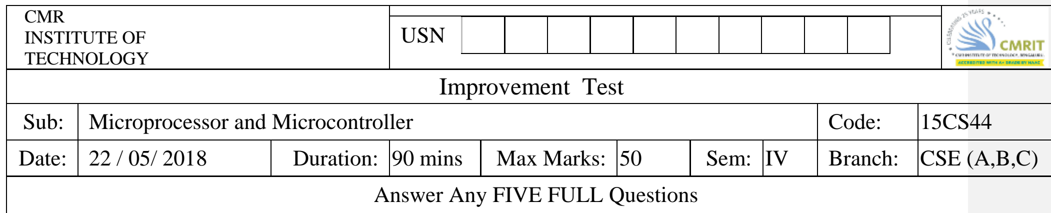

CMR INSTITUTE OF TECHNOLOGY		USN								
Improvement Test										
Sub:	Microprocessor and Microcontroller							Code:	15CS44	
Date:	22 / 05/ 2018	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	CSE (A,B,C)	
Answer Any FIVE FULL Questions										

		Marks	OBE CO	OBE RBT
1	Explain the architecture of ARM based embedded system hardware and software with the help of suitable diagrams.	[10]	CO4	L1
2	(a) Explain the STACK operations in ARM7. Describe different addressing methods for stack operations. Give examples.	[05]	CO4	L2
	(b) Write an assembly language code which uses BL instruction to call a subroutine to perform addition of three data words stored in consecutive memory locations. Specify the return statement with in the body of subroutine.	[05]	CO4	L3
3	(a) What is CPSR? Explain relevant bits for ARM7. (b) Explain in detail the processor modes available for ARM7.	[5+5]	CO4	L1
4	Write note on Core extensions for ARM processor.	[10]	CO4	L1
5	(a) Explain the various syntax for barrel shifter data processing instruction of ARM?	[05]	CO4	L2
	(b) Explain ARM7 single register load and store instructions with relevant examples indicating Pre and post execution conditions.	[05]	CO4	L3
6	(a) Explain different types of program status register instructions with syntax. Give example.	[05]	CO4	L2
	(b) What is SWI? Explain with proper syntax and an example for interrupt handler.	[05]	CO4	L2
7	(a) Explain following instruction with suitable example. 1. UMULL 2. RSB 3. RRx	[06]	CO4	L2
	(b) Develop an ALP to copy a block of data from one location to other using ARM instructions.	[04]	CO4	L3

1. Explain the architecture of ARM based embedded system hardware and software with the help of suitable diagrams.

Embedded systems can control many different devices, from small sensors found on a production line, to the real-time control systems used on a NASA space probe. All these devices use a combination of software and hardware components. Each component is chosen for efficiency and, if applicable, is designed for future extension and expansion.

Embedded System Hardware:

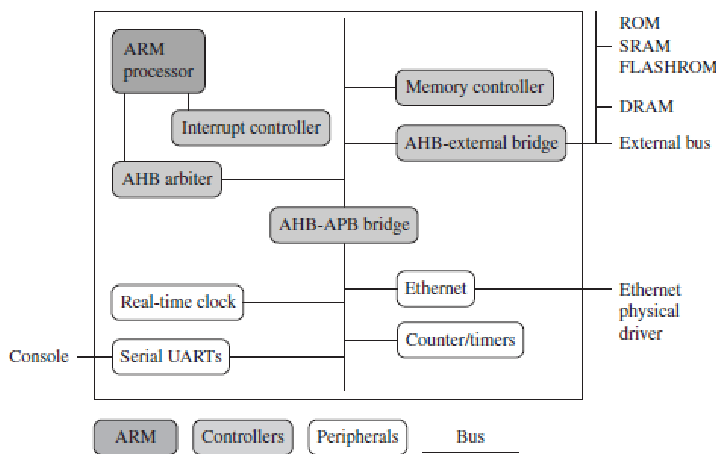

Figure 1.1 ARM based embedded device

Figure1.1 shows a typical embedded device based on an ARM core. We can separate the device into four main hardware components:

1. The ARM processor controls the embedded device. Different versions of the ARM processor are available to suit the desired operating characteristics. An ARM processor comprises a core (the execution engine that processes instructions and manipulates data) plus the surrounding components that interface it with a bus. These components can include memory management and caches.
2. Controllers coordinate important functional blocks of the system. Two commonly found controllers are interrupt and memory controllers.
3. The peripherals provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.

4. A bus is used to communicate between different parts of the device.

ARM Bus Technology: embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.

There are two different classes of devices attached to the bus. The ARM processor core is a bus master—a logical device capable of initiating a data transfer with another device across the same bus. Peripherals tend to be bus slaves—logical devices capable only of responding to a transfer request from a bus master device.

- AMBA Bus Protocol: The Advanced Microcontroller Bus Architecture (AMBA) has been widely adopted as the on-chip bus architecture used for ARM processors. The first AMBA buses introduced were the ARM System Bus (ASB) and the ARM Peripheral Bus (APB). Later ARM introduced another bus design, called the ARM High Performance Bus (AHB).
- Using AMBA, peripheral designers can reuse the same design on multiple projects. A peripheral can simply be bolted on to the on-chip bus without having to redesign an interface for each different processor architecture.
- ASB is a bidirectional bus design.
- APB is used with slower peripherals.
- AHB is based on a centralized multiplexed bus scheme, thus runs at higher clock speeds and provides higher data through put. AHB bus is used for the high performance peripherals.

Memory:

An embedded system has to have some form of memory to store and execute code. Cost, performance, and power consumption are the parameters considered while deciding upon specific memory characteristics, such as hierarchy, width, and type. Like if memory has to run twice as fast to maintain a desired bandwidth, then the memory power requirement may be higher.

- Hierarchy: Memory can be Cache, Main memory or Secondary memory.

The fastest memory cache is physically located nearer the ARM processor core and the slowest secondary memory is set further away. Generally the closer memory is to the processor core, the more it costs and the smaller its capacity. The cache is placed between main memory and the core. It is used to speed up data transfer between the processor and main memory. The main memory is large and is generally stored in separate chips. Load and store instructions access the main memory unless the values have been stored in the cache for fast access. Secondary storage is the largest and slowest form of memory. Hard disk drives and CD-ROM drives are examples of secondary storage. Many small embedded systems do not require the performance benefits of a cache.

- Width: The memory width is the number of bits the memory returns on each access—typically 8, 16, 32, or 64 bits. The memory width has a direct effect on the overall performance and cost ratio. If you have an un-cached system using 32-bit ARM instructions and 16-bit-wide memory chips, then the processor will have to make two memory fetches per instruction.

Each fetch requires two 16-bit loads. This obviously has the effect of reducing system performance, but the benefit is that 16-bit memory is less expensive. In contrast, if the core executes 16-bit Thumb instructions, it will achieve better performance with a 16-bit memory. The higher performance is a result of the core making only a single fetch to memory to load an instruction. Hence, using Thumb instructions with 16-bit-wide memory devices provides both improved performance and reduced cost.

- Types: RAM or ROM
    - Read only memory (ROM) is the least flexible of all memory types because it contains an image that is permanently set at production time and cannot be reprogrammed. ROMs are used in high-volume devices that require no updates or corrections. Many devices also use a ROM to hold boot code.
    - Random Access memory (RAM)- SRAM,DRAM or SDRAM

Peripherals

Embedded systems that interact with the outside world need some form of peripheral device. A peripheral device performs input and output functions for the chip by connecting to other devices that are off-chip.

All ARM peripherals are memory mapped—the programming interface is a set of memory-addressed registers. The address of these registers is an offset from a specific peripheral base address.

Controllers are specialized peripherals that implement higher levels of functionality within an embedded system. Two important types of controllers are memory controllers and interrupt controllers. Memory controllers connect different types of memory to the processor bus. On power-on a memory controller is configured in hardware to allow certain memory devices to be active. These memory devices allow the initialization code to be executed. Some memory devices must be set up by software.

An interrupt controller provides a programmable governing policy that allows software to determine which peripheral or device can interrupt the processor at any specific time by setting the appropriate bits in the interrupt controller registers. There are two types of interrupt controller available for the ARM processor: the standard interrupt controller and the vector interrupt controller (VIC). The standard interrupt controller sends an interrupt signal to the processor core when an external device requests servicing. It can be programmed to ignore or mask an individual device or set of devices. The VIC is more powerful than the standard interrupt controller because it prioritizes interrupts and simplifies the determination of which device caused the interrupt.

Embedded System Software

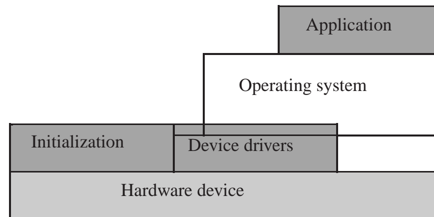

Figure 1.2 Embedded system software

An embedded system needs software to drive it. Figure 1.2 shows four typical software components required to control an embedded device. Each software component in the stack uses a higher level of abstraction to separate the code from the hardware device. The software components can run from ROM or RAM. ROM code that is fixed on the device (for example, the initialization code) is called firmware.

- The **initialization code** is the first code executed on the board and is specific to a particular target. It sets up the minimum parts of the board before handing control over to the operating system. Initialization code (or boot code) takes the processor from the reset state to a state where the operating system can run. It usually configures the memory controller and processor caches and initializes some devices. In a simple system the operating system might be replaced by a simple scheduler or debug monitor. The initialization code handles a number of administrative tasks prior to handing control over to an operating system image: initial hardware configuration, diagnostics, and booting.
- The **operating system** provides an infrastructure to control applications and manage hardware system resources. Many embedded systems do not require a full operating system but merely a simple task scheduler that is either event or poll driven. An operating system organizes the system resources: the peripherals, memory, and processing time. With an operating system controlling these resources, they can be efficiently used by different applications running with in the operating system environment.
    - Operating systems may be divided into two main categories: real-time operating systems (RTOSs) and platform operating systems. RTOSs provide guaranteed response times to events. Platform operating systems require a memory management unit to manage large, non-real-time applications and tend to have secondary storage. These two categories of

operating system are not mutually exclusive, there are operating systems that use an ARM core with a memory management unit and have real-time characteristics.

- The **device drivers** provide a consistent software interface to the peripherals on the hardware device.
- An **application** performs one of the tasks required for a device. There may be multiple applications running on the same device, controlled by the operating system.

2. (a) Explain the STACK operations in ARM7. Describe different addressing methods for stack operations. Give examples.

- Stacks are highly flexible in the ARM architecture, since the implementation is completely left to the software.
- Stack Instructions. The ARM instruction set does not contain any stack specific instructions like push and pop. The instruction set also does not enforce in anyway the use of a stack. Push and pop operations are performed by memory access instructions, with auto-increment addressing modes.
- Stack Pointer: In ARM7 environment the register r13 is used as stack pointer.
- Programmer can use the LDM and STM instructions to implement pop and push operations respectively and use a suffix to indicate the stack type.

Stack Types:

Since it is left to the software to implement a stack, different implementation choices result different types of stacks. There are two types of stack depending on how the stack grows.

Ascending stack

In a push operation the stack pointer is incremented, i.e the stack grows towards higher address.

Descending stack

In a push operation the stack pointer is decremented, i.e the stack grows towards lower address.

There are two types of stack depending on what the stack pointer points to.

Empty stack

Stack pointer points to the location in which the next item will be stored. A push will store the value, and increment the stack pointer.

Full stack

Stack pointer points to the location in which the last item was stored. A push will increment the stack pointer and store the value.

- Four different stacks are possible - full-ascending, full-descending, empty-ascending, empty-descending. All 4 can be implemented using the register load store instructions.
- To make it easier for the programmer, stack-oriented suffixes can be used instead of the increment or decrement, and before or after suffixes. The following table shows the stack oriented suffixes and their equivalent addressing mode suffixes for load and store instructions

Stack-oriented suffixes and equivalent addressing mode suffixes:

Stack-oriented suffix	For store or push instructions	For load or pop instructions
FD (Full Descending stack)	DB (Decrement Before)	IA (Increment After)
FA (Full Ascending stack)	IB (Increment Before)	DA (Decrement After)
ED (Empty Descending stack)	DA (Decrement After)	IB (Increment Before)
EA (Empty Ascending stack)	IA (Increment After)	DB (Decrement Before)

Suffixes for load and store multiple instructions

Stack type	Store	Load
Full descending	STMFD (STMDB, Decrement Before)	LDMFD (LDMIA, increment after)
Full ascending	STMFA (STMIB, Increment Before)	LDMFA (LDMDA, Decrement After)
Empty descending	STMED (STMDA, Decrement After)	LDMED (LDMIB, Increment Before)
Empty ascending	STMEA (STM, increment after)	LDMEA (LDMDB, Decrement Before)

2 ***(b) Write an assembly language code which uses BL instruction to call a subroutine to perform addition of three data words stored in consecutive memory locations. Specify the return statement with in the body of subroutine.***

```
        area pgm1,code,readonly
entry
main
        ldr r0,=n
```

```
        mov r1,#5
        mov r3,#0
        bl subadd
        ldr r7,=sum
        str r3,[r7]
stop b stop
subadd
back ldr r2,[r0],#4
        add r3,r3,r2
        subs r1,r1,#1
        bne back
        mov pc,lr
n        dcd 0x6,0x7,0x8,0x9,0x10
        area pgm2,data,readwrite
sum space 4
        end
```

3. (a) What is CPSR? Explain relevant bits

- The CPSR is a dedicated 32-bit register which resides in the register file.
- The ARM core uses the CPSR to monitor and control internal operations.

The Figure 3a.1 shows the CPSR layout.

Fig 3a.1 CPSR Layout

The CPSR is divided into four fields, each 8bits wide: **flags, status, extension and control**. In current designs the extension and status fields are reserved for future use.

Control Field: The control field contains the processor mode, state, and interrupt mask bits.

Processor Modes: The processor mode determines which registers are active and the access rights to the CPSR register itself. The least significant 5 bits in the CPSR determines the mode.

Each processor mode is either privileged or non-privileged.

- A privileged mode allows full read-write access to the CPSR. Conversely, a non-privileged 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 non-privileged mode (user).
- The processor can change mode by either writing directly in to the control field (b0-b4) when it is in a privileged mode or when exceptions or interrupts happens.

State and Instruction Sets: The state of the core determines which instruction set is being executed. There are three instruction sets: ARM, Thumb, and Jazelle.

- The ARM instruction set is only active when the processor is in ARM state. Similarly the Thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions. Jazelle executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java bytecodes.
- The Jazelle J (bit 24, which falls in flag field) and Thumb T bits in the CPSR reflect the state of the processor.
- When both J and T bits are 0, the processor is in ARM state and executes ARM instructions. If J=1 then the core is in Jazelle state and T=1 then the core is in Thumb state.

Interrupt masks: ARM7 entertains two kinds of hardware interrupts interrupt request (IRQ) and fast interrupt request (FIQ). Bit 6 and Bit 7 of CPSR is used to mask these interrupt requests.

- If I=1 then IRQ is disabled and if F=1 FRQ is disabled.
- When processor mode changes the exception or interrupt handler makes IRQ bit 1 to disable further interrupt requests.

Flags: The flags field contains the condition flags.

Some ARM processor cores have extra bits allocated. For example, the J bit (24), which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8 bit java code.

The bits are described as given below along with condition to set the bits.

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

Q (bit 27)-Saturation: the result causes an overflow and/or saturation when extended instructions are used. eg: QADD

3 (b) Explain in detail the processor modes available for ARM7.

Processor Modes: The processor mode determines which registers are active and the access rights to the CPSR register itself. The least significant 5 bits in the CPSR determines the mode. Refer table 3b.1

Each processor mode is either privileged or non-privileged.

- A privileged mode allows full read-write access to the CPSR. Conversely, a non-privileged 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 non-privileged mode (user).
- The processor can change mode by either writing directly in to the control field (b0-b4) when it is in a privileged mode or when exceptions or interrupts happens.
- The following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*. Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

Table 3b.1 Processor mode selection bits

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.
- *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 when the processor encounters an instruction that is undefined or not supported by the implementation.
- *User* mode is used for programs and applications.

Banked Registers

All processor modes except *system* mode have a set of associated banked registers that are a subset of the main 16 registers. A banked register maps one-to one onto a *user* mode register. If the processor mode changes change processor mode, a banked register from the new mode will replace an existing register.

Banked registers of a particular mode are denoted by an underline character post-fixed to the mode Mnemonic or *_mode*.

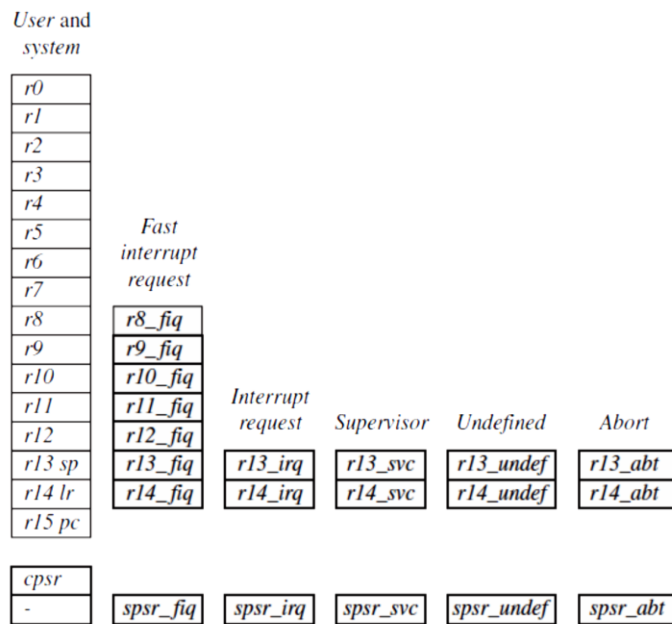

Fig 3b.1 banked registers

Figure 3b.2 shows all 37 registers in the register file. Of those, 20 registers are hidden from a program at different times. These registers are called *banked registers.* They are available only when the processor is in a particular mode.
For example, when the processor is in the interrupt request mode, the instructions user execute still access registers named r13 and r14. However, these registers are the banked registers r13_irq and r14_irq. The user mode registers r13_usr and r14_usr are not affected by the instruction referencing these registers. A program still has normal access to the other registers r0 to r12.

- The r14_irq contains the return address and r13_irq contains the stack pointer for interrupt request mode, the *cpsr_usr* will be copied into *spsr_irq*.

- To return back to user mode, a special return instruction is used that instructs the core to restore the original cpsr from the spsr_irq and bank in the user registers r13 and r14.

- Another important feature is that the cpsr is not copied into the spsr when a mode change is forced due to a program writing directly to the cpsr. The saving of the cpsr only occurs when an exception or interrupt is raised.

4. Write note on Core extensions for ARM processor.
5. (a)Explain the various syntax for barrel shifter data processing instruction of ARM?

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.

The barrel shifter can perform the following types of operation:

LSL |shift left by n bits;

LSR |logical shift right by n bits;

ASR |arithmetic shift right by n bits (the bits fed
|into the top end of the operand are copies of the
|original top (or sign) bit);

ROR |rotate right by n bits;

RRX |rotate right extended by 1 bit. This is a 33 bit
|rotate, where the 33rd bit is the PSR C flag.

Commented [JM1]:

Various syntax available are given as below:

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

Example:

PRE
cpsr = nzcvqiFt_USER
r0 = 0x00000000
r1 = 0x80000004
MOVS r0, r1, LSL #1
POST
cpsr = nzCvqiFt_USER
r0 = 0x00000008
r1 = 0x80000004

(b) Explain ARM7 single register load and store instructions with relevant examples indicating Pre and post execution conditions.

Syntax:

<LDR|STR> {<cond>}{<size>} Rd, <address>

- The basic load and store instructions are:
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB
- ARM Architecture also support half-words and signed data.
  - Load and Store Half-word
    - LDRH / STRH

- Load Signed Byte or Half-word: *load value and sign extend it to 32 bits.*
  - LDRSB / LDRSH

<LDR|STR>{<cond>}{B} Rd,addressing1
LDR{<cond>}SB|H|SH Rd, addressing2
STR{<cond>}H Rd, addressing2

Examples:

STR r0, [r1] ; Store contents of r0 to location pointed to ; by contents of r1.
LDR r2, [r1] ; Load r2 with contents of memory location ; pointed to by contents of r1.

Load and store instructions support different addressing modes.

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The base register is unchanged. The assembly language syntax for this mode is:

Rn, offset

Pre-indexed addressing:

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register. The assembly language syntax for this mode is:
[***Rn, offset***] - Pre-indexed without write back
[***Rn, offset***]! - Pre-indexed with write back

Post-indexed addressing:

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register. The assembly language syntax for this mode is:
[***Rn***]***, offset***

In each case, Rn is the base register and offset can be:

- An immediate constant.
- An index register, Rm.
- A shifted index register, such as Rm, LSL #shift.
  - In addressing mode 1/; all the above offset modes are permitted with immediate offset limited to 12bit information
  - In addressing mode 2/; The first two offset modes are permitted with immediate offset limited to 8bit information

PRE r0 = 0x00000000
r1 = 0x00090000
mem32[0x00009000] = 0x01010101

mem32[0x00009004] = 0x02020202
LDR r0, [r1, #4]!
Preindexing with writeback:
POST(1) r0 = 0x02020202
r1 = 0x00009004

6. (a) Explain different types of program status register instructions with syntax. Give example.

The ARM instruction set provides two instructions to directly control a program status register (psr), Read or write on to PSR

- The MRS instruction transfers the contents of either the cpsr or spsr into a register
- The MSR instruction transfers the contents of a register into the cpsr or spsr.

Syntax:

1. MRS{<cond>} Rd,<cpsr|spsr> : MRS copy program status register to a general-purpose register, Rd=psr
2. MSR{<cond>} <cpsr|spsr>_<fields>, Rm : MSR move a general-purpose register to a program status register, psr[field]=Rm
3. MSR{<cond>} <cpsr|spsr>_<fields>, #immediate : MSR move an immediate value to a program status register , psr[field]=immediate

Fields: can be any combination of control (c), extension (x), status (s), and flags (f). All fields can be edited in SUPERVISOR mode. Only flag field can be edited in USER mode (even though user mode can read all fields)

Example: Write a code to enable IRQ interrupts by clearing the mask bit I.

PRE- Conditions:

cpsr = nzcvqIFt_SVC ; I,F=1; T=0, all flag bits zero

MRS R1, CPSR ; Read CPSR in to r1 register

BIC R1, R1, #0X80 ; IRQ mask bit is B7 of CPSR, Bitwise clear B7.

MSR CPSR_C, R1 ; write back into CPSR condition field with r1 data

POST Execution Condition:

cpsr = nzcvqiFt_SVC ; F=1,I=0,T=0, all flag bits zero

6(b) What is SWI? Explain with proper syntax and an example for interrupt handler.

A software interrupt instruction (SWI) causes a software interrupt exception. It provides a mechanism for applications to call operating system routines, like

- Read or write operation on hard disc
- Parallel port printing
- Invoke Serial or parallel communication

SWIs allow the Operating System to have a modular structure, which means that the code required to create a complete operating system can be split up into a number of small parts (modules) and a module handler. When the SWI handler gets a request for a particular routine number it finds the position of the routine and executes it, passing any data.

No IRQ request is entertained while executing SWI instruction.

When the processor executes an SWI instruction, it sets the program counter pc to the offset 0x8 in the vector table. The instruction forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.

Syntax: SWI {<cond>} SWI_number

SWI number is used to represent a particular function call or feature. The SWI number is determined by

SWI_Number = <SWI instruction> AND NOT (0xff000000); In the SWI instruction opcode the MSB two nibbles correspond to SWI and the rest to the SWI_ Number.

A code called the SWI handler is required to process the SWI call. The handler fetches SWI opcode using the address of the executed SWI instruction, which is calculated from the link register content, to obtain the SWI number.

<u>On execution of SWI, the following updates take place:</u>

LR_SVC=address of instruction following the SWI

SPSR_SVC=CPSR ; Putting the processor into Supervisor mode switches out 2 registers r13 and r14 and replaces these with r13_svc and r14_svc.

PC= IVT_Base address+0x8

CPSR mode=SVC

CPSR I=1 (mask IRQ interrupts)

SWI Handler Implementation example

1. Store r0-r12 and LR on to stack
2. It subtracts 4 from r14 to obtain the address of the SWI instruction.
3. Loads the instruction into a register.
4. Clears the most significant 8 bits of the instruction, getting rid of the Opcode and giving just the SWI number.
5. Uses this value to find to address of the routine of the code to be executing (using lookup tables etc.).
6. Restore the registers r0-r12 and PC
7. Takes the processor out of Supervisor mode.
8. Jumps to the address of the routine.

The following code fragment determines what SWI number is being called and places that number into register r10. The load instruction first copies the complete SWI instruction into register r0. The BIC instruction masks off the top bits of the instruction, leaving the SWI number.

SWI_handler

```
STMFD sp!, {r0-r12, lr} ; Store registers r0-r12 and the link register

LDR r10, [lr, # -4] ; Read the SWI instruction into r10

BIC r10, r10, #0xff000000; Mask off top 8 bit, r10 - contains the SWI number

BL  service_routine ;Branch to corresponding OS service routine using
SWI_Number

LDMFD sp!, {r0-r12, pc}ˆ ; return from SWI handler
```

7. (a)Explain following instruction with suitable example.

1. UMULL 2. RSB 3. RRx

1.UMULL

Unsigned and signed long multiply: 32-bit by 32-bit, 64-bit accumulate or result.

Syntax

UMULL{*cond*}{S} *RdLo*, *RdHi*, *Rm*, *Rs*

RdLo, RdHi

are ARM registers for the result.

Rm, Rs

are ARM registers holding the operands.

- ***RdLo*, *RdHi*, and *Rm* must all be different registers.**
- The UMULL instruction interprets the values from *Rm* and *Rs* as unsigned integers.
- It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

PRE
r0 = 0x00000000
r1 = 0x00000000
r2 = 0xf0000002
r3 = 0x00000002
UMULL r0, r1, r2, r3 ; [r1,r0] = r2*r3
POST
r0 = 0xe0000004 ; = RdLo
r1 = 0x00000001 ; = RdHi

2. RSB
Syntax

RSB{*cond*}{S} *Rd*, *Rn*, *N ;* RSB reverse subtract of two 32-bit values *Rd = N – Rn*

- *N can be register, Immediate data or Barrel shifted register*

PRE
r0 = 0x00000000
r1 = 0x00000077
RSB r0, r1, #0 ; Rd = 0x0 - r1
POST
r0 = -r1 = 0xffffff89

3. RRx
Syntax
Rm, RRX

Rotate right extended by 1 bit. This is a 33 bit rotate, where the 33rd bit is the PSR C flag.
PRE
cpsr = nzCvqIFt_USR
r0 = 0x00000000
r1 = 0x00000000
MOV r0, r1,RRX ; r0= r1,RRX
POST

cpsr = nzcvqIFt_USR
r0 = 0x10000000

(b)Develop an ALP to copy a block of data from one location to other using ARM instructions.

```
         area pgm1,code,readonly
entry
main
        ldr r0,=src
        ldmia r0!,{r2-r6}
        ldr r7,=dest
        stmia r7!,{r2-r6}
stop    b stop
src     dcd 0x6,0x7,0x8,0x9,0x10
        area pgm2,data,readwrite
dest    space 20
        end
```