CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test - II

Sub:	Embedded System Design							Code:	BEC601
Date:	27/05/2025	Duration:	90 mins	Max Marks:	50	Sem:	6th	Branch:	ECE

Answer Any FIVE FULL Questions

		Marks	OBE CO	OBE RBT
1.	Compare (i) DFG and CDFG models with an example.	[10]	CO2	L2

Data Flow Graph/Diagram (DFG) Model

The Data Flow Graph (DFG) model translates the data processing requirements into a data flow graph. The Data Flow Graph (DFG) model is a data driven model in which the program execution is determined by data. This model emphasises on the data and operations on the data which transforms the input data to output data. Indeed Data Flow Graph (DFG) is a visual model in which the operation on the data (process) is represented using a block (circle) and data flow is represented using arrows. An inward arrow to the process (circle) represents input data and an outward arrow from the process (circle) represents output data in DFG notation.

Embedded applications which are computational intensive and data driven are modeled using the DFG model. DSP applications are typical examples for it. Now let's have a look at the implementation of a DFG. Suppose one of the functions in our application contains the computational requirement $x = a + b$; and $y = x - c$. Figure 7.1 illustrates the implementation of a DFG model for implementing these requirements.

In a DFG model, a data path is the data flow path from input to output. A DFG model is said to be acyclic DFG (ADFG) if it doesn't contain multiple values for the input variable and multiple output values for a given set of input(s). Feedback inputs (Output is fed back to Input), events, etc. are examples for non-acyclic inputs. A DFG model translates the program as a single sequential process execution.

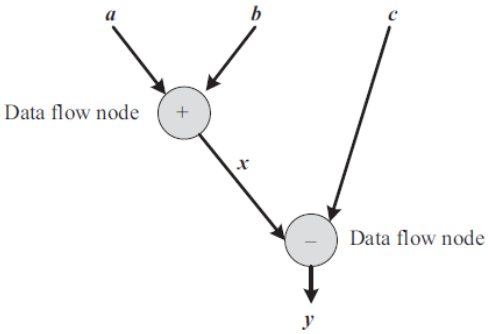


Fig. 7.1 Data flow graph (DFG) model

[4]

7.2.2 Control Data Flow Graph/ Diagram (CDFG)

We have seen that the DFG model is a data driven model in which the execution is controlled by data and it doesn't involve any control operations (conditionals). The Control DFG (CDFG) model is used for modelling applications involving conditional program execution. CDFG models contains both data operations and control operations. The CDFG uses Data Flow Graph (DFG) as element and conditional (constructs) as decision makers. CDFG contains both data flow nodes and decision nodes, whereas DFG contains only data flow nodes. Let us have a look at the implementation of the CDFG for the following requirement.

If flag = 1, $x = a + b$; else $y = a - b$;

This requirement contains a decision making process. The CDFG model for the same is given in Fig. 7.2.

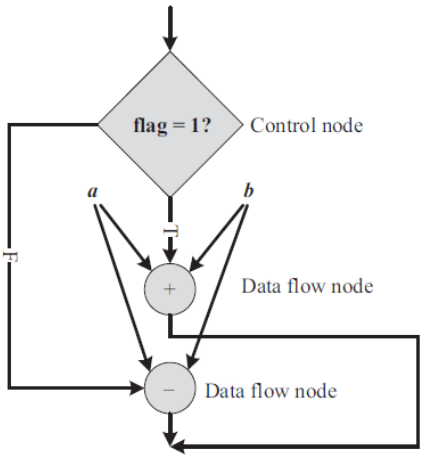


Fig. 7.2 Control Data Flow Graph (CDFG) Model

The control node is represented by a 'Diamond' block which is the decision making element in a normal flow chart based design. CDFG translates the requirement, which is modeled to a concurrent process model. The decision on which process is to be executed is determined by the control node.

A real world example for modelling the embedded application using CDFG is the capturing and saving of the image to a format set by the user in a digital still camera where everything is data driven starting from the Analog Front End which converts the CCD sensor generated analog signal to Digital Signal and the task which stores the data from ADC to a frame buffer for the use of a media processor which performs various operations like, auto correction, white balance adjusting, etc. The decision on, in which format the image is stored (formats like JPEG, TIFF, BMP, etc.) is controlled by the camera settings, configured by the user.

(ii) C v/s Embedded C

	'C' is a well structured, well defined and standardised general purpose programming language with extensive bit manipulation support. 'C' offers a combination of the features of high level language and assembly and helps in hardware access programming (system level programming) as well as business package developments (Application developments like pay roll systems, banking applications, etc). The conventional 'C' language follows ANSI standard and it incorporates various library files for different operating systems. A platform (operating system) specific application, known as, compiler is used for the conversion of programs written in 'C' to the target processor (on which the OS is running) specific binary files. Hence it is a platform specific development. Embedded 'C' can be considered as a subset of conventional 'C' language. Embedded 'C' supports all 'C' instructions and incorporates a few target processor specific functions/instructions. It should be noted that the standard ANSI 'C' library implementation is always tailored to the target processor/controller library files in Embedded 'C'. The implementation of target processor/controller specific functions/instructions depends upon the processor/controller as well as the supported cross-compiler for the particular Embedded 'C' language. A software program called 'Cross-compiler' is used for the conversion of programs written in Embedded 'C' to target processor/controller specific instructions (machine language). (iii) Compiler v/s Cross-Compiler. Compiler is a software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture (e.g. Intel x86/Pentium). Here the operating system, the compiler program and the application making use of the source code run on the same target processor. The source code is converted to the target processor specific machine instructions. The development is platform specific (OS as well as target processor on which the OS is running). Compilers are generally termed as '*Native Compilers*'. A native compiler generates machine code for the same machine (processor) on which it is running. Cross-compilers are the software tools used in cross-platform development applications. In cross-platform development, the compiler running on a particular target processor/OS converts the source code to machine code for a target processor whose architecture and instruction set is different from the processor on which the compiler is running or for an operating system which is different from the current development environment OS. Embedded system development is a typical example for cross-platform development where embedded firmware is developed on a machine with Intel/AMD or any other target processors and the same is converted into machine code for any other target processor architecture (e.g. 8051, PIC, ARM etc). Keil C51 is an example for cross-compiler. The term 'Compiler' is used interchangeably with 'Cross-compiler' in embedded firmware applications. Whenever you see the term 'Compiler' related to any embedded firmware application, it could be referring to a cross-compiler.	[3] [3]		
2.	With the help of a neat diagram, explain the FSM model for automatic seat belt warning system. The State Machine model is used for modelling reactive or event-driven embedded systems whose processing behaviour are dependent on state transitions. Embedded systems used in the control and industrial applications are typical examples for event driven systems. The State Machine model describes the system behaviour with 'States', 'Events', 'Actions' and 'Transitions'. *State* is a representation of a current situation. An *event* is an input to the *state*. The *event* acts as stimuli for state transition. *Transition* is the movement from one state to another. *Action* is an activity to be performed by the state machine. 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.	[10] [3]	CO2	L2

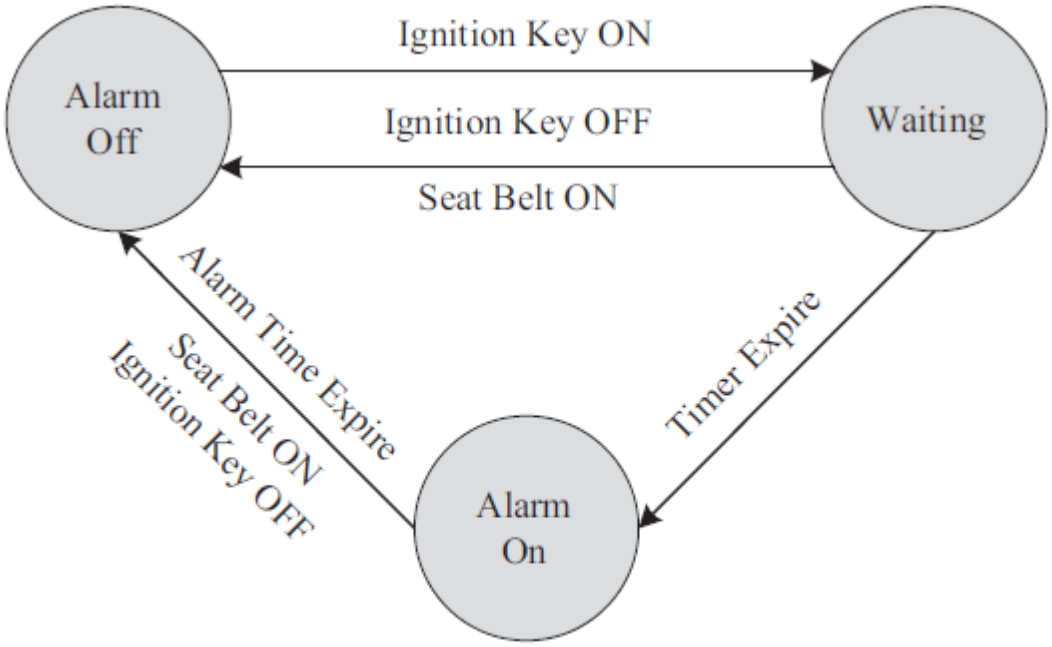


Fig. 7.3 FSM Model for Automatic seat belt warning system

The 'Ignition Key ON' event triggers the 10 second timer and transitions the state to 'Waiting'. If a 'Seat Belt ON' or 'Ignition Key OFF' event occurs during the wait state, the state transitions into 'Alarm Off'.

When the wait timer expires in the waiting state, the event 'Timer Expire' is generated and it transitions the state to 'Alarm On' from the 'Waiting' state. The 'Alarm On' state continues until a 'Seat Belt ON' or 'Ignition Key OFF' event or 'Alarm Time Expire' event, whichever occurs first. The occurrence of any of these events transitions the state to 'Alarm Off'. The wait state is implemented using a timer. The timer also has certain set of states and events for state transitions. Using the FSM model, the timer can be modeled as shown in Fig. 7.4.

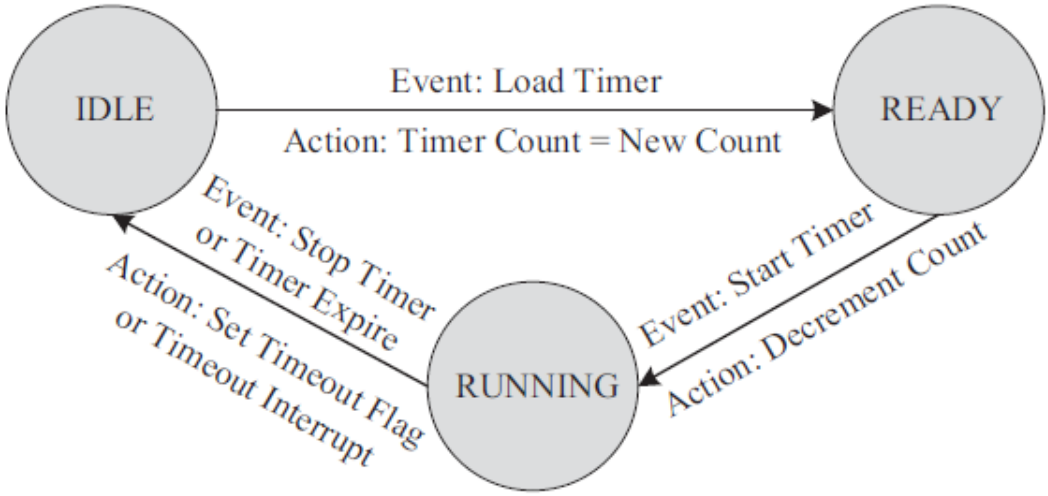


Fig. 7.4 FSM Model for timer

As seen from the FSM, the timer state can be either 'IDLE' or 'READY' or 'RUNNING'. During the normal condition when the timer is not running, it is said to be in the 'IDLE' state. The timer is said to be in the 'READY' state when the timer is loaded with the count corresponding to the required time delay. The timer remains in the 'READY' state until a 'Start Timer' event occurs. The timer changes its state to 'RUNNING' from the 'READY' state on receiving a 'Start Timer' event and remains in the 'RUNNING' state until the timer count expires or a 'Stop Timer' even occurs. The timer state changes to 'IDLE' from 'RUNNING' on receiving a 'Stop Timer' or 'Timer Expire' event.

[4]

[3]

3.	With the help of a neat diagram, explain the operating system architecture.	[10]	CO3	L3

The operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services. The OS manages the system resources and makes them available to the user applications/tasks on a need basis. A normal computing system is a collection of different I/O subsystems, working, and storage memory. The primary functions of an operating system is

LO 1 Understand the basics of an operating system and the need for an operating system

[5]

- Make the system convenient to use
- Organise and manage the system resources efficiently and correctly

Figure 10.1 gives an insight into the basic components of an operating system and their interfaces with rest of the world.

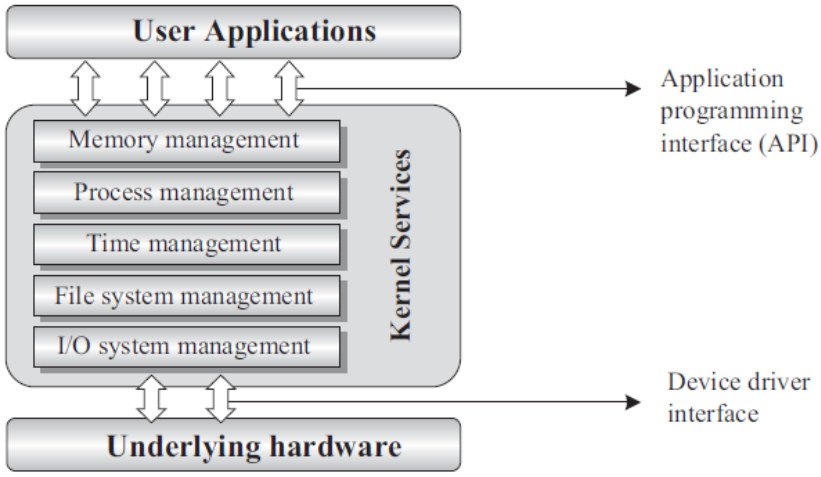


Fig. 10.1 The Operating System Architecture

10.1.1 The Kernel

The kernel is the core of the operating system and is responsible for managing the system resources and the communication among the hardware and other system services. Kernel acts as the abstraction layer between system resources and user applications. Kernel contains a set of system libraries and services. For a general purpose OS, the kernel contains different services for handling the following.

[3]

Process Management Process management deals with managing the processes/tasks. Process management includes setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, scheduling and managing the execution of the process, setting up and managing the Process Control Block (PCB), Inter Process Communication and synchronisation, process termination/ deletion, etc. We will look into the description of process and process management in a later section of this chapter.

Primary Memory Management The term primary memory refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored. The Memory Management Unit (MMU) of the kernel is responsible for

- Keeping track of which part of the memory area is currently used by which process
- Allocating and De-allocating memory space on a need basis (Dynamic memory allocation).

File System Management File is a collection of related information. A file could be a program (source code or executable), text files, image files, word documents, audio/video files, etc. Each of these files differ in the kind of information they hold and the way in which the information is stored. The file operation is a useful service provided by the OS. The file system management service of Kernel is responsible for

- The creation, deletion and alteration of files
- Creation, deletion and alteration of directories
- Saving of files in the secondary storage memory (e.g. Hard disk storage)
- Providing automatic allocation of file space based on the amount of free space available
- Providing a flexible naming convention for the files

The various file system management operations are OS dependent. For example, the kernel of Microsoft® DOS OS supports a specific set of file system management operations and they are not the same as the file system operations supported by UNIX Kernel.

I/O System (Device) Management Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system. In a well-structured OS, the direct accessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the kernel. The kernel maintains a list of all the I/O devices of the system. This list may be available in advance, at the time of building the kernel. Some kernels, dynamically updates the list of available devices as and when a new device is installed (e.g. Windows NT kernel keeps the list updated when a new plug '*n*' play USB device is attached to the system). The service 'Device Manager' (Name may vary across different OS kernels) of the kernel is responsible for handling all I/O device related operations. The kernel talks to the I/O device through a set of low-level systems calls, which are implemented in a service, called device drivers. The device drivers are specific to a device or a class of devices. The Device Manager is responsible for

- Loading and unloading of device drivers
- Exchanging information and the system specific control signals to and from the device

Secondary Storage Management The secondary storage management deals with managing the secondary storage memory devices, if any, connected to the system. Secondary memory is used as backup medium for programs and data since the main memory is volatile. In most of the systems, the secondary storage is kept in disks (Hard Disk). The secondary storage management service of kernel deals with

- Disk storage allocation
- Disk scheduling (Time interval at which the disk is activated to backup data)
- Free Disk space management

Protection Systems Most of the modern operating systems are designed in such a way to support multiple users with different levels of access permissions (e.g. Windows 10 with user permissions like 'Administrator', 'Standard', 'Restricted', etc.). Protection deals with implementing the security policies to restrict the access to both user and system resources by different applications or processes or users. In multiuser supported operating systems, one user may not be allowed to view or modify the whole/portions of another user's data or profile details. In addition, some application may not be granted with permission to make use of some of the system resources. This kind of protection is provided by the protection services running within the kernel.

Interrupt Handler Kernel provides handler mechanism for all external/internal interrupts generated by the system.

4.	What is a process/task? Explain with the help of a neat diagram the structure and memory organization of a process.	[10]	CO3	L2

A '*Process*' is a program, or part of it, in execution. Process is also known as an instance of a program in execution. Multiple instances of the same program can execute simultaneously. A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange, etc. A process is sequential in execution.

The concept of '*Process*' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilisation of the CPU and other system resources. Concurrent execution is achieved through the sharing of CPU among the processes. A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process. This can be visualised as shown in Fig. 10.4. [3]

A process which inherits all the properties of the CPU can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor. When the process gets its turn, its registers and the program counter register becomes mapped to the physical registers of the CPU. From a memory perspective, the memory occupied by the *process* is segregated into three regions, namely, Stack memory, Data memory and Code memory (Fig. 10.5).

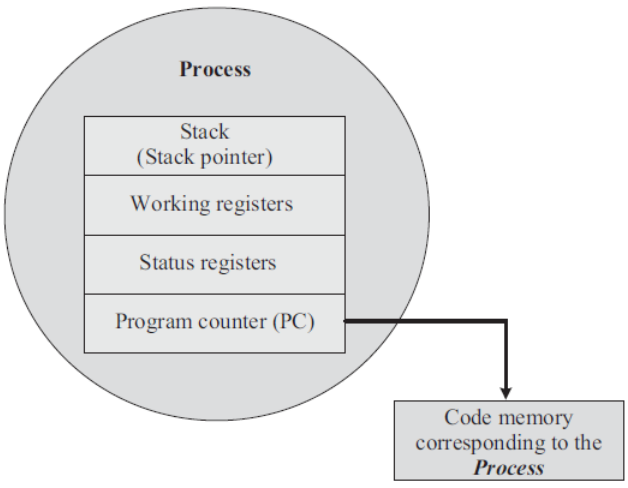


Fig. 10.4 Structure of a Process

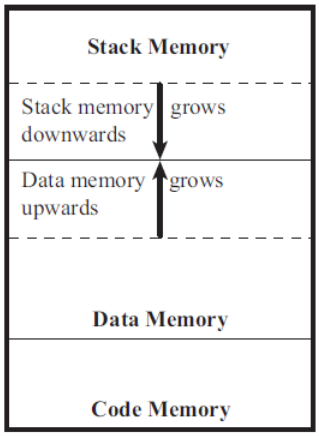
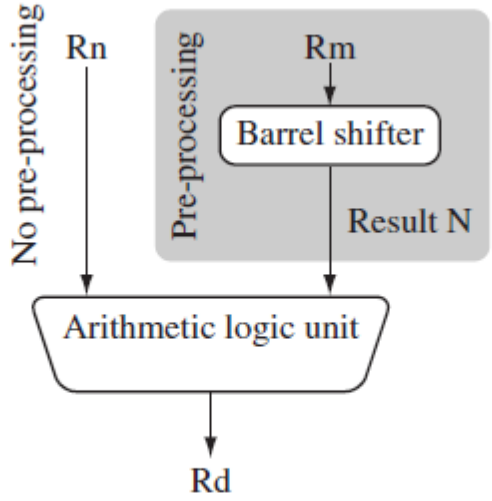


Fig. 10.5 Memory organisation of a Process

[4]

	The 'Stack' memory holds all temporary data such as variables local to the process. Data memory holds all global data for the process. The code memory contains the program code (instructions) corresponding to the process. On loading a process into the main memory, a specific area of memory is allocated for the process. The stack memory usually starts (OS Kernel implementation dependent) at the highest memory address from the memory area allocated for the process. Say for example, the memory map of the memory area allocated for the process is 2048 to 2100, the stack memory starts at address 2100 and grows downwards to accommodate the variables local to the process.	[3]		
5.	What is a barrel shifter with respect to an ARM processor? Explain with the help of a neat diagram and example. **BARREL SHIFTER** In Example 3.1 we showed a MOV instruction where *N* is a simple register. But *N* can be more than just a register or immediate value; it can also be a register *Rm* that has been preprocessed by the barrel shifter prior to being used by a data processing instruction. Data processing instructions are processed within the arithmetic logic unit (ALU). A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU. This shift increases the power and flexibility of many data processing operations. There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions. Pre-processing or shift occurs within the cycle time of the instruction. This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.	[10] [3]	CO4	L2
	 Figure 3.1 Barrel shifter and ALU. To illustrate the barrel shifter we will take the example in Figure 3.1 and add a shift operation to the move instruction example. Register *Rn* enters the ALU without any pre-processing of registers. Figure 3.1 shows the data flow between the ALU and the barrel shifter.	[3]		

	EXAMPLE 3.1 This example shows a simple move instruction. The MOV instruction takes the contents of register *r5* and copies them into register *r7*, in this case, taking the value 5, and overwriting the value 8 in register *r7*.
	`PRE r5 = 5`
	`r7 = 8`
	`MOV r7, r5 ; let r7 = r5`
	`POST r5 = 5`
	`r7 = 5`
		[4]		

Table 3.2 Barrel shifter operations.

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*	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	(c flag $\ll 31$) | ((unsigned)$x \gg 1$)	none

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

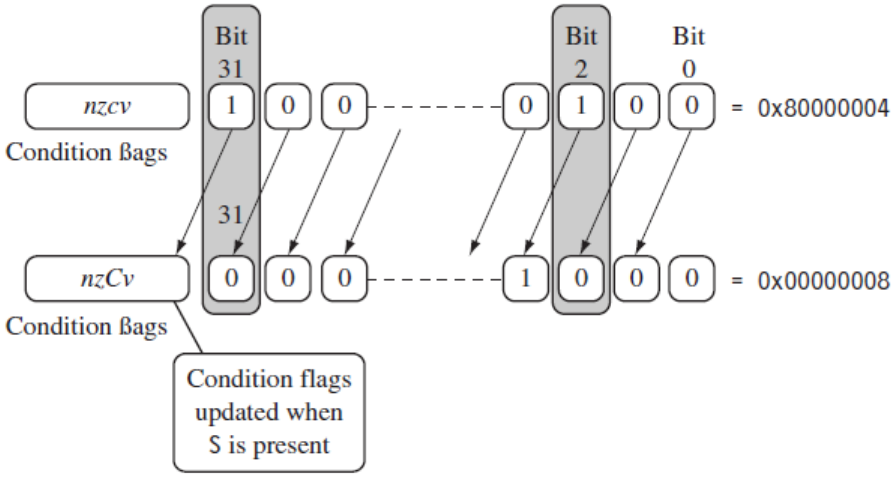


Figure 3.2 Logical shift left by one.

Table 3.3 Barrel shift operation syntax for data processing instructions.

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`

6.	List the different registers of ARM CORTEX – M3 and mention their use. Explain the use of R13, R14 and R15 registers. **REGISTERS** General-purpose registers hold either data or an address. They are identified with the letter *r* prefixed to the register number. For example, register 4 is given the label *r4*. Figure 2.2 shows the active registers available in *user* mode—a protected mode normally	[10] [3]	CO4	L2

used when executing applications. The processor can operate in seven different modes, which we will introduce shortly. All the registers shown are 32 bits in size.

There are up to 18 active registers: 16 data registers and 2 processor status registers. The data registers are visible to the programmer as *r0* to *r15*.

The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*. They are frequently given different labels to differentiate them from the other registers.

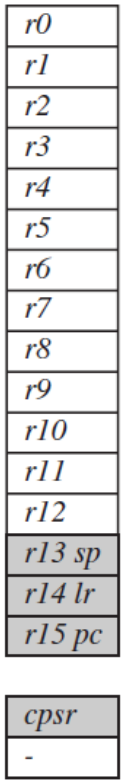


Figure 2.2 Registers available in *user* mode.

In Figure 2.2, the shaded registers identify the assigned special-purpose registers:

- Register *r13* is traditionally used as the stack pointer (*sp*) and stores the head of the stack in the current processor mode.
- Register *r14* is called the link register (*lr*) and is where the core puts the return address whenever it calls a subroutine.
- Register *r15* is the program counter (*pc*) and contains the address of the next instruction to be fetched by the processor.

Depending upon the context, registers *r13* and *r14* can also be used as general-purpose registers, which can be particularly useful since these registers are banked during a processor mode change. However, it is dangerous to use *r13* as a general register when the processor is running any form of operating system because operating systems often assume that *r13* always points to a valid stack frame.

In ARM state the registers *r0* to *r13* are *orthogonal*—any instruction that you can apply to *r0* you can equally well apply to any of the other registers. However, there are instructions that treat *r14* and *r15* in a special way.

In addition to the 16 data registers, there are two program status registers: *cpsr* and *spsr* (the current and saved program status registers, respectively).

The register file contains all the registers available to a programmer. Which registers are visible to the programmer depend upon the current mode of the processor.

[4]

[3]

7.	Explain the following ARM instructions with an example with pre and post execution conditions: a) SUBS r1,r1,#1	[10]	CO5	L2

The SUBS instruction is useful for decrementing loop counters. In this example we subtract the immediate value one from the value one stored in register *r1*. The result value zero is written to register *r1*. The *cpsr* is updated with the *ZC* flags being set. [3]

```
PRE     cpsr = nzcvqiFt_USER
        r1 = 0x00000001

POST    cpsr = nZCvqiFt_USER
        r1 = 0x00000000
```

b) RSB r0,r1,#0

This reverse subtract instruction (RSB) subtracts *r1* from the constant value #0, writing the result to *r0*. You can use this instruction to negate numbers. [3]

```
PRE     r0 = 0x00000000
        r1 = 0x00000077

        RSB r0, r1, #0      ; Rd = 0x0 - r1

POST    r0 = -r1 = 0xffffff89
```

c) BIC r0,r1,r2

This example shows a more complicated logical instruction called BIC, which carri a logical bit clear. [4]

```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC   r0, r1, r2

POST    r0 = 0b1010
```

This is equivalent to

```
        Rd = Rn AND NOT(N)
```

In this example, register *r2* contains a binary pattern where every binary 1 in *r2* clears a corresponding bit location in register *r1*. This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the *cpsr*.

The logical instructions update the *cpsr* flags only if the S suffix is present. These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.