



A T M E

College of Engineering



MICROCONTROLLER AND EMBEDDED SYSTEMS-18CS44

Module-2 Introduction to ARM Instruction Sets

Dr. Puttegowda D

Professor and Head

Dept. of Computer Science & Engineering

ATMECE, Mysuru

ARM Instruction Set

Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative ± 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (<i>cpsr</i> or <i>spsr</i>)
MSR	v3	move to a status register (<i>cpsr</i> or <i>spsr</i>) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values

SBC	v1	subtract with carry of two 32-bit values
SMLAxy	v5E	signed multiply accumulate instructions $((16 \times 16) + 32 = 32\text{-bit})$
SMLAL	v3M	signed multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
SMLALxy	v5E	signed multiply accumulate long $((16 \times 16) + 64 = 64\text{-bit})$
SMLAWy	v5E	signed multiply accumulate instruction $((32 \times 16) \gg 16) + 32 = 32\text{-bit})$
SMULL	v3M	signed multiply long $(32 \times 32 = 64\text{-bit})$
<hr/>		
SMULxy	v5E	signed multiply instructions $(16 \times 16 = 32\text{-bit})$
SMULWy	v5E	signed multiply instruction $((32 \times 16) \gg 16 = 32\text{-bit})$
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
UMULL	v3M	unsigned multiply long $(32 \times 32 = 64\text{-bit})$

Data Processing Instructions

- The data processing instructions manipulate data within registers. They are
 - move instructions,
 - arithmetic instructions,
 - logical instructions,
 - comparison instructions, and
 - 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

- 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	$Rd = \sim N$

Examples

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

Condition Embedded in ADD Instruction

{cond} Rd, N

MOVEQ r0, r1 ; If zero flag set then...
; ... r0 = r1

Example

AREA MOV1, CODE, READONLY

START

MOV R0,#5

MOV R1,#6

SUBS R0,R0,#4 ; FIRST SUBTRACT 5 AND NEXT 4

MOVEQ R2,R1

MOV R3,#10

BACK B BACK

END

- For example to move the value zero to destination register and set the condition flags:

MOV**S** Rd, N

MOV r0,r1 ; r0 = r1

; ... and set flags

Example

AREA MOV2, CODE, READONLY

START

MOV R0,#0

MOV R1,#6

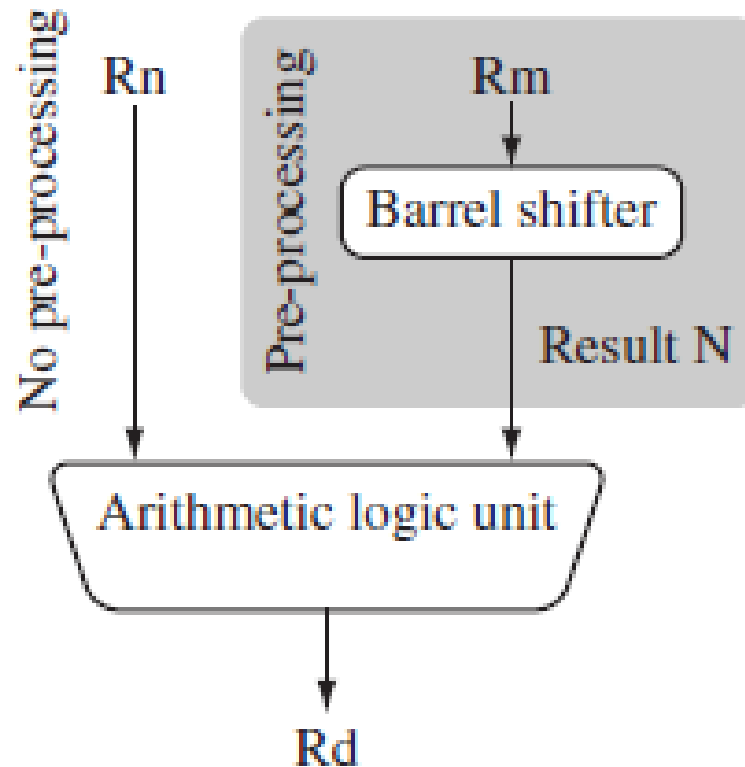
MOVS R2,R0 ; Z

MOVS R3,R1 ; z

BACK B BACK

END

Barrel Shifter



Barrel shifter and ALU.

- In MOV instruction *N is a simple register/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.
- Majority of the data processing instructions are processed within the arithmetic logic unit (ALU).
- **A unique and powerful feature of the ARM processor** is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- This shift increases the power and flexibility of many data processing operations.
- There are data processing instructions that do not use the barrel shift, for example,
 - the MUL (multiply),
 - CLZ (count leading zeros), and
 - QADD (signed saturated 32-bit add)

Example

PRE $r5 = 5$
 $r7 = 8$

MOV $r7, r5, LSL \#2$; let $r7 = r5 * 4 = (r5 \ll 2)$

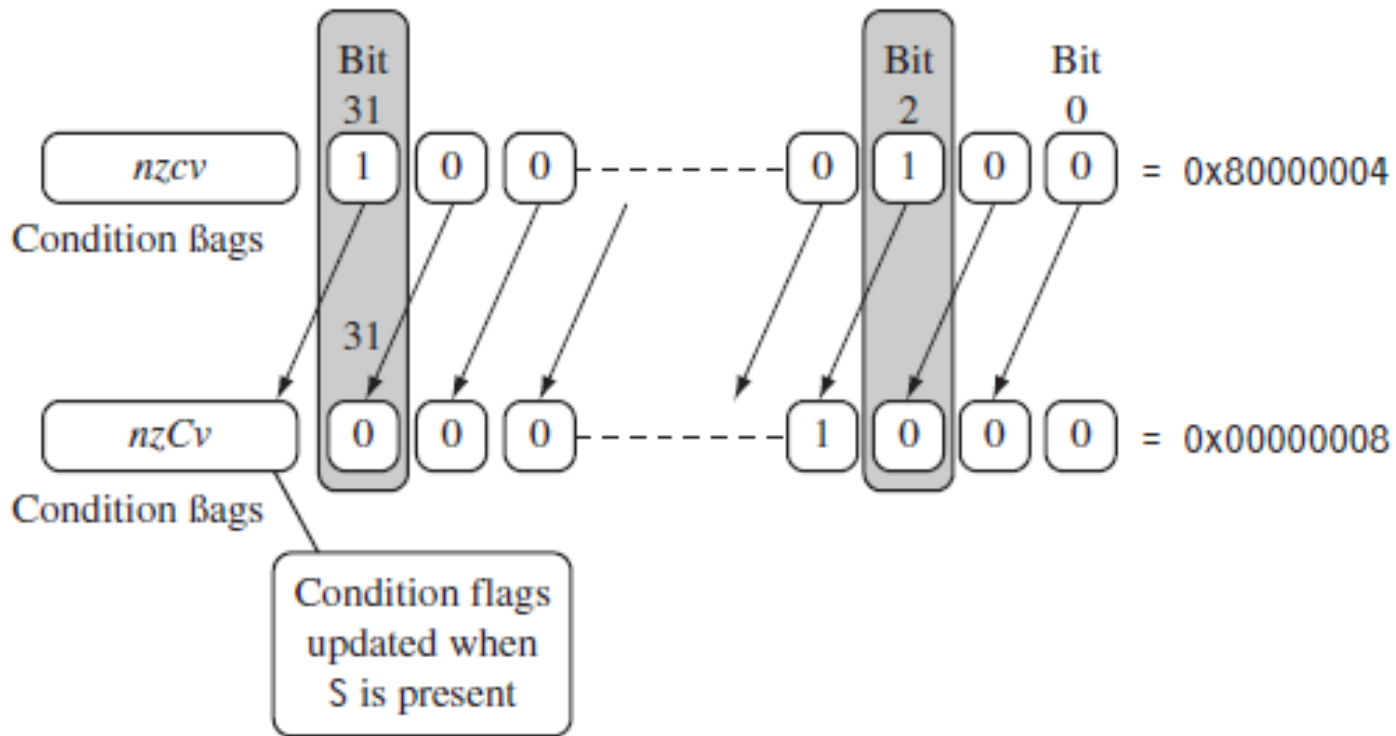
POST $r5 = 5$
 $r7 = 20$

Barrel shifter operations

Table 3.2 Barrel shifter operations.

Mnemonic	Description	Shift
LSL	logical shift left	$x\text{LSL } y$
LSR	logical shift right	$x\text{LSR } y$
ASR	arithmetic right shift	$x\text{ASR } y$
ROR	rotate right	$x\text{ROR } y$
RRX	rotate right extended	$x\text{RRX}$

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



Logical shift left by one.

Barrel Shifter - Left Shift

- Shifts left by the specified amount
- Multiplies by powers of two
- e.g.

LSL #5 = multiply by 32

Logical Shift Left (LSL)

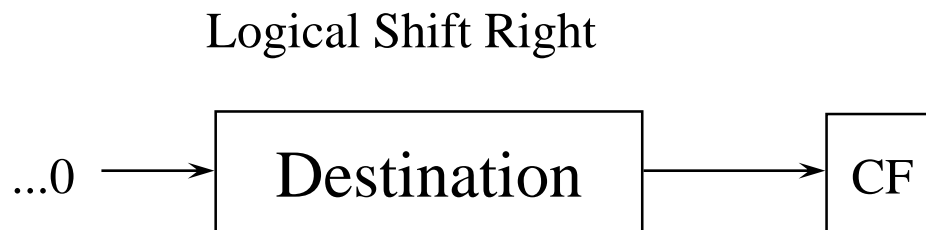


Barrel Shifter - Right Shifts

Logical Shift Right

- Shifts right by the specified amount (divides by powers of two)
- e.g.

LSR #5 = divide by 32



Arithmetic Shift Right

Arithmetic Shift Right

- Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations.
- e.g.

ASR #5 = divide by 32



Sign bit shifted in

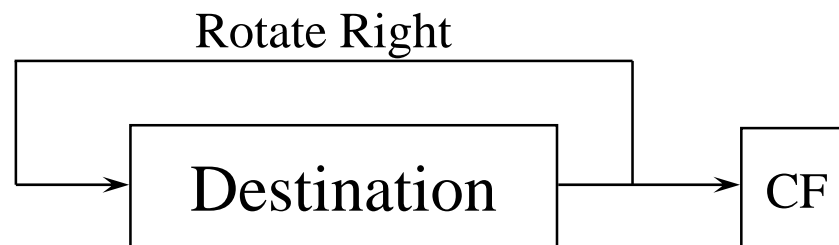
Barrel Shifter - Rotations

Rotate Right (ROR)

- Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

e.g. ROR #5

- Note the last bit rotated is also used as the Carry Out.

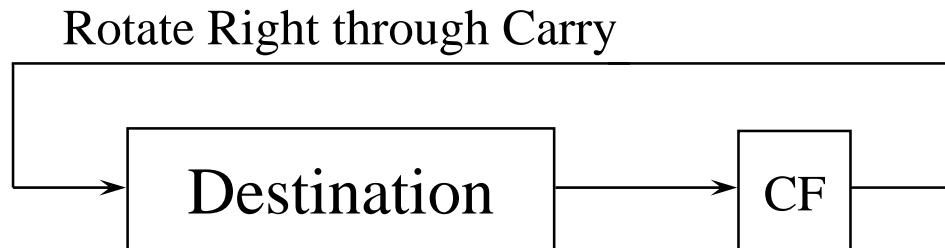


Barrel Shifter - Rotations

Rotate Right Extended (RRX)

Rotate Right Extended (RRX)

- This operation uses the CPSR C flag as a 33rd bit.
- Rotates right by 1 bit. Encoded as ROR #0.



MOVS - example

PRE `cpsr = nzcvtqifT_USER`

`r0 = 0x00000000`

`r1 = 0x80000004`

`MOVS r0, r1, LSL #1`

POST `cpsr = nzcvtqifT_USER`

`r0 = 0x00000008`

`r1 = 0x80000004`

Barrel shift operation syntax for data processing instructions.

<i>N</i> 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

Arithmetic Instructions

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	$Rd = Rn - N$

Example-1

- 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`

Example-2

- 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 = 0xffffffff89
```

Example-3

- 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 = nzcvtqifT_USER  
        r1 = 0x00000001
```

```
SUBS r1, r1, #1
```

```
POST   cpsr = nZCvtqifT_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.

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

```
ADD     r0, r1, r1, LSL #1
```

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



A T M E

College of Engineering

Example-1



AREA MOV2, CODE, READONLY

START

MOV R0,#5

MOV R1,#6

add R2,R0,R1, LSL #0x2

BACK B BACK

END

Example-2

AREA ADD1, CODE, READONLY

START

MOV R0,#5

MOV R1,#6

SUBS R0,R0,#4 ; FIRST SUBTRACT 5 AND NEXT 4

ADDEQ R2,R1,R0

MOV R3,#10

BACK B BACK

END

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

Example-1

- 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-2

- 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

$$R_d = R_n \text{ AND NOT}(N)$$

Example-3

AREA ADD1, CODE, READONLY

START

MOV R0,#5

MOV R1,#6

AND R2,R1,R0

BACK B BACK

END

Example-4

AREA ADD1, CODE, READONLY

START

MOV R0,#5

MOV R1,#6

AND R2,R1,R0

BACK B BACK

END

Example-5

AREA ADD1, CODE, READONLY

START

mov R0,#15

mov R1,#5

bic R2,R0,R1 ; r2=r0 & (~R1)

BACK B BACK

END

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 then 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 $Rn \& N$

CMP Example

```
PRE      cpsr = nzcvtqifT_USER
          r0 = 4
          r9 = 4

          CMP    r0, r9

POST     cpsr = nZcvtqifT_USER
```

CMN Instruction

- The CMN instruction adds the value of N to the value in Rn .
- This is the same as an ADDS instruction, except that the result is discarded.
- The conditional flags are updated accordingly.

RO=50

R1=10

CMN RO, R1

TEQ

- Test Equivalence.
- The TEQ instruction performs a bitwise Exclusive OR operation on the value in Rn and the value of N . the result is discarded.
- Use the TEQ instruction to test if two values are equal, without affecting the V or C flags (as CMP does). It affects the N and Z flags.

TEQ- Example

AREA TEST, CODE, READONLY
ENTRY

START

MOV R0,#6

MOV R1,#5

TEQ R0,R1

BACK B BACK

END

TST - Instruction

- This instruction tests the value in a register against N. It updates the condition flags on the result, but does not place the result in any register.
- The TST instruction performs a bitwise AND operation on the value in Rn and the value of N.
- This is the same as an ANDS instruction, except that the result is discarded.

TST – EXAMPLE-1

AREA TEST, CODE, READONLY

ENTRY

START

MOV R0,#6

MOV R1,#9

TST R0,R1 ; 6 & 9 = 0 Z

BACK B BACK

END

TST – EXAMPLE-2

AREA TEST, CODE, READONLY
ENTRY

START

MOV R0,#6

MOV R1,#5

TST R0,R1 ; 6 & 5 = 4 z

BACK B BACK

END

Multiply Instructions

- The multiply instructions multiply the contents of a pair of registers.

Syntax: $\text{MLA}\{\text{<cond>}\}\{S\} \text{ Rd, Rm, Rs, Rn}$
 $\text{MUL}\{\text{<cond>}\}\{S\} \text{ Rd, Rm, Rs}$

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

MLA- EXAMPLE-1

AREA TEST, CODE, READONLY
ENTRY

START

MOV R0,#6

MOV R1,#5

MOV R2,#1

MLA R3,R0,R1,R2

$;R3=(R0 \times R1) + R2$

BACK B BACK

END

MLA- EXAMPLE-2

AREA TEST, CODE, READONLY
ENTRY

START

MOV R0,#6

MOV R1,#5

MOV R2,#1

MLA R0,R0,R1,R2 ; $R0 = (R0 * R1) + R2$ NOT ALLOWED

BACK B BACK

END

MLA- EXAMPLE-3

AREA TEST, CODE, READONLY
ENTRY

START

MOV R0,#6

MOV R1,#5

MOV R2,#1

MLA R1,R0,R1,R2

;R1=(R0*R1) + R2 ALLOWED

BACK B BACK

END

MLA- EXAMPLE-4

AREA TEST, CODE, READONLY
ENTRY

START

MOV R0,#6

MOV R1,#5

MOV R2,#1

MLA R2,R0,R1,R2

;R2=(R0*R1) + R2 ALLOWED

BACK B BACK

END

MUL – EXAMPLE-1

AREA TEST, CODE, READONLY
ENTRY

START

MOV R0,#6

MOV R1,#5

MUL R3,R0,R1 ;R3=R0*R1

BACK B BACK

END

MUL- EXAMPLE-2

AREA TEST, CODE, READONLY
ENTRY

START

MOV R0,#6

MOV R1,#5

MUL R0,R0,R1 ; THIS REGISTER SEQUENCE NOT
;ALLOWED

BACK B BACK

END

MUL- EXAMPLE-2

AREA TEST, CODE, READONLY

ENTRY

START

MOV R0,#6

MOV R1,#5

MUL R1,R0,R1 ; THIS REGISTER SEQUENCE IS
;ALLOWED BECAUSE N IS COPIED TO TEMP REG

BACK B BACK

END

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	$[RdHi, RdLo] = Rm * Rs$

SMLAL - EXAMPLE

AREA TEST, CODE, READONLY
ENTRY

START

ldr R0,=0x5

ldr R1,=0x5

LDR R2,=0X2

LDR R3,=0X4

SMLAL R3,R2,R0,R1 ;[R3Lo,R2Hi]=[R3,R2]+(R0*R1)

BACK B BACK

END

SMULL - EXAMPLE

AREA TEST, CODE, READONLY
ENTRY

START

ldr R0,=0x5

ldr R1,=0x5

LDR R2,=0X2

LDR R3,=0X4

SMULL R3,R2,R0,R1 ;[R3Lo,R2Hi]=(R0*R1)

BACK B BACK

END

UMULL- Example

AREA UMULL1, CODE, READONLY

START

ldr R0,=0xf0000002

ldr R1,=0x00000002

umull r3,R2,R1,R0

MOV R3,#10

BACK B BACK

END

UMLAL- Example

AREA TEST, CODE, READONLY
ENTRY

START

ldr R0,=0x5

ldr R1,=0x5

LDR R2,=0X2

LDR R3,=0X4

UMLAL R3,R2,R0,R1 ;[RdHi, RdLo] = [RdHi, RdLo] +
(Rm *Rs)

BACK B BACK

END

Branch Instructions

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

- *The label is the address 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.*

Forward and Backward branch

- These loops are address specific, we do not include the pre- and post-conditions.
- Following example 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
```

BL (Branch with Link instruction)

- BL instruction is similar to the B instruction but overwrites the link register *lr* with a return address.
- *It performs a subroutine call.*

pc = label

lr = address of the next instruction after the BL

- *Example:*

```
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
```

BX (Branch Exchange), BLX (Branch Exchange with Link)

- The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction.
- The BX instruction uses an absolute address stored in register *Rm*.
- *It* is 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.

$$pc = Rm \& 0xfffffffffe, T = Rm \& 1$$

- Similarly the BLX instruction updates the *T* bit of the *cpsr* with the least significant bit and additionally sets the link register with the return address.
 - $pc = label, T = 1$
 - $pc = Rm \& 0xfffffffffe, T = Rm \& 1$
 - *lr* = address of the next instruction after the BLX

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, and
 - swap.

Single-Register Transfer

- 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	$Rd \leftarrow mem32[address]$
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	$Rd \leftarrow SignExtend(mem16[address])$

- For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on.
- This example shows a load from a memory address contained in register *r1*, *followed by a store back to the same address* in memory.

```
;
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
        LDR    r0, [r1]           ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address pointed to by
; register r1.
;
        STR    r0, [r1]           ; = STR r0, [r1, #0]
```


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 methods.

Index method	Data	Base address register	Example
Preindex with writeback	$mem[base + offset]$	$base + offset$	LDR r0, [r1, #4] !
Preindex	$mem[base + offset]$	<i>not updated</i>	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.

Preindexing with writeback: (Base Register Updated)

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

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

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

Preindexing: (Base Register not updated)

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

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

Preindexing:

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

Postindexing: (Base Register Updated)

```
PRE      r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x0009000] = 0x01010101
          mem32[0x0009004] = 0x02020202
          LDR      r0, [r1], #4
```

Postindexing:

```
POST(3)  r0 = 0x01010101
          r1 = 0x00090004
```

Table 3.6 Examples of LDR instructions using different addressing modes.

	Instruction	$r0 =$	$r1 + =$
Preindex with writeback	LDR $r0, [r1, \#0x4]!$	$\text{mem32}[r1 + 0x4]$	$0x4$
	LDR $r0, [r1, r2]!$	$\text{mem32}[r1 + r2]$	$r2$
Preindex	LDR $r0, [r1, r2, \text{LSR} \#0x4]!$	$\text{mem32}[r1 + (r2 \text{ LSR } 0x4)]$	$(r2 \text{ LSR } 0x4)$
	LDR $r0, [r1, \#0x4]$	$\text{mem32}[r1 + 0x4]$	<i>not updated</i>
	LDR $r0, [r1, r2]$	$\text{mem32}[r1 + r2]$	<i>not updated</i>
Postindex	LDR $r0, [r1, -r2, \text{LSR} \#0x4]$	$\text{mem32}[r1 - (r2 \text{ LSR } 0x4)]$	<i>not updated</i>
	LDR $r0, [r1], \#0x4$	$\text{mem32}[r1]$	$0x4$
	LDR $r0, [r1], r2$	$\text{mem32}[r1]$	$r2$
	LDR $r0, [r1], r2, \text{LSR} \#0x4$	$\text{mem32}[r1]$	$(r2 \text{ LSR } 0x4)$

Table 3.8 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	<i>not updated</i>
	STRH r0,[r1,r2]	mem16[r1+r2]=r0	<i>not updated</i>
Postindex	STRH r0,[r1],#0x4	mem16[r1]=r0	0x4
	STRH r0,[r1],r2	mem16[r1]=r0	r2

Multiple-Register Load-Store Instructions

- 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.
- Load-store multiple instructions increases interrupt latency.

- For example, on an ARM7 a load multiple instruction takes
 - $2 + Nt$ cycles,
 - where N is the number of registers to load
 - t is the number of cycles required for each sequential access to memory.
- *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

Addressing mode for 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	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

Example

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`

Graphical Representation (IA).

Address pointer	Memory address	Data	
	0x80020	0x00000005	
$r0 = 0x8001c \rightarrow$	0x8001c	0x00000004	
	0x80018	0x00000003	$r3 = 0x00000003$
	0x80014	0x00000002	$r2 = 0x00000002$
	0x80010	0x00000001	$r1 = 0x00000001$
	0x8000c	0x00000000	

: Post-condition for LDMIA instruction.

Graphical Representation (IB).

Memory		
Address pointer	address	Data
	0x80020	0x00000005
$r0 = 0x8001c \rightarrow$	0x8001c	0x00000004
	0x80018	0x00000003
	0x80014	0x00000002
	0x80010	0x00000001
	0x8000c	0x00000000

$r3 = 0x00000004$

$r2 = 0x00000003$

$r1 = 0x00000002$

Post-condition for LDMIB instruction.

Load-store multiple pairs when base update used.

Load-store multiple pairs when base update used.

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



A T M E

College of Engineering



EXAMPLE-1 LDMIA

AREA ASCENDING , CODE, READONLY

ENTRY ;Mark first instruction to execute
START

LDR R2,=CVALUE ; ADDRESS OF CODE REGION
LDMIA R2!,{R3-R6}

BACK B BACK

CVALUE

```
DCD    0X44444444 ;
DCD    0X11111111 ;
DCD    0X33333333 ;
DCD    0X22222222 ;
```

END

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	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

EXAMPLE-2 LDMIB

AREA ASCENDING , CODE, READONLY

ENTRY ;Mark first instruction to execute
START

LDR R2,=CVALUE ; ADDRESS OF CODE REGION
LDMIB R2!,{R3-R6}

BACK B BACK

CVALUE

DCD 0X44444444 ;
DCD 0X11111111 ;
DCD 0X33333333 ;
DCD 0X22222222 ;

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	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

END

EXAMPLE (STMIA)

AREA ASCENDING , CODE, READONLY

ENTRY ;Mark first instruction to execute

START

LDR R2,=CVALUE ; ADDRESS OF CODE REGION

LDMIA R2!,{R3-R6}

LDR R7,=DVALUE

STMIA R7!,{R3-R6}

BACK B BACK

CVALUE

```
DCD 0X44444444 ;
DCD 0X11111111 ;
DCD 0X33333333 ;
DCD 0X22222222 ;
```

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	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

AREA DATA1, DATA, READWRITE

DVALUE DCD 0X00

END

EXAMPLE (STMIB and LDMDA)

AREA ASCENDING , CODE, READONLY

ENTRY
START

;Mark first instruction to execute

```
LDR R0,=0X40000000
LDR R1,=0x00000009
LDR R2,=0x00000008
LDR R3,=0x00000007
STMIB r0!, {r1-r3}
MOV r1, #1
MOV r2, #2
MOV r3, #3
LDMDA r0!, {r1-r3}
```

BACK B BACK

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

$Rn - 4*N$

$Rn - 4$

$Rn - 4*N$

EXAMPLE (STMIB and LDMDB)

AREA ASCENDING , CODE, READONLY

ENTRY
START

;Mark first instruction to execute

```
LDR R0,=0X40000000
LDR R1,=0x00000009
LDR R2,=0x00000008
LDR R3,=0x00000007
STMIB r0!, {r1-r3}
MOV r1, #1
MOV r2, #2
MOV r3, #3
LDMDB r0!, {r1-r3}
```

BACK B BACK

end

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	$Rn - 4 * N$	$Rn - 4$	$Rn - 4 * N$

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*
- We 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.

- In *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 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).

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

Example-STMFD

- The STMFD instruction pushes registers onto the stack, updating the *sp*. *Below example push onto a full descending stack.*
- When the stack grows the stack pointer (*sp*) points to the **last full entry in the stack**. Hence it is called as full and descending because address decreases.

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

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

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

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

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

Example-STMED

- The STMED instruction pushes the registers onto the stack but updates register *sp* to point to the next empty location downward.

```
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

Example-1

POPING ONE AT A TIME

AREA ASCENDING , CODE, READONLY

ENTRY ;Mark first instruction to execute

START

LDR SP,=0X40000000

LDR R1,=0x00000009

LDR R2,=0x00000008

LDR R3,=0x00000007

STMFA SP!, {r1-r3}

LDMFA SP!,{R4}

LDMFA SP!,{R5}

LDMFA SP!,{R6}

BACK B BACK

END

Note: STMFA == STMIB, LDMFA == LDMDA

Example-2

POPING ALL 3 ELEMENTS

AREA ASCENDING , CODE, READONLY

ENTRY ;Mark first instruction to execute
START

```
LDR SP,=0X40000000  
LDR R1,=0x00000009  
LDR R2,=0x00000008  
LDR R3,=0x00000007  
STMFA SP!, {r1-r3}  
LDMFA SP!,{R4-R6}
```

BACK B BACK
END

Example-3

POPING FROM EMPTY STACK

AREA ASCENDING , CODE, READONLY

ENTRY ;Mark first instruction to execute
START

```
LDR SP,=0X40000000
LDR R1,=0x00000009
LDR R2,=0x00000008
LDR R3,=0x00000007
STMFA SP!, {r1-r3}
LDMFA SP!,{R4-R6}
LDMFA SP!,{R7} ; UNDER FLOW
```

BACK B BACK
END

Example-4

Full Descending Stack

AREA ASCENDING , CODE, READONLY

ENTRY ;Mark first instruction to execute
START

```
LDR SP,=0X4000001F
LDR R1,=0x00000009
LDR R2,=0x00000008
LDR R3,=0x00000007
STMFD SP!, {r1-r3}
LDMFD SP!,{R4-R6}
LDMFD SP!,{R7}
```

BACK B BACK
END

Checking Stack Under Flow

- In handling the stack there are three attributes that need to be preserved:
 - The *stack base*,
 - *the stack pointer*, and
 - *the stack limit*.
- *The stack base is the starting address of the stack in memory.*
- The stack pointer initially points to the stack base; as data is pushed onto the stack, the stack pointer descends memory and continuously points to the top of stack.
- If the stack pointer passes the stack limit, then a *stack overflow occurs*

- Small piece of code that checks for stack overflow errors for a descending stack

```
; check for stack overflow
```

```
SUB sp, sp, #size
```

```
CMP sp, r10
```

```
BLLO _stack_overflow ; condition
```

Checking Stack Under Flow

; check for stack overflow

Mov r10,rn

CMP sp, r10

BLT _stack_underflow ; condition

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.

Swap Syntax

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	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

Note: Swap cannot be interrupted by any other instruction or any other bus access. The system “holds the bus” until the transaction is complete

Example-1

PRE `mem32[0x9000] = 0x12345678`

`r0 = 0x00000000`

`r1 = 0x11112222`

`r2 = 0x00009000`

`SWP r0, r1, [r2]`

POST `mem32[0x9000] = 0x11112222`

`r0 = 0x12345678`

`r1 = 0x11112222`

`r2 = 0x00009000`

Example-2

- This example shows a simple data guard that can be used to protect data from being written by another task.
- The SWP instruction “holds the bus” until the transaction is complete.

spin

```
MOV    r1, =semaphore
MOV    r2, #1
SWP    r3, r2, [r1] ; hold the bus until complete
CMP    r3, #1
BEQ    spin
```

- The address pointed to by the semaphore either contains the value 0 or 1.
- If current value of the semaphore is 1, the resource is currently used by other process.
- If current value of the semaphore is 0, the resource is currently available/ not used by any of the process.
- The process that intends to use the resource (register/ memory) will make the change the semaphore from 0 to 1
- The process that intends to release the resource (register/memory) will change the semaphore from 1 to 0

EXAMPLE-3

AREA ASCENDING , CODE, READONLY

ENTRY ;Mark first instruction to execute

START

LDR R0,=0X11111111

LDR R1,=CVALUE

LDR R2,[R1]

LDR R1,=DVALUE

STR R2,[R1]

SWP R3,R0,[R1]

BACK B BACK

CVALUE DCD 0X22222222

AREA DATA1, DATA, READWRITE

DVALUE DCD 0X22222222

END

Software Interrupt Instruction

- A software interrupt instruction - (SWI)
- This causes a software interrupt exception,
- 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 0x8 in the vector table.
- The processor mode changes to *SVC*, This 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

```
AREA HelloW, CODE, READONLY; declare area
SWIWrite EQU 0x00           ; Angel SWI number
ENTRY                       ; code entry point
START ADR r1, TEXT-1        ; r1 -> "Hello World" -1
LOOP MOV r0, #0x1           ; Angel write char in [r1]
    LDRB r2, [r1, #1]!      ; get the next k 21CS43
    CMP r2, #0              ; check for text end
    SWINE SWIWrite          ; if not end print ..
    BNE LOOP                ; .. and loop back
    MOV r0, #0x18           ; Angel exception call
    LDR r1, =0x20026         ; Exit reason
    SWI &11
TEXT = "Hello World", 0xA, 0xD, 0
END
```

Mrs. Hussana Johar R B

Asst. Professor

Dept. CSE(AI & ML)

ATMECE, Mysuru

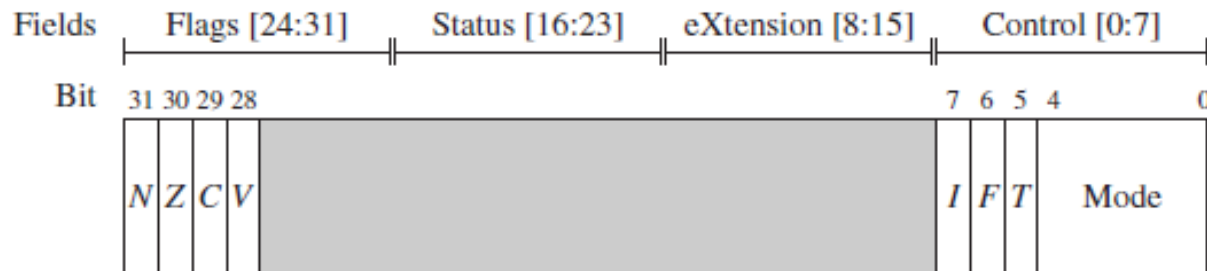
Program Status Register Instructions

- 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; 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.*

MRS, MSR Syntax

- In syntax the label called fields *can be any combination of control (c), extension (x), status (s), and flags (f)*.

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	$psr[field] = immediate$

Example

```
PRE    cpsr = nzcvqIFt_SVC
```

```
        MRS    r1, cpsr
```

```
        BIC    r1, r1, #0x80 ; 0b01000000
```

```
        MSR    cpsr_c, r1
```

```
POST   cpsr = nzcvqIFt_SVC
```

- 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.
- This code preserves all the other settings in the *cpsr* and only modifies the *I* bit in the control field.
- This example is in SVC mode. In user mode you can read all *cpsr* bits, but you can only update the condition flag field *f*.

Coprocessor Instructions

- Coprocessor instructions are used to **extend the instruction set**.
- A coprocessor provides **additional computation capability**.
- The coprocessor instructions include
 - data processing,
 - register transfer, and
 - memory transfer instructions.
- Note that these instructions are only used by **cores with a coprocessor**.

Co-processor Instruction Syntax

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

- 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.
- Coprocessor 15 (CP15) is reserved** for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

Example

- Transferring the contents of CP15 coprocessor register c0 to register r10.

; transferring the contents of CP15 register c0 to register r10

MRC p15, 0, r10, c0, c0, 0

Here CP15 *register-0* contains the processor identification number. This register is copied into the general-purpose register *r10*.

Loading Constants

- In ARM 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	<i>Rd</i> = 32-bit constant
ADR	load address pseudoinstruction	<i>Rd</i> = 32-bit relative address

INSTRUCTION SCHEDULING

The time taken to execute instructions depends on the implementation pipeline.

Rules to apply:

1. ALU operations such as addition, subtraction, and logical operations take one cycle.
2. Load instructions that load N 32-bit words of memory such as LDR and LDM take N cycles to issue, but the result of the last word loaded is not available on the following cycle.
3. Load instructions that load 16-bit or 8-bit data such as LDRB, LDRSB, LDRH, and LDRSH take one cycle to issue.

1. Branch instructions take three cycles.
2. Store instructions that store N values take N cycles.
3. Multiply instructions take a varying number of cycles depending on the value of the second operand in the product .

- The ARM9TDMI processor performs five operations in parallel:

1. *Fetch*
2. *Decode*
3. *ALU*
4. *LS1*
5. *LS2*



Example

- Example 1: This example shows the case where there is no interlock.

ADD r0, r0, r1

ADD r0, r0, r2

- Example 2: This example shows a one-cycle interlock caused by load use.

LDR r1, [r2, #4]

ADD r0, r0, r1

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	ADD	LDR	...
Cycle 3		...	ADD	—	LDR

One-cycle interlock caused by load use.

Example

This example shows a one-cycle interlock caused by delayed load use.

LDRB r1, [r2, #1]

ADD r0, r0, r2

EOR r0, r0, r1

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	EOR	ADD	LDRB
Cycle 2	...	EOR	ADD	LDRB	...
Cycle 3		...	EOR	ADD	LDRB
Cycle 4		...	EOR	—	ADD

One-cycle interlock caused by delayed load use.

Example

This example shows why a branch instruction takes three cycles. The processor must flush the pipeline when jumping to a new address.

case1

MOV r1, #1

B case1

AND r0, r0, r1

EOR r2, r2, r3

...

case1

SUB r0, r0, r1

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	AND	B	MOV	...	
Cycle 2	EOR	AND	B	MOV	...
Cycle 3	SUB	—	—	B	MOV
Cycle 4	...	SUB	—	—	B
Cycle 5		...	SUB	—	—

Pipeline flush caused by a branch.

Register Allocation

- You can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the stack pointer *r13* and the program counter *r15*. For a function to be ATPCS compliant it must preserve the callee values of registers *r4* to *r11*. ATPCS also specifies that the stack should be eight-byte aligned;

routine_name

STMFD sp!, {r4-r12, lr} ; stack saved registers

; body of routine

; the fourteen registers r0-r12 and lr are available

LDMFD sp!, {r4-r12, pc} ; restore registers and return

ALLOCATING VARIABLES TO REGISTER NUMBERS

- When you write an assembly routine, it is best to start by using names for the variables, rather than explicit register numbers. This allows you to change the allocation of variables to register numbers easily. You can even use different register names for the same physical register number when their use doesn't overlap. Register names increase the clarity and readability of optimized code.
 - *Argument registers.* The ATPCS convention defines that the first four arguments to a function are placed in registers *r0* to *r3*.
 - *Registers used in a load or store multiple.* Load and store multiple instructions LDM and STM operate on a list of registers in order of ascending register number
 - *Load and store double word.* The LDRD and STRD instructions introduced in ARMv5E operate on a pair of registers with sequential register numbers, *Rd* and *Rd + 1*.

CONDITIONAL ExECUTION

The processor core can conditionally execute most ARM instructions. This conditional execution is based on one of 15 condition codes. If you don't specify a condition, the assembler defaults to the execute always condition (AL).

The following C code converts an unsigned integer $0 \leq i \leq 15$ to a hexadecimal character c :

```
if (i<10)
{
    c=i+ '0';
}
else
{
    c=i+ 'A'-10;
}
```

We can write this in assembly using conditional execution rather than conditional branches:

```
CMP    i, #10
```

```
ADDLO c, i, #'0'
```

```
ADDHSc, i, #'A'-10
```

The sequence works since the first ADD does not change the condition codes. The second ADD is still conditional on the result of the compare.

The following C code identifies if c is a vowel:

```
if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')  
{  
vowel++;  
}
```

In assembly you can write this using conditional comparisons:

```
TEQ c, #'a'  
TEQNE c, #'e'  
TEQNE c, #'i'  
TEQNE c, #'o'  
TEQNE c, #'u'  
ADDEQ vowel, vowel, #1
```

LOOPING CONSTRUCTS

DECREMENTED COUNTED LOOPS

For a decrementing loop of N iterations, the loop counter i counts down from N to 1 inclusive. The loop terminates with $i = 0$. An efficient implementation is

```
MOV i, N
```

```
Loop
```

```
; loop body goes here and  $i=N, N-1, \dots, 1$ 
```

```
SUBS i, i, #1
```

```
BGT loop
```

LOOPING CONSTRUCTS

DECREMENTED COUNTED LOOPS

For a decrementing loop of N iterations, the loop counter i counts down from N to 1 inclusive. The loop terminates with $i = 0$. An efficient implementation is

```
MOV i, N
```

```
Loop
```

```
; loop body goes here and  $i=N, N-1, \dots, 1$ 
```

```
SUBS i, i, #1
```

```
BGE loop
```

UNROLLED COUNTED LOOPS

- This brings us to the subject of loop unrolling. Loop unrolling reduces the loop overhead by executing the loop body multiple times.

```
for(i=1000; i!=0; i--)
    a[i] = a[i] + 5;
```



```
Loop: LW    R2, 0(R1)
      ADD   R2, R2, R3
      SW    R2, 0(R1)
      ADDI  R1, R1, -4
      BNE   R1, R5, Loop
```

```
for(i=1000; i!=0; i=i-2)
    a[i] = a[i] + 5;
    a[i-1] = a[i-1] + 5;
```



```
Loop: LW    R2, 0(R1)
      ADD   R2, R2, R3
      SW    R2, 0(R1)
      LW    R2, -4(R1)
      ADD   R2, R2, R3
      SW    R2, -4(R1)
      ADDI  R1, R1, -8
      BNE   R1, R5, Loop
```

Loading Constants

- ARM instruction to move a 32-bit constant into a register. Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32-bit constant.

Syntax: LDR Rd, =constant

ADR Rd, label

LDR	load constant pseudoinstruction	<i>Rd</i> = 32-bit constant
ADR	load address pseudoinstruction	<i>Rd</i> = 32-bit relative address

Writing assembly code

- An ARM assembly language module has several parts.
- ELF sections (defined by the AREA directive).
- Application entry (defined by the ENTRY directive).
- Application execution.
- Application termination.
- Program end (defined by the END directive).

Writing assembly code

```
#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;
```

```
}
```

```
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
```

Profiling and Schedule Counting

- The first stage of any optimization process is to identify the critical routines and measure their current performance.
- A profiler is a tool that measures the proportion of time or processing cycles spent in each subroutine.
- A profiler is used to identify the most critical routines.
- A cycle counter measures the number of cycles taken by a specific routine. You can measure your success by using a cycle counter to benchmark a given subroutine before and after an optimization.

Profiling and Schedule Counting

- The ARM simulator used by the ADS1.1 debugger is called the ARMulator and provides profiling and cycle counting features.
- The ARMulator profiler works by sampling the program counter pc at regular intervals. The profiler identifies the function the pc points to and updates a hit counter for each function it encounters.
- Another approach is to use the trace output of a simulator as a source for analysis.

Profiling and Schedule Counting

- ARM implementations do not normally contain cycle-counting hardware, so to easily measure cycle counts you should use an ARM debugger with ARM simulator.
- You can configure the ARMulator to simulate a range of different ARM cores and obtain cycle count benchmarks for a number of platforms.



A T M E

College of Engineering



MODULE- III

Memory Systems

Dr. Puttegowda D

Professor & HOD

Dept. of Computer Science & Engineering
ATMECE, Mysuru

What is an Embedded System?

- An **embedded system** is an electronic/electro-mechanical system designed to perform a specific function and is a combination of both hardware and firmware (software).
- Every embedded system is unique and the hardware as well as the firmware is highly specialised to the application domain.

Embedded Systems vs. General Computing Systems

- The computing revolution began with the general purpose computing requirements. Later it was realised that the general computing requirements are not sufficient for the embedded computing requirements.
- The embedded computing requirements demand 'something special' in terms of response to stimuli, meeting the computational deadlines, power efficiency, limited memory capability, etc.

General Purpose Computing System	Embedded System
A system which is a combination of a generic hardware and a General Purpose Operating System for executing a variety of applications	A system which is a combination of special purpose hardware and embedded OS for executing a specific set of applications
Contains a General Purpose Operating System (GPOS)	May or may not contain an operating system for functioning
Applications are alterable (programmable) by the user (It is possible for the end user to re-install the operating system, and also add or remove user applications)	The firmware of the embedded system is pre-programmed and it is non-alterable by the end-user (There may be exceptions for system supporting OS kernel image flashing through special hardware settings)
Performance is the key deciding factor in the selection of the system. Always, 'Faster is Better'	Application-specific requirements (like performance, power requirements, memory usage, etc.) are the key deciding factors
Less/not at all tailored towards reduced operating power requirements, options for different levels of power management	Highly tailored to take advantage of the power saving modes supported by the hardware and the operating system
Response requirements are not time-critical	For certain category of embedded systems like mission critical systems, the response time requirement is highly critical
Need not be deterministic in execution behaviour	Execution behaviour is deterministic for certain types of embedded systems like 'Hard Real Time' systems

History of Embedded Systems

- ❑ Embedded systems were in existence even before the IT revolution.
 - ❑ Built around the old vacuum tube and transistor technologies.
- ❑ Advances in semiconductor and nanotechnology and IT revolution gave way to the development of miniature embedded systems.
- ❑ The first recognised modern embedded system is the **Apollo Guidance Computer (AGC)** developed by the MIT Instrumentation Laboratory for the lunar expedition.
 - ❑ It had 36K words of fixed memory and 2K words of erasable memory.
 - ❑ The clock frequency of was 1.024 MHz and it was derived from a 2.048 MHz crystal clock.
- ❑ The first mass-produced embedded system was the **Autonetics D-17** guidance computer for the Minuteman-I missile in 1961.
 - ❑ It was built using discrete transistor logic and a hard-disk for main memory.
- ❑ The first integrated circuit was produced in September 1958 and computers using them began to appear in 1963.

Classification of Embedded Systems

- Some of the criteria used in the classification of embedded systems are:
 1. Based on generation
 2. Complexity and performance requirements
 3. Based on deterministic behaviour
 4. Based on triggering

Classification Based on Generation

- First Generation
- Second Generation
- Third Generation
- Fourth Generation
- Next Generation

Classification Based on Generation (continued)

- **First Generation**

- Early embedded systems were built around 8-bit microprocessors like 8085 and Z80 and 4-bit microcontrollers.
- Simple in hardware circuits with firmware developed in assembly code.
- E.g.: Digital telephone keypads, stepper motor control units, etc.

Classification Based on Generation (continued)

- **Second Generation**

- Embedded systems built around 16-bit microprocessors and 8-bit or 16-bit microcontrollers.
- Instruction set were much more complex and powerful than the first generation.
- Some of the second generation embedded systems contained embedded operating systems for their operation.
- E.g.: Data acquisition systems, SCADA systems, etc.

Classification Based on Generation (continued)

• Third Generation

- Embedded systems built around 32-bit microprocessors and 16-bit microcontrollers.
- Application and domain specific processors/controllers like Digital Signal Processors (DSP) and Application Specific Integrated Circuits (ASICs) came into picture.
- The instruction set of processors became more complex and powerful and the concept of instruction pipelining also evolved.
- Dedicated embedded real time and general purpose operating systems entered into the embedded market.
- Embedded systems spread its ground to areas like robotics, media, industrial process control, networking, etc.

Classification Based on Generation (continued)

- **Fourth Generation**

- The advent of System on Chips (SoC), reconfigurable processors and multicore processors are bringing high performance, tight integration and miniaturisation into the embedded device market.
- The SoC technique implements a total system on a chip by implementing different functionalities with a processor core on an integrated circuit.
- They make use of high performance real time embedded operating systems for their functioning.
- E.g.: Smart phone devices, Mobile Internet Devices (MIDs), etc.



A T M E

College of Engineering



Classification Based on Generation (continued)

- **Next Generation**

- The processor and embedded market is highly dynamic and demanding.
- The next generation embedded systems are expected to meet growing demands in the market.

Classification Based on Complexity and Performance

- Small-Scale Embedded Systems
- Medium-Scale Embedded Systems
- Large-Scale Embedded Systems

Classification Based on Complexity and Performance (continued)

- **Small-Scale Embedded Systems**
 - Simple in application needs and the performance requirements are not time critical.
 - E.g.: An electronic toy
 - Usually built around low performance and low cost 8-bit or 16-bit microprocessors/microcontrollers.
 - May or may not contain an operating system for its functioning.

Classification Based on Complexity and Performance (continued)

- **Medium-Scale Embedded Systems**

- Slightly complex in hardware and firmware (software) requirements.
- Usually built around medium performance, low cost 16-bit or 32-bit microprocessors/microcontrollers or digital signal processors.
- Usually contain an embedded operating system (either general purpose or real time operating system) for functioning.

Classification Based on Complexity and Performance (continued)

- **Large-Scale Embedded Systems**

- Highly complex in hardware and firmware (software) requirements.
- They are employed in mission critical applications demanding high performance.
- Usually built around high performance 32-bit or 64-bit RISC processors/controllers or Reconfigurable System on Chip (RSoC) or multi-core processors and programmable logic devices.
- May contain multiple processors/controllers and co-units/hardware accelerators for offloading the processing requirements from the main processor of the system.
- Decoding/encoding of media, cryptographic function implementation, etc. are examples of processing requirements which can be implemented using a co-processor/hardware accelerator.
- Usually contain a high performance real time operating system (RTOS) for task scheduling, prioritization and management.



A T M E

College of Engineering



Classification Based on Deterministic Behaviour

- Applicable for 'Real Time' systems.
- The application/task execution behaviour can be either **deterministic** or **non-deterministic**.
- Based on the execution behaviour, real time embedded systems are classified into **Hard Real Time** and **Soft Real Time** systems.

Classification Based on Triggering

- Embedded systems which are 'Reactive' in nature (like process control systems in industrial control applications) can be classified based on the trigger.
- Reactive systems can be either **event-triggered** or **time-triggered**.

Major Application Areas of Embedded Systems

1. **Consumer electronics:** Camcorders, cameras, etc.
2. **Household appliances:** Television, DVD players, washing machine, refrigerators, microwave oven, etc.
3. **Home automation and security systems:** Air conditioners, sprinklers, intruder detection alarms, closed circuit television (CCTV) cameras, fire alarms, etc.
4. **Automotive industry:** Anti-lock braking systems (ABS), engine control, ignition systems, automatic navigation systems, etc.
5. **Telecom:** Cellular telephones, telephone switches, handset multimedia applications, etc.

Major Application Areas of Embedded Systems (continued)

6. **Computer peripherals:** Printers, scanners, fax machines, etc.
7. **Computer networking systems:** Network routers, switches, hubs, firewalls, etc.
8. **Healthcare:** Different kinds of scanners, EEG, ECG machines, etc.
9. **Measurements & Instrumentation:** Digital multimeters, digital CROs, logic analyzers, PLC systems, etc.
10. **Banking & Retail:** Automated teller machines (ATM) and currency counters, point of sales (POS), etc.
11. **Card readers:** Barcode, smart card readers, hand held devices, etc.

Purpose of Embedded Systems

- Each embedded system is designed to serve the purpose of any one or a combination of the following tasks:
 1. Data Collection/Storage/Representation
 2. Data Communication
 3. Data (Signal) Processing
 4. Monitoring
 5. Control
 6. Application Specific User Interface

Purpose of Embedded Systems (continued)

- **Data Collection/Storage/Representation**

- Embedded systems designed for the purpose of data collection performs acquisition of data from the external world.
- Data collection is usually done for storage, analysis, manipulation and transmission.
- The term "data" refers all kinds of information, viz. text, voice, image, video, electrical signals and any other measurable quantities.
- Data can be either analog (continuous) or digital (discrete).
- The collected data may be stored or transmitted or it may be processed or it may be deleted instantly after giving a meaningful representation.



- A **digital camera** is a typical example of an embedded system with data collection/storage/representation of data.
- Images are captured and the captured image may be stored within the memory of the camera.
- The captured image can also be presented to the user through a graphic LCD unit.

Purpose of Embedded Systems (continued)

- **Data Communication**

- Embedded data communication systems are deployed in applications ranging from complex satellite communication systems to simple home networking systems.
- The transmission is achieved either by a wire-line medium or by a wireless medium.
- The data collecting embedded terminal itself can incorporate data communication units like wireless modules (Bluetooth, ZigBee, Wi-Fi, EDGE, GPRS, etc.) or wire-line modules (RS-232C, USB, TCP/IP, PS2, etc.).



Fig: A **wireless network router** for data communication

- Network hubs, routers, switches, etc. are typical examples of dedicated data transmission embedded systems.
- They act as mediators in data communication and provide various features like data security, monitoring etc.

Purpose of Embedded Systems (continued)

- **Data (Signal) Processing**

- The data (voice, image, video, electrical signals and other measurable quantities) collected by embedded systems may be used for various kinds of data processing.
- Embedded systems with signal processing functionalities are employed in applications demanding signal processing like speech coding, synthesis, audio video codec, transmission applications, etc.



- A **digital hearing aid** is a typical example of an embedded system employing data processing.
- Digital hearing aid improves the hearing capacity of hearing impaired persons.

Purpose of Embedded Systems (continued)

- **Monitoring**

- Almost embedded products coming under the medical domain are used for monitoring.

- A very good example is the electro cardiogram (ECG) machine for monitoring the heartbeat of a patient.
- The machine is intended to do the monitoring of the heartbeat.
- It cannot impose control over the heartbeat.
- The sensors used in ECG are the different electrodes connected to the patient's body.
- Some other examples of embedded systems with monitoring function are measuring instruments like digital CRO, digital multimeters, logic analyzers, etc. used in Control & Instrumentation applications.



Fig: A patient monitoring system for monitoring heartbeat

Purpose of Embedded Systems (continued)

- **Control**

- Embedded systems with control functionalities impose control over some variables according to the changes in input variables.
- A system with control functionality contains both sensors and actuators.
- Sensors are connected to the input port for capturing the changes in environmental variable or measuring variable.
- The actuators connected to the output port are controlled according to the changes in input variable to put an impact on the controlling variable to bring the controlled variable to the specified range.



- An **Air Conditioner System** used to control the room temperature to a specified limit is a typical example for embedded system for control purpose.
- An air conditioner contains a room temperature-sensing element (sensor) which may be a thermistor and a handheld unit for setting up (feeding) the desired temperature.

Purpose of Embedded Systems (continued)

- **Application Specific User Interface**
 - These are embedded systems with application-specific user interfaces like buttons, switches, keypad, lights, bells, display units, etc.
 - Mobile phone is an example for this.
 - In mobile phone the user interface is provided through the keypad, graphic LCD module, system speaker, vibration alert, etc.



- A **mobile phone** is an example for embedded system with an application-specific user interfaces.

A Typical Embedded System

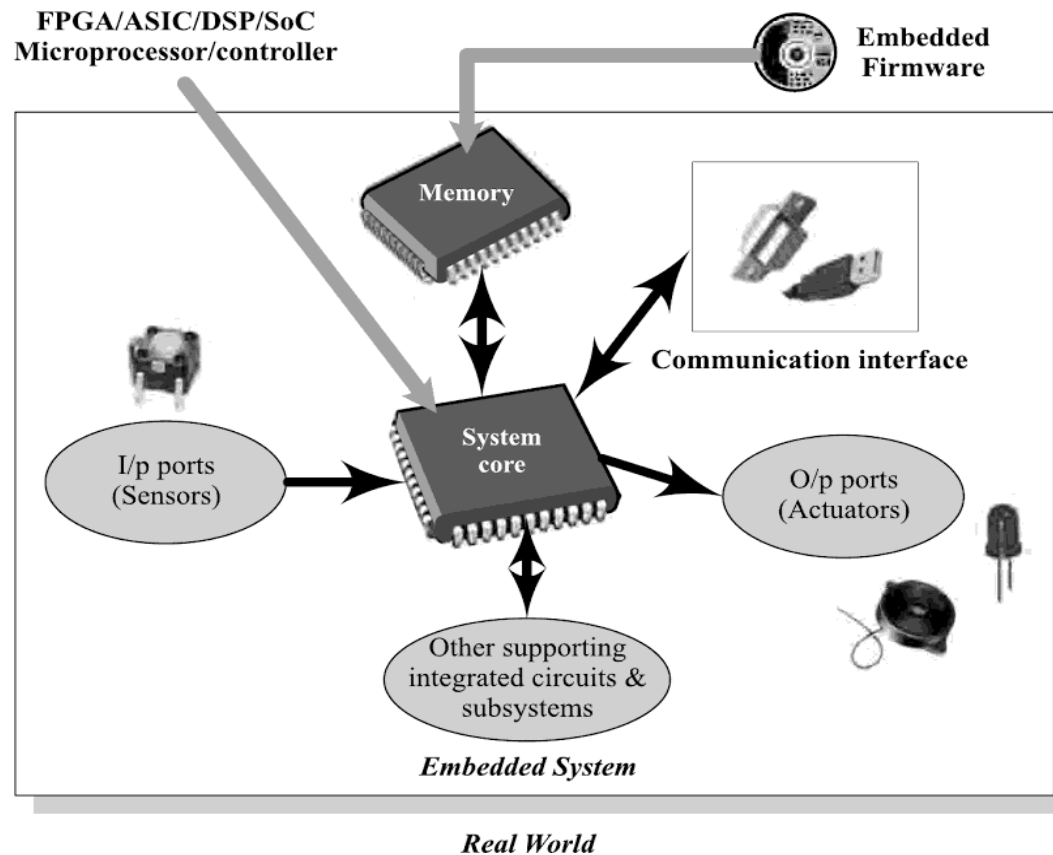


Fig: Elements of an Embedded System

Core of the Embedded System

- Embedded systems are domain and application specific and are built around a central core.
- The core of the embedded system falls into any one of the following categories:
 1. General Purpose and Domain Specific Processors
 1. Microprocessors
 2. Microcontrollers
 3. Digital Signal Processors
 2. Application Specific Integrated Circuits (ASICs)
 3. Programmable Logic Devices (PLDs)
 4. Commercial off-the-shelf Components (COTS)



A T M E

College of Engineering



General Purpose and Domain Specific Processors

- Almost 80% of the embedded systems are processor/controller based.
- The processor may be a microprocessor or a microcontroller or a digital signal processor, depending on the domain and application.
- Most of the embedded systems in the industrial control and monitoring applications make use of the commonly available microprocessors or microcontrollers.
- Domains which require signal processing such as speech coding, speech recognition, etc. make use of special kind of digital signal processors.

Microprocessors

- A **Microprocessor** is a silicon chip representing a central processing unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of instructions.
- In general the CPU contains the Arithmetic and Logic Unit (ALU), control unit and working registers.
- A microprocessor is a dependent unit and it requires the combination of other hardware like memory, timer unit, and interrupt controller, etc. for proper functioning.
- Intel, AMD, Freescale, IBM, TI, Cyrix, Hitachi, NEC, LSI Logic, etc. are the key players in the processor market.

General Purpose Processor (GPP) vs. Application- Specific Instruction Set Processor (ASIP)

General Purpose Processor (GPP)

- A General Purpose Processor or GPP is a processor designed for general computational tasks.
 - The processor running inside laptop or desktop is a typical example for general purpose processor.
- Due to the high volume production, the per unit cost for a chip is low.
- A typical general purpose processor contains an Arithmetic and Logic Unit (ALU) and Control Unit (CU).

Application-Specific Instruction Set Processor (ASIP)

- Application Specific Instruction Set Processors (ASIPs) are processors with architecture and instruction set optimised to specific-domain/application requirements like network processing, automotive, telecom, media applications, digital signal processing, control applications, etc.
- Most of the embedded systems are built around application specific instruction set processors.
 - Some microcontrollers (like automotive AVR, USB AVR from Atmel), system on chips, digital signal processors, etc. are examples for application specific instruction set processors (ASIPs).
- ASIPs incorporate a processor and on-chip peripherals, demanded by the application requirement, program and data memory.

Microcontrollers

- A **Microcontroller** is a highly integrated chip that contains a CPU, scratch pad RAM, special and general purpose register arrays, on chip ROM/FLASH memory for program storage, timer and interrupt control units and dedicated I/O ports.
- A microcontroller contains all the necessary functional blocks for independent working.
 - Have greater place in embedded domain in place of microprocessors.
 - They are cheap, cost effective and are readily available in the market.
- Atmel, Texas Instruments, Toshiba, Philips, Freescale, NEC, Zilog, Hitachi, Mitsubishi, Infineon, ST Micro Electronics, National, Microchip, Analog Devices, Daewoo, Intel, Maxim, Sharp, Silicon Laboratories, TDK, Triscend, Winbond, etc. are the key players in the microcontroller market.

Microprocessor vs. Microcontroller

Microprocessor	Microcontroller
A silicon chip representing a central processing unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of instructions	A microcontroller is a highly integrated chip that contains a CPU, scratchpad RAM, special and general purpose register arrays, on chip ROM/FLASH memory for program storage, timer and interrupt control units and dedicated I/O ports
It is a dependent unit. It requires the combination of other chips like timers, program and data memory chips, interrupt controllers, etc. for functioning	It is a self-contained unit and it doesn't require external interrupt controller, timer, UART, etc. for its functioning
Most of the time, general purpose in design and operation	Mostly application-oriented or domain-specific
Doesn't contain a built in I/O port. The I/O port functionality needs to be implemented with the help of external programmable peripheral interface chips like 8255	Most of the processors contain multiple built-in I/O ports which can be operated as a single 8 or 16 or 32 bit port or as individual port pins
Targeted for high end market where performance is important	Targeted for embedded market where performance is not so critical (At present this demarcation is invalid)
Limited power saving options compared to microcontrollers	Includes lot of power saving features

Digital Signal Processors

- **Digital Signal Processors (DSPs)** are powerful special purpose 8/16/32 bit microprocessors designed specifically to meet the computational demands and power constraints of today's embedded audio, video, and communications applications.
- Digital signal processors are 2 to 3 times faster than the general purpose microprocessors in signal processing applications.
 - This is because of the architectural difference between the two.
 - DSPs implement algorithms in hardware which speeds up the execution whereas general purpose processors implement the algorithm in firmware and the speed of execution depends primarily on the clock for the processors.
- Audio video signal processing, telecommunication and multimedia applications are typical examples where DSP is employed.
- Digital signal processing employs a large amount of real-time calculations.
- Sum of products (SOP) calculation, convolution, fast fourier transform (FFT), discrete fourier transform (DFT), etc, are some of the operations performed by digital signal processors.

Digital Signal Processors (continued)

- A typical digital signal processor incorporates the following key units:
- **Program Memory:** Memory for storing the program required by DSP to process the data
- **Data Memory:** Working memory for storing temporary variables and data/signal to be processed.
- **Computational Engine:** Performs the signal processing in accordance with the stored program memory.
 - It incorporates many specialised arithmetic units and each of them operates simultaneously to increase the execution speed.
 - It also incorporates multiple hardware shifters for shifting operands and thereby saves execution time.
- **I/O Unit:** Acts as an interface between the outside world and DSP.
 - It is responsible for capturing signals to be processed and delivering the processed signals.

RISC vs. CISC

Processors/Controllers

- RISC stands for **Reduced Instruction Set Computing**.
 - All RISC processors/controllers possess lesser number of instructions, typically in the range of 30 to 40.
 - E.g.: Atmel AVR microcontroller – its instruction set contains only 32 instructions.
- CISC stands for **Complex Instruction Set Computing**.
 - The instruction set is complex and instructions are high in number.
 - E.g.: 8051 microcontroller – its instruction set contains 255 instructions.

RISC	CISC
Lesser number of instructions	Greater number of instructions
Instruction pipelining and increased execution speed	Generally no instruction pipelining feature
Orthogonal instruction set (Allows each instruction to operate on any register and use any addressing mode)	Non-orthogonal instruction set (All instructions are not allowed to operate on any register and use any addressing mode. It is instruction-specific)
Operations are performed on registers only, the only memory operations are load and store	Operations are performed on registers or memory depending on the instruction
A large number of registers are available	Limited number of general purpose registers
Programmer needs to write more code to execute a task since the instructions are simpler ones	Instructions are like macros in C language. A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC
Single, fixed length instructions	Variable length instructions
Less silicon usage and pin count	More silicon usage since more additional decoder logic is required to implement the complex instruction decoding.
With Harvard Architecture	Can be Harvard or Von-Neumann Architecture

Harvard vs. Von-Neumann Processor/Controller Architecture

- Von-Neumann Architecture

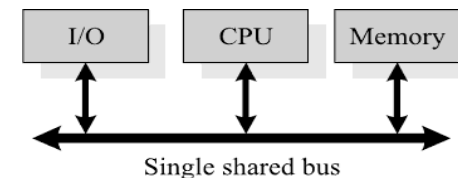
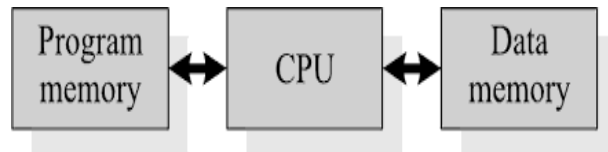
- Microprocessors/controllers based on the Von-Neumann architecture share a **single common bus** for fetching both instructions and data.
- Program instructions and data are stored in a common main memory.
- They first fetch an instruction and then fetch the data to support the instruction from code memory.
 - The two separate fetches slows down the controller's operation.
- Von-Neumann architecture is also referred as **Princeton architecture**, since it was developed by the Princeton University.

Harvard vs. Von-Neumann Processor/Controller Architecture (continued)

- **Harvard Architecture**
 - Microprocessors/controllers based on the Harvard architecture will have **separate data bus and instruction bus**.
 - This allows the data transfer and program fetching to occur simultaneously on both buses.
 - The data memory can be read and written while the program memory is being accessed.
 - These separated data memory and code memory buses allow one instruction to execute while the next instruction is fetched ("pre- fetching").
 - The pre-fetch theoretically allows much faster execution than Von-Neumann architecture.

Harvard vs. Von-Neumann Processor/Controller Architecture (continued)

Harvard Architecture	Von-Neumann Architecture
Separate buses for instruction and data fetching	Single shared bus for instruction and data fetching
Easier to pipeline, so high performance can be achieved	Low performance compared to Harvard architecture
Comparatively high cost	Cheaper
No memory alignment problems	Allows self modifying codes
Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory	Since data memory and program memory are stored physically in the same chip, chances for accidental corruption of program memory



Big-Endian vs. Little-Endian Processors/Controllers

- Endianness specifies the order in which the data is stored in the memory by processor operations in a multi byte system.
- Suppose the word length is two byte then data can be stored in memory in two different ways:
 1. Higher order of data byte at the higher memory and lower order of data byte at location just below the higher memory – **Little-Endian**
 - E.g.: Intel x86 Processors
 2. Lower order of data byte at the higher memory and higher order of data byte at location just below the higher memory – **Big-Endian**
 - E.g.: Motorola 68000 Series Processors

Big-Endian vs. Little-Endian Processors/Controllers (continued)

- **Little-endian** means the lower-order byte of the data is stored in memory at the lowest address, and the higher-order byte at the highest address. (The little end comes first.)
- For example, a 4 byte long integer **Byte3 Byte2 Byte1 Byte0** will be stored in the memory as shown below:

Base Address + 0	Byte 0	Byte 0	0x20000 (Base Address)
Base Address + 1	Byte 1	Byte 1	0x20001 (Base Address + 1)
Base Address + 2	Byte 2	Byte 2	0x20002 (Base Address + 2)
Base Address + 3	Byte 3	Byte 3	0x20003 (Base Address + 3)

Big-Endian vs. Little-Endian Processors/Controllers (continued)

- **Big-endian** means the higher-order byte of the data is stored in memory at the lowest address, and the lower-order byte at the highest address. (The big end comes first.)
- For example, a 4 byte long integer **Byte3 Byte2 Byte1 Byte0** will be stored in the memory as shown below:

Base Address + 0	Byte 3	Byte 3	0x20000 (Base Address)
Base Address + 1	Byte 2	Byte 2	0x20001 (Base Address + 1)
Base Address + 2	Byte 1	Byte 1	0x20002 (Base Address + 2)
Base Address + 3	Byte 0	Byte 0	0x20003 (Base Address + 3)

Load Store Operation and Instruction Pipelining

- The memory access related operations are performed by the special instructions **load** and **store**.
 - If the operand is specified as memory location, the content of it is loaded to a register using the **load** instruction.
 - The instruction **store** stores data from a specified register to a specified memory location.
- The concept of Load Store Architecture is illustrated with the following example:
 - Suppose x , y and z are memory locations and we want to add the contents of x and y and store the result in location z . Under the load store architecture the same is achieved with 4 instructions as shown:

Load Store Operation and Instruction Pipelining (continued)

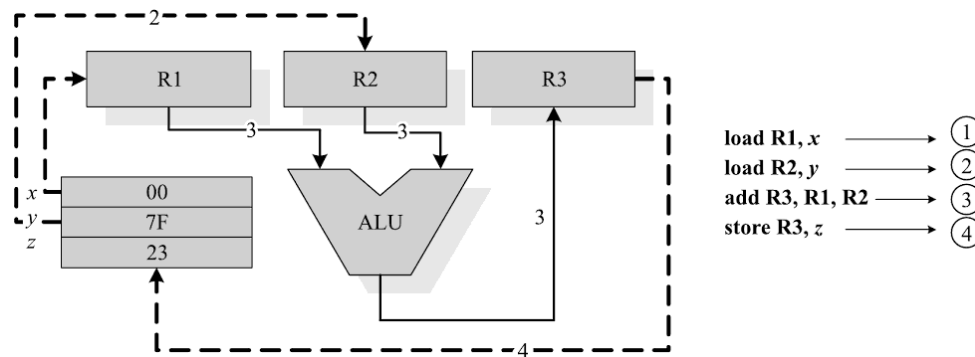


Fig: The concept of load store architecture

- The first instruction *load R1, x* loads the register R1 with the content of memory location x.
- The second instruction *load R2, y* loads the register R2 with the content of memory location y.
- The instruction *add R3, R1, R2* adds the content of registers R1 and R2 and stores the result in register R3.
- The next instruction *store R3, z* stores the content of register R3 in memory location z.

Load Store Operation and Instruction Pipelining (continued)

- The conventional instruction execution by the processor follows the fetch-decode-execute sequence.
- The **fetch** part fetches the instruction from program memory or code memory.
- The **decode** part decodes the instruction to generate the necessary control signals.
- The **execute** stage reads the operands, perform ALU operations and stores the result.
- In conventional program execution, the fetch and decode operations are performed in sequence. For simplicity let's consider decode and execution together.

Load Store Operation and Instruction Pipelining (continued)

- During the decode operation, the memory address bus is available and if it is possible to effectively utilise it for an instruction fetch, the processing speed can be increased.
- **Instruction pipelining** refers to the overlapped execution of instructions – i.e., while the current instruction is being decoded and executed, the next instruction will be fetched.
- If the current instruction in progress is a program control flow transfer instruction like jump or call instruction, the instruction fetched is flushed and a new instruction fetch is performed to fetch the instruction.
- Whenever the current instruction is executing the program counter will be loaded with the address of the next instruction.
- In case of jump or branch instruction, the new location is known only after completion of the jump or branch instruction.

Load Store Operation and Instruction Pipelining (continued)

- Depending on the stages involved in an instruction (fetch, read register and decode, execute instruction, access an operand in data memory, write back the result to register, etc.), there can be multiple levels of instruction pipelining.
- Figure illustrates the concept of instruction pipelining for single stage pipelining.

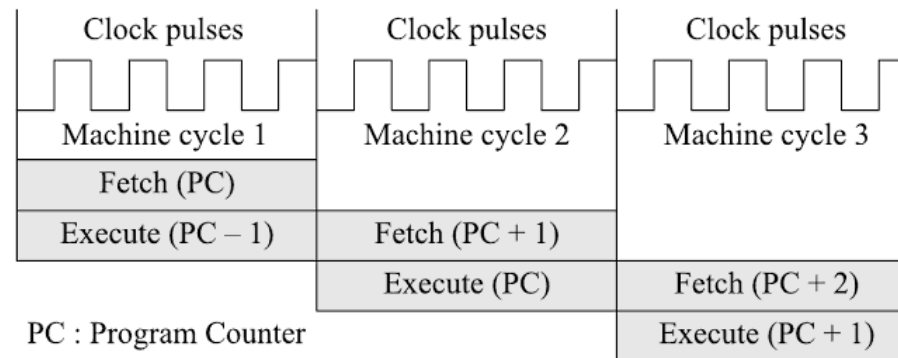


Fig: The single-stage pipelining concept

Application Specific Integrated Circuits (ASICs)

- **Application Specific Integrated Circuit (ASIC)** is a microchip designed to perform a specific or unique application.
 - Used as replacement to conventional general purpose logic chips.
- It integrates several functions into a single chip and thereby reduces the system development cost.
- ASIC consumes a very small area in the total system.
 - Helps in the design of smaller systems with high capabilities/functionalities.
- Fabrication of ASICs requires a non-refundable initial investment for the process technology and configuration expenses. This investment is known as **Non-Recurring Engineering Charge (NRE)** and it is a one time investment.
- If the Non-Recurring Engineering Charges (NRE) is borne by a third party and the ASIC is made openly available in the market, then it is referred as **Application Specific Standard Product (ASSP)**.
 - E.g.: ADE7760 Energy Meter ASIC developed by Analog Devices for Energy metering applications

Programmable Logic Devices

- **Logic devices** provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform.
- Logic devices can be classified into two broad categories—**fixed** and **programmable**.
- The circuits in a fixed logic device are permanent, they perform one function or set of functions—once manufactured, they cannot be changed.
- **Programmable Logic Devices (PLDs)** offer customers a wide range of logic capacity, features, speed, and voltage characteristics and these devices can be re-configured to perform any number of functions at any time.
 - Designers use inexpensive software tools to quickly develop, simulate, and test their designs.
 - Then, a design can be quickly programmed into a device, and immediately tested in a live circuit.

Programmable Logic Devices (continued)

- There are no NRE costs and the final design is completed much faster than that of a custom, fixed logic device.
- Another key benefit of using PLDs is that during the design phase customers can change the circuitry as often as they want until the design operates to their satisfaction.
 - PLDs are based on re-writable memory technology to change the design, the device is simply reprogrammed.
- Once the design is final, customers can go into immediate production by simply programming as many PLDs as they need with the final software design file.
- The two major types of programmable logic devices are **Field Programmable Gate Arrays (FPGAs)** and **Complex Programmable Logic Devices (CPLDs)**.

CPLDs and FPGAs

CPLDs

- CPLDs offer much smaller amounts of logic—up to about 10,000 gates.
- But, CPLDs offer very predictable timing characteristics and are therefore ideal for critical control applications.
- CPLDs such as the Xilinx CoolRunner series also require extremely low amounts of power and are very inexpensive, making them ideal for cost-sensitive, battery-operated, portable applications such as mobile phones and digital handheld assistants.

• FPGAs

- The FPGAs offer the highest amount of logic density,
 - the most features, and the highest performance.
- The largest FPGA now shipping, part of the Xilinx Virtex line of devices, provides eight million "system gates" (the relative density of logic).
- These advanced devices also offer features such as built-in hardwired processors (such as the IBM power PC), substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies.
- FPGAs are used in a wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing.

Advantages of PLD

- PLDs offer customers much more flexibility during the design cycle because design iterations are simply a matter of changing the programming file, and the results of design changes can be seen immediately in working parts.
- PLDs do not require long lead times for prototypes or production parts—the PLDs are already on a distributor's shelf and ready for shipment.
- PLDs do not require customers to pay for large NRE costs and purchase expensive mask sets—PLD suppliers incur those costs when they design their programmable devices and are able to amortize those costs over the multi-year lifespan of a given line of PLDs.
- PLDs allow customers to order just the number of parts they need, when they need them, allowing them to control inventory.
- PLDs can be reprogrammed even after a piece of equipment is shipped to a customer. The manufacturers can add new features or upgrade products that already are in the field. To do this, they simply upload a new programming file to the PLD, via the Internet, creating new hardware logic in the system.

Commercial Off-the-Shelf Components (COTS)

- A **Commercial Off-the-Shelf (COTS)** product is one which is used 'as-is'.
- COTS products are designed in such a way to provide easy integration and interoperability with existing system components.
- The COTS component itself may be developed around a general purpose or domain specific processor or an Application Specific Integrated Circuit or a Programmable Logic Device.
- **Advantages:**
 - They are readily available in the market
 - Cheap
 - Developer can cut down his development time to a great extent
 - Reduces the time to market

Commercial Off-the-Shelf Components (COTS) (continued)

- Typical examples of COTS hardware unit are **remote controlled toy car control units** including the RF circuitry part, high performance, high frequency microwave electronics (2—200 GHz), high bandwidth analog-to-digital converters, devices and components for operation at very high temperatures, electro-optic IR imaging arrays, UV/IR detectors, etc.
- E.g.: The TCP/IP plug-in module available from various manufactures like 'WIZnet', 'Freescale', 'Dynalog', etc. are very good examples of COTS product.



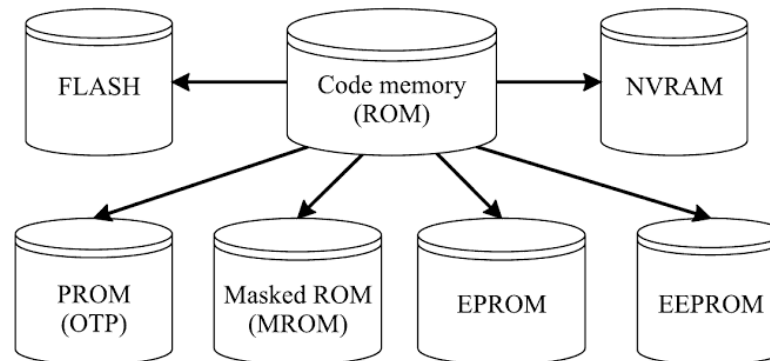
Fig: An example of a COTS product for TCP/IP plug-in from WIZnet (WIZnet NM7010A Plug in Module)

Memory

- **Memory** is an important part of a processor/controller based embedded systems.
- Some of the processors/controllers contain built in memory and this memory is referred as **on-chip memory**.
- Others do not contain any memory inside the chip and requires external memory to be connected with the controller/processor to store the control algorithm. It is called **off-chip memory**.
- Also some working memory is required for holding data temporarily during certain operations.

Program Storage Memory (ROM)

- The program memory or code storage memory of an embedded system stores the program instructions.
- The code memory retains its contents even after the power is turned off. It is generally known as **non-volatile** storage memory.
- It can be classified into different types as shown:



Masked ROM (MROM)

- Masked ROM is a one-time programmable device.
- Masked ROM makes use of the hardwired technology for storing data.
- The device is factory programmed by masking and metallisation process at the time of production itself, according to the data provided by the end user.
- Advantage – low cost for high volume production.
- Limitation - inability to modify the device firmware against firmware upgrades.
 - Since the MROM is permanent in bit storage, it is not possible to alter the bit information.

Masked ROM (MROM) (continued)

- Different mechanisms are used for the masking process of the ROM, like
 1. Creation of an enhancement or depletion mode transistor through channel implant.
 2. By creating the memory cell either using a standard transistor or a high threshold transistor.
 - In the high threshold mode, the supply voltage required to turn ON the transistor is above the normal ROM IC operating voltage.
 - This ensures that the transistor is always off and the memory cell stores always logic 0.

Programmable Read Only Memory (PROM) / (OTP)

- One Time Programmable Memory (OTP) or PROM is not pre-programmed by the manufacturer.
 - The end user is responsible for programming these devices.
- This memory has nichrome or polysilicon wires arranged in a matrix. These wires can be functionally viewed as fuses.
 - It is programmed by a PROM programmer which selectively burns the fuses according to the bit pattern to be stored.
 - Fuses which are not blown/burned represents a logic "1" whereas fuses which are blown/burned represents a logic 0 .
 - The default state is logic "1".
- OTP is widely used for commercial production of embedded systems whose prototyped versions are proven and the code is finalised.
 - It is a low cost solution for commercial production.
- OTPs cannot be reprogrammed.

Erasable Programmable Read Only Memory (EPROM)

- Erasable Programmable Read Only Memory (EPROM) gives the flexibility to re-program the same chip.
- EPROM stores the bit information by charging the floating gate of an FET.
- Bit information is stored by using an EPROM programmer, which applies high voltage to charge the floating gate.
- EPROM contains a quartz crystal window for erasing the stored information.
 - If the window is exposed to ultraviolet rays for a fixed duration, the entire memory will be erased.
- Even though the EPROM chip is flexible in terms of re-programmability, it needs to be taken out of the circuit board and put in a UV eraser device for 20 to 30 minutes.
 - It is a tedious and time-consuming process.

Electrically Erasable Programmable Read only Memory (EEPROM)

- The information contained in the EEPROM memory can be altered by using electrical signals at the register/byte level.
- They can be erased and reprogrammed in-circuit.
- These chips include a chip erase mode and in this mode they can be erased in a few milliseconds.
- It provides greater flexibility for system design.
- The only limitation is their capacity is limited (a few kilobytes) when compared with the standard ROM.

FLASH

- FLASH memory is a variation of EEPROM technology – It combines the re- programmability of EEPROM and the high capacity of standard ROMs.
- FLASH is the latest ROM technology.
 - Most popular ROM technology used in today's embedded designs.
- FLASH memory is organised as sectors (blocks) or pages.
- FLASH memory stores information in an array of floating gate MOSFET transistors.
- The erasing of memory can be done at sector level or page level without affecting the other sectors or pages.
- Each sector/page should be erased before re-programming.
- The typical erasable capacity of FLASH is 1000 cycles.
- E.g.: W27C512 from WINBOND is an example of 64KB FLASH memory.

Non-Volatile RAM (NVRAM)

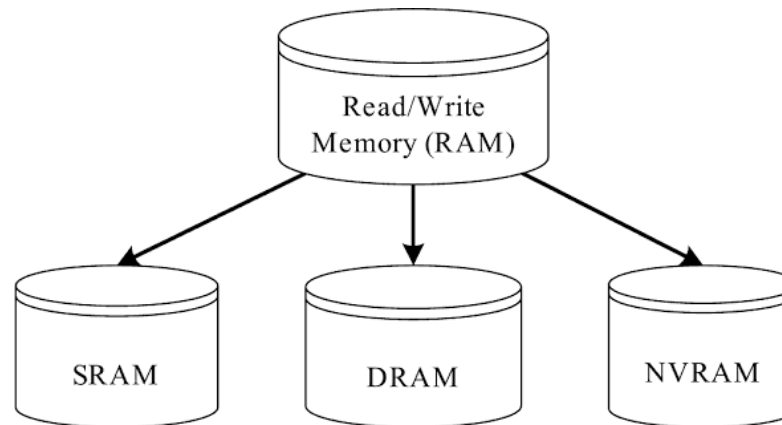
- Non-volatile RAM is a random access memory with battery backup.
- It contains static RAM based memory and a minute battery for providing supply to the memory in the absence of external power supply.
- The memory and battery are packed together in a single package.
- The life span of NVRAM is expected to be around 10 years.
- E.g.: DS1644 from Maxim/Dallas is an example of 32KB NVRAM.

Read-Write Memory/Random Access Memory (RAM)

- RAM is the data memory or working memory of the controller/processor.
- Controller/processor can read from it and write to it.
- RAM is volatile – when the power is turned off, all the contents are destroyed.
- RAM is a direct access memory – we can access the desired memory location directly without the need for traversing through the entire memory locations to reach the desired memory position (i.e. random access of memory location).
 - This is in contrast to the Sequential Access Memory (SAM), where the desired memory location is accessed by either traversing through the entire memory or through a 'seek' method. Magnetic tapes, CD ROMs, etc. are examples of sequential access memories.

Read-Write Memory/Random Access Memory (RAM) (continued)

- RAM generally falls into three categories: Static RAM (SRAM), Dynamic RAM (DRAM) and Non-Volatile RAM (NVRAM).



Static RAM (SRAM)

- Static RAM stores data in the form of voltage.
- They are made up of flip-flops.
- Static RAM is the fastest form of RAM available.
 - Fast due to its resistive networking and switching capabilities.
- In typical implementation, an SRAM cell (bit) is realised using six transistors (or 6 MOSFETs).
 - Four of the transistors are used for building the latch (flip-flop) part of the memory cell and two for controlling the access.

Static RAM (SRAM) (continued)

- In its simplest representation an SRAM cell can be visualised as shown in the figure below:

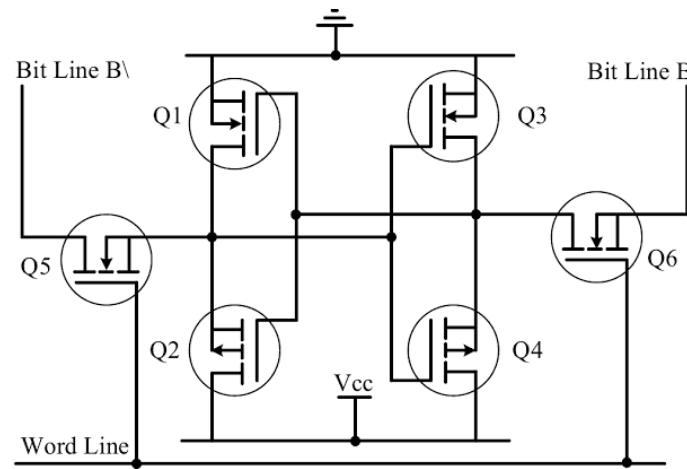


Fig: SRAM cell implementation

Static RAM (SRAM) (continued)

- This implementation in its simpler form can be visualised as two cross-coupled inverters with read/write control through transistors.
 - The four transistors in the middle form the cross-coupled inverters.
- This can be visualised as shown in the figure below:

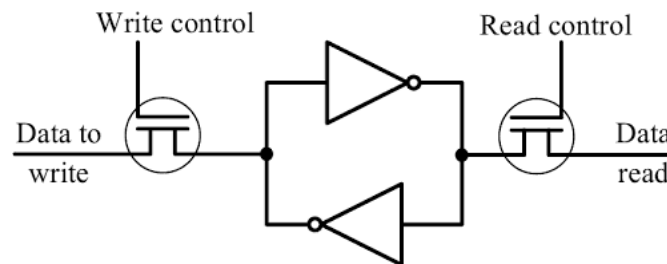


Fig: Visualisation of SRAM cell