UNIT-II

ARM PROGRAMMING MODEL

PART A: ARM INSTRUCTION SET

Contents:

INTRODUCTION:

- In instructions data will be represent hexadecimal numbers with the prefix **0x** and binary numbers with the prefix **0b**.
- The examples follow this format:

  PRE <pre-conditions>

  <instruction/s>

  POST <post-conditions>
- In the pre- and post-conditions, memory is denoted as

  Mem <data_size> [address]
- This refers to *data_size* bits of memory starting at the given byte *address*.
- For example *mem32* [1024] is the 32-bit value starting at address 1 KB.
- ARM instructions process data held in registers and only access memory with load and store instructions.
- ARM instructions commonly take two or three operands. For instance the ADD instruction below adds the two values stored in registers *r1* and *r2* (the source registers). It writes the result to register *r3* (the destination register).

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

Data Processing Instructions:

- The data processing instructions manipulate data within registers.
- They are

- Move instructions
- Arithmetic instructions
- Logical instructions
- Comparison instructions
- Multiply instructions

- Most data processing instructions can process one of their operands using the barrel shifter.
- If you use the S suffix on a data processing instruction, then it updates the flags in the CPSR.
- Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z*.
    - The carry flag is set from the result of the barrel shift as the last bit shifted out.
    - The *N* flag is set to bit 31 of the result.
    - The *Z* flag is set if the result is zero.

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	$Rd = \sim N$

- The second operand *N* for all data processing instructions is a register R_m or a constant preceded by #.

Example: 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

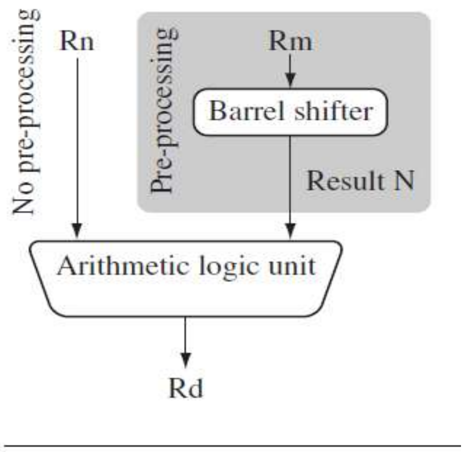
  POST r5 = 5

  r7 = 5

Barrel Shifter:

- In above example 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 Pre-processed 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.

- 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 below figure shows the data flow between the ALU and the barrel shifter.



Barrel shifter and ALU.

- Register *Rn* enters the ALU without any pre-processing of registers.

Example: Apply a logical shift left (LSL) to register *Rm* before moving it to the destination 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

- The example multiplies register *r5* by four and then places the result into register *r7*.
- The five different shift operations that you can use within the barrel shifter are summarized in below table.

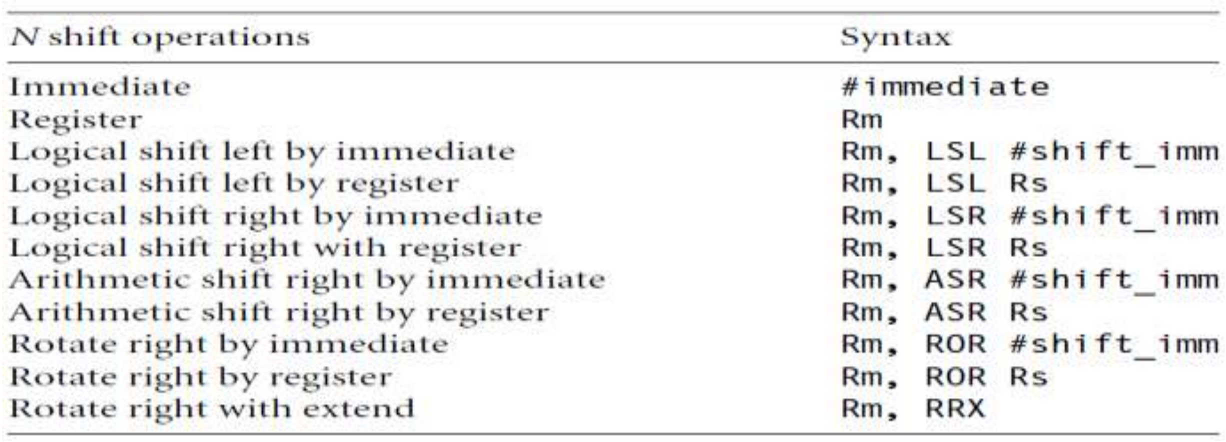
Table 3.2 Barrel shifter operations.

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*	$(\text{unsigned})x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x* ASR *y*	$(\text{signed})x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x* ROR *y*	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$	#1–31 or *Rs*
RRX	rotate right extended	*x* RRX	$(c\ \text{flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$	none

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

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

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

Example: 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

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	$Rd = Rn + N +$ `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	$Rd = Rn - N$

Example: This simple subtract instruction subtracts a value stored in register *r2* from a value store in register *r1*. The result is stored in register *r0*.

PRE r0 = 0x00000000

r1 = 0x00000002

r2 = 0x00000001

SUB r0, r1, r2

POST r0 = 0x00000001

Example: This reverse subtract instruction (RSB) subtracts *r1* from the constant value #0, writing the result to *r0*. This instruction use to negate numbers.

PRE r0 = 0x00000000

r1 = 0x00000077

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

POST r0 = -r1 = 0xffffff89

Example: 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 = nzcviFt_USER

r1 = 0x00000001

SUBS r1, r1, #1

POST cpsr = nZCviFt_USER

r1 = 0x00000000

Using the Barrel Shifter with Arithmetic Instructions:

Example: 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

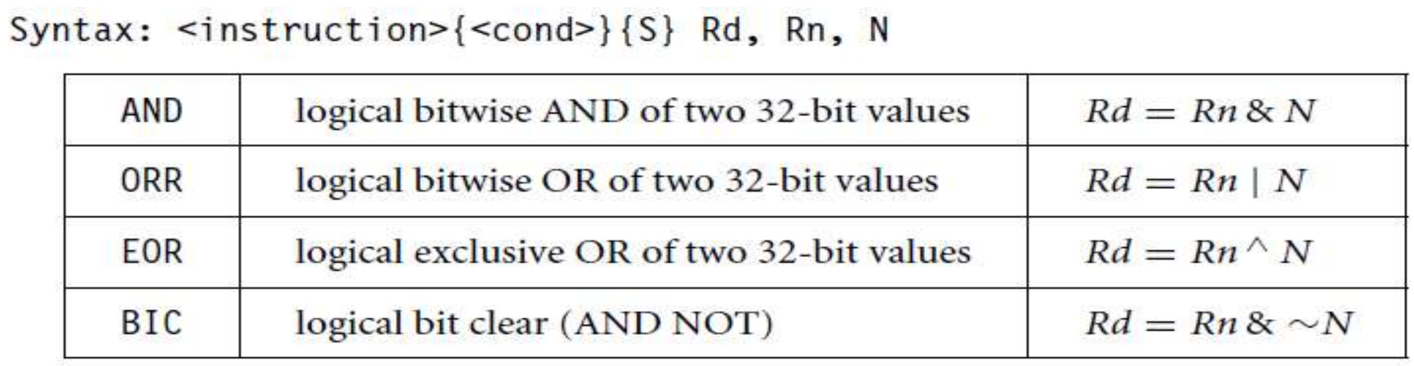
ADD r0, r1, r1, LSL #1

POST r0 = **0x0000000f**

r1 = 0x00000005

Logical Instructions:

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

Example: Shows a logical OR operation between registers *r1* and *r2*. *r0* holds the result.

PRE r0 = 0x00000000

r1 = 0x02040608

r2 = 0x10305070

ORR r0, r1, r2

POST r0 = **0x12345678**

Example: Shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

PRE r1 = 0b1111

r2 = 0b0101

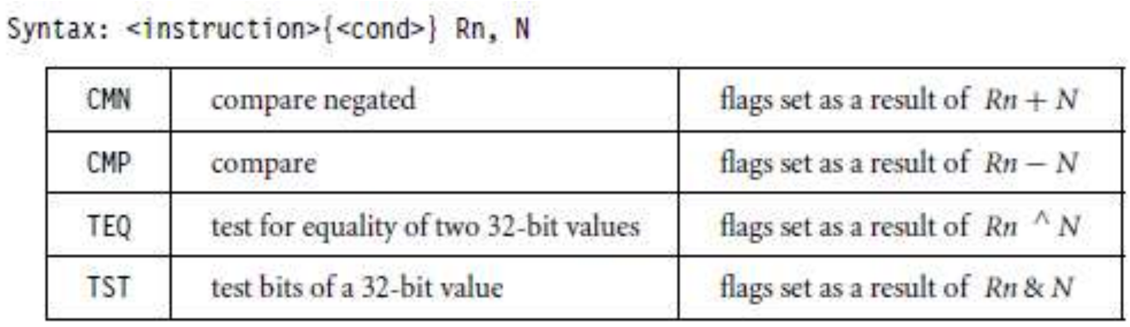
BIC r0, r1, r2

POST r0 = **0b1010**

- This is equivalent to Rd = Rn **AND NOT**(N)
- 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.

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.
- For these instructions no needs to apply the S suffix for update the flags.



```
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 ^ N
TST	test bits of a 32-bit value	flags set as a result of Rn & N

Example: This example shows a CMP comparison instruction. You can see that both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *z* flag prior to execution is 0 and is represented by a lowercase *z*. After execution the *z* flag changes to 1 or an uppercase *Z*. This change indicates *equality*.

PRE cpsr = nzcviFt_USER

r0 = 4; r9 = 4

CMP r0, r9

POST cpsr = nZcviFt_USER

- The CMP is effectively a subtract instruction with the result discarded.
- TST instruction is a logical AND operation
- TEQ is a logical exclusive OR operation.
- Foreach, 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 with another register.

- The long multiplies accumulate onto a pair of registers representing a 64-bit value. The final result is placed in a destination register or a pair of registers.

```
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	$[RdHi, RdLo] = Rm*Rs$

Example: This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into 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.
- The result is too large to fit a single 32-bit register so the result is placed in two registers labeled *RdLo* and *RdHi*.
- *RdLo* holds the lower 32 bits of the 64-bit result, and *RdHi* holds the higher 32 bits of the 64-bit result.

Example: Shows an example of a long unsigned multiply instruction. The instruction multiplies registers *r2* and *r3* and places the result into register *r0* and *r1*. Register *r0* contains the lower 32 bits, and register *r1* contains the higher 32 bits of the 64-bit result.

PRE r0 = 0x00000000

r1 = 0x00000000

r2 = 0xf0000002

r3 = 0x00000002

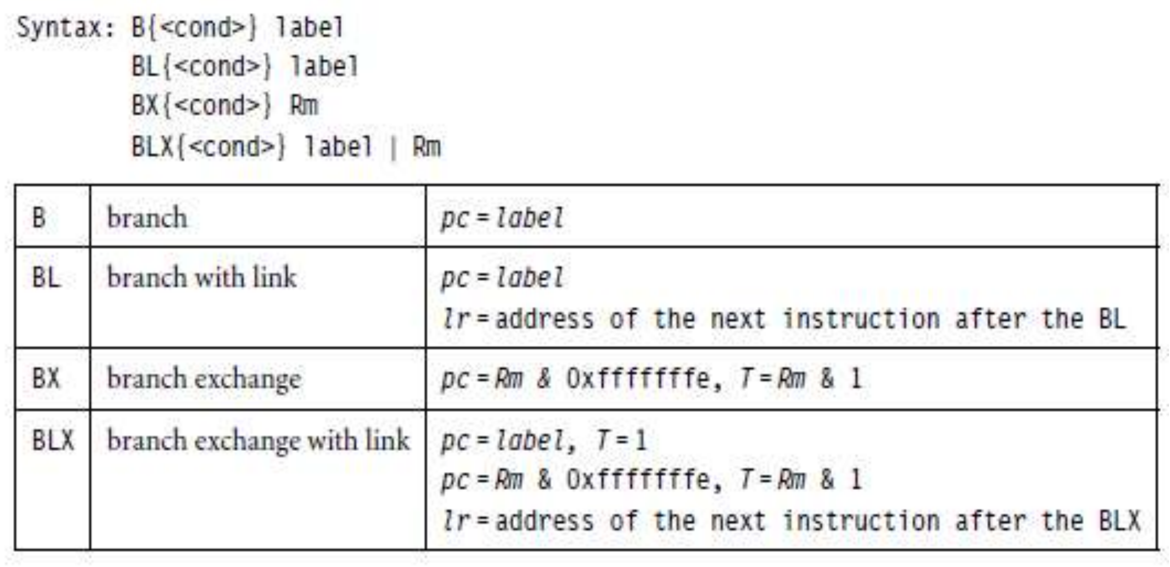
UMULL r0, r1, r2, r3 ; [r1,r0] = r2*r3

POST r0 = **0xe0000004** ; = RdLo

r1 = **0x00000001** ; = RdHi

Branch Instructions:

- A branch instruction changes the flow of execution or is used to call a routine.
- 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* = 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 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *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.

Example: This example shows a forward and backward branch.

```
        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
```

Example: The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register *lr* with a return address. It performs a subroutine call.

```
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
```

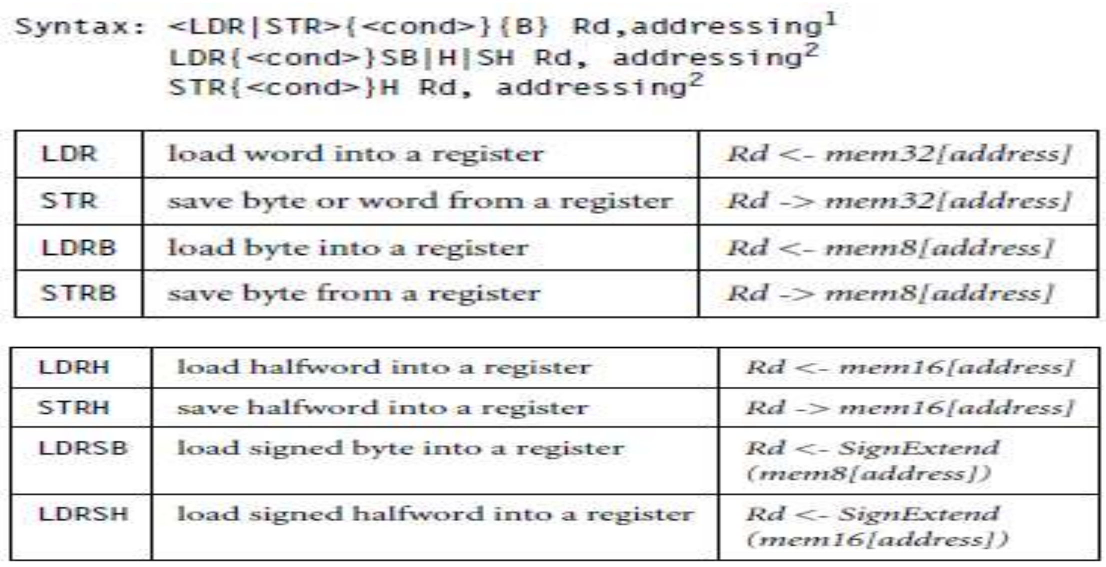
- The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction.
- It is primarily used to branch to and from Thumb code.

Load-Store 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
  - swap

Single-Register Transfer:

- These instructions are used for moving a single data item in and out of a register.
- The data types supported are signed and unsigned words (32-bit), half words (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])*

Example: LDR r0, [r1]

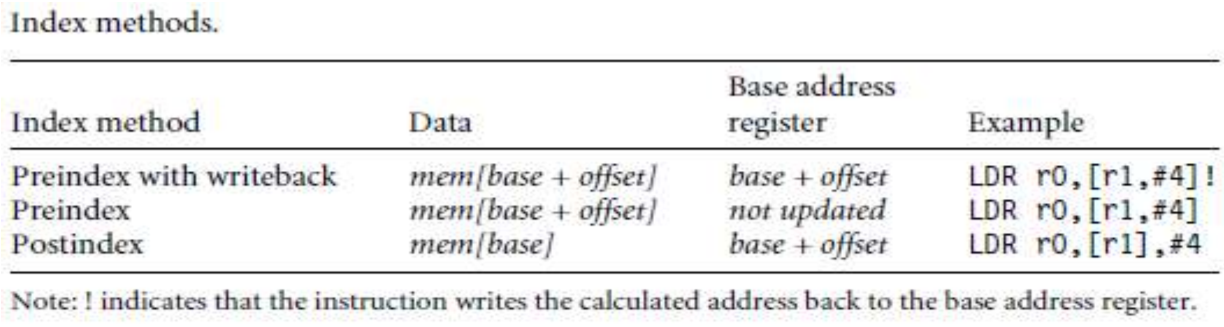
STR r0, [r1]

- The first instruction loads a word from the address stored in register *r1* and places it into register *r0*.
- The second instruction goes the other way by storing the contents of register *r0* to the address contained in register *r1*.
- Register *r1* is called the *base address register*.

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 write back,
  - Preindex
  - Postindex
- ***Preindex with write back:*** It calculates an address from a base register plus address offset and then updates that address base register with the new address.
- ***Preindex:*** It calculates an address from a base register plus address offset but does not update the address base register.
- ***Postindex***: It only updates the address base register after the address is used.

Note: The pre-index mode is useful for accessing an element in a data structure. The post index and pre index with write back modes are useful for traversing an array.


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.

- The offset address can provide in the instructions in different types. They are
  - ***Immediate:*** It means the address is calculated using the base address register and a 12-bit offset encoded in the instruction.
  - ***Register:*** It means the address is calculated using the base address register and a specific register's contents.
  - ***Scaled:*** It means the address is calculated using the base address register and a barrel shift operation.

Example: Index addressing modes

PRE r0 = 0x00000000

r1 = 0x00090000

mem32 [0x00009000] = 0x01010101

mem32 [0x00009004] = 0x02020202

<u>Preindexing with write back:</u> **LDR r0, [r1, #4]!**

POST (1) r0 = 0x02020202

r1 = 0x00009004

<u>Preindexing:</u> **LDR r0, [r1, #4]**

POST (2) r0 = 0x02020202

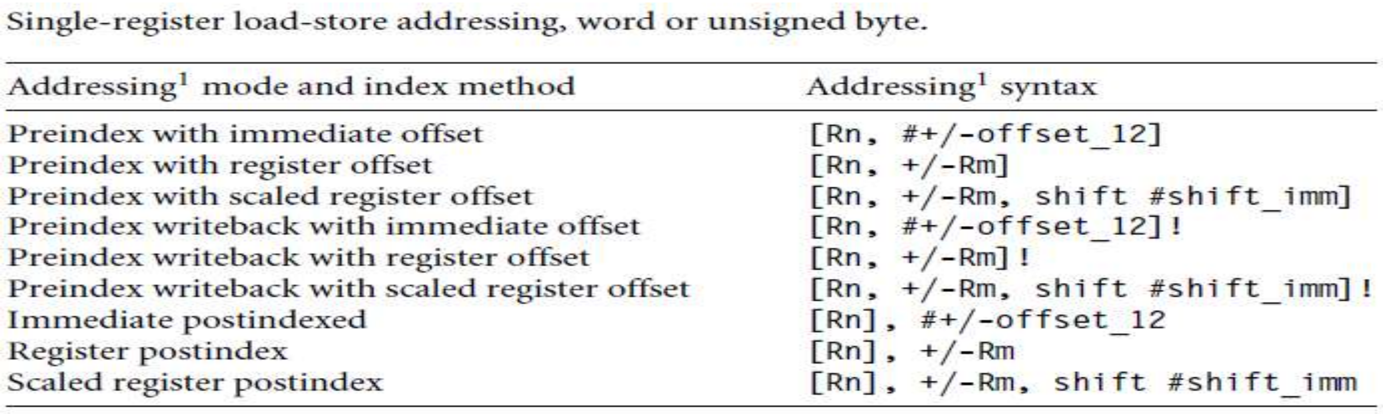
r1 = 0x00009000

Postindexing: **LDR r0, [r1], #4**

POST (3) r0 = 0x01010101

r1 = 0x00009004

- Table below shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.
- A signed offset or register is denoted by "+/–", identifying that it is either a positive or negative offset from the base address register *Rn*.
- The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.


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`

- Table below provides an example of the different variations of the LDR instruction.

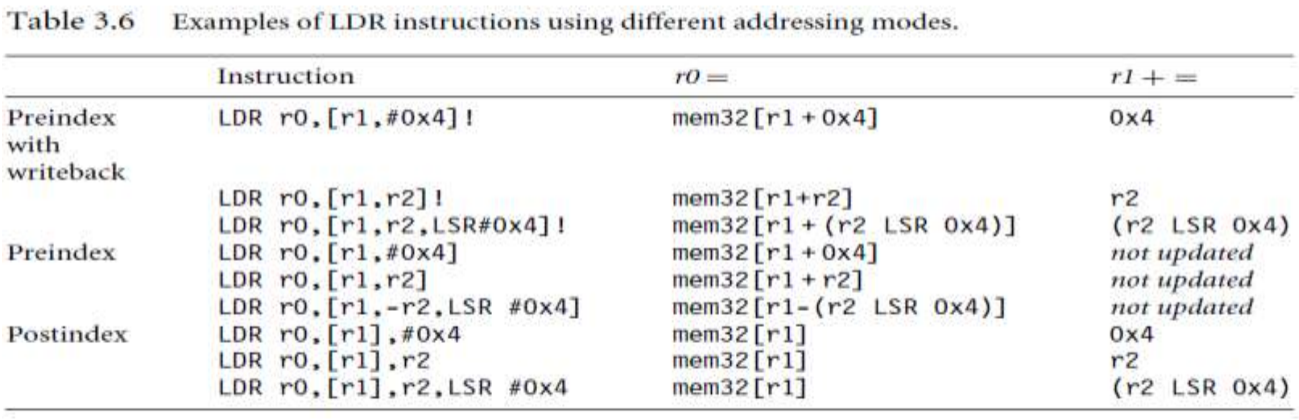
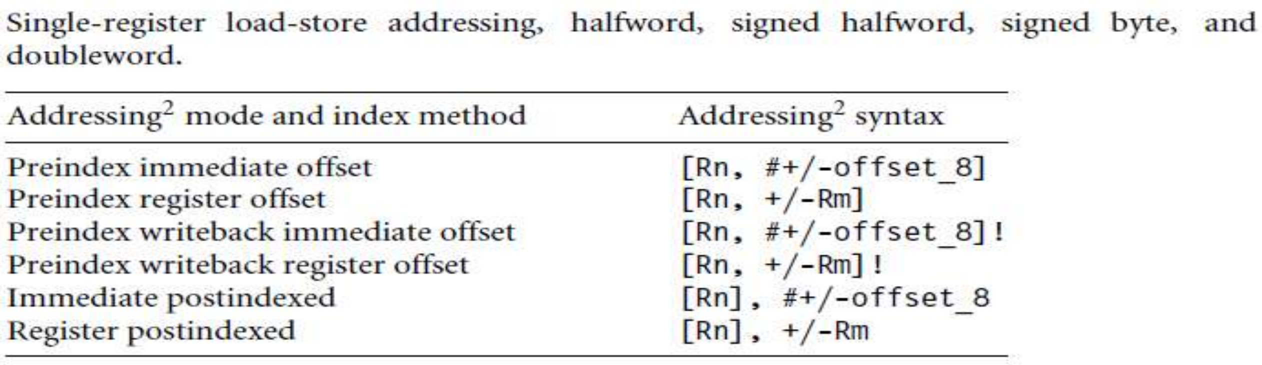


Table 3.6 Examples of LDR instructions using different addressing modes.

	Instruction	*r0* =	*r1* +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	mem32[r1 + 0x4]	0x4
	`LDR r0,[r1,r2]!`	mem32[r1+r2]	r2
	`LDR r0,[r1,r2,LSR#0x4]!`	mem32[r1 + (r2 LSR 0x4)]	(r2 LSR 0x4)
Preindex	`LDR r0,[r1,#0x4]`	mem32[r1 + 0x4]	*not updated*
	`LDR r0,[r1,r2]`	mem32[r1 + r2]	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	mem32[r1-(r2 LSR 0x4)]	*not updated*
Postindex	`LDR r0,[r1],#0x4`	mem32[r1]	0x4
	`LDR r0,[r1],r2`	mem32[r1]	r2
	`LDR r0,[r1],r2,LSR #0x4`	mem32[r1]	(r2 LSR 0x4)

- Table below shows the addressing modes available on load and store instructions using 16-bit half word or signed byte data.


Single-register load-store addressing, halfword, signed halfword, signed byte, and doubleword.

Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	`[Rn, #+/-offset_8]`
Preindex register offset	`[Rn, +/-Rm]`
Preindex writeback immediate offset	`[Rn, #+/-offset_8]!`
Preindex writeback register offset	`[Rn, +/-Rm]!`
Immediate postindexed	`[Rn], #+/-offset_8`
Register postindexed	`[Rn], +/-Rm`

- There are no STRSB or STRSH instructions since STRH store both a signed and unsigned half word; similarly STRB stores signed and unsigned bytes.

- Table below shows the variations for STRH instructions.

Variations of STRH instructions.

	Instruction	Result	*r1 + =*
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	0x4
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	r2
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	0x4
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	r2

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.
- Load-store multiple instructions can increase interrupts latency.
- ARM implementations do not usually interrupt instructions while they are executing.
- If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete.

```
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

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

Addressing mode for load-store multiple instructions.

Addressing mode	Description	Start address	End address	*Rn!*
IA	increment after	Rn	$Rn + 4^*N - 4$	$Rn + 4^*N$
IB	increment before	$Rn + 4$	$Rn + 4^*N$	$Rn + 4^*N$
DA	decrement after	$Rn - 4^*N + 4$	Rn	$Rn - 4^*N$
DB	decrement before	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

- Here *N* is the number of registers in the list of registers. The base register *Rn* determines the source or destination address for a load store multiple instruction.
- This register can be optionally updated following the transfer when register *Rn* is followed by the '!' character.

Example: Register *r0* is the base register *Rn* and is followed by !, indicating that the register is updated after the instruction is executed.

PRE	mem32 [0x80018] = 0x03	r0 = 0x00080010
	mem32 [0x80014] = 0x02	r1 = 0x00000000
	mem32 [0x80010] = 0x01	r2 = 0x00000000

r3 = 0x00000000

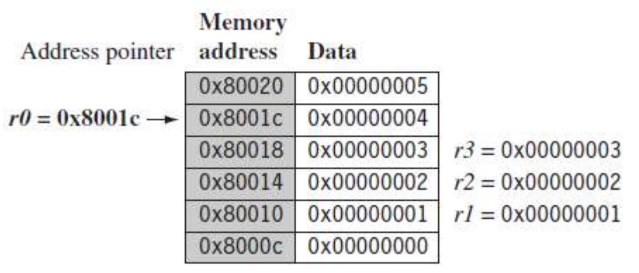
LDMIA r0!, {r1-r3}/LDMIB r0!, {r1-r3}

POST r0 = 0x0008001c

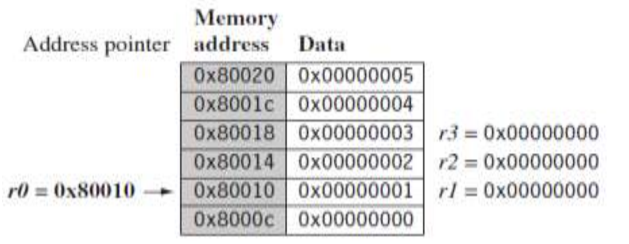
r1 = 0x00000001

r2 = 0x00000002

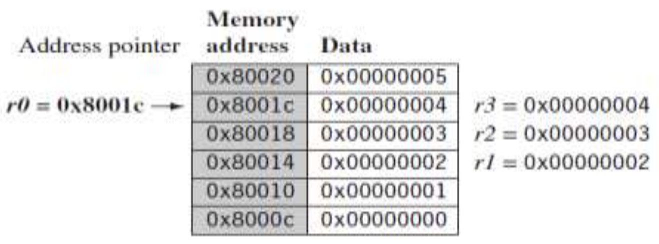
r3 = 0x00000003



Post-condition for LDMIA instruction.



Pre-condition for LDMIA instruction.



Post-condition for LDMIB instruction.

Load-store multiple pairs when base update used.

Store multiple	Load multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

- The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store to ascending memory locations.
- This is equivalent to descending memory but accessing the register list in reverse order.

Example: This example shows an STM *increment before* instruction followed by an LDM *decrement after* instruction.

PRE r0 = 0x00009000

r1 = 0x00000009

r2 = 0x00000008

r3 = 0x00000007

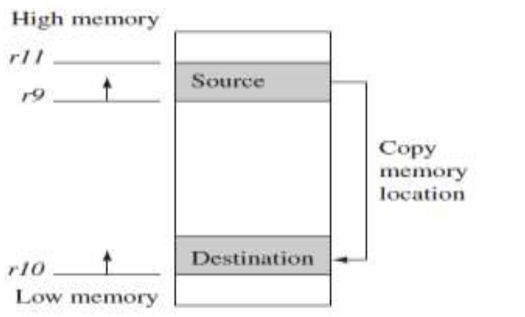
STMIB r0!, {r1-r3}

MOV r1, #1

MOV r2, #2

```
        MOV r3, #3
PRE(2)  r0 = 0x0000900c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
        LDMDA r0!, {r1-r3}
POST     r0 = 0x00009000
        r1 = 0x00000009
        r2 = 0x00000008
        r3 = 0x00000007
```

- Load-store multiple instructions with a ***block memory copy*** example. This example is a simple routine that copies blocks of 32 bytes from a source address location to a destination address location.
- The example has two load-store multiple instructions, which use the same *increment after* addressing mode.



Block memory copy in the memory map.

```
; r9 points to start of source data
; r10 points to start of destination data
; r11 points to end of the source

Loop         ; load 32 bytes from source and update r9 pointer
LDMIA r9!, {r0-r7}

; store 32 bytes to destination and update r10 pointer

STMIA r10!, {r0-r7} ; and store them

; have we reached the end

CMP r9, r11

BNE loop
```

- CMP and BNE compare pointers *r9* and *r11* to check whether the end of the block copy has been reached.

- If the block copy is complete, then the routine finishes; otherwise the loop repeats with the updated values of register *r9* and *r10*.
- The BNE is the branch instruction B with a condition mnemonic NE (not equal).

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.
  - 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.
  - Descending stacks grow towards lower memory addresses.
- 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).
- 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

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

- ARM has specified an ARM-Thumb Procedure Call Standard (ATPCS) that defines how routines are called and how registers are allocated.

Example: The STMFD instruction pushes registers onto the stack, updating the *sp*.

PRE r1 = 0x00000002

r4 = 0x00000003

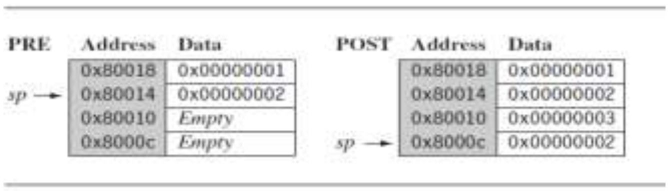
sp = 0x00080014

STMFD sp!, {r1,r4}

POST r1 = 0x00000002

r4 = 0x00000003

sp = 0x0008000c



PRE	Address	Data	POST	Address	Data
	0x80018	0x00000001		0x80018	0x00000001
sp →	0x80014	0x00000002		0x80014	0x00000002
	0x80010	Empty		0x80010	0x00000003
	0x8000c	Empty	sp →	0x8000c	0x00000002

STMFD instruction—full stack push operation.

Example: A push operation on an empty stack using the STMED instruction. The STMED instruction pushes the registers onto the stack but updates register *sp* to point to the next empty location.

PRE r1 = 0x00000002 r4 = 0x00000003

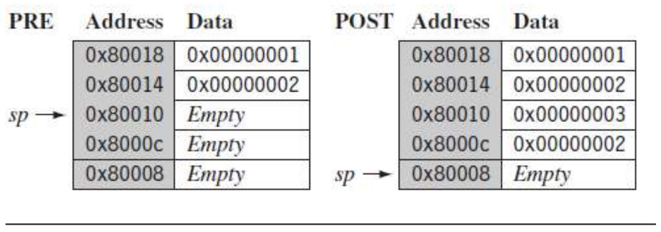
sp = 0x00080010

STMED sp!, {r1,r4}

POST r1 = 0x00000002

r4 = 0x00000003

sp = 0x00080008



STMED instruction—empty stack push operation.

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.
- 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]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

Example: The swap instruction loads a word from memory into register *r0* and overwrites the memory with register *r1*.

PRE mem32 [0x9000] = 0x12345678

r0 = 0x00000000

r1 = 0x11112222

r2 = 0x00009000

SWP r0, r1, [r2]

POST mem32 [0x9000] = **0x11112222**

r0 = 0x12345678

r1 = 0x11112222

r2 = 0x00009000

- This instruction is particularly useful when implementing semaphores and mutual exclusion in an operating system.

Software Interrupt Instruction:

- 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 *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *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.

Example: 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**

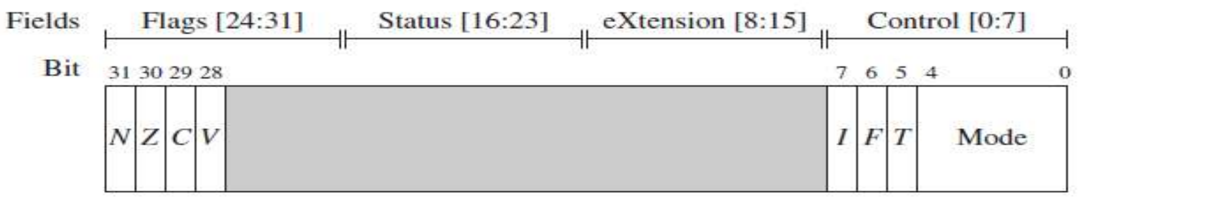
Program Status Register Instructions:

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

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

- 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 below Figure.



psr byte fields.

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 MSR first copies the *cpsr* into register *r1*. The BIC instruction clears bit 7 of *r1*.
- Register *r1* is then copied back into the *cpsr*, which enables IRQ interrupts.

PRE cpsr = nzcvqIFt_SVC

MRS r1, cpsr

BIC r1, r1, #0x80 ; 0b01000000

MSR cpsr_c, r1

POST cpsr = nzcvqiFt_SVC

<u>Loading Constants:</u>

- There is no ARM instruction to move a 32-bit constant into a register.
- To aid programming there are two pseudo instructions to move a 32-bit value into 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 first pseudo instruction writes a 32-bit constant to a register using whatever instructions are available.
- The second pseudo instruction writes a relative address into a register, which will be encoded using a *pc*-relative expression.

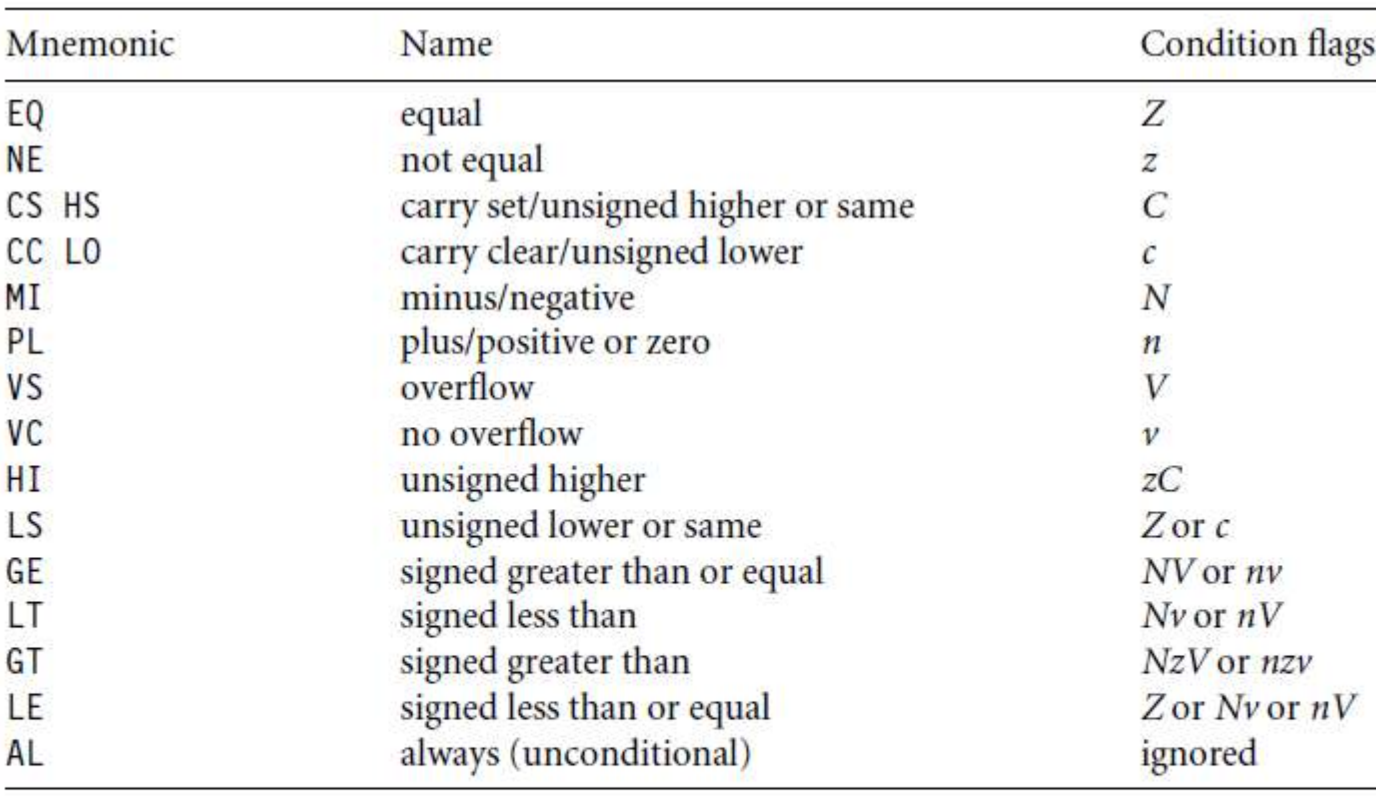
Conditional Execution:

- Most ARM instructions are conditionally executed—you can specify that the instruction only executes if the condition code flags pass a given condition or test.
- By using conditional execution instructions you can increase performance and code density.
- The condition field is a two-letter mnemonic appended to the instruction mnemonic.
- The default mnemonic is AL, or always execute.
- Conditional execution depends upon two components
    - The condition field: Is located in instruction (bit 31 – bit 28)
    - The condition flags: Are located in the *cpsr* (bit 31 – bit 28)

Example: This example shows an ADD instruction with the EQ condition appended. This instruction will only be executed when the zero flag in the *cpsr* is set to 1.

```
; r0 = r1 + r2 if zero flag is set
 ADDEQ r0, r1, r2
```

Condition mnemonics.



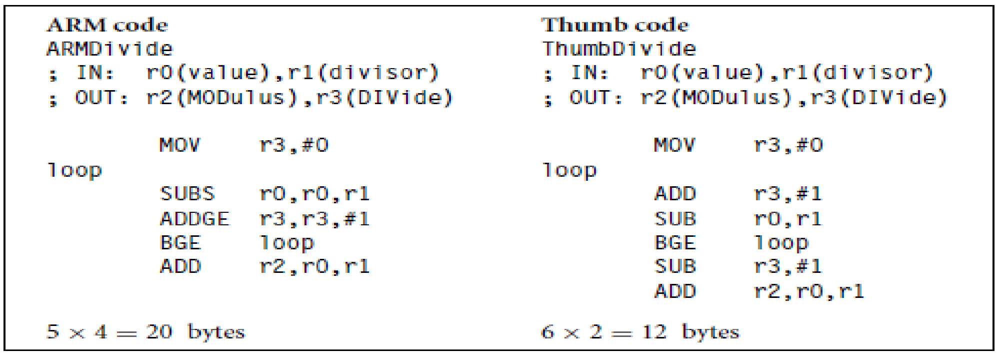
Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater than or equal	*NV* or *nv*
LT	signed less than	*Nv* or *nV*
GT	signed greater than	*NzV* or *nzv*
LE	signed less than or equal	*Z* or *Nv* or *nV*
AL	always (unconditional)	ignored

PART – B: THUMB INSTRUCTION SET

Contents:

Introduction:

- ➢ Thumb encodes a subset of the 32-bit ARM instructions into a 16-bit instruction set space.
- ➢ Thumb has higher performance than ARM on a processor with a 16-bit data bus, but lower performance than ARM on a 32-bit data bus.
- ➢ Use Thumb for memory-constrained systems
- ➢ Thumb has higher *code density*
- ➢ For memory-constrained embedded systems, for example, mobile phones and PDAs, code density is very important.
- ➢ A Thumb implementation of the same code takes up around 30% less memory than the equivalent ARM implementation.
- ➢ Figure below shows the same divide code routine implemented in ARM and Thumb assembly code.



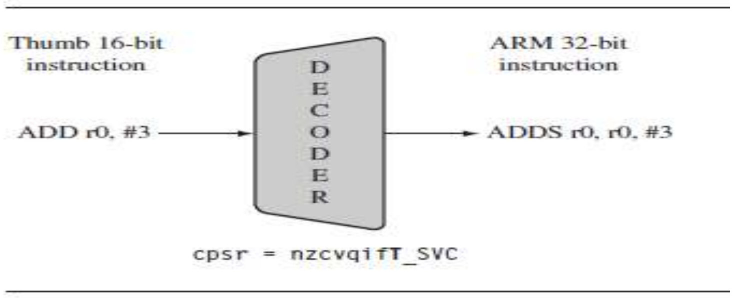
```
ARM code                                    Thumb code
ARMDivide                                   ThumbDivide
; IN:  r0(value),r1(divisor)                ; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)               ; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0                               MOV     r3,#0
loop                                        loop
        SUBS    r0,r0,r1                            ADD     r3,#1
        ADDGE   r3,r3,#1                            SUB     r0,r1
        BGE     loop                                BGE     loop
        ADD     r2,r0,r1                            SUB     r3,#1
                                                    ADD     r2,r0,r1

5 × 4 = 20 bytes                            6 × 2 = 12 bytes
```

Code density.

- Each Thumb instruction is related to a 32-bit ARM instruction.
- Figure below shows a simple Thumb ADD instruction being decoded into an equivalent ARM ADD instruction.
- The limited space available in 16 bits causes the barrel shift operations ASR, LSL, LSR, and ROR to be separate instructions in the Thumb ISA.



- Thumb instruction decoding.

Thumb Register Usage:

- In Thumb state, you do not have direct access to all registers. Only the low registers *r0* to *r7* are fully accessible, as shown in below Table.

Summary of Thumb register usage.

Registers	Access
r0–r7	fully accessible
r8–r12	only accessible by MOV, ADD, and CMP
r13 sp	limited accessibility
r14 lr	limited accessibility
r15 pc	limited accessibility
cpsr	only indirect access
spsr	no access

- The higher registers *r8 to r12* are only accessible with MOV, ADD, or CMP instructions.
- CMP and all the data processing instructions that operate on low registers update the condition flags in the *cpsr.*
- There are no MSR- and MRS-equivalent Thumb instructions.
- To alter the *cpsr* or *spsr,* you must switch into ARM state to use MSR and MRS.

ARM – Thumb Interworking:

- *ARM-Thumb interworking* is the name given to the method of linking ARM and Thumb code together for both assembly and C/C++.
- It handles the transition between the two states.
- To call a Thumb routine from an ARM routine, the core has to change state.
- This state change is shown in the *T* bit of the *cpsr.*
- The *BX* and *BLX* branch instructions cause a switch between ARM and Thumb state while branching to a routine.

```
Syntax: BX     Rm
        BLX    Rm | label
```

BX	Thumb version branch exchange	$pc = Rn$ & 0xfffffffe $T = Rn[0]$
BLX	Thumb version of the branch exchange with link	$lr =$ (instruction address after the BLX) + 1 $pc = label,\ T = 0$ $pc = Rm$ & 0xfffffffe, $T = Rm[0]$

Unlike the ARM version, the Thumb BX instruction cannot be conditionally executed.

<u>Other Branch Instructions:</u>

- There are two variations of the standard branch instruction, or B.
- The first is similar to the ARM version and is conditionally executed; the branch range is limited to a signed 8-bit immediate, or −256 to +254 bytes.
- The second version removes the conditional part of the instruction and expands the effective branch range to a signed 11-bit immediate, or −2048 to +2046 bytes (+/- 4MB).
- The conditional branch instruction is the only conditionally executed instruction in Thumb state.

```
Syntax: B<cond> label
        B label
        BL label
```

B	branch	$pc = label$
BL	branch with link	$pc = label$ $lr =$ (instruction address after the BL) + 1

<u>Data Processing Instructions:</u>

- The data processing instructions manipulate data within registers.
- They include move instructions, arithmetic instructions, shifts, logical instructions, comparison instructions, and multiply instructions.
- The Thumb data processing instructions are a subset of the ARM data processing instructions.

```
Syntax:
    <ADC|ADD|AND|BIC|EOR|MOV|MUL|MVN|NEG|ORR|SBC|SUB>  Rd, Rm
    <ADD|ASR|LSL|LSR|ROR|SUB> Rd, Rn #immediate
    <ADD|MOV|SUB> Rd,#immediate
    <ADD|SUB> Rd,Rn,Rm
     ADD Rd,pc,#immediate
     ADD Rd,sp,#immediate
    <ADD|SUB> sp, #immediate
    <ASR|LSL|LSR|ROR> Rd,Rs
    <CMN|CMP|TST> Rn,Rm
     CMP Rn,#immediate
     MOV Rd,Rn
```

ADC	add two 32-bit values and carry	$Rd = Rd + Rm + C$ flag
ADD	add two 32-bit values	$Rd = Rn + immediate$ $Rd = Rd + immediate$ $Rd = Rd + Rm$ $Rd = Rd + Rm$ $Rd = (pc \& 0xfffffffc) + (immediate \ll 2)$ $Rd = sp + (immediate \ll 2)$ $sp = sp + (immediate \ll 2)$

AND	logical bitwise AND of two 32-bit values	$Rd = Rd \& Rm$
ASR	arithmetic shift right	$Rd = Rm \gg immediate,$ C flag $= Rm[immediate - 1]$ $Rd = Rd \gg Rs, C$ flag $= Rd[Rs - 1]$
BIC	logical bit clear (AND NOT) of two 32-bit values	$Rd = Rd$ AND NOT(Rm)
CMN	compare negative two 32-bit values	$Rn + Rm$ *sets flags*
CMP	compare two 32-bit integers	$Rn - immediate$ *sets flags* $Rn - Rm$ *sets flags*
EOR	logical exclusive OR of two 32-bit values	$Rd = Rd$ EOR Rm
LSL	logical shift left	$Rd = Rm \ll immediate,$ C flag $= Rm[32 - immediate]$ $Rd = Rd \ll Rs, C$ flag $= Rd[32 - Rs]$
LSR	logical shift right	$Rd = Rm \gg immediate,$ C flag $= Rd[immediate - 1]$ $Rd = Rd \gg Rs, C$ flag $= Rd[Rs - 1]$
MOV	move a 32-bit value into a register	$Rd = immediate$ $Rd = Rn$ $Rd = Rm$
MUL	multiply two 32-bit values	$Rd = (Rm * Rd)[31:0]$
MVN	move the logical NOT of a 32-bit value into a register	$Rd =$ NOT(Rm)
NEG	negate a 32-bit value	$Rd = 0 - Rm$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rd$ OR Rm
ROR	rotate right a 32-bit value	$Rd = Rd$ RIGHT_ROTATE $Rs,$ C flag $= Rd[Rs-1]$
SBC	subtract with carry a 32-bit value	$Rd = Rd - Rm -$ NOT$(C$ flag$)$
SUB	subtract two 32-bit values	$Rd = Rn - immediate$ $Rd = Rd - immediate$ $Rd = Rn - Rm$ $sp = sp - (immediate \ll 2)$
TST	test bits of a 32-bit value	Rn AND Rm *sets flags*

Single Register Load – Store Instructions:

- The Thumb instruction set supports load and storing registers, or LDR and STR.
- These instructions use two preindexed addressing modes:
    - Offset by register
    - Offset by immediate

```
Syntax: <LDR|STR>{<B|H>} Rd, [Rn,#immediate]
        LDR{<H|SB|SH>} Rd,[Rn,Rm]
        STR{<B|H>} Rd,[Rn,Rm]
        LDR Rd,[pc,#immediate]
        <LDR|STR> Rd,[sp,#immediate]
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save 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])*

Addressing modes.

Type	Syntax
Load/store register	[Rn, Rm]
Base register + offset	[Rn, #immediate]
Relative	[pc|sp, #immediate]

Multiple Register Load – Store Instructions:

- The Thumb versions of the load-store multiple instructions are reduced forms of the ARM load-store multiple instructions.
- They only support the increment after (IA) addressing mode.

```
Syntax : <LDM|STM>IA Rn!, {low Register list}
```

LDMIA	load multiple registers	$\{Rd\}^{*N}$ <- mem32[Rn + 4*N], Rn = Rn + 4*N
STMIA	save multiple registers	$\{Rd\}^{*N}$ -> mem32[Rn + 4*N], Rn = Rn + 4*N

Stack Instructions:

- The Thumb stack operations are different from the equivalent ARM instructions because they use the more traditional POP and PUSH concept.

```
Syntax: POP {low_register_list{, pc}}
        PUSH {low_register_list{, lr}}
```

POP	pop registers from the stacks	Rd*N <- mem32[sp + 4*N], sp = sp + 4*N
PUSH	push registers on to the stack	Rd*N -> mem32[sp + 4*N], sp = sp − 4*N

- The interesting point to note is that there is no stack pointer in the instruction.
- This is because the stack pointer is fixed as register *r13 in* Thumb operations and sp is automatically updated.
- The list of registers is limited to the low registers *r0 to r7.*
- The stack instructions only support full descending stack operations.

Software Interrupt Instructions:

- The Thumb software interrupt (SWI) instruction causes a software interrupt exception.
- If any interrupt or exception flag is raised in Thumb state, the processor automatically reverts back to ARM state to handle the exception.
- The Thumb SWI instruction has the same effect and nearly the same syntax as the ARM equivalent.
- It differs in that the SWI number is limited to the range 0 to 255 and it is not conditionally executed.

```
Syntax: SWI immediate
```

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts) *cpsr T* = 0 (ARM state)