Embedded System Design (BEC601) **June/July 2025**

VTU Question Paper Scheme & Solution

1. a) What is embedded system? List the applications of embedded system. (6M)
Answer: An embedded system is an electronic/electro-mechanical system designed to perform a specific function and is a combination of both hardware and firmware (software).

trackers etc.). The application areas and the products in the embedded domain are countless. A few of the important domains and products are listed below:

(1) *Consumer electronics:* Camcorders, cameras, etc.
(2) *Household appliances:* Television, DVD players, washing machine, fridge, microwave oven, etc.
(3) *Home automation and security systems:* Air conditioners, sprinklers, intruder detection alarms, closed circuit television cameras, fire alarms, etc.
(4) *Automotive industry:* Anti-lock breaking systems (ABS), engine control, ignition systems, automatic navigation systems, etc.
(5) *Telecom:* Cellular telephones, telephone switches, handset multimedia applications, etc.
(6) *Computer peripherals:* Printers, scanners, fax machines, etc.
(7) *Computer networking systems:* Network routers, switches, hubs, firewalls, etc.
(8) *Healthcare:* Different kinds of scanners, EEG, ECG machines etc.
(9) *Measurement & Instrumentation:* Digital multimeters, digital CROs, logic analysers PLC systems, etc.
(10) *Banking & Retail:* Automatic teller machines (ATM) and currency counters, point of sales (POS)
(11) *Card Readers:* Barcode, smart card readers, hand held devices, etc.
(12) *Wearable Devices:* Health and Fitness Trackers, Smartphone Screen extension for notifications, etc.
(13) Cloud Computing and Internet of Things (IOT)

b) Give the difference between microcontroller and Microprocessor. (6M)

Answer:

Microprocessor	Microcontroller
It is a dependent unit. It requires the combination of other chips like timers, program and data memory chips, interrupt controllers, etc. for functioning	It is a self-contained unit and it doesn't require external interrupt controller, timer, UART, etc. for its functioning
Most of the time general purpose in design and operation	Mostly application-oriented or domain-specific
Doesn't contain a built in I/O port. The I/O port functionality needs to be implemented with the help of external programmable peripheral interface chips like 8255	Most of the processors contain multiple built-in I/O ports which can be operated as a single 8 or 16 or 32 bit port or as individual port pins
Targeted for high end market where performance is important	Targeted for embedded market where performance is not so critical (At present this demarcation is invalid)
Limited power saving options compared to microcontrollers	Includes lot of power saving features

c) Explain about opto coupler and push button switch with neat diagram. (8M)

2.3.3.3 Optocoupler Optocoupler is a solid state device to isolate two parts of a circuit. Optocoupler combines an LED and a photo-transistor in a single housing (package). Figure 2.16 illustrates the functioning of an optocoupler device.

In electronic circuits, an optocoupler is used for suppressing interference in data communication, circuit isolation, high voltage separation, simultaneous separation and signal intensification, etc. Optocouplers can be used in either input circuits or in output circuits. Figure 2.17 illustrates the usage of optocoupler in input circuit and output circuit of an embedded system with a microcontroller as the system core.

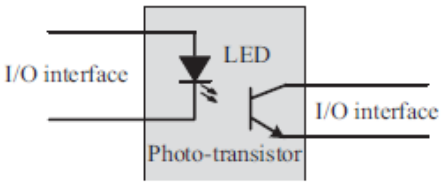


Fig. 2.16 An optocoupler device

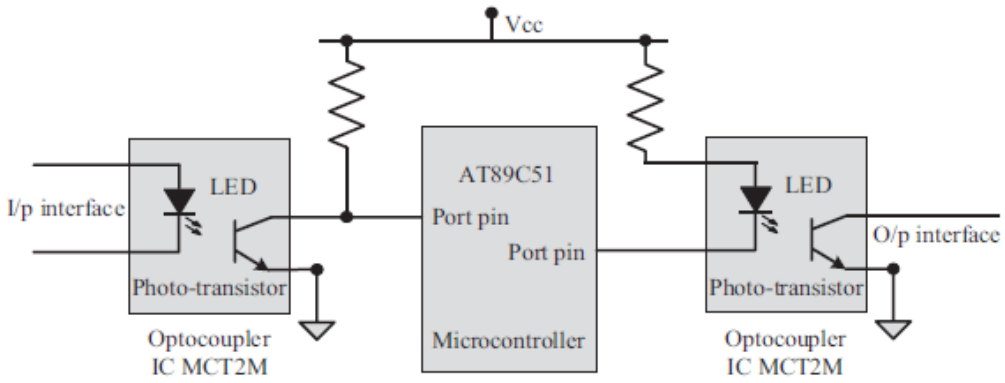


Fig. 2.17 Optocoupler in Input and Output circuit

2.3.3.7 Push Button Switch It is an input device. Push button switch comes in two configurations, namely 'Push to Make' and 'Push to Break'. In the 'Push to Make' configuration, the switch is normally in the open state and it makes a circuit contact when it is pushed or pressed. In the 'Push to Break' configuration, the switch is normally in the closed state and it breaks the circuit contact when it is pushed or pressed. The push button stays in the 'closed' (For Push to Make type) or 'open' (For Push to Break type) state as long as it is kept in the pushed state and it breaks/makes the circuit connection when it is released. Push button is used for generating a momentary pulse. In embedded application push button is generally used as reset and start switch and pulse generator. The Push button is normally connected to the port pin of the host processor/ controller. Depending on the way in which the push button interfaced to the controller, it can generate either a 'HIGH' pulse or a 'LOW' pulse. Figure 2.23 illustrates how the push button can be used for generating 'LOW' and 'HIGH' pulses.

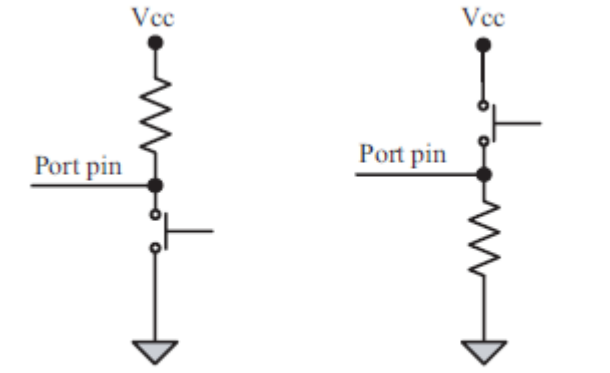


Fig. 2.23 Push button switch configurations

2 a) Give the classification of Embedded System with examples.

1.4 CLASSIFICATION OF EMBEDDED SYSTEMS

LO 4 Classify embedded systems based on performance, complexity and the era in which they evolved

It is possible to have a multitude of classifications for embedded systems, based on different criteria. Some of the criteria used in the classification of embedded systems are as follows:

(1) Based on generation
(2) Complexity and performance requirements
(3) Based on deterministic behaviour
(4) Based on triggering.

The classification based on deterministic system behaviour is applicable for 'Real Time' systems. The application/task execution behaviour for an embedded system can be either deterministic or non- deterministic. Based on the execution behaviour, Real Time embedded systems are classified into *Hard* and *Soft*. We will discuss about hard and soft real time systems in a later chapter. Embedded Systems which are 'Reactive' in nature (Like process control systems in industrial control applications) can be classified based on the trigger. Reactive systems can be either *event triggered* or *time triggered*.

1.4.1 Classification Based on Generation

This classification is based on the order in which the embedded processing systems evolved from the first version to where they are today. As per this criterion, embedded systems can be classified into the following:

1.4.1.1 First Generation The early embedded systems were built around 8bit microprocessors like 8085 and Z80, and 4bit microcontrollers. Simple in hardware circuits with firmware developed in Assembly code. Digital telephone keypads, stepper motor control units etc. are examples of this.

1.4.1.2 Second Generation These are embedded systems built around 16bit microprocessors and 8 or 16 bit microcontrollers, following the first generation embedded systems. The instruction set for the second generation processors/controllers were much more complex and powerful than the first generation processors/ controllers. Some of the second generation embedded systems contained embedded operating systems for their operation. Data Acquisition Systems, SCADA systems, etc. are examples of second generation embedded systems.

1.4.1.3 Third Generation With advances in processor technology, embedded system developers started making use of powerful 32bit processors and 16bit microcontrollers for their design. A new concept of

application and domain specific processors/controllers like Digital Signal Processors (DSP) and Application Specific Integrated Circuits (ASICs) came into the picture. The instruction set of processors became more complex and powerful and the concept of instruction pipelining also evolved. The processor market was flooded with different types of processors from different vendors. Processors like Intel Pentium, Motorola 68K, etc. gained attention in high performance embedded requirements. Dedicated embedded real time and general purpose operating systems entered into the embedded market. Embedded systems spread its ground to areas like robotics, media, industrial process control, networking, etc.

1.4.1.4 Fourth Generation The advent of System on Chips (SoC), reconfigurable processors and multicore processors are bringing high performance, tight integration and miniaturisation into the embedded device market. The SoC technique implements a total system on a chip by integrating different functionalities with a processor core on an integrated circuit. We will discuss about SoCs in a later chapter. The fourth generation embedded systems are making use of high performance real time embedded operating systems for their functioning. Smart phone devices, mobile internet devices (MIDs), etc. are examples of fourth generation embedded systems.

1.4.1.5 What Next? The processor and embedded market is highly dynamic and demanding. So 'what will be the next smart move in the next embedded generation?' Let's wait and see.

1.4.2 Classification Based on Complexity and Performance

This classification is based on the complexity and system performance requirements. According to this classification, embedded systems can be grouped into the following:

1.4.2.1 Small-Scale Embedded Systems Embedded systems which are simple in application needs and where the performance requirements are not time critical fall under this category. An electronic toy is a typical example of a small-scale embedded system. Small-scale embedded systems are usually built around low performance and low cost 8 or 16 bit microprocessors/microcontrollers. A small-scale embedded system may or may not contain an operating system for its functioning.

1.4.2.2 Medium-Scale Embedded Systems Embedded systems which are slightly complex in hardware and firmware (software) requirements fall under this category. Medium-scale embedded systems are usually built around medium performance, low cost 16 or 32 bit microprocessors/microcontrollers or digital signal processors. They usually contain an embedded operating system (either general purpose or real time operating system) for functioning.

1.4.2.3 Large-Scale Embedded Systems/Complex Systems Embedded systems which involve highly complex hardware and firmware requirements fall under this category. They are employed in mission critical applications demanding high performance. Such systems are commonly built around high performance 32 or 64 bit RISC processors/controllers or Reconfigurable System on Chip (RSoC) or multi-core processors and programmable logic devices. They may contain multiple processors/controllers and co-units/hardware accelerators for offloading the processing requirements from the main processor of the system. Decoding/encoding of media, cryptographic function implementation, etc. are examples for processing requirements which can be implemented using a co-processor/hardware accelerator. Complex embedded systems usually contain a high performance Real Time Operating System (RTOS) for task scheduling, prioritisation, and management.

2b) Give the difference between Von-Neumann and Harvard Architecture. (6M)

The following table highlights the differences between Harvard and Von-Neumann architecture.

Harvard Architecture	Von-Neumann Architecture
Separate buses for instruction and data fetching	Single shared bus for instruction and data fetching
Easier to pipeline, so high performance can be achieved	Low performance compared to Harvard architecture
Comparatively high cost	Cheaper
No memory alignment problems	Allows self modifying codes*
Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory	Since data memory and program memory are stored physically in the same chip, chances for accidental corruption of program memory

2c) Explain Piezo buzzer, sensor and actuators in embedded system with neat diagram. (8M)

2.3 SENSORS AND ACTUATORS

LO 3 Analyse the role of sensors, actuators, and their interfacing with the I/O subsystems of an embedded system

At the very beginning of this chapter it is already mentioned that an embedded system is in constant interaction with the Real world and the controlling/ monitoring functions executed by the embedded system is achieved in accordance with the changes happening to the Real world. The changes in system environment or variables are detected by the sensors connected to the input port of the embedded system. If the embedded system is designed for any controlling purpose, the system will produce some changes in the controlling variable to bring the controlled variable to the desired value. It is achieved through an actuator connected to the output port of the embedded system. If the embedded system is designed for monitoring purpose only, then there is no need for including an actuator in the system. For example, take the case of an ECG machine. It is designed to monitor the heart beat status of a patient and it cannot impose a control over the patient's heart beat and its order. The sensors used here are the different electrode sets connected to the body of the patient. The variations are captured and presented to the user (may be a doctor) through a visual display or some printed chart.

2.3.1 Sensors

A sensor is a transducer device that converts energy from one form to another for any measurement or control purpose. This is what I "by-hearted" during my engineering degree from the transducers paper.

Looking back to the 'Wearable devices' example given at the end of Chapter 1, we can identify that the sensor which counts steps for pedometer functionality is an Accelerometer sensor and the sensor used in some of the smartwatch devices to measure the light intensity is an Ambient Light Sensor (ALS)

2.3.2 Actuators

Actuator is a form of transducer device (mechanical or electrical) which converts signals to corresponding physical action (motion). Actuator acts as an output device.

Looking back to the 'Wearable devices' example given at the end of Chapter 1, we can see that certain smartwatches use Ambient Light Sensor to detect the surrounding light intensity and uses an electrical/ electronic actuator circuit to adjust the screen brightness for better readability.

2.3.3.6 Piezo Buzzer Piezo buzzer is a piezoelectric device for generating audio indications in embedded application. A piezoelectric buzzer contains a piezoelectric diaphragm which produces audible sound in response to the voltage applied to it. Piezoelectric buzzers are available in two types. 'Self-driving' and 'External driving'. The 'Self-driving' circuit contains all the necessary components to generate sound at a predefined tone. It will generate a tone on applying the voltage. External driving piezo buzzers supports the generation of different tones. The tone can be varied by applying a variable pulse train to the piezoelectric buzzer. A piezo buzzer can be directly interfaced to the port pin of the processor/control. Depending on the driving current requirements, the piezo buzzer can also be interfaced using a transistor based driver circuit as in the case of a 'Relay'.

3a) Explain the characteristics and quality attributes of Embedded System. (6M)

3.1 CHARACTERISTICS OF AN EMBEDDED SYSTEM

LO 1 Understand the characteristics describing an embedded system

Unlike general purpose computing systems, embedded systems possess certain specific characteristics and these characteristics are unique to each embedded system. Some of the important characteristics of an embedded system are as follows:

(1) Application and domain specific
(2) Reactive and Real Time
(3) Operates in harsh environments

(4) Distributed
(5) Small size and weight
(6) Power concerns

3.1.1 Application and Domain Specific

If you closely observe any embedded system, you will find that each embedded system is designed to perform a set of defined functions and they are developed in such a manner to do the intended functions only. They cannot be used for any other purpose. It is the major criterion which distinguishes an embedded system from a general purpose computing system. For example, you cannot replace the embedded control unit of your microwave oven with your air conditioner's embedded control unit, because the embedded control units of microwave oven and airconditioner are specifically designed to perform certain specific tasks. Also, you cannot replace an embedded control unit developed for a particular domain say telecom with another control unit designed to serve another domain like consumer electronics.

3.1.2 Reactive and Real Time

As mentioned earlier, embedded systems are in constant interaction with the Real world through sensors and user-defined input devices which are connected to the input port of the system. Any changes happening in the Real world (which is called an Event) are captured by the sensors or input devices in Real Time and the control algorithm running inside the unit reacts in a designed manner to bring the controlled output variables to the desired level. The event may be a periodic one or an unpredicted one. If the event is an unpredicted one then such systems should be designed in such a way that it should be scheduled to capture the events without missing them. Embedded systems produce changes in output in response to the changes in the input. So they are generally referred as Reactive Systems.

Real Time System operation means the timing behaviour of the system should be deterministic; meaning the system should respond to requests or tasks in a known amount of time. A Real Time system should not miss any deadlines for tasks or operations. It is not necessary that all embedded systems should be Real Time in operations. Embedded applications or systems which are mission critical, like flight control systems, Antilock Brake Systems (ABS), etc. are examples of Real Time systems. The design of an embedded Real time system should take the worst case scenario into consideration.

3.1.3 Operates in Harsh Environment

It is not necessary that all embedded systems should be deployed in controlled environments. The environment in which the embedded system deployed may be a dusty one or a high temperature zone or an area subject to vibrations and shock. Systems placed in such areas should be capable to withstand all these adverse operating conditions. The design should take care of the operating conditions of the area where the system is going to implement. For example, if the system needs to be deployed in a high temperature zone, then all the components used in the system should be of high temperature grade. Here we cannot go for a compromise

3.1.4 Distributed

The term distributed means that embedded systems may be a part of larger systems. Many numbers of such distributed embedded systems form a single large embedded control unit. An automatic vending machine

is a typical example for this. The vending machine contains a card reader (for pre-paid vending systems), a vending unit, etc. Each of them are independent embedded units but they work together to perform the overall vending function. Another example is the Automatic Teller Machine (ATM). An ATM contains a card reader embedded unit, responsible for reading and validating the user's ATM card, transaction unit for performing transactions, a currency counter for dispatching/vending currency to the authorised person and a printer unit for printing the transaction details. We can visualise these as independent embedded systems. But they work together to achieve a common goal.

Another typical example of a distributed embedded system is the Supervisory Control And Data Acquisition (SCADA) system used in Control & Instrumentation applications, which contains physically distributed individual embedded control units connected to a supervisory module.

3.1.5 Small Size and Weight

Product aesthetics is an important factor in choosing a product. For example, when you plan to buy a new mobile phone, you may make a comparative study on the pros and cons of the products available in the market. Definitely the product aesthetics (size, weight, shape, style, etc.) will be one of the deciding factors to choose a product. People believe in the phrase "Small is beautiful". Moreover it is convenient to handle a compact device than a bulky product. In embedded domain also compactness is a significant deciding factor. Most of the application demands small sized and low weight products.

mobile phone, you may make a comparative study on the pros and cons of the products available in the market. Definitely the product aesthetics (size, weight, shape, style, etc.) will be one of the deciding factors to choose a product. People believe in the phrase "Small is beautiful". Moreover it is convenient to handle a compact device than a bulky product. In embedded domain also compactness is a significant deciding factor. Most of the application demands small sized and low weight products.

3.1.6 Power Concerns

Power management is another important factor that needs to be considered in designing embedded systems. Embedded systems should be designed in such a way as to minimise the heat dissipation by the system. The production of high amount of heat demands cooling requirements like cooling fans which in turn occupies additional space and make the system bulky. Nowadays ultra low power components are available in the market. Select the design according to the low power components like low dropout regulators, and controllers/ processors with power saving modes. Also power management is a critical constraint in battery operated application. The more the power consumption the less is the battery life.

3.2 QUALITY ATTRIBUTES OF EMBEDDED SYSTEMS

LO 2 Explain the non-functional requirements that needs to be addressed in the design of an embedded system

Quality attributes are the non-functional requirements that need to be documented properly in any system design. If the quality attributes are more concrete and measurable it will give a positive impact on the system development process and the end product. The various quality attributes that needs to be addressed in any Embedded System development are broadly classified into two, namely 'Operational Quality Attributes' and 'Non-Operational Quality Attributes'.

3.2.1 Operational Quality Attributes

The operational quality attributes represent the relevant quality attributes related to the Embedded System when it is in the operational mode or 'online' mode. The important quality attributes coming under this category are listed below:

(1) Response
(2) Throughput
(3) Reliability
(4) Maintainability

(5) Security
(6) Safety

3.2.1.1 Response Response is a measure of quickness of the system. It gives you an idea about how fast your system is tracking the changes in input variables. Most of the embedded systems demand fast response which should be almost Real Time. For example, an embedded system deployed in flight control application should respond in a Real Time manner. Any response delay in the system will create potential impact to the safety of the flight as well as the passengers. It is not necessary that all embedded systems should be Real Time in response. For example, the response time requirement for an electronic toy is not at all time-critical. There is no specific deadline that this system should respond within this particular timeline.

3.2.1.2 Throughput Throughput deals with the efficiency of a system. In general it can be defined as the rate of production or operation of a defined process over a stated period of time. The rates can be expressed in terms of units of products, batches produced, or any other meaningful measurements. In the case of a Card Reader, throughput means how many transactions the Reader can perform in a minute or in an hour or in a day. Throughput is generally measured in terms of 'Benchmark'. A 'Benchmark' is a reference point by which something can be measured. Benchmark can be a set of performance criteria that a product is expected to meet or a standard product that can be used for comparing other products of the same product line.

3.2.1.3 Reliability Reliability is a measure of how much % you can rely upon the proper functioning of the system or what is the % susceptibility of the system to failures.

Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR) are the terms used in defining system reliability. MTBF gives the frequency of failures in hours/weeks/months. MTTR specifies how long the system is allowed to be out of order following a failure. For an embedded system with critical application need, it should be of the order of minutes.

3.2.1.4 Maintainability Maintainability deals with support and maintenance to the end user or client in case of technical issues and product failures or on the basis of a routine system checkup. Reliability and maintainability are considered as two complementary disciplines. A more reliable system means a system with less corrective maintainability requirements and vice versa. As the reliability of the system increases, the chances of failure and non-functioning also reduces, thereby the need for maintainability is also reduced. Maintainability is closely related to the system availability. Maintainability can be broadly classified into two categories, namely, 'Scheduled or Periodic Maintenance (preventive maintenance)' and 'Maintenance to unexpected failures (corrective maintenance)'. Some embedded products may use consumable components

Periodic maintenance'. If the paper feeding part of the printer fails the printer fails to print and it requires immediate repairs to rectify this problem. This is an example of '**Maintenance to unexpected failure**'. In both of the maintenances (scheduled and repair), the printer needs to be brought offline and during this time it will not be available for the user. Hence it is obvious that maintainability is simply an indication of the availability of the product for use. In any embedded system design, the ideal value for availability is expressed as

$$A_i = \text{MTBF}/(\text{MTBF} + \text{MTTR})$$

where A_i = Availability in the ideal condition, MTBF = Mean Time Between Failures, and MTTR = Mean Time To Repair

3.2.1.5 Security 'Confidentiality', 'Integrity', and 'Availability' (The term 'Availability' mentioned here is not related to the term 'Availability' mentioned under the 'Maintainability' section) are the three major measures of information security. Confidentiality deals with the protection of data and application from unauthorised disclosure. Integrity deals with the protection of data and application from unauthorised modification. Availability deals with protection of data and application from unauthorised users. A very good example of the 'Security' aspect in an embedded product is a Personal Digital Assistant (PDA). The PDA can be either a shared resource (e.g. PDAs used in LAB setups) or an individual one. If it is a shared one

3.2.1.6 Safety 'Safety' and 'Security' are two confusing terms. Sometimes you may feel both of them as a single attribute. But they represent two unique aspects in quality attributes. Safety deals with the possible damages that can happen to the operators, public and the environment due to the breakdown of an embedded system or due to the emission of radioactive or hazardous materials from the embedded products. The breakdown of an embedded system may occur due to a hardware failure or a firmware failure. Safety analysis is a must in product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of the damages to an acceptable level. As stated before, some of the safety threats are sudden (like product breakdown) and some of them are gradual (like hazardous emissions from the product).

3b) Explain the working of washing machine with a neat functional diagram. (6M)

4.1 WASHING MACHINE—APPLICATION-SPECIFIC EMBEDDED SYSTEM

LO 1 Illustrate the application specific aspect of embedded systems with examples

People experience the power of embedded systems and enjoy the features and comfort provided by them, but they are totally unaware or ignorant of the intelligent embedded players working behind the products providing enhanced features and comfort. Washing machine is a typical example of an embedded system providing extensive support in home automation applications (Fig. 4.1).

EWF 1495

Fig. 4.1 Washing Machine – Typical example of an embedded system
(Photo courtesy of Electrolux Corporation (www.electrolux.com/au))

As mentioned in an earlier chapter, an embedded system contains sensors, actuators, control unit and application-specific user interfaces like keyboards, display units, etc. You can see all these components in a washing machine if you have a closer look at it. Some of them are visible and some of them may be invisible to you.

The actuator part of the washing machine consists of a motorised agitator, tumble tub, water drawing pump and inlet valve to control the flow of water into the unit. The sensor part consists of the water temperature sensor, level sensor, etc. The control part contains a microprocessor/controller based board with interfaces to the sensors and actuators. The sensor data is fed back to the control unit and the control unit generates the necessary actuator outputs. The control unit also provides connectivity to user interfaces like keypad for setting the washing time, selecting the type of material to be washed like light, medium, heavy duty, etc. User feedback is reflected through the display unit and LEDs connected to the control board. The functional block diagram of a washing machine is shown in Fig. 4.2.

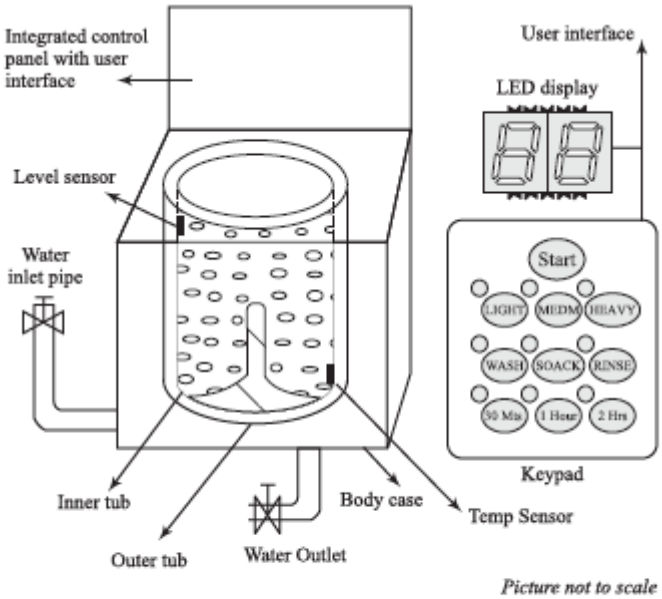


Fig. 4.2 Washing machine – Functional block diagram

Washing machine comes in different designs, like top loading and front loading machines. In top loading models the agitator of the machine twists back and forth and pulls the cloth down to the bottom of the tub.

On reaching the bottom of the tub the clothes work their way back up to the top of the tub where the agitator grabs them again and repeats the mechanism. In the front loading machines, the clothes are tumbled and plunged into the water over and over again. This is the first phase of washing.

In the second phase of washing, water is pumped out from the tub and the inner tub uses centrifugal force to wring out more water from the clothes by spinning at several hundred Rotations Per Minute (RPM). This is called a '*Spin Phase*'. If you look into the keyboard panel of your washing machine you can see three buttons namely* *Wash*, *Spin* and *Rinse*. You can use these buttons to configure the washing stages. As you can see from the picture, the inner tub of the machine contains a number of holes and during the spin cycle the inner tub spins, and forces the water out through these holes to the stationary outer tub from which it is drained off through the outlet pipe.

It is to be noted that the design of washing machines may vary from manufacturer to manufacturer, but the general principle underlying in the working of the washing machine remains the same. The basic controls consist of a timer, cycle selector mechanism, water temperature selector, load size selector and start button. The mechanism includes the motor, transmission, clutch, pump, agitator, inner tub, outer tub and water inlet valve. Water inlet valve connects to the water supply line using at home and regulates the flow of water into the tub.

The integrated control panel consists of a microprocessor/controller based board with I/O interfaces and a control algorithm running in it. Input interface includes the keyboard which consists of wash type selector namely* *Wash*, *Spin* and *Rinse*, cloth type selector namely* *Light*, *Medium*, *Heavy duty*, and washing time setting, etc. The output interface consists of LED/LCD displays, status indication LEDs, etc. connected to the I/O bus of the controller. It is to be noted that this interface may vary from manufacturer to manufacturer and model to model. The other types of I/O interfaces which are invisible to the end user are different kinds of sensor interfaces, namely, water temperature sensor, water level sensor, etc. and actuator interface including motor control for agitator and tub movement control, inlet water flow control, etc.

3c) Design an automatic tea/coffee vending machine based on FSM model. (8M)

Design an automatic tea/coffee vending machine based on FSM model for the following requirement.

The tea/coffee vending is initiated by user inserting a 5 rupee coin. After inserting the coin, the user can either select 'Coffee' or 'Tea' or press 'Cancel' to cancel the order and take back the coin.

The FSM representation for the above requirement is given in Fig. 7.5.

In its simplest representation, it contains four states namely; 'Wait for coin' 'Wait for User Input', 'Dispense Tea' and 'Dispense Coffee'. The event 'Insert Coin' (5 rupee coin insertion), transitions the state to 'Wait for User Input'. The system stays in this state until a user input is received from the buttons 'Cancel', 'Tea' or 'Coffee' (Tea and Coffee are the drink select button). If the event triggered in 'Wait State' is 'Cancel' button press, the coin is pushed out and the state transitions to 'Wait for Coin'. If the event received in the 'Wait State' is either 'Tea' button press, or 'Coffee' button press, the state changes to 'Dispense Tea' and 'Dispense Coffee' respectively. Once the coffee/tea vending is over, the respective states transitions back to the 'Wait for Coin' state. A few modifications like adding a timeout for the 'Wait State' (Currently the 'Wait State' is infinite; it can be re-designed to a timeout based 'Wait State'. If no user input is received within the timeout period, the coin is returned back and the state automatically transitions to 'Wait for Coin' on the timeout event) and capturing another events like, 'Water not available', 'Tea/Coffee Mix not available' and changing the state to an 'Error State' can be added to enhance this design. It is left to the readers as exercise.

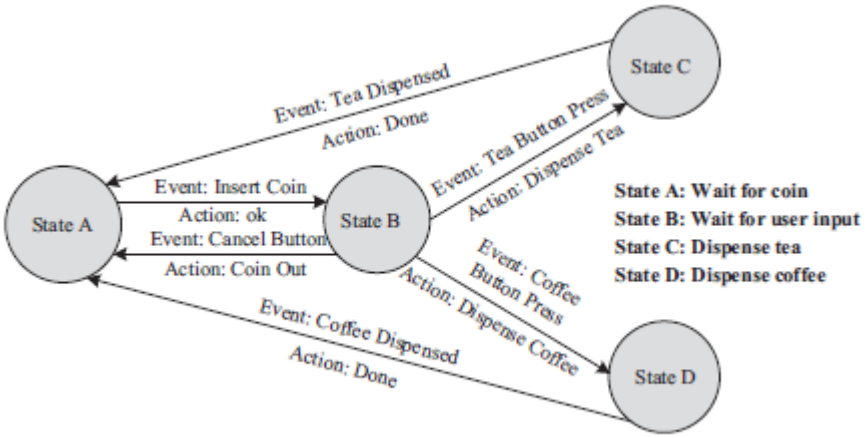


Fig. 7.5 FSM Model for Automatic Tea\Coffee Vending Machine

4a) Explain operational and non-operational attributes of embedded systems. (6M)

3.2 QUALITY ATTRIBUTES OF EMBEDDED SYSTEMS

LO 2 Explain the non-functional requirements that needs to be addressed in the design of an embedded system

Quality attributes are the non-functional requirements that need to be documented properly in any system design. If the quality attributes are more concrete and measurable it will give a positive impact on the system development process and the end product. The various quality attributes that needs to be addressed in any Embedded System development are broadly classified into two, namely 'Operational Quality Attributes' and 'Non-Operational Quality Attributes'.

3.2.1 Operational Quality Attributes

The operational quality attributes represent the relevant quality attributes related to the Embedded System when it is in the operational mode or 'online' mode. The important quality attributes coming under this category are listed below:

(1) Response
(2) Throughput
(3) Reliability
(4) Maintainability

(5) Security
(6) Safety

3.2.1.1 Response Response is a measure of quickness of the system. It gives you an idea about how fast your system is tracking the changes in input variables. Most of the embedded systems demand fast response which should be almost Real Time. For example, an embedded system deployed in flight control application should respond in a Real Time manner. Any response delay in the system will create potential impact to the safety of the flight as well as the passengers. It is not necessary that all embedded systems should be Real Time in response. For example, the response time requirement for an electronic toy is not at all time-critical. There is no specific deadline that this system should respond within this particular timeline.

3.2.1.2 Throughput Throughput deals with the efficiency of a system. In general it can be defined as the rate of production or operation of a defined process over a stated period of time. The rates can be expressed in terms of units of products, batches produced, or any other meaningful measurements. In the case of a Card Reader, throughput means how many transactions the Reader can perform in a minute or in an hour or in a day. Throughput is generally measured in terms of 'Benchmark'. A 'Benchmark' is a reference point by which something can be measured. Benchmark can be a set of performance criteria that a product is expected to meet or a standard product that can be used for comparing other products of the same product line.

3.2.1.3 Reliability Reliability is a measure of how much % you can rely upon the proper functioning of the system or what is the % susceptibility of the system to failures.

Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR) are the terms used in defining system reliability. MTBF gives the frequency of failures in hours/weeks/months. MTTR specifies how long the system is allowed to be out of order following a failure. For an embedded system with critical application need, it should be of the order of minutes.

3.2.1.4 Maintainability Maintainability deals with support and maintenance to the end user or client in case of technical issues and product failures or on the basis of a routine system checkup. Reliability and maintainability are considered as two complementary disciplines. A more reliable system means a system with less corrective maintainability requirements and vice versa. As the reliability of the system increases, the chances of failure and non-functioning also reduces, thereby the need for maintainability is also reduced. Maintainability is closely related to the system availability. Maintainability can be broadly classified into two categories, namely, 'Scheduled or Periodic Maintenance (preventive maintenance)' and 'Maintenance to unexpected failures (corrective maintenance)'. Some embedded products may use consumable components

Periodic maintenance'. If the paper feeding part of the printer fails the printer fails to print and it requires immediate repairs to rectify this problem. This is an example of '**Maintenance to unexpected failure**'. In both of the maintenances (scheduled and repair), the printer needs to be brought offline and during this time it will not be available for the user. Hence it is obvious that maintainability is simply an indication of the availability of the product for use. In any embedded system design, the ideal value for availability is expressed as

$$A_i = \text{MTBF}/(\text{MTBF} + \text{MTTR})$$

where A_i = Availability in the ideal condition, MTBF = Mean Time Between Failures, and MTTR = Mean Time To Repair

3.2.1.5 Security 'Confidentiality', 'Integrity', and 'Availability' (The term 'Availability' mentioned here is not related to the term 'Availability' mentioned under the 'Maintainability' section) are the three major measures of information security. Confidentiality deals with the protection of data and application from unauthorised disclosure. Integrity deals with the protection of data and application from unauthorised modification. Availability deals with protection of data and application from unauthorised users. A very good example of the 'Security' aspect in an embedded product is a Personal Digital Assistant (PDA). The PDA can be either a shared resource (e.g. PDAs used in LAB setups) or an individual one. If it is a shared one

3.2.1.6 Safety 'Safety' and 'Security' are two confusing terms. Sometimes you may feel both of them as a single attribute. But they represent two unique aspects in quality attributes. Safety deals with the possible damages that can happen to the operators, public and the environment due to the breakdown of an embedded system or due to the emission of radioactive or hazardous materials from the embedded products. The breakdown of an embedded system may occur due to a hardware failure or a firmware failure. Safety analysis is a must in product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of the damages to an acceptable level. As stated before, some of the safety threats are sudden (like product breakdown) and some of them are gradual (like hazardous emissions from the product).

3.2.2 Non-Operational Quality Attributes

The quality attributes that needs to be addressed for the product 'not' on the basis of operational aspects are grouped under this category. The important quality attributes coming under this category are listed below.

(1) Testability & Debug-ability
(2) Evolvability
(3) Portability
(4) Time to prototype and market
(5) Per unit and total cost.

3.2.2.1 Testability & Debug-ability Testability deals with how easily one can test the design, application and by which means he/she can test it. For an embedded product, testability is applicable to both the embedded hardware and firmware. Embedded hardware testing ensures that the peripherals and the total hardware functions in the desired manner, whereas firmware testing ensures that the firmware is functioning in the expected way. Debug-ability is a means of debugging the product as such for figuring out the probable sources that create unexpected behaviour in the total system. Debug-ability has two aspects in the embedded system development context, namely, hardware level debugging and firmware level debugging. Hardware debugging is used for figuring out the issues created by hardware problems whereas firmware debugging is employed to figure out the probable errors that appear as a result of flaws in the firmware.

3.2.2.2 Evolvability Evolvability is a term which is closely related to Biology. Evolvability is referred as the non-heritable variation. For an embedded system, the quality attribute 'Evolvability' refers to the ease with which the embedded product (including firmware and hardware) can be modified to take advantage of new firmware or hardware technologies.

3.2.2.3 Portability Portability is a measure of 'system independence'. An embedded product is said to be portable if the product is capable of functioning 'as such' in various environments, target processors/ controllers and embedded operating systems. The ease with which an embedded product can be ported on to a new platform is a direct measure of the re-work required. A standard embedded product should always be flexible and portable. In embedded products, the term 'porting' represents the migration of the embedded firmware written for one target processor (e.g. Intel x86) to a different target processor (say an ARM Cortex M3 processor from Freescale). If the firmware is written in a high level language like 'C' with little target processor-specific functions (operating system extensions or compiler specific utilities), it is very easy to port the firmware for the new processor by replacing those 'target processor-specific functions' with the ones for the new target processor and re-compiling the program for the new target processor-specific settings. Re-compiling the program for the new target processor generates the new target processor-specific machine codes. If the firmware is written in Assembly Language for a particular family of processor (say x86 family), it will be very difficult to translate the assembly language instructions to the new target processor specific language and so the portability is poor.

If you look into various programming languages for application development for desktop applications, you will see that certain applications developed on certain languages run only on specific operating systems and some of them run independent of the desktop operating systems. For example, applications developed using Microsoft technologies (e.g. Microsoft Visual C++ using Visual studio) is capable of running only on Microsoft platforms and may not function on other operating systems; whereas applications developed using 'Java' from Sun Microsystems works on any operating system that supports Java standards.

3.2.2.4 Time-to-Prototype and Market Time-to-market is the time elapsed between the conceptualisation of a product and the time at which the product is ready for selling (for commercial product) or use (for non-commercial products). The commercial embedded product market is highly competitive and time to market the product is a critical factor in the success of a commercial embedded product. There may be multiple players in the embedded industry who develop products of the same category (like mobile phone, portable media players, etc.). If you come up with a new design and if it takes long time to develop and market it, the competitor product may take advantage of it with their product. Also, embedded technology is one where rapid technology change is happening. If you start your design by making use of a new technology and if it takes long time to develop and market the product, by the time you market the product, the technology might have superseded with a new technology. Product prototyping helps a lot in reducing time-to-market. Whenever you have a product idea, you may not be certain about the feasibility of the idea. Prototyping is an informal kind of rapid product development in which the important features of the product under consideration are developed. The time to prototype is also another critical factor. If the prototype is developed faster, the actual estimated development time can be brought down significantly. In order to shorten the time to prototype, make use of all possible options like the use of off-the-shelf components, re-usable assets, etc.

3.2.2.5 Per Unit Cost and Revenue Cost is a factor which is closely monitored by both end user (those who buy the product) and product manufacturer (those who build the product). Cost is a highly sensitive factor for commercial products. Any failure to position the cost of a commercial product at a nominal rate,

may lead to the failure of the product in the market. Proper market study and cost benefit analysis should be carried out before taking a decision on the per-unit cost of the embedded product. From a designer/product development company perspective the ultimate aim of a product is to generate marginal profit. So the budget and total system cost should be properly balanced to provide a marginal profit.

The Product Life Cycle (PLC) Every embedded product has a product life cycle which starts with the design and development phase. The product idea generation, prototyping, Roadmap definition, actual product design and development are the activities carried out during this phase. During the design and development phase there is only investment and no returns. Once the product is ready to sell, it is introduced to the market. This stage is known as the Product Introduction stage. During the initial period the sales and revenue will be low. There won't be much competition and the product sales and revenue increases with time. In the growth phase, the product grabs high market share. During the maturity phase, the growth and sales will be steady and the revenue reaches at its peak. The Product Retirement/Decline phase starts with the drop in sales volume, market share and revenue. The decline happens due to various reasons like competition from similar product with enhanced features or technology changes, etc. At some point of the decline stage, the manufacturer announces discontinuing of the product. The different stages of the embedded products life cycle–revenue, unit cost and profit in each stage–are represented in the following Product Life-cycle graph (Fig. 3.1).

4b) Explain the hardware and software co-design in embedded system. (6M)

7.1 FUNDAMENTAL ISSUES IN HARDWARE SOFTWARE CO-DESIGN

LO 1 Know the co-design approach for embedded hardware and firmware development

The hardware software co-design is a problem statement and when we try to solve this problem statement in real life we may come across multiple issues in the design. The following section illustrates some of the fundamental issues in hardware software co-design.

Selecting the model In hardware software co-design, models are used for capturing and describing the system characteristics. A model is a formal system consisting of objects and composition rules. It is hard to make a decision on which model should be followed in a particular system design. Most often designers switch between a variety of models from the requirements specification to the implementation aspect of the system design. The reason being, the objective varies with each phase; for example at the specification stage, only the functionality of the system is in focus and not the implementation information. When the design moves to the implementation aspect, the information about the system components is revealed and the designer has to switch to a model capable of capturing the system's structure. We will discuss about the different models in a later section of this chapter.

Selecting the Architecture A model only captures the system characteristics and does not provide information on 'how the system can be manufactured?'. The *architecture* specifies how a system is going to implement in terms of the number and types of different components and the interconnection among them. Controller Architecture, Datapath Architecture, Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), Very Long Instruction Word Computing (VLIW), Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), etc. are the commonly used architectures in system design. Some of them fall into Application Specific Architecture Class (like Controller Architecture), while others fall into either general purpose architecture class (CISC, RISC, etc.) or Parallel processing class (like VLIW, SIMD, MIMD, etc.).

The **Controller Architecture** implements the finite state machine model (which we will discuss in a later section) using a state register and two combinational circuits (we will discuss about combinational circuits in a later chapter). The state register holds the present state and the combinational circuits implement the logic for next state and output.

The **Datapath Architecture** is best suited for implementing the data flow graph model where the output is generated as a result of a set of predefined computations on the input data. A datapath represents a channel between the input and output and in datapath architecture the datapath may contain registers, counters, register files, memories and ports along with high speed arithmetic units. Ports connect the datapath to multiple buses. Most of the time the arithmetic units are connected in parallel with pipelining support for bringing high performance.

The **Finite State Machine Datapath** (FSMD) architecture combines the controller architecture with datapath architecture. It implements a controller with datapath. The controller generates the control input whereas the datapath processes the data. The datapath contains two types of I/O ports, out of which one acts as the control port for receiving/sending the control signals from/to the controller unit and the second I/O port interfaces the datapath with external world for data input and data output. Normally the datapath is implemented in a chip and the I/O pins of the chip acts as the data input output ports for the chip resident data path.

The **Complex Instruction Set Computing** (CISC) architecture uses an instruction set representing complex operations. It is possible for a CISC instruction set to perform a large complex operation (e.g. Reading a register value and comparing it with a given value and then transfer the program execution to a new address location (The CJNE instruction for 8051 ISA)) with a single instruction. The use of a single complex instruction in place of multiple simple instructions greatly reduces the program memory access and program memory size requirement. However it requires additional silicon for implementing microcode decoder for decoding the CISC instruction. The datapath for the CISC processor is complex. On the other hand, Reduced Instruction Set Computing (RISC) architecture uses instruction set representing simple operations and it requires the execution of multiple RISC instructions to perform a complex operation. The data path of RISC architecture contains a large register file for storing the operands and output. RISC instruction set is designed to operate on registers. RISC architecture supports extensive pipelining.

The **Very Long Instruction Word** (VLIW) architecture implements multiple functional units (ALUs, multipliers, etc.) in the datapath. The VLIW instruction packages one standard instruction per functional unit of the datapath.

Parallel processing architecture implements multiple concurrent Processing Elements (PEs) and each processing element may associate a datapath containing register and local memory. Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) architectures are examples for parallel processing architecture. In SIMD architecture, a single instruction is executed in parallel with the help of the Processing Elements. The scheduling of the instruction execution and controlling of each PE is performed through a single controller. The SIMD architecture forms the basis of re-configurable processor (We will discuss about re-configurable processors in a later chapter). On the other hand, the processing elements of the MIMD architecture execute different instructions at a given point of time. The MIMD architecture forms the basis of multiprocessor systems. The PEs in a multiprocessor system communicates through mechanisms like shared memory and message passing.

Selecting the language A programming language captures a 'Computational Model' and maps it into architecture. There is no hard and fast rule to specify this language should be used for capturing this model. A model can be captured using multiple programming languages like C, C++, C#, Java, etc. for software implementations and languages like VHDL, System C, Verilog, etc. for hardware implementations. On the other hand, a single language can be used for capturing a variety of models. Certain languages are good in capturing certain computational model. For example, C++ is a good candidate for capturing an object oriented model. The only pre-requisite in selecting a programming language for capturing a model is that the language should capture the model easily.

Partitioning System Requirements into hardware and software So far we discussed about the models for capturing the system requirements and the architecture for implementing the system. From an implementation perspective, it may be possible to implement the system requirements in either hardware or software (firmware). It is a tough decision making task to figure out which one to opt. Various hardware software trade-offs are used for making a decision on the hardware-software partitioning. We will discuss them in detail in a later section of this chapter.

4c) With the help of FSM model, explain the system design and operation of automatic seat belt warning. (8M)

A Finite State Machine (FSM) model is one in which the number of states are finite. In other words the system is described using a finite number of possible states. As an example let us consider the design of an embedded system for driver/passenger 'Seat Belt Warning' in an automotive using the FSM model. The system requirements are captured as.

1. When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds.
2. The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.

Here the states are 'Alarm Off', 'Waiting' and 'Alarm On' and the events are 'Ignition Key ON', 'Ignition Key OFF', 'Timer Expire', 'Alarm Time Expire' and 'Seat Belt ON'. Using the FSM, the system requirements can be modeled as given in Fig. 7.3.

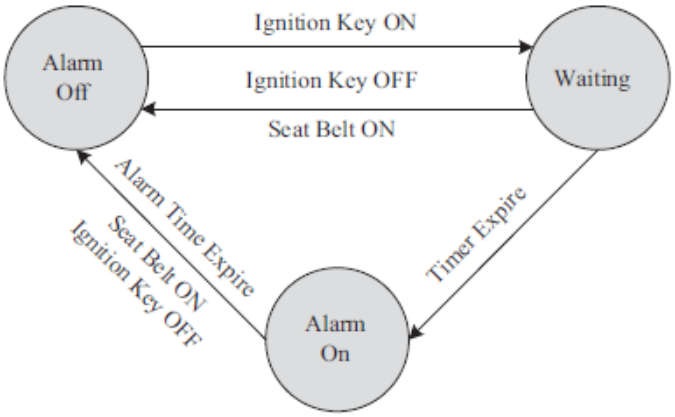


Fig. 7.3 FSM Model for Automatic seat belt warning system

5a) Explain monolithic and microkernel with suitable example for each. (6M)

10.1.1.2 Monolithic Kernel and Microkernel

As we know, the kernel forms the heart of an operating system. Different approaches are adopted for building an Operating System kernel. Based on the kernel design, kernels can be classified into '*Monolithic*' and '*Micro*'.

Monolithic Kernel In monolithic kernel architecture, all kernel services run in the kernel space. Here all kernel modules run within the same memory space under a single kernel thread. The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilisation of the low-level features of the underlying system. The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application. LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel. The architecture representation of a monolithic kernel is given in Fig. 10.2.

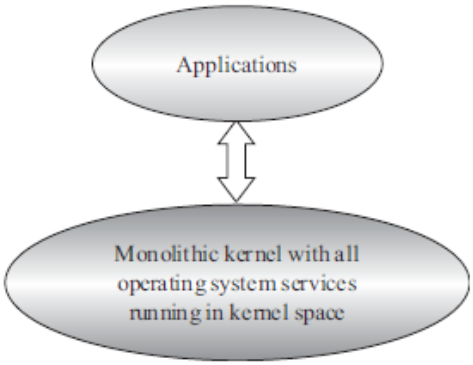


Fig. 10.2 The Monolithic Kernel Model

Microkernel The microkernel design incorporates only the essential set of Operating System services into the kernel. The rest of the Operating System services are implemented in programs known as '*Servers*' which runs in user space. This provides a highly modular design and OS-neutral abstraction to the kernel. Memory management, process management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel. Mach, QNX, Minix 3 kernels are examples for microkernel. The architecture representation of a microkernel is shown in Fig. 10.3.

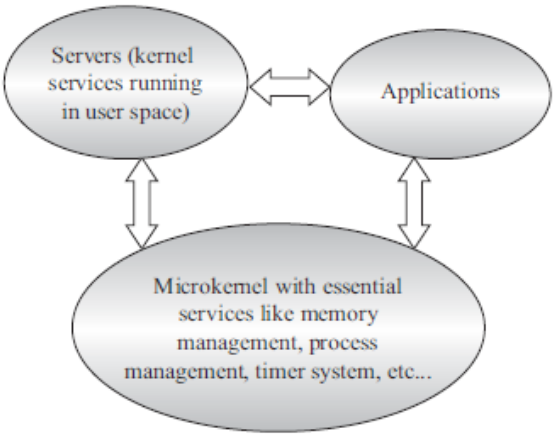


Fig. 10.3 The Microkernel model

Microkernel based design approach offers the following benefits

- **Robustness:** If a problem is encountered in any of the services, which runs as '*Server*' application, the same can be reconfigured and re-started without the need for re-starting the entire OS. Thus, this approach is highly useful for systems, which demands high '*availability*'. Refer Chapter 3 to get an understanding of '*availability*'. Since the services which run as '*Servers*' are running on a different memory space, the chances of corruption of kernel services are ideally zero.
- **Configurability:** Any services, which run as '*Server*' application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

5b) Explain different conditions that favour deadlock. (6M)

10.8.1.2 Deadlock

A race condition produces incorrect results whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes. A situation very similar to our traffic jam issues in a junction as illustrated in Fig. 10.24.

Fig. 10.24 Deadlock visualisation

In its simplest form 'deadlock' is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process (Fig. 10.25). To elaborate: Process A holds a resource *x* and it wants a resource *y* held by Process B. Process B is currently holding resource *y* and it wants the resource *x* which is currently held by Process A. Both hold the respective resources and they compete each other to get the resource held by the respective processes. The result of the competition is 'deadlock'. None of the competing process will be able to access the resources held by other processes since they are locked by the respective processes (If a mutual exclusion policy is implemented for shared resource access, the resource is locked by the process which is currently accessing it).

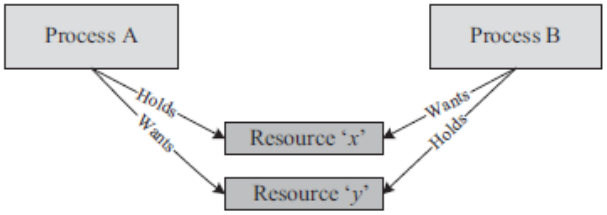


Fig. 10.25 Scenarios leading to deadlock

The different conditions favouring a deadlock situation are listed below.

Mutual Exclusion: The criteria that only one process can hold a resource at a time. Meaning processes should access shared resources with mutual exclusion. Typical example is the

Hold and Wait: The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.

No Resource Preemption: The criteria that operating system cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.

Circular Wait: A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general, there exists a set of waiting process P0, P1 … P*n* with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held by P0, …, P*n* is waiting for a resource held by P0 and P0 is waiting for a resource held by P*n* and so on… This forms a circular wait queue.

5c) Describe pre-entire SIF scheduling and calculate all the performance factors. (8M)

10.5.2.1 Preemptive SJF Scheduling/Shortest Remaining Time (SRT)

The non-preemptive SJF scheduling algorithm sorts the 'Ready' queue only after completing the execution of the current process or when the process enters 'Wait' state, whereas the preemptive SJF scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated time for the currently executing process. If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution. Thus preemptive SJF scheduling always compares the execution completion time (It is same as the remaining time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution. Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling.

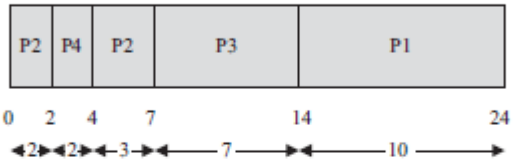
Now let us solve Example 2 given under the Non-preemptive SJF scheduling for preemptive SJF scheduling. The problem statement and solution is explained in the following example.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process P4 with estimated completion time 2 ms enters the 'Ready' queue after 2 ms. Assume all the processes contain only CPU operation and no I/O operations are involved.

At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the SRT scheduler picks up the process with the shortest remaining time for execution completion (In this example, P2 with remaining time 5 ms) for scheduling. The execution sequence diagram for this is same as that of example 1 under non-preemptive SJF scheduling.

Now process P4 with estimated execution completion time 2 ms enters the '*Ready*' queue after 2 ms of start of execution of P2. Since the SRT algorithm is preemptive, the remaining time for completion of process P2 is checked with the remaining time for completion of process P4. The remaining time for completion of P2 is 3 ms which is greater than that of the remaining time for completion of the newly entered process P4 (2 ms). Hence P2 is preempted and P4 is scheduled for execution. P4 continues its execution to finish since there is no new process entered in the 'Ready' queue during its execution. After 2 ms of scheduling P4 terminates and now the scheduler again sorts the '*Ready*' queue based on the remaining time for completion of the processes present in the 'Ready' queue. Since the remaining time for P2 (3 ms), which is preempted by P4 is less than that of the remaining time for other processes in the 'Ready' queue, P2 is scheduled for execution. Due to the arrival of the process P4 with execution time 2 ms, the '*Ready*' queue is re-sorted in the order P2, P4, P2, P3, P1. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram



The waiting time for all the processes are given as

Waiting time for P2 = 0 ms + (4 – 2) ms = 2 ms (P2 starts executing first and is interrupted by P4 and has to wait till the completion of P4 to get the next CPU slot)

Waiting time for P4 = 0 ms (P4 starts executing by preempting P2 since the execution time for completion of P4 (2 ms) is less than that of the Remaining time for execution completion of P2 (Here it is 3 ms))

Waiting time for P3 = 7 ms (P3 starts executing after completing P4 and P2)

Waiting time for P1 = 14 ms (P1 starts executing after completing P4, P2 and P3)

Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P4+P2+P3+P1)) / 4
= (0 + 2 + 7 + 14)/4 = 23/4
= 5.75 milliseconds

Turn Around Time (TAT) for P2 = 7 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 2 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (2 – 2) + 2)

Turn Around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P2+P4+P3+P1)) / 4
= (7+2+14+24)/4 = 47/4
= 11.75 milliseconds

Now let's compare the Average Waiting time and Average Turn Around Time with that of the Average waiting time and Average Turn Around Time for non-preemptive SJF scheduling (Refer to Example 2 given under the section Non-preemptive SJF scheduling)

Average Waiting Time in non-preemptive SJF scheduling = 6 ms
Average Waiting Time in preemptive SJF scheduling = 5.75 ms
Average Turn Around Time in non-preemptive SJF scheduling = 12 ms
Average Turn Around Time in preemptive SJF scheduling = 11.75 ms

6a) Explain Process, task, threads in ARM Processor. (6 Marks)

Solution:

- A **process** is a self-contained execution environment with its **own memory space**, code, data, and system resources.

- It can be considered as a **program in execution**.

- In an **ARM-based embedded system**, a process is used to execute different applications or services independently.

- Each process has:

    - Its own **address space**
    - **Stack**, **heap**, **code**, and **data**
    - Unique **Process ID (PID)**

- Context switching between processes requires saving and restoring the full state of the processor (registers, stack, etc.).

- A **task** is a **smaller unit of execution within a system**, often used interchangeably with **thread** in embedded systems.

- In **Real-Time Operating Systems (RTOS)** like **Keil RTX**, a task refers to a specific function or job that is scheduled by the kernel.

- ARM processors running RTOS create multiple tasks for concurrent execution, enhancing **real-time performance**.

- Tasks may share memory (unlike processes), reducing context-switch overhead.

- A **thread** is the **smallest unit of execution** within a process.

- Multiple threads within the same process share:

    - **Code**
    - **Global variables**
    - **Heap memory**
    - But each thread has **its own stack and registers**.

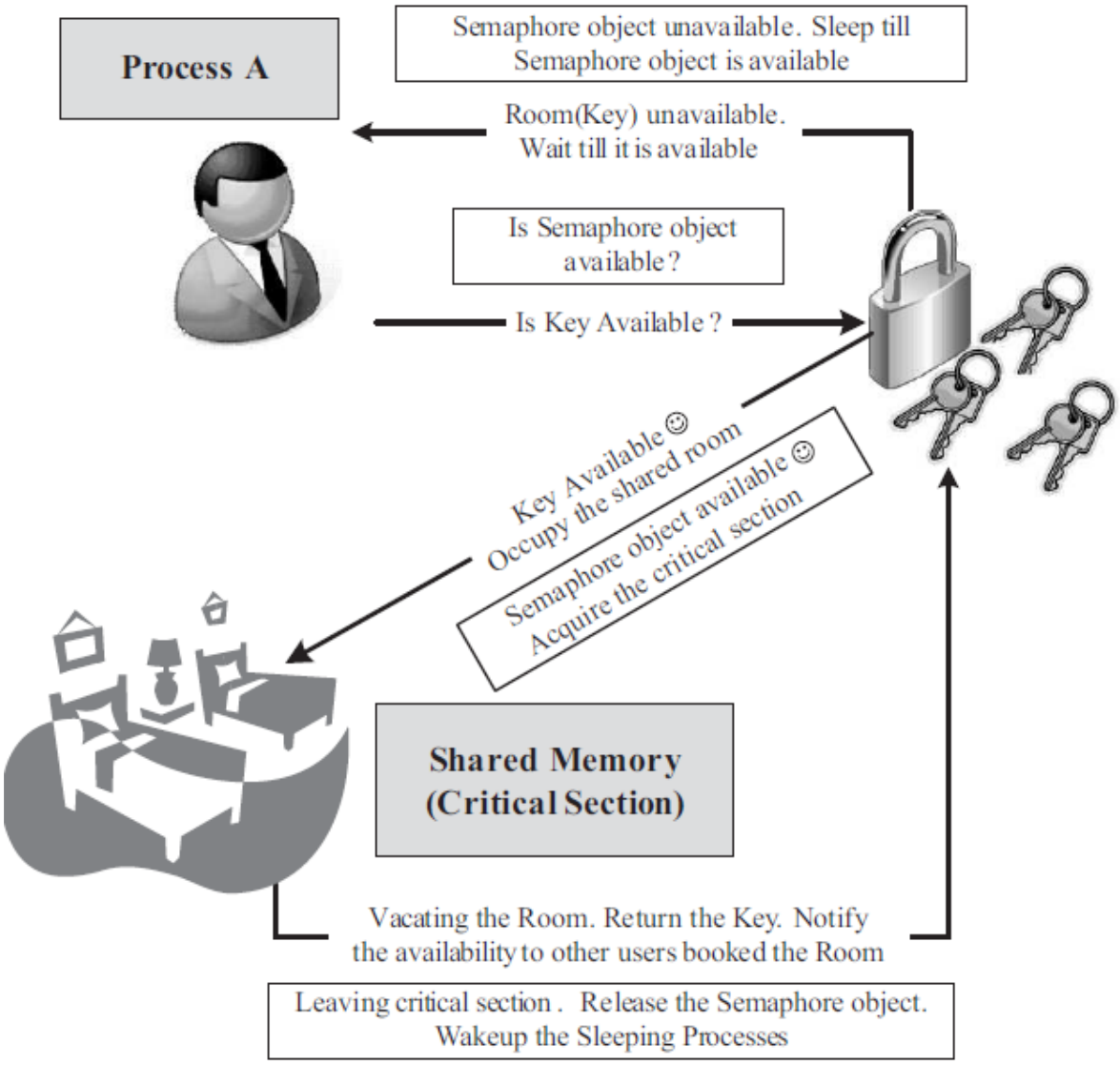
- Threads are useful in **multi-core ARM processors** or RTOS to perform **concurrent operations**, such as UI update and background data processing.

- Thread switching is **faster than process switching**.

6b) with a diagram explain the concept of counting semaphore with an example. (6 Marks)

Solution:

The 'Counting Semaphore' limits the access of resources by a fixed number of processes/threads. 'Counting Semaphore' maintains a count between zero and a maximum value. It limits the usage of the resource to the maximum value of the count supported by it. The state of the counting semaphore object is set to 'signalled' when the count of the object is greater than zero. The count associated with a 'Semaphore object' is decremented by one when a process/thread acquires it and the count is incremented by one when a process/thread releases the 'Semaphore object'. The state of the 'Semaphore object' is set to non-signalled when the semaphore is acquired by the maximum number of processes/threads that the semaphore can support (i.e. when the count associated with the 'Semaphore object' becomes zero). A real world example for the counting semaphore concept is the dormitory system for accommodation. A dormitory contains a fixed number of beds (say 5) and at any point of time it can be shared by the maximum number of users supported by the dormitory. If a person wants to avail the dormitory facility, he/she can contact the dormitory caretaker for checking the availability. If beds are available in the dorm the caretaker will hand over the keys to the user. If beds are not available currently, the user can register his/her name to get notifications when a slot is available. Those who are availing the dormitory shares the dorm facilities like TV, telephone, toilet, etc. When a dorm user vacates, he/she gives the keys back to the caretaker. The caretaker informs the users, who booked in advance, about the dorm availability.



The creation and usage of 'counting semaphore object' is OS kernel dependent.

6c) Explain the IDE Environment for embedded system design with a neat diagram. (8 Marks)

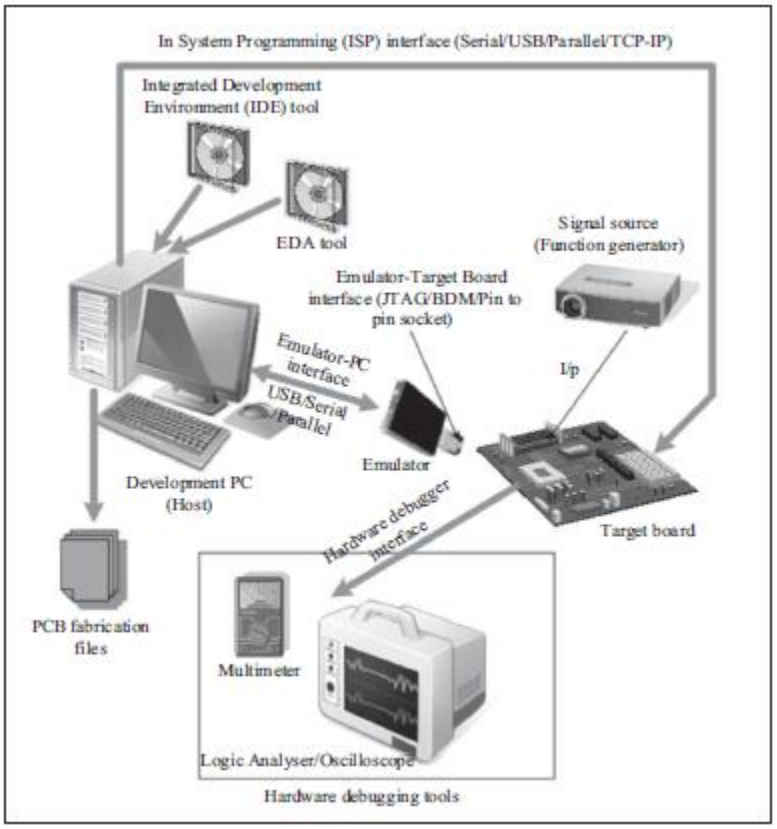
Solution:

In embedded system development context, Integrated Development Environment (IDE) stands for an integrated environment for developing and debugging the target processor specific embedded firmware. IDE is a software package which bundles a 'Text Editor (Source Code Editor)', 'Cross-compiler (for cross platform development and compiler for same platform development)', 'Linker' and a 'Debugger'. Some IDEs may provide interface to target board emulators, Target processor's/controller's Flash memory programmer, etc. and incorporate other software development utilities like 'Version Control Tool', 'Help File for the Development Language', etc. IDEs can be either command line based or GUI based. Command line based IDEs may include little or less GUI support. The old version of TURBO C IDE for developing applications in C/C++ for x86 processor on Windows platform is an example for a

generic IDE with command line interface. GUI based IDEs provide a Visual Development Environment with user interactions through touch/mouse click interface. Such IDEs are generally known as Visual IDEs. Visual IDEs are very helpful in firmware development. A typical example for a Visual IDE is Microsoft® Visual Studio for developing Visual C++ and Visual Basic programs. Other examples are NetBeans and Eclipse.

IDEs used in embedded firmware development are slightly different from the generic IDEs used for high level language based development for desktop applications. In Embedded Applications, the IDE is either supplied by the target processor/controller manufacturer or by third party vendors or as Open Source. MPLAB is an IDE tool supplied by microchip for developing embedded firmware using their PIC family of microcontrollers. Keil µVision5 (spelt as micro vision five) from ARMKeil is an example for a third party IDE, which is used for developing embedded firmware for 8051/ARM family microcontrollers. CodeWarrior Development Studio is an IDE for ARM family of processors/MCUs and DSP chips from Freescale. It should be noted that in embedded firmware development applications each IDE is designed for a specific family of controllers/processors and it may not be possible to develop firmware for all family of controllers/processors using a single IDE (as of now there is no known IDE with support for all family of processors/controllers).

However there is a rapid move happening towards the open source IDE, Eclipse for embedded development. Most of the processor/control manufacturers and third party IDE providers are trying to build the IDE around the popular Eclipse open source IDE. This may lead to a single IDE based on Eclipse for embedded system development in the near future. Since this book is primarily focusing on 8051 based embedded firmware development, the IDE chosen for demonstration is Keil µVision5. A demo version of the tool for Microsoft Windows OS based development is available for free download from



7a) Explain the functions of various units in ARM Cortex M3 Processor architecture in brief. (8 Marks)

Solution:

The ARM Cortex-M3 processor is a 32-bit RISC processor designed for low-cost, high-performance embedded applications. Its architecture is built around the ARMv7-M instruction set and includes several key units, each with specific roles:

1. Processor Core

- Executes instructions using the **Harvard architecture** (separate instruction and data buses).
- Supports **Thumb-2 instruction set** for higher code density and performance.
- Includes a **3-stage pipeline**: Fetch, Decode, Execute.

2. Nested Vectored Interrupt Controller (NVIC)

- Manages **interrupts and exceptions** with up to **240 external interrupts**.
- Supports **preemptive priority levels**, improving response time.
- Offers **low-latency interrupt handling** (12 clock cycles or less).

3. Bus Interface Unit

- Interfaces with memory and peripherals via the **AMBA AHB-Lite** bus.
- Manages instruction and data transfers efficiently.
- Supports **memory-mapped I/O** and **bit-banding** for atomic operations.

4. Register Bank

- Includes **13 general-purpose registers (R0–R12)**, **SP (R13)**, **LR (R14)**, **PC (R15)**.
- Also contains **Program Status Registers (xPSR)**:
    - **APSR**, **IPSR**, **EPSR**
    - Hold flags, current exception number, execution state, etc.

5. System Control Block (SCB)

- Controls system exceptions like **Hard Fault**, **NMI**, and **SysTick**.
- Manages system-level configuration and fault status.
- Contains configuration registers for **vector table**, **exception handling**, etc.

6. Memory Protection Unit (MPU)

- Provides basic **memory protection and access control**.
- Allows definition of memory regions with **privilege and access rights**.
- Helps prevent accidental corruption of memory by faulty code.

7. SysTick Timer

- A 24-bit **count-down timer** built for RTOS **task switching**.
- Generates periodic **system tick interrupts**.
- Helps implement **real-time scheduling**.

8. Debug Unit

- Supports **Serial Wire Debug (SWD)** and **JTAG**.
- Allows for **breakpoints, watchpoints, and step-through debugging**.
- Essential for firmware testing and validation.

9. Exception Model

- Supports **system exceptions** (Reset, NMI, HardFault) and **programmable interrupts**.
- Uses **vector table** to store addresses of exception handlers.
- Facilitates fast and deterministic exception response.

7b) Explain the various interrupts and exception along with vector address. (6 Marks)

Solution:

EXCEPTIONS, INTERRUPTS, AND THE VECTOR TABLE

When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address. The address is within a special address range called the *vector table*. The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.

The memory map address `0x00000000` is reserved for the vector table, a set of 32-bit words. On some processors the vector table can be optionally located at a higher address in memory (starting at the offset `0xffff0000`). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see Table 2.6). Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:

- *Reset vector* is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
- *Undefined instruction vector* is used when the processor cannot decode an instruction.
- *Software interrupt vector* is called when you execute a `SWI` instruction. The `SWI` instruction is frequently used as the mechanism to invoke an operating system routine.
- *Prefetch abort vector* occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
- *Data abort vector* is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.
- *Interrupt request vector* is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the *cpsr*.
- *Fast interrupt request vector* is similar to the interrupt request but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the *cpsr*.

The vector table.

Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

7c) Explain the ARM Core data flow model with a neat diagram. (6 Marks)

Solution:

Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item. Figure 2.1 shows a Von Neumann implementation of the ARM—data items and instructions share the same bus. In contrast, Harvard implementations of the ARM use two different buses.

The instruction decoder translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.

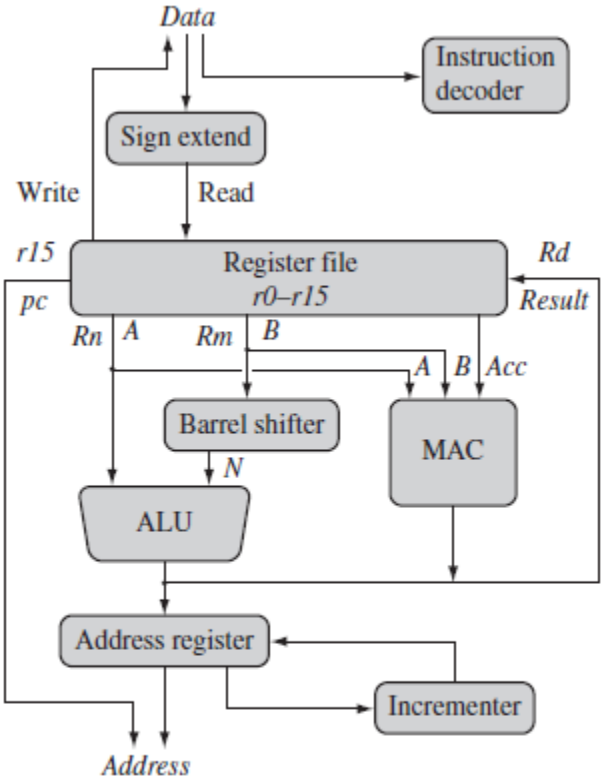
The ARM processor, like all RISC processors, uses a *load-store architecture*. This means it has two instruction types for transferring data in and out of the processor: load instructions copy data from memory to registers in the core, and conversely the store

instructions copy data from registers to memory. There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out solely in registers.

Data items are placed in the *register file*—a storage bank made up of 32-bit registers. Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.

ARM instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*. Source operands are read from the register file using the internal buses *A* and *B*, respectively.

The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values *Rn* and *Rm* from the *A* and *B* buses and computes a result. Data processing instructions write the result in *Rd* directly to the register file. Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the *Address* bus.



ARM core dataflow model.

Figure 2.1 ARM core dataflow model.

One important feature of the ARM is that register *Rm* alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.

After passing through the functional units, the result in *Rd* is written back to the register file using the *Result* bus. For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location. The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

8a) Explain the Program Status register in Cortex-M3 along with Vector address. (8 Marks)

Solution:

The ARM core uses the *cpsr* to monitor and control internal operations. The *cpsr* is a dedicated 32-bit register and resides in the register file. Figure 2.3 shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.

The *cpsr* is divided into four fields, each 8 bits wide: flags, status, extension, and control. In current designs the extension and status fields are reserved for future use. The control field contains the processor mode, state, and interrupt mask bits. The flags field contains the condition flags.

Some ARM processor cores have extra bits allocated. For example, the *J* bit, which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8-bit instructions.

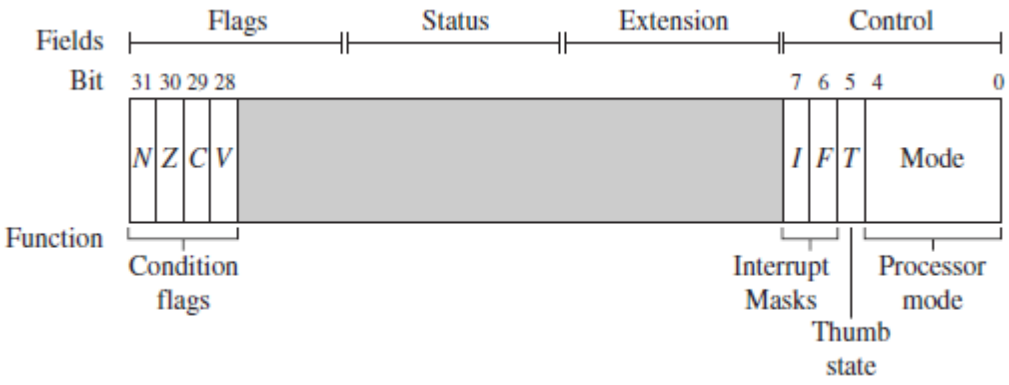


Figure 2.3 A generic program status register (*psr*).

ARM and Thumb instruction set features.

	ARM (*cpsr* $T = 0$)	Thumb (*cpsr* $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +*pc*	8 general-purpose registers +7 high registers +*pc*

Interrupt Masks

Interrupt masks are used to stop specific interrupt requests from interrupting the processor. There are two interrupt request levels available on the ARM processor core—*interrupt request* (IRQ) and *fast interrupt request* (FIQ).

The *cpsr* has two interrupt mask bits, 7 and 6 (or *I* and *F*), which control the masking of IRQ and FIQ, respectively. The *I* bit masks IRQ when set to binary 1, and similarly the *F* bit masks FIQ when set to binary 1.

Condition Flags

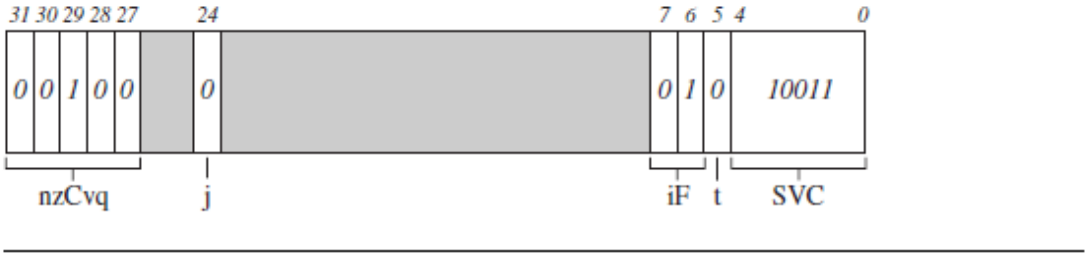
Condition flags are updated by comparisons and the result of ALU operations that specify the S instruction suffix. For example, if a SUBS subtract instruction results in a register value of zero, then the *Z* flag in the *cpsr* is set. This particular subtract instruction specifically updates the *cpsr*.

Condition flags.

Flag	Flag name	Set when
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero, frequently used to indicate equality
N	Negative	bit 31 of the result is a binary 1

With processor cores that include the DSP extensions, the *Q* bit indicates if an overflow or saturation has occurred in an enhanced DSP instruction. The flag is "sticky" in the sense that the hardware only sets this flag. To clear the flag you need to write to the *cpsr* directly.

In Jazelle-enabled processors, the *J* bit reflects the state of the core; if it is set, the core is in Jazelle state. The *J* bit is not generally usable and is only available on some processor cores. To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems.



Example: *cpsr = nzCvqjiFt_SVC.*

8b) Explain the five applications of Cortex-M3 based on its features. (6 Marks)

Solution:

The application areas and the products in the embedded domain are countless. A few of the Important domains and products are listed below:
(1) Consumer electronics: Camcorders, cameras, etc.
(2) Household appliances: Television, DVD players, washing machine, fridge, microwave oven, etc.
(3) Home automation and security systems: Air conditioners, sprinklers, intruder detection alarms, closed circuit television cameras, fi re alarms, etc.
(4) Automotive industry: Anti-lock breaking systems (ABS), engine control, ignition systems, automatic Navigation systems, etc.
(5) Telecom: Cellular telephones, telephone switches, handset multimedia applications, etc.
(6) Computer peripherals: Printers, scanners, fax machines, etc.
(7) Computer networking systems: Network routers, switches, hubs, firewalls, etc.
(8) Healthcare: Different kinds of scanners, EEG, ECG machines etc.
(9) Measurement & Instrumentation: Digital multimeters, digital CROs, logic analysers PLC systems, etc.
(10) Banking & Retail: Automatic teller machines (ATM) and currency counters, point of sales (POS)
(11) Card Readers: Barcode, smart card readers, hand held devices, etc.
(12) Wearable Devices: Health and Fitness Trackers, Smartphone Screen extension for notifi cations, etc.
(13) Cloud Computing and Internet of Things (IOT)

The ARM Cortex-M3 processor is widely used in embedded and real-time systems due to its low power consumption, high performance, and rich peripheral support. Based on these features, five key applications are:

1. Industrial Automation

Application: Used in motor controllers, PLC systems, and factory sensors for fast and reliable task execution.

2. Consumer Electronics

Application: Found in smart appliances, digital cameras, remote controls, and set-top boxes where efficient power management is needed.

3. Automotive Systems

Application: Used in dashboard displays, body electronics, and airbag systems in cars.

4. Medical Devices

Application: Integrated into portable health monitors, digital thermometers, and blood pressure devices for safe, accurate operation.

5. Internet of Things (IoT) Devices

Application: Used in smart home systems, wearables, wireless sensor networks, and connected meters.

8c) with a diagram, explain two operation modes and privilege levels in Cortex M3. (6 Marks)

Solution:

1. Operation Modes

Mode	Description
Thread Mode	Normal program execution mode (main code or after interrupt handling).
Handler Mode	Entered automatically to **handle exceptions or interrupts**.

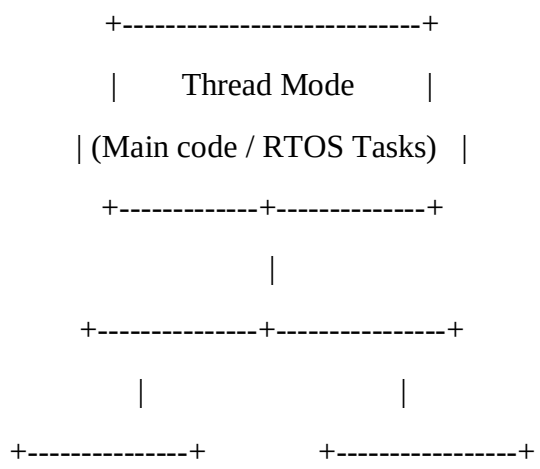
- **Thread Mode** is used for application-level code.
- **Handler Mode** is used during interrupt or exception handling.

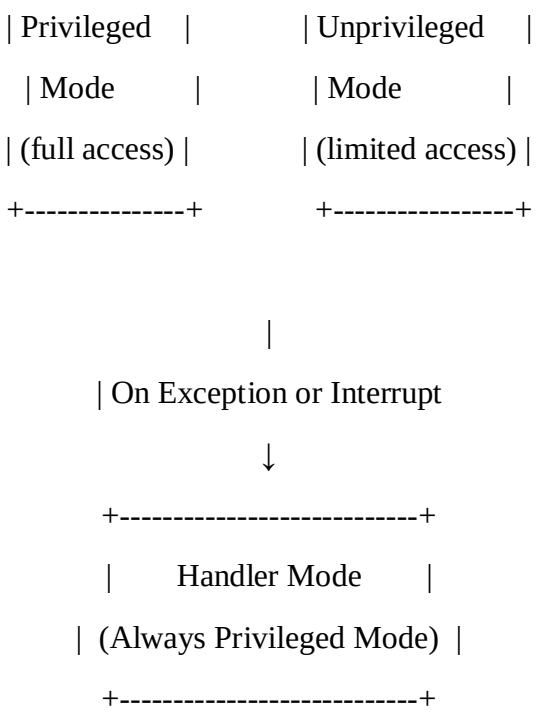
---

2. Privilege Levels

Privilege Level	Description
Privileged	Can access **all instructions and system resources** (e.g., NVIC, MPU).
Unprivileged	Access to **limited instructions and memory** (safe for user code).

- Privilege level can **change dynamically** using the **CONTROL register**.
- Only **privileged code** can switch back from unprivileged to privileged.





User Mode It is the main execution mode for user applications. This mode is also known as 'Unprivileged Mode'. It enables the protection and isolation of operating system from user applications.

Fast Interrupt Processing (FIQ) Mode The processor enters this mode when a high priority interrupt is raised.

Normal Interrupt Processing (IRQ) Mode The processor enters this mode when a normal priority interrupt (Interrupts other than high priority) is raised.

Supervisor/Software Interrupt Mode The processor enters this mode on reset and when a software interrupt instruction is executed.

Abort Mode Enters this mode when a memory access violation occurs

Undefined Instruction Mode Enters this mode when the processor tries to execute an undefined instruction.

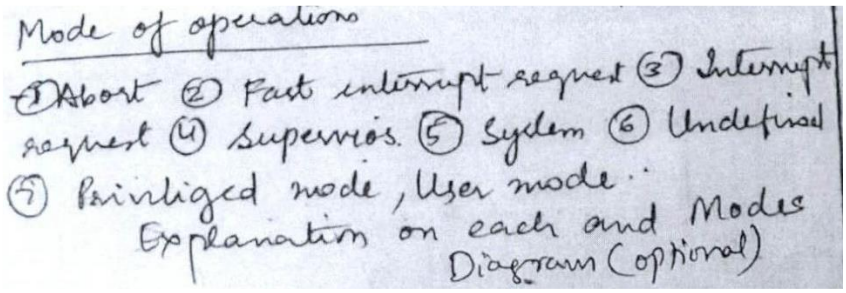
System Mode This mode is used for running operating system tasks. It uses the same register as the User mode.

PROCESSOR MODES

The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or nonprivileged: A privileged mode allows full read-write access to the *cpsr*. Conversely, a nonprivileged mode only allows read access to the control field in the *cpsr* but still allows read-write access to the condition flags.

There are seven processor modes in total: six privileged modes (*abort*, *fast interrupt request*, *interrupt request*, *supervisor*, *system*, and *undefined*) and one nonprivileged mode (*user*).

The processor enters *abort* mode when there is a failed attempt to access memory. *Fast interrupt request* and *interrupt request* modes correspond to the two interrupt levels available on the ARM processor. *Supervisor* mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in. *System* mode is a special version of *user* mode that allows full read-write access to the *cpsr*. *Undefined* mode is used when the processor encounters an instruction that is undefined or not supported by the implementation. *User* mode is used for programs and applications.



9a) Write an ALP to add first 10 Integer number using Cortex M3 Processor. (6 Marks)

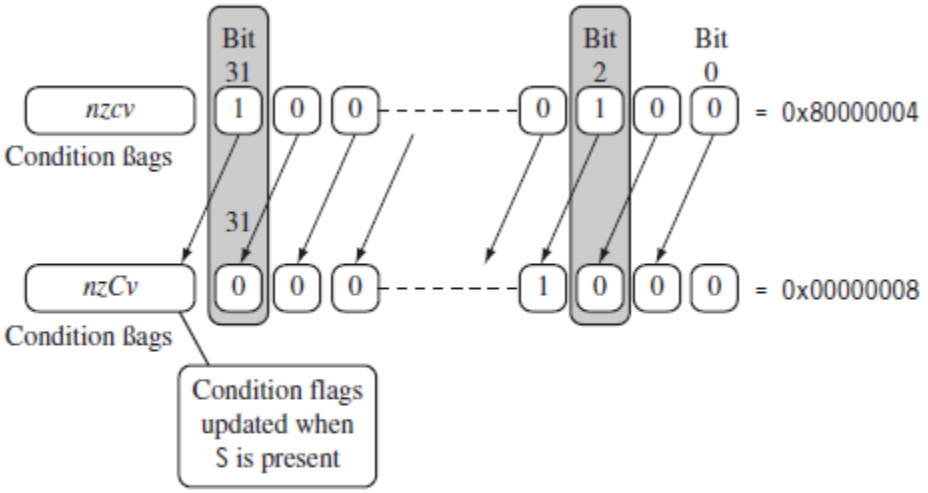
Solution:

```
        AREA ADD_NUMBERS, CODE, READONLY
        ENTRY
        MOV     R0, #1        ; R0 = counter = 1
        MOV     R1, #0        ; R1 = sum = 0
LOOP    ADD     R1, R1, R0    ; sum = sum + counter
        ADD     R0, R0, #1    ; counter = counter + 1
        CMP     R0, #11       ; check if counter <= 10
        BNE     LOOP          ; repeat if not equal to 11
STOP    B       STOP          ; infinite loop to end program
        END
```

9b) Explain Shift and Rotate Instructions of CORTEX M3 with examples (6 Marks)

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$	none

Note: *x* represents the register being shifted and *y* represents the shift amount.



Logical shift left by one.

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

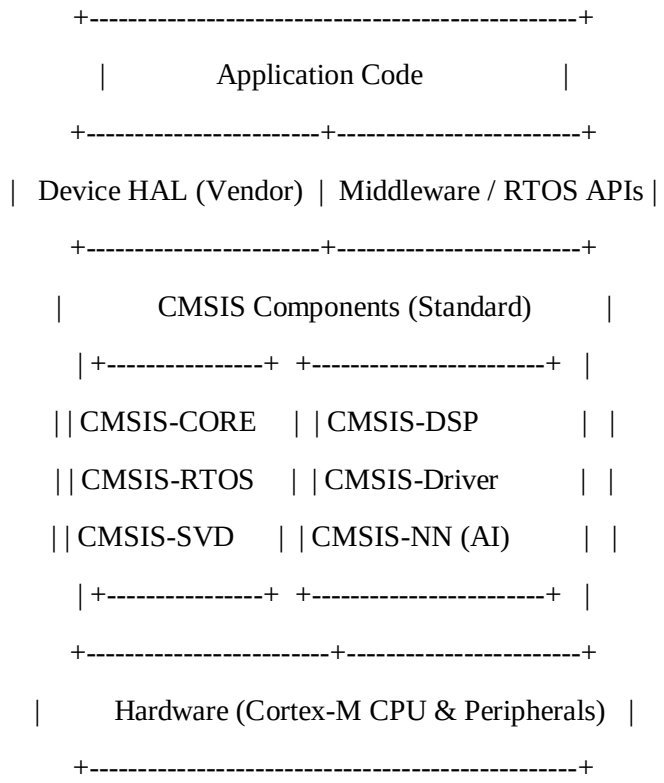
This example of a MOVS instruction shifts register *r1* left by one bit. This multiplies register *r1* by a value 2^1. As you can see, the *C* flag is updated in the *cpsr* because the S suffix is present in the instruction mnemonic.

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 0x00000000
        r1 = 0x80000004

        MOVS    r0, r1, LSL #1

POST    cpsr = nzCvqiFt_USER
        r0 = 0x00000008
        r1 = 0x80000004
```

9c) Describe CMSIS with Diagram and its functions.(6 Marks)

CMSIS (Cortex Microcontroller Software Interface Standard) is a vendor-independent hardware abstraction layer for ARM Cortex-M processors, developed by ARM Ltd.

It provides a standardized software interface to access processor features, peripherals, and real-time operating systems (RTOS), improving portability and code reusability across different microcontroller platforms.



10a) Explain 16-bit instructions with example (6 Marks)

a) ADD
b) CMP
c) ASR

a) The ADD instruction performs addition between two registers or between a register and an immediate value.

ADD (Register)

Adds the values of two registers and stores the result in a register.

Syntax: ADD Rd, Rn, Rm

Example: ADD R2, R1, R0 ; R2 = R1 + R0

Syntax: ADD Rd, Rn, #imm

Example: ADD R1, R1, #5 ; R1 = R1 + 5

Syntax: ADD Rd, SP, #imm

Example: ADD R0, SP, #16 ; R0 = SP + 16

b) CMP stands for Compare.

It subtracts one value from another without storing the result. Flags are updated (Zero, Negative, Carry, Overflow) based on the result. Used mainly for conditional branching (e.g., BEQ, BNE, BGT, etc.)

MOV R0, #5

MOV R1, #5

CMP R0, R1 ; Updates flags (Z=1, since R0 == R1)

CMP Rn, #imm8

c) `ASR` stands for **Arithmetic Shift Right**.

- It shifts the bits of a value to the **right**, preserving the **sign bit** (bit 31).
- Used to perform **signed division by powers of 2**.
- It is different from **logical shift right (LSR)**, which fills with 0

ASR with Immediate Shift Amount

ASR Rd, Rm, #imm

ASR with Register

ASR Rd, Rm, Rs

Shifts value in Rm by the amount specified in Rs.

10b) Write an assembly language to determine the parity of 32 bit number. (6 Marks)

```
    AREA PARITY_CHECK, CODE, READONLY
    ENTRY

    MOV     R0, #0xA5      ; Example lower 8 bits: 10100101
    LSL     R0, R0, #24    ; Make it a full 32-bit value
    MOV     R1, #0x5A      ; Example upper 8 bits: 01011010
    ORR     R0, R0, R1     ; R0 = 0xA500005A

    MOV     R1, #0         ; R1 = parity counter = 0
    MOV     R2, #32        ; Loop counter for 32 bits

PARITY_LOOP
    TST     R0, #1         ; Test LSB
    ADDNE   R1, R1, #1     ; If LSB = 1, increment counter
    LSR     R0, R0, #1     ; Logical shift right
    SUBS    R2, R2, #1     ; Decrement loop counter
    BNE     PARITY_LOOP

    ANDS    R1, R1, #1     ; Check LSB of count → parity
                           ; R1 = 0 → even parity, 1 → odd parity

    B       .              ; In
```

10c) Explain 32-bit instruction with example (8 Marks)

a) ADC
b) BFC
c) LSL
d) PUSH

a) ADC Rd, Rn, Rm

Adds Rn and Rm along with the carry flag (C) from previous operations. Useful for multi-word arithmetic.

```
MOVS   R0, #0xFF

MOVS   R1, #0x01

ADCS   R2, R0, R1    ; R2 = R0 + R1 + Carry
```

b) BFC - Bit Field Clear
BFC Rd, #lsb, #width
Clears (0) a range of bits in a destination register from bit position lsb for width bits.

```
MOV     R0, #0xFFFF
BFC     R0, #4, #4     ; Clears bits [7:4], R0 becomes 0xFF0F
```

c) LSL - Logical Shift Left

LSL Rd, Rm, #n

Shifts bits in register Rm to the left by n positions. Fills zero in the rightmost bits. Used for multiplication by powers of 2.

```
MOV   R1, #0x01

LSL   R2, R1, #3      ; R2 = R1 << 3 = 0x08
```

d) PUSH - Stack Push Multiple Registers

PUSH {Rlist}

```
PUSH {R4-R7, LR}   ; Push R4, R5, R6, R7, and Link Register to stack
```