

## MODULE 2

**Data Processing Instructions**

- The data processing instructions manipulate data within registers. They are move instructions, arithmetic instructions, logical instructions, compare instructions and multiply instructions.
- Most data processing instructions can process one of their operands using the barrel shifter.
- If S is suffixed on a data processing instruction, then it updates the flags in the cpsr.

**MOVE INSTRUCTIONS:**

- 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	$Rd = \sim N$

- In the example shown below, the MOV instruction takes the contents of register r5 and copies them into register r7.

```

PRE   r5 = 5
      r7 = 8
      MOV r7, r5 ; let r7 = r5
POST  r5 = 5
      r7 = 5

```

**USING BARREL SHIFTER WITH DATA TRANSFER INSTRUCTION:**

- Data processing instructions are processed within the arithmetic and 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.
- For example, We apply a logical shift left (LSL) to register Rm before moving it to the destination register.

```

PRE   r5=5
      r7=8
      MOV r7, r5, LSL #2
POST  r5=5
      r7=20

```

- The above example shift logical left r5=5 (00000101 in binary) by two bits and then r7=20 (00010100 in binary).

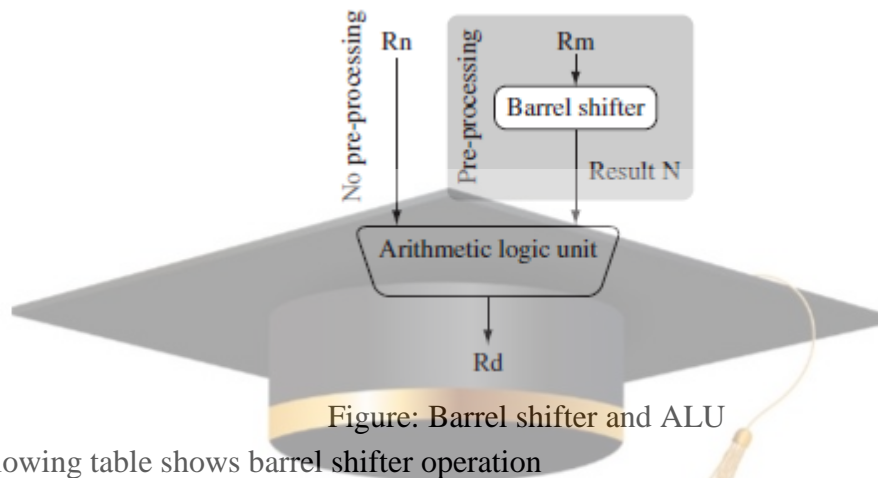


Figure: Barrel shifter and ALU

- Following table shows barrel shifter operation

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

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

Barrel shift operation syntax for data processing instructions.

$N$ shift operations	Syntax
Immediate	#immediate
Register	$R_m$
Logical shift left by immediate	$R_m, \text{LSL } \# \text{shift\_imm}$
Logical shift left by register	$R_m, \text{LSL } R_s$
Logical shift right by immediate	$R_m, \text{LSR } \# \text{shift\_imm}$
Logical shift right with register	$R_m, \text{LSR } R_s$
Arithmetic shift right by immediate	$R_m, \text{ASR } \# \text{shift\_imm}$
Arithmetic shift right by register	$R_m, \text{ASR } R_s$
Rotate right by immediate	$R_m, \text{ROR } \# \text{shift\_imm}$
Rotate right by register	$R_m, \text{ROR } R_s$
Rotate right with extend	$R_m, \text{RRX}$

**ARITHMETIC INSTRUCTIONS:**

- The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

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

ADC	add two 32-bit values and carry	$Rd = Rn + N + \text{carry}$
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	$Rd = N - Rn - !(\text{carry flag})$
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N - !(\text{carry flag})$
SUB	subtract two 32-bit values	$Rd = Rn - N$

- In the following example, subtract instruction subtracts a value stored in register r2 from a value stored in the register r1. The result is stored in register r0.

```

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

      SUB r0, r1, r2

POST  r0 = 0x00000001

```

- In the following example, the reverse subtract instruction (RSB) subtract r1 from the constant value #0, writing the result in r0.

```

PRE   r0 = 0x00000000
      r1 = 0x00000077

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

POST  r0 = -r1 = 0xffffffff89

```

**USING THE BARREL SHIFTER WITH ARITHMETIC INSTRUCTIONS:**

- Example below illustrates the use of the inline barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register r1 by three.
- 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

```

**LOGICAL INSTRUCTIONS:**

- Logical instructions perform bitwise operations on the two source registers.

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

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn   N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

- In the example shown below, a logical OR operation between registers r1 and r2 and the result is in r0.

```

PRE    r0 = 0x00000000
        r1 = 0x02040608
        r2 = 0x10305070
        ORR    r0, r1, r2

POST   r0 = 0x12345678

```

**COMPARISON INSTRUCTIONS:**

- The comparison instructions are used to compare or test a register with a 32-bit value. They update the cpsr flag bits according to the result, but do not affect other registers.
- After the bits have been set, the information can be used to change program flow by using conditional execution.

**Syntax:** <instruction> {<cond>} Rn, N

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

- Example shown below for CMP instruction, both r0 and r1 are equal before the execution of the instruction. The value of the z flag prior to the execution is 0 and after the execution z flag changes to 1 (upper case of Z).

```

PRE    cpsr = nzcvqiFt_USER
        r0 = 4
        r9 = 4

        CMP    r0, r9

POST   cpsr = nZcvqiFt_USER

```

- The CMP is effectively a subtract instruction with the result discarded;
- Similarly the TST instruction is a logical AND operation and TEQ is a logical XOR operation. For each, the results are discarded but the condition bits are updated in the cpsr.

### MULTIPLY INSTRUCTIONS:

- The multiply instructions multiply the contents of a pair of registers and depending upon the instruction, accumulate the results in another register.
- The long multiplies accumulate onto a pair of registers representing a 64-bit value.

**Syntax: MLA {<cond>} {S} Rd, Rm, Rs, Rn**

**MUL {<cond>} {S} Rd, Rm, Rs**

MLA	multiply and accumulate	$Rd = (Rm * Rs) + Rn$
MUL	multiply	$Rd = Rm * Rs$

Syntax: <instruction> {<cond>} {S} RdLo, RdHi, Rm, Rs

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

- In the following example below shows a multiply instruction that multiplies registers r1 and r2 and places the result into the register r0.

```

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

        MUL    r0, r1, r2    ; r0 = r1*r2

POST   r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002

```

- The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) produce a 64-bit result.

```

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

```

## BRANCH INSTRUCTIONS

Q2. Explain briefly branch instructions of ARM processor.

**Answer:**

- A branch instruction changes the flow of execution or is used to call a routine.
- This type of instruction allows programs to have subroutines, if-then-else structures, and loops.
- The change of execution flow forces the program counter (pc) to point to a new address.

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

B	branch	$pc = label$
BL	branch with link	$pc = label$ $lr = \text{address of the next instruction after the BL}$
BX	branch exchange	$pc = Rm \ \& \ 0xffffffe$ , $T = Rm \ \& \ 1$
BLX	branch exchange with link	$pc = label$ , $T = 1$ $pc = Rm \ \& \ 0xffffffe$ , $T = Rm \ \& \ 1$ $lr = \text{address of the next instruction after the BLX}$

- T refers to the Thumb bit in the cpsr.
- When instruction set T, the ARM switches to Thumb state.
- The example shown below is a forward branch. The forward branch skips three instructions.

```

B      forward
ADD    r1, r2, #4
ADD    r0, r6, #2
ADD    r3, r7, #4

forward
SUB    r1, r2, #4

```

- The branch with link (BL) instruction changes the execution flow in addition overwrites the link register lr with a return address. The example shows below a fragment of code that branches to a subroutine using the BL instruction.

```

        BL      subroutine      ; branch to subroutine
        CMP     r1, #5          ; compare r1 with 5
        MOVEQ   r1, #0          ; if (r1==5) then r1 = 0
        :
subroutine
    <subroutine code>
    MOV     pc, lr              ; return by moving pc = lr

```

- The branch exchange (BX)** instruction uses an absolute address stored in register Rm. It is primarily used to branch to and from Thumb code. The T bit in the cpsr is updated by the least significant bit of the branch register.
- Similarly, **branch exchange with link (BLX)** instruction updates the T bit of the cpsr with the least significant bit and additionally sets the link register with the return address.

### LOAD-STORE INSTRUCTIONS ( Memory Access Instructions)

- Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.
- a) **Single-Register Transfer**
- These instructions are used for moving a single data item in and out of a register.
- Here are the various load-store single-register transfer instructions.

**Syntax:** <LDR|STR>{<cond>}{B} Rd, addressing<sup>1</sup>  
 LDR{<cond>}SB|H|SH Rd, addressing<sup>2</sup>  
 STR{<cond>}H Rd, addressing<sup>2</sup>

LDR	load word into a register	$Rd \leftarrow mem32[address]$
STR	save byte or word from a register	$Rd \rightarrow mem32[address]$
LDRB	load byte into a register	$Rd \leftarrow mem8[address]$
STRB	save byte from a register	$Rd \rightarrow mem8[address]$
LDRH	load halfword into a register	$Rd \leftarrow mem16[address]$
STRH	save halfword into a register	$Rd \rightarrow mem16[address]$
LDRSB	load signed byte into a register	$Rd \leftarrow SignExtend(mem8[address])$
LDRSH	load signed halfword into a register	$Rd \leftarrow SignExtend(mem16[address])$

- Example:
  1. LDR r0, [r1]
    - This instruction loads a word from the address stored in register r1 and places it into register r0.
  2. STR r0, [r1]
    - This instruction goes the other way by storing the contents of register r0 to the address contained in register r1.

### b) Multiple-Register Transfer

- Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction. The transfer occurs from a base address register Rn pointing into memory.
- Multiple-register transfer instructions are more efficient from single-register transfers for moving blocks of data around memory and saving and restoring context and stacks.

**Syntax:** <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}

LDM	load multiple registers	{Rd}*N <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	{Rd}*N -> mem32[start address + 4*N] optional Rn updated

- Here N is the number of registers in the list of registers.

### c) SWAP Instruction

- The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register.

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

SWP	swap a word between memory and a register	<i>tmp = mem32[Rn] mem32[Rn] = Rm Rd = tmp</i>
SWPB	swap a byte between memory and a register	<i>tmp = mem8[Rn] mem8[Rn] = Rm Rd = tmp</i>

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

SWP r0, r1, [r2]

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

Addressing modes:**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

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

Example:

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

Addressing mode for load-store multiple instructions

- Table below shows the different addressing modes for the load-store multiple instructions.

Addressing mode	Description
IA	increment after
IB	increment before
DA	decrement after
DB	decrement before

Example:

```

mem32[0x8001c] = 0x04
PRE  mem32[0x80018] = 0x03
      mem32[0x80014] = 0x02

      mem32[0x80010] = 0x01
      r0 = 0x00080010
      r1 = 0x00000000
      r2 = 0x00000000
      r3 = 0x00000000

      LDMIA    r0!, {r1-r3}

POST  r0 = 0x0008001c
      r1 = 0x00000001
      r2 = 0x00000002
      r3 = 0x00000003

```

- If LDMIA is replaced with LDMIB post execution the content of registers is shown below

```

r3 = 0x00000004
r2 = 0x00000003
r1 = 0x00000002
r0 = 0x8001c

```

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 you use a **full stack (F)**, the stack pointer *sp* points to an address that is the last used or full location.
- In contrast, if you use an **empty stack (E)** the *sp* points to an address that is the first unused or empty location.
- 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.
- Addressing modes for stack operation

Addressing mode	Description
FA	full ascending
FD	full descending
EA	empty ascending
ED	empty descending

- The LDMFD and STMFD instructions provide the pop and push functions, respectively.
- Example1: With full descending

PRE    *r1* = 0x00000002  
        *r4* = 0x00000003  
        *sp* = 0x00080014

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

POST   *r1* = 0x00000002  
        *r4* = 0x00000003  
        *sp* = 0x0008000c

PRE	Address	Data	POST	Address	Data
	0x80018	0x00000001		0x80018	0x00000001
<i>sp</i> →	0x80014	0x00000002		0x80014	0x00000002
	0x80010	Empty		0x80010	0x00000003
	0x8000c	Empty	<i>sp</i> →	0x8000c	0x00000002

Figure: STMFD instruction full stack push operation.

Example 2: With empty descending

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

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

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

PRE	Address	Data	POST	Address	Data
	0x80018	0x00000001		0x80018	0x00000001
	0x80014	0x00000002		0x80014	0x00000002
<i>sp</i> →	0x80010	Empty		0x80010	0x00000003
	0x8000c	Empty		0x8000c	0x00000002
	0x80008	Empty	<i>sp</i> →	0x80008	Empty

Figure: STMED instruction empty stack push operation.

**SOFTWARE INTERRUPT INSTRUCTION**

Q3. Explain briefly the software interrupt instruction.

**Answer:**

- 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	<i>lr_svc</i> = address of instruction following the SWI <i>spsr_svc</i> = <i>cpsr</i> <i>pc</i> = vectors + 0x8 <i>cpsr</i> mode = SVC <i>cpsr</i> I = 1 (mask IRQ interrupts)
-----	--------------------	---

- When the processor executes an SWI instruction, it sets the program counter *pc* to the offset 0xB 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.
- The example below shows an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI.

```

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
  
```

- Since SWI instructions are used to call operating system routines, it is required some form of parameter passing.
- This achieved by using registers. In the above example, register *r0* is used to pass parameter 0x12. The return values are also passed back via register.

## Program Status Register Instructions

Q4. Explain briefly program status register instructions.

**Answer:**

- 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 to general purpose register.
- The MSR instruction transfers the contents of a general purpose register to cpsr or spsr.
- Together these instructions are used to read and write the cpsr and spsr.

**Syntax: MRS {<cond>} Rd <cpsr |spsr>**

**MSR {<cond>} <cpsr|spsr> \_<fields>,Rm**

**MSR {<cond>} <cpsr|spsr> \_<fields>, #immediate**

- The table shows the program status register instructions

MRS	copy program status register to a general-purpose register	<i>Rd = psr</i>
MSR	move a general-purpose register to a program status register	<i>psr[field] = Rm</i>
MSR	move an immediate value to a program status register	<i>psr[field] = immediate</i>

## Coprocessor Instructions

Q5. Explain briefly coprocessor instructions.

**Answer:**

- 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.
- These instructions are used only by core 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 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.
- For example: The instruction below copies coprocessor CP15 register c0 into a general purpose register r10.

MRC p15, 0, r10, c0, c0, 0 ; CP15 register-0 is copied into general purpose register r10.

- For example: The instruction below moves the contents of CP15 control register c1 into register r1 of the processor core.

MRC p15, 0, r1, c1, c0, 0

### Loading Constants

Q6. Explain briefly the loading constants.

Answer:

- There are two pseudo instructions to move a 32-bit constant value to a register.

Syntax: LDR Rd, =constant

ADR Rd, label

LDR	load constant pseudoinstruction	Rd = 32-bit constant
ADR	load address pseudoinstruction	Rd = 32-bit relative address

- The example below shows an LDR instruction loading a 32-bit constant 0xff00ffff into register r0.

LDR r0, =0xff00ffff

**Programs:**

- Write ALP program for ARM7 demonstrating the data transfer.

Answer:

```

AREA DATATRANSFER, CODE, READONLY
ENTRY
LDR R9,=SRC ;LOAD STARTING ADDRESS OF SOURCE
LDR R10,=DST ;LOAD STARTING ADDRESS OF DESTINATION

LDMIA R9!,{R0-R7}
STMIA R10!,{R0-R7}

SRC DCD 1,2,3,4,5,6,7,8
AREA BLOCKDATA, DATA, READWRITE
DST DCD 0,0,0,0,0,0,0,0
END
  
```

2. Write ALP program for ARM7 demonstrating logical operation.

Answer:

```
AREA LOGIC, CODE, READONLY
ENTRY
LDR R0, =5
LDR R1, =3
AND R4, R0, R1
ORR R5, R0, R1
EOR R6, R0, R1
BIC R7, R0, R1
END
```

3. Write ALP program for ARM7 demonstrating arithmetic operation.

Answer:

```
AREA ARITH, CODE, READONLY
ENTRY
LDR R1, =20
LDR R2, =25
ADD R3, R1, R2
MUL R4, R1, R2
SUB R5, R1, R2
END
```

4. Write ALP using ARM instructions that calls subroutine fact to find factorial of a given number.

Answer:

```
AREA FACTORIAL, CODE, READONLY
ENTRY
START LDR R0, #5
      BL FACT          // BRANCH WITH LINK
      LDR R4, =DST      // LOCATION TO STORE RESULT
      STR R5, [R4]
STOP   B STOP
```

```
FACT
      MOVS R1, R0      ; If R1= 0, ZF =1
      MOVEQ R5, #1     ; If ZF =1, Return 1
LOOP
      SUBNES R1, R1, #1 ; R1 = R1-1 if R1 not 0
      MULNE R0, R1, R0 ; R0 = R1 * R0
      BNE LOOP         ; IF (R1 != 0) LOOP.
```

```

MOV R5, R0
MOV PC, R14          ; RETURN WITH RESULT IN R5.
AREA FACT, DATA, READWRITE
DST DCD 0

END

```

5. Write ALP program to add array of 16 bit numbers and store the result in memory.

Answer:

```

AREA AryAdd, CODE, READONLY
ENTRY
LDR R0, =SRC          ; pointer to source array
LDR R1, =DST          ; pointer to destination
MOV R2, #5            ; count of numbers
MOV R5, #0            ; initial sum
UP LDRH R3, [R0]       ; 1st number in R2
ADD R5, R5, R3         ; add numbers
ADD R0, R0, #2        ; increment pointer to next number
SUBS R2, R2, #1       ; decrement count by 1
CMP R2, #0
BNE UP
STRH R5, [R1]
STOP B STOP
SRC DCW 10, 20, 30, 40, 50
AREA BLOCKDATA, DATA, READWRITE
DST DCW 0
END

```

6. Write ALP program to generate Fibonacci series.

```

AREA FIB, CODE, READONLY
ENTRY
MOV R0, #00           ; FIRST FIBONACCI NUMBER
SUB R0, R0, #01       ; R0= -1
MOV R1, #01
MOV R4, #05           ; NO OF FIBONACCI NUMBERS TO GENERATE
LDR R2, =FIBO         ; ADDRESS TO STORE FIBONACCI NUMBERS
BACK ADD R0, R1        ; ADDING THE PREVIOUS TWO NUMBERS
STR R0, [R2]          ; STORING THE NUMBER IN A MEMORY
ADD R2, #04           ; INCREMENTING THE ADDRESS
MOV R3, R0
MOV R0, R1

```

```

MOV R1, R3
SUB R4, #01           ;DECREMENTING THE COUNTER
CMP R4, #00           ;COMPARING THE COUNTER TO ZERO
BNE BACK              ;LOOPING BACK
STOP B STOP
AREA FIBONACCI, DATA, READWRITE
FIBO DCD 0,0,0,0,0
END

```

7. Write an ALP to copy a block (Block1 ) to another block (Block2) using ARM instructions. ( Lab 6b program)

```

area word, code, readonly ;name the block of code
num equ 20                 ;set number of words to be copied
entry                      ;mark the first instruction called

Start
ldr r0, =src               ;r0 = pointer to source block
ldr r1, =dst               ;r1 = pointer to destination block
mov r2, #num               ;r2 = number of words to copy

Wordcopy
ldr r3, [r0], #4           ;load a word from the source(src) and
str r3, [r1], #4           ;store it to the destination(dst)
subs r2, r2, #1            ;decrement the counter(num)
bne wordcopy               ;... copy more

Stop
src dcd 1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
area blockdata, data, readwrite
dst dcd 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

End

```

8. Write an ALP to display message “HELLO WORLD” using ARM7 instructions

```

#include <lpc214x.h>
void uart_interrupt(void) __irq ;
unsigned char temp , temp1 = 0x00 ;
unsigned char rx_flag = 0 , tx_flag = 0 ;
int main(void)
{
    PINSEL0=0X00000005; //select TXD0 and RXD0 lines
    U0LCR = 0X00000083; //enable baud rate divisor loading and
    U0DLM = 0X00; //select the data format
    U0DLL = 0x13; //select baud rate 9600 bps
    U0LCR = 0X00000003;
    U0IER = 0X03; //select Transmit and Recieve interrupt
    VICVectAddr0 = (unsigned long)uart_interrupt; //UART 0 INTERRUPT

```

```

VICVectCntl0 = 0x20|6; // Assign the VIC channel uart-0 to interrupt priority 0
VICIntEnable = 0x00000040; // Enable the uart-0 interrupt
rx_flag = 0x00;
tx_flag = 0x00;
while(1)
{
    while(rx_flag == 0x00); //wait for receive flag to set
    rx_flag = 0x00; //clear the flag
    U0THR = temp1 ;
    while(tx_flag == 0x00); //wait for transmit flag to set
    tx_flag = 0x00; //clear the flag
}
}
void uart_interrupt(void) __irq
{
    temp = U0IIR;
    temp = temp & 0x06; //check bits, data sending or receiving
    if(temp == 0x02) //check data is sending
    {
        tx_flag = 0xff; // flag that indicate data is sending via UART0
        VICVectAddr=0;
    }

    else if(temp == 0x04) // check any data available to receive
    {
        // U0THR = U0RBR;
        emp1 = U0RBR ; // copy data into variable
        rx_flag = 0xff; // set flag to indicate that data is received
        VICVectAddr=0;
    }
}

```

### WRITING AND OPTIMIZING ARM ASSEMBLY CODE

Writing assembly by hand gives you direct control of three optimization tools that you cannot explicitly use by writing C source:

- **Instruction scheduling:** Reordering the instructions in a code sequence to avoid processor stalls. Since ARM implementations are pipelined, the timing of an instruction can be affected by neighboring instructions.
- **Register allocation:** Deciding how variables should be allocated to ARM registers or stack locations for maximum performance. Our goal is to minimize the number of memory accesses.
- **Conditional execution:** Accessing the full range of ARM condition codes and conditional instructions.

## Writing Assembly Code

### Example 6.1

This example shows how to convert a C function to an assembly function—usually the first stage of assembly optimization. Consider the simple C program `main.c` following that prints the squares of the integers from 0 to 9:

```
#include <stdio.h>

int square(int i);

int main(void)
{
    int i;
    for (i=0; i<10; i++)
    {
        printf("Square of %d is %d\n", i, square(i));
    }
}

int square(int i)
{
    return i*i;
}
```

- Let's see how to replace `square` by an assembly function that performs the same action. Remove the C definition of `square`, but not the declaration (the second line) to produce a new C file `main1.c`. Next add an *armasm* assembler file `square.s` with the following contents:

```
AREA    |.text|, CODE, READONLY
EXPORT  square

; int square(int i)
square
    MUL    r1, r0, r0    ; r1 = r0 * r0
    MOV    r0, r1        ; r0 = r1
    MOV    pc, lr        ; return r0
END
```

- The `AREA` directive names the area or code section that the code lives in. If you use nonalphanumeric characters in a symbol or area name, then enclose the name in vertical bars.
- The `EXPORT` directive makes the symbol `square` available for external linking.
- The input argument is passed in register `r0`, and the return value is returned in register `r0`.

- The multiply instruction has a restriction that the destination register must not be the same as the first argument register. Therefore we place the multiply result into r1 and move this to r0.
- The END directive marks the end of the assembly file. Comments follow a semicolon.
- Example 6.1 only works if you are compiling your C as ARM code. If you compile your C as Thumb code, then the assembly routine must return using a BX instruction as shown below

```

AREA    |.text|, CODE, READONLY
EXPORT  square

; int square(int i)
square
    MUL    r1, r0, r0    ; r1 = r0 * r0
    MOV    r0, r1        ; r0 = r1
    BX     lr            ; return r0
END

```

### Example 6.2

This example shows how to call a subroutine from an assembly routine. We will take Example 6.1 and convert the whole program (including main) into assembly. We will call the C library routine printf as a subroutine. Create a new assembly file main3.s with the following contents:

```

AREA    |.text|, CODE, READONLY
EXPORT  main

IMPORT  |Lib$$Request$$armlib|, WEAK
IMPORT  _main    ; C library entry
IMPORT  printf    ; prints to stdout

i
RN 4

; int main(void)
main
    STMFD    sp!, {i, lr}
    MOV     i, #0

loop
    ADR     r0, print_string
    MOV     r1, i
    MUL     r2, i, i
    BL     printf
    ADD     i, i, #1
    CMP     i, #10
    BLT     loop
    LDMFD    sp!, {i, pc}

print_string
    DCB     "Square of %d is %d\n", 0

END

```

- We have used a new directive, `IMPORT`, to declare symbols that are defined in other files.
- The imported symbol `Lib$$Request$$armlib` makes a request that the linker links with the standard ARM C library. The `WEAK` specifier prevents the linker from giving an error if the symbol is not found at link time. If the symbol is not found, it will take the value zero.
- The second imported symbol `__main` is the start of the C library initialization code.
- You only need to import these symbols if you are defining your own main; a main defined in C code will import these automatically for you. Importing `printf` allows us to call that C library function.
- The `RN` directive allows us to use names for registers. In this case we define *i* as an alternate name for register *r4*. Using register names makes the code more readable.
- Recall that ATPCS states that a function must preserve registers *r4* to *r11* and *sp*. We corrupt *i(r4)*, and calling `printf` will corrupt *lr*.
- Therefore we stack these two registers at the start of the function using an `STMFD` instruction. The `LDMFD` instruction pulls these registers from the stack and returns by writing the return address to *pc*.
- The `DCB` directive defines byte data described as a string or a comma-separated list of bytes.
- Note that Example 6.3 also assumes that the code is called from ARM code. If the code can be called from Thumb code as in Example 6.2 then we must be capable of returning to Thumb code.

```
LDMFD    sp!, {i, lr}  
BX       lr
```

Vtudeveloper.in