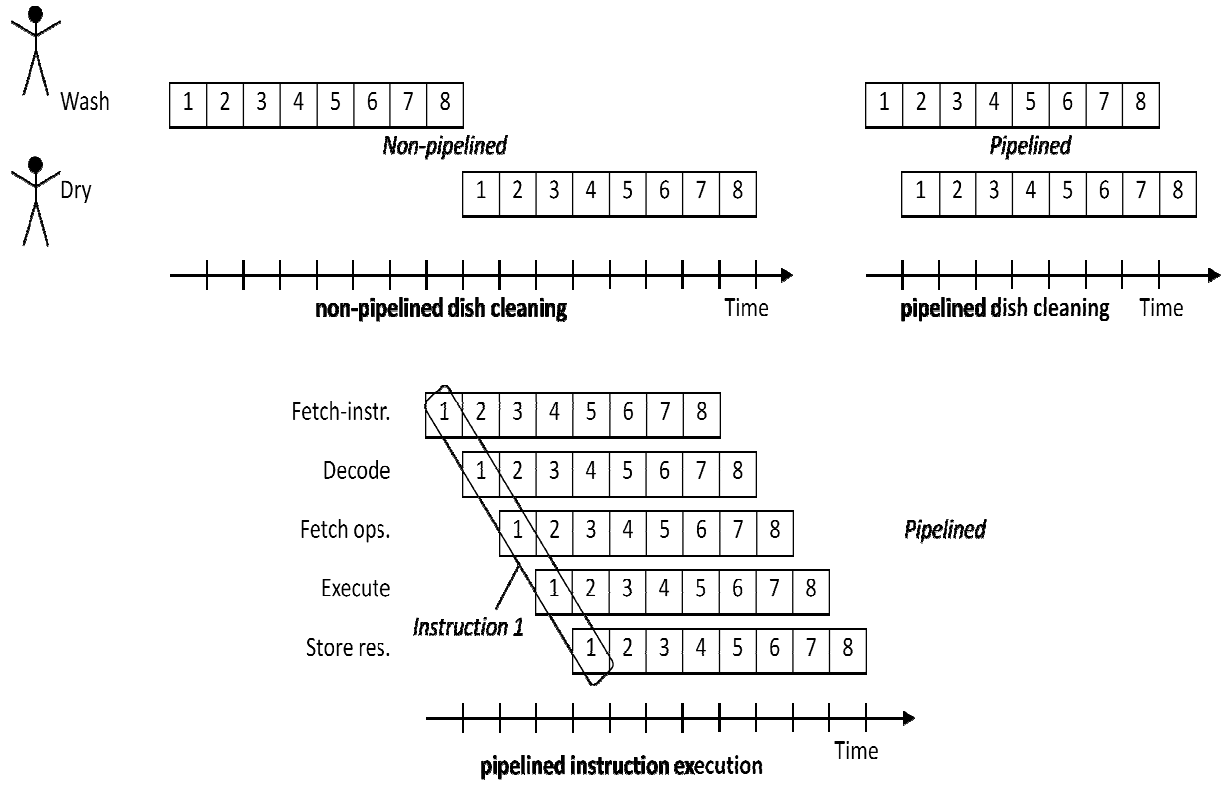


Figure: non-pipelined dish cleaning pipelined dish cleaning and pipelined instruction execution

Let us make an analogy to explain the concept of pipelining. In the first approach the first person washes all the dishes and second person dries all the dishes. Assuming 1 minute per dish per person, this approach requires 16 minutes.
This approach is inefficient since at any instant of time only one person is working while other is idle. In the second approach the second person begin to dry the dish immediately after it has been washed. This approach requires only 9 minutes. 1 minute for the first dish to be washed and then 8 more minutes until the last dish are finally dried.

Problems with pipelining
Branch hazard:
Branches pose a problem for pipelining, since we don't know the next instruction until the current instruction has reached the execution stage.
One solution is to **stall the pipeline**, waiting for the execution stage before fetching the next instruction.

Another solution is to guess the branch instruction so that that instruction can be loaded ahead for concurrent output. Modern processors has sophisticated **branch predictors** built in which minimizes branch hazard.

3.2.0 Superscalar and VLIW architecture

- → A **Superscalar architecture** is one in which several instructions can be initiated simultaneously and executed independently
- → Pipelining allows several instructions to be executed at the same time, but they have to be in different pipeline stage at a given moment.
- → Superscalar architecture includes all features of pipelining but in addition there can be several instructions executing simultaneously in the same pipeline stage.

- → **VLIW architectures** rely on compile-time detection of parallelism. The compiler analysis the program and detects operations to be executed in parallel. Such operations are packaged into a large instruction.
- → After one instruction has been fetched all the corresponding operations are issued in parallel.
- → No hardware is needed for run-time detection of parallelism
- → The instruction window problem is solved; the compiler can potentially analyse the whole program in order to detect parallel operation.

3.3.0 Programmer View

- → A programmer writes the program instructions that carry out the desired functionality on the GPP(general purpose processor)
- → A programmer does not need to know the detailed information about the processor architecture, but instead need to know what instructions can be executed.

There are three levels of programming

1. **Assembly level:**
   - → In assembly language, program is written using processor specific instruction.
2. **Structured level**
   - → In structured level, programming language like c/c++, java are preferred, which are processor independent instruction.

→ A compiler automatically translates the high level instruction to processor-specific instruction.

3. **Machine level**
   → Machine level is lower than assembly level. Program instructions are written in binary format. Such programs are tedious and are rarely used.
   → Due to invention of assembler (which translates assembly language in to machine language) program are not written at machine level.

3.4.0 Operating system

An OS is a layer of software that provides low-level services to the application layer. The task of managing the application the application layers layer involves loading and executing of program sharing and allocating system resources and protecting these resources from being captured by non-owner programs.

One of the most important resources of a system is the CPU which is typically shared among a number of executing program. The OS is responsible for deciding what happens when a new program wants to obtain the processor. The OS is also responsible for deciding what program to run next and for how long.

This is called process scheduling and it is determined by the operating system's pre-emption policy. OS provides software required for servicing various hardware interrupt. It also provides device drivers services for driving peripherals.

On system start-up, an OS initiates all the peripherals devices such as disk-controller, timers and I/O devices and install hardware interrupts service routine to handle various signals generated by these devices. It also installs software interrupts to process system calls (call made by High Level application software to request OS service)

A system call is a mechanism for an application to invoke the OS. It is similar to a procedural function called in high-level programming languages.

When a program requires some service from the OS, it generates a predefined software interrupt that is serviced by the OS.

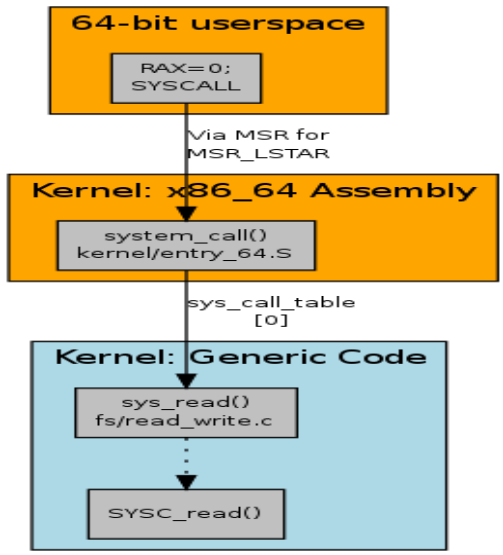
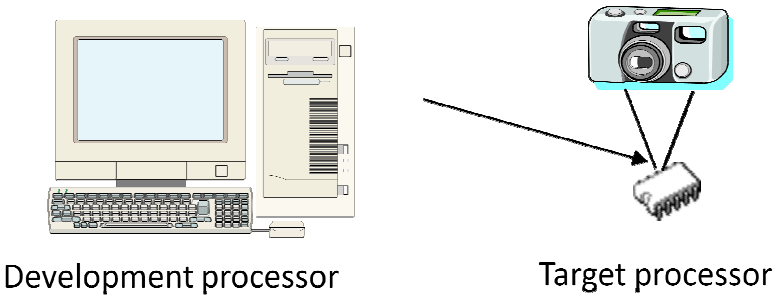


Figure: invocation of system call

In short, OS abstracts the detail of the underlying hardware and provides the application layer an interface to the hardware through the system call mechanism.

3.5.0 Development Environment

Development environment comprises of general software design tools that are used by embedded system designer in system design, system testing, debugging and verification of embedded software.



3.5.1 Design flow and tools

Development of embedded software comprises of two basic environments

- Host environment
- Target environment

3.5.1.1 Host Environment

Host environment consists of host processor. The host environment consists of necessary tools for writing and debugging the program. Host environment can be our personal computer where we write, debug and simulate our design as well as check and debug the system.

3.5.1.2 Target Environment

The target environment consists of target processor along with other peripherals. The verified program in Host environment is loaded to the target processor which is a part of embedded system. e.g. we may use Pentium processor to develop our program and use Motorola as target. Some time both host and target processor can be same.

→ The general design flow starts with writing source code in an editor. The editor may helps in syntax correction. Usually we write source code in multiple numbers of files for modularity.
→ Then we compile/assemble the code in each file by using compile/assembler into corresponding binary files.
→ Then we use a linker to combine the binary files into final executable file. All of these steps are collectively known as implementation phase.

The next phase is debugging/testing phase. In this phase we use debugger to test the executable file which is the output of linker. Debugging can be done by using profiler as well. A profiler is used to pinpoint performance bottleneck of the program. Profiler evaluates how well program perform on new architecture and how well instruction scheduling or branch prediction algorithm is performed. If we discover error or performance bottlenecks, we returns to the implementation phase, make improvement and repeat the process.

→ Typically, all of the above mentioned tools are combined into a single program called IDE (Integrated Development Environment), which greatly simplifies the design process.
→ The implementation phase is the process of editing compiling, assembling and linking the program.
→ The verification phase is the process of testing the final executable.

3.5.2 Development Tools for Implementation phase

1. **Assembler**
   Assembler translates assembly instruction to binary instruction. All the mnemonics are converted into machine language. The mapping of assembly instruction to machine instruction is unique.
2. **Compiler**
   Compiler translates high-level (structured) instruction into machine instruction. Compiler design has advanced tremendously, thus they are capable of doing many optimization in source code.
   A cross compiler executes on the host environment but create the code for the target processor.
3. **Linker**
   Combines machine instructions from various separately compiled/assembled file into a single file. A linker also tries to eliminate binary code associated with uncalled procedures and function as well as memory allocated to unused variables in order to reduce the overall program footprint.

3.5.3 Testing and Debugging

Testing and debugging is the major part of the overall design process, Errors are inevitable in large program. The most common method of verifying the correctness of a program is executing it with ample input data and checks the programs behavior, using boundary cases.
Many embedded system are real time. A distinguishing characteristic of a real-time system is that it must compute correct results within a specified amount of time.
In addition, a program running in an embedded system works in conjunction with input sensors and output transducers.

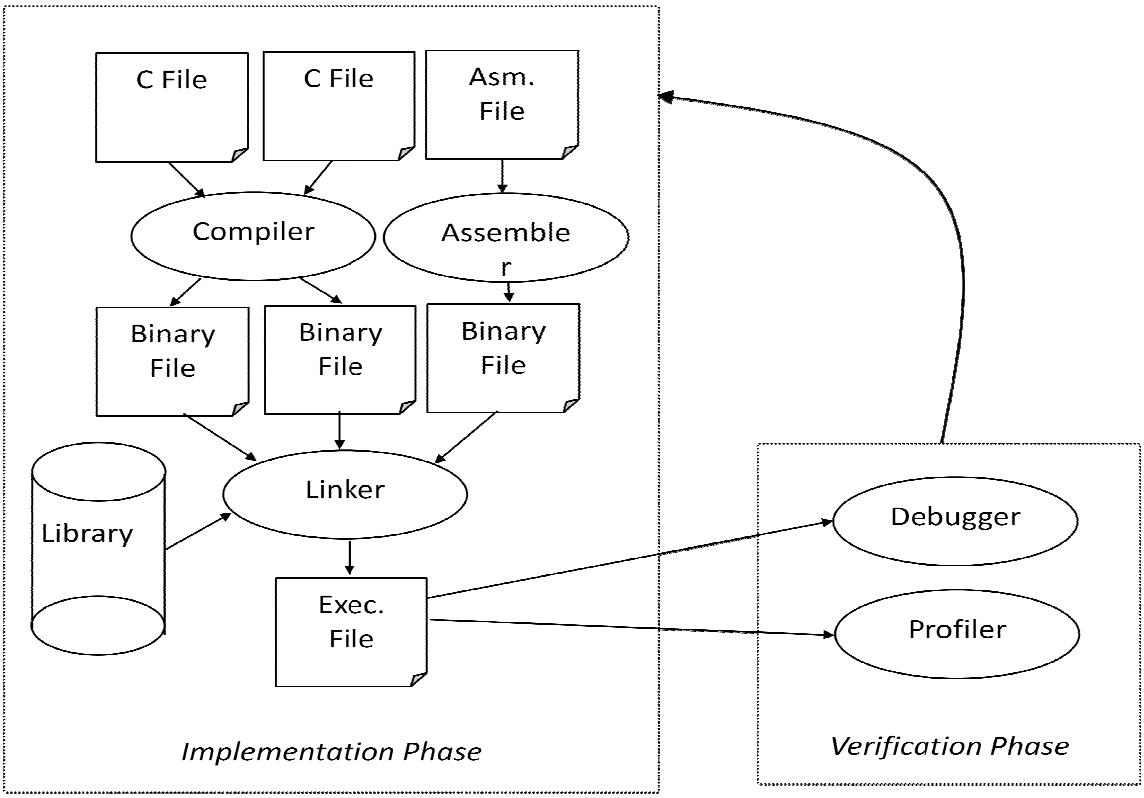


Figure: Software Development Process

3.5.3.1 Tools for Debugging/Testing

1. **Debuggers**

   Debuggers helps programmer evaluate and correct their program. They run on host processor and support stepwise program execution, executing one instruction and then stopping, proceeding to the next instruction when instructed by the user.

   Debugger also supports program running up to break point, which are crucial for large and complex programs.

   Debugger also allows programmer to see the internal register contents, memory contents.

   Since debugger run the code of the target processor residing in the Host processor, such debugger mimics the target processor and are called **instruction set simulator** (ISS) or virtual machine.

2. **Emulators:**

   Emulators support debugging of the program while it executes on the target processor. An emulator typically consists of a debugger coupled with a board connected to the development processor. The board consists of the target processor. In circuit emulator enables the control and monitor of the program execution in the actual embedded system circuit. In circuit emulators are available for most all the processor used in embedded system.

3. **Device programmer**

   Device programmer downloads a binary machine program from the development processors memory into the target processor's memory. Once the target processor has been programmed the entire embedded system can be tested in the most realistic form.

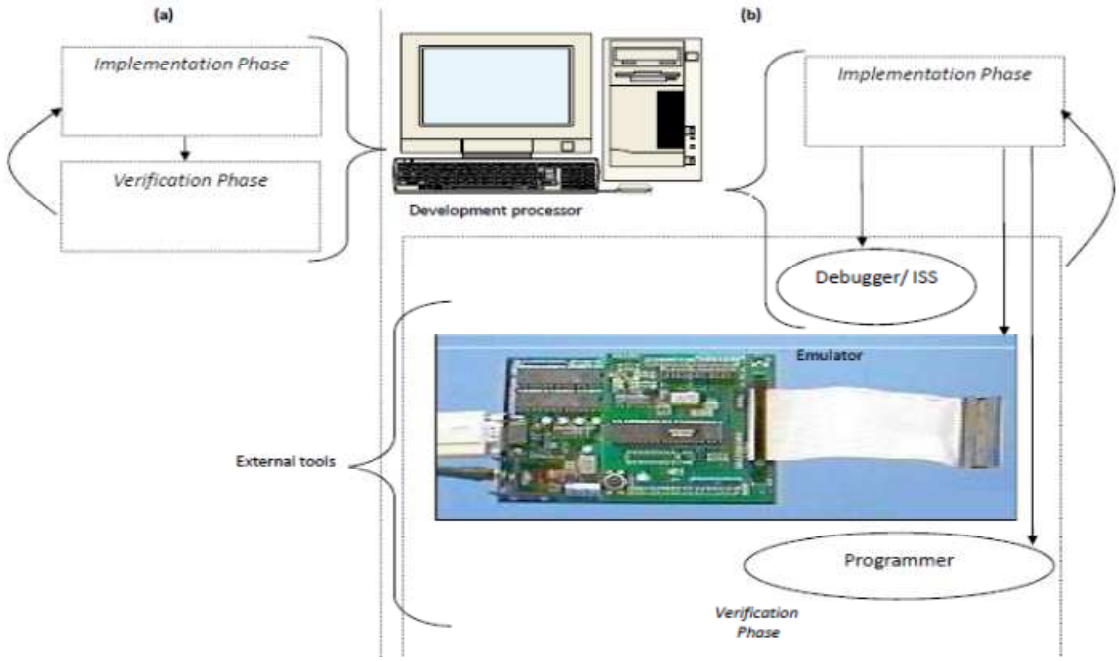


Figure: Testing and debugging methods

In summary, programs intended for embedded system can be tested in three ways.

- Debugging using ISS

- Emulation using an emulators
- Field testing

→ Debugging using IDE is fast but is inaccurate since it only interacts with system and environment to limited degree.
→ Design cycle using emulators is quiet long, since the code must be downloaded into the emulator hardware; however it can interact with system and allow for more accurate testing.
→ The design cycle for field testing is very long. The processor must be programmed using programmer and kept in target environment. In this method the system interacts with the external environment more freely however debug control is very less.

3.6.0 Application-Specific Instruction-set Processor

General purpose processors can't fulfill the performance, power, cost or size demand of embedded applications that requires high computation power and very specific functionality. Custom single-purpose processors are inflexible and often too prohibitive. A solution is to use an instruction set processor that is specific to that application domain.

ASIPs provide an intermediate solution between GPPs and custom single purpose processor because they are designed for an application. ASIPs are instruction-set processor and can be programmed by writing software resulting in short time to market and good flexibility. ASIPs tend to come in three major verities, microcontrollers, DSPs and less-general ASIPs.

1. Microcontrollers:

Microcontrollers are basically specific to application that performs a large amount of control oriented task. Such microcontrollers includes several peripherals devices such as timers, ADCs, Serial communication etc on same IC package along with processor.

Microcontrollers provide direct pin access the programmer. Thus it is easy to connect sensors/transducer, monitor the status and so on in microcontroller.

2. Digital Signal Processors(DSPs)

Digital signal processors (DSPs) are highly optimized for processing large amount of data. DSPs contain numerous register flags, memory blocks, multipliers and other arithmetic units.

DSPs also provide instructions that are center to DSP such as filtering, amplifying, interpolation, averaging etc.

In DSPs, frequently used arithmetic functions such as multiply and accumulate are implemented in hardware thus, execute order of magnitude faster than software implementation.

DSPs may allow for execution of some function in parallel thus resulting in a boost in performance.

DSPs are widely used in application like image processing, audio processing and complex processing. DSPs also incorporate many peripherals that are useful in signal processing on a single IC.

3. Less General ASIP

In contrast to microcontrollers and DSPs which are used in variety of embedded system less-general ASIP are designed to perform some very domain specific processing while allowing some degree of programmability.

E.g. ASIP designed for network hardware may be diesigned to be programmable with different network routing algorithm, checksum and packet processing protocols.

3.7.0 Selecting a microcontroller:

The choice of processor depends on technical and non-technical aspects.

- → From technical prospective, one must choose a processor that can achieve the desired speed with certain power, size and cost constraints.
- → From non-technical aspect choose a processor based on its prior expertise with a processor, its development environment, special licensing arrangement and so on.

Issue on speed: how to evaluate processor's speed

Speed of processor is particularly difficult to measure and compare.

1. Clock speed
   Comparing processor based on number of instruction per clock cycle may be difficult, because number of instructions per clock cycle differ greatly among processors.
2. Instruction per second
   Comparing processors using instructions per second is also not feasible because the complexity of each instruction differ greatly among processors.
   For e.g. one processor may require 100 instructions while another processor may require 300 instructions to perform the same computation.
3. Dhrystone
   A fair new means of comparing processor is the Dhrystone benchmark. The Dhrystone benchmark is simple a program which performs no work. It is used to compare the performance of various processors. It focuses on processor's arithmetic and string handling capacities. Such program is executed on different processors, thus a processor is said to be able to execute so many

Processor	Clock speed	Periph.	Bus Width	MIPS	Power	Trans.	Price
General Purpose Processors							
Intel PIII	1GHz	2x16 K L1, 256K L2, MMX	32	~900	97W	~7M	$900
IBM PowerPC 750X	550 MHz	2x32 K L1, 256K L2	32/64	~1300	5W	~7M	$900
MIPS R5000	250 MHz	2x32 K 2 way set assoc.	32/64	NA	NA	3.6M	NA
StrongARM SA-110	233 MHz	None	32	268	1W	2.1M	NA
Microcontroller							
Intel 8051	12 MHz	4K ROM, 128 RAM, 32 I/O, Timer, UART	8	~1	~0.2W	~10K	$7
Motorola 68HC811	3 MHz	4K ROM, 192 RAM, 32 I/O, Timer, WDT, SPI	8	~.5	~0.1W	~10K	$5
Digital Signal Processors							
TI C5416	160 MHz	128K, SRAM, 3 T1 Ports, DMA, 13 ADC, 9 DAC	16/32	~600	NA	NA	$34
Lucent DSP32C	80 MHz	16K Inst., 2K Data, Serial Ports, DMA	32	40	NA	NA	$75

Sources: Intel, Motorola, MIPS, ARM, TI, and IBM Website/Datasheet; Embedded Systems Programming, Nov. 1998

Figure: General Purpose Processor

Dhrystone per seconds.

- **MIPS**(Millions of instruction per second)
  1 MIPS=1757 Dhrystone per second(based on Digital VAX 11/70; first computer to execute 1 MIPS).
  Some time MIPS are also referred by Dhrystone MIPS.
  → 750 MIPS=750×1757=1,317,750 Dhrystone/second

The use and validity of benchmark data is a subject of great controversy. Thus there is a clear need for benchmark that measure performance of embedded processor. One such effort is EEMC (pronounced as embassy) after EDN embedded benchmark consortium.
EEMBC has five benchmarking suites of program corresponding to automotive/industrial, consumer electronics, networking, office automation and telecommunication.

3.8.0 General Purpose processor design

A general purpose processor is just a single-purpose processor whose purpose is to process instruction stored in a program memory.
A general purpose processor can also be designed using single purpose processor design technique.

a. Create FSMD describing the processor's behavior
b. Build a data path that can carry out operation of the FSMD
c. Rewrite the FSMD to obtain the FSM of the controller.
d. Design the controller using combinational or sequential logic.

Example
Let us design a general- purpose processor having the basic architecture as shown below.

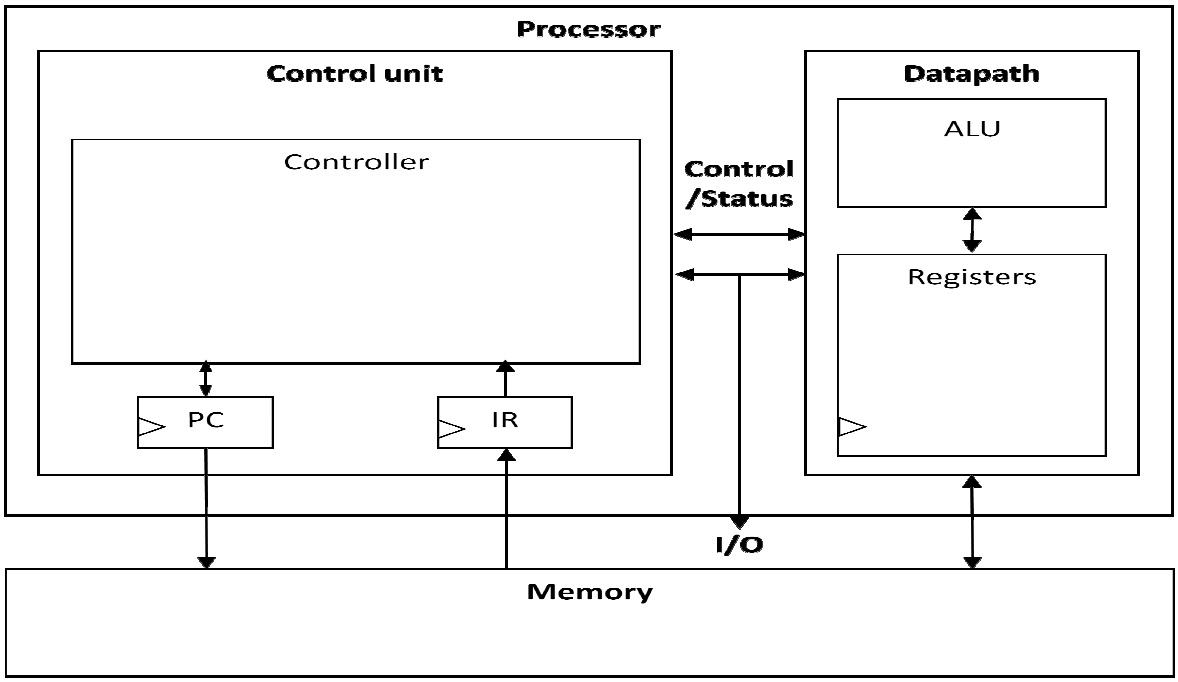


Figure: General Purpose processor basic architecture

This processor supports the following instruction sets.

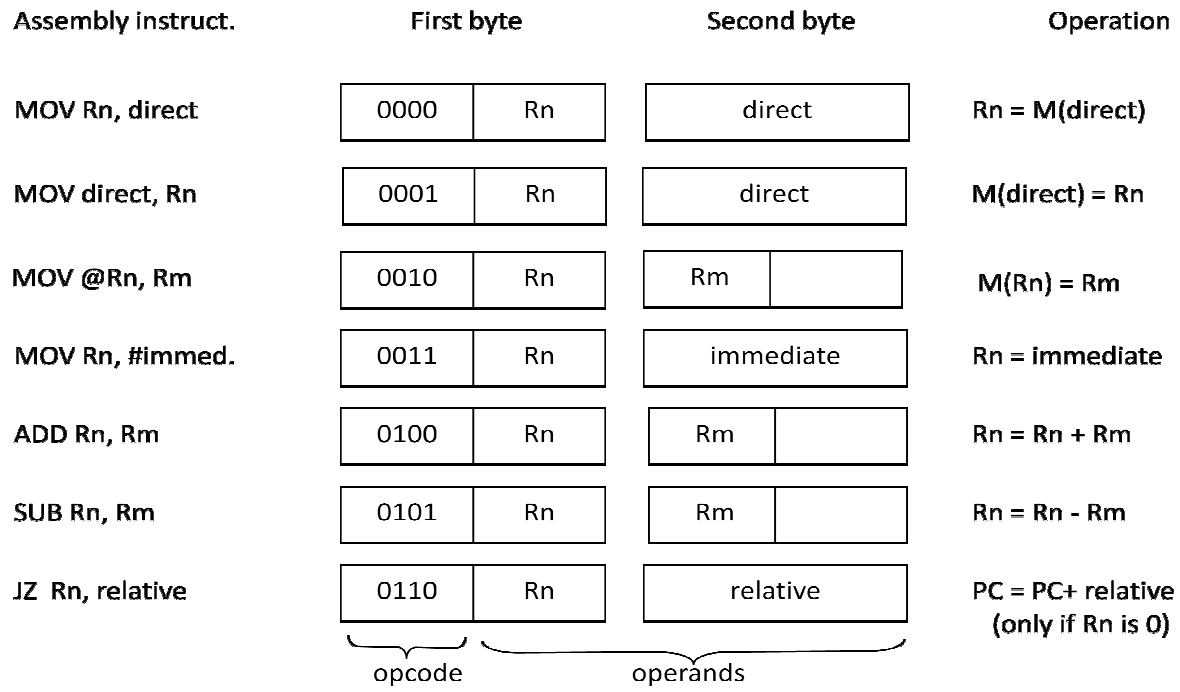


Figure: A simple (trivial) instruction sets

1. We begin by creating FSMD as shown below which describe processor behavior

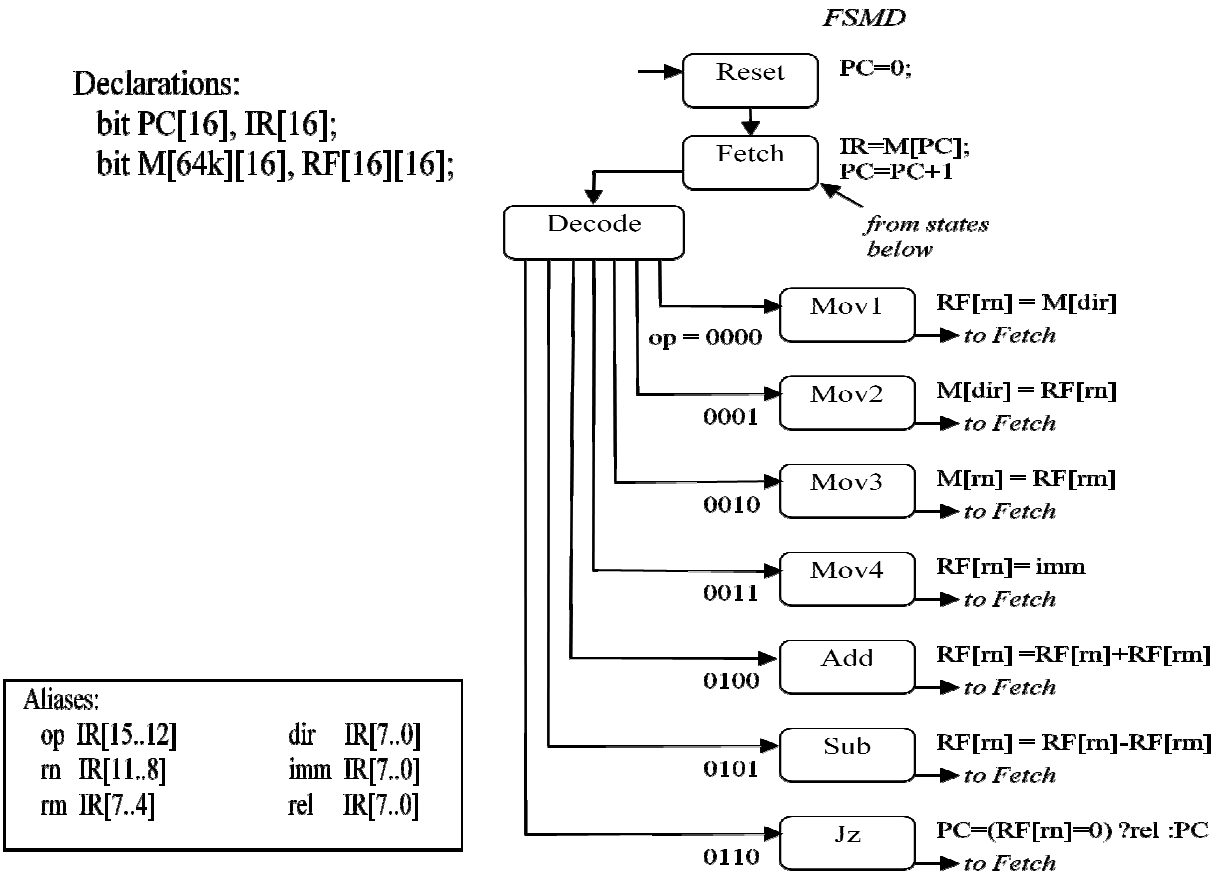


Figure: A simple processor FSMD

We have

a) 16 bit program counter PC
b) 16-bit Instruction Register IR
c) 64×16 bit memory M
d) 16×16 bit register file RF

→ FSMD initial state is reset state; which clears PC.
→ The fetch state reads M[PC] into IR
→ The decode state adds the extra cycle necessary to update IR
→ The decode state also detects a particular instruction opcode causing a transition to corresponding execute state.

2. We now build a data path that carry out the operation of FSMD

a) Create a storage device (Register) for each declared variable. So we instantiate register PC and IR, memory M and register file RF.
b) Create functional units to carry out all the FSMD operation. We can use single ALU capable of carrying out all the operations.
c) Add the connections among the components port as required by the FSMD operations, adding multiplexor when there is more than one connection being input to port.
d) Finally we create the unique identifier for each control signals.

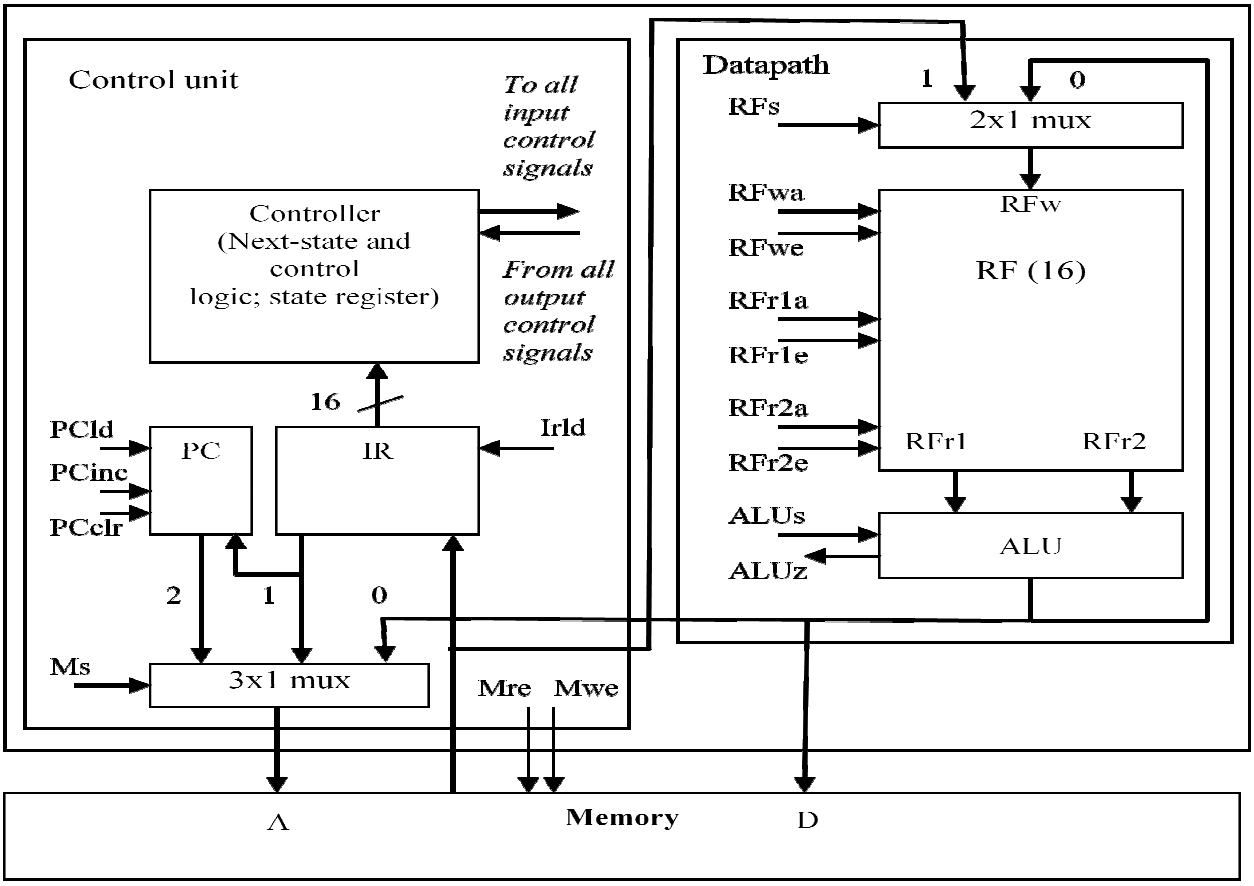


Figure: Architecture of a simple microprocessor

3. Create the controller as follows:

a) We now create FSM from FSMD
b) Each FSMD operation is replaced by binary operation on control signal
c) Finally we design controller using combinational and sequential logic.

PCclr=1; (program counter clear)
MS=10; (Memory select)
Irld=1;(Instruction register load)
Mre=1;(Memory read enable)
PCinc=1;(Program counter enable)

Difference between Single-Purpose Processor and General-Purpose Processor

→ In SPP program is inside the control logic whereas in GPP program is kept in program memory.
→ While designing GPP datapath, we don't have knowledge about the type of program that we are going to run; however for SPP program is known beforehand.

Memory

3.0 Introduction:

Any embedded system functionality consists of three aspects

- → Processing
- → Storage and
- → Communication

i. **Processing:**
Processing is the transformation of data

ii. **Storage:**
It is the retention of data for later use

iii. **Communication:**
Communication is the transfer of data

We use processor to implement processing, memory to implement storage and buses to implement communication.

3.1 Memory

A memory stores large number of bits. These bits can be viewed as 'm' words of n bit each, for a total of m×n bits as shown below.

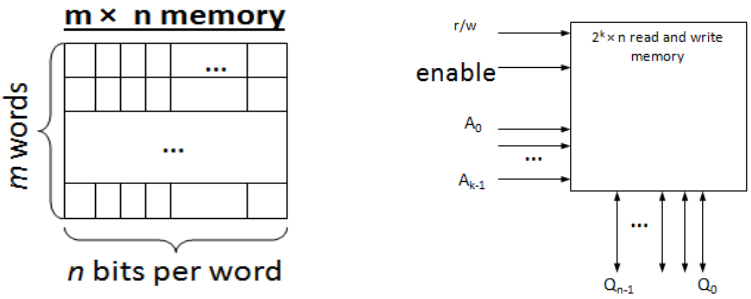


Figure: memory block diagram

If a memory has K address inputs, it can have up to 2^k words n data lines are required to output(and possibly input) a selected words.

For e.g. 4096×8 memory can store 32768 bits of information. We can calculate the address line required to interface this memory chip as:

2^k =memory capacity =4098×8 bytes

$2^k=2^2 2^{10}=2^{12}$

K=12

Thus we need 12 address signal and eight input/output lines are required

- → Memory access refers to either read or write. For memory access we need additional control signal r/w, to select whether we want read access or write access.
- → Memory devices also have chip enable (CS) control signal to select or de-select the memory chip for read/write access.

Advancement in technology has made it possible to use both RAM/ROM to store information and flash them as well. Thus in embedded system the memories are classified on the basis of

- Write ability and
- Storage performance

3.2 Memory writes ability and storage performance

3.2.1 Write Ability

Write ability refers to the manner and speed that a particular memory can be written. All types of memory can be read or written by the processor, however the manner and speed of such writing varies greatly among memory types.

Basically we have three levels of write ability:

i. **High end**
   Processor can write simply and quickly by setting such memory's address line, data input bits and control lines appropriately

ii. **Lower end**
   Memory can be written to by special equipment called programmer. This device must apply special voltage levels to write to the memory, also known as burning/programming the memory.

iii. **Low ends**
   Memory that can only have their bits stored during its fabrication

   The term "in-**system programmable**" is used to divide memories into two categories along the write ability axis.
   They are:
   - → In system programmable
   - → Non in system programmable

a. **In system programmable:**

   The ability of some programmable logic devices, microcontroller and other embedded devices to be programmed while installed in a computer system rather than requiring the chip to be programmed prior to installing it into the system

b. **Non in-system programmable:**

   The chip is removed from the target board and placed in a programmable device to be programmed. Memories in the lower range and low end fall under this category.

4.2.2 Storage Performance

It is the ability of memory to hold its stored bits after those bits have been written. On the basis of storage permanence memory devices can be categorized as:

i. High end:
   They will essentially never lose its bits as long as the memory chip is not damage.

ii. Middle range:
   It holds bits for days, months or even years after power turned off. Eg Non-volatile RAM.

iii. Lower Range:
   It holds bits as long as power is supplied to the memory. Eg SRAM.

iv. Low end:
   It begins to lose its bits immediately after these bits written and must be continually refreshed. Eg DRAM.
   The term "non-volatile" and "volatile" are commonly used to divide memory types into two categories
   - → Non-volatile memory
   - → Volatile memory

1. **Non-volatile Memory**
   Non volatile memories can hold its bits even after power is no longer supplied. High end and middle range of storage permanence memory chip falls under this category.

2. **Volatile**
   It requires continual power to retain its data. Lower-range and low-end of storage permanence fall under this category.

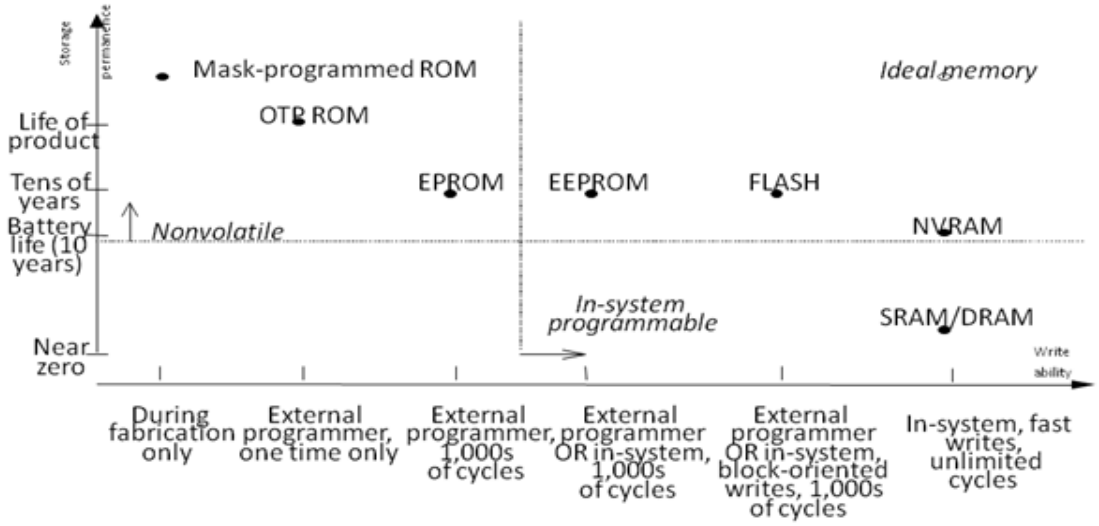


Figure: write ability and storage permanence of memories, showing relative degrees along with axis

3.2.3 Trade-Offs:

Some of the trade-offs are as follows:

- Design metrics compete with each other.(Memory write ability and storage permanence are two such metrices)
- Memory write ability and storage permance tents to be inversely proportional to each other.
- Highly writeable memory typically requires more area and/or power than less-writable memory.

Ideally we want a memory with the highest write ability and the highest storage permanence

3.3 Common Memory Types

3.3.0 Introduction to Read-only memory (ROM)

ROM or read-only memory is a non-volatile memory that can be read from, but not written into by a processor in an embedded system.
Thus there is a mechanism for setting the bits in the memory by programming such memory off-line, when the memory is offline. Such memory is programmed prior before inserting it into the embedded system.

Some of the uses of ROM are:

- → To store a software program for a general-purpose processor.
- → To store constant data needed by a system
- → ROMs can be used to implement combinational circuits.

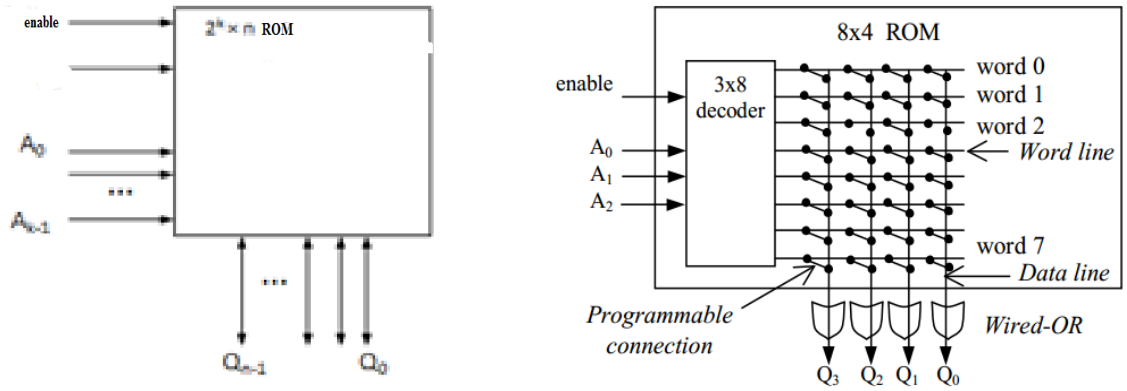


Figure: External Block diagram and Internal Block diagram

Figure above shows a general block-diagram of ROM

- → A_0-A_{K-1} represents the address bus and it selects the storage location to be accessed.
- → Q_0-Q_{n-1} represents data bus. Information is accessed by micro-controller via this data bus.
- → Enable control pin is used to select the particular ROM.

Internal view of ROM:

Figure above provides symbolic view of internal design of 8×4 ROM

- → Word lines runs horizontally
- → Data lines run vertically
- → Word lines only connect to data lines via the programmable connection lines
- → If address input (A0,A1,A2) is equal to 010 the decoder will select the second word line the data line Q3 and Q1 is set thus the ROM contents will be 1010.

3.3.1 Types of ROM

a. Mask-Programmed ROM

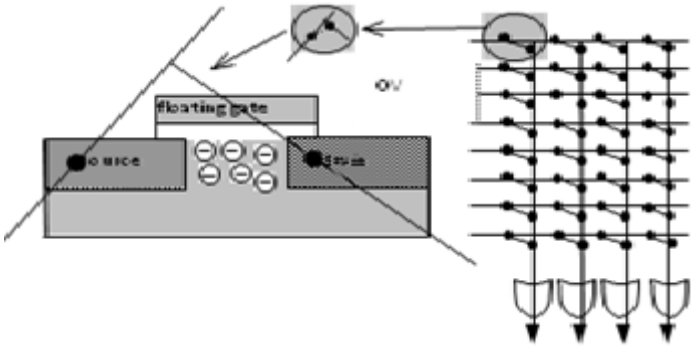
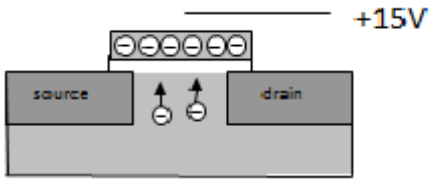
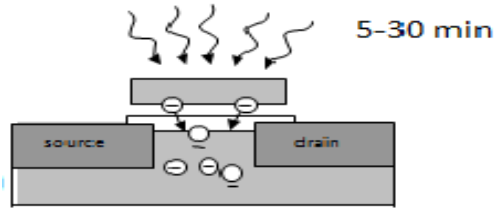
- → In mask-programmed ROM, the connection is programmed when the chip is being fabricated by creating a appropriate set of masks.
- → They have low write ability and highest storage permanence.
- → Such ROMs are typically used only after a final design has been determined and when used in high volume NRE cost can be amortized.

b. OTP-One Time Programmable ROM

- → User programmable ROMs are generally referred to as programmable ROMS(PROM).
- → They are better situated to prototyping and to low-volume applications than earlier.
- → PROM uses fuse for each programmable connections
- → ROM programmer is a piece of hardware that configures such connections as desired. Fuses are blowed by passing a large current whenever a connection should not exit. Once blowed, the connection cannot be reestablished. Thus PROM is also referred as OTP ROM.
- → They have low write ability high storage permanence and they are commonly used in final product because of its lower cost per unit .

c. EPROM-Erasable Programmable ROM

- → EPROM uses MOS transistor as its programmable component. The transistor has a "floating gate". That is gate is not connected and instead surrounded by an insulator.
- → Compared to OTP ROM, EPROM have improved write ability and they can be reprogrammed thousand of time however they have low storage permanence.

a.	Initially, the negative charges form a channel between the source and drain of the transistor storing a logic 1 at that cell location.	
b.	By applying a large positive voltage at the gate of the transistor the negative charges move out the channel area and get trapped in the floating gate, storing a logic 0 at that cell's location.	
c.	By shining UV ray on the surface of the floating-gate the negative charges moves down into the channel restoring the logic 1 at the cell's location.	
d.	An EEPROM package with a quartz window through which UV light can pass.	

d. EEPROM-Electrically Erasable Programmable ROM

- → EEPROM use the same floating gate principle as the PROM
- → EEPROM contains program operation in the program operation the floating gate is negatively charged with electrons tunneling from the drain through the thick oxide, which is achieved by applying +ve voltage to gate of transistor while the source, drain and substrate are grounded.
- → They can be erased and programmed thousand of time.

e. Flash

- → It is an extension of EEPROM
- → It also uses same floating gate principle and has the same write ability and storage permanence as EEPROM
- → Large blocks of memory can be erased at once.
- → Drawback of flash memory is that writing to a single word may be slower because an entire block need to be read the word within is updated and then the block written back.

3.4.0 Introduction to Read-Write Memory (RAM)

RAM is a volatile memory which can be read and written. Writing to RAM is as fast as reading the content from it. Thus they are not programmed ahead rather they are programmed within the embedded system.

3.4.1 Internal Structure of RAM

Figure below shows an internal structure of 4×4 RAM (Typically RAM have thousands of word, not just four as shown).

- → Each word consists of number of memory cell, each storing one bit of information.
- → Each input data lines connects to every cell in its column
- → Each output data line connects to every cell in its column with the output of a memory cell being ORed with the output data line from above.
- → Each word enable line from the decoder connects to every cell in its rows.
- → The read/write input(rd/wr) is connected to every cell.
- → When row is enabled and rd/wr assert write each cell must store the contents similarly if the row is enable and rd/wr assert read signal, it must output the contents to its data bus(output)

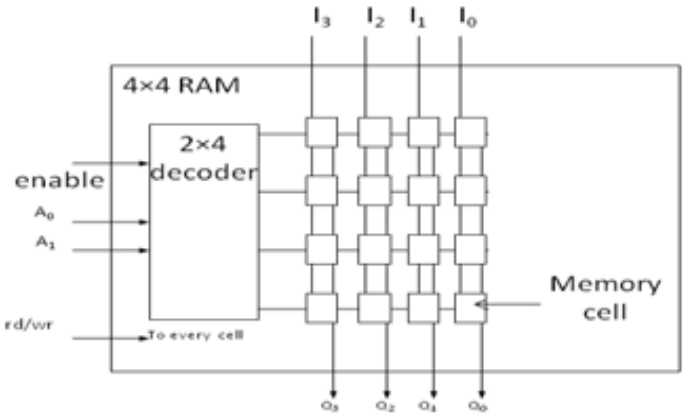


Figure: Internal Structure of RAM

3.4.2 Types of RAM

There are two types of RAM

- → SRAM(static RAM)
- → DRAM(Dynamic RAM)

1. SRAM

This memory cell uses a flip-flop to store bits. It requires six transistors, 4 from a bistable inverter and remaining 2 as access transistor.

It is called static because it will hold data as long as power is supplied and needs no refreshing while being read.

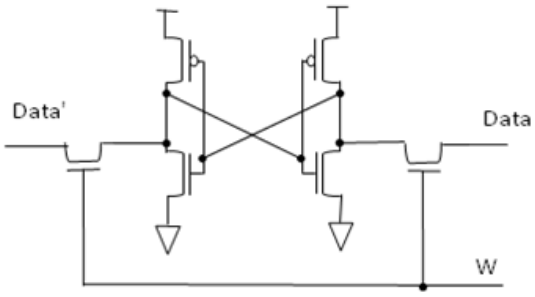


Figure: Memory Cell internal

Each SRAM cell has three different states

- → Hold
- → Read and
- → Write

2. DRAM

- → Each memory cell consists of MOS transistor and a capacitor to store a bit.
- → Each bit requires only one transistor resulting in a more compact memory than SRAM.
- → The charge stored leaks gradually leading to discharge and eventually loss of data.
- → To prevent each cell from loss of data, it must be regularly refreshed or charged.

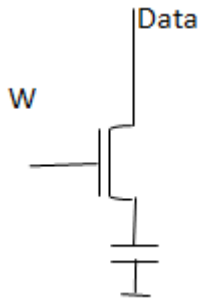


Figure: Memory cell internal of DRAM

3. PSRAM-Pseudo Static RAM

- → These are DRAMs with a memory refresh controller builtin. A PSRAM refreshes itself when access. This could slow down access time and add system complexity.

4. NVRAM-Nonvolatile RAM

- → **It is a special RAM that is able to hold its data even after external power supply is removed.**
- → **There are two types of NVRAM**
  - i. Battery-backed RAM
    - ➢ It contains static RAM along with its own permanently connected battery. When external power is removed, internal battery maintains power in SRAM. And thus, memory continues to store its bits.
  - ii. Second type of NVRAM contains a static RAM as well as EEPROM or flash having same capacity as static RAM. This type of NVRAM store its complete RAM contents in to EEPROM/flash just before power is turned off, and then reloads that data from EEPROM/flash back into RAM after power supply is turned on.

3.5 Composing Memory

An embedded system designer is faced with situation of needing a particular sized memory(RAM or ROM) but available memories are of different size. For example, the designer needs a 2k×8 ROM but may have 4k×16 ROMS. Similarly the designer may need 4k×16 ROM, but may have 2k×8 ROMS. Thus we have two cases;

- → Available is greater than needed and
- → Need is greater than available

a. In case where available memory is larger than needed requires less design effort. Here we simply use the needed lower words in the memory thus ignoring unneeded higher words and their higher-order address bit.

We only use the lower data input/output lines, thus ignoring unneeded higher data lines. Also we could use higher word and address ignoring lower words and address.

b. The case where available memory is smaller than needed requires composing several smaller memories to behave as a large memory. More design effort is needed in this case. Some of the cases are:
   - → Correct number of words, but not wide enough
   - → Correct word width but not enough words
   - → Smaller words width as well as fewer words then necessary

1. Increase width of words

The available memory has correct number of words but each word is not wide enough.

- → In such case we connect memories side by side
  Eg we need ROM that is 3-times wider than what is available; here we connect 3-ROMs side by side sharing the same address and enable lines and concatening data lines to form desired word width

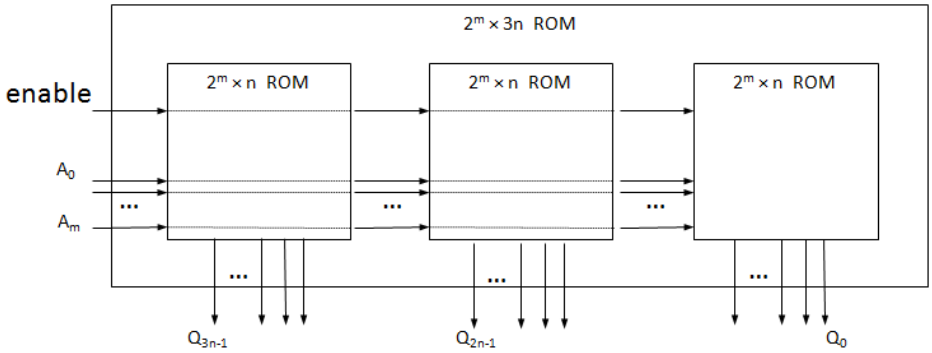


Figure: Increase width of words

2. Increase number of words

Available memories have correct words width but not enough words

- → In such case we connect available memories top to bottom.
  Eg we need a ROM with twice as many words than what is available; here we connect the ROMs top to bottom, ORing the corresponding data lines of each.
- → Here, we connect the ROMs top to bottom, ORing the corresponding data lines of each
- → To select between two ROMs, we need extra address line. 1×2 decoder can select between two ROMs.

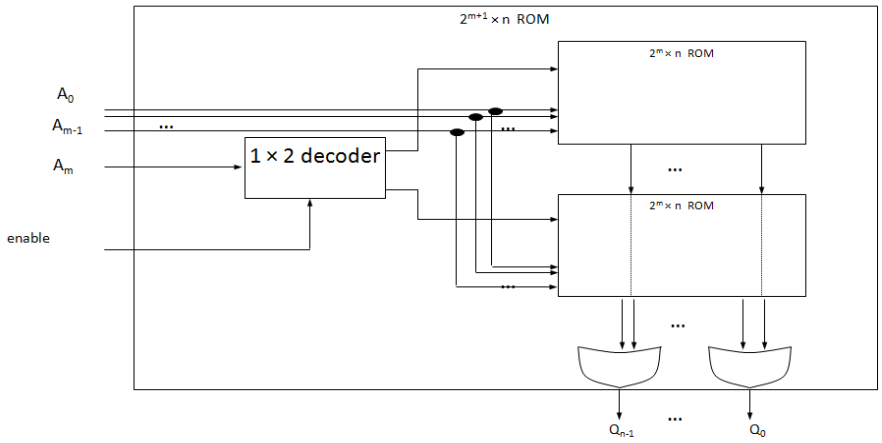


Figure: Increase number of words

3. Increase number of words and width of words

The available memory has smaller word width and fewer word than needed. So we combine technique 1 and 2.

→ First we create number of column of memories necessary to achieve needed word width secondly we create number of rows of memories necessary along a decoder to achieve needed number of words.

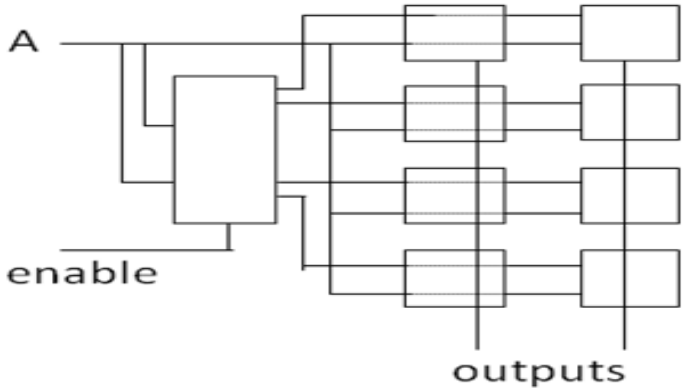


Figure: Increase number of words and width of words

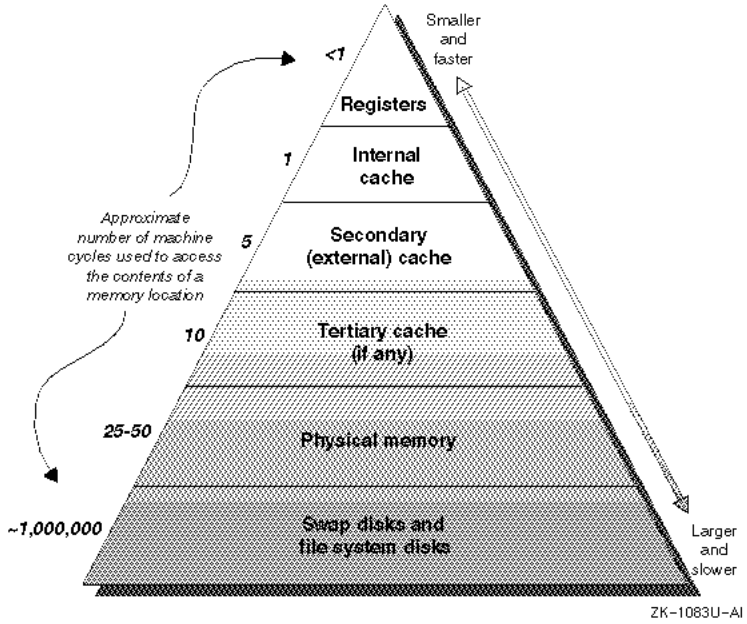
3.6 Memory Hierarchy and Cache

- Capacity, cost and speed of different types of memory play a vital role while designing a memory system for computers.
- If the memory has large capacity, more applications will get space to run smoothly.
- It's better to have fastest memory as far as possible to achieve greater performance. Moreover for the practical system, the cost should be reasonable.
- There is tradeoff between these three characteristics cost, capacity and access time. One cannot achieve all these quantities in same memory module because
  - → if capacity increases, access time increases(slower) and due to which cost per bit decreases.
  - → If access time decreases (faster) capacity decreases and due to which cost per bit increases.
- The designer tries to increase capacity because cost per bit decreases and more application program can be accommodated. But at the same time access time increases and hence increases the performance.

Thus the best idea will be to use Memory hierarchy.

- Memory hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system.
- Not all accumulated information is needed by the CPU at the same time, thus it is more economical to use low-cost storage to serve as backup for storing the information that is not currently used by CPU.
- The memory unit that directly communicates with CPU is called the main memory.
- Devices that provide backup storage are called auxiliary memory.
- The memory hierarchy system consists of all storage devices employed in a computer system from the slow by high-capacity auxiliary memory to a relatively fast main memory, to a even smaller and fast cache memory.

- The main memory occupies a central position by being able to communicate directly with the CPU and auxiliary memory devices through an I/O processor.
- A special very-high-speed memory called cache is used to increase the speed of processing by making current programs and data available to the CPU at the rapid rate.
- CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory.
- The cache is used for storing segments of programs currently being executed in the CPU and temporarily data frequently needed in the present calculation.
- The memory hierarchy system consists of all storage devices employed in a computer system from a slow but high capacity auxiliary memory to a relatively fast cache memory accessible to high speed processing logic.
  The figure below illustrate memory hierarchy.



3.6.1 Cache operation

- → When we want the processor to access (read/write) a main memory address, we first check for a copy of that location in cache.
- → If that copy is in the cache, it is called a cache hit and can be accessed quickly.
- → It that copy isnot available in cache, it is called a cache miss and we most first read the address of main memory into cache.

3.6.2 Cache Mapping Techniques

The cache mapping method address following two questions

- When we copy a block of data from main memory to cache, where exactly should we put it
- How can we tall if a word is already in the cache or if it has to be fetched from main memory first.

There are three main techniques for accomplishing cache mapping

1. Direct mapping

- → In direct mapping as shown in figure below main memory address is divided into two fields, index and tag.
- → The index represents the cache address and thus the number of index bit is determined by the cache size

  Index length $=\log_2$(cache row) bits
- → The index describes which cache row that data has input in
- → Many different main memory address will map to the same cache address.
- → When we store the contents of a main memory address in the cache, we also store the tag.

→ The main memory address is also further divided into offset. The block offset specifies desired data within the stored data block within a cache row.

Offset length=$\log_2$(bytes per data block) bits.

→ The tag contains most significant bits of main memory address which are checked against current cache row to see if its one we need.

Tag length=[main address length-index length-offset length] bits

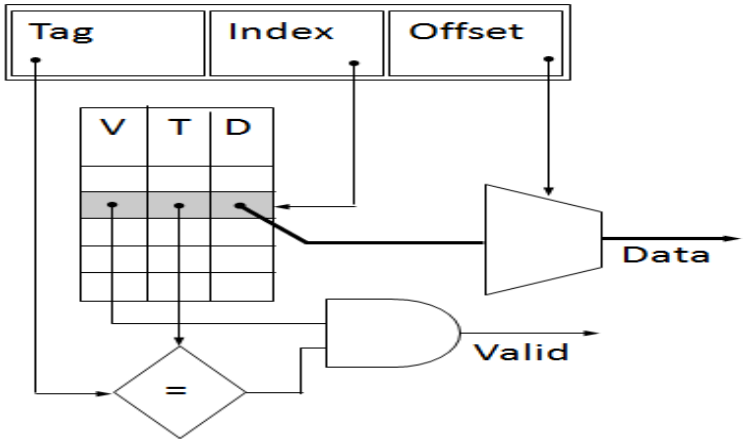


Figure: Direct Mapping

→ To determine if the desired main memory address is in cache. Let go cache address indicated by the index and then compare tag of cache with tag of main memory address. If tags match then we check valid bit using AND gate. If result is valid then data from main memory has been loaded into cache. We use offset portion of memory address to specify desired data with stored data within cache.

2. Fully associative cache mapping

→ In fully associative mapping each cache address not only contains of a main memory address, but also completes main memory address.

→ The data from any location in main memory can be stored in any location within cache

→ To determine if a desired main memory address is in the cache, we simultaneously compare all the addresses stored in the cache with the desired address.

→ For a fully associative cache, there is no index field.

→ The lower[$\log_2$(cache row)] bits define the offset

→ The remaining upper bits are the tag field.

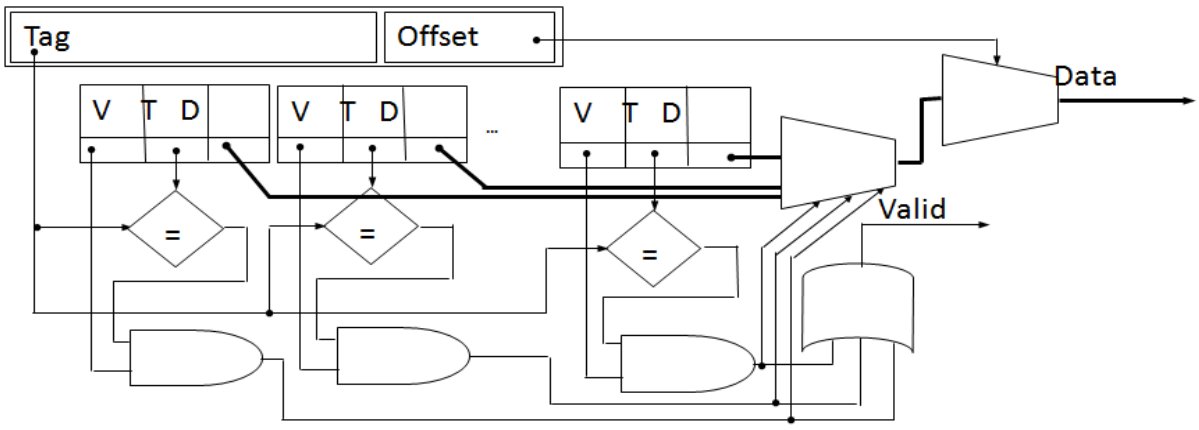


Figure: Cache Mapping fully set associative

3. Set associative mapping

- → In set-associative mapping a compromise is reached between direct and fully associative mapping
- → Each memory address maps to exactly one set in cache but data may be placed in any block within that set.
- → A cache with a set of size "N" is called N-way set associative cache.
- → If a cache has 2s sets and each block has 2n bytes then main memory address can be partitioned as follows.

→ To determine if a desired main memory is in cache, we go to the cache address indicated by index and then simultaneously compare all the tags of that set with desired tag.

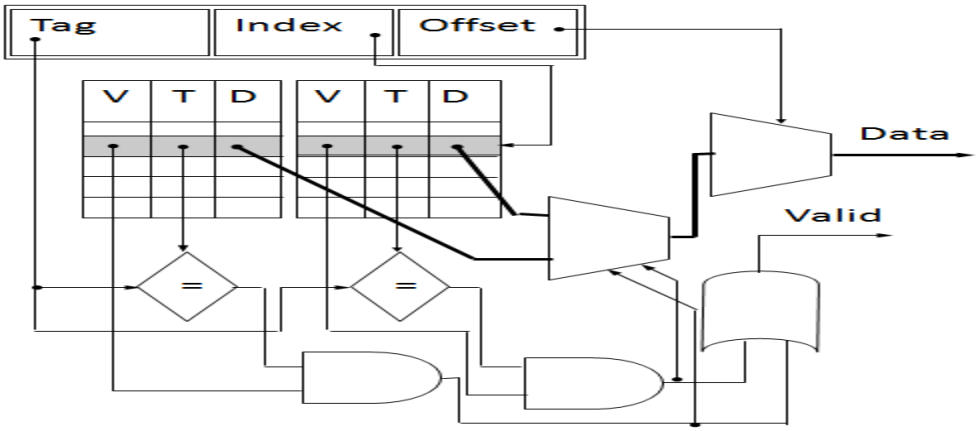


Figure: Two way set associative

Note:

- → Direct mapped caches are easy to implement but result in numerous misses. A main memory address always maps to same cache address and replaces whatever block is already there.
- → In fully associative and set-associative cache there is more than one location in cache where every location from main memory can be mapped.

3.7 cache Replacement Policy

The cache replacement policy is technique for choosing which cache block to replace when a fully associative cache is full or when a set-associative cache if full. There are three primary replacement policies

a. Random Replacement Policy

This technique is very simple because one block is selected at random and replaced. While its simple to implement, this policy doesnot prevent replacing a block i.e likely to be used again soon.

b. Least Recently Used(LRU) Replacement policy

In this approach, access to cache is recorded and the block that will be replaced is the one that has been unused (unaccessed) for longest period of time.

If the blocks tends to be accessed then it seems natural to discard the block the one that has been of little use in past. This replacement policy provides for an excellent hit/miss ration. But it requires expensive hardware to keep track of times blocks of cache are assessed.

c. FIFO Replacement Policy

It uses a queue of size "N" pushing each block address onto queue. When address is accessed and then choosing block to be replaced by popping queue. The replacement policy doesnot take into account the addressing pattern of past; it may happen that a block has been heavily used in previous addressing cycle and yet it is chosen for replacement.

3.8 Cache Write Technique

There are two cache write technique

- → Write through
- → Write back

i. Write through

Whenever data is written to cache, data is also updated in main memory. While updating main memory processor needs to wait until write to memory completes. This technique may results in several unnecessary writes to main memory.

For example, suppose a program writes to a block in the cache, then reads it, and then writes it again,with the block staying in the cache during all three access. There would have been no need to update main memory after first write since second writes over-writes first write.

ii. Write-back

The write back technique reduces the number of writes to main memory by writing a block to main memory only. When the block is replaced, and then only if the block was written to during its stay in the cache. This technique requires an extra bit called a dirty bit. This bit is set whenever we write to the block in cache, thus this bits keep track if block was modified or not.

3.9 Cache Impact on system performance

The design and configuration of cache have large impact on performance and power consumption of a system. from performance point of view, the most important parameter in cache design are

- → Total size of cache
- → Its degree of associatively and
- → Data block size i.e. read or write during each cache access.

The total size of the cache is measured as the total number of data bytes that the cache can hold. The cache also stores other information, such as tags and valid bit, which donot contribute to the cache. By making a cache larger, one achieves lower miss rates, which is one of the design goals. However accessing words within a large cache will be slower than accessing words in a smaller cache.

Design Example:

Lets design 2k byte of cache for a processor.
We have miss rate from measurement
=15%(i.e. 15 out of 100 cache access result in a miss)
Similarly miss cost(i.e. cost of memory access when there is miss)=20 cycles.
Also, we have hit cost(i.e. cost of memory access)
=2 cycles
Therefor, average cost of memory access
$=(0.85\times2)+(0.15\times20)=4.7$ cycles
Now doubling the cache size, and assuming this improves hit ratio to 93.5% at the expense of slowing the cache by extra clock.
Now , average cost of memory access

=(0.935×3)+(0.065×20)=4.105 cycles
Again doubling the cache, resulting in one additional clock per hit, achieving 94.435% of hit
The average cost of memory access=(0.94435×4)+(0.05565×20)=4.8904 cycles
This larger cache will perform even worst than first two design

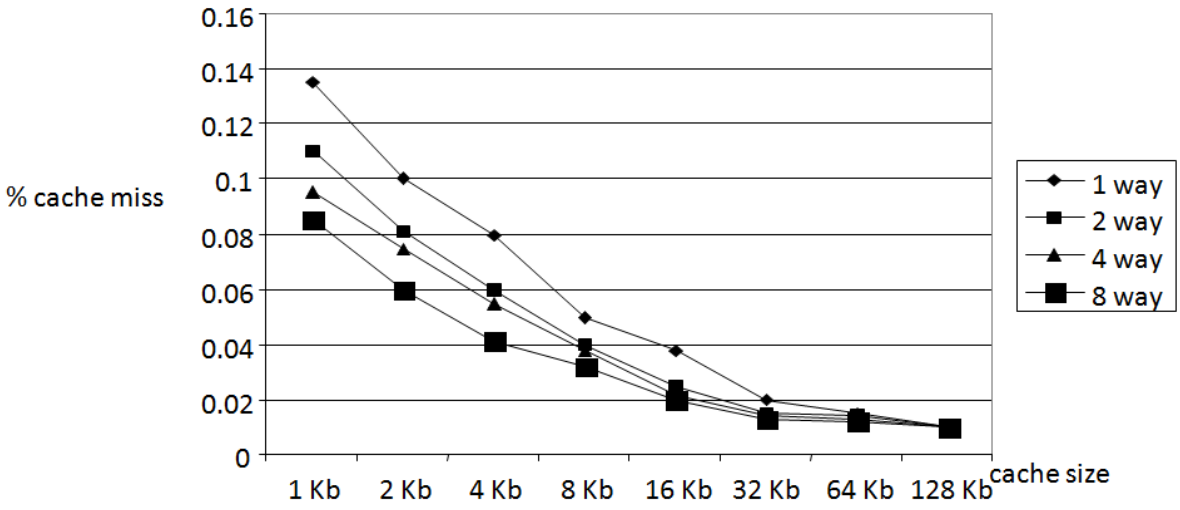


Figure: Sample Cache performance trade-off

- → **The problem of making cache larger is additional access time penalty, which quickly offsets the benefits of improved hit rates.**
- → **Designer use other method to improve cache hit rate without increasing cache size. They change either set associativity of cache or increase data block size of the cache.**

Interfacing

4.0 Communication Basic

4.1.0 Basic Terminology

Figure below shows the bus structure or the wires connecting the processor and the memory.

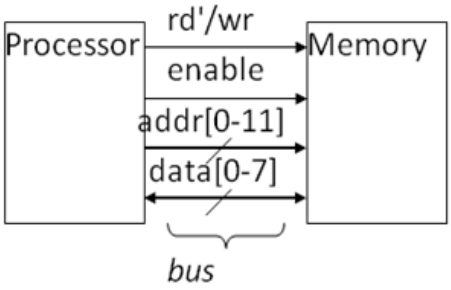


Figure: bus structure

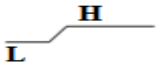
- → A rd/wr line indicates whether the processor is reading or writing
- → An enable line is used by the processor to carry out the read or write operation
- → The address line addr(12 lines) indicates the memory address that the processor wishes to read or write
- → The data line "data" is set by processor when writing or set by memory when the processor is reading.

Such connecting wires may be unidirectional or bi-directional. The term bus refers to set of wires with a single function within a communication. For eg address bus and data bus as shown above.

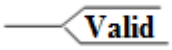
4.1.1 Timing Diagram

Timing diagram describes the actual process of operation of devices. They are most common method for describing communication protocol.

- → Control signal is denoted by



- → Data signal is denoted by



Protocol consists of set of rules for communication. Protocols may have subprotocols eg bus cycle such as read and write.

1. **Read Protocol:**
   The processor sets rd/wr to 0 (low) and places a valid address on "addr" for at least ($_{tsetup}$) time before "enable" is activated.
   - → Enable triggers memory to place data on "data-line" by time(t_{read})

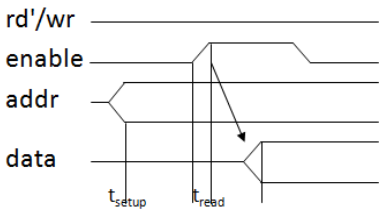


Figure: Read Protocol

2. Write Protocol

→ The processor sets rd/wr to 1(high) and places a valid address on "addr" place data on data lines and activate enable signal by time(t_{write}).

→ A port is actual conducting device on the periphery of a processor, through which a signal is input to or output from the processor, port can also refer to pins on the periphery of an IC package, or pads of metal underneath an IC.

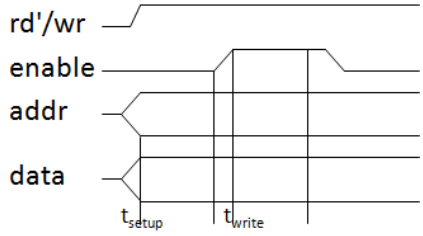


Figure: write Protocol

4.1.2 Basic Protocol Concept

i. An actor

→ An actor is a processor or memory involved in the data transfer.

→ A protocol typically involves two actors a master and a servant.

→ A master initiates the data transfer and slave or servant responds to the initiation requirement.

→ Master is usually general-purpose processor and slaves are usually peripheral and memories.

ii. Data direction

→ Data direction denotes the direction that the transferred data moves between the actuators.

iii. Address

→ Address represents a special type of data used to indicate where regular data should go to or come from.

→ An address is necessary when a general purpose processor communicates with multiple peripherals over a single bus; the address not only specifies a particular peripheral but also specifies particular register within that peripheral.

iv. Time multiplexing

→ Multiplex means to share a single set of wires for multiple piece of data.

→ In the multiplexing, multiple pieces of data are sent one at a time over the shared wires.

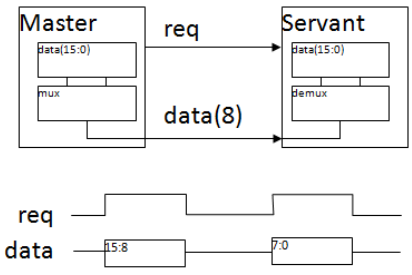


Figure: Data Serializing

→ The above figure shows a "Master" sending 16-bit of data over an 8-bit bus. The master first sends the high-order byte and then the low-order byte. The servant must receive the bytes and then de-multiplex the data.

→ A master sends both an address and data to a servant.

→ Rather than using separate sets of lines for address and data, we can multiplex the address and data over a shared of lines.

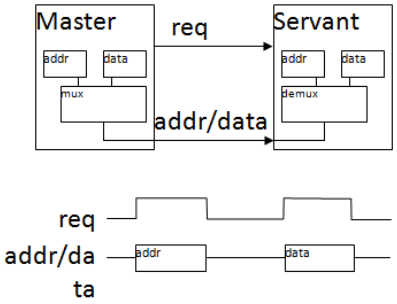


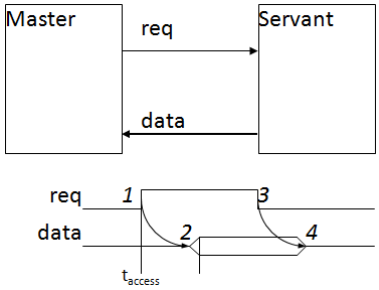
Figure: address/data multiplexing

v. Control Method

- → Control methods are schemes for initiating and ending the transfer
- → Two of the most common method is "strobe" and "handshake".

i. Strobe

- → In strobe protocol, the master uses one control line, called the request line to initate the data transfer and the transfer is considered to be complete after some fixed time interval after initiation.



1. Master asserts req to receiver data.
2. Servant puts data on bus, within time t_{access}
3. Master receives data and deasserts req
4. Servant ready for next request

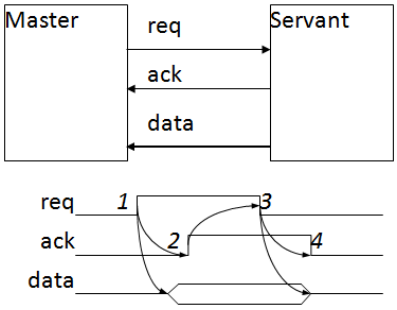
Figure: Strobe

- → The master activates the request line to initiate a transfer denoted by point 1
- → The slave then puts data on the data bus, within the time(t_{access}) denoted by point 2
- → The master then reads the data bus. The master then deactivates the request line so that the slave can stop putting the data on the data bus, denoted by point 3.
- → Both actors are now ready for the next transfer denoted by point 4.

An analogy is a demanding boss who tells an employee" I want that report (the data) on my desk (the data bus) in an hour (t_{access}) and merely expect the report to be on the desk in an hour.

ii. Handshake

- → In this protocol the master uses a request line to initiate the transfer and the servant uses an acknowledge line to inform the master when data is ready.



1. Master asserts req to receiver data.
2. Servant puts data on bus and asserts ack
3. Master receives data and asserts req
4. Servant ready for next request

Figure: handshake

- → The master first asserts the request line to initiate the transfer, denoted by point 1
- → The slave takes as much time as necessary to put the data on the data bus and asserts the acknowledge line to inform the master that the data is valid, denoted by point 2.
- → The master reads the data bus, and then deactivates the request line, denoted by point 3, so that slave can stop putting data on the data bus.
- → The slave then deactivates the "acknowledge" line at point 4 and both actors are ready for the next transfer.

An analogy is a boss telling an employee" I want the report on my desk soon, let me know when it is ready"

iii. Strobe/Handshake compromise:

To achieve both the speed of strobe protocol and the varying response time tolerance of a hand shake protocol, a compromise protocol is used as shown below.

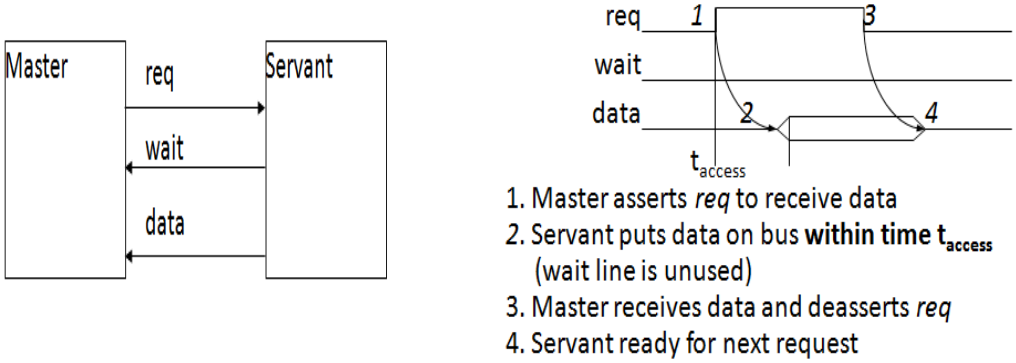


Figure: A strobe/handshake compromise fast-response

- → When the slave puts the data on the bus within the time(t_{access}), the protocol is identical to a strobe protocol
- → At point 1, the master activates the request line to receive data.
- → At point 2, the slave puts data on the bus" within time (t_{access})". There is no use of wait line.
- → At point 3, the master receives the data and deactivates the request line.
- → At point 4, the servant is ready for next request.

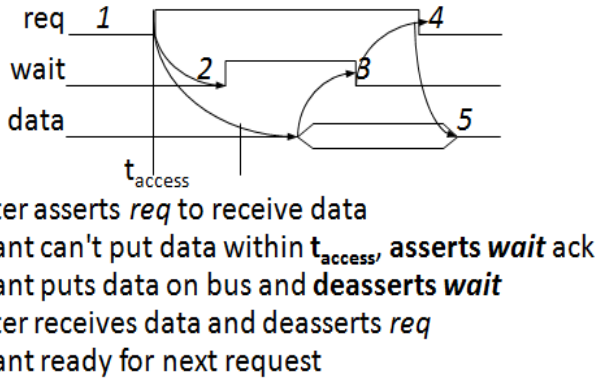


Figure: strobe/handshake compromise fast-response

- → However, if the servant can't put the data on the bus in time, it instead tells the master to wait longer.
- → At point 1 the master activates the request line to receive data.
- → At point 2, the slave cannot put the data on the data line within the (t_{access}) and so activates the wait line.
- → At point 3, the slave puts data on the data bus and deactivates wait.
- → At point 4, the master receives the data and deactivates the request.

→ At point 5, the slave is ready for the next request.

An analogy is the boss telling an employee " I want that report on my desk in an hour, if you cant finish by then, let me know that and let me know when it is ready".

4.2 Microprocessing Interfacing (I/0 addressing)

A microprocessor has many pins for various purposes. Many of the available pins are used to communicate data to and from the processor, which is called processor I/O.
There are two common methods for using pins to support I/O.

→ Port-based I/O
→ Bus-based I/O

i. Port-based(Parallel) I/O

→ A port can be directly read and written by processor instructions, just like any other registers in the microprocessor. A port is connected to a dedicated registers.
→ Processor has one or more N-bit ports.
→ In c-language, writing to a port may look like P0=255, which would set eight pins of P0 to one.
→ Reading a value from a port to a variable may look like a=P1.
→ Ports are often bit-addressable meaning that a programmer can read or write specific bits of the ports. E.g. x=P0 which gives value of pin2 of port 0 to x.

ii. Bus Based I/O

→ In bus-based I/O, the microprocessor has set of address, data and control ports corresponding to bus lines, and uses the bus to access memory as well as peripherals.
→ The microprocessor has the bus protocol built into its hardware. The software doesn't implement the protocol but merely executes a single instruction that in turn causes the hardware to write or read data over the bus.
→ The I/O peripherals are connected via address bus and are accessed by the processor as if it is any other memory address.
The data is sent to or fetched from the peripheral over the data bus.

Parallel I/O peripheral with bus-based I/O processor:

→ A system may require parallel I/O (port based I/O) but a microprocessor may only support bus based I/O.
→ In such case a parallel I/O peripheral may be used as shown below.

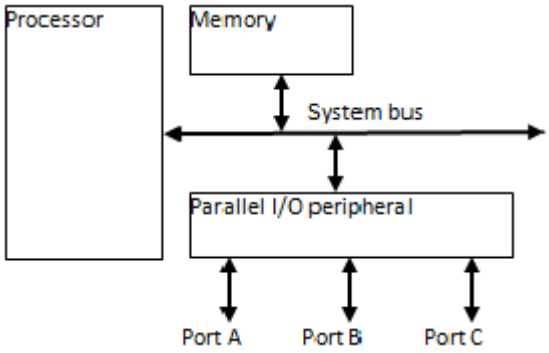


Figure: Adding parallel I/O to a bus-based I/O processor

→ The peripheral is connected to the system bus on one side, with corresponding address, data and control lines and has several ports on the other side.
→ The ports are connected to register inside the peripheral, and the microprocessor can read and write those registers in order to read and write the ports.

Extended Parallel I/O

→ Even when a microprocessor supports port-based I/O, we may require more ports than available. In such case, a parallel I/O peripheral can again be used as shown in figure below.

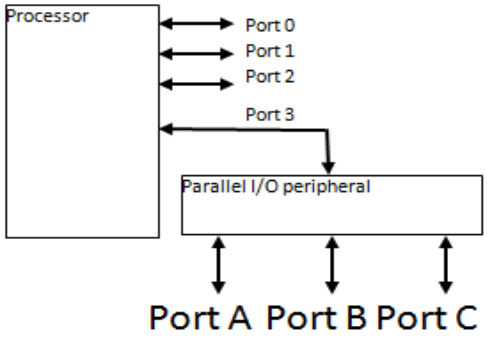


Figure: Extended Parallel input/output

→ The microprocessor has four ports in this example, one of which is used to interface with a parallel I/O peripheral, which itself has three ports. Thus we have extended the number of available ports from four to six.

4.3 Types of bus-based I/O

→ In bus based I/O there are two methods for a microprocessor to communicate with peripherals, known as

- Memory Mapped I/O
- Standard I/O

a. Memory Mapped I/O

- In memory mapped I/O, peripherals occupy specific address in the same address space as memory.
  For example, for a bus with 16-bit address
- Lower 32K address may corresponds to memory
- Upper 32K address may correspond to peripherals.

b. Standard IO(I/O-mapped I/O)

- In standard I/O (also known as I/O–mapped I/O) the bus includes an additional pin which is label M/IO, to indicate whether the access is to memory or to a peripheral (i.e. I/O device).
- E.g. when M/IO is 1, the address on the address bus corresponds to a memory address.
- When M/IO is 0, the address corresponds to peripherals.
-

4.3.1 Memory Mapped I/O vs. Standard IO

	Memory Mapped I/O	Standard I/O
1.	16-bit addresses are provided for IO devices.	8-bit addresses are provided for IO devices.
2.	The devices are accessed by memory read or memory write cycles.	The devices are accessed by IO read or IO write cycle. During these cycles, the 8-bit address is available on both low order address lines and high order address lines
3.	The IO ports or peripherals can be treated like memory locations and so all instructions related to memory can be used for data transfer between IO device and the processor.	Only IN and OUT instructions can be used for data transfer between the IO device and the processor.

4.	In memory-mapped ports, the data can be moved form any register to the ports and vice versa.	In IO mapped ports, the data transfer can take place only between the accumulator and the ports.
5.	When memory mapping is used for IO devices, the full memory address space cannot be used for addressing memory. Hence memory mapping is useful only for small systems, where the memory requirement is less.	When IO mapping is used for IO devices, then the full memory address space can be used for addressing the memory. Hence it is suitable for system which requires a large memory capacity.
6.	In memory mapped IO devices, a large number of IO ports can be interfaced	In IO mapping only 256(2^8) can be interfaced

4.4 Microprocessor Interfacing: Interrupts

- → A peripheral posse's data that need to be sent to the processor for servicing. How can program determine when the peripheral has new data?
- → The processor can "poll" the peripherals regularly to see if data has arrived. However, this repeated checking waste many clock cycle.
- → The other possibility is for the peripheral to interrupt the processor when it has data. This feature is called "external interrupt".
- → The external interrupt solution requires the processor to have an extra pin. E.g. INT.
- → When INT is activated, the processor jumps to a particular address at which a subroutine exits that service the interrupt.
  This subroutine is called interrupt service routine (ISR) and such I/O is called interrupt driven I/O.

There are two methods by which a microprocessor using interrupts determines the address, known as the interrupt address vector at which ISR resides.

a. Fixed interrupt:

- → The address to which the processor jumps on an interrupt is built into the processor, so it is fixed and can't be changed
- → The assembly programmer either puts the ISR at that address or if not enough bytes are available in that region of memory the programmer puts a jump to the actual ISR at that address.
- → In microprocessor with fixed ISR addresses, there may be several interrupt pins to support interrupt from multiple peripherals.

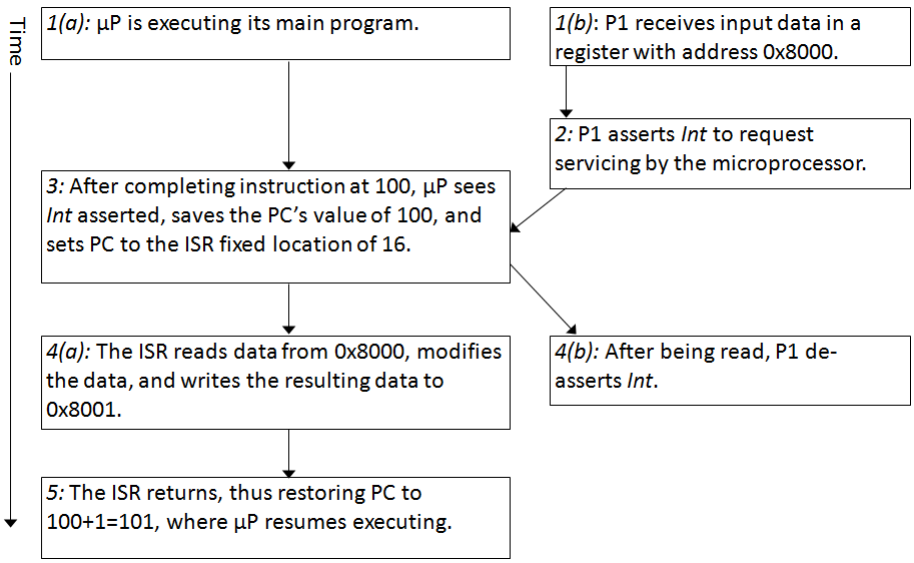


Figure: interrupt driven I/O using fixed ISR location.

- In this example, data received by peripheral 1 must be read, transformed and then written to peripheral 2.
  Peripheral 1 might represent a sensor and peripheral 2 is a display.
- The processor is running its main program located in program memory starting at address 100.
- When peripheral 1 receives data, it asserts INT to request the microprocessor to service the data.
- After microprocessor completes execution of its current instruction, it stores its states and jumps to the ISR location at the fixed program memory location of 16.
- The ISR reads the data from peripheral 1 transforms it and writes the result to peripheral 2.
- The last ISR instruction is a return from interrupt, causing the microprocessor to restore its state and resume execution of its main program.

b. Vector Interrupt:

- Microprocessor use vector interrupt to determine the address at which the ISR resides.
- This approach is common in system with a system bus; there can be numerous peripherals that can request service.
- The processor has an interrupt pin (INTR) which a peripheral can activate.
- After detecting the interrupt, the processor sends an "interrupt acknowledge" (INTRA) to the peripheral to request the address of the relevant ISR.
- The peripheral provides the address on the data bus, and the processor reads the address and jumps to the corresponding ISR.

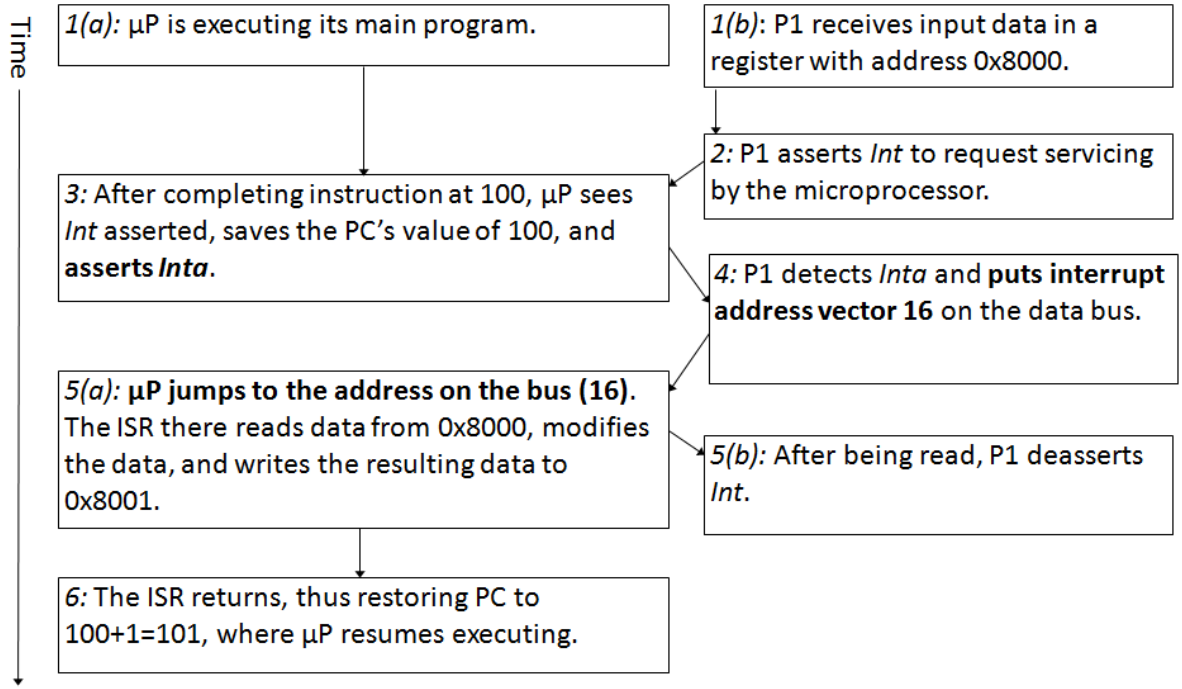


Figure: interrupt-driven I/O using vectored interrupt

4.4.1 Compromise between Fixed and Vector Interrupt

- As a compromise between the fixed and vector interrupt method, interrupt address table is an alternative
- It still needs one interrupt pin on the processor
- Creates an interrupt address table in the processor's memory holding the ISR addresses.
- The peripheral doesn't provide ISR address but rather an index into the table.
- Fewer bits are required for the index and this is important when the data bus is not wide enough to hold complete ISR address
- Allows movement of ISR location, without having to change anything in the peripheral.

4.4.2 Maskable and Non-Maskable Interrupt

- → In maskable interrupt, the programmer can set a bit that causes the processor to ignore the interrupt
- → This is important, when the processor is executing a time-critical code, such as routine that generates pulse of certain duration
- → A Non-maskable interrupt can't be masked by the programmer
- → Non-maskable interrupts are typically reserved for critical situation, such as power failure requiring immediate back up of data or to re-start the system.

4.5 Microprocessor Interfacing: Direct Memory Access

- → DMA is the ability of an I/O subsystem to transfer data to and from memory without processor intervention.
- → DMA eliminates the need of an IAR whenever a peripheral request data to be stored to or retrieved from memory. Jumping to an ISR requires processor to store its state, and then to restore its state when returning from the ISR.
  This storing and restoring of the processor state consumes many clock cycles and is inefficient.
- → DMA requires a DMA controller whose sole purpose is to transfer data between memories and peripherals
- → The peripheral request servicing from the DMA controller, which then request control of the system bus from the microprocessor.
- → The processor only need to release control of the system bus from the processor the processor does not need to jump to an ISR, and so the overhead of storing and restoring the processor state is eliminated.
- → Furthermore, the microprocessor can execute its regular program while the DMA controller has bus control, as long as that regular program doesnot use of the bus(at which point the microprocessor would then have to wait for the DMA to complete)

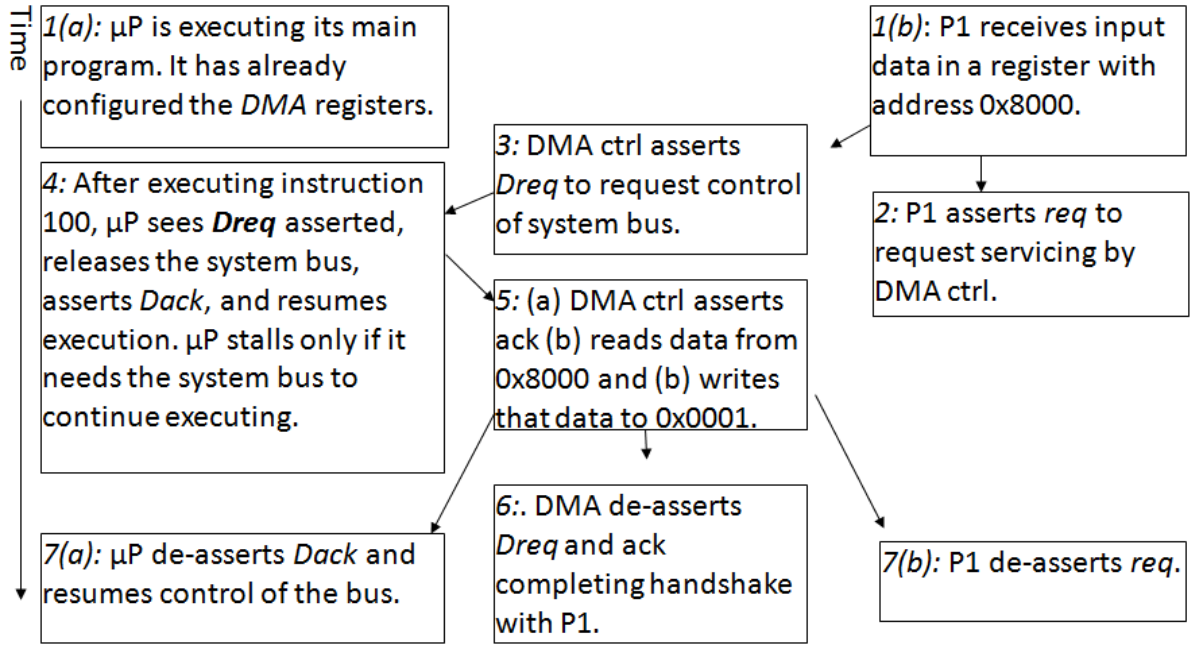


Figure: Peripheral to memory transfer with DMA

4.6 Arbitration

Multiple peripherals might request service from a single resource as well as multiple peripherals might share single DMA controller that services their DMA request.
In such situations, two or more peripherals may request service simultaneously. Thus there need to be some-way to arbitrate (decide) the contending request.

1. Priority Arbiter:

Multiple peripherals request servicing from the processor. Each of the peripheral makes its request to the arbiter. The arbiter in turns asserts the microprocessor interrupt, and wait for the interrupt acknowledgement. The arbiter then provides an acknowledgement to exactly one peripheral to put its interrupt vector address on the data line.

Priority arbiter uses two common schemes to determine priority among peripherals

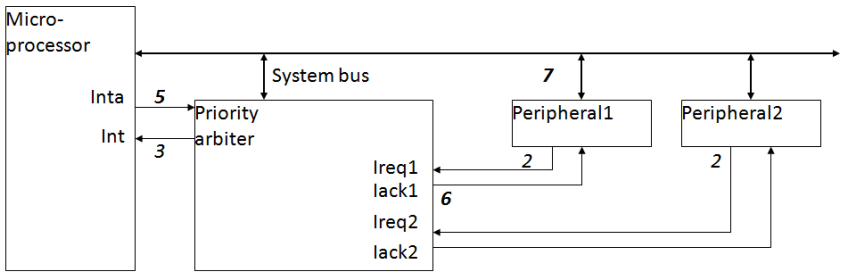
- → Fixed priority
- → Rotating Priority

a. Fixed Priority

- → In fixed priority arbitration, each peripheral has a unique rank among all the peripherals. The rank can be represented as 1,2,3,…
- → If two peripherals simultaneously seek servicing the arbiter chooses the one with the higher rank

b. Rotating Priority Arbitration(round robin)

- → In rotating priority arbitration, the arbiter changes priority of peripheral based on the history of servicing of those peripherals.
- → This priority scheme grants services to the least-recently serviced of the contending peripherals.
- → Rotating priority ensures more equitable distribution of servicing the peripherals.



1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. Microprocessor stops executing its program and stores its state.
5. Microprocessor asserts *Inta*.
6. Priority arbiter asserts *Iack1* to acknowledge Peripheral1.
7. Peripheral1 puts its interrupt address vector on the system bus
8. Microprocessor jumps to the address of ISR read from data bus, ISR executes and returns (and completes handshake with arbiter).
9. Microprocessor resumes executing its program.

Figure: Arbitration using Priority arbiter

We prefer fixed priority when there is clear difference in priority among peripherals. However, in many cases the peripherals are somewhat equal, so arbitrary ranking them could cause high-ranked peripherals to get more servicing than low-ranked ones.

2. Daisy-chain arbitration

- → Daisy-chain arbitration method builds arbitration right into the peripherals.
- → Each peripheral has a request output and an acknowledge input
- → Each peripheral also has a request input and an acknowledge input
- → A peripheral activates its request output if it requires servicing or if its request input is activated. The latter means one of the " upstream" device is requesting servicing.
- → If any peripherals need servicing, its request will flow though the "downstream" peripheral and eventually reach the micro-processor.
- → If more than one peripheral request servicing the microprocessor will see only one request.
- → The microprocessor acknowledges signal connects the first peripheral. If this peripheral is requesting service, it proceeds to put its interrupt vector address on the system bus.

→ If it doesnot need service, then it instead passes the acknowledgement upstream to the next peripheral, by asserting its acknowledge output
→ This acknowledgement is received by the peripheral requesting the service, the peripheral places the vector address on the databus.
→ The peripheral at the front of the chain i.e. the one to which the processor acknowledge is connected has highest priority and the peripheral at the end of the chain has the lowest priority.

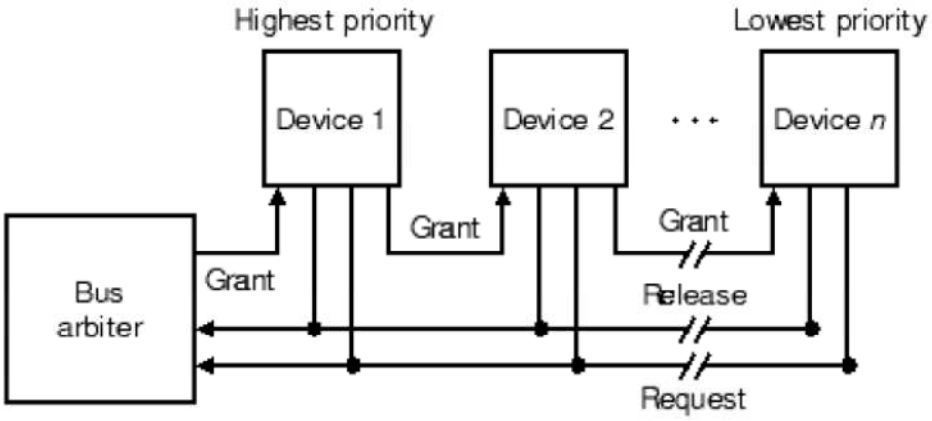


Figure: Daisy chained

1. Processor is executing its program
2. Any peripheral needing service activates " Req-out". This Req-out goes to the Req-in of the subsequent device in the chain
3. The peripheral nearest to the microprocessor asserts "Int".
4. The processor stores executing instruction and stores its state.
5. The processor asserts "Inta" to nearest device.
6. The "Inta" signal passes through the chain till it find a flag which is set by the device which has generated the interrupt.
7. The interrupting device sends the interrupt address vector to the processor for its interrupt service routine.
8. The processor jumps to the address of ISR reads from the data bus, ISR executes the return.
9. The flag is reset.
10. The processor checks for the next device that has interrupted.

3. Network-oriented Arbitration Method

→ Many embedded systems contain multiple microprocessors communicating via a shared bus; such a bus is sometime called a network.
→ Key feature of such connection is that a processor about to write to the bus has no way of knowing whether another processor is about to simultaneously write to the bus. This result in collision of data, thus the processor must be able to detect this collision stop transmission and wait for some time and then transmit again.
→ The protocol must ensure that the contending processor do not start sending again at the same time.
    This can be done by using statically methods that reduce of processor from sending data again at the same time.
→ Typically used to connect multiple on-chip processors.
→ CAN bus uses a cleaver address encoding scheme such that if two addresses are written simultaneously by different processor using the bus, the higher-priority address will override the lower-priority one.

4.7 Multilevel Bus Architecture

- → Within an embedded system, numerous type of communication take place, varying in their frequency and speed requirement.
- → The most frequent and high-speed communication will likely be between the microprocessor and its memory.
- → Least frequent communications, requiring less speed will be between the processor and its peripherals.
- → We often design system with two levels of buses a high-speed processor local bus and lower-speed peripheral bus.
- → The processor local bus connects the cache, memory controller and certain high-speed coprocessor.
- → The peripheral bus supports portability of peripherals it is slower than a processor local bus, requires fewer pins, fewer gates and less power for interfacing
- → A bridge connects the peripheral bus with the processor-local bus. The bridge is a single purpose processor that carries communication between buses.

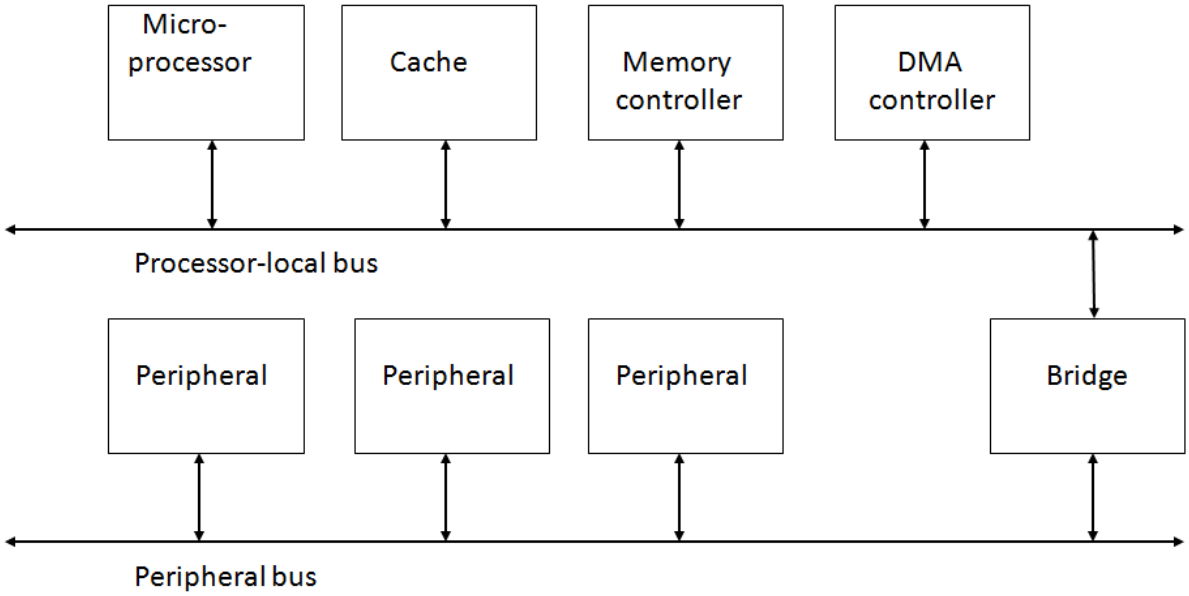


Figure: Multilevel Bus Architecture

4.8 Advance Communication Principle

a. Parallel Communication

- → Parallel communication takes place when the physical layer is capable of carrying multiple bits of data from one device to another.
- → Parallel communication has the advantage of high data throughput, if the length of bus is short
- → The length of a parallel bus must be kept short because long parallel wires will result in high capacitance value and variations in the length of individual wires of a parallel bus can cause the received bits to arrive at different times causing misalignment of data.
- → Parallel communication is used to connect device residing on same IC or same circuit board since the length of buses will be short.

b. Serial Communication

- → Serial communication involves a physical layer that carries one bit of data at a time.
- → Serial buses are capable of higher throughput than parallel when distance is significant buses because a serial bus will have less average capacitance, enabling it to send more bits per uint of time
- → Serial communication requires more complex interfacing logic and communication protocol because
  - **i. Sender needs to decompose words into bits**
  - **ii. Receiver needs to recompose bits into words**

iii. **Control signal often sent on same wire as data increasing protocol complexity.**

→ Data to be sent is placed within a start and stop bit.

c. Wireless Communication

→ Wireless communication eliminates the need for devices to be physically connected in order to communicate.

→ Physical layer for wireless communication is either as infrared(channel) or a radio frequency(RF) channel.

1. Infrared(IR)

- Use EM wave frequencies just below visible light spectrum
- An infrared diode emits infrared light to generate signal
- An IR transistor conducts when exposed to IR light and detects the signal
- Need line of sight for operation and thus has a limited communication range.

2. Radio Frequency(RF)

- Use EM wave frequencies in the radio spectrum
- Analog circuitry and antenna needs on both sides i.e. at the transmitter as well as receiver
- Line of sight isn't necessary, transmitter power determines the range of communication
- Transmitter and receiver must agree on a specific frequency in order to send and receive data.

d. Layering

→ Layering is a hierarchical organization of a communication protocol where lower level of the protocol provides services to the higher level.

→ The lowest layers consist of physical layer and higher layer consists of application layers.

→ The physical layer is medium to carry data from one action to another.

e. Error Detection and Correction

→ Error detection is the ability of a receiver to detect errors during transmission.

→ Error correction refers to the ability of a receiver and transmitter to cooperate to correct errors. This is usually done by acknowledgement/retransmission protocol

→ Parity is an extra bit sent with a word for error detection. In odd parity, data plus parity bit contains odd number of ones and for even parity they contain even number of ones.

→ Parity always detects single bit error.

→ Checksum is a stronger form of error checking while using checksum, an extra word is sent with data packet. The extra word contains XOR sum of all data words in the packets.

UNIT 5
Real time operating system

1.0.0 Definition of RTOS

A real time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources and provide consistent foundation for developing application code.

Application code designed on an RTOS can be quite diverse, ranging from simple application for a digital stopwatch to much more complex application for aircraft navigation. Goods RTOS, therefore are scalable in order to meet different set of requirement for different applications.

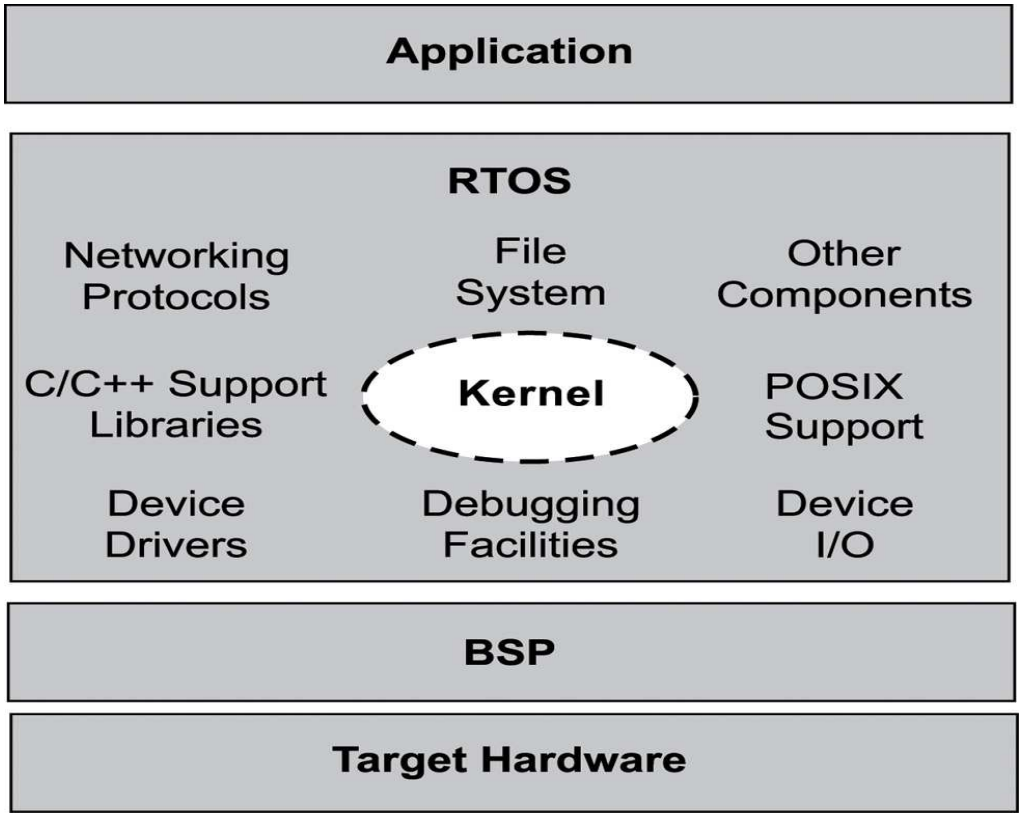


Figure: High-level view of an RTOS, its kernel, and other components found in embedded systems.

For example, in some application an RTOS comprises only a kernel, which is the core supervisory software that provides minimal logic scheduling and resource-management algorithm. Every RTOS has a kernel. On the other hand an RTOS can be a combinational of various modules, including the kernel, file system, networking protocol stack and other components required for particular application.

1.1.0 Key characteristics of an RTOS

1. **Reliability**

→ Embedded system must be reliable. Depending on the application, the system might need to operate for long period without human intervention.

2. **Predictability**
   → Meeting time requirement is key to ensure proper operation in Real Time embedded system. RTOS used in this case need to be predictable to certain degree.

3. **Performance**
   → This requirement dictates that an embedded system must perform fast enough to fulfill its timing requirement.
   → Throughput also measure the overall performance of the system, with hardware and software combined. One definition of throughput is the rate at which a system can generate output based on the input coming in.

4. **Compactness**
   → Application designed constraints and cost constraints help determine how compact an embedded system can be. A cell phone clearly must be small, portable and low cost. The design requirement limits system memory, which in turn limits the size of application and operating system.

5. **Scalability**
   → Because RTOSes can be used in a wide variety of embedded system, they must be able to scale up or down to meet application specific requirements. Depending on how much functionality is required, an RTOS should be capable of adding or deleting modular components, including file system and protocol stack.

1.2.0 Real time kernel

Most kernels contain the following components:

1. **Scheduler**

Scheduler is contained in each kernel and follows a set of algorithm that determines which task executes when. Some common examples of scheduling algorithm includes round robin and pre-emptive scheduling

2. **Objects**

Objects are special kernel construct that help developers create applications for real-time embedded systems. Common kernel objects includes task, semaphore and message queue.

3. **Services**

Services are operations that the kernel performs on an object, generally operation such as timing interrupt handling and resource management.

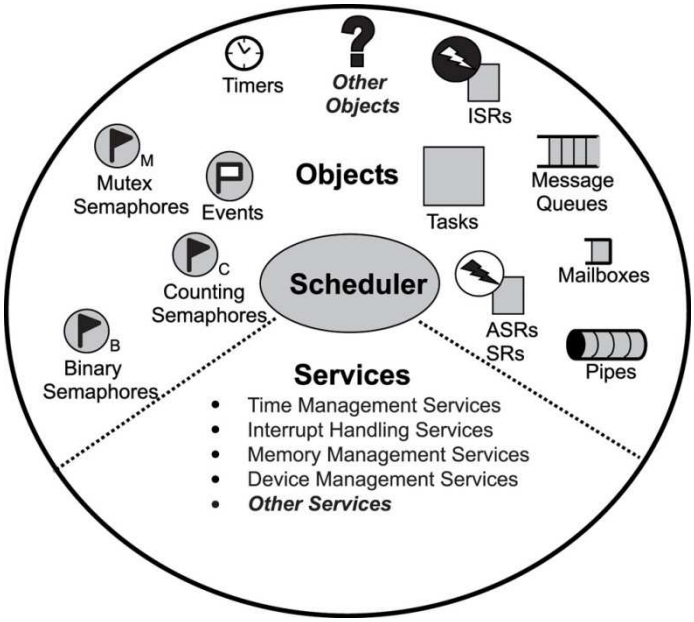


Figure: Common components in an RTOS kernel that including objects, the scheduler, and some services.

1.3.0 The scheduler

The scheduler is at the heart of every kernel. A scheduler provides the algorithm needed to determine which task executes when.

1.3.1 Schedulable Entity

A schedulable entity is a kernel object that can compete for execution time on a system, based on pre-defined scheduling algorithms. Task and processes are examples of schedulable entities found in most kernels.

1.3.2 Multitasking

Multitasking is the ability of the operating system to handle multiple activities within set deadlines. A real time kernel might have multiple tasks that it has schedule to run.

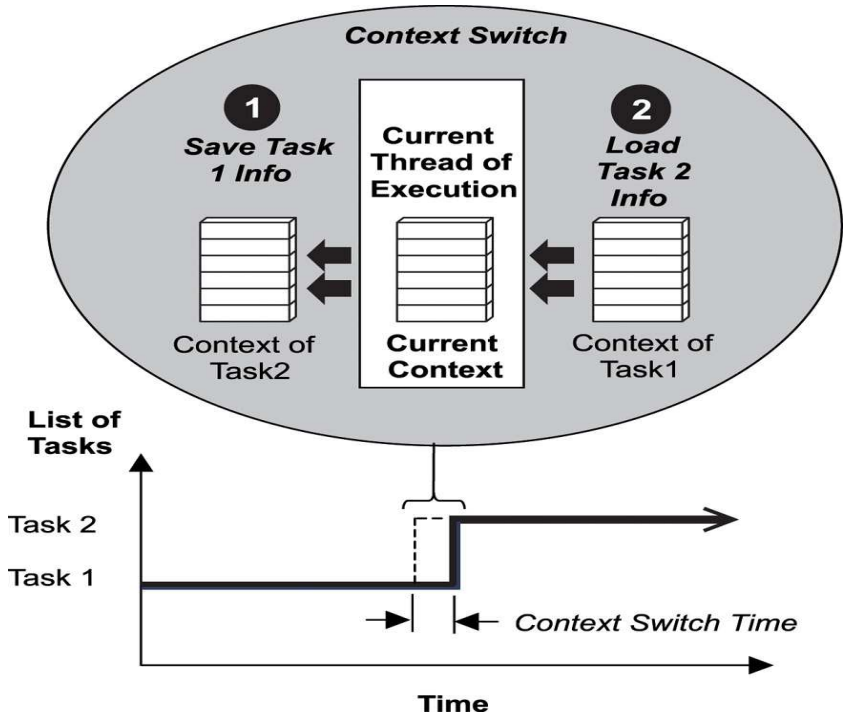


Figure: Multitasking using a context switch.

1.3.3 Context switch

Each task has its own context, which is the state of the CPU registers required each time it scheduled to run. A context switch occurs when the scheduler swithches from one task to another. Every time a new task is created, the kernel also creates and maintains an associated task control block (TCB). TCBs are system data structures that the kernel uses to maintain task specific information. When a task is running, its context is highly dynamic. This dynamic context is maintained in the TCB. When the task is not running, its context is frozen within the TCB, to be restored the next time the kernel runs.

As shown in figure above, when the kernel scheduler determines that it is needed to stop a running task 1 and start running task 2, then it takes following steps.

a) The kernel saves task 1's context information in its TCB.
b) It loads task 2's context information from its TCB, which becomes the current thread of execution.
c) The context of task 1 is frozen while task 2 executes, but if the scheduler needs to run task 1 again, task 1 continues from where it left off just before the context switch.

The time it takes for the scheduler to switch from one task to another is context switch. It is relatively insignificant compared to most operations that a task performs.

1.3.4 The dispatcher

The dispatcher is the part of the scheduler that performs context switching and changes the flow of execution.

1.4.0 Scheduling Algorithm

The scheduler determines which task run by following scheduling algorithm. Most kernels nowadays support two common scheduling algorithms.

I. Pre-emptive priority based scheduling

In this type of scheduling, the task that gets to run at any point is the task with highest priority among all other task ready to run in the system.
As shown in figure task 1 is pre-empted by higher priority task 2, which is pre-empted by task 3. When task 3 completes, task 2 resumes, likewise when task 2 completes, task 1 resumes.

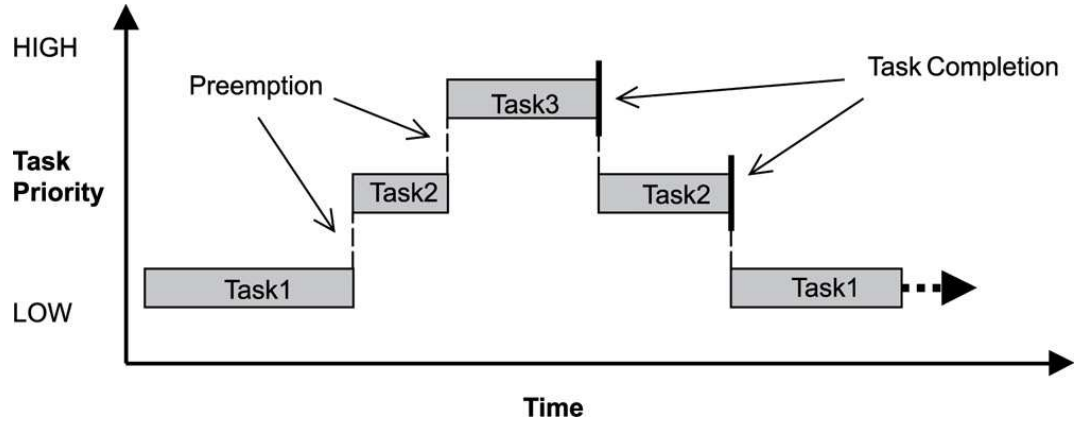


Figure: Preemptive priority-based scheduling

II. Round –Robin Scheduling

Round-Robin scheduling provides each task an equal share of the CPU execution time. Pure Round-Robin scheduling cannot satisy real time system requirement because in real-time system, task performs work of varying degrees of importance.
Instead pre-empted priority-based scheduling can be augmented with round-robin scheduling which uses time slicing to achieve equal allocation of cpu for task of same priority.

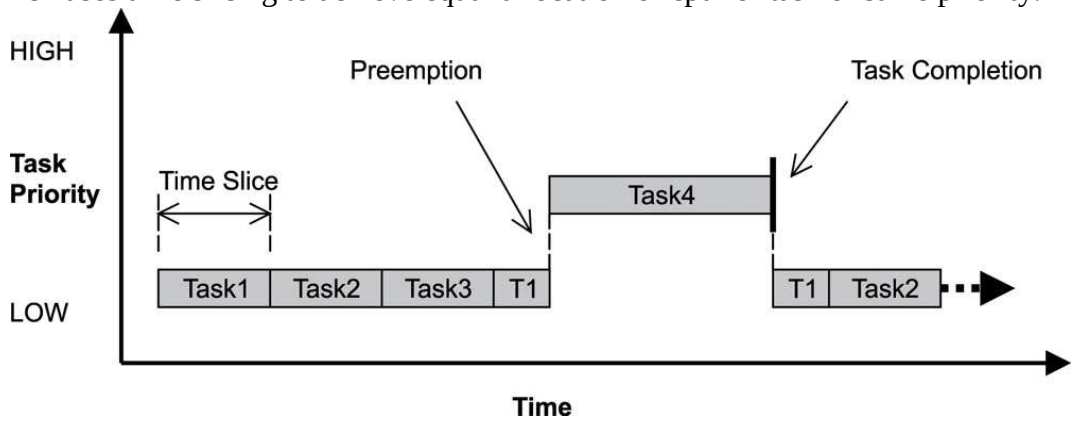


Figure: Round-robin and preemptive scheduling.

1.5.0 Objects

Kernel objects are special constructs that are the building block of application development for real-time embedded systems.
The most common RTOS kernel is

a) Task:

Tasks are concurrent and independent thread of execution that can compete for cpu execution time.

b) Semaphore

Semaphores are token-like objects that can be incremented or decremented by tasks for synchronization or mutual exclusion.

c) Message queue

Message Queue is buffer-like data structure that can be used for synchronization, mutual exclusion and data exchange by passing message between tasks.

1.6.0 Services

Most kernels provide services that help developer create applications for real time embedded system. These services comprise of set of API's calls that can be used to perform operation on kernel object or can be used in general to facilitate timer management, interrupt handling, device I/O and memory management.

1.7.0 Task

A task is an independent thread of execution that can compete with other concurrent tasks for processor execution time.

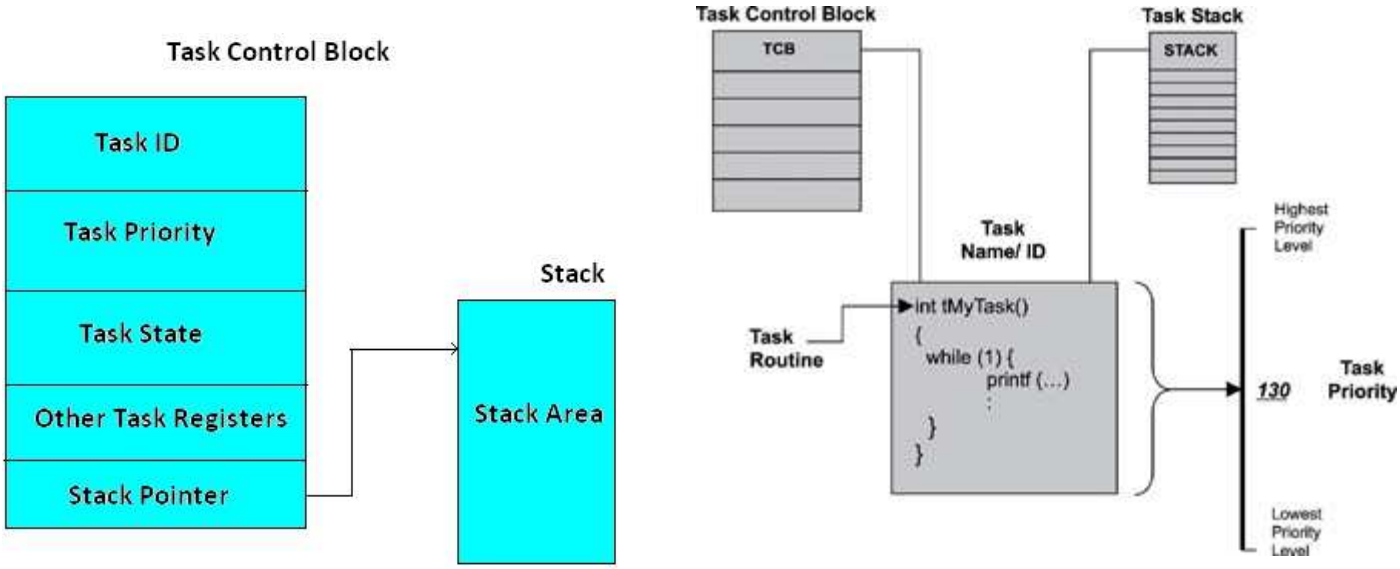


Figure: A task, its associated parameters, and supporting data structures.

A task is schedulable. A task is defined by its distinct set of parameters and supporting data structure.

Specifically, upon creation, each task has an associated name, a unique ID, priority, a task control block (TCB), a stack and task routine.

1.8.0 Task states and scheduling

Whether it is a system task or an application task at any time each task exists in
One of a small number of states, including

- Ready
- Running or
- Blocked

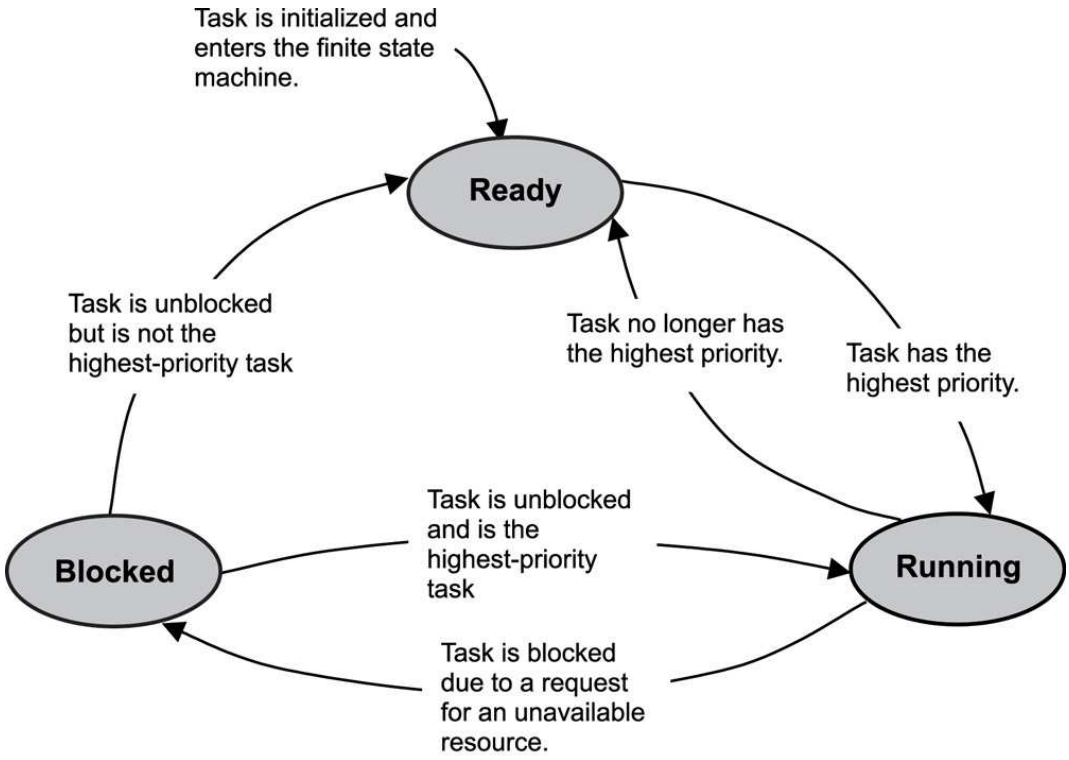


Figure: A typical finite state machine for task execution states.

Generally three main states are used in most typical pre-emptive scheduling kernel.

1. Ready-state
   → The task is ready to run but cannot because a higher priority task is executing.
2. Blocked state
   → The task has requested a resource that is not available, has requested to wait until some event occurs, or has delayed itself for some duration.
3. Running state
   → The task is the highest priority task and is running.

1.8.1 Ready State

When a task is created and made ready to run, the kernel puts it into the ready state. In this state, the task actively competes with all other ready task for processor execution time.
Figure below illustrates how a kernel scheduler might use a task-ready list to move from the ready state to the running state.
This example assumes a single processor system and a priority-based pre-emptive scheduling algorithm in which 255 is the lowest priority and 0 is the highest priority.

Figure: Five steps showing the way a task-ready list works.

In this example, task 1, 2, 3, 4, and 5 are ready to run, and the kernel queues them by priority in a task-ready list. Task 1 is the highest priority task (70); task 2, 3, and 4 are at the next-higher priority level (80) and task 5 is the lowest priority (90).

1. Task 1,2,3,4 and 5 are ready to run and are waiting in the task-ready list.
2. Because task 1 has the highest priority(70) it is first task ready to run. If nothing higher is running, the kernel removes task 1 form the ready list and moves it to the running state.
3. During execution, task 1 makes a blocking call. As a result, the kernel moves task 1 to the blocked state; takes task 2, which is first in the list of next higher priority task (80), off the ready list; and moves task 2 to the running state.
4. Next, task 2 makes a blocking call. The kernel moves task 2 to the blocked state; takes task 3, which is next in line of the priority 80 takes, off the ready list; and moves task 3 to the running state.
5. As task 3 runs, frees the resources that task 2 requested. The kernel returns task 2 to the ready state and inserts it at the end of the list of tasks ready to run at priority level 80. Task 3 continues as the currently running task.

1.8.2 Running State

On a single-processor system, only one task can run at a time. In this case, when a task is moved to the running state, the processor loads its registers with this task's context. The processor can then execute the task's instruction and manipulate the associated stack.
When a task moves from the running state to ready state, it is pre-empted by a higher priority task. In this case, the preempted task is put in the appropriate, priority based location in the task-ready list, and the higher priority task is moved form the ready state to the running state.
Running task can move to the blocked state in any of the following ways:

→ By making a call that request an unavailable resources
→ By making a call that request to wait for an event to occurs and
→ By making a call to delay the task for some duration.

1.8.3 Blocked state

If higher priority tasks are not designed to block, cpu starvation can result. CPU starvation occurs when higher priority task uses all of the cpu execution time and lower priority tasks do not get to run.
A task can only move to the blocked state by making a blocking call, requesting that some blocking condition be met. A blocked task remains blocked until the blocking condition is met.
Some of the blocking calls met are:

→ A semaphore taken for which a task is waiting is released.
→ A message on which the task is waiting arrives in a message queue or
→ A time delay imposed on the task expiries.

When tasks become unblocked, the task might move from the blocked state to the ready state if it is not the highest priority task.

2.0.0 Synchronization communication and concurrency

Tasks synchronize and communicate among themselves by using inter task primitives, which are kernel objects that facilities synchronization and communication between two or more thread of execution.

Examples of such objects include semaphore, message queue, signals and pipes.

Multiple concurrent threads of execution within an application must be able to synchronize their execution and coordinate mutually exclusive to access shared resources. To address these requirements, RTOS kernel provides a semaphore object and associated semaphore management service.

2.1.0 Defining Semaphore

- → A semaphore (sometime called semaphore token) is a kernel object that one or more thread of execution can acquire or release for the purpose of synchronization or mutual exclusion.
- → When a semaphore is first created the kernel assign to it an associated semaphore control block (SCB), a unique ID, a value (binary or count) and a task waiting list.

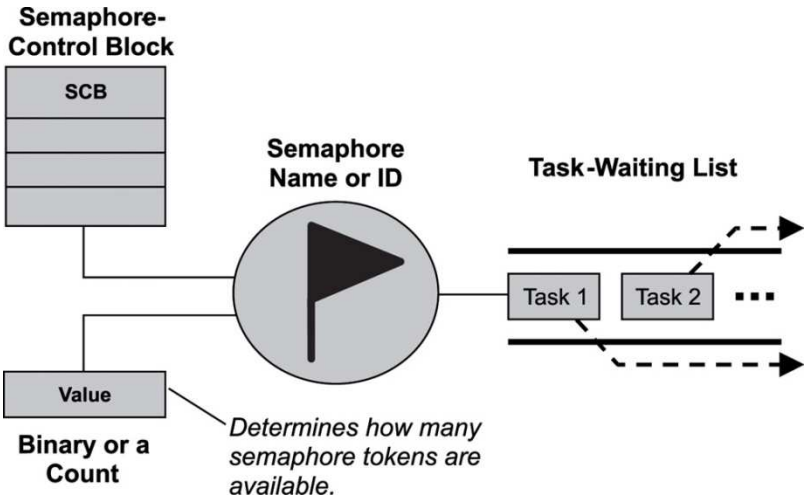


Figure: A semaphore, its associated parameters, and supporting data structures.

- → A semaphore is like a key that allows a task to carry out some operation or to access a resource.
- → If the task can acquire the semaphore, it can carry out the intended operation or access the resource. A single semaphore can be acquired a finite number of times. In this sense, acquiring a semaphore is like acquiring the duplicate of a key from an apartment manager, when the apartment manager runs out of duplicates keys the manager can give out no more keys. Likewise, when a semaphore's limit is reached it can't give any more token to waiting task.

The kernel tracks the number of time a semaphore has been acquired or released by maintaining a token count, which is initialized to a value when the semaphore is created.

- → As the task acquires the semaphore, the token count is decremented; as token releases the semaphore count is incremented.

- → If the token count reaches 0, the semaphore has no token left. A requesting task therefore cannot acquire the semaphore and the task block if it chooses to wait for the semaphore to become available.
- → The task-waiting list tracks all task blocked while waiting on an unavailable semaphore. These blocked tasks are kept in the task-waiting list either first in/ first out (FIFO) order or higher priority order.
- → When an unavailable semaphore becomes available, the kernel allows the first task in the task-waiting list to acquire it. The kernel moves this unblocked task to either to running state, if it is the highest priority task or to the ready state, until it become the highest priority task and is able to run.
- → A kernel can support many different types of semaphore, including binary, counting and Mutex semaphore.

2.1.2 Typical Semaphore Use

Semaphores are useful either for synchronizing execution of multiple tasks or for coordinating access to shared resources.

1. **Wait and signal synchronization**
   - → Two tasks can communicate for the purpose of synchronization without exchanging data. For e.g. a binary semaphore can be used between two tasks to coordinate the transfer of execution control as shown below.

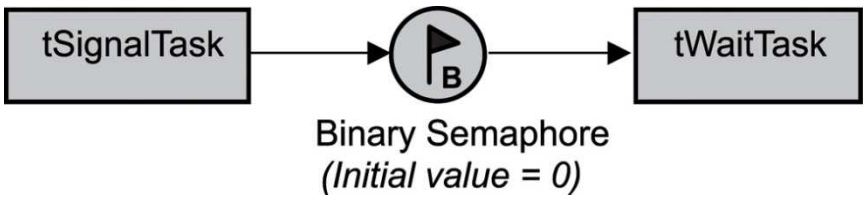


Figure: Wait-and-signal synchronization between two tasks.

In this situation, a binary semaphore is initially unavailable (value of 0). tWaitTask has higher priority and run first. The task makes a request to acquire the semaphore but is blocked because the semaphore is unavailable.
This step gives the lower priority tSignalTask a chance to run; at some point tSignalTask release the binary semaphore and unblocks the tWaitTask.

```
tWaitTask( )
{
    :
    Acquire binary semaphore token
    :
}

tSignalTask( )
{
```

:
Release binary semaphore token
:
}

Because tWaitTask's priority is higher than tSignalTask's priority as soon as the semaphore is released, tWaitTask pre-empts tSignalTask and starts to execute.

2. Multiple-Task wait and signal synchronization

→ When coordinating the synchronization of more than two tasks, use the flush operation on the task-waiting list of a binary semaphore.

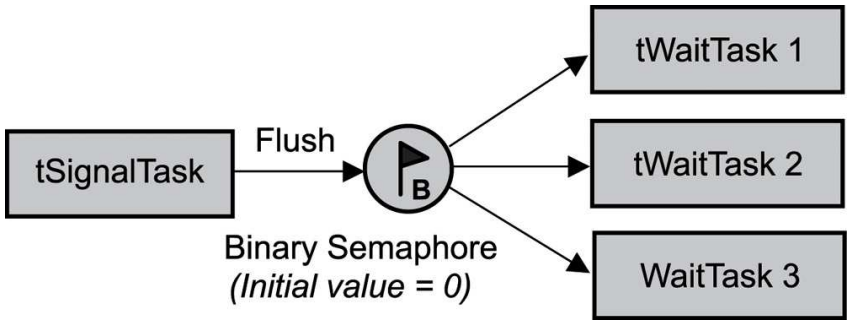


Figure: Wait-and-signal synchronization between multiple tasks

→ The binary semaphore is initially unavailable. The higher priority tWaitTask 1, 2, and 3 all do some processing; when they are done; they may try to acquire the unavailable semaphore and as a result, blocks.

This action gives tSignalTask a chance to complete its processing and execute a flush command on the semaphore, effectively unblocking the three tWaitTask.

3. Single Shared Resource access synchronization

One of the most common uses of semaphore is to provide for mutually exclusive access to a shared resource.

A shared resource might be a memory location, a data-structure or an I/O device essentially anything that might have to be shared between two or more concurrent threads of execution.

A semaphore can be used to serialize access to a shared resource as shown below.

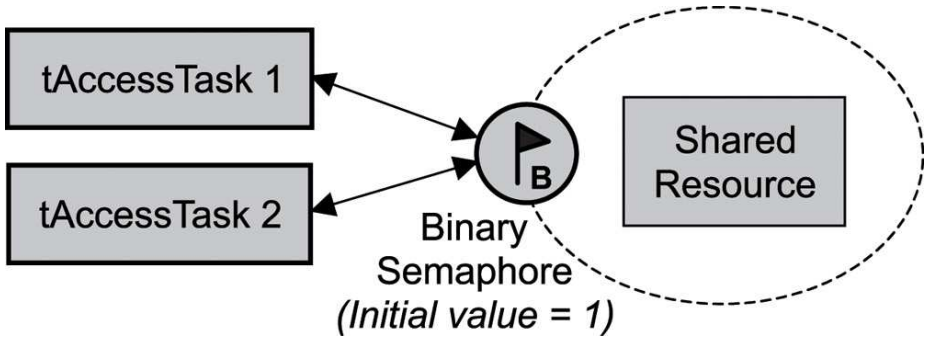


Figure: Single shared-resource-access synchronization

In this scenario, a binary semaphore is initially created in the available state (value =1) and is used to protect the shared resource.
To access the shared resource, task 1 or 2 need to first successfully acquire the binary semaphore before reading from or writing to the shared resource.

```
tAccessTask( )
{
        :
        Acquire binary semaphore token
        Read or write to shared resource
        Release binary semaphore token
}
```

→ This code serializes the access to the shared resource. If tAccessTask1 executes first it makes a request to acquire the semaphore and is successfully become the semaphore is available.
→ Having acquired the semaphore; this task is granted access to the shared resource and can read and write to it.

Mean while, the higher priority tAccessTask2 wakes up and runs due to timeout or some external events. It tries to access the same semaphore but is blocked because tAccessTask1 currently has accessed to it. After tAccessTask1 release the semaphore, tAccessTask2 is unblocked and starts to execute.

One of the danger of this design is that any task can accidently release the binary semaphore, even one that never acquire the semaphore in the first place. If this issue were to happen in this scenario, both tAccessTaks1 and tAccessTaks2 could end up acquiring the semaphore and reading and writing to the shared resource at the same time, which could lead to the incorrect program behavior.

To ensure that this problem does not happen, use a Mutes semaphore instead. Because a mutex supports the concept of ownership, it ensures that only the task that successfully acquire (locked) the mutex can release (unlock) it.

4. Recursive Shared-Resource access synchronization

→ Sometimes a developer might want to access a shared resource recursively. This situation might exist if tAccessTask calls routine A that calls routine B, and all three need access to the shared resource as shown in figure below.

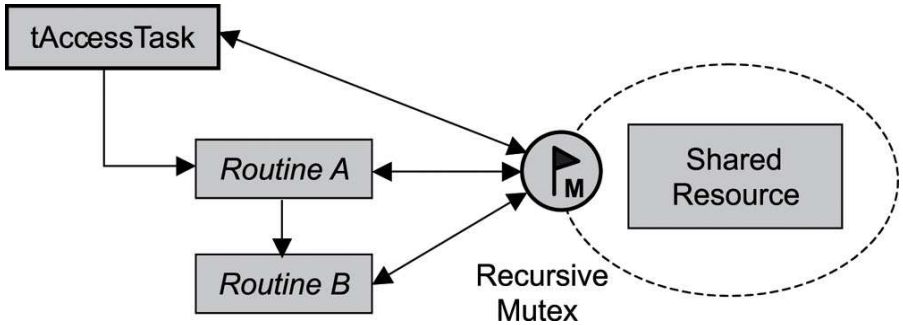


Figure: Recursive shared- resource-access synchronization

→ If a semaphore were used in this scenario, the task would end up blocking; causing a deadlock, when a routine is called from a task the routine effectively becomes a part of the task.

→ When routine A runs, therefore it is running as a part of tAccessTaks. Routine A trying to acquire the semaphore is effectively the same as tAccessTask trying to acquire the same semaphore. In this case, tAccessTask would end up blocking while waiting for the unavailable semaphore that it already has.

→ One solution to this situation is to use a recursive mutex
After tAccessTaks locks the mutex, the task own it. Additional attempts from the task itself or from routine that it calls to lock the mutex succeeded. As a result, when routine A and Routine B attempt to lock the mutex, they succeed without blocking.

```
tAccessTaks( )
{
        :
        Acquire the mutex
        Access the shared resource
        Call Routine A
        Release Mutex
        :
}

Routine A( )
{
        :
        Acquire the mutex
        Access shared resource
        Call routine B
        Release Mutex
        :
}

Routine B()
{
        :
        Acquire the mutex
        Access shared resource
        Release mutex
        :
}
```

5.0 Multiple shared resource access synchronization

→ For case in which multiple equivalent shared resources are used a counting semaphore comes in handy as shown in figure below

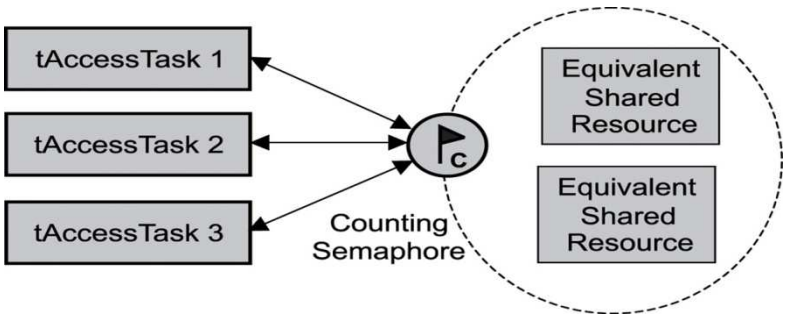


Figure: Single shared-resource-access synchronization

→ Note that this scenario doesnot work if the shared resource are not equivalent. The counting semaphore's count is initially set to number of equivalent shared resources; in this example 2 as a result, the first two tasks requesting a semaphore token are successful. However, the third task ends up blocking until one of the previous two tasks release a semaphore token, as shown in figure.

3.0.0 Message Queue:

Introduction:

Tasks must also be able to exchange message. To facilitate inter-data communication kernel provides a message queue object and message queue management service.

3.1.0 Defining Message Queue

A message queue is a buffer-like object through which tasks and ISRs send and receive message to communicate and synchronize with data.

A message queue is like a pipeline. It temporary holds message from the sender until the intended receiver is ready to read them.

When a message queue is created, it is assigned an associated queue control block(QCB), a message queue, a message queue name, a unique ID memory buffers, a queue length, a maximum message length and one or more task waiting list as shown in figure below:

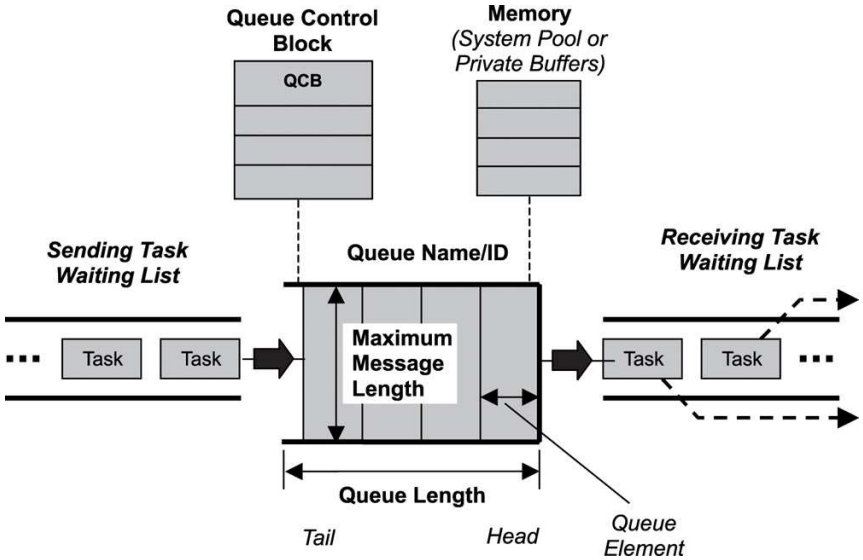


Figure: A message queue, its associated parameters, and supporting data structures

3.2.0 Typical Message Queue Use

→ One of the simplest scenario for message base communication require a sending task(also called message source), a message queue , and a receiving task(also called a message sink) as shown below

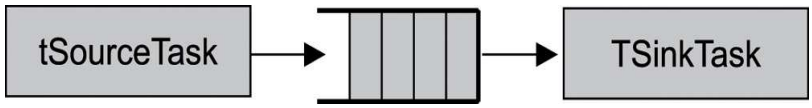


Figure: Non-interlocked, one-way data communication

→ tsourceTask simply sends a message; it doesn't require acknowledgement from tSinkTask

```
tSourceTask( )
{
        :
        Send message to message queue
        :
}

tSinkTask( )
{
        :
        Receive message from message queue
        :
}
```

→ If tSinkTask is set to highest priority, it runs first until it blocks on an empty message queue. As soon as tSourceTask sends the message to the queue, tSinkTask receive the message and starts to execute again,

→ If tSinkTask is set to a lower priority, tSourceTask fills the message queue with message. Eventually, tSourceTask can be made to block when sending a message to a full queue. This action makes tSinkTask wakeup and start taking message out of the message queues.

4.0.0 Pipes:

Pipes are kernel objects that provide unstructured data exchange and facilitate synchronization among tasks.

In a traditional implementation, a pipe is a unidirectional data exchange facility, as shown below.

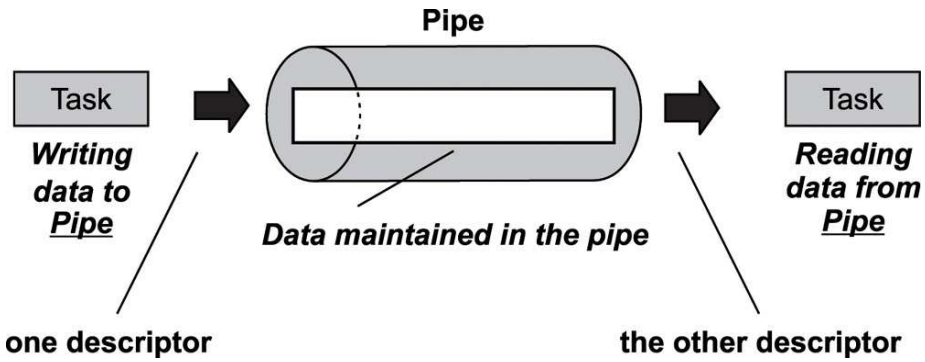


Figure: A common pipe unidirectional

Two descriptors one for each end of the pipes is returned when a pipe is created. Data is written via one descriptor and read via another. The data remains in the pipe as an unstructured byte stream. Data is read from the pipe in FIFO order.

A pipe provides a simple data flow facility so that the reader becomes blocked when pipe is empty and the writer becomes blocked when the pipe is full.

Typically, a pipe is used to exchange data between a data producing task and a data consuming task, as shown in figure below. It is also permissible to have several writers for the pipe with multiple readers on it.

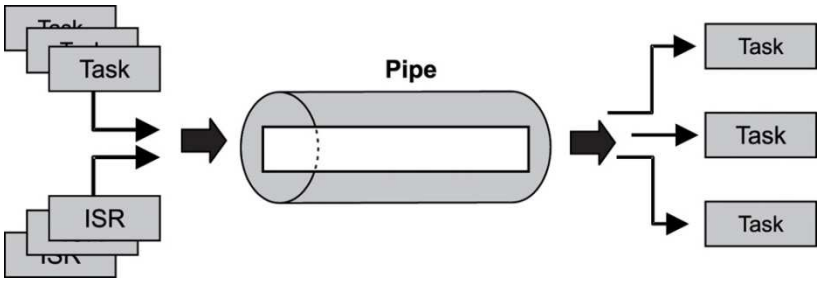


Figure: Common pipe operation.

5.0.0 Event Register

Some kernel provides a special register as part of each task's control block as shown in figure below. This register called event register is an object belonging to a task and consists of a group of binary events flags used to track the occurrence of specific events.

Depending upon kernel implementation of this mechanism, an event register can be 8-16 or 32 bit wide, maybe even more. Each bit in the event register is treated like a binary flag and can be either set or cleared.

Through the event register, a task can check for the presence of particular events that can control its execution.

An external source such as another task or ISR, can set bits in the event register to inform the task that a particular event has occurred.

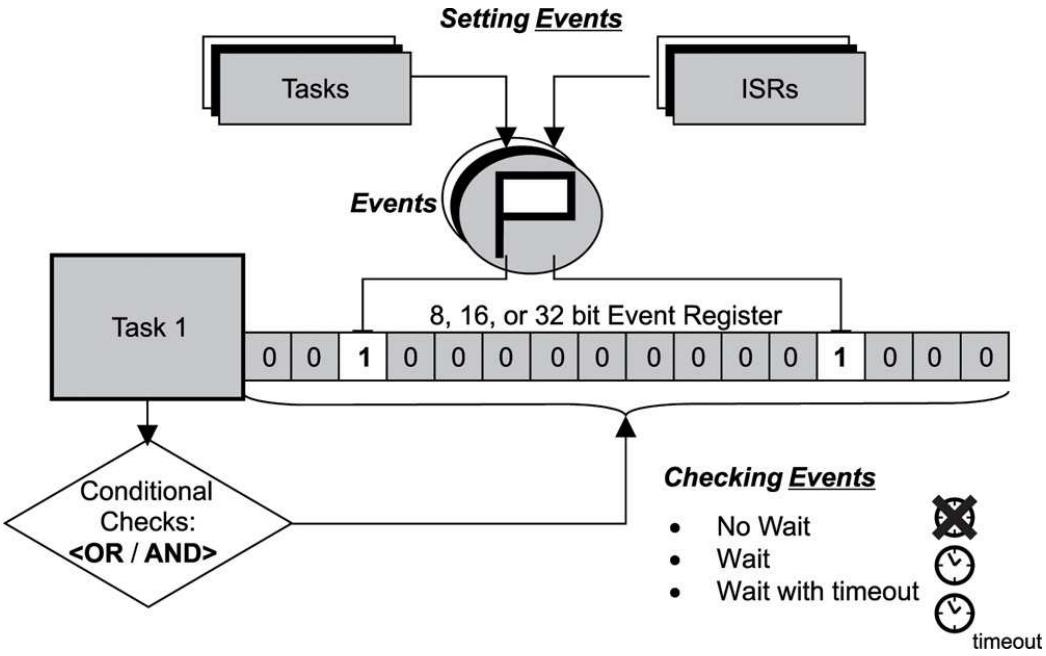


Figure: Event registers.

Application defines the event associated with an event flag. This definition must be agreed upon between the event sender and receiver using the event registers.

6.0.0 Mailbox:

A mailbox is an object used to achieve synchronization and communication by passing messages in system (shared) memory space. Functions are provided for creating and deleting a mailbox, sending and receiving messages in a mailbox, and referencing the mailbox status. A mailbox is an object identified by an ID number called a mailbox ID.

A mailbox has a message queue for sent messages, and a task queue for tasks waiting to receive messages. At the message sending end (making event notification), messages to be sent go in the message queue. On the message receiving end (waiting for event notification), a task fetches one message from the message queue. If there are no queued messages, the task goes to a state of waiting for receipt from the mailbox until the next message is sent. Tasks waiting for message receipt from a mailbox are put in the task queue of that mailbox.

7.0.0 Kernel Services:

Some of the notable services provides by kernel are

- Memory Management
- Timer Functions
- Interrupt Processing

7.1.0 Memory management

Embedded system developers commonly implement custom memory-management facilities on top of what the underlying RTOS provides. Understanding memory management is therefore an important aspect of developing embedded systems.

Knowing the capability of the memory management system can aid application design and help avoid pitfalls.

For example, in many existing embedded application, the dynamic memory allocation routine, malloc is often called. It can create an undesirable side effect called memory fragmentation.
We know that the program code, program data and the system stack occupy the physical memory after program initialization completes. Either RTOS or kernel uses the remaining physical memory for dynamic memory allocation. This memory area is called the heap.
The heap is broken into smaller, fixed size block. Each block has a unit size that is power of two to ease translating a requested size into the corresponding required number of units. Here the unit size is 32 byte.
The dynamic memory allocation function, malloc has an input parameter that specifies the size of the allocation requested in bytes.
Malloc allocates larger block, which is made up of one or more of the smaller fixed size block. The size of this larger memory block is at least as large as the requested size. For eg, if the allocation requests 100 bytes, the returned block has a size of 128 bytes (4 units' ×32 bytes/unit). As a result, the requestor does not use 28 bytes of the allocated memory which is called memory fragmentation.
This specific form of fragmentation is called internal fragmentation because it is internal to the allocated block.

Malloc operation involves the following steps.

1. Examine the heap to determine if a free block that is large enough for the allocation request exists.
2. If no such block exists, return an error to the caller
3. Retrieve the starting allocation-array index of the free range of the heap.
4. Update the allocation array by marking the newly allocated block.
5. If the entire block is used to satisfy the allocation, update the heap by deleting the largest node, otherwise update the size.
6. Rearrange the heap array.

The Free Operation

The main operation of the free function is to determine if the block being freed can be merged with its neighbors.
The free operation involves the following steps.

1. Update the allocation array and merge neighboring blocks if possible.
2. If the newly freed block cannot be merged with any of its neighbors, insert a new entry into the heap array.
3. If the newly freed block can be merged with one of its neighbors, the heap entry representing the neighboring block must be updated and the update entry rearranged according to its new size.
4. If the newly freed block can be merged with both of its neighbors, the heap entry representing one of the neighboring block must be deleted from the heap, and the heap entry representing other neighboring block must be updated and re-arranged according to its new size.

8.0.0 Fixed size memory management in embedded system

- → Another approach to memory management uses the method of fixed size memory pools. As shown in figure below, the available memory space is divided into variously sized memory pools.
- → All blocks of the same memory pool have the same size. Here the memory space is divided into three pools of block sizes 32, 50 and 128 respectively.
- → The memory pools are linked together and sorted by size. Finding the smallest size adequate for an allocation requires searching through this link and examining each control structure for the first adequate block size.

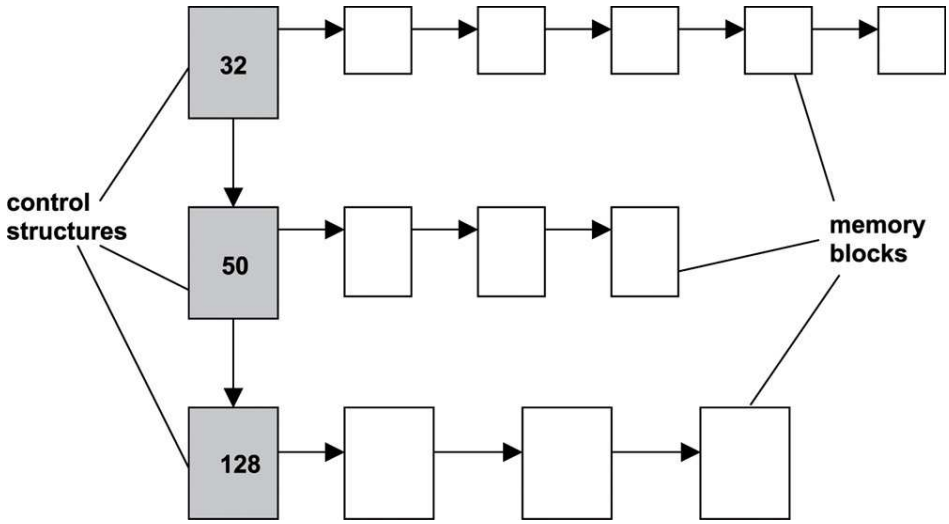


Figure: Management based on memory pools.

- → A successful allocation results in an entry being removed from the memory pool. A successful de-allocation results in an entry being inserted back into memory pool.

8.2.0 Blocking vs. Non-Blocking Memory Function

The malloc and free functions do not allow the calling task to block and wait for memory to become available. When a memory allocation request fails, the task must back track to an execution checkpoint and perhaps restart an operation. This issue is undesirable as the operation can be expensive. If tasks have built-in knowledge that the memory congestion condition can occurs but only momentarily, the task can be designed to be more flexible.

If such task can tolerate the allocation delays, the task can choose to wait for memory to become available instead of either failing entirely or backtracking.

In practice, a well designed memory allocation function should allow for allocation that permits blocking forever, blocking for a time out, or no-blocking at all.

As shown in figure below, a blocking memory allocation function can be implemented using both a counting semaphore and a mutex lock.

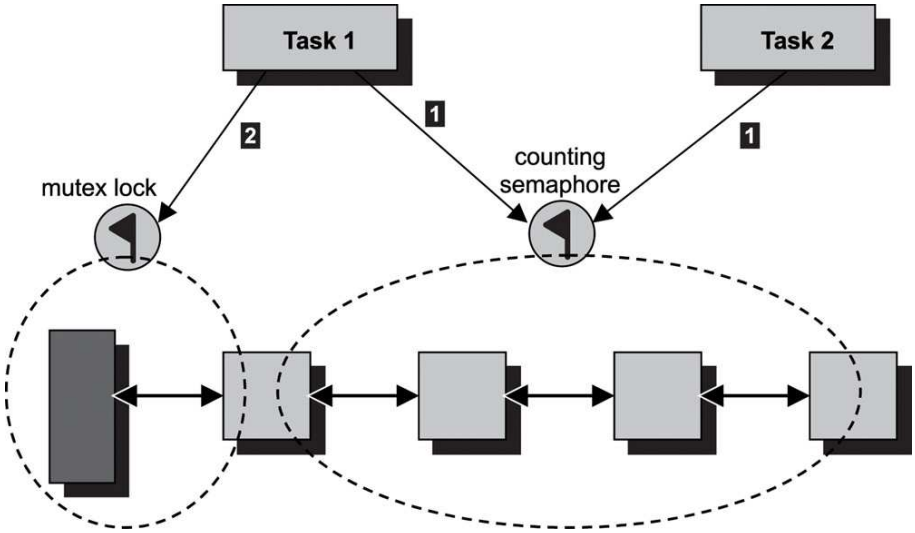


Figure: Implementing a blocking allocation function using a mutex and a counting semaphore

The counting semaphore is initialized with the total number of available memory block at the creating of memory pool.

- → Multiple tasks can access the free-blocks list of memory pool. The control structure is updated each time an allocation or a de-allocation occurs.
- → A mutex lock is used to guarantee a task exclusive access to both the free-block list and the control structure. A task might wait for a block to become available, acquire the block and then continue its execution.
- → For an allocation request to succeed, the task must first successfully acquire the counting semaphore, followed by a successful acquisition of the mutex lock.
- → The successful acquisition of the counting semaphore reserves a piece of the available blocks from the pool. A task first tries to acquire the counting semaphore. If no blocks are available the task blocks on the counting semaphore, assuming the task is prepared to wait for it.
- → If a resource is available, the task acquires the counting semaphore successfully. The counting semaphore token count is now one less it was.
- → At this point the task has reserved a piece of the available block but has yet to obtain the block. Next the task tries to lock the mutex. If another task is currently getting a block out of the memory pool or if another task is currently freeing a block back into the memory pool, the mutex is in the locked state. The task blocks waiting for the mutex to unlock. After the task locks the mutex, the task retrieves the resource from the list.

The counting semaphore is released when the task finishes the memory block.

Acquire (Counting_semaphore)

{

Lock(Mutex)

Retrive the memory block from the pool

Unlock(Mutex)

}

For De-allocation:

Lock (Mutex)

Release the memory block back into the pool

Unlock (Mutex)

Release (Counting_semaphore)

9.0.0 Exceptions and Interrupts

Introduction:

Exceptions and interrupts are part of a mechanism provided by the majority of embedded processor architectures to allow for the disruption of the processor's normal execution path. This disruption can be triggered either intentionally by application software or by an error, unusual condition, or some unplanned external event.

9.0.1 What are Exceptions and Interrupts?

An exception is any event that disrupts the normal execution of the processor and forces the processor into execution of special instructions in a privileged state. Exceptions can be classified into two categories: synchronous exceptions and asynchronous exceptions.
Exceptions raised by internal events, such as events generated by the execution of processor instructions, are called synchronous exceptions. Examples of synchronous exceptions include the following:

- → On some processor architectures, the read and the write operations must start at an even memory address for certain data sizes. Read or write operations that begin at an odd memory address cause a memory access error event and raise an exception (called an alignment exception).
- → An arithmetic operation that results in a division by zero raises an exception.

Exceptions raised by external events, which are events that do not relate to the execution of processor instructions, are called asynchronous exception

Examples of asynchronous exceptions include the following:

- → Pushing the reset button on the embedded board triggers an asynchronous exception (called the system reset exception).
- → The communications processor module that has become an integral part of many embedded designs is another example of an external device that can raise asynchronous exceptions when it receives data packets.

An interrupt, sometimes called an external interrupt, is an asynchronous exception triggered by an event that an external hardware device generates. Interrupts are one class of exception

What differentiates interrupts from other types of exceptions, or more precisely what differentiates synchronous exceptions from asynchronous exceptions, is the source of the event.
The event source for a synchronous exception is internally generated from the processor due to the execution of some instruction. On the other hand, the event source for an asynchronous exception is an external hardware device.

9.0.2 Applications of Exceptions and Interrupts

From an application's perspective, exceptions and external interrupts provide a facility for embedded hardware (either internal or external to the processor) to gain the attention of application code. Interrupts are a means of communicating between the hardware and an application currently running on an embedded processor.

In general, exceptions and interrupts help the embedded engineer in three areas:

- → Internal errors and special conditions management
- → Hardware concurrency, and
- → Service requests management.

1. Internal Errors and Special Conditions Management

Handling and appropriately recovering from a wide range of errors without coming to a halt is often necessary in the application areas in which embedded systems are typically employed. Exceptions signal can arises due to some internal errors. Such signal helps the designer to find out the internal errors and also exception helps in programmer from entering in to privilege mode or unprivileged mode for making some configurations.

2. Hardware Concurrency and Service Request Management

The ability to perform different types of work simultaneously is important in embedded systems. Many external hardware devices can perform device-specific operations in parallel to the core processor. These devices require minimum intervention from the core processor. The key to concurrency is knowing when the device has completed the work previously issued so that additional jobs can be given. External interrupts are used to achieve this goal.

For example, an embedded application running on a core processor issues work commands to a device. The embedded application continues execution, performing other functions while the device tries to complete the work issued. After the work is complete, the device triggers an external interrupt to the core processor, which indicates that the device is now ready to accept more commands. This method of hardware concurrency and use of external interrupts is common in embedded design.

9.1.0 Processing General Exceptions

The processor takes the following steps when an exception or an external interrupt is raised:

- → Save the current processor state information
- → Load the exception or interrupt handling function into the program counter
- → Transfer control to the handler function and begin execution
- → Restore the processor state information after the handler function completes.
- → Return from the exception or interrupt and resume previous execution.

9.1.2 Saving Processor States

When an exception or interrupt comes into context and before invoking the service routine, the processor must perform a set of operations to ensure a proper return of program execution after the service routine is complete. Exception and interrupt service routines also to store blocks of information, called processor state information, somewhere in memory.

The processor typically saves a minimum amount of its state information, including the status register (SR) that contains the current processor execution status bits and the program counter (PC) that contains the returning address, which is the instruction to resume execution after the exception.

10.0.0 Interrupt processing:

Interrupt routines in most RTOS environment must follow two rules that do not apply to task code.

Rule 1:

An interrupt routine must not call any RTOS function that might block the caller.

- → Interrupt routines must not get semaphores, read from queues or mailboxes that might be empty, or wait for events and so on.
- → If an interrupt routine calls an RTOS function and gets blocked, then in addition to interrupt routine, the task that was running when the interrupt occurred will be blocked even if the task is the highest-priority task.
- → Most interrupt routines must run to completion to reset the hardware to be ready for the next interrupt.

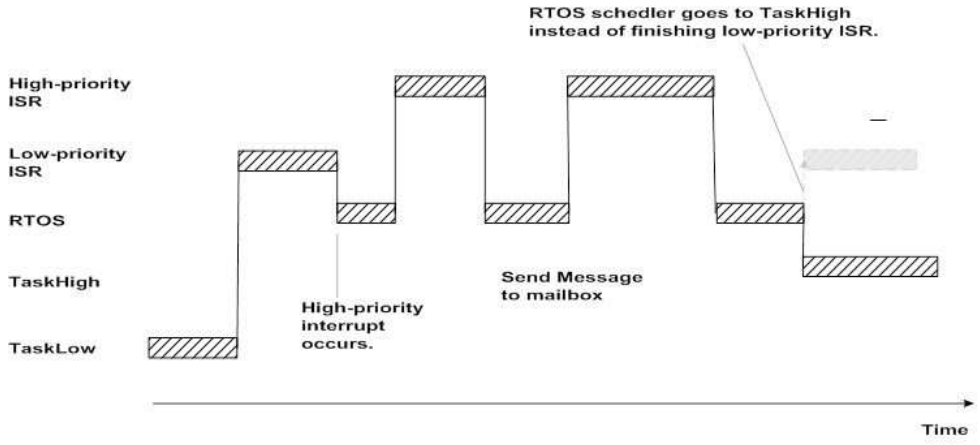


Figure: Nested Interrupts and the RTOS

```
Example:
Static int iTemperatures[2];
Void interrupt vReadTemperatures(void)
{
GetSemaphore(SEMAPHORE_TEMPERATURE);/*Not Allowed */
iTemperatures[0]=!! Read in value from hardware;
iTemperatures[1]=!! Read in value from hardware;
GiveSemaphore(SEMAPHORE_TEMPERATURE);
}
Void interrupt vTaskTestTemperatures(void)
{
int iTemp0, iTemp1;
While(true){
GetSemaphore(SEMAPHORE_TEMPERATURE);
iTemp0=iTemperatures[0];
iTemp1=iTemperatures[1];
 GiveSemaphore(SEMAPHORE_TEMPERATURE);
If(iTemp0!=iTemp1)
        !!set off howling alarm.
}
```

- Here in the code above the task code and the interrupt routine share the same temperature data with the semaphore.
- This code violates the rule 1.
- If the interrupt routine happens to interrupt vTaskTestTemperature while it had the semaphore, then when the interrupt routine called GetSemaphore, the RTOS would notice that the semaphore was already taken and block.
- This will stop the interrupt routine and vTaskTestTemperature function since both function is blocked waiting for semaphore

Rule 2

An interrupt routine may not call any RTOS function that might cause the RTOS to switch tasks unless the RTOS knows that an interrupt routine, and not the task is executing

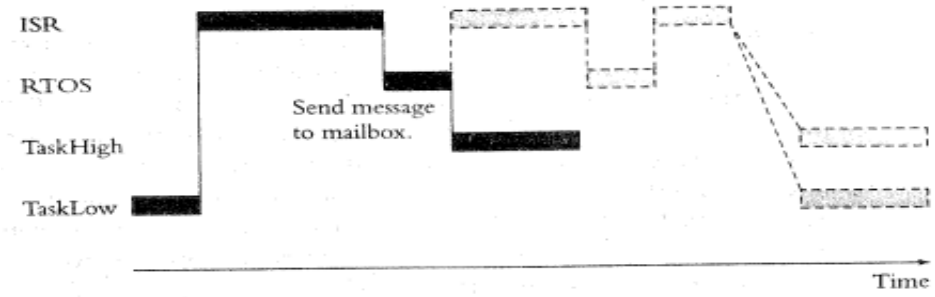


Figure above shows the microprocessor attention shifted from one part of the code to another over time.

- → The interrupt routine interrupts the lower-priority task and then calls RTOS to write a message to a mailbox (legal according to rule 1)
- → When the interrupt routine exits, the RTOS arranges for the processor to execute either the original task or if the higher-priority task was waiting on the mailbox, that higher-priority task.
- → If the higher-priority task is blocked on the mailbox, then as soon as the interrupt routine writes to the mailbox, the RTOS unblocks the higher-priority task.
- → In the worst case, if higher-priority task is blocked on the mailbox, then as soon as interrupt routine writes to the mailbox, the RTOS unblocks the higher-priority task.
- → RTOS knowing nothing about the interrupt routine notices that the task that it thinks is running is no longer the highest-priority task that is ready to run.
- → **Instead of returning to the interrupt routine, the RTOS switches to the higher priority task. The interrupt routine doesn't get to finish until later**.

10.0.1 *Various method used to solve this problem is:*

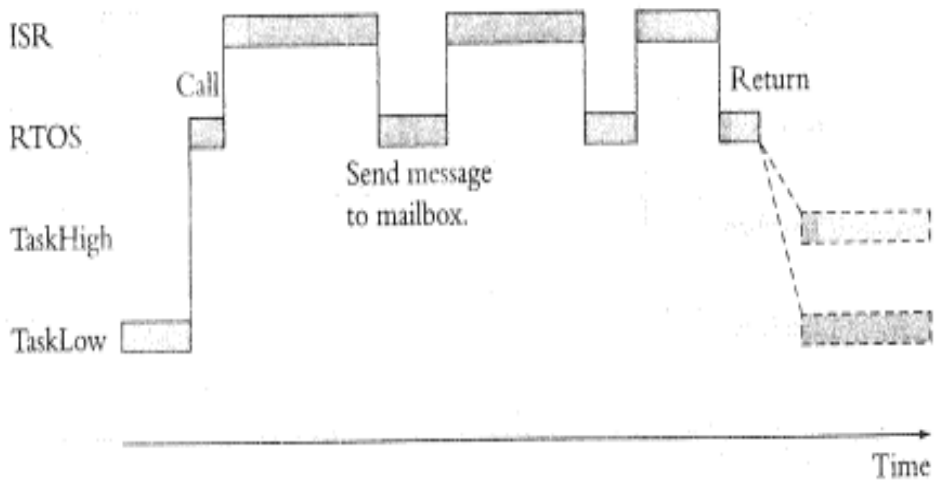
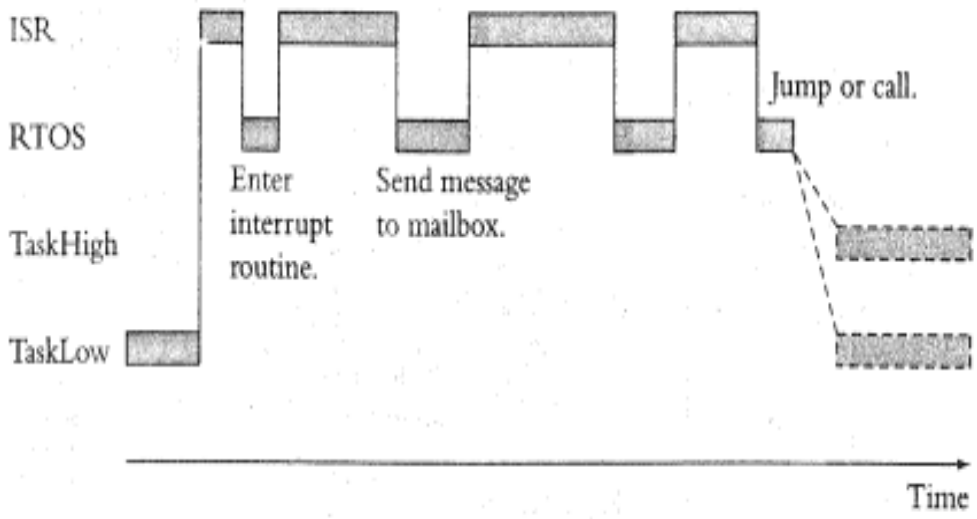


Figure: how interrupt should work

- → In this scheme, RTOS intercepts all the interrupts and then call interrupts routine.
- → Doing this helps RTOS to find out when an interrupt routine has started. When the interrupt routine later writes to the mailbox, RTOS knows to return to the interrupt routine and not to switch tasks, no matter what task is unblocked by the write to the mailbox. When the interrupt routine is over, it returns and the RTOS gets control again. The RTOS scheduler then figures out what task should not get the microprocessor.
- → If RTOS uses this method, then we need to call some function within the RTOS that tells the RTOS where interrupt routines are and which hardware interrupts corresponds to which interrupt routine.



- → In this scheme RTOS provides a function that the interrupt routine calls to let the RTOS know that an interrupt is running.
- → After the call to that function, the RTOS knows that an interrupt routine is in progress, and when the interrupt routine writes to the mailbox, the RTOS always returns to the interrupt routine no matter what task is ready.
- → When the interrupt routine is over, it jumps to or calls some other function in the RTOS, which calls the scheduler to figure out what task should now get the processor. Essentially this procedure disables the scheduler for the duration of the interrupt routine.
- → In this scheme, interrupt routine must call the appropriate RTOS function at the right moments.

Rule 2 and Nested Interrupt

In case of interrupt nest, if a higher-priority interrupt interrupts a lower-priority interrupt routine, then another scheme comes into play.

- → If the higher-priority interrupt routine makes any call to RTOS function, then the lower-priority interrupt routine must let RTOS know when the lower-priority interrupt occurs.
- → When higher-priority interrupt routine ends the RTOS scheduler may run some other task rather letting the lower-priority interrupt routine to complete.

→ RTOS scheduler should not run until all interrupt routines are complete to fulfill this scenario.

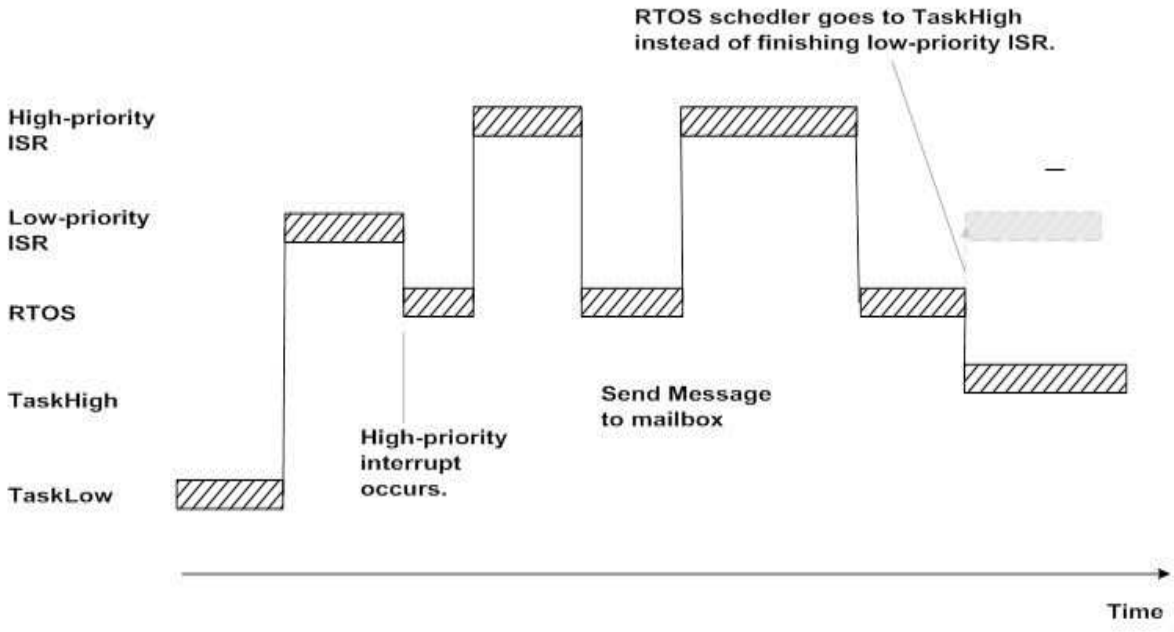


Figure: Nested Interrupt and the RTOS

11.0.0 Process

→ Process is an instance of a program on execution.
→ Requires various system resources such as CPU for execution process, memory for storing the code corresponding to the process and associated variables input output devices information exchange etc.
→ A program by itself is not process, a program is passive entity, such as file containing a list of instructions stored on the disk(executable file)
→ A process is an active entity. A program becomes a process when an executable file is loaded in to memory.

11.0.1 Process life cycle:

Process changes its state from newly created to execution completed with the following states.

1. Created state: a process is being created is referred. OS recognizes a process but no resources are allocated to the process.
2. Ready state: the state, where a process is incepted into the memory and awaiting the processor time for execution. Ready list contains the queue maintained by os.
3. Running state: the state where in source code instruction corresponding to process is being executed.
4. Blocked state/wait state: refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources.
5. Completed state: a state where process completes its execution.
6. State transition: the transition of a process from one state to another as shown below.

11.0.2 Process management:

- → Deals with the creation of a process
- → Setting up the memory space for the process
- → Loading the processor's code into the memory space
- → Allocating system resources
- → Setting up a process control block (PCB) for the process termination/deletion.

12.0.0 Threads:

- → Is the primitive that can execute code?
- → Is a single sequential flow of control within a process
- → Also known as light weight process
- → A process can have many thread of execution
- → Different threads which are part of a process share the same address space; meaning they share the data memory and the heap memory area.
- → Threads maintains their own thread state (CPU register values), program counter (PC) and stack.

12.0.1 Multithreading:

- → Application may complex and lengthy
- → Various sub-operation like getting input from I/O devices connected to the processor
- → Performing some internal calculations/operations
- → Updating some I/O devices.
- → All the sub-functions of a task are executed in sequence – the CPU utilization may not be efficient.

12.0.2 Advantages of Multiple threads to execute:

- → Better memory utilization (same process share the address space of the same memory and reduces complexity of inter thread communication.)
- → Speed up execution of the process(splitting into different threads when one threads enters a wait state, the CPU can utilized by the other threads of the process, that do not require the event, which other thread is waiting for processing.
- → Efficient CPU utilization, CPU-engaged all time.
- → Threads standards: deals with different standards available for thread creation and management utilized by OS.

Thread class libraries are:

1. POSIX Thread(portable operating system interface)
2. Win 32 threads
3. Java Threads.

12.0.3 Difference between threads and process

Threads	Process
1. Threads is a single unit of execution and is part of the process	A process is a program in execution and combines one or more threads
2. A thread shares the code data, heap with other threads of the same process	A process has its own code, data and stack memory.
3. A thread cannot live independently within a process	A process contains at least one threads
4. Threads are inexpensive	Process creation involves many overhead and are expensive to create
5. Context switching is inexpensive and fast	Context switching is expensive and slow
6. If a thread expires, its stack is reclaimed by the process	If a process expires the resources allocated to are reclaimed by the OS and the associat threads of the process also expires.

Embedded System Development Tool

Unit 6

6.0 Developing Software for Embedded system

Steps involved in preparing embedded software are similar to general programming. The following are basic steps involved in software design process.

1. Requirement gathering and analysis
2. Design
3. Implementation or coding
4. Testing
5. Deployment
6. Maintenance

However a spiral model is commonly used in embedded system development. The programming language is chosen as per the hardware supports. Most of the application programs are written in C language. However, higher programming language like C++ and java are also used for application programming. Knowing the capabilities and limitation of processor compiler is must for embedded system programming. The target platform may be unique which leads to additional software complexity. Thus software engineers must be aware of architecture of hardware before software build process.

Software development is performed in Host computer; they produce an executable binary image that will run on target embedded system.

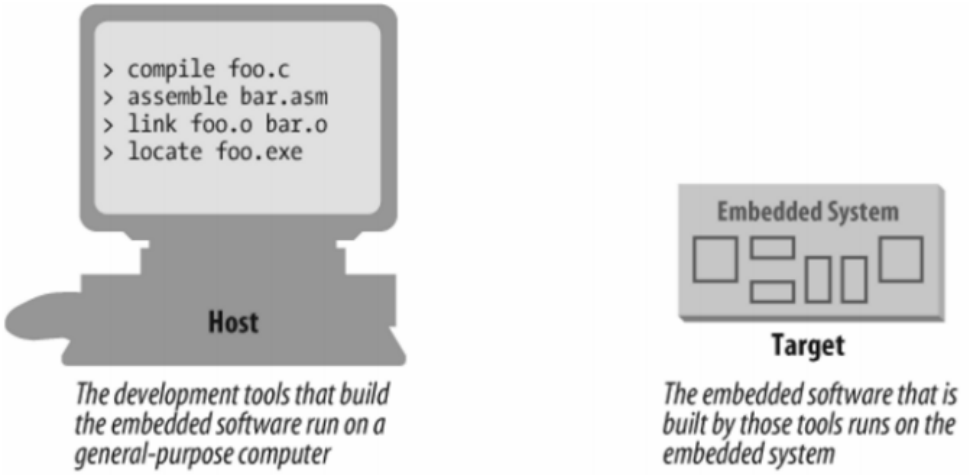


Figure: Host and Target platforms

Embedded system programming requires a more complex software build process. Target hardware platform consists of

- → Target hardware(processor, memory, I/O)
- → Runtime environment(Operating System/Kernel)

Thus target hardware platform contains only what is needed for final deployment. Such hardware does not contain development tool(editor, compiler and debugger). Target platform is completely different from development platform. On the other hand development platform

called Host Computer is typically a general purpose computer. Host computer runs compilers, assemblers, linkers, locator to create binary image that will run on target embedded system.

6.1 Steps to develop software for embedded system

- → Create source file(on Host)
- → Type in C code (on Host)
- → Compile/Assemble: translate into machine code(on Host)
- → Link : combine all object files and libraries, resolve all symbols'(on Host)
- → Locate: assign memory address to code and data(on Host)
- → Download : copy executable image into target processor memory
- → Execute: reset target processor

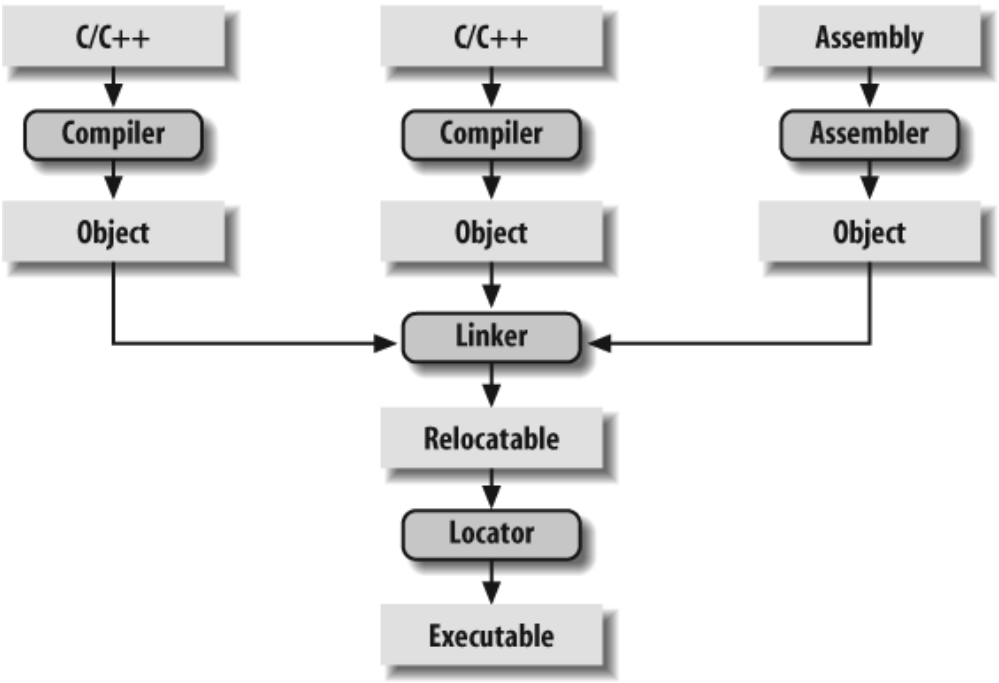


Figure: overall compilation phase

6.2 Compiling Embedded system

Compiler translate program written in human readable language into machine language. A compiler produces object code from the high level language(C/C++).

Source Code → Object file

Object file is a binary file that contains set of machine language instructions (opcode) and data resulting from language translation process. Machine language instructions are specific to a particular processor. Thus program compiled for one processor does not necessarily run for other processor.

There are two types of compiler

a. Native compiler
b. Cross compiler

Native Compiler

A native compiler runs on host plat form and produce code for that same computer platform.

Cross Compiler

A cross compiler runs on one computer platform, and produce codes for another computer platform.

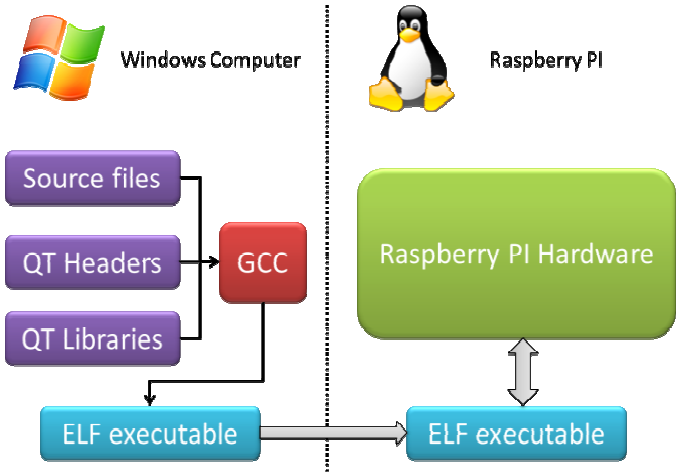


Figure: Cross Compilation

6.2.1 Assembler

Assembler performs one to one translation from human readable assembly language mnemonics to equivalent machine language opcode.

6.2.2 Interpreter

Interpreter constantly runs and interprets source code as set of directives. They perform syntax checking. However they are much slower than equivalent compiler.

6.2.3 Linker

The linker combines object files (from compiler) and resolves variable and function references. Some steps that linker performs are

- → Sources code may be contained in more than one file, which must be combined.
- → Resolve variables which may be referenced in one file and defined in another file
- → Resolve calls to library functions, like sqrt
- → May include operating system.

Doing all these steps linker creates a "relocatable" version of the program. So the program is complete, except no memory address is assigned.

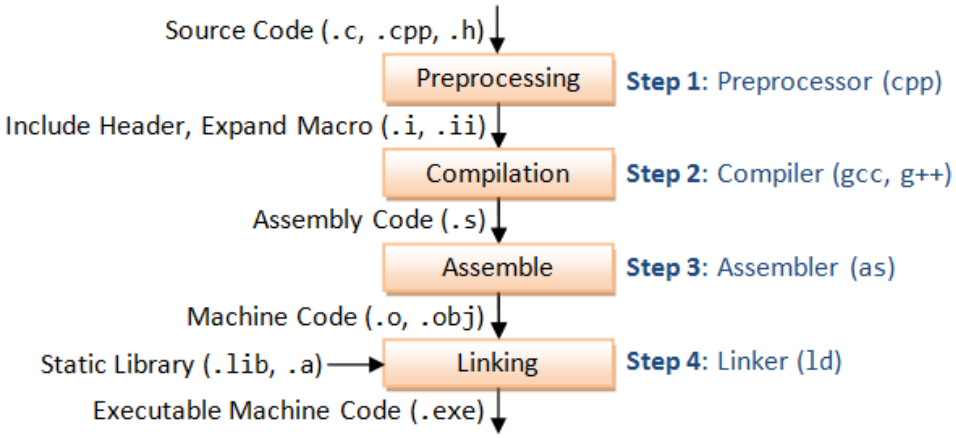


Figure: Internal details of compilation process

6.2.4 Locator

A locator is the tool that performs the conversion from relocatable program to executable binary image. The locator assigns physical memory address to code and data sections within the relocatable program. The locator produces the binary image that can be loaded inot the target ROM. In contrast, on general purpose computer the operating system assign the address at load time.

6.2.5 Downloader

Once a program has been successfully compiled, linked, and located, it must be moved to the target platform. Downloader downloads the binary image to the embedded system. This executable file can be loaded into ROM via a device programmer, which "burns" a chip that is then re-inserted into the embedded system or we have in-system downloader where binary image can be loaded into ROM without removing the IC. Once the program is successfully loaded the processor begin to execute its loaded instruction on power supply and once it is reset.

6.3 Debugging Embedded Software

Once the software has been downloaded to the target processor, we need to find if it is working properly or not. Basically we have two types of errors.

a. **Run time errors**
   When a program fails, usually the processor crash or lock-up. If we follow all the procedures as above we will not have run time errors. Such errors are usually warned during compilation and downloading process.

b. **Logical errors**
   Logical errors are really difficult to debug. It is really hard to find whether the program we are executing.
   Some of the commonly used Debuggers are
   1. Simulator

2. Remote Debugger
3. ICE(In Circuit Emulator)

1. Simulator

Simulator is host-based program that simulates functionality and instruction set of target processor. They usually have text or GUI-based windows for source code, register contents etc. They are valuable during early stages of development. They usually simulate the processor but not the peripherals.

2. Remote Debugger

Remote debugger is used to monitor/control embedded software. They are used to download, execute and debug embedded software over communication link(e.g. Serial Port). Usually remote debugger have front end and back end interfaces. Front end has text or GUI-Based window for source code, register contents etc. Backend provides low-level control of target processor, run on target processor and communicate to front-end over communication link. Debugger and software being debugged are executed on two different computer system. they support higher level of interaction between host and target. They allow start/restart/kill and stepping through program. They also have software breakpoint feature (stop execution if instruction X is fetched). We can also read and write individual register or data at specified address. The disadvantage is that the target processor should also run this software package other than the actual application program.

3. In-Circuit Emulators(ICE)

It takes the place of (i.e. Emulates) target processor. It contains copy of target processor, plus RAM, ROM and its own embedded software. It allows examining state of processor while program is running. It also provide Remote debugging interface. It support both hardware and software breakpoint (stop execution on memory and I/O read write, interrupt) "stops on write to variable num". It supports real time tracking of instruction being executed as well as their order of execution. They are usually very expensive.

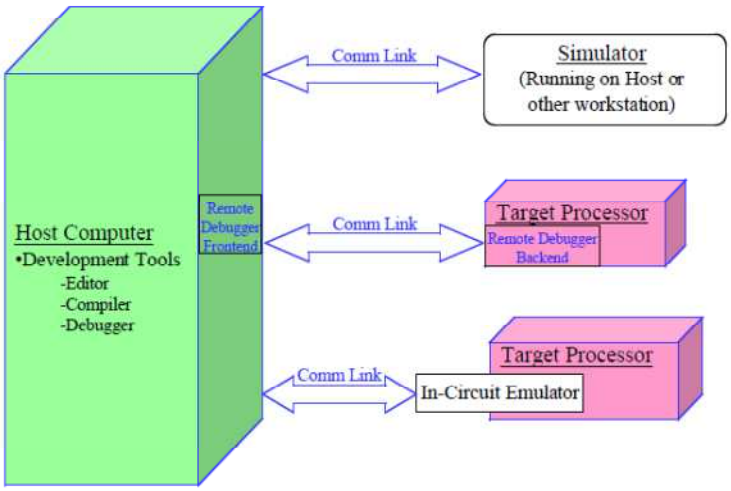


Figure: Debugging Tool

Microcontrollers and Embedded System

7.1 Microcontroller versus General-Purpose microprocessor.

	Microprocessor	Microcontroller
1	A silicon Chip representing CPU which is capable of performing arithmetic and logic operation as per defined set of instructions	Highly integrated chip containing cpu, scratch pad RAM, on chip ROM or Flash memory for programme storage, timers interrupt controller units and dedicated I/O units.
2	Dependent unit requires other chips as memory interrupt controller units etc	Dependent or self contained chip
3	General purpose in design and application	Mostly application specific or domain specific
4	No I/O parts to be implemented with help of PPI chips as 8255	No need such devices
5	Targeted for high end market where performance is important	Targeted for embedded market where performance only is not important
6	Limited power saving options	Lot of power saving options
7	Not limited in the context of data and program Memory	Limited in the context of data and program memory.

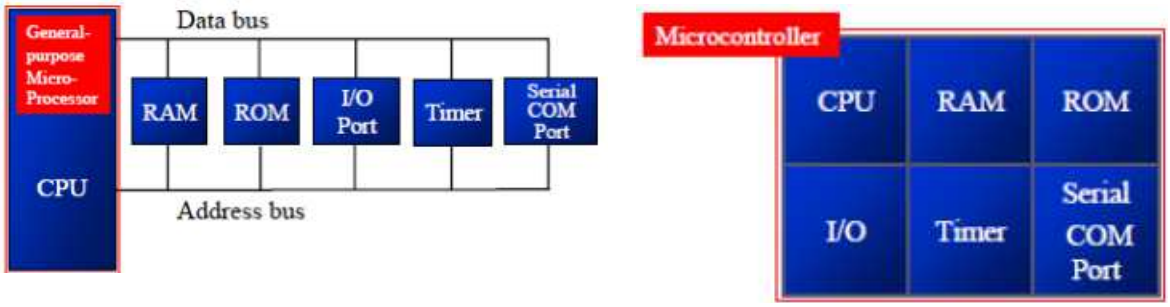


Figure: General Purpose Processor and Typical Microcontroller

7.2 Criteria for choosing a microcontroller

There are three major factors to be considered while choosing the microcontroller

1. Availability of software development tools
   - → How easy it is to develop product around it
   - → Key considerations-availability of assembler, debuggers, code efficient language compilers, emulators, technical supports and both in house and outside expertise.
2. Wide Availability and Reliable Source:
   - → Its ready availability in needed quantities both how and in the future.
   - → May be more important than other two criteria in some design.
3. Meeting the computing need of the task at hand efficient and cost effectively

→ Cost per unit –important in terms of final cost of products.

7.3 Features

The main features of 8051 microcontroller are

- RAM -128 Bytes(Data Memory)
- ROM -4Kbytes (ROM signify the on-chip program space)
- 8-bit data bus- it can access 8 bits of data in one operation.
- 16-bit data bus- It can access 216 memory locations for dual purpose -64KB(65536 location) each of RAM and ROM
- Serial Port –Using UART makes it simpler to interface for serial communication
- Two 16 bit Counter/Timer
- Input/output Pins -4 ports of 8 bits each on a single chips.
- 6 interrupt sources
- 8-bit ALU (Arithmetic Logic Unit), Accumulator and 8-bit Register; hence it is an 8-bit microcontroller.
- Harvard Memory and CISC Architecture
- 8051 can execute 1 millions one-cycle instructions per second with a clock frequency of 12MHz.
- 8051 consists of 16-bit program counter and data pointer
- 8051 also consists of 32 general purpose register each of 8 bits.

7.4 Block Diagram Of 8051

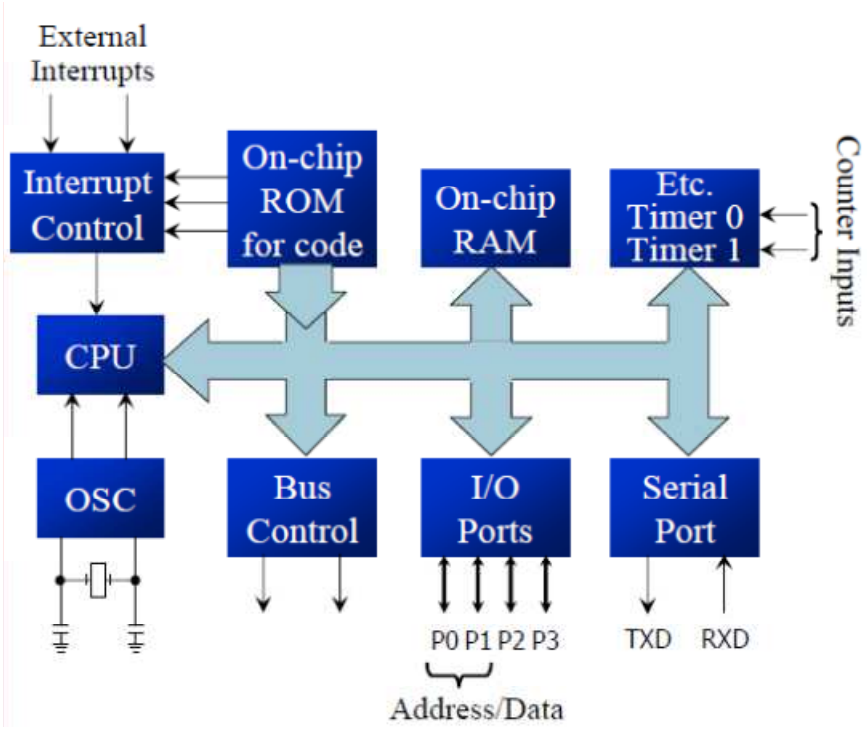
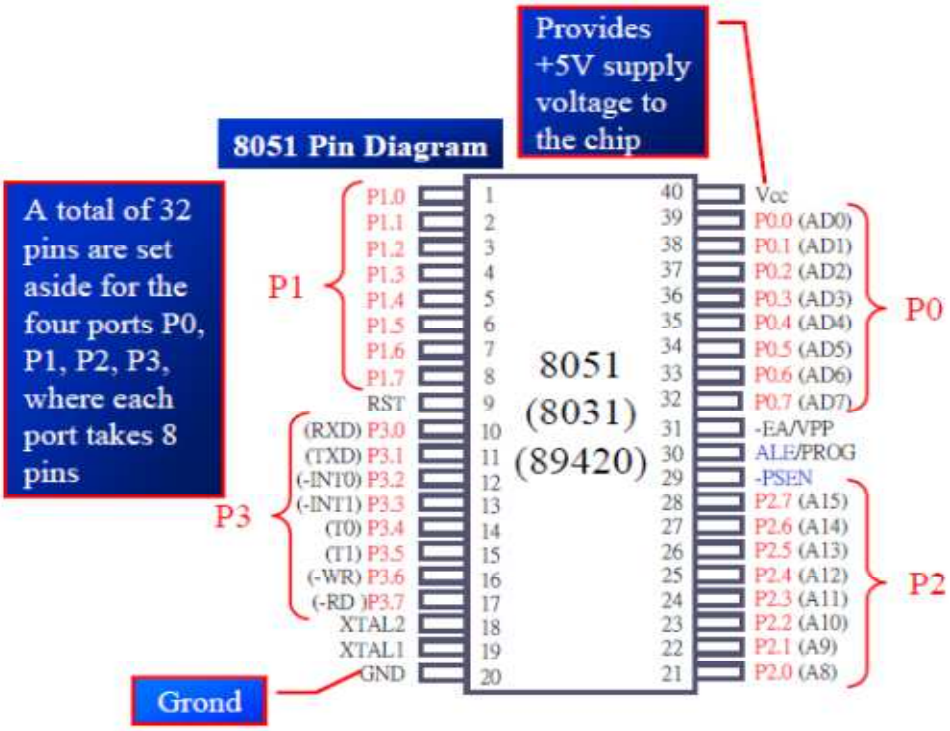


Figure: Block Diagram of 8051

7.5 8051 Pin Diagram



7.6 8051 Memory Organization

The 8051 has three basic memory address spaces

- 64K-bytes of Program Memory
- 64K-bytes of External Data Program Memory
- 256-bytes of Internal Data Memory

7.6.1 Data Memory Address Space Memory Organization of 8051

The data memory address space consists of an internal and an external memory space. Internal data memory is divided into the following three physically separate and distinct blocks.

- Lower 128 bytes of RAM
- Upper 128 bytes of RAM(accessible in the 8032/8052 only)
- 128-bytes of Special Function Register(SFR) area

Lower 128 bytes of RAM address range is 00H to 7FH. How manufacturer can divided these 128 bytes as given in below table:

RAM AREA NAME	ADDRESS RANGE	TOTAL(in bytes)
Register banks and the Stack	00H-1FH	32
Bit Addressable Area	20H-2FH	16
Scratch Pad Area	30H-7FH	80

The upper RAM area and the SFR(Special Function Register) area share the same address location. They are accessed through different addressing mode. Any location in the general purpose RAM can be accessed freely using the direct or indirect addressing modes.

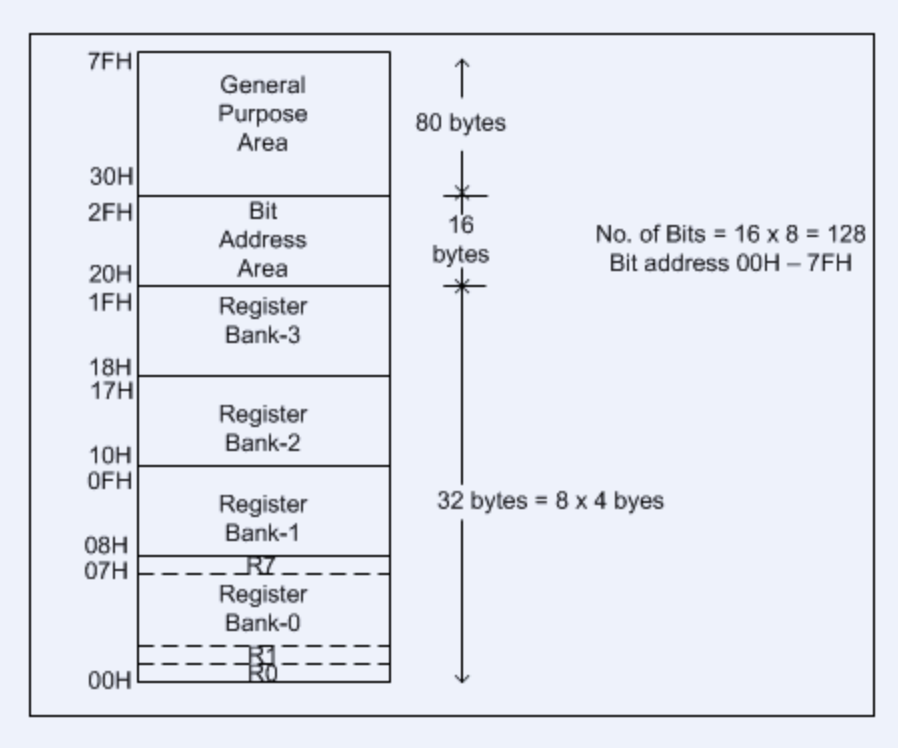


Figure: Internal RAM Structure

Register Banks in 8051

Manufacturer allocated 32 bytes for Register Banks. These 32 bytes are divided into 4 banks of registers in which each banks has 8 registers, named as R0-R7. Each register it takes 1 bytes. So, each bank occupies 8 bytes.

How can they give the address to threat register in each and every bank is shown below diagram.

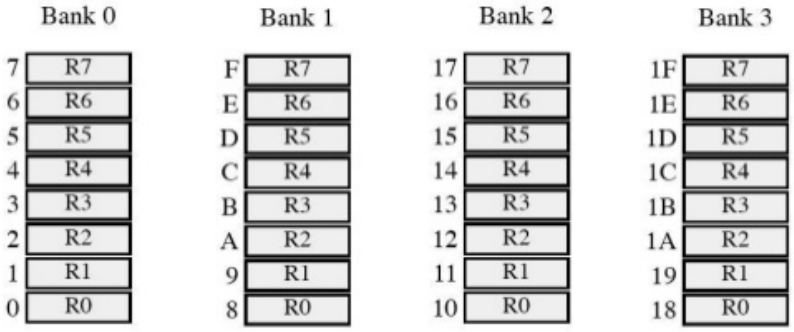


Figure: Register Bank and their location

Bank 1 uses the same ram space as the stack. Register bank 0 is the default when the 8051 is powered up. We can switch to other banks by use of the PSW (Program Status Word) register. PSW is one of the SFR (Special Function Register) and also this one is bit addressable register. So, we can access bit addressable instructions SETB and CLR. In this register PSW.3 and PSW.4 bits are used to select the desired register bank as shown in below table.

Bank	**RS1(PSW.4)**	**RS0(PSW.3)**
BANK 0	0	0
BANK 1	0	1
BANK 2	1	0
BANK 3	1	1

If we are using CLR PSW.x instruction then it makes zero value in that particular bit, and if we are using SETB PSW.x instruction then one(1) value passes in that particular bit. Where 'x' represents either 3 or 4.

7.7 Special Function Registers (SFRS) Memory Organization

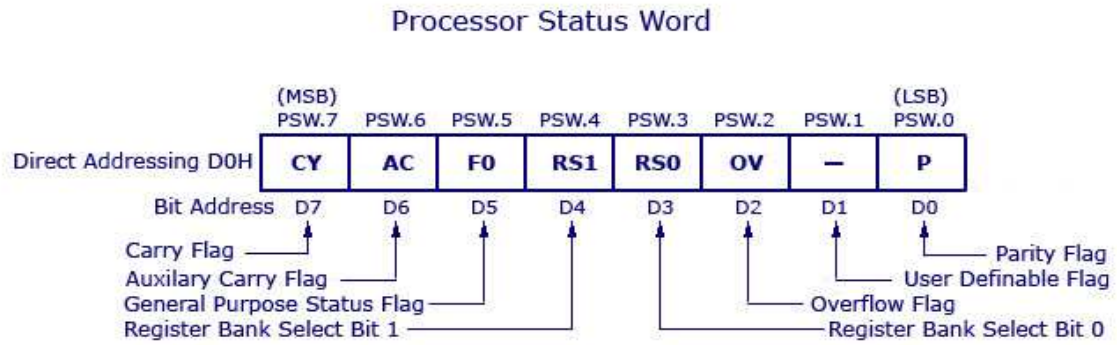
- Special Function Registers (SFRs) are areas of memory that control specific functionality of the 8051 processor. SFRs are accessed as if they were normal Internal RAM. The only difference is that internal RAM is from address 00H through 7FH whereas SFR registers exist in the address range of 80H through FFH. Each SFR has an address (80H though FFH) and a name.
- SFRs related to the I/O ports. The 8051 has four I/O ports of 8 bits, for a total of 32 I/O lines. Whether a given I/O line is high or low and the value read from the line are controlled by these SFRs.
- Total SFR memory is 128 bytes in that manufacturer allotted 21 bytes for 21 registers. Each and every register is used for some specific application. That is why these registers are called as Special Function Registers.
- In total 21 SFRs only 11 SFRs are Bit-addressable SFRs and these SFRs also byte addressable SFRs.
- SFRs which in some way control the operation or the configuration of some aspects of the 8051. For example, TCON controls the timers, SCON controls the serial port. The remaining SFRs are that they don't directly configure the 8051 but obviously the 8051 cannot operate without them.

F8									FF
F0	B								F7
E8									EF
E0	ACC								E7
D8									DF
D0	PSW								D7
C8									CF
C0									C7
B8	IP								BF
B0	P3								B7
A8	IE								AF
A0	P2								A7
98	SCON	SBUF							9F
90	P1								97
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
80	P0	SP	DPL	DPH				PCON	87

Bit-addressable Registers

Figure: SFR map

7.7.1 Program Status Word (PSW)



7.8 Instruction set of 8051

1. Data transfer instructions

a. MOV <dest-byte>,<src-byte>

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

1. mov direct , A
2. mov A, @R_i
3. mov A, R_n

4. mov direct, direct
5. mov A, #data

Example

MOV A,@R0	; moves the content of memory pointed to by R0 into A
MOV A, R_1	; moves the content of Register R_1 to Accumulator A
MOV 20h,30h	;moves the content of memory location 30h to 20h
MOV A,#45h	;moves 45h to Accumulator A

b. MOV <dest-bit>,<src-bit>

Function: Move bit data

Description: MOV <dest-bit>,<src-bit> copies the Boolean variable indicated by the second operand into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example

MOV P1.3, C ; moves the carry bit to 3rd bit of port1

c. MOV DPTR,#data16

Function: Load Data Pointer with a 16-bit constant

Description: MOV DPTR, #data16 loads the Data Pointer with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the lower-order byte. No flags are affected.

This is the only instruction which moves 16 bits of data at once.

Example:

MOV DPTR, # 4567H

This instruction loads the value 4567H into the Data Pointer. DPH holds 45H, and DPL holds 67H.

d. MOVC A,@A+ <base-reg>

Function: Move Code byte

Description:

The MOVC instructions load the Accumulator with a code byte or constant from program memory. The address of the byte fetched is the sum of the original unsigned 8-bit Accumulator contents and the contents of a 16-bit base register, which may be either the Data Pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the Accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may

propagate through higher-order bits. No flags are affected.

e. MOVC A,@A+PC

(PC)	←	(PC)+1
(A)	←	(A)+(PC)

f. MOVX <dest-byte>,<src-byte>

Description:

The MOVX instructions transfer data between the Accumulator and a byte of external data memory, which is why "X" is appended to MOV. There are two types of instructions, differing in whether they provide an 8-bit or 16-bit indirect address to the external data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an 8-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins are controlled by an output instruction preceding the MOVX

In the second type of MOVX instruction, the Data Pointer generates a 16-bit address. P2 outputs the high-order eight address bits (the contents of DPH), while P0 multiplexes the low-order eight bits (DPL) with data. The P2 Special Function Register retains its previous contents, while the P2 output buffers emit the contents of DPH.

This form of MOVX is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports.

It is possible to use both MOVX types in some situations. A large RAM array with its high-order address lines driven by P2 can be addressed via the Data Pointer, or with code to output high-order address bits to P2, followed by a MOVX instruction using R0 or R1.

Example:

An external 256 byte RAM using multiplexed address/data lines is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12H and

34H. Location 34H of the external RAM holds the value 56H. The instruction sequence, MOVX A,@R1

MOVX @R0, A

copies the value 56H into both the Accumulator and external RAM location 12H.

MOVX A,@DPTR

(A) ← ((DPTR))

PUSH direct

Function: Push onto stack

Description: The Stack Pointer is incremented by one. The contents of the indicated variable are then copied into the internal RAM location addressed by the Stack Pointer. No flags are affected.

Example: On entering an interrupt routine, the Stack Pointer contains 09H. The Data Pointer holds the value 0123H. The following instruction sequence,

PUSH DPL

PUSH DPH

leaves the Stack Pointer set to 0BH and stores 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

POP direct

Function: Pop from stack.

Description: The contents of the internal RAM location addressed by the Stack Pointer are read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

Example: The Stack Pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The following instruction sequence,

POP DPH

POP DPL

leaves the Stack Pointer equal to the value 30H and sets the Data Pointer to 0123H.

2. Arithmetic Group of Instructions

a. ADD A,<src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not bit 6; otherwise, OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example:

The Accumulator holds 0C3H (11000011B), and register 0 holds 0AAH (10101010B). The following instruction,

ADD A,R0

leaves 6DH (01101101B) in the Accumulator with the AC flag cleared and both the carry

flag and OV set to 1.

b. ADD A, direct

(A) ⟵ (A) + (direct)

c. ADD A, @Ri

(A) ⟵ (A) + data

d. ADDC A, <src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example

The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set.

ADDC A, R0

This instruction leaves 6EH (01101110B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

e. ADDC A, Rn

Operation: ADDC

(A) ⟵ (A) + (C) + (Rn)

f. ADDC A, direct

Operation: ADDC

(A) ⟵(A) + (C) + (direct)

g. ADDC A,@Ri

Operation: ADDC

(A) ⟵ (A)+(C)+((Ri))

h. ADDC A, #data

Operation: ADDC

(A) ⟵ (A)+(C)+#data

a. SUBB A,<src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7 and clears C otherwise. (If C was set *before* executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple-precision subtraction, so the carry is subtracted from the Accumulator along with the source operand.)

AC is set if a borrow is needed for bit 3 and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not bit 6.

When subtracting signed integers, OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example:

The Accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction,

SUBB A,R2

will leave the value 74H (01110100B) in the accumulator, with the carry flag and AC cleared but OV set.

b. SUBB A,Rn

Operation: SUBB

(A) ⟵ (A)-(C) - (Rn)

c. SUBB A, direct

Operation: SUBB

(A) ⟵(A)-(C)-(direct)

d. SUBB A,@Ri

Operation: SUBB

(A) ⟵ (A)-(C) - ((Ri))

a. SWAP A

Function: Swap nibbles within the Accumulator

Description: SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the Accumulator (bits 3 through 0 and bits 7 through 4). The operation can also be thought of as a 4-bit rotate instruction. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B).

SWAP A

The instruction leaves the Accumulator holding the value 5CH (01011100B)

Operation: SWAP

(A3-0) D (A7-4) // verify this statement

a. XCH A,<byte>

Function: Exchange Accumulator with byte variable

Description: XCH loads the Accumulator with the contents of the indicated variable, at the same time writing the original Accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

Example: R0 contains the address 20H. The Accumulator holds the value 3FH (0011111lB). Internal RAM location 20H holds the value 75H (01110101B).
The following instruction,
XCH A,@R0
leaves RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

1. XCHD A,@Ri

Function: Exchange Digit

Description: XCHD exchanges the low-order nibble of the Accumulator (bits 3 through 0), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register.

The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Example: R0 contains the address 20H. The Accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (01110101B).
The following instruction,
XCHD A,@R0
leaves RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the Accumulator.

CPL A

Function: Complement Accumulator

Description: CPLA logically complements each bit of the Accumulator (one's complement).

Bits which previously contained a 1 are changed to a 0 and vice-versa. No flags are affected.
Example: The Accumulator contains 5CH (01011100B). The following instruction,
CPL A
leaves the Accumulator set to 0A3H (10100011B).

CPL bit

Function: Complement bit
Description: CPL bit complements the bit variable specified. A bit that had been a 1 is changed to 0 and vice-versa. No other flags are affected. CLR can operate on the carry or any directly addressable bit.
Example: Port 1 has previously been written with 5BH (01011101B). The following instruction sequence, CPL P1.1
CPL P1.2 leaves the port set to 5BH (01011011B).

DAA

Function: Decimal-adjust Accumulator for Addition
Description: DA A adjusts the eight-bit value in the Accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition.
If Accumulator bits 3 through 0 are greater than nine or if the AC flag is one, six is added to the Accumulator producing the proper BCD digit in the low-order nibble. This internal addition sets the carry flag if a carry-out of the low-order four-bit field propagates through all high-order bits, but it does not clear the carry flag otherwise.
If the carry flag is now set, or if the four high-order bits now exceed nine, these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this sets the carry flag if there is a carry-out of the high-order bits, but does not clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal additions. OV is not affected.

DEC byte

Function: Decrement
Description: DEC byte decrements the variable indicated by 1. An original value of 00H underflows to 0FFH. No flags are affected.
Example: Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively.
The following instruction sequence,
DEC @R0
DEC R0
DEC @R0
leaves register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

DEC A
DEC Rn
DEC direct
DEC @Ri

DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B.

The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags are cleared.

Exception: if B had originally contained 00H, the values returned in the Accumulator and B-registers are undefined and the overflow flag are set. The carry flag is cleared in any case.

Example: The Accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The following instruction,

DIV AB

leaves 13 in the Accumulator (0DH or 00001101B) and the value 17 (11H or 00010001B) in B, since

251 = (13 x 18) + 17. Carry and OV are both cleared.

INC <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of 0FFH overflows to 00H. No flags are affected.

Example: Register 0 contains 7EH (011111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H,

respectively. The following instruction sequence,

INC @R0
INC R0
INC @R0

leaves register 0 set to 7FH and internal RAM locations 7EH and 7FH holding 00H and 41H, respectively.

INC A

Operation: INC

(A) ← (A) + 1

INC DPTR

Function: Increment Data Pointer

Description: INC DPTR increments the 16-bit data pointer by 1. A 16-bit increment

(modulo 216) is performed, and an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H increments the high-order byte (DPH).
No flags are affected.
This is the only 16-bit register which can be incremented.
Example: Registers DPH and DPL contain 12H and 0FEH, respectively. The following instruction sequence,

INC DPTR
INC DPTR
INC DPTR
changes DPH and DPL to 13H and 01H.

MUL AB

Function: Multiply
Description: MUL AB multiplies the unsigned 8-bit integers in the Accumulator and register B. The low-order byte of the 16-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH), the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.
Example: Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,
MUL AB
will give the product 12,800 (3200H), so B is changed to 32H (00110010B) and the Accumulator is cleared. The overflow flag is set, carry is cleared.

NOP

Function: No Operation
Description: Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

3. Logical instructions

ANL <dest-byte>,<src-byte>

Function: Logical-AND for byte variables
Description: ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected.
The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.
Example: If the Accumulator holds 0C3H (11000011B), and register 0 holds 55H (01010101B), then the following instruction,
ANL A, R0

leaves 41H (01000001B) in the Accumulator.

When the destination is a directly addressed byte, this instruction clears combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the Accumulator at run-time. The following instruction,

ANL P1, #01110011B

clears bits 7, 3, and 2 of output port 1.

ANL A, Rn
Operation: ANL
(A) ←(A) ^ (Rn)

ANL A,@Ri
Operation: ANL

(A) ← (A)^ ((Ri))

ANL direct,#data
Operation: ANL

(direct) (direct) ^ #data

ORL <dest-byte> <src-byte>

Function: Logical-OR for byte variables

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

Example: If the Accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101B) then the following instruction,

ORL A,R0

leaves the Accumulator holding the value 0D7H (1101011lB). The instruction,

ORL P1,#00110010B

sets bits 5, 4, and 1 of output Port 1.

ORL A, Rn ; or the content of Accumulator and Register Rn and store the result in Accumulator

ORL A, direct ; or the content of Accumulator and the memory and store the result in Accumulator

ORL A, @Ri ; or the content of accumulator and the memory location whose address is specified in Ri

ORL C,<src-bit>

Function: Logical-OR for bit variables

Description: Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. A slash (/) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Example:

ORL C, ACC.7 ;OR carry with the acc. bit 7

ORL C, /OV ;OR carry with the inverse of ov.

SETB

Operation: SETB

Function: Set Bit

Syntax: SETB *bit addr*

Description: Sets the specified bit.

XRL <dest-byte>,<src-byte>

Function: Logical Exclusive-OR for byte variables

Description: XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Example: If the Accumulator holds 0C3H (1100001lB) and register 0 holds 0AAH (10101010B) then the instruction,

XRL A,R0

leaves the Accumulator holding the value 69H (01101001B).

Rotate Instructions

RLA

Function: Rotate Accumulator Left

Description: The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The following instruction, RLA

leaves the Accumulator holding the value 8BH (10001011B) with the carry unaffected.

RLC A

Function: Rotate Accumulator Left through the Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.

Example: The Accumulator holds the value 0C5H(11000101B), and the carry is zero. The following instruction,
RLC A
leaves the Accumulator holding the value 8BH (10001010B) with the carry set.

RRC A

Function: Rotate Accumulator Right through Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B), the carry is zero. The following instruction,
RRC A leaves the Accumulator holding the value 62 (01100010B) with the carry set.

4. Branch instructions

4.1 Unconditional Branch Instructions

Operation: AJMP

Function: Absolute Jump Within 2K Block

Description: AJMP unconditionally jumps to the indicated *code address*. The new value for the Program Counter is calculated by replacing the least-significant-byte of the Program Counter with the second byte of the AJMP instruction, and replacing bits 0-2 of the most-significant-byte of the Program Counter with 3 bits that indicate the page *of the byte following the AJMP instruction*. Bits 3-7 of the most-significant-byte of the Program Counter remain unchanged.

Since only 11 bits of the Program Counter are affected by AJMP, jumps may only be made to code located within the same 2k block as the first byte that follows AJMP.

Operation: LJMP

Function: Long Jump

Syntax: LJMP *code address*.

Description: LJMP jumps unconditionally to the specified *code address*.

Operation: SJMP

Function: short Jump

Syntax: SJUMP reladdr

Description: SJMP jumps unconditionally to the address specified *reladdr*. *Reladdr* must be within -128 or +127 bytes of the instruction that follows the SJMP instruction

4.2 Conditional Branch Instructions

Operation: JNC

Function: Jump if Carry Not Set

Syntax: JNC *reladdr*

Description: JNC branches to the address indicated by *reladdr* if the carry bit is not set. If the carry bit is set program execution continues with the instruction following the JNB instruction.

Operation: JC

Function: Jump if Carry Set

Syntax: JC *reladdr*

Description: JC will branch to the address indicated by *reladdr* if the Carry Bit is set. If the Carry Bit is not set program execution continues with the instruction following the JC instruction.

Operation: JNB

Function: Jump if Bit Not Set

Syntax: JNB *bit addr*,*reladdr*

Description: JNB will branch to the address indicated by *reladdress* if the indicated bit is not set. If the bit is set program execution continues with the instruction following the JNB instruction.

Operation: JB

Function: Jump if Bit Set

Syntax: JB *bit addr*, *reladdr*

Description: JB branches to the address indicated by *reladdr* if the bit indicated by *bit addr* is set. If the bit is not set program execution continues with the instruction following the JB instruction.

Operation: JNZ

Function: Jump if Accumulator Not Zero

Syntax: JNZ *reladdr*

Description: JNZ will branch to the address indicated by *reladdr* if the Accumulator contains any value except 0. If the value of the Accumulator is zero program execution

continues with the instruction following the JNZ instruction.

Operation: JZ

Function: Jump if Accumulator Zero

Syntax: JNZ *reladdr*

Description: JZ branches to the address indicated by *reladdr* if the Accumulator contains the value 0. If the value of the Accumulator is non-zero program execution continues with the instruction following the JNZ instruction.

Operation: DJNZ

Function: Decrement and Jump if Not Zero

Syntax: DJNZ register, *reladdr*

Description: DJNZ decrements the value of *register* by 1. If the initial value of *register* is 0, decrementing the value will cause it to reset to 255 (0xFF Hex). If the new value of *register* is not 0 the program will branch to the address indicated by *relative addr*. If the new value of *register* is 0 program flow continues with the instruction following the DJNZ instruction.

Operation: CJNE

Function: compare and Jump If NOT Equal

Syntax: CJNE operand1,operand2,reladdr

Description: CJNE compares the value of *operand1* and *operand2* and branches to the indicated relative address if *operand1* and *operand2* are not equal. If the two operands are equal program flow continues with the instruction following the CJNE instruction. The **Carry bit (C)** is set if *operand1* is less than *operand2*, otherwise it is cleared.

8051 Example Programs

1. Write A Program to move a block of data within the internal RAM

```
        Org 0h
start1: MOV  R0, #40h      ; R0 pointed to internal RAM 40h
        MOV  R1, #30h      ; R1 pointing to internal RAM 030h
        MOV  R2, #5        ; R2 loaded with no. of elements in the array
Start:
        MOV A,@R0          ;data transfer
        MOV @R1,A
        INC R0
        INC R1
        DJNZ  R2,start     ;decrement  R2,if not equal to 0,continue with data
                           ;transfer process.
```

```
        SJMP  Start1
        end
```

2. Write a program to toggle led connected to port P0

```
BACK:   MOV   A,#55
        MOV   P0,A
        ACALL DELAY
        MOV   A, #0AAH
        MOV   P0,A
        ACALL DELAY
        SJUMP BACK
```

3. Write a program to configure Port 0 as input and receive data from that port and send the data to port P1

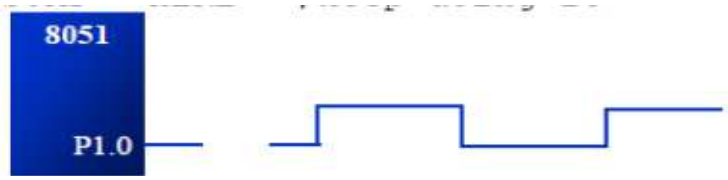
```
        MOV A, #FFH     ;A=FF hex
        MOV P0, A       ; makes P0 an i/p port by writing it all 1s
BACK :  MOV A,P0        ; get data from P0
        MOV P1,A        ;send it to port 1
        SJMP BACK       ;keep doing it
```

4. write a program to create a square wave of 50% duty cylcle on bit 0 of Port 1

Solution:

The 50% duty cycle means that the on and off state have same length.

```
HERE:   SETB P1.0       ;set to high bit 0 of port 1
        LCALL DELAY     ;call the delay subroutine
        CLR P1.0        ;P1.0=0
        LCALL DELAY
        SJMP HERE       ; keep doing it
```



5. Let us assume that bit P2.3 is an input and represent the condition of an oven. If it goes

high, it means that the oven is hot. Monitor the bit consentaneously. Whenever it goes, high send a high-to-low pulse to port P1.5 to turn on a buzzer.

Solution:

```
HERE:   JNB P2.3,HERE     ;keep monitoring for high
        SETB P1.5         ;set bit P1.5=1
        CLR P1.5          ;make high-to-low
        SJMP HERE         ;keep repeating
```

6. A switch is connected to pin P1.7. Write a program to check the status of SW and perform the following

   a. If SW=0, send letter "N" to P2

   b. If SW=1, send letter "Y" to P1

Solution:

```
        SETB P1.7         ;makes P1.7 an input
AGAIN:  JB P1.2,OVER      ;jump if P1.7=1
        MOV P2, #'N'      ;if SW=0, issue 'N' to P2
        SJMP AGAIN        ;keep monitoring
OVER:   MOV P2,#'Y'       ;SW=1, issue 'Y' to P2
        SJMP AGAIN        ;keep monitoring
```

7. Write a program to copy the value 55H into RAM memory locations 40H to 41H using

   a. Direct addressing mode

   b. Register indirect addressing mode without a loop, and

   c. With a loop

Solution:

(a)

```
        MOV A, #55H       ; load A with value 55H
        MOV 40H,A         ;copy A to RAM location 40H
        MOV 41H,A         ;copy A to RAM location 41H
```

(b)

```
        MOV A, #55H       ;load A with value 55H
        MOV RO, #40H      ;load the pointer R0=40H
        MOV @R0,A         ;copy A to RAM R0 points to
        INC R0            ;increment  pointer. Now R0 =41H
        MOV @R0,A         ;copy A to RAM R0 points to
```

(c)

```
        MOV A,#55H        ;A=55H
        MOV R0,#40H       ;load pointer ,R0=40H
```

```
                MOV R2,#02          ;load counter, R2=3
AGAIN:          MOV @R0,A           ;copy 55 to RAM RO points to
                INC R0              ;increment R0 pointer
                DJNZ R2,AGAIN       ;loop until counter=256
```

8. Write a program to copy a block of 10 bytes of data from 35H to 60H.

Solution:

```
                MOV R0, #35H        ;source pointer
                MOV R1, #60H        ;destination pointer
                MOV R3, #10H        ;counter
BACK:           MOV A, @R0          ;get a byte from source
                MOV @R1,A           ;copy it to destination
                INR R0              ;increment source pointer
                INC R1              ;increment destination pointer
                DJNZ R3,BACK        ;keep doing for ten bytes
```

9. Write a program to generate 1ms delay using 12MHz crystal frequency.

Solution:

```
                CSEG AT 0           ;start from 0th memory location
                MOV R7,#250         ;for MOV instruction it takes 1 machine cycle
L1:             DJNZ R7, L1         ;
                MOV R6, #249
L2:             DJNZ R6, 12
                END
```

10. Write an assembly language to interface seven segment display

Seven segment interfacing

```
        Org 0000h                   ; initialization location
Repeat:  MOV P0, #10000001b         ;displaying 0
        acall delay
        MOV P0 , #11001111b         ;displaying 1
        acall delay
        MOV P0, #10010010b          ;displaying 2
        acall delay
        MOV P0,# 10000110b          ;displaying 3
        acall delay
        MOV P0,# 11001100b          ;displaying 4
```

```
        acall delay
        MOV P0,# 10100100b          ;displaying 5
        acall delay
        MOV P0,# 10100000b          ;displaying 6
        acall delay
        MOV P0,# 10001111b          ;displaying 7
        acall delay
        MOV P0,# 10000000b          ;displaying 8
       acall delay
       MOV P0,# 10000100b           ;displaying 9
       acall delay
       SJMP Repeat
delay:
      MOV R3,#010H
 L3:  MOV R2,#0FFH
 L2:  MOV R1,#0FFH
 L1:  DJNZ R1,L1
      DJNZ R2,L2
      DJNZ R3,L3
```

11. Assume that the on-chip ROM has a message. Write a program to copy it from code space into the upper memory space starting at address 80H. Also as you place a byte in upper RAM give a copy to P0

```
            ORG 0
            MOV DPTR ,#MYDATA       ; access the upper 128 bytes on-chip RAM
            MOV R1, #80
B1:         CLR A
            MOVC A,@A+DPTR          ;copy from code ROM space
            MOV @R1,A               ;store in upper RAM space
            MOV P0,A                ; give a copy to P0
            JZ EXIT                 ; exit if last byte
            INC DPTR                ; increment DPTR
            INC R1                  ; increment R1
            SJMP B1                 ; repeat until the last byte
EXIT:       SJMP $                  ; stay here when finished
            ORG 300H
MYDATA:     DB " The promise of world peace",0
            END
```

VHDL

8.0 VHDL Overview

VHDL is an acronym for VHSIC Hardware Description language (VHSIC is an acronym for Very High Speed Integrated Circuits). It is a hardware description language that can be used to model a digital system at many levels of abstractions, ranging from the algorithmic level to the gate level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete digital electronic system, or anything in between. The digital system can also be described hierarchically.
Like any hardware description language, it is used for much purpose

- → For describing hardware
- → As a modeling language
- → For simulation of hardware
- → For early performance estimation of system architecture
- → For synthesis of hardware
- → For fault simulation, test and verification of design etc.

8.1 Levels of representation and abstractions:

- → This keeps description and design of complex manageable.

❖ **Behavioral :**

Highest level of abstraction that describes a system in terms of what it does rather than in terms of its components and interconnection between them. It describes relationship between input and output signal.

Let's consider a simple circuit that warns car passengers when the door isn't locked and the seatbelt is not used whenever a car key is inserted in ignition lock. At behavioral level, this could be expressed as

Warning: Ignitin_on AND(Door_open OR Seatbelt_off)

❖ Structural:

On the other hand, structural describes a system as a collection of gates and components that are interconnected to perform a desired function. It is usually closer to physical realization of system.

8.2 Basic Structure of a VHDL file

- → A digital system in VHDL consists of a design entity that can contain other entities that are then considered components of the top-level entity. Each entity is modeled by an entity declaration and an architecture body.
- → Entity declaration interfaces to outside world that defines input and output signals
- → Architecture body contains description of entity and is composed of interconnected entities, processes as shown in figure along with components and all operations concurrently.
- → VHDL uses reserved keywords which can't be used as signal names or identifiers.
- → Keywords and user-defined identifiers are case insensitive.
- → The comments starts with two adjacent hyphens (--) and will be ignored by the compiler.
- → VHDL also ignores line breaks and extra spaces.
- → VHDL is strongly typed language which implies that one has always to declared type of every object that can have a value, such as signals, constants and variables.

8.3 Design Elements in VHDL

a. ENTITY

- → An ENTITY represents a template for a hardware block
- → It describes just the outside view of a hardware model – namely its interface with other modules in terms of input and output signals.
- → The hardware block can be the entire design, a part of it or indeed an entire "test-bench"
- → A test bench includes the circuit being designed, blocks which apply test signals to it and those which monitor its output
- → The inner operation of the entity is described by an ARCHITECTURE associated with it.

Entity Declaration

The declaration of an ENTITY describes the signals which connect this hardware to the outside. These are called port signals. It also provides optional values of manifest constants. These are called generics.

General Syntax

```
entity NAME_OF_ENTITY
Port( signal_name: mode type;
Signal_name: mode type;
:
:
Signal_name: mode type);
End NAME_OF_ENTITY;
```

- → The NAME_OF_ENTITY is user selected identifier
- → Signal_names consists of comma separated list of one or more user-selected identifiers that specify external interface signals
- → An entity starts with keyword "entity" and end with keyword "end". Port declaration using keyword "port"
- → Mode: one of reserved words to indicate signal direction.
  - In: indicates signal is an input
  - Out: indicates signal is an output that can only be read by other entity that use it
  - Buffer: indicates signal is an output of entity whose value can be read inside entity architecture
  - Inout: signal can be input or output.

- → type: a built-in or user defined signal type
  - bit: can have value 0 or 1
  - bit_vector: vector of bit value 0 to 7
  - std_logic, std_ulogic,std_logic_vector,std_ulogic_vector: can have nine values to indicate value and strength of a signal
  - boolean: can have value TRUE and FALSE
  - integer: can have a range of integer values
  - real: can have a range of real values
  - time: to indicate time

b. ARCHITECTURE

ARCHITECTURE describes how an ENTITY operates. An ARCHTECTURE is always associated with an ENTITY. There can be multiple ARCHITECTURES associated with an ENTITY. ARCHITECTURE can describe an entity in a structural style, behavioral style or mixed style. The language provides constructs for describing components, their interconnects and composition (structural description). The language also includes signal assignment, sequential and concurrent statements for describing data and control flow, and for behavioral design.

Syntax:

```
    architecture architecture_name of NAME_OF_ENTITY is
    --Declarations
    --component declarations
    --signal declarations
    --constant declaration
--type declaration
begin
--statements
end architecture_name;
```

8.4 Behavior Model

- → Header of architecture body defines architecture name e.g. behavior and associates it with entity BUZZER
- → The architecture name can be any legal identifier
- → The main body starts with keyword “begin” and ends with keyword “end”.
- → The “<=” symbol is an assignment operator and assigns value of expression on right to signal on left
- → Complete program looks as follows

```
library ieee;
use ieee.std_logic_1164.all;

entity BUZZER is
        port( DOOR,IGNITION,SBELT:in std_logic;
        WARNING:out std_logic);
        end BUZZER;

architecture behavior of BUZZER is
begin
WARNING<=(not DOOR and IGNITION) or(not SBELT and IGNITION);
end behavioural;
```

8.5 Structural Description

- → The above program can also be described using a structural model that specifies what gates are used and how they are interconnected. For example

```
architecture structural of BUZZER is
-- Declaration
        Component AND2
                Port( in1,in2: in std_logic;
```

```
            Out1: out std_logic);
        end component;
        component OR2
            port( in1,in2: in std_logic;
            out1:out std_logic);
        end component;
        component NOT1
            port(in1: in std_logic;
            out1: out std_logic);
        end component;
    -- declaration of signals used to interconnect gates
    Signal DOOR_NOT, SBELT_NOT,B1,B2: std_logic;
    begin
        --  component instantiatins statements
    U0: NOT1 port map(DOOR->DOOR_NOT);
    U1:NOT1 port map(SBELT->SBELT_NOT);
    U2:AND2 port map(IGNITION,DOOR_NOT,B1);
    U3:AND2 port map(IGNITION,SBELT_NOT,B2);
    U4:OR2 port map(B1,B2,WARNING);
    end structural;
```

- → Following header is declarative part that gives gates which will be used in description of circuits.
- → The statements after begin keyword gives instantiations of components and describes how these are interconnected.

Label: component_name port map (port 1=> signal1, port2=> signal2, ……portn=>signal);

- → A library is a place where compiler stores information about a design project. A VHDL package is a file or module that contains declarations of commonly used objects data types, components declarations, signal, procedures and functions that can be shared among different VHDL models.
- → std_logic is defined in package ieee.std_logic_1164 in the ieee library. In order to use std.logic one need to specify library and package.
- → Its done at beginning of VHDL file using the library and use keywords as follows;
  - Library ieee;
  - Use ieee.std_logic_1164.all;
- → The .all extension indicates to use all of ieee.std_logic_1164 package.
- → The Xilinx foundation express comes with several packages.

8.6 VHDL Language Elements

1. Identifiers

- → They are user defined words used to name objects
- → While choosing identifiers one needs to follow these basic rules:
  - ❖ Many contains only alpha-numeric characters (A-Z,a-z,0-9) and underscore(_) character.
  - ❖ First character must be a letter and last one can't be an underscore
  - ❖ It can't include two consecutive underscores

- ❖ It is case 'insensitive'. (E.g. AND2 AnD2 or and2 refer to same object)
- ❖ An identifier can be of any length.

2. Data objects

→ A data object holds a value of a specified type. It is created by means of an object declaration. An example is
variable COUNT:INTEGER

→ This results in the creation of a data object called COUNT, which can hold values. The object COUNT is also declared to be of variable class

Every data object belongs to one of the following four classes:

i. Constant:

→ A constant can have a single value of a given type and cannot be changed during simulation. A constant is declared as follows:

constant list_of_name_of_constant: type [:=initial value]

→ Initial value is optional, constants can be declared at start of an architecture and can then be used anywhere within the architecture.

→ Constants declared within a process can only be used inside that specific process

constant RISE_ TIME:TIME:=2ns;
(It declares the object RISE_TIME which can hold a value of type TIME (a predefined type in the language) and the value assigned to the object at the start of simulation is 2ns)

constant DATA_BUS:INTEGER:=16;

(It declares a constant DATA_BUS of type INTEGER with a value of 8)

ii. Variable Declaration

A variable may be changed during program execution. Variable value is updated using a variable assignment statement. The variable is updated without any delay as soon as the statement is executed. Variable must be declared inside a process (and are local to process). The variable declaration is as follows:

Syntax

variable list_of_variable_name:type[:=initial value];

Some of the examples are:

variable GNIR_BIT:bit:=0;
variable VARS:boolean:=FALSE;
variable SUM: integer range 0 t0 256:=16;
variable STS_BIT:std_logic_vector(7 downto 0);

iii. Signal

Signals are similar to wires on a schematic and can be used to interconnect concurrent elements of design.

Syntax

signal list_of_signal_names: type[: initial value];

Examples:

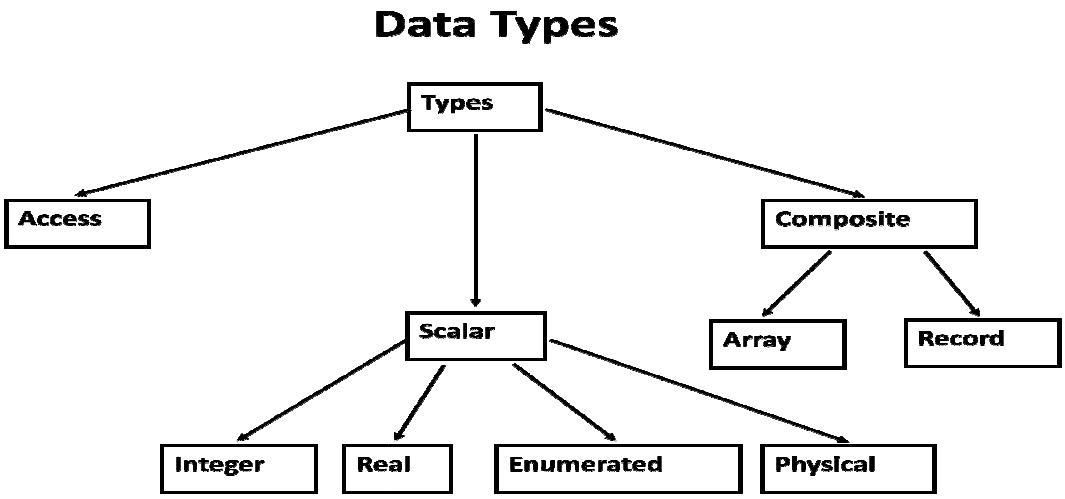
signal SUM, CARRY:std_logic;
signal CLOCK:bit;
signal TRIGGER:integer:=0;

signal DATA_BUS:std_logic_vector(7 **downto** 0);
signal VALUE:integer range 0 to 100;

Signals are updated when their signal assignment statements is executed, after a certain delay.

8.7 Data Types

Every data object in VHDL can hold a value that belongs to a set of values. This set of values is specified by using a type declaration. A type is a name that has associated with it a set of values and a set of operations.



a. Enumerated

An enumerated type declaration defines a type that has a set of user-defined value consisting of identifiers and character literals.

type MVL **is** ('U', '0' , '1', 'T');
type MICRO_OP **is** (LOAD, STORE, ADD, SUB, MUL, DIV);

b. Integer

An integer type defines a type whose set of values fall within a specified integer range.
Example

type INDEX **is range** 0 **to** 15;
type WORD_LENGTH **is range** 31 **downto** 0;

c. Real

A floating point type has a set of values in a given range of real numbers. Examples of floating point type declaration are

type TTL_VOLTAGE **is range** -5.5 **to** -1.4
type REAL_DATA **is range** 0.0 **to** 31.9

d. Physical

A physical type contains values that represents measurement of some physical quantity, like time, length, voltage or current.

```
type  current is range 0 to 10000000000000
    units
        nA;
        uA  = 1000nA;
    mA      = 1000uA;
```

end units;

8.8 Operators

The predefined operators in VHDL are classified into the following six catagories.

a. Logical operators
b. Relational operators
c. Shift operators
d. Adding operators
e. Multiplying operators
f. Miscellaneous operators

a. Logical Operators

→ The seven logical operators are **and, or, nand, nor, xor, xnor, not.** These operators are defined for the predefined types BIT and BOOLEAN.

b. Relational Operators

They are

i. =(equal)
ii. /=(not equal)
iii. < (less than)
iv. > (greater than)
v. <=(less than or equal to)
vi. >=(greater than or equal to)

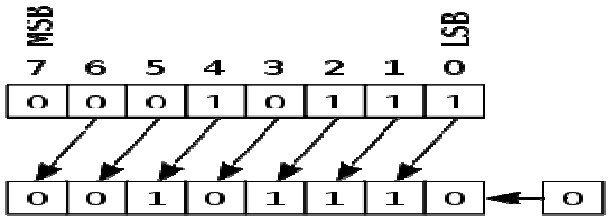
The result type for all relational operations is always the predefined type BOOLEAN.

c. Shift Operators

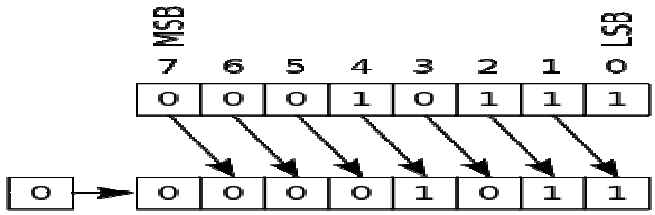
These are

- sll- shift left logical
- srl- shift right logical
- sla – shift left arithmetic
- sra – shift right arithmetic.
- rol- rotate left
- ror – rotate right

Sll- shift left logical

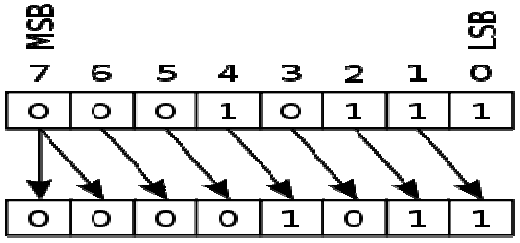


Srl- shift right logical



Sla is similar to that of sll.

Sra – shift right Arithmetic



ROL – rotate left

10010011 rol 1 = 00100111

ROR – rotate right

10010011 ror 1 = 11001001

d. Addition

These are + (addition)**, -** (subtraction), **&** (concatenation). The operands for addition and subtraction operators must be of the same numeric type with the result being of the same numeric type. The operands for the concatenation operator can be either a one-dimensional array type or an element type. The result is always an array type.

For example:

'0' & '1' results in an array of characters "01"

e. Multiply

These are * (multiplication) **/** (division), **rem** (remainder) and **mod**(modulus). Examples

7 **mod** 4 – has value 3

(-7) **rem** 4 – has value -3

f. Miscellaneous

The miscellaneous operators are **abs** (absolute), ****** (exponential)

8.9 IEEE library

- → std_logic_1164 package defines standard data types
- → std_logic_arithpackage: provides arithmetic conversin and comparision functions for signed, unsigned integer, std_ulogic, std_logic and std_logic_vector types
- → std_logic misc package: defines supplemental types, subtypes constants and functions for std_logic_vector types.
- → std_logic misc package: defines supplemental types, subtypes, constants and functions for std_logic_1164 package.
- → To use any of these one must includes library and use clause:

```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

9.0 Type Conversion

The VHDL is a strongly typed language one can't assign a value of one data type to a signal of a different data type.
Conversion function is supported by std_logic_1164 package is:

10.0 Sequential Statement:

1. **Process Statement**

A process is a sequential section of VHDL code. It's characterized by presence of IF, WAIT CASE, LOOP and a sensitivity list (except when WAIT is used). Process is executed every time a signal in sensitivity list changes (or condition related to WAIT is fulfilled). Its syntax is given below:

```
[process-label:] process[(sensitivity-list)] [is]
[process-item-declaration]
    begin
            sequential-statements; they are -->
            variable-assignment-statements
            signal-assignment-statements
            wait-statements
            if-statements
            case-statements
            loop-statements
            null-statements
            exit-statements
            assertion-statements
            report-statements
            procedure-call-statement
            return-statement
end process[process-label]
```

An example of a positive edge-triggered D flipflop

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity DFF_CLEAR is
Port(CLK, CLEAR,D: in std_logic;
Q:out std_logic);
end DFF_CLEAR;

Architecture BEHAV_DFF or DFF_CLEAR is
begin
        DFF_PROCESS: process(CLK,CLEAR)
begin
        if(CLEAR= '1') then
                Q<=D;
        elsif(CLK event and CLK= '1')then
                Q<=D;
```

```
        end if;
    end process;
    end BEHAV_DFF;
```

2. If Statement

The if statement executes a sequence of statements whose sequence depends on one or more conditions. The syntax is as follows:

```
if boolean-expression then
    sequential statements;
elseif boolean-expression then
    sequential statements;
else
    sequential statements;
end if;
```

Example

```
If S1 = '0' and S0 = '0' then
    Z<=A;
elseif condition then
    Sequential statement;
else
    Sequential statements;
end if;
```

3. Case Statement:

The case statement executes one of several sequences of statements, based on value of a single expression. The syntax is as follows:

```
case expression is
when choices => Sequential statements; -- branch #1
when choices => Sequential statements; -- branch #2
    :                    -- branches are allowed
    :                    -- branches are allowed

when others => sequential statements;
end case;
```

Example:

```
        When choices =>
            Sequential statement
        When choices=>
            Sequential statements
            -- branches are allowed
            When others => sequential statements;
        end case;

    library IEEE;
```

```
use IEEE.STD_LOGIC_1164.all;
    entity MUX is
    port( A, B, C, D : in STD_LOGIC;
    CTRL        : in STD_LOGIC_VECTOR(0 to 1);
    Z    : out STD_LOGIC);
    end MUX;

architecture MUX_BEHAVIOR of MUX is
    Constant MUX_DELAY:TIME:=10 ns;
begin
    PMUX: process( A, B, C, D, CTRL)
    Variable TEMP: STD_LOGIC;
    begin
Case CTRL is
    When "00" => TEMP:=A;
    When "01"=> TEMP:=B;
    When "10" => TEMP:=C;
    When "11"=>:=D;
    When others => TEMP:= 'X';
end case;
        Z<=TEMP after MUX_DELAY;
        end process PMUX;
end MUX_BEHAVIOR;
```

4. Loop Statements

The loop statement is used whenever an operation needs to be repeated. Loop statements are used when powerful iteration capability is needed to implement a model.
The syntax of a loop statement is

```
[loop-label:] iteration-scheme loop
        sequential-statement;
        end loop[loop-label];
```

There are three types of iterations schemes. The first is the for iteration scheme, which has the form
for identifier **in** range

Example:

```
    FACTORIAL:=1;
    for NUMBER in 2 to N loop
    FACTORIAL := FACTORIAL * NUMBER;
    end loop
```

The second form of the iteration scheme is the while scheme, which has the form
while boolean-expression
e.g.
j:=0; SUM:=10;

```
WH_LOOP: while j<20 loop  -- this loop has label, WH_LOOP
SUM:=SUM*2;
J:=J+3;
end loop;
```

The third form of the iteration scheme is one where no iterations scheme is specified. In this form of loop statement, all statements in the loop body are repeatedly executed until some other action causes the loop to terminate.

e.g

```
SUM:=1; J:=0;
L2:loop
J:=J+21;
SUM:=SUM*10;
exit when SUM>=100;
end loop L2;
```

5. Exit Statement

The exit statement is a sequential statement that can be used only inside a loop. It causes execution to jump out of the innermost loop or the loop whose label is specified.

Syntax

exit [loop-label][**when** condition];

Example

```
SUM:=1;J:=0;
L3:loop
        J:=J+21;
        SUM:=SUM*10;
If SUM>100 then
        exit L3;
end if;
end loop L3;
```

Next Statement

The next statement can be used only inside a loop. The syntax is

next[loop-label][when condition];

The next statement results in skipping the remaining statement in the current iteration of the specified loop;

Example

```
for J in 10 downto 5 loop
if SUM<TOTAL_SUM then
SUM:=SUM+2;
elsif SUM=TOTAL_SUM then
next;
else
null;
```

end if;
K:=K+1;
end loop;

Null Statement
The statement **null;** is a sequential statement that does not cause any action to take place; execution continues with the next statement.

Wait Statements
The wait statement gives the designer the ability to suspend the sequential execution of a processor or subprogram. The conditions for resuming execution of the suspended process or subprogram cab be specified by the following three different means.

- → **wait on** sensitivity-list;
- → **wait until** boolean-expression;
- → **wait for** time-expression

Design Examples

1. Driver

```
library ieee;
use ieee.std_logic_1164.all;

entity Driver is
      port( x     : in std_logic;
            F     : out std_logic
            );
end Driver;

 architecture behv2 of Driver is
       begin
       F <= x;
 end behv2;
```

2. OR gate

```
library ieee;
use ieee.std_logic_1164.all;

    entity OR_ent is
      port(   x   : in std_logic;
              y   : in std_logic;
              F   : out std_logic
            );
```

```
    end OR_ent;

    architecture OR_beh of OR_ent is
    begin
            F <= x or y;
            end OR_beh;
```

3. Multiplexer

```
    -- Multiplexor is a device to select different
    -- inputs to outputs. we use 3 bits vector to
    -- describe its I/O ports

    library ieee;
    use ieee.std_logic_1164.all;

entity Mux is
    port(   I3    :     in std_logic_vector(2 downto 0);
            I2    :     in std_logic_vector(2 downto 0);
            I1    :     in std_logic_vector(2 downto 0);
            I0    :     in std_logic_vector(2 downto 0);
             S    :  in std_logic_vector(1 downto 0);
            O     :  out std_logic_vector(2 downto 0)
        );
end Mux;

architecture behv1 of Mux is
begin
    process(I3,I2,I1,I0,S)
    begin

        -- use case statement
        case S is
        when "00" =>       O <= I0;
        when "01" =>    O <= I1;
        when "10" =>    O <= I2;
        when "11" =>    O <= I3;
        when others =>  O <= "ZZZ";
    end case;

    end process;
end behv1;
```

4. Decoder

```
    -- decoder is a kind of inverse process
    -- of multiplexor

    library ieee;
    use ieee.std_logic_1164.all;

entity DECODER is
```

```
    port(   I     :  in std_logic_vector(1 downto 0);
            O     :  out std_logic_vector(3 downto 0)
         );
end DECODER;

architecture behv of DECODER is
begin

    -- process statement

    process (I)
    begin

        -- use case statement

        case I is
        when "00" => O <= "0001";
        when "01" => O <= "0010";
        when "10" => O <= "0100";
        when "11" => O <= "1000";
        when others => O <= "XXXX";
    end case;
    end process;
end behv;
-- entity can have more than one architecture
architecture when_else of DECODER is
begin

    -- use when..else statement

     O <= "0001" when I = "00" else
        "0010" when I = "01" else
        "0100" when I = "10" else
        "1000" when I = "11" else
        "XXXX";
end when_else;
```

5. Latch

```
    -- latch is simply controlled by enable bit
    -- but has nothing to do with clock signal
    -- notice this difference from flip-flops

library ieee ;
use ieee.std_logic_1164.all;

entity D_latch is
    port(   data_in     :   in std_logic;
            enable      :   in std_logic;
            data_out    :   out std_logic
        );
end D_latch;
```

```
architecture behv of D_latch is
begin
     -- compare this to D flipflop
     process(data_in, enable)
     begin
          if (enable='1') then
               -- no clock signal here
          data_out <= data_in;
     end if;
     end process;
end behv;
```

6. Tri state driver

```
    -- VHDL model for tri state driver
    -- this driver often used to control system outputs

library ieee;
use ieee.std_logic_1164.all;

entity tristate_dr is
    port(   d_in     :   in std_logic_vector(7 downto 0);
            en       :   in std_logic;
            d_out    :   out std_logic_vector(7 downto 0)
        );
end tristate_dr;

architecture behavior of tristate_dr is
begin
  process(d_in, en)
  begin
    if en='1' then
      d_out <= d_in;
    else
      -- array can be created simply by using vector
      d_out <= "ZZZZZZZZ";
    end if;
  end process;

end behavior;
```

7. Counter

```
    -- this is the behavior description of n-bit counter
    -- another way can be used is FSM model.

library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
generic(n: natural :=2);
    port(clock    :  in std_logic;
```

```
          clear     :  in std_logic;
          count     :  in std_logic;
         Q          :  out std_logic_vector(n-1 downto 0)
          );
end counter;

architecture behv of counter is
    signal Pre_Q: std_logic_vector(n-1 downto 0);
begin
    -- behavior describe the counter
    process(clock, count, clear)
    begin
    if clear = '1' then
        Pre_Q <= Pre_Q - Pre_Q;
    elsif (clock='1' and clock'event) then
        if count = '1' then
        Pre_Q <= Pre_Q + 1;
        end if;
    end if;
    end process;

    -- concurrent assignment statement
    Q <= Pre_Q;
end behv;
```

8. ALU

```
    -- ALU stands for arithmatic logic unit.
    -- It perform multiple operations according to
    -- the control bits.
    -- we use 2's complement subraction in this example
    -- two 2-bit inputs & carry-bit ignored

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity ALU is
    port(A   :  in std_logic_vector(1 downto 0);
         B   :  in std_logic_vector(1 downto 0);
         Sel:    in std_logic_vector(1 downto 0);
        Res :    out std_logic_vector(1 downto 0)
        );
end ALU;

architecture behv of ALU is
begin
    process(A,B,Sel)
    begin

    -- use case statement to achieve
    -- different operations of ALU
```

```
    case Sel is
        when "00" =>
        Res <= A + B;
        when "01" =>
            Res <= A + (not B) + 1;
            when "10" =>
        Res <= A and B;
        when "11" =>
        Res <= A or B;
        when others =>
        Res <= "XX";
        end case;
    end process;
end behv;
```

9. N-Bit Adder

```
    -- function of adder:
    -- A plus B to get n-bit sum and 1 bit carry
    -- we may use generic statement to set the parameter
    -- n of the adder.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ADDER is
generic(n: natural :=2);
    port(   A    :    in std_logic_vector(n-1 downto 0);
            B    :    in std_logic_vector(n-1 downto 0);
        Carry    :    out std_logic;
        Sum      :    out std_logic_vector(n-1 downto 0)
         );

end ADDER;

architecture behv of ADDER is
-- define a temparary signal to store the result

signal result: std_logic_vector(n downto 0);
 begin

    -- the 3rd bit should be carry
    result <= ('0' & A)+('0' & B);
    sum <= result(n-1 downto 0);
    carry <= result(n);
end behv;
```

9. Multiplication

```
    -- Example of doing multiplication showing
    -- (1) how to use variable with in process
    -- (2) how to use for loop statement
```

```
    -- (3) algorithm of multiplication

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- two 4-bit inputs and one 8-bit outputs
entity multiplier is
    port(num1, num2  : in std_logic_vector(1 downto 0);
         product     :     out std_logic_vector(3 downto 0)
        );
end multiplier;

architecture behv of multiplier is
begin
process(num1, num2)

  variable num1_reg: std_logic_vector(2 downto 0);
  variable product_reg: std_logic_vector(5 downto 0);

begin

      num1_reg    := '0' & num1;
      product_reg := "0000" & num2;

  -- use variables doing computation
  -- algorithm is to repeat shifting/adding
  for i in 1 to 3 loop
    if product_reg(0)='1' then
      product_reg(5 downto 3) := product_reg(5 downto 3)
      + num1_reg(2 downto 0);
    end if;

    product_reg(5 downto 0) := '0' & product_reg(5 downto 1);
  end loop;

  -- assign the result of computation back to output signal
  product <= product_reg(3 downto 0);
end process;
end behv;
```

10. Register

```
    -- register
    library ieee ;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;

entity reg is
generic(n: natural :=2);
      port(I:  in std_logic_vector(n-1 downto 0);
      clock :  in std_logic;
      load  :  in std_logic;
```

```
      clear :  in std_logic;
      Q     :  out std_logic_vector(n-1 downto 0)
);
end reg;

architecture behv of reg is
    signal Q_tmp: std_logic_vector(n-1 downto 0);
begin
    process(I, clock, load, clear)
    begin
    if clear = '0' then
            -- use 'range in signal assigment
            Q_tmp <= (Q_tmp'range => '0');
    elsif (clock='1' and clock'event) then
        if load = '1' then
        Q_tmp <= I;
        end if;
    end if;
    end process;
    -- concurrent statement
    Q <= Q_tmp;
end behv;
```

11. FSM

```
    -- VHDL FSM (Finite State Machine) modeling
    -- FSM model consists of two concurrent processes
    -- state_reg and comb_logic
    -- we use case statement to describe the state
    -- transistion. All the inputs and signals are
    -- put into the process sensitive list.
    library ieee ;
    use ieee.std_logic_1164.all;

    entity seq_design is
    port(   a:      in std_logic;
        clock:      in std_logic;
        reset:      in std_logic;
        x:      out std_logic
    );
    end seq_design;
-------------------------------------------------
architecture FSM of seq_design is

     -- define the states of FSM model

    type state_type is (S0, S1, S2, S3);
    signal next_state, current_state: state_type;

begin

     -- cocurrent process#1: state registers
    state_reg: process(clock, reset)
    begin

    if (reset='1') then
            current_state <= S0;
    elsif (clock'event and clock='1') then
```

```
        current_state <= next_state;
    end if;
    end process;

    -- cocurrent process#2: combinational logic
    comb_logic: process(current_state, a)
    begin

    -- use case statement to show the
    -- state transistion

    case current_state is
        when S0 =>  x <= '0';
            if a='0' then
                next_state <= S0;
            elsif a ='1' then
                next_state <= S1;
            end if;
        when S1 =>  x <= '0';
            if a='0' then
                next_state <= S1;
            elsif a='1' then
                next_state <= S2;
            end if;
        when S2 =>  x <= '0';
            if a='0' then
                next_state <= S2;
            elsif a='1' then
                next_state <= S3;
            end if;

        when S3 =>  x <= '1';
            if a='0' then
                next_state <= S3;
            elsif a='1' then
                next_state <= S0;
            end if;

        when others =>
            x <= '0';
            next_state <= S0;

    end case;
    end process;
end FSM;
```

Design a 3 bit gray code synchronous Counter

Step 1: Transition state diagram

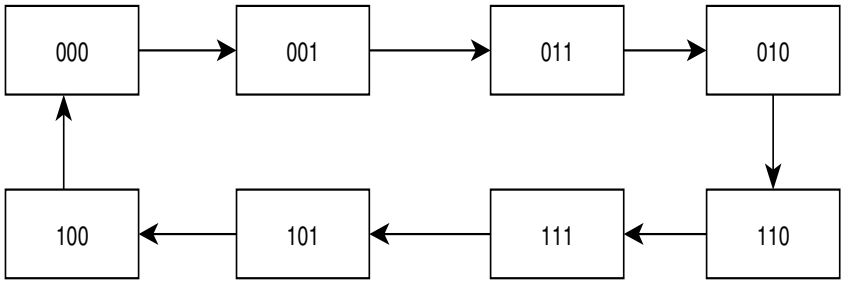


Figure: state transition diagram

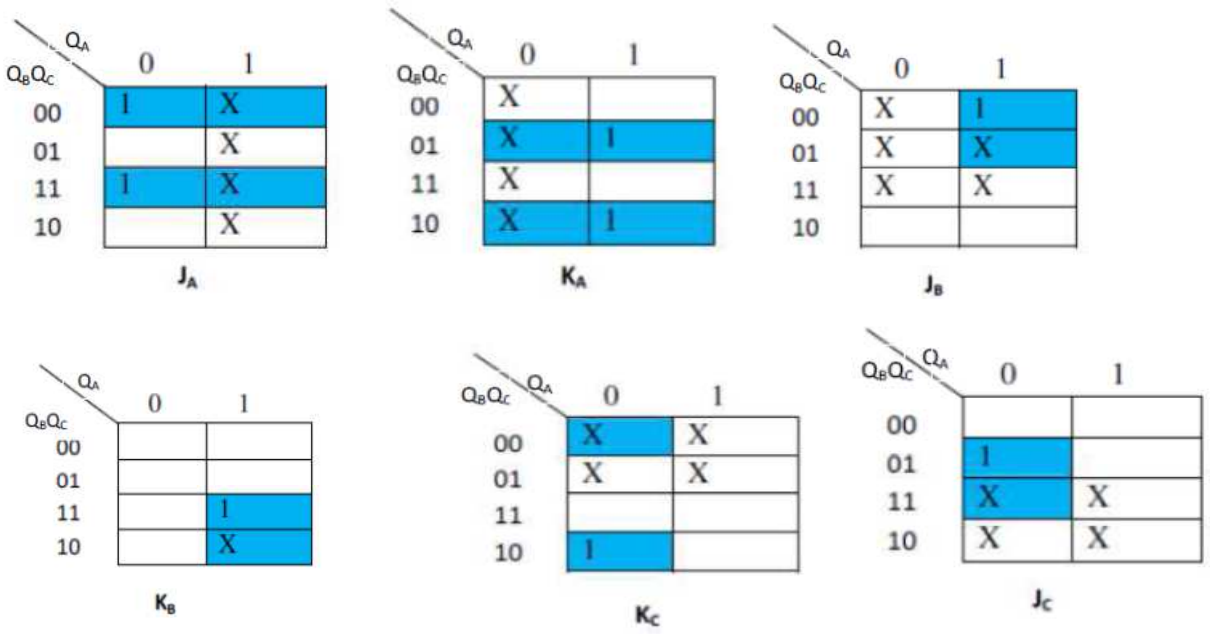
Step 2: Table showing present state and next state

Present State			Next State		
Q_C	Q_B	Q_A	Q_C	Q_B	Q_A
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	0	1	0
0	1	0	1	1	0
1	1	0	1	1	1
1	1	1	1	0	1
1	0	1	1	0	0
1	0	0	0	0	0

Step 3: Add flip-flop transition table

Present State			Next State			Flip-Flop					
						C		B		A	
Q_C	Q_B	Q_A	Q_C	Q_B	Q_A	J_C	K_C	J_B	K_B	J_A	K_A
0	0	0	0	0	1	0	×	0	×	1	×
0	0	1	0	1	1	0	×	1	×	×	0
0	1	1	0	1	0	0	×	×	0	×	1
0	1	0	1	1	0	1	×	×	0	0	×
1	1	0	1	1	1	×	0	×	0	1	×
1	1	1	1	0	1	×	0	×	1	×	0
1	0	1	1	0	0	×	0	0	×	×	1
1	0	0	0	0	0	×	1	0	×	0	×

Step 4: Karnaugh Map based on transition table



Step 5: Logic expression for Flip flop

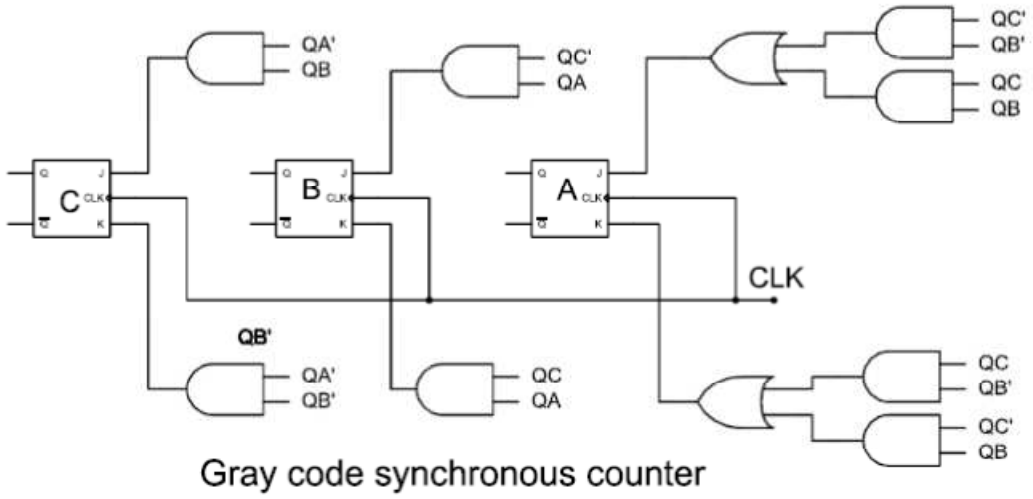
Boolean expression for J_A,K_A,J_B,K_B…. are

$J_A=Q_CQ_B+Q_CQ_B$ $K_A=Q_CQ_B+Q_CQ_B$

$J_B=Q_CQ_A$ $K_B=Q_CQ_A$

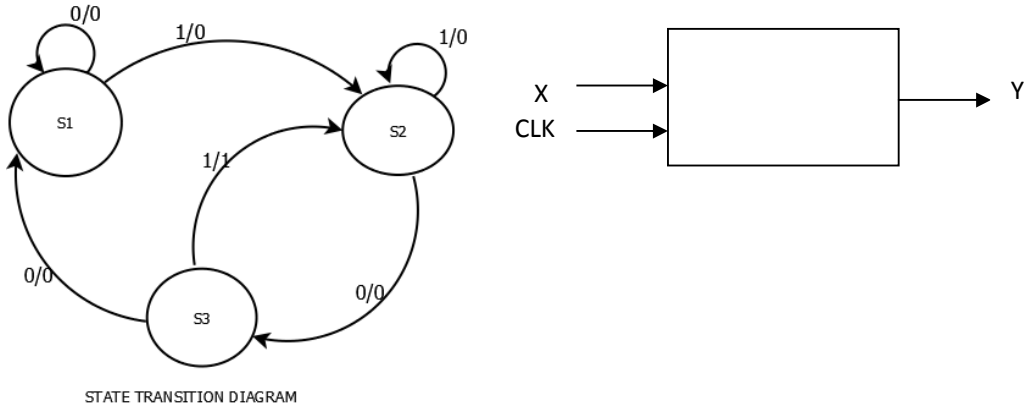
$J_C=Q_AQ_B$ $K_B=Q_AQ_B$

Step 6: Final Circuit implementation



Sequence Detector

1. Create a state diagram
   Let's design a sequence detector 101



STATE TRANSITION DIAGRAM

2. Create a state table

Present State	Next State		Output	
	X=0	X=1	X=0	X=1
S1	S1	S2	0	0
S2	S3	S2	0	0
S3	S1	S2	0	1

3. **State Assignment**
   There are 3 states and one flip-flop can store two bits. Therefore two flip-flop can be used for assigning 3 states as follows.

State	AB
S1	00
S2	01
S3	10

4. **State transition table using state assignment**
   Substitute state names S1, S2 and S3 with 00,01and 10

Present State	Next State(A^+ B^+)		Output	
	X=0	X=1	X=0	X=1
00	01	01	0	0
01	10	01	0	0
10	00	01	0	1

5. **Draw K-map from the state transition table**

X \ AB	00	01	11	10
0		1	X	
1			X	

$A^+=B\overline{X}$

X \ AB	00	01	11	10
0			X	
1	1	1	X	1

$B^+=X$

X \ AB	00	01	11	10
0			X	
1			X	1

Z=XA

6. **Find next state and output equation**

   From the k-map from step 5 the next equation and output equation
   Next equation is
   $A^+= B\overline{X}$
   $B^+=X$
   And output Z=XA

7. **Select Flip-flop and find out characteristics equation**
   Lets choose D-flip flop then the characteristics equation is
   Q+=D
   For JK Q(NEXT)=JQ'+K'Q

8. **Find flip flop input and output equation**
   Insert next state and output equation into the characteristics equation to find out the flip flop input and output equation
   From step 5 and 6
   $D_A=A^+=BX'$
   $D_B=B^+=X$
   Z=XA
9. **Draw circuit diagram**