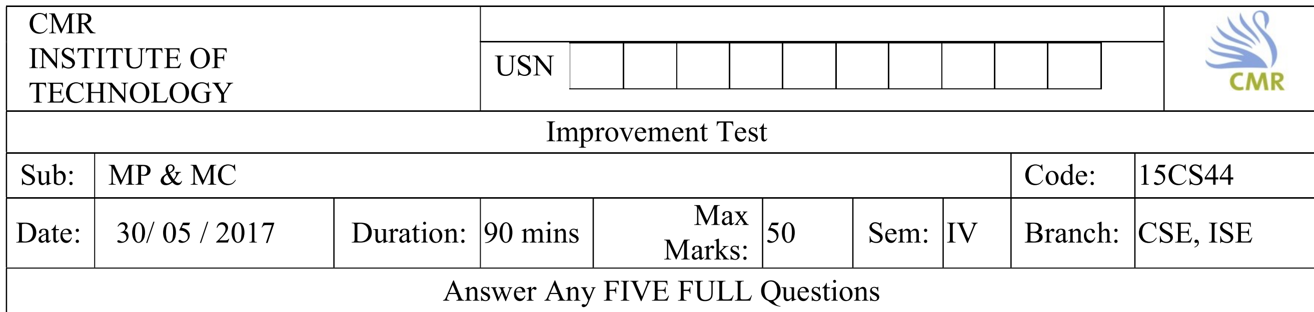

| CMR INSTITUTE OF TECHNOLOGY | USN | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|

| Improvement Test | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Sub: | MP & MC | | | | | | | Code: | 15CS44 |
| Date: | 30/ 05 / 2017 | Duration: | 90 mins | Max Marks: | 50 | Sem: | IV | Branch: | CSE, ISE |

Answer Any FIVE FULL Questions

| | | Marks | OBE CO | OBE RBT |
|---|---|---|---|---|
| 1 | (a) Explain ARM7 move instructions with relevant examples indicating Pre and post execution conditions. | [05] | CO4 | L1 |
| | (b) Explain the various syntax for barrel shifter data processing instruction of ARM? | [05] | | |
| 2 | (a) Explain the syntax of arithmetic instructions to implement addition and subtraction of 32-bit signed and unsigned values. Give examples for each instruction. | [05] | CO4 | L1 |
| | (b) Explain the syntax and usage of B, BL, BX and BLX instructions with necessary examples. | [05] | | |
| 3 | (a) Write an ARM assembly language code snippet to create an infinite loop. | [05] | CO4 | L3 |
| | (b) Write an assembly language code which uses BL instruction to call a subroutine to perform addition of three data words stored in registers. Specify the return statement with in the body of subroutine. | [05] | | |
| 4 | (a) Explain with examples the different addressing modes available with single register transfer instructions. | [05] | CO4 | L2 |
| | (b) Given: mem32[0x80018] = 0x03, mem32[0x80014] = 0x02, mem32[0x80010] = 0x01, r0 = 0x00080010, r1 = 0x00000000 ,r2 = 0x00000000, r3 = 0x00000000, r4= 0x000800C<br>Show the values updated after execution of<br>• LDMIA r0!, {r1-r3}<br>• STMDB r4!,{r1-r3} | [05] | | |
| 5 | (a) Explain the STACK operations in ARM7. Describe different addressing methods for stack operations. | [05] | CO4 | L1 |
| | (b) Explain SWP instruction. Describe any one use of SWP instruction with necessary code snippet. | [05] | | |
| 6 | (a) What is SWI? Explain with proper syntax and an example. | [05] | CO4 | L1 |
| | (b) Demonstrate all Program Status Register Instructions with proper syntax formats. | [05] | | |

| | | | | |
|---|---|---|---|---|
| 7 | (a) Explain different types of coprocessor instructions with their syntax. | [05] | CO4 | L1 |
| | (b) For the given set of Instructions write the post condition of CPSR register: Assume suitable data for cpsr.PRE cpsr=nzcvqIFt_svc<br><br>MRS r1, cpsr<br>BIC r1, r1, #0x80<br>MSR cpsr_c, r1 | [05] | CO4 | L2 |
| 8 | Explain different types of functions provided by INT 10H and INT 21H. | [10] | CO2 | L1 |
| 9 | Write a program using INT 10H to:<br>(a) Change the video mode<br>(b) Display the letter "D" in 200H locations with attributes black on white blinking. | [10] | CO2 | L3 |
| 10 | Write an ALP that does the following:<br>(a) Clears the screen<br>(b) Set the cursor to the center of the screen | [10] | CO2 | L3 |

1 a. Explain ARM7 move instructions with relevant examples indicating Pre and post execution conditions.

**Move Instructions**

- Move is the simplest ARM instruction. It copies *N into a destination register Rd, where*

*N is a register or immediate value.*

- *This instruction is useful for setting initial values and* transferring data between registers.

```
Syntax: <instruction>{<cond>}{S} Rd, N
```

| MOV | Move a 32-bit value into a register | $Rd = N$ |
|---|---|---|
| MVN | move the NOT of the 32-bit value into a register | $Rd = \sim N$ |

MPLE 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
```

Note: second operand *N for all data processing instructions. Usually it is a register Rm or* a constant preceded by #.

1b. Explain the various syntax for barrel shifter data processing instruction of ARM?

- 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).


- 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).


- 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 preprocessing* of registers.** Figure 3.1 shows the data flow between the ALU and the barrel shifter.

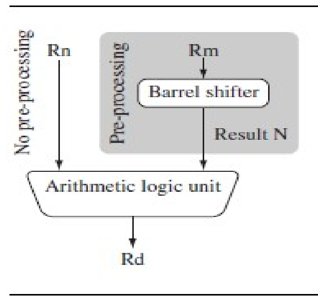


**Figure 3.1** Barrel shifter and ALU.

Example 3.2

**We apply a logical shift left (LSL) to register *Rm before moving it to the destination register.***
This is the same as applying the standard C language shift operator  to the register. The
MOV instruction copies the shift operator result *N into register Rd. N represents the result* of the LSL operation

**PRE r5 = 5** r7 = 8

```
     MOV    r7, r5, LSL #2   ; let r7 = r5*4 = (r5<<2)

POST  r5 = 5
      r7 = 20
```

| 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* |

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 |

EXAMPLE 3.3 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
```

Table 3.3 lists the syntax for the different barrel shift operations available on data processing instructions. The second operand *N* can be an immediate constant preceded by #, a register value *Rm*, or the value of *Rm* processed by a shift.

2 a. Explain the syntax of arithmetic instructions to implement addition and subtraction of 32-bit signed and unsigned values. Give examples for each instruction.

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

| | | |
|---|---|---|
| ADC | add two 32-bit values and carry | $Rd = Rn + N +$ carry |
| ADD | add two 32-bit values | $Rd = Rn + N$ |
| RSB | reverse subtract of two 32-bit values | $Rd = N - Rn$ |
| RSC | reverse subtract with carry of two 32-bit values | $Rd = N - Rn -$ !(carry flag) |
| SBC | subtract with carry of two 32-bit values | $Rd = Rn - N -$ !(carry flag) |
| SUB | subtract two 32-bit values | $Rd = Rn - N$ |

*N* is the result of the shifter operation. The syntax of shifter operation is shown in Table 3.3.

AMPLE 3.4 This simple subtract instruction subtracts a value stored in register *r2* from a value stored in register *r1*. The result is stored in register *r0*.

```
PRE    r0 = 0x00000000
       r1 = 0x00000002
       r2 = 0x00000001

       SUB r0, r1, r2

POST   r0 = 0x00000001
```

AMPLE 3.5 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.

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

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

POST   r0 = -r1 = 0xffffff89
```

AMPLE 3.6 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.

```
PRE    cpsr = nzcvqiFt_USER
       r1 = 0x00000001

       SUBS r1, r1, #1

POST   cpsr = nZCvqiFt_USER
       r1 = 0x00000000
```

Using the Barrel Shifter with Arithmetic Instructions

- The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set. Example 3.7 illustrates the use of the inline barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register r1 by three.

stored in register r1 by three.

EXAMPLE 3.7 Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

```
PRE    r0 = 0x00000000
       r1 = 0x00000005

       ADD     r0, r1, r1, LSL #1

POST   r0 = 0x0000000f
       r1 = 0x00000005
```

2b. Explain the syntax and usage of B, BL, BX and BLX instructions with necessary examples.

- A branch instruction changes the flow of execution or is used to call a routine

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

| B | branch | *pc* = *label* |
|---|---|---|
| BL | branch with link | *pc* = *label*<br>*lr* = address of the next instruction after the BL |
| BX | branch exchange | *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 |
| BLX | branch exchange with link | *pc* = *label*, *T* = 1<br>*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1<br>*lr* = address of the next instruction after the BLX |

The address *label* is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction. *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

3a. Write an ARM assembly language code snippet to create an infinite loop.

```
Backward ADD r1, r2, #4
         CMP r1, #2
         MOVEQ r5, r2
         B Backward
```

EXAMPLE 3.13 This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.

```
        B     forward
        ADD   r1, r2, #4
        ADD   r0, r6, #2
        ADD   r3, r7, #4
forward
        SUB   r1, r2, #4
_____________________________
backward
        ADD   r1, r2, #4
        SUB   r1, r2, #4
        ADD   r4, r6, r7
        B     backward
```

3 b . Write an assembly language code which uses BL instruction to call a subroutine to perform addition of three data words stored in registers. Specify the return statement with in the body of subroutine.

```
mov r1,#0x32
mov r2,#0x20
mov r3,#0x16
BL addition; call subroutine addition
mov r5,r4,lsl #2
.
.
.
.
.
.
.

addition  add r4,r1,r2
          add r4,r4,r3
          mov pc,lr      ;return statement
```

End; end of the code

4a. Explain with examples the different addressing modes available with single register transfer instructions.

- These instructions are used for moving a single data item in and out of a register.
- The datatypes supported are signed and unsigned words (32-bit), halfwords (16-bit), and bytes.

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

| LDR | load word into a register | *Rd <- mem32[address]* |
|---|---|---|
| STR | save byte or word from a register | *Rd -> mem32[address]* |
| LDRB | load byte into a register | *Rd <- mem8[address]* |
| STRB | save byte from a register | *Rd -> mem8[address]* |

| LDRH | load halfword into a register | *Rd <- mem16[address]* |
|---|---|---|
| STRH | save halfword into a register | *Rd -> mem16[address]* |
| LDRSB | load signed byte into a register | *Rd <- SignExtend (mem8[address])* |
| LDRSH | load signed halfword into a register | *Rd <- SignExtend (mem16[address])* |

Tables 3.5 and 3.7, to be presented is Section 3.3.2, describe the *addressing*[1] and *addressing*[2] syntax.

Single-Register Load-Store Addressing Modes

- The ARM instruction set provides different modes for addressing memory. These modes
- incorporate one of the indexing methods: **preindex with writeback, preindex, and postindex**

Table 3.4 Index methods.

| Index method | Data | Base address register | Example |
|---|---|---|---|
| Preindex with writeback | *mem[base + offset]* | *base + offset* | LDR r0,[r1,#4]! |
| Preindex | *mem[base + offset]* | *not updated* | LDR r0,[r1,#4] |
| Postindex | *mem[base]* | *base + offset* | LDR r0,[r1],#4 |

Note: ! indicates that the instruction writes the calculated address back to the base address register.

```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00009000] = 0x01010101
        mem32[0x00009004] = 0x02020202

        LDR    r0, [r1, #4]!
```

Preindexing with writeback:

```
POST(1) r0 = 0x02020202
        r1 = 0x00009004
```

```
        LDR    r0, [r1, #4]
```

Preindexing:

```
POST(2) r0 = 0x02020202
        r1 = 0x00009000

        LDR    r0, [r1], #4
```

Postindexing:

```
POST(3) r0 = 0x01010101
        r1 = 0x00009004
```

Table 3.5 Single-register load-store addressing, word or unsigned byte.

| Addressing[1] mode and index method | Addressing[1] syntax |
|---|---|
| Preindex with immediate offset | [Rn, #+/-offset_12] |
| Preindex with register offset | [Rn, +/-Rm] |
| Preindex with scaled register offset | [Rn, +/-Rm, shift #shift_imm] |
| Preindex writeback with immediate offset | [Rn, #+/-offset_12]! |
| Preindex writeback with register offset | [Rn, +/-Rm]! |
| Preindex writeback with scaled register offset | [Rn, +/-Rm, shift #shift_imm]! |
| Immediate postindexed | [Rn], #+/-offset_12 |
| Register postindex | [Rn], +/-Rm |
| Scaled register postindex | [Rn], +/-Rm, shift #shift_imm |

4b.Given: mem32[0x80018] = 0x03, mem32[0x80014] = 0x02, mem32[0x80010] = 0x01, r0 = 0x00080010, r1 = 0x00000000 ,r2 = 0x00000000, r3 = 0x00000000, r4= 0x000800C

Show the values updated after execution of

- LDMIA r0!, {r1-r3}
- STMDB r4!,{r1-r3}

Solution:

1) PRE
r0 = 0x00080010, r1 = 0x00000000 ,r2 = 0x00000000, r3 = 0x00000000
mem32[0x80018] = 0x03, mem32[0x80014] = 0x02, mem32[0x80010] = 0x01
LDMIA r0!, {r1-r3}

POST
r1 = 0x01, r2 = 0x02, r3 = 0x03
r0 = 0x0008001C

2) PRE
r1 = 0x01, r2 = 0x02, r3 = 0x03
r4= 0x0000800C

STMDB r4!,{r1-r3}

POST

mem32 [0x8008] = 0x03, mem32 [0x8004] = 0x02, mem32 [0x8000] = 0x01
r4= 0x00008000
r1 = 0x01, r2 = 0x02, r3 = 0x03

| Memory Adress | content |
|---|---|
| 0x00008010 | ------- |

| 0x0000800C | ------- |
|---|---|
| 0x00008008 | 0x03 |
| 0x00008004 | 0x02 |
| 0x00008000 | 0x01 |

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

The ARM architecture uses the load-store multiple instructions to carry out stack operations. The *pop* operation (removing data from a stack) uses a load multiple instruction; similarly, the *push* operation (placing data onto the stack) uses a store multiple instruction. When using a stack you have to decide whether the stack will grow up or down in memory. A stack is either *ascending* (A) or *descending* (D). Ascending stacks grow towards higher memory addresses; in contrast, descending stacks grow towards lower memory addresses.

When you use a *full stack* (F), the stack pointer *sp* points to an address that is the last used or full location (i.e., *sp* points to the last item on the stack). In contrast, if you use an *empty stack* (E) the *sp* points to an address that is the first unused or empty location (i.e., it points after the last item on the stack). There are a number of load-store multiple addressing mode aliases available to support stack operations (see Table 3.11). Next to the *pop* column is the actual load multiple instruction equivalent. For example, a full ascending stack would have the notation FA appended to the load multiple instruction—LDMFA. This would be translated into an LDMDA instruction. ARMhas specified an ARM-Thumb Procedure Call Standard (ATPCS) that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the pop and push functions, respectively.

Addressing methods for stack operations:

| Addressing mode | Description | Pop | = LDM | Push | = STM |
|---|---|---|---|---|---|
| FA | full ascending | LDMFA | LDMDA | STMFA | STMIB |
| FD | full descending | LDMFD | LDMIA | STMFD | STMDB |
| EA | empty ascending | LDMEA | LDMDB | STMEA | STMIA |
| ED | empty descending | LDMED | LDMIB | STMED | STMDA |

**PRE** r1 = 0x00000002
r4 = 0x00000003
sp = 0x00080010

**STMED sp!, {r1,r4}**

**POST** r1 = 0x00000002
r4 = 0x00000003
sp = 0x00080008

(b)Explain SWP instruction. Describe any one use of SWP instruction with necessary code snippet.

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register. This instruction is an *atomic operation*—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

**Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]**

| | | |
|---|---|---|
| SWP | swap a word between memory and a register | *tmp = mem32[Rn]*<br>*mem32[Rn] = Rm*<br>*Rd = tmp* |
| SWPB | swap a byte between memory and a register | *tmp = mem8[Rn]*<br>*mem8[Rn] = Rm*<br>*Rd = tmp* |

**PRE** mem32[0x9000] = 0x12345678
r0 = 0x00000000
r1 = 0x11112222
r2 = 0x00009000

**SWP r0, r1, [r2]**

**POST** mem32[0x9000] = **0x11112222**
r0 = 0x12345678
r1 = 0x11112222
r2 = 0x00009000

Swap cannot be interrupted by any other instruction or any other bus access. We say the system "holds the bus" until the transaction is complete.

6 (a) What is SWI? Explain with proper syntax and an example.

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

**Syntax: SWI{<cond>} SWI_number**

| | | |
|---|---|---|
| SWI | software interrupt | *lr_svc* = address of instruction following the SWI<br>*spsr_svc* = *cpsr*<br>*pc* = vectors + 0x8<br>*cpsr* mode = *SVC*<br>*cpsr I* = 1 (mask IRQ interrupts) |

When the processor executes an SWI instruction, it sets the program counter *pc* to the offset 0x8 in the vector table. The instruction also forces the processor mode to *SVC*, which allows an operating system routine to be called in a privileged mode. Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

**PRE** cpsr = nzcVqift_USER

pc = 0x00008000
lr = 0x003fffff; lr = r14
r0 = 0x12

**0x00008000 SWI 0x123456**

**POST** cpsr = **nzcVqIft_SVC**
spsr = **nzcVqift_USER**
pc = **0x00000008**
lr = **0x00008004**
r0 = **0x12**

(b) Demonstrate all Program Status Register Instructions with proper syntax formats.

The ARM instruction set provides two instructions to directly control a program status register (*psr*). The MRS instruction transfers the contents of either the *cpsr* or *spsr* into a register; in the reverse direction, the MSR instruction transfers the contents of a register into the *cpsr* or *spsr*. Together these instructions are used to read and write the *cpsr* and *spsr*. In the syntax you can see a label called *fields*. This can be any combination of control (*c*), extension (*x*), status (*s*), and flags (*f* ). These fields relate to particular byte regions in a *psr*, as shown in Figure.

**Syntax: MRS{<cond>} Rd,<cpsr|spsr>**
**MSR{<cond>} <cpsr|spsr>_<fields>,Rm**
**MSR{<cond>} <cpsr|spsr>_<fields>,#immediate**

| | | |
|---|---|---|
| MRS | copy program status register to a general-purpose register | *Rd = psr* |
| MSR | move a general-purpose register to a program status register | *psr[field] = Rm* |
| MSR | move an immediate value to a program status register | *psr[field] = immediate* |

The *c* field controls the interrupt masks, Thumb state, and processor mode. Example shows how to enable IRQ interrupts by clearing the *I* mask. This operation involves using both the MRS and MSR instructions to read from and then write to the *cpsr*.

**PRE** cpsr = nzcvqIFt_SVC

```
MRS r1, cpsr
BIC r1, r1, #0x80 ; 0b01000000
MSR cpsr_c, r1
```

**POST** cpsr = nzcvqiFt_SVC

7 (a) Explain different types of coprocessor instructions with their syntax.

Coprocessor instructions are used to extend the instruction set. A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management. The coprocessor instructions include data processing, register transfer, and memory transfer instructions. We will provide only a short overview since these instructions are coprocessor specific. Note that these instructions are only used by cores with a coprocessor.

**Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}**
**<MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}**
**<LDC|STC>{<cond>} cp, Cd, addressing**

| CDP | coprocessor data processing—perform an operation in a coprocessor |
|---|---|
| MRC MCR | coprocessor register transfer—move data to/from coprocessor registers |
| LDC STC | coprocessor memory transfer—load and store blocks of memory to/from a coprocessor |

In the syntax of the coprocessor instructions, the *cp* field represents the coprocessor number between *p0* and *p15*. The *opcode* fields describe the operation to take place on the coprocessor. The *Cn*, *Cm*, and *Cd* fields describe registers within the coprocessor. The coprocessor operations and registers depend on the specific coprocessor you are using. Coprocessor 15 (CP15) is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

(b) For the given set of Instructions write the post condition of CPSR register: Assume suitable data for cpsr.
PRE cpsr=nzcvqIFt_svc

```
MRS r1, cpsr
BIC r1, r1, #0x80
MSR cpsr_c, r1
```

**POST cpsr = nzcvqiFt_SVC**

8. Explain different types of functions provided by INT 10H and INT 21H.

## DOS Interrupt – INT 21H

*Format: MOV AH, #FUCTION CODE*

*INT 21H*

*Function Codes:*

1. AH = 01h - READ CHARACTER FROM STANDARD INPUT, WITH ECHO

Return: AL = character read

2. AH = 02h -WRITE CHARACTER TO STANDARD OUTPUT
   Entry: DL = character to write
   Return: AL = last character output

3. AH=07h - DIRECT CHARACTER INPUT, WITHOUT ECHO

Return: AL = character read from standard input

4. AH = 09h - WRITE STRING TO STANDARD OUTPUT

Entry: DS: DX -> '$'-terminated string address

The string must be terminated by a '$' character. DS must point to the string's segment, and DX must contain the string's offset

Return: AL = 24h

5. AH = 0Ah - BUFFERED INPUT

Entry: DS: DX -> buffer

Return: buffer filled with user input

Format of DOS input buffer:

| Offset | Size | Description |
|---|---|---|
| 00 | 1 | maximum characters buffer can hold |
| 01 | 1 | number of chars from last input which may be recalled OR number of characters actually read, excluding CR |
| 02 | n | actual characters read, including the final carriage return |

6. AH=0Bh - GET STDIN STATUS

Return:

- AL = 00h if no character available
- AL = FFh if character is available

7. H = 2Ah - GET SYSTEM DATE
- Return: CX = year (1980-2099) DH = month DL = day AL = day of week (00h=Sunday)
8. AH = 2Bh - SET SYSTEM DATE

Entry: CX = year (1980-2099) DH = month DL = day

Return:

- AL = 00 successful
- FFh invalid date, system date unchanged

9. AH = 2Ch - GET SYSTEM TIME

Return: CH = hour CL = minute DH = second DL = 1/100 seconds

Note: on most systems, the resolution of the system clock is about 5/100sec, so returned times generally do not increment by 1 on some systems, DL may always return 00h

SeeAlso: AH=2Ah,AH=2Dh,AH=E7h

10. AH = 2Dh - SET SYSTEM TIME

Entry: CH = hour CL = minute DH = second DL = 1/100 seconds

Return:

- AL = 00h successful
- FFh if invalid time, system time unchanged

11. AH = 4Ch - "EXIT" - TERMINATE WITH RETURN CODE

- Entry: AL = return code
- Return: never returns
- Notes: unless the process is its own parent, all open files are closed and all memory belonging to the process is freed

## BIOS INTERRUPT (INT 10H)

### INT 10h Functions

One way to display text on the screen quickly is to use the BIOS interrupt 10h functions. See the INT 10h function list elsewhere for a complete description of these functions. A brief list of the more useful functions is given here:

| | |
|---|---|
| Function 0 | Set Video Mode |
| Function 2 | Set Cursor Position |
| Function 6 | Scroll Active Page Up |
| Function 9 | Write Attribute/character at Current Cursor Position |

INT 10h / AH = 0 - set video mode.

input:

AL = desired video mode.

these video modes are supported:

00h - text mode. 40x25. 16 colors. 8 pages.

03h - text mode. 80x25. 16 colors. 8 pages.

13h - graphical mode. 40x25. 256 colors. 320x200 pixels. 1 page.

INT 10h / AH = 2 - set cursor position.

input:

DH = row.

DL = column.

BH = page number (0..7).

INT 10h / AH = 03h - get cursor position and size.

input:

BH = page number.

return:

DH = row.

DL = column.

CH = cursor start line.

CL = cursor bottom line

INT 10h / AH = 06h - scroll up window.

INT 10h / AH = 07h - scroll down window.

**input:**

AL = number of lines by which to scroll (00h = clear entire window).

BH = attribute used to write blank lines at bottom of window.

CH, CL = row, column of window's upper left corner.

DH, DL = row, column of window's lower right corner.

INT 10h / AH = 09h - write character and attribute at cursor position.

input:

AL = character to display.

BH = page number.

BL = attribute.

CX = number of times to write character.

9 Write a program using INT 10H to:

(a) Change the video mode
(b) Display the letter “D” in 200H locations with attributes black on white blinking

```
MODEL SMALL

.CODE

; To change to video mode monochrome

START:MOV AH, 00; SET VIDEO MODE

MOV AL, 07; GREY/MONOCHROME TEXT

INT 10H

; Subcode for display character is AH=09H, BL specifies the attribute, BH specifies the page
number, AL should contain the ascii value of the character to be displayed and CX contains the
number of times the character to be displayed

MOV AH,09H

MOV BL,00

MOV AL,44H ;CHARCTER “D”

MOV CX,200H

MOV BL,0F0H

INT 10H

MOV AH,4CH
INT 21H

END  START
```

10 Write an ALP that does the following:

(a) Clears the screen
(b) Set the cursor to the center of the screen

```
.MODEL SMALL
.CODE
MOV AX,@DATA
MOV DS,AX
;TO CLEAR THE SCREEN
MOV AH,06H ; SCROLL UP
MOV AL,00 ;CLEAR ENTIRE WINDOW
MOV BH,07 ; NORMAL ATTRIBUTE
MOV CX,0000H ; ROW NAND COLUMN OF TOP LEFT
MOV DX, 184FH; ROW AND COLUMN OF BOTTOM RIGHT
INT 10H
; TO SET CURSOR AT THE CENTRE
MOV AH,02; TO SET CURSOR
MOV BH,00; PAGE 0
MOV DL, 39; COLUMN
MOV DH, 12; ROW
INT 10H

MOV AH,4CH
INT 21H
END
```