Module 2

Introduction to the ARM Instruction Set

Contents **Page No.**

Module 2 - Syllabus

Introduction to the ARM Instruction Set: Data Processing Instructions , Programme Instructions, Software Interrupt Instructions, Program Status Register Instructions, Coprocessor Instructions, Loading Constants

ARM programming using Assembly language: Writing Assembly code, Profiling and cycle counting, instruction scheduling, Register Allocation, Conditional Execution, Looping Constructs

Text book 1: Chapter 3:Sections 3.1 to 3.6 (Excluding 3.5.2), Chapter 6(Sections 6.1 to 6.6)

RBT: L1, L2

Different ARM architecture revisions support different instructions. However, new revisions usually addinstructions and remain backwardly compatible. Code you write for architecture ARMv4T should execute on an ARMv5TE processor.

The Table 3.1provides a complete list of ARM instructions available in the *ARMv5E instruction set architecture* (*ISA*). This ISA includes all the core ARM instructions as well as some of the newer features in the ARM instruction set. The "ARM ISA" column lists the ISA revision in which the instruction was introduced. Some instructions have extended functionality in later architectures.

Processor operations using example with pre and post-conditions describing registers and memory before and after the instruction or instructions are executed. In the following sections, the hexadecimal numbers are represented with the prefix *0x* and binary numbers with the prefix *0b*. The examples follow this format:

```
PRE    <pre-conditions>

<instruction/s>

POST   <post-conditions>
```

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

mem<data_size>[address]

This refers to *data_size*bits of memory starting at the given byte address. For example, *mem32[1024]* is the 32-bit value starting at address 1 KB.

Table 3.1 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 (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) 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
SMLA*xy*	v5E	signed multiply accumulate instructions ((16 × 16) + 32 = 32-bit)
SMLAL	v3M	signed multiply accumulate long ((32 × 32) + 64 = 64-bit)
SMLAL*xy*	v5E	signed multiply accumulate long ((16 × 16) + 64 = 64-bit)
SMLAW*y*	v5E	signed multiply accumulate instruction (((32 × 16) ≫ 16) + 32 = 32-bit)
SMULL	v3M	signed multiply long (32 × 32 = 64-bit)

continued

Table 3.1 ARM instruction set. (*Continued*)

Mnemonics	ARM ISA	Description
SMULxy	v5E	signed multiply instructions (16 × 16 = 32-bit)
SMULWy	v5E	signed multiply instruction ((32 × 16) ≫ 16 = 32-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 × 32) + 64 = 64-bit)
UMULL	v3M	unsigned multiply long (32 × 32 = 64-bit)

ARM instructions process data held in registers and memory is accessed only with load and store instructions.

- For instance, the ADD instruction below adds the two values stored in registers *r1* and *r2* (the source registers). It writes the result to register *r3* (the destination register).

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

- ARM instructions are mainly classified into four groups. They are:
  1. Data Processing Instructions
  2. Branch Instructions
  3. Load-Store Instructions
  4. Software Interrupt Instruction
  5. Program Status Register Instructions

2.1 Data Processing Instructions

- The data processing instructions manipulate data within registers. They are:
  - Move Instructions
  - Arithmetic Instructions
  - Logical Instructions
  - Comparison Instructions

- Multiply Instructions

- The above listed data processing instructions can process one of their operands using the barrel shifter.
- Suppose if you use the suffix on a data processing instruction, then it updates the flags in the *cpsr or* they make use of cprs flag values to perform an operation.

  Example: *MOVS R1,R2 ;updates the cpsr flags i.e c if the MSB bit of R1 is binary 1*

- Move and logical operations update the **carry flag *C*, negative flag *N*, and zero flag *Z*.**
  - The *C* 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.

2.1.1 MOVE Instructions

- Move instruction 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	$Rd = \sim N$

Example 3.1: This example shows a simple move instruction. The **MOV** instruction takes the contents of **register *r5***and copies them into **register *r7***, in this case, taking the *value 5, and overwriting the value 8 in register r7.*

PRE R5=5

R7=8

MOV R7,R5 ;LET R7=R5

POST R5=5

R7=5

2.1.2 Barrel Shifter

- In above Example, we showed a MOV instruction where *N* is a simple register. But *N* can be more than just a register or immediate value; it can also be a register *Rm* that has been preprocessed by the barrel shifter prior to being used by a data processing instruction.
- Data processing instructions are processed within the arithmetic logic unit (ALU).

- A unique and powerful feature of the ARM processor is the ability to ***shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.***
- Pre-processing or shift occurs within the cycle time of the instruction.
- This shift increases the power and flexibility of many data processing operations.This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.
- ***There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.***

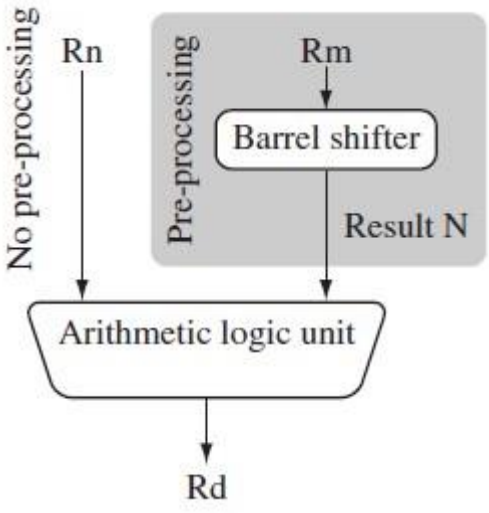


Figure: Barrel Shifter and ALU

- Figure shows the data flow between the ALU and the barrel shifter.Register *Rn* enters the ALU without any pre- processing of registers.
- When the example 3.2 is submitted to the ALU processing will be done as follows:
    - We apply a logical shift left (LSL) to register *R5*before moving it to the destination register. This is the same as applying the standard *C* language shift operator « to the register.
    - The MOV instruction copies the shift operator result *N* into register *Rd*. *N* represents the result of the LSL operation described in the Table 3.2.

Table 3.2 Barrel shifter operations.

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

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

Example 3.2: The example multiplies register R5 with 4 and then places the result into register R7.

PRE R5=5

R8=8

MOV R7,R5,LSL #2 ; let R7 = R5*4= (R5<<2)

POST R5=5

R7=20

- The following Figure illustrates a logical shift left by one.

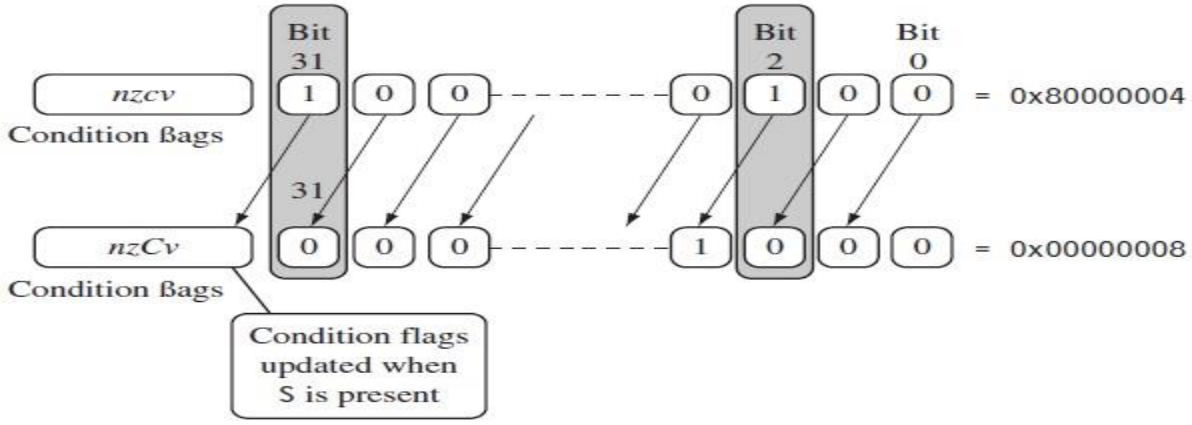


Figure: Logical Shift Left by One

- For example, the contents of bit *0* are shifted to bit *1*. Bit 0 is cleared. The C flag is updated with the last bit shifted out of the register.
- This is bit (32 - y) of the original value, where *y* is the shift amount. When *y*is greater than one, then a shift by *y* positions is the same as a shift by one position executed *y* times.

Example3.3: This example of a MOVS instruction shifts register *r1* left by one bit. This multiplies register *r1* by a value *21*. As you can see, the C flag is updated in the *cpsr*because the S suffix is present in the instruction mnemonic.

PRE *cpsr = nzcvqiFt_USER*

r0 = 0x00000000

r1 = 0x80000004

MOVS r0, r1, LSL #1

POST *cpsr = nzCvqiFt_USER*

r0 = 0x00000008

r1 = 0x80000004

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

Table 3.3 Barrel shift operation syntax for data processing instructions.

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

2.1.3 Arithmetic Instructions

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

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

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

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

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

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

       SUB r0, r1, r2

POST   r0 = 0x00000001
```

EXAMPLE 3.5 This reverse subtract instruction (RSB) subtracts *r1* from the constant value #0, writing the result to *r0*. You can use this instruction to negate numbers.

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

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

POST   r0 = -r1 = 0xffffff89
```

EXAMPLE 3.6 The SUBS instruction is useful for decrementing loop counters. In this example we subtract the immediate value one from the value one stored in register *r1*. The result value zero is written to register *r1*. The *cpsr* is updated with the *ZC* flags being set.

```
PRE    cpsr = nzcvqiFt_USER
       r1 = 0x00000001

       SUBS r1, r1, #1
```

2.1.4 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.
- The Example 3.7 illustrates the use of the inline barrel shifter with an arithmetic instruction.
- The instruction multiplies the value stored in register *r1* by three.

  Example 3.7: Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to

register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

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

        ADD     r0, r1, r1, LSL #1

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

2.1.5 Logical Instructions

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

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

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

Example 3.8: This example shows a logical OR operation between registers r1 and r2. Register r0 holds the result.

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

ORR r0, r1, r2

POST r0 = 0x12345678

EXAMPLE 3.9 This example shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC   r0, r1, r2

POST    r0 = 0b1010
```

This is equivalent to

```
Rd = Rn AND NOT(N)
```

- In this example, register *r2* contains a binary pattern where every binary *1* in *r2* clears a corresponding bit location in register *r1*.
- This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the *cpsr*.

  NOTE:**The logical instructions update the *cpsr*flags only if the S suffix is present. These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.**

2.1.6 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.
- It is not required to apply the S suffix for comparison instructions to update the flags.

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

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

N is the result of the shifter operation.

EXAMPLE 3.10 This example shows a CMP comparison instruction. You can see that both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *z* flag prior to execution is 0 and is represented by a lowercase *z*. After execution the *z* flag changes to 1 or an uppercase *Z*. This change indicates *equality*.

```
PRE   cpsr = nzcvqiFt_USER
      r0 = 4
      r9 = 4

      CMP   r0, r9

POST  cpsr = nZcvqiFt_USER
```

- The CMP is effectively a subtract instruction with the result discarded; similarly the TSTinstruction is a logical AND operation, and TEQ is a logical exclusive OR operation.
- For each, the results are discarded but the condition bits are updated in the *cpsr*.
- It is important to understand that comparison instructions only modify the condition flags of the*cpsr*and do not affect the registers being compared.

2.1.7 Multiply Instructions:

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

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

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

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

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

- The number of cycles taken to execute a multiply instruction depends on the processor implementation. For some implementations the cycle timing also depends on the value in *Rs*.

EXAMPLE 3.11 This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into register *r0*. In this example, register *r1* is equal to the value 2, and *r2* is equal to 2. The result, 4, is then placed into register *r0*.

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

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

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

- The long multiply instructions **(SMLAL, SMULL, UMLAL, and UMULL)** produce a 64-bit result. The result is too large to fit a single 32-bit register so the result is placed in two registers labelled***RdLo***and ***RdHi***.

- ***RdLo*holds the lower 32 bits of the 64-bit result, and *RdHi*holds the higher 32 bits of the 64-bit result.** The following shows an example of a long unsigned multiply instruction.

EXAMPLE 3.12 The instruction multiplies registers *r2* and *r3* and places the result into register *r0* and *r1*. Register *r0* contains the lower 32 bits, and register *r1* contains the higher 32 bits of the 64-bit result.

```
PRE    r0 = 0x00000000
       r1 = 0x00000000
       r2 = 0xf0000002
       r3 = 0x00000002

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

POST   r0 = 0xe0000004 ; = RdLo
       r1 = 0x00000001 ; = RdHi
```

2.2 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. The ARMv5Einstruction set includes four different branch instructions.

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

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

The address ***label***is stored in the instruction as a signed ***pc*-relative offset** and must be within approximately 32 MB of the branch instruction.

- *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

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

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

- In this example, ***forward*** **and** ***backward***are the labels. The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.
- **The branch with link, or BL**, instruction is similar to the B instruction but overwrites the link register ***lr***with a return address. It performs a subroutine call.

EXAMPLE 3.14 The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register *lr* with a return address. It performs a subroutine call. This example shows a simple fragment of code that branches to a subroutine using the BL instruction. To return from a subroutine, you copy the link register to the *pc*.

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

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

2.3 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.
- There are 3 types of load and store instructions. They are:
    1. Single-Register Transfer
    2. Multiple Register Transfer
    3. Swap

2.3.1 Single-Register Transfer

- These instructions are used for moving a single data item in and out of a register.
- The data types supported are signed and unsigned words (32-bit), half-words (16-bit), and bytes. Here are the various load-store single-register transfer instructions.

Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1

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

STR{<cond>}H Rd, addressing2

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*
LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

- LDR and STR instructions can load and store data on a boundary alignment that is the same as the data type size being loaded or stored.

EXAMPLE 3.15 LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored. 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]
```

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

2.3.2 Single-Register Load-Store Addressing Modes

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

Table: Index Methods

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

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

- ***Preindex with writeback*** calculates an address from a base register plus address offset and then updates that address base register with the new address.

  Example:

  PRE r0 =0x00000000 r1 = 0x00090000

  mem32[0x00009000] = 0x01010101

  mem32[0x00009004] = 0x02020202

  LDR r0, [r1, #4]! ;Preindexing with writeback:

POST

r0 = 0x02020202

r1 = 0x00009004

- ***Preindex***offset is the same as the preindex with writeback but does not update the address base register.

  Example:

  PRE r0 =0x00000000 r1 = 0x00090000

  mem32[0x00009000] = 0x01010101

  mem32[0x00009004] = 0x02020202

  LDR r0, [r1, #4] ;Preindexing

  POST

  r0 = 0x02020202

  r1 = 0x00009000

  The preindex mode is useful for accessing an element in a data structure.

- ***Postindex***only updates the address base register after the address is used. The postindex and preindex with writeback modes are useful for traversing an array.

  Example:

  PRE r0 =0x00000000 r1 = 0x00090000

  mem32[0x00009000] = 0x01010101

  mem32[0x00009004] = 0x02020202

  LDR r0, [r1],#4 ;Postindexing

  POST

  r0 = 0x01010101

  r1 = 0x00009004

- The addressing modes available with a particular load or store instruction depend on the instruction class. The following Table shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.

Table: Single-Register Load-Store Addressing, Word or Unsigned Byte

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

- A signed offset or register is denoted by "+/-", identifying that it is either a positive or negative offset from the base address register *Rn*. The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.
- **Immediate offset** means the address is calculated using the base address register and a 12-bit offset encoded in the instruction.
- **Register offset** means the address is calculated using the base address register and a specific register's contents.
- **Scaled Register Offset** means the address is calculated using the base address register and a barrel shift operation.
- The following Table provides an example of the different variations of the LDR instruction.

Table: Examples of LDR Instructions using Different Addressing Modes

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

- The following Table shows the addressing modes available on load and store instructions using 16-bithalfword or signed byte data.

Table: Single-Register Load-Store Addressing, Halfword, Signed Halfword, Signed Byte and Doubleword

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

- **These operations cannot use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword;**
- **Similarly STRB stores signed and unsigned bytes. The following Table shows the variations for STRH instructions**.

Table: Variations of STRH Instructions

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

2.3.3 Multiple-Register Transfer

- Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction.
- **The transfer occurs from a base address register *Rn* pointing into memory**.
- Multiple-register transfer instructions are more efficient from single-register transfers for moving blocks of data around memory and saving and restoring context and stacks.
- Load-store multiple instructions can increase interrupt latency. ARM implementations do not usually interrupt instructions while they are executing.

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

- **Compilers, such as *armcc*,** provide a switch to control the maximum number of registers being transferred on a load-store, which limits the maximum interrupt latency.

```
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

- The following Table shows the different addressing modes for the load-store multiple instructions. Here *N*is the number of registers in the list of registers.

Table: 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

- Any subset of the current bank of registers can be transferred to memory or fetched from memory.
- **The base register *Rn* determines the source or destination address for a load-store multiple instruction. This register can be optionally updated following the transfer. This occurs when register *Rn* is followed by the ! character, similar to the single-register load-store using preindex with writeback.**

Example 3.17: In this example, register *r0* is the base register *Rn* and is followed by !, indicating that the register is updated after the instruction is executed. You will notice within the load multiple instruction that the registers are not individually listed. **Instead the "-" character is used to identify a range of registers. In this case the range is from register *r1* to *r3* inclusive.Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.**

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

- The following Figure shows a graphical representation.

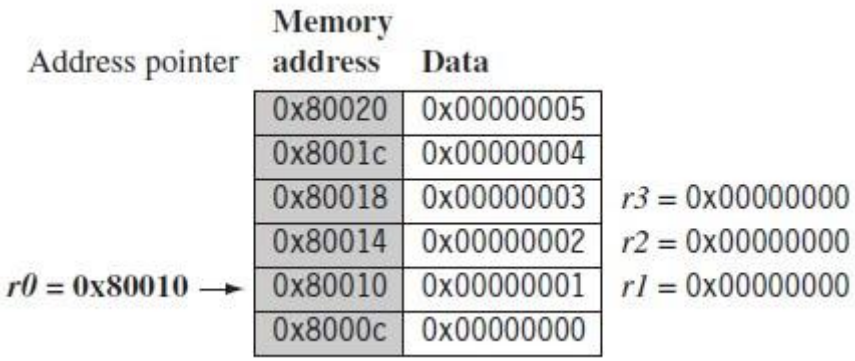


Figure: Pre-condition for LDMIA Instruction

- The base register *r0* points to memory address 0x80010 in the PRE condition.
- Memory addresses 0x80010, 0x80014, and 0x80018 contain the values 1, 2, and 3 respectively.
- After the load multiple instruction executes, registers *r1*, *r2*, and *r3* contain these values as shown in the following Figure.

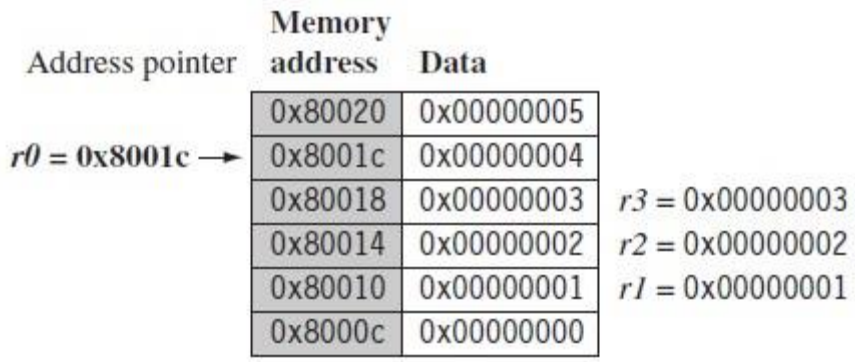


Figure: Post Condition for LDMIA Instruction

- The base register *r0* points to memory address 0x80010 in the PRE condition.

- Memory addresses 0x80010, 0x80014, and 0x80018 contain the values 1, 2, and 3 respectively.
- After the load multiple instruction executes, registers *r1*, *r2*, and *r3* contain these values as shown in the following Figure.

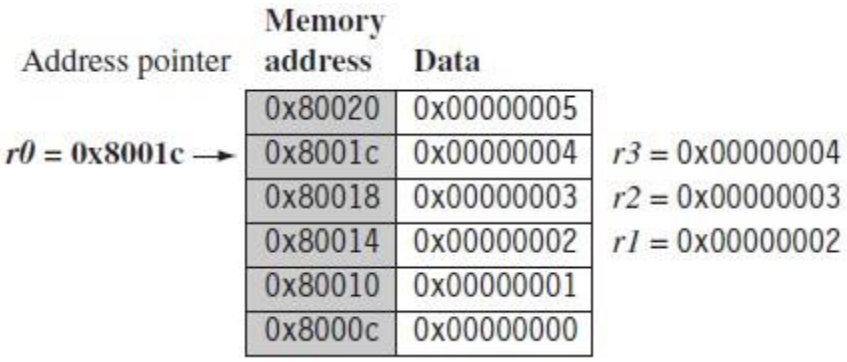


Figure: Post Condition for LDMIB Instruction

- After execution, register *r0* now points to the last loaded memory location. This is in contrast with the LDMIA example, which pointed to the next memory location.
- **The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store to ascending memory locations.**
- This is equivalent to descending memory but accessing the register list in reverse order.
- **With the increment and decrement load multiples; you can access arrays forwards or backwards.**
- **They also allow for stack push and pull operations**.
- The following Table shows a list of load-store multiple instruction pairs.

Table: Load-Store Multiple Pairs when Base Update used

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

- **If you use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer.**
- This is useful when you need to temporarily save a group of registers and restore them later.

Example 3.18: This example shows an STM increment before instruction followed by an LDM decrement after instruction.

PRE r0 = 0x00009000 r1 = 0x00000009 r2 = 0x00000008 r3 = 0x00000007

STMIB r0!, {r1–r3} MOV r1, #1

MOV r2, #2

MOV r3, #3

PRE(2) r0 = 0x0000900c r1 = 0x00000001 r2 = 0x00000002 r3 = 0x00000003

LDMDA r0!, {r1–r3}

POST r0 = 0x00009000 r1 = 0x00000009 r2 = 0x00000008 r3 = 0x00000007

- The STMIB instruction stores the values 7, 8, 9 to memory. We then corrupt register *r1* to *r3*. The LDMDA reloads the original values and restores the base pointer *r0*.

Example 3.19: We illustrate the use of the load-store multiple instructions with a block memory copy example. This example isa simple routine that copies blocks of 32 bytes from a source addresslocation to a destination address location.The example has two load-store multiple instructions, which use the same increment after addressing mode.

```
        ; r9 points to start of source data
        ; r10 points to start of destination data
        ; r11 points to end of the source
loop
        ; load 32 bytes from source and update r9 pointer
                LDMIA r9!, {r0–r7}
        ; store 32 bytes to destination and update r10 pointer
                STMIA r10!, {r0–r7}     ; and store them
        ; have we reached the end
                CMP   r9, r11
                BNE loop
```

- **This routine relies on registers *r9*, *r10*, and *r11* being set up before the code is executed.**
- Registers *r9* and *r11* determine the data to be copied, and register *r10* points to the destination in memory for the data.
- **LDMIA loads the data pointed to by register *r9* into registers *r0* to *r7*. It also updates *r9* to point to the next block of data to be copied.**
- **STMIA copies the contents of registers r0 to r7 to the destination memory address pointed to by register r10. It also updates r10 to point to the next destination location.**

- **CMP and BNE** compare pointers *r9* and *r11* to check whether the end of the block copy has been reached.
- If the block copy is complete, then the routine finishes; otherwise the loop repeats with the updated values of register *r9* and *r10*.
- The BNE is the branch instruction B with a condition mnemonic NE (not equal). If the previous compare instruction sets the condition flags to not equal, the branch instruction is executed.
- The following Figure shows the memory map of the block memory copy and how the routine moves through memory.

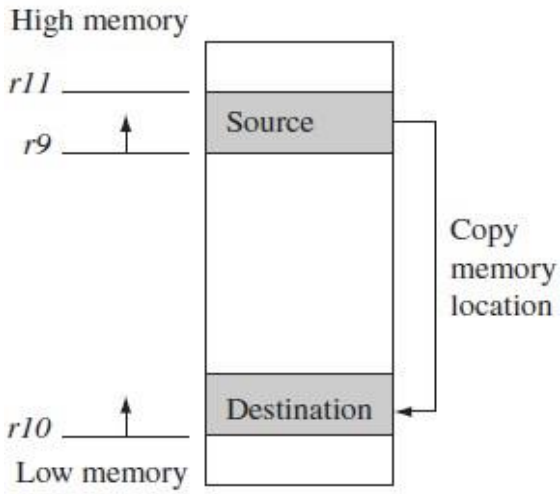


Figure: Block Memory Copy in the Memory map

- Theoretically this loop can transfer 32 bytes (8 words) in two instructions, for a maximum possible throughput of 46 MB/second being transferred at 33 MHz. These numbers assume a perfect memory system with fast memory.

Stack Operation: The ARM architecture uses the load-store multiple instructions to carry out stack operations.

- The ***pop operation***(removing data from a stack) uses a load multiple instruction.
- The ***push operation***(placing data onto the stack) uses a store multiple instruction.
- When using a stack you have to decide whether the stack will grow up or down in memory.
    - **A stack is either –**
        - ***ascending (A)***– stacks grow towards higher memory addresses or
        - ***descending (D)*** – stacks grow towards lower memory addresses.
- When you use a ***full stack (F),*** the stack pointer ***sp*** points **to an address that is the last used or full location** (i.e., *sp* points to the last item on the stack).

- If you use **an *empty stack (E)*** the ***sp*** **points to an address that is the first unused or empty location** (i.e., it points after the last item on the stack).
- There are number of load-store multiple addressing mode aliases available to support stack operations (see the following Table).

Table: 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

- Next to the ***pop*** column is the actual load multiple instruction equivalent.
- For example, **a full ascending stack would have the notation FA appended to the load multiple instruction—LDMFA.** This would be translated into an LDMDA instruction.
- ARM has specified an ***ARM-Thumb Procedure Call Standard (ATPCS)*** that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the *pop* and *push* functions, respectively.

Example 3.20: The STMFD instruction pushes registers onto the stack, updating the *sp*. The following Figure shows a *push* onto a full descending stack.

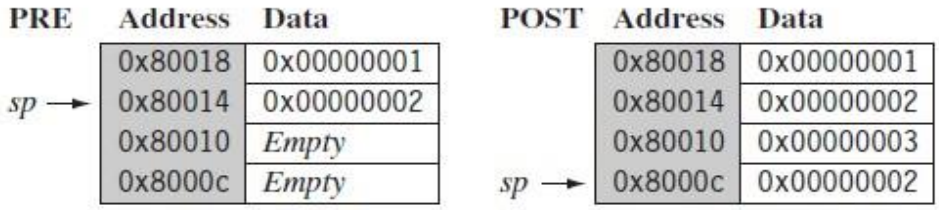


Figure: STMFD Instruction – Full Stack *push* Operation

- You can see that when the stack grows the stack pointer points to the last full entry in the stack.

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

STMFD sp!, {r1, r4}

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

Example 3.21: The following Figure shows a *push* operation on an empty stack using the STMED instruction.

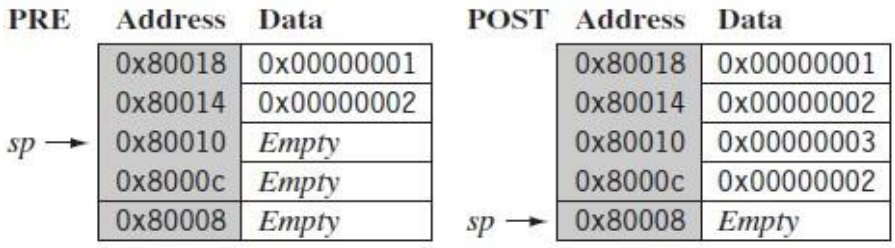


Figure: STMED Instruction – Empty Stack *push* Operation

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

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

  STMED sp!, {r1, r4}

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

- When handling a checked 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 error has occurred.**
- Here is a 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

- ATPCS defines register *r10* as the stack limit or *sl*. This is optional since it is only used when stack checking is enabled.
- The **BLLO** instruction is a branch with link instruction plus the condition mnemonic LO.
  - If ***sp*** is less than register *r10* after the new items are pushed onto the stack, then ***stack overflow*** error has occurred.
  - If the stack pointer goes back past the stack base, then a ***stack underflow*** error has occurred.

2.3.4 Swap Instructions

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

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

SWP	swap a word between memory and a register	$tmp = mem32[Rn]$ $mem32[Rn] = Rm$ $Rd = tmp$
SWPB	swap a byte between memory and a register	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

- Swap cannot be interrupted by any other instruction or any other bus access. We say the system **"holds the bus"** until the transaction is complete. Also**, swap instruction allows for both a word and a byte swap.**
- **Example:** The swap instruction loads a word from memory into register *r0* and overwrites the memory with register *r1*.

PRE mem32[0x9000] = 0x12345678 r0 = 0x00000000

r1 = 0x11112222 r2 = 0x00009000

SWP r0, r1, [r2]

POST mem32[0x9000] = 0x11112222 r0 = 0x12345678

r1 = 0x11112222 r2 = 0x00009000

Example 3.23: 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*. When the semaphore equals *1*, then the service in question is being used by another process. The routine will continue to loop around until the service is released by the other process—in other words, when the semaphore address location contains the value 0.

2.4 SOFTWARE INTERRUPT INSTRUCTION

- A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.
- When the processor executes an SWI instruction, it sets the program counter pc to the offset 0x8 in the vector table.
- **The instruction also forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.**
- **Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.**

Example 3.24: Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM Toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

```
PRE   cpsr  = nzcVqift_USER
      pc  =  0x00008000
      lr = 0x003fffff          ;lr = r14 r0 = 0x12
      0x00008000   SWI   0x123456
POST  cpsr  = nzcVqIft_SVC
      spsr  =  nzcVqift_USER pc = 0x00000008
      lr = 0x00008004 r0 = 0x12
```

- Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers. In this example, register r0 is used to pass the parameter 0x12. The return values are also passed back via registers.
- Code called the ***SWI handler*** is required to process the SWI call. The handler obtains the SWI number using the address of the executed instruction, which is calculated from the link register *lr*.

The SWI number is determined by

SWI_Number = <SWI instruction>AND NOT (0xff000000)

- Here the *SWI instruction* is the actual 32-bit SWI instruction executed by the processor.

Example 3.25: This example shows the start of an SWI handler implementation. The code fragment determines what SWI number is being called and places that number into register *r10*. You can see from this example that the load instruction first copies the complete SWI instruction into register *r10*. The BIC instruction masks off the top bits of the instruction, leaving the SWI number. We assume the SWI has been called from ARM state.

```
SWI_handler
        ; Store registers r0-r12 and the link register
        STMFD sp!, {r0–r12, lr}      ; Read the SWI instruction
        LDR r10, [lr, #–4]           ; Mask off top 8 bits
        BIC r10, r10, #0xff000000    ; r10 - contains the SWI number
        BL service_routine           ; return from SWI handler
        LDMFD sp!, {r0–r12, pc}
```

- The number in register *r10* is then used by the SWI handler to call the appropriate SWI service routine.

2.5 PROGRAM STATUS REGISTER INSTRUCTIONS

- The ARM instruction set provides two instructions to directly control a ***program status register (psr)*.**
- The ***MRS instruction*** transfers the contents of either the ***cpsr* or *spsr*** into a register.
- The ***MSR instruction*** transfers the contents of a register into the *cpsr*or *spsr*. Together these instructions are used to read and write the *cpsr*and *spsr*.
- In the syntax we can see a *label* called fields. This 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*

- These fields relate to particular byte regions in a psr, as shown in the following Figure.

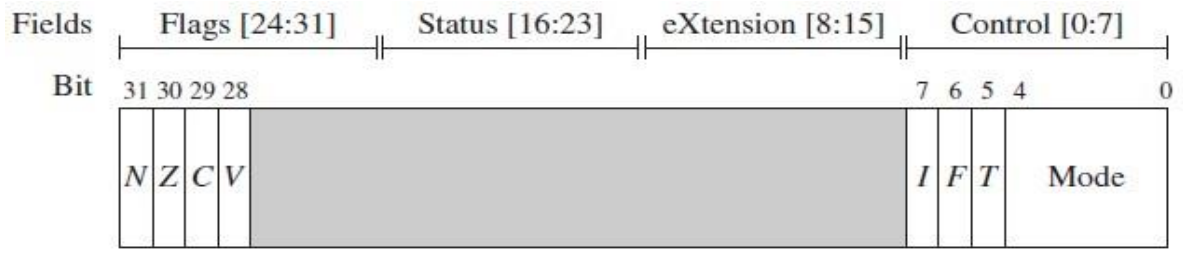


Figure: psr Byte Fields

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

Example 3.26: 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. You can see from this example that this code preserves all the other settings in the cpsr and only modifies the I bit in the control field.

PRE cpsr = nzcvqIFt_SVC

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

POST cpsr = nzcvqiFt_SVC

- 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.

2.5.1 Coprocessor Instructions

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

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

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

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

Example 3.27: This example shows a CP15 register being copied into a general-purpose register.

```
; 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.

2.6 LOADING CONSTANTS

- You might have noticed that there is no 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.
- To aid programming there are two pseudo-instructions to move a 32-bit value into a register.

```
Syntax: LDR Rd, =constant
        ADR Rd, label
```

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

- The first pseudo-instruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.

- The second pseudo-instruction writes a relative address into a register, which will be encoded using a pc-relative expression.

Example 3.28: This example shows an LDR instruction loading a 32-bit constant *0xff00ffff* into register *r0*.

```
        LDR r0, [pc, #constant_number-8-{PC}]
        :
CONSTANT_NUMBER
        DCD   0xff00ffff
```

- This example involves a memory access to load the constant, which can be expensive for time-critical routines.

Example 3.29: The following Example shows an alternative method to load the same constant into register r0 by using an MVN instruction. Loading the constant 0xff00ffff using an MVN.

```
PRE   none...
      MVN r0, #0x00ff0000
POST  r0 = 0xff00ffff
```

- As you can see, there are alternatives to accessing memory, but they depend upon the constant you are trying to load.
- The LDR pseudo-instruction either inserts an MOV or MVN instruction to generate a value (if possible) or generates an LDR instruction with a pc-relative address to read the constant from a literal pool—a data area embedded within the code.
- The following Table shows two pseudo-code conversions.

Table: LDR pseudo-instruction Conversion

Pseudoinstruction	Actual instruction
`LDR r0, =0xff`	`MOV r0, #0xff`
`LDR r0, =0x55555555`	`LDR r0, [pc, #offset_12]`

- The first conversion produces a simple MOV instruction; the second conversion produces a pc-relative load.
- Another useful pseudo-instruction is the ADR instruction, or address relative. This instruction places the address of the given label into register Rd, using a pc-relative add or subtract.

2.7 Assembly Programming using Assembly Language

- Optimizing code reduces the

- System power consumption and
- Reduce the clock speed needed for real-time operation.

- Optimization can turn an infeasible system into a feasible one, or an uncompetitive system into a competitive one.
- Maximum performance can be achieved using hand-written assembly code.
- Writing assembly code gives you direct control of three optimization tools that you cannot explicitly use by writing C source:

```
#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;
 }
```

Three optimization Tools

1. Instruction scheduling: *Reordering the instructions in a code sequence to avoid processor Stalls because of dependency*.

2. Register allocation: *Deciding how variables should be allocated to ARM registers or stack* locations for maximum performance.

3. *Conditional execution:* *Accessing the full range of ARM condition codes and conditional* instructions.

2.7.1 Writing Assembly Code

Here we use ARM Macro Assembler ***armasm*** for example. You can also use the GNU assembler ***gas.***

This section gives examples showing how to convert the C function to basic assembly code.

Example 1: This example shows how to convert a C function to an assembly function. Consider the simple C program main.cfollowing that prints the squares of the integers from 0 to 9:

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

Main1.c

```
#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));
  }
}
```

Square.s

```
        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
```

- **AREA** directive names the area or code section that the code lives in.
- **.text.** : Read Only code **| .text |**: Alphanumeric character in prgm name
- **EXPORT** directive makes the symbol square available for external linking.
- **armasm** treats non-indented text as a label definition.
- Parameter passing is defined by the **ARM-Thumb Procedure Call Standard (ATPCS)**.
- The input argument is passed in **register r0,** and the return value is returned in **register r0.**
- The multiply instruction has a restriction that the destination register must not be the same as the first argument register.
- **END** directive marks the end of the assembly file and Comments follow a semicolon.

The following script illustrates how to build this example using command line tools.

```
armcc -c main1.c
armasm square.s
armlink -o main1.axf main1.o square.o
```

2.7.2 Profiling and Cycle 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. You use a profiler 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.

- The ARM simulator used by the ***ADS1.1 debugger*** is called the ***ARMulator*** and provides profiling and cycle counting features.

2.7.3 Instruction Scheduling

- The time taken to execute instructions depends on the implementation pipeline.
- Here, we consider the ***ARM9TDMI*** pipeline timings.
- Instructions that are conditional on the value of the ARM condition codes in the *cpsr* take one cycle if the condition is not met. If the condition is met, then the following rules apply:

1. ***ALU*** operations such as ***addition, subtraction, and logical operations*** take ***one cycle***. This includes a ***shift by an immediate value***. If you use a ***register-specified shift, then add one cycle***. If the ***instruction writes to the pc***, then ***add two cycles***.
2. Load instructions that load ***32-bit words of memory such as LDR and LDM take N cycles*** to issue. If the instruction loads ***pc, then add two cycles***.
3. Load instructions that load ***16-bit or 8-bit data*** such as ***LDRB, LDRSB, LDRH,*** and ***LDRSH*** take ***one cycle to issue***.
4. ***Branch instructions*** take ***three cycles.***
5. ***Store instructions*** that ***store N values take N cycles***.
6. ***Multiply instructions*** take a ***varying number of cycles*** depending on the value of the second operand

ARM Pipeline and Dependencies

- To understand how to schedule code efficiently on the ARM, we need to understand the ARM pipeline and dependencies.
- The ARM9TDMI processor performs five operations in parallel:

1. Fetch

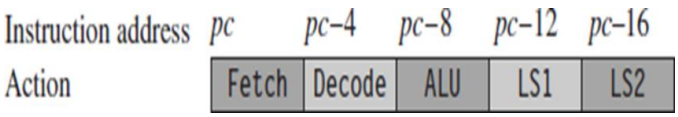
2. Decode

3. Execute (ALU)

4. LS1

5. LS2

- **Fetch:** Fetches instruction from memory using *pc.*
  - ***The instruction is then loaded into the* core** and then processed down the core pipeline.
- **Decode**: Decode the instruction that was fetched in the previous cycle.
  - The processor also reads the input operands from the register bank or if they are not available via one of the forwarding paths.
- **Execute (ALU):** Executes the instruction that was decoded in the previous cycle.
  - This instruction was originally fetches the address from ***pc −8 (ARM state) or pc − 4 (Thumb state).***
  - This stage calculates
    - The answer for a data processing operation,
    - The address for a load, store, or branch operation.



  - Some instructions take several cycles in this stage.

Example: multiply and register-controlled shift operations take several ALU cycles.

- ***LS1****: This stage Load or store the data specified by a load or store instruction.*
  - *If the instruction is* not a load or store, then this stage has no effect or bypassed.
- ***LS2:*** *Sign-extend the data loaded by a byte or half-word load* instruction.
  - If the instruction is not a load of an 8-bit byte or 16-bit half-word item, then this stage has no effect.

If an instruction requires the result of a previous instruction that is not available, then the processor stalls. This is called a **Pipeline Hazard or Pipeline Interlock**.

Example1: Case with No Interlock

```
ADD   r0, r0, r1   Cycle 1: r0+r1
ADD   r0, r0, r2   Cycle 2: r0+r2
```

Example2: Case with 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

- Instruction pair takes three cycles.
- The ALU calculates the address *r2 + 4* in the first cycle while decoding the *ADD* instruction in parallel.
- *ADD* cannot proceed on the second cycle because the load instruction has not yet loaded the value of *r1*. Therefore the *pipeline stalls* (*pipeline bubble*) for one cycle.
- Now that *r1* is ready, the processor executes the *ADD* on the third cyclExample3: One-cycle interlock caused by delayed load use.

Example 3: 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

- Instruction triplet takes four cycles.
- *ADD* proceeds on the cycle following the load byte, the *EOR* instruction cannot start on the third cycle.
- The *r1* value is not ready until the load instruction completes the *LS2* stage of the pipeline. The processor stalls the *EOR* instruction for one cycle.
- Note that the *ADD* instruction does not affect the timing at all. The sequence takes four cycles whether it is there or not!
- The *ADD* doesn't cause any stalls since the *ADD* does not use *r1*, the result of the load.

Example 4: Why a branch instruction takes three cycles? The processor must flush the pipeline when jumping to a new address

```
        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	–	–

- Three executed instructions take a total of five cycles.
- The *MOV* instruction executes on the first cycle.
- In the second cycle, the branch instruction calculates the destination address. This causes the core to flush the pipeline and refill it using this new *pc* value. The refill takes two cycles.
- Finally, the *SUB* instruction executes normally

Scheduling of Load Instruction

- Load instructions occur frequently in the compiled code accounting for approximately one-third of all instructions.
- Careful scheduling of load instructions so that pipeline stalls don't occur can improve performance.
- The compiler attempts to schedule the code as best it can, but the aliasing problem of *C* limits the available optimizations.
- The compiler cannot move a load instruction before a store instruction unless it is certain that the two pointers used do not point to the same address.
- Let's consider an example of a memory-intensive task.
- The following function, *str_tolower*, copies a zero-terminated string of characters from *in* to *out*. It converts the string to lowercase in the process

```
void str_tolower(char *out, char *in)
{
  unsigned int c;

  do
  {
    c = *(in++);
    if (c>='A' && c<='Z')
    {
      c = c + ('a' -'A');
    }
    *(out++) = (char)c;
  } while (c);
}
```

```
str_tolower
        LDRB    r2,[r1],#1    ; c = *(in++)
        SUB     r3,r2,#0x41   ; r3 = c -'A'
        CMP     r3,#0x19      ; if (c <='Z'-'A')
        ADDLS   r2,r2,#0x20   ;    c +='a'-'A'
        STRB    r2,[r0],#1    ; *(out++) = (char)c
        CMP     r2,#0         ; if (c!=0)
        BNE     str_tolower   ;    goto str_tolower
        MOV     pc,r14        ; return
```

- Compiler optimizes the condition (*c >= 'A' && c <= 'Z'*) to the check that *0 <= c-'A' <= 'Z'-'A'*.
- The compiler can perform this check using a single unsigned comparison
- Unfortunately, the *SUB* instruction uses the value of directly after the *LDRB* instruction that loads *c*.
- Consequently, the *ARM9TDMI* pipeline will stall for two cycles. The compiler can't do any better since everything following the load of *c* depends on its value.

The ADS1.1 compiler generates the following compiled output. Notice that the compiler optimizes the condition (c>='A'&&c<='Z')to the check that 0<=c-'A'<='Z'-'A. The compiler can perform this check using a single unsigned comparison.

Cycle Count

```
str_tolower
    LDRB r2,[r1],#1              ; c = *(in++)          cycle-1
                                                        cycle-2 stall
                                                        cycle-3 stall
    SUB r3,r2,#0x41              ; r3 = c -'A'          cycle-4
    CMP r3,#0x19                 ; if (c <='Z'-'A')     cycle-5
    ADDLS r2,r2,#0x20            ; c +='a'-'A'          cycle-6
    STRB r2,[r0],#1              ; *(out++) = (char)c   cycle-7
    CMP r2,#0                    ; if (c!=0)            cycle -8
    BNE str_tolower              ; goto str_tolower     cycle-9
                                                        cycle-10 flush
                                                        cycle-11 flush
MOV pc,r14 ; return
```

Unfortunately, the SUB instruction uses the value of c directly after the LDRB instruction that loads c. Consequently, the ARM9TDMI pipeline will stall for two cycles. The compiler can't do any better since everything following the load of cdepends on its value.

- However, there are two ways you can alter the structure of the algorithm to avoid the cycles by using assembly.

- We call these methods load scheduling by ***preloading*** and ***unrolling.***
- **Preloading:** Load the data required for the loop at the end of the previous loop, rather than at the beginning of the current loop. To get performance improvement with little increase in code size, we don't unroll the loop.

- **Unrolling:** This method of load scheduling works by unrolling and then interleaving the body of the loop. For example, we can perform loop iterations *i*, *i* + 1, *i* + 2 interleaved. When the result of an operation from loop *i* is not ready, we can perform an operation from loop *i* + 1 that avoids waiting for the loop *i* result.

2.7.4 Register Allocation

- 14 of the 16 visible ARM registers can be used o hold general-purpose data.
- The other two registers are: ***stack pointer (r13)***, and the ***program counter, (r15)***.
- For a function to be *ATPCS* compliant it must preserve the calle values of registers *r4* to *r11*.
- *ATPCS* also specifies that the stack should be eight-byte aligned; therefore you must preserve this alignment if calling subroutines.
- Use the following template for optimized assembly routines requiring many registers:

```
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
```

- r12 is also stacked just to keep the stack eight-byte aligned.
    - Address starts from 0 and ends at 28 for r4-r11 (28 is not multiple of 8)
    - Address starts from 0 and ends at 32 for r4-r12 (32 is multiple of 8)
- r12 need not be stacked if your routine doesn't call other ATPCS routines
- For ARMv5 and above you can use the preceding template even when being called from Thumb code.
- For ARM4 you have to use the following template

```
routine_name
       STMFD sp!,      {r4-r12, lr}      ; stack saved registers
         ; body of routine
         ; registers r0-r12 and lr available
       LDMFD sp!,      {r4-r12, lr}      ; restore registers
       BX         lr                     ; return, with mode switch
```

Allocating Variables to Register Members

- It is best to use alternate names to the registers, rather than explicit register numbers.
- Benefits of using alternate names:
  - Allows you to change the allocation of variables to register numbers easily.
  - Allows to 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.
- The following figure you can notice that the register 5 is renamed with X_0 and register 6 with X_1 etc….

```
x_0     RN  5
x_1     RN  6
x_2     RN  7
x_3     RN  8
x_4     RN  9
x_5     RN  10
x_6     RN  11
x_7     RN  12
y_0     RN  4
```

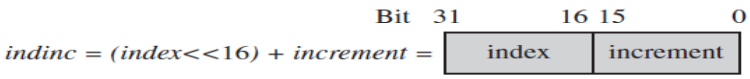
- **There are several cases where the physical number of the register is important:**
  - ***Argument registers:*** The *ATPCS* convention defines that the first four arguments to a function are placed in registers *r0* to *r3*. Further arguments are placed on the stack.
  - ***Registers used in a load or store multiple:*** Load and store multiple instructions *LDM* and *STM* operate on a list of registers in order of ascending register number.
    - ✓ If *r0* and *r1* appear in the register list, then the processor will always load or store *r0* using a lower address than *r1* and so on.
  - ***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*. Furthermore, *Rd* must be an even register number.
  - ***Using More Than 14 Variables:*** If you need more than 14 local 32-bit variables in a routine, then you must store some variables on the stack.

- ✓ The standard procedure is to work outwards from the innermost loop of the algorithm, since the innermost loop has the greatest performance impact.

- ***Making the Most of Available Registers:*** On load-store architecture such as the ARM, it is more efficient to access values held in registers than values held in memory.

  - ✓ There are several tricks you can use to fit several sub-32-bit length variables into a single 32-bit register and thus can reduce code size and increase performance

Example:



There are several possible ways we can proceed when we run out of registers:

- Reduce the number of registers we require by performing fewer operations in each loop. In this case we could load four words in each load multiple rather than eight.
- Use the stack to store the least-used values to free up more registers. In this case we could store the loop counter *N* on the stack. (See Section 6.4.2 for more details on swapping registers to the stack.)
- Alter the code implementation to free up more registers. This is the solution we consider in the following text. (For more examples, see Section 6.4.3.)

We often iterate the process of implementation followed by register allocation several times until the algorithm fits into the 14 available registers. In this case we notice that the carry value need not stay in the same register at all! We can start off with the carry value in y_0 and then move it to y_1 when x_0 is no longer required, and so on. We complete the routine by allocating *kr* to *lr* and recoding so that carry is not required.

2.7.5 Conditional Execution

- The processor core can conditionally execute most ARM instructions.
- By combining conditional execution and conditional setting of the flags, it is possible to implement simple if statements without any need for branches.
- This improves efficiency since branches can take many cycles and also reduces code size.

Example 1: 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'
ADDHS   c, 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. Section 6.3.1 shows a similar use of conditional execution to convert to lowercase.

Conditional execution is even more powerful for cascading conditions.

Example 2: 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
```

- As soon as one of the TEQ comparisons detects a match, the *Z* flag is set in the *cpsr*. The following TEQNE instructions have no effect as they are conditional on $Z = 0$.
- The next instruction to have effect is the ADDEQ that increments vowel. You can use this method whenever all the comparisons in the if statement are of the same type.

Example 3: Consider the following code that detects if c is a letter:

```
if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
{
   letter++;
}
```

To implement this efficiently, we can use an addition or subtraction to move each range to the form $0 \le c \le limit$.

Then we use unsigned comparisons to detect this range and conditional comparisons to chain together ranges. The following assembly implements this efficiently:

```
SUB     temp, c, #'A'
CMP     temp, #'Z'-'A'
SUBHI   temp, c, #'a'
CMPHI   temp, #'z'-'a'
ADDLS   letter, letter, #1
```

2.7.6 Looping Constructs

- Most routines critical to performance will contain a loop.
- Note that, ARM loops are fastest when they count down towards zero.
- Different types of looping constructs available in ARM are:
    1. Decremented Counted Loops
    2. Unrolled Counted Loops
    3. Multiple Nested Loops
    4. Other Counted Loops
- Here we understand to implement count down loops efficiently in assembly and unroll loops for maximum performance.

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,...,1
        SUBS i, i, #1
        BGT  loop
```

- The loop overhead consists of a subtraction setting the condition codes followed by a conditional branch.
- On *ARM7* and *ARM9* this overhead costs four cycles per loop.
- If *i* is an array index, then you may want to count down from *N−1* to *0* inclusive instead so that you can access array element zero. You can implement this in the same way by using a different conditional branch:

```
        SUBS i, N, #1
loop
        ; loop body goes here and i=N-1,N-2,...,0
        SUBS i, i, #1
        BGE  loop
```

- In this arrangement the *Z* flag is set on the last iteration of the loop and cleared for other iterations.
- If there is anything different about the last loop, then we can achieve this using the *EQ* and *NE* conditions.
- There is no reason why we must decrement by one on each loop. Suppose we require *N/3* loops; rather than attempting to divide *N* by three, it is far more efficient to subtract three from the loop counter on each iteration:

```
        MOV i, N
loop
        ; loop body goes here and iterates (round up)(N/3) times
        SUBS i, i, #3
        BGT  loop
```

- The loop overhead consists of a subtraction setting the condition codes followed by a conditional branch.
- On *ARM7* and *ARM9* this overhead costs four cycles per loop.
- If *i* is an array index, then you may want to count down from *N−1* to *0* inclusive instead so that you can access array element zero. You can implement this in the same way by using a different conditional branch:

```
        SUBS i, N, #1
loop
        ; loop body goes here and i=N-1,N-2,...,0
        SUBS i, i, #1
        BGE  loop
```

Unrolled Counted Loops: Loop unrolling reduces the loop overhead by executing the loop body multiple times. However, there are problems to overcome

Multiple Nested Loops: How many loop counters does it take to maintain multiple nested loops?

- Actually, one will suffice—or more accurately, one provided the sum of the bits needed for each loop count does not exceed 32.

Other Counted Loops: You may want to use the value of a loop counter as an input to calculations in the loop. It's not always desirable to count down from *N* to *1* or *N −1* to *0*.

Use of Looping Structures that count in different patterns.

- **Negative Indexing:** This loop structure counts from $-N$ to *0* (inclusive or exclusive) in steps of size *STEP*.

```
        RSB     i, N, #0        ; i=-N
loop
        ; loop body goes here and i=-N,-N+STEP,...,
        ADDS    i, i, #STEP
        BLT     loop            ; use BLT or BLE to exclude 0 or not
```

- **Logarithmic Indexing:** This loop structure counts down from 2^N to *1* in powers of two. For example, if $N = 4$, then it counts 16, 8, 4, 2, 1.

```
                MOV     i, #1
                MOV     i, i, LSL N
        loop
                ; loop body
                MOVS    i, i, LSR#1
                BNE     loop
```

  - The following loop structure counts down from an *N-bit* mask to a one-bit mask. For example, if $N = 4$, then it counts 15, 7, 3, 1.

```
        MOV     i, #1
        RSB     i, i, i, LSL N  ; i=(1<<N)-1
loop
        ; loop body
        MOVS    i, i, LSR#1
        BNE     loop
```

**********End of Module 2 **********

Module 3

Embedded System Components

Contents **Page No.**

Module 3 - Syllabus

Embedded System Components: Embedded Vs General computing system, History of embedded systems, Classification of Embedded systems, Major applications areas of sembedded systems, purpose of embedded systems Core of an Embedded System including all types of processor/controller, Memory, Sensors, Actuators, LED, 7 segment LED display, stepper motor, Keyboard, Push button switch, Communication Interface (onboard and external types), Embedded firmware, Other system components.

Text book 2: Chapter 1(Sections 1.2 to 1.6), Chapter 2(Sections 2.1 to 2.6) RBT: L1, L2

Definition: An embedded system is an **electronic/electro-mechanical** system designed to perform a **specific function** and a combination of both **hardware and firmware** (software).

- Every embedded system is **unique** and the hardware as well as the firmware **is highly specialized** to the application domain.
- Embedded systems are becoming **an inevitable part** of any product or equipment in all fields including household appliances, telecommunications, medical equipment, industrial control consumer products, etc.
- E.g. Electronic Toys, Mobile Handsets, Washing Machines, Air Conditioners, Automotive Control Units, Set Top Box, DVD Player etc…
- The following image shows the example embedded system devices.

3.1 Embedded Vs General computing system

- The computing revolution began with the general purpose computing requirements. Later it was realized that the general computing requirements are not sufficient for the embedded computing requirements.
- Major differences between the embedded systems and the general computing system are listed out in the following table.

Criteria	General Purpose Computing System	Embedded System
Contents	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.
OS	It contains a general purpose operating system (GPOS).	It may or not contain an operating system for functioning.
Alterations	Applications are alterable (programmable) by the user. (It is possible for the end user to re-install the OS 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.
Key factor	Performance is the key deciding factor in the selection of the system. Faster is better.	Application specific requirements (like performance, power requirements, memory usage, etc.) are key deciding factors.
Power Consumption	More	Less
Response Time	Not critical	Critical for some applications
Execution	Need not be deterministic	Deterministic for certain types of ES like 'Hard Real Time' systems.

3.2History of embedded systems

- Embedded systems were in existence before the IT revolution. In the olden days, embedded systems were built around the old vacuum tubes and transistor technologies and algorithms were developed in lower level languages.
- The first recognised modern embedded system is the Apollo Guidance Computer (AGC) developed by MIT Instrumentation Laboratory for the lunar expedition.
- They ran the inertial guidance system of both the Command Module(CM) and the Lunar Excursion Module(LEM).
- The Command Module was designed to encircle the moon while the Lunar Module and its crew were designed to go down the moon surface and land there safely.
- The first mass-produced embedded system was the guidance computer for the Minuteman-I missile in 1961.

3.3Classification of Embedded systems

The criteria used in the classification of embedded systems are as follows:

- Based on Generation
- Based on Complexity & Performance Requirements
- Based on deterministic behavior
- Based on Triggering

3.3.1 Classification based on Generation

First Generation: The early embedded systems built around 8 bit microprocessors like 8085 and Z80 and 4 bit microcontrollers. Hardware circuits are simple and assembly code used for firmware. **Example:** digital telephone keypads, stepper motor control units etc…

Second Generation: Embedded Systems built around 16 bit microprocessors and 8 or 16 bit microcontrollers, following the first generation embedded systems. Instruction set is more complex and powerful than first generation. **Example:** data acquisition system, SCADA systems ect…

Third Generation: Embedded Systems built around high performance 16/32 bit Microprocessors/controllers. Instruction set of these processors became more complex and powerful. It spread its ground to areas like media, networking, robotics etc. **Example:** Application Specific Instruction set processors like Digital Signal Processors (DSPs), and Application Specific Integrated Circuits (ASICs).

Fourth Generation: Embedded Systems built around System on Chips (SoCs), Re-configurable processors and multi-core processors. These systems made use of high performance real time embedded OS for functioning. **Example:** Smart Phones.

3.3.2 Classification based on Complexity & Performance

Small Scale: The early embedded systems built around 8bit microprocessors like 8085 and Z80 and 4bit microcontrollers

Medium Scale: Embedded Systems built around 16bit microprocessors and 8 or 16bit microcontrollers, following the first generation embedded systems

Large Scale/Complex: Embedded Systems built around high performance 16/32 bit Microprocessors/controllers, Application Specific Instruction set processors like Digital Signal Processors (DSPs), and Application Specific Integrated Circuits (ASICs)

3.4 Major applications areas of embedded systems

The application areas and the products in the embedded domain are countless. A few of the important domains and products are listed below:

i. **Consumer electronics:** Camcorders, cameras, etc.
ii. **Household appliances:** Television, DVD players, washing machine, fridge, microwave oven, etc.
iii. **Home automation and security systems:** Air conditioners, sprinklers, intruder detection alarms, closed circuit television cameras, fire alarms, etc.
iv. **Automotive industry:** Anti-lock Braking Systems (ABS), engine control, ignition systems, automatic navigation systems, etc.
v. **Telecom:** Cellular telephones, telephone switches, handset multimedia applications, etc.
vi. **Computer peripherals:** Printers, scanners, fax machines, etc.
vii. **Computer Networking systems:** Network routers, switches, hubs, firewalls, etc.
viii. **Healthcare:** Different kinds of scanners, EEG, ECG machines etc.
ix. **Measurement & Instrumentation:** Digital multi meters, digital CROs, logic analyzers PLC systems, etc.
x. **Banking & Retail:** Automatic teller machines (ATM) and currency counters, point of sales (POS).
xi. **Card Readers:** Barcode, smart card readers, hand held devices, etc.

3.5 Purpose of embedded systems

Embedded systems are used in various domains like consumer electronics, home automation, telecommunications, automotive industry, healthcare, control & instrumentation, retail and banking applications, etc. Within the domain itself, according to the application usage context, they may have different functionalities. Each embedded system is designed to serve the purpose of any one or a combination of the following tasks:

i. Data collection/Storage/Representation

ii. Data Communication
iii. Data (signal) processing
iv. Monitoring
v. Control
vi. Application specific user interface

Each of them are discussed detail as follows:

(i) Data Collection/Storage/Representation

- Embedded systems designed for the purpose of data collection perform 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, such as text, voice, image, video, electrical signals and any other measurable quantities.
- Data can be either analog (continuous) or digital (discrete).
- Embedded systems with analog data capturing techniques collect data directly in the form of analog signal whereas embedded systems with digital data collection mechanism converts the analog signal to the digital signal using analog to digital (A/D) converters and then collects the binary equivalent of the analog data.
- If the data is digital, it can be directly captured without any additional interface by digital embedded systems.
- The collected data may be stored directly in the system or may be transmitted to some other systems or it may be processed by the system or it may be deleted instantly after giving a meaningful representation.
- These actions are purely dependent on the purpose for which the embedded system is designed.

(ii) Data Communication

- Embedded data communication systems are deployed in applications from complex satellite communication systems to simple home networking systems.
- The data collected by an embedded terminal may require transferring of the same to some other system located remotely.
- The transmission is achieved either by a wire-line medium or by a wire-less medium.

- Wire-line medium was the most common choice in all olden days embedded systems.
- As technology is changing, wireless medium is becoming the standard for data communication in embedded systems. It offers cheaper connectivity solutions and make the communication link free from the hassle of wire bundles.
- 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).

- Certain embedded systems act as a dedicated transmission unit between the sending and receiving terminals, offering sophisticated functionalities like data packetizing, encrypting and decrypting.
- 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.

(iii) 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 improve the hearing capacity of hearing-impaired persons.

(iv) Monitoring

- Almost all embedded products coming under the medical domain are with monitoring functions only.
- They are used for determining the state of some variables using input sensors. They cannot impose control over variables.
- 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 of a patient but it cannot impose control over the heartbeat.
- The sensors used in ECG are the different electrodes connected to the patient's body.
- Other examples with monitoring function are measuring instruments like digital CRO, digital multimeters, logic analyzers., etc. used in control & instrumentation applications.
- They are used for knowing (monitoring) the status of some variables like current, voltage, etc. They cannot control the variables in turn.

(v) Control

- Embedded systems with control functionalities impose control over some variables according to the change 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 the input variable to put an impact on the controlling variable to bring the controlled variable to the specified range.
- Air conditioner system used in our home 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 thermistor and a handheld unit for setting up (feeding) the desired temperature.
- The handheld unit may be connected to the central embedded unit residing inside the air conditioner through a wireless link or through a wired link.
- The air compressor unit acts as the actuator. The compressor is controlled according to the current room temperature and the desired temperature set by the end user.

- The input variable is the current room temperature and the controlled variable is also the room temperature. The controlling variable is cool air flow by the compressor unit.
- If the controlled variable and input variable are not at the same value, the controlling variable tries to equalize them through taking actions on the cool air flow.

(vi) Applications specific user interface

- Buttons, switches, keypad, lights, speakers, display units, etc. are application-specific user interfaces.
- Mobile phone is an example of application specific user interface. In mobile phone, the user interface is provided through the keypad, graphic LCD module, system speaker, vibration alert, etc.

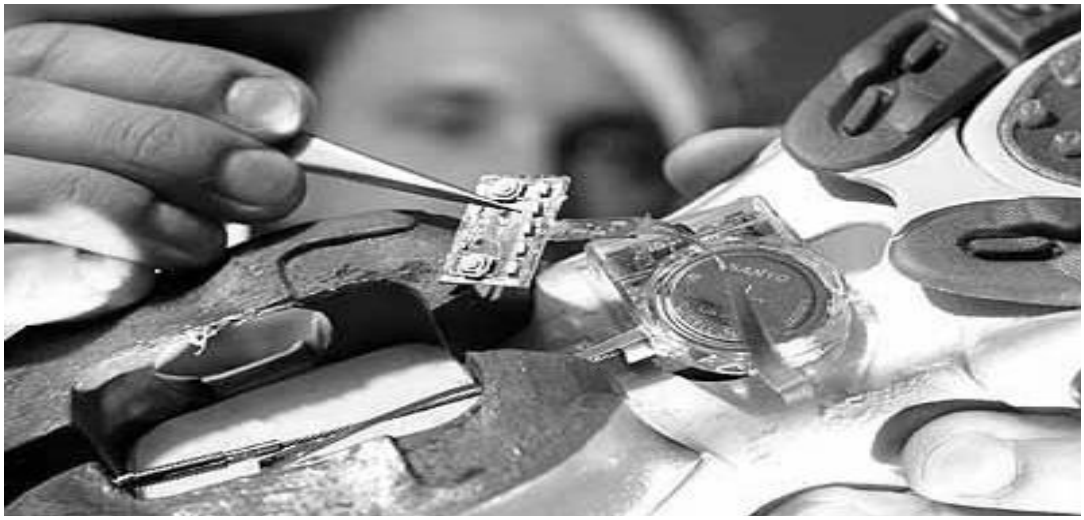
Patient Monitoring system
Photo courtesy of Philips Medical Systems
(www.medical.philips.com/)

Example: 'SMART'RUNNING SHOES FROM ADIDAS—THE INNOVATIVE

BONDING OF LIFESTYLE WITH EMBEDDED TECHNOLOGY

- After three years of extensive research work, Adidas launched the "Smart" running shoes in the market in April 2005.
- The shoe constantly adapts its shock-absorbing characteristics to customize its value to the individual runner, depending on the running style, pace, body weight, and running surface.
- The shoe uses a magnetic sensing system to measure cushioning level, which is adjusted via a digital signal processing unit that controls a motor-driven cable system.

- A hall effect sensor is positioned at the top of the "cushioning element", and the magnet is placed at the bottom of the element. As the cushioning compresses on each impact, the sensor measures the distance from top to bottom of mid-sole (accurate to 0.1 mm).
- About 1000 readings per second are taken and relayed to the shoe's microprocessor. The Microprocessor (MPU) is positioned under the arch of the shoe. It runs an algorithm that compares the compression messages received from the sensor to a preset range of proper cushioning levels, so it understands if the shoe is too soft or too firm.
- Then the MPU sends a command to a micro motor, housed in the mid-foot. The micro motor turns a lead screw to lengthen or shorten a cable secured to the walls of a plastic-cushioning element.
- When the cable is shortened, the cushioning element is pulled taut and compresses very little. A longer cable allows for a more cushioned feel. A replaceable 3V battery powers the motor and lasts for about 100 hours of running.



The Typical Embedded System Components

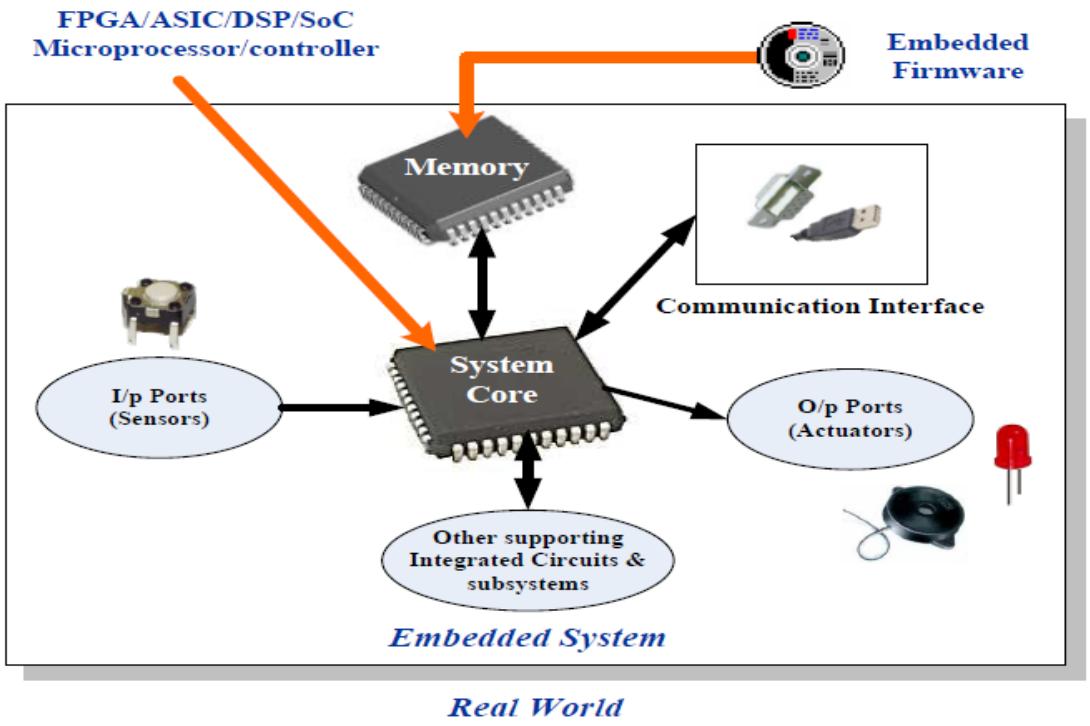


Fig: Elements of an Embedded System

- A typical embedded system contains a single chip controller, which acts as the master brain of the system.
- The controller can be a Microprocessor (e.g. Intel 8085) or a microcontroller (e.g. Atmel AT89C51) or a Field Programmable Gate Array (FPGA) device (e.g. Xilinx Spartan) or a Digital Signal Processor (DSP) (e.g. Blackfin® Processors from Analog Devices) or an Application Specific Integrated Circuit (ASIC)/Application Specific Standard Product (ASSP) (e.g. ADE7760 Single Phase Energy Metering IC from) Analog Devices for energy metering applications).
- Embedded hardware/software systems are basically designed to regulate a physical variable or to manipulate the state of some devices by sending some control signals to the Actuators or devices connected to the O/p ports of the system, in response to the input signals provided by the end users or Sensors which are connected to the input ports.
- Hence an embedded system can be viewed as a reactive system. The control is achieved by processing the information coming from the sensors and user interfaces, and controlling some actuators that regulate the physical variable.

- Key boards, push button switches, etc. are examples for common user interface input devices whereas LEDs, liquid crystal displays, piezoelectric buzzers, etc. are examples for common user interface output devices for a typical embedded system.
- The Memory of the system is responsible for holding the control algorithm and other important configuration details. For most of embedded systems, the memory for storing the algorithm or configuration data is of fixed type, which is a kind of Read Only Memory (ROM)
- The most common types of memories used in embedded systems for control algorithm storage are OTP, PROM, UVEPROM, EEPROM and FLASH. Depending on the control application, the memory size may vary from a few bytes to megabytes.
- Sometimes the system requires temporary memory for performing arithmetic operations or control algorithm execution and this type of memory is known as "working memory". Random Access Memory (RAM) is used in most of the systems as the working memory. Various types of RAM like SRAM, DRAM and NVRAM are used for this purpose.
- The size of the RAM also varies from a few bytes to kilobytes or megabytes depending on the application

3.6 Core of an Embedded System including all types of processor/controller

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 Processor
    a. Microprocessors
    b. Microcontrollers
    c. Digital Signal Processors
2. Application Specific Integrated Circuits (ASICs)
3. Programmable Logic Devices (PLDs)
4. Commercial off-the-shelf Components (COTS)

3.6.1. 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 whereas domains which require signal processing such as speech coding, speech recognition, etc. make use of special kind of digital signal processors supplied by manufacturers like, Analog Devices, Texas Instruments, etc.

Microprocessors

- The CPU contains the Arithmetic and Logic Unit (ALU), Control Unit and Working registers
- 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 claims the credit for developing the first Microprocessor unit Intel 4004, a 4 bit processor which was released in Nov 1971

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

- General Purpose Processor or GPP is a processor designed for general computational tasks GPPs are produced in large volumes and targeting the general market. Due to the high-volume production, the per unit cost for a chip is low compared to ASIC or other specific ICs.
- A typical general-purpose processor contains an Arithmetic and Logic Unit (ALU) and Control Unit (CU).
- Application Specific Instruction Set processors (ASIPs) are processors with architecture and instruction set optimized to specific domain/application requirements like Network processing, Automotive, Telecom, media applications, digital signal processing, control applications etc.

Microcontrollers

- A highly integrated silicon chip containing 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.
- Microcontrollers can be considered as a super set of Microprocessors
- Microcontroller can be general purpose (like Intel 8051, designed for generic applications and domains) or application specific (automotive applications)

- Since a microcontroller contains all the necessary functional blocks for independent working, they found greater place in the embedded domain in place of microprocessors
- Microcontrollers are cheap, cost effective and are readily available in the market
- Texas Instruments TMS 1000 is considered as the world‟s first microcontroller

Microprocessor vs Microcontroller

- The following table summarizes the differences between a microcontroller and microprocessor.

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, 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
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 (DSPs)

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** (DSP) are 2 to 3 times faster than the general-purpose microprocessors in signal processing applications
- 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
- DSP can be viewed as a microchip designed for performing high speed computational operations for ‘addition’, ‘subtraction’, ‘multiplication’ and ‘division’
- A typical Digital Signal Processor incorporates the following key units
    - ✓ Program Memory
    - ✓ Data Memory
    - ✓ Computational Engine
    - ✓ I/O Unit
- Audio video signal processing, telecommunication and multimedia applications are typical examples where DSP is employed

RISC vs. CISC Processors/Controllers

The term **RISC** stands for **Reduced Instruction Set Computing**. As the name implies, all RISC processors/controllers possess lesser number of instructions, typically in the range of 30 to 40. **CISC** stands for **Complex Instruction Set Computing**. From the definition itself it is clear that the instruction set is complex and instructions are high in number. **From a programmer’s point of view RISC processors are comfortable since s/he needs to learn only a few instructions, whereas for a CISC processor s/he needs to learn more number of instructions and should understand the context of usage of each instruction** .(This scenario is explained on the basis of a programmer following Assembly Language coding. For a programmer following C coding it doesn’t matter since the cross-compiler is responsible for the conversion of the high-level language instructions to machine dependent code). Atmel AVR microcontroller is an example for a RISC processor and its instruction set

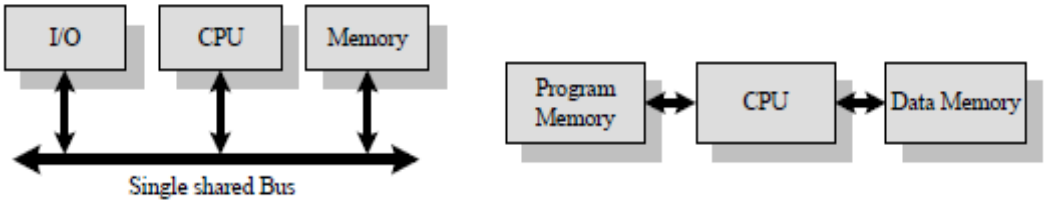
contain only 32 instructions. The original version of 8051 microcontroller (e.g. AT89C51) is a CISC controller and its instruction set contains 255 instructions. There are some other factors like pipelining features, instruction set type, etc. for determining the RISC/CISC criteria. Some of the important criteria are listed below:

RISC	CISC
Lesser no. of instructions	Greater no. 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
Large number of registers are available	Limited no. 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 V/s Von-Neumann Processor/Controller Architecture

- The terms Harvard and Von-Neumann refers to the processor architecture design.
- Microprocessors/controllers based on the Von-Neumann architecture shares a single common bus for fetching both instructions and data. Program instructions and data are stored in a common main memory.
- Microprocessors/controllers based on the Harvard architecture will have separate data bus and instruction bus. With Harvard architecture, 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.

- Since some additional hardware logic is required for the generation of control signals for this type of operation it adds silicon complexity to the system. Figure explains the Harvard and Von-Neumann architecture concept.



Von-Neumann Architecture Harvard Architecture

- The following table highlights the differences between Harvard and Von-Neumann architecture.

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 V/s Little-endian processors

- Endianness specifies the order in which the data is stored in the memory by processor operations in a multi byte system (Processors whose word size is greater than one byte). Suppose the word length is two byte then data can be stored in memory in two different ways

Higher order of data byte at the higher memory and lower order of data byte at location just below the higher memory

Lower order of data byte at the higher memory and higher order of data byte at location just below the higher memory

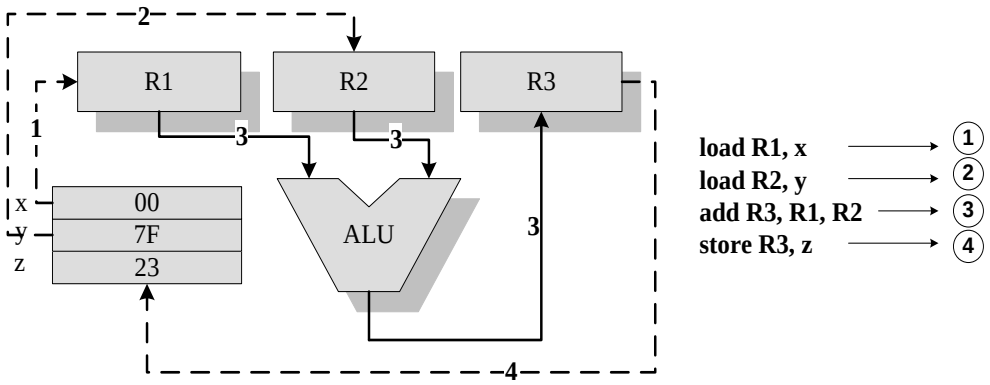
Little-endian means the **lower-order byte** of the data is stored in memory at the lowest address, and the **higher-order byte** at the highest address. (The little end comes first)

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.)

Little-endian Operation **Big-endian Operation**

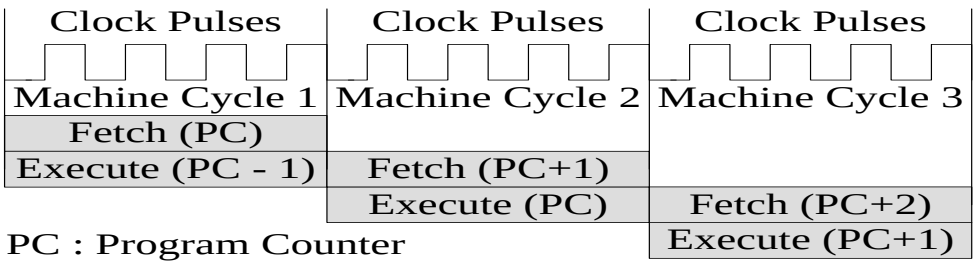
Load Store Operation & Instruction Pipelining

The RISC processor instruction set is orthogonal and it operates on registers. 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 instruction The concept of Load Store Architecture is illustrated with the following example:



Load Store Operation

- 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 2. Under the load store architecture, the same is achieved with 4 instructions as shown in Figure.
- The first instruction load R. 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 adds R3. R1, R2 adds the content of registers R1 and R2 and stores the result in register R3. The next instruction store R3.z stones the content of register R3 in memory location z.
- The conventional instruction execution by the processor follows the fetch-decode-execute sequence. Where the „fetch‟ part fetches the instruction from program memory or code memory and 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. During the decode operation the memory address bus is available and if it is possible to effectively utilize it for an instruction fetch, the processing speed can be increased.
- In its simplest form instruction Pipelining refers to the overlapped execution of instruction. Under normal program execution how it is meaningful to fetch the next instruction to execute, while decoding and execution of the current instruction is in progress.
- 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.



3.6.2 Application Specific Integrated Circuits (ASICs)

- Application Specific Integrated Circuit (ASIC) is a microchip designed to perform a specific or unique application. It is used as replacement to conventional general-purpose logic chips.
- It integrates several functions into a single chip and there by reduces the system development cost.
- As a single chip, ASIC consumes a very small area in the total system and there by helps in the design of smaller systems with high capabilities/functionalities.
- ASICs can be pre-fabricated for a special application or it can be custom fabricated by using the components from a re-usable „building block" library of components for a particular customer application.
- ASIC based systems are profitable only for large volume commercial productions.
- 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 Application Specific Integrated Circuit (ASIC) is made openly available in the market, the ASIC is referred as Application Specific Standard Product (ASSP).

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 can use inexpensive software tools to quickly develop, simulate, and test their logic designs in PLD based design. The design can be quickly programmed into a device, and immediately tested in a live circuit

- PLDs are based on re-writable memory technology and the device is reprogrammed to change the design

Advantages of PLD:

- Programmable logic devices offer a number of important advantages over fixed logic devices, including: 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! Customers who use fixed logic devices often end up with excess inventory which must be scrapped, or if demand for their product surges, they may be caught short of parts and face production delays.
- PLDs can be reprogrammed even after a piece of equipment is shipped to a customer In fact, thanks to programmable log1c devices, a number of equipment manufacturers now tout the ability to 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.

CPLDS and FPGAs

- The two major types of programmable logic devices are **Field Programmable Gate Arrays (FPGAS) and Complex Programmable Logic Devices (CPLDS).**
- Of the two, FPGAS offer the highest amount of logic density, the most features, and the highest performance.
- The largest FPGA now shipping, part of the Xilinx VirtexTM 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.

- PGAs are used in a wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing.

Commercial off the Shelf Component (COTS)

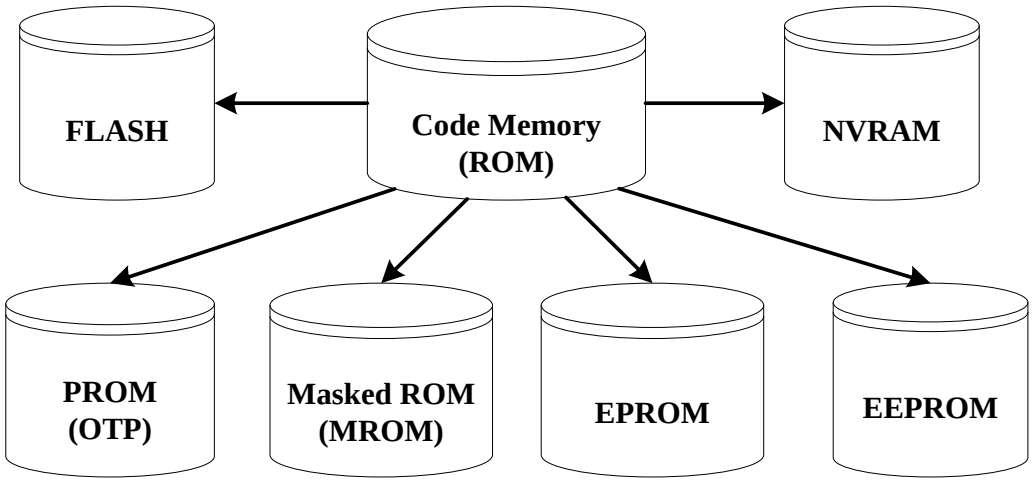
- A Commercial off-the-shelf (COTS) product is one which is used '*as-is*'
- COTS products are designed in such a way to provide easy integration and interoperability with existing system components
- Typical examples for the COTS hardware unit are Remote Controlled Toy Car control unit including the RF Circuitry part, High performance, high frequency microwave electronics (2 to 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
- A COTS component in turn contains a General Purpose Processor (GPP) or Application Specific Instruction Set Processor (ASIP) or Application Specific Integrated Chip (ASIC)/Application Specific Standard Product (ASSP) or Programmable Logic Device (PLD)
- The major advantage of using COTS is that they are readily available in the market, cheap and a developer can cut down his/her development time to a great extent.

3.7Memory

- Memory is an important part of an embedded system. The memory used in embedded system can be either Program Storage Memory (ROM) or Data memory (RAM)
- Certain Embedded processors/controllers contain built in program memory and data memory and this memory is known 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.

3.7.1 Program Storage Memory (ROM)

- Stores the program instructions
- Retains its contents even after the power to it is turned off. It is generally known as Non-volatile storage memory
- Depending on the fabrication, erasing and programming techniques they are classified into



Masked ROM (MROM)

- One-time programmable memory. Uses hardwired technology for storing data. The device is factory programmed by masking and metallization process according to the data provided by the end user.
- *The primary advantage of MROM is low cost for high volume production. They are the least expensive type of solid state memory.*
- Different mechanisms are used for the masking process of the ROM, like

Creation of an enhancement or depletion mode transistor through channel implant

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.

- *The limitation with MROM based firmware storage is the 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*

Programmable Read Only Memory (PROM) / (OTP)

- Unlike MROM it is not pre-programmed by the manufacturer
- PROM/OTP 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” where as 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 proto-typed versions are proven and the code is finalized
- It is a low cost solution for commercial production. OTPs cannot be reprogrammed

Erasable Programmable Read Only Memory (EPROM)

- Erasable Programmable Read Only (EPROM) memory 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 Ultra violet 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 needs to be put in a UV eraser device for 20 to 30 minutes

Electrically Erasable Programmable Read Only Memory (EEPROM)

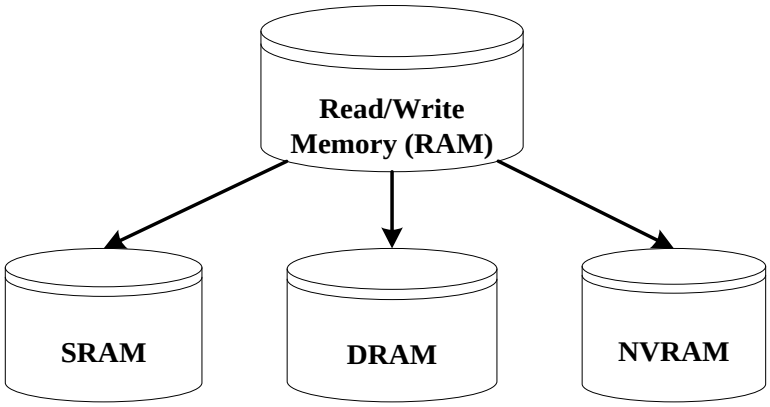
- Erasable Programmable Read Only (EPROM) memory gives the flexibility to re-program the same chip using electrical signals
- The information contained in the EEPROM memory can be altered by using electrical signals at the register/Byte level
- They can be erased and reprogrammed within the circuit
- 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 when compared with the standard ROM (A few kilobytes).

FLASH

- FLASH memory is a variation of EEPROM technology
- It combines the re-programmability of EEPROM and the high capacity of standard ROMs
- FLASH memory is organized 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

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

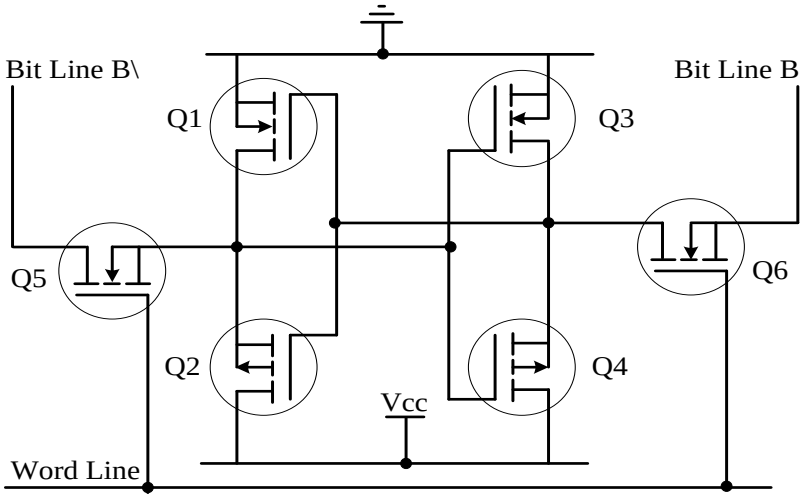
- RAM is the data memory or working memory of the controller/processor.
- Controller/processor can read from it and write to it.
- RAM is volatile, meaning when the power is turned off, all the contents are destroyed.
- RAM is a direct access memory, meaning we can access the desired memory location, directly without the need for traversing through the entire memory locations to reach the desired memory position) (i.e. random access of memory location).
- 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
- In typical implementation, an SRAM cell (bit) is realized using 6 transistors (or 6 MOSFETs). Four of the transistors are used for building the latch (flip-flop) part of the memory cell and 2 for controlling the access.

- Static RAM is the fastest form of RAM available. SRAM is fast in operation due to its resistive networking and switching capabilities. In its simplest representation and SRAM cell can be visualized as shown in Fig:

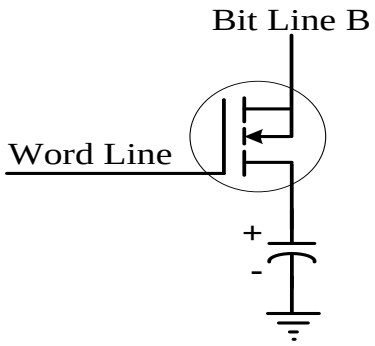


- This implementation in its simpler form can be Visualized as two cross coupled inverters with read/ write control through transistors. The four transistors in the middle form the cross-coupled inverters. This can be visualized as shown in Fig.
- From the SRAM implementation diagram, it is clear that access to the memory cell is controlled by the line Word Line, which controls the access transistors (MOSFETS) Q5 and Q6. The access transistors control the connection to bit lines B & B\. In order to write a value to the memory cell, apply the desired value to the bit control lines (For writing 1, make B = 1 and B =0; for writing 0, make B = 0 and B\ =1) and assert the Word Line (Make Word line high). This operation latches the bit written in the dip-hop. For reading the content of the memory cell, assert both 8 and B\ bit lines to l and set the Word line to 1.
- The major limitations of SRAM are low capacity and high cost. Since a minimum of six transistors are required to build a single memory cell, imagine how many memory cells we can fabricate on a silicon wafer.

Dynamic RAM (DRAM)

- Dynamic RAM stores data in the form of charge. They are made up of MOS transistor gates
- The advantages of DRAM are its high density and low cost compared to SRAM
- The disadvantage is that since the information is stored as charge it gets leaked off with time and to prevent this they need to be refreshed periodically

- Special circuits called DRAM controllers are used for the refreshing operation. The refresh operation is done periodically in milliseconds interval
- Figure illustrates the typical implementation of a DRAM cell.
- The MOSFET acts as the gate for the incoming and outgoing data whereas the capacitor acts as the bit storage unit.



- Table given below summarizes the relative merits and demerits of SRAM and DRAM technology.

SRAM Cell	DRAM Cell
Made up of 6 CMOS transistors (MOSFET)	Made up of a MOSFET and a capacitor
Doesn't Require refreshing	Requires refreshing
Low capacity (Less dense)	High Capacity (Highly dense)
More expensive	Less Expensive
Fast in operation. Typical access time is 10ns	Slow in operation due to refresh requirements. Typical access time is 60ns. Write operation is faster than read operation.

Non-Volatile RAM (NVRAM)

- 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.
- NVRAM is used for the non-volatile storage of results of operations or for setting up of flags etc.
- The life span of NVRAM is expected to be around 10 years.
- DS1744 from Maxim/Dallas is an example for 32KB NVRAM

3.8Sensors and Actuators

Sensor: *A transducer device which converts energy from one form to another for any measurement or control purpose. Sensors acts as input device*

Example: Hall Effect Sensor which measures the distance between the cushion and magnet in the Smart Running shoes from adidas.

Actuator: *A form of transducer device (mechanical or electrical) which converts signals to corresponding physical action (motion). Actuator acts as an output device*

Example: Micro motor actuator which adjusts the position of the cushioning element in the Smart Running shoes from adidas.

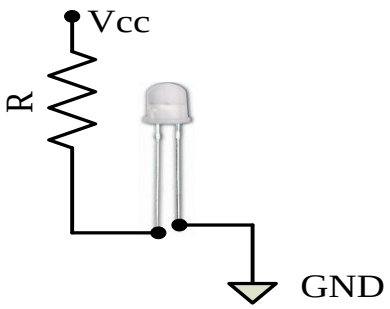
The I/O Subsystem

- The I/O subsystem of the embedded system facilitates the interaction of the embedded system with external world.
- The interaction happens through the sensors and actuators connected to the Input and output ports respectively of the embedded system.
- The sensors may not be directly interfaced to the Input ports, instead they may be interfaced through signal conditioning and translating systems like ADC, Optocouplers etc.

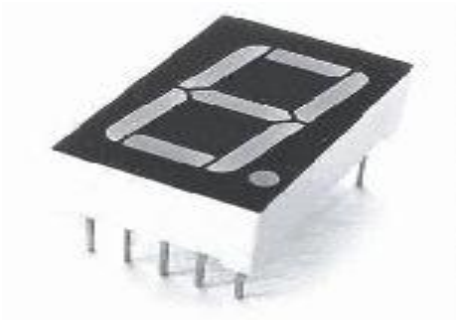
3.8.1 LED

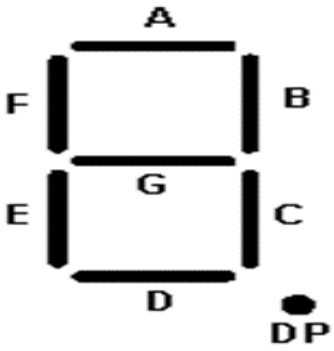
- Light Emitting Diode (LED) is an important output device for visual indication in any embedded system.
- LED can be used as an indicator for the status of various signal or situations. Typical examples are indicating the presence of power conditions like “Device ON” Battery low or “Charging of battery” for a battery operated hand held embedded devices.
- Light Emitting Diode is a pn junction diode and it contains an anode and a cathode. For proper functioning of the LED, the anode of it should be connected to +ve terminal of the supply voltage and cathode to the -ve terminal of supply voltage.
- The current flowing through the LED must be limited to a value below the maximum current that it can conduct.

- A resister is used in series between the power supply and the LED to limit the current through the LED. The ideal LED interfacing circuit is shown in Figure.



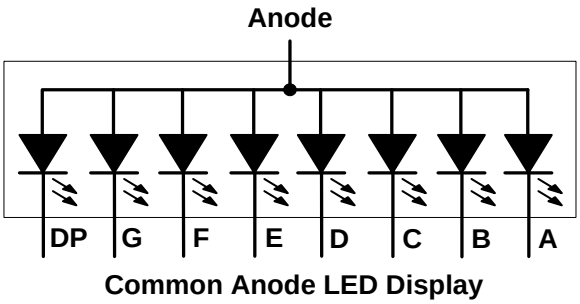
3.8.2 7 segment LED display

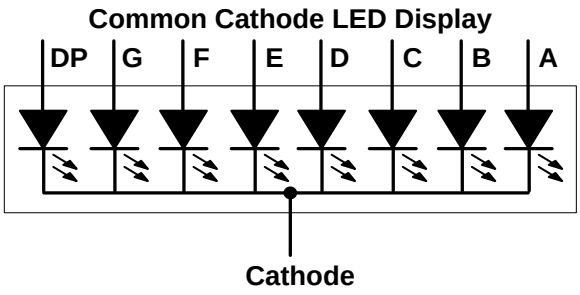




- The 7 – segment LED display is an output device for displaying alpha numeric characters.
- It contains 8 light-emitting diode (LED) segments arranged in a special form. Out of the 8 LED segments, 7 are used for displaying alpha numeric characters and 1 is used for representing decimal point.
- The LED segments are named A to G and the decimal point LED segment is named as DP.
- The LED Segments A to G and DP should be lit accordingly to display numbers and characters.
- The 7 – segment LED displays are available in two different configurations, namely; Common anode and Common cathode.
- In the Common anode configuration, the anodes of the 8 segments are connected commonly whereas in the Common cathode configuration, the 8 LED segments share a common cathode line.
- Based on the configuration of the 7 – segment LED unit, the LED segment anode or cathode is connected to the Port of the processor/controller in the order “A” segment to the Least significant port Pin and DP segment to the most significant Port Pin.
- The current flow through each of the LED segments should be limited to the maximum value supported by the LED display unit.

- The typical value for the current falls within the range of 20mA.
- The current through each segment can be limited by connecting a current limiting resistor to the anode or cathode of each segment.





3.8.3 Stepper motor, Keyboard

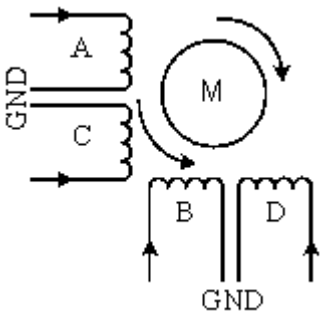
3.8.3.1 Stepper Motor

- Stepper motor is an electro mechanical device which generates discrete displacement (motion) in response to dc electrical signals.
- It differs from the normal dc motor in its operation. The dc motor produces continuous rotation on applying dc voltage whereas a stepper motor produces discrete rotation in response to the dc voltage applied to it.
- Stepper motors are widely used in industrial embedded applications, consumer electronic products and robotics control systems.
- The paper feed mechanism of a printer/fax makes use of stepper motors for its functioning.
- Based on the coil winding arrangements, a two-phase stepper motor is classified into

Unipolar

Bipolar

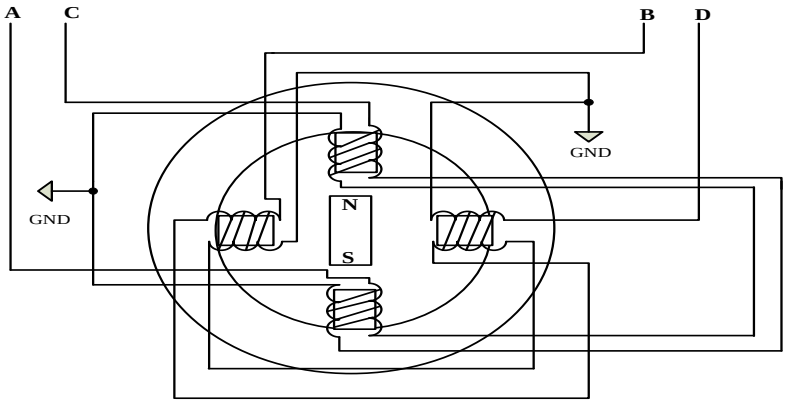
Unipolar: A unipolar stepper motor contains two windings per phase. The direction of rotation (clockwise or anticlockwise) of a stepper motor is controlled by changing the direction of current flow. Current in one direction flows through one coil and in the opposite direction flows through the other coil. It is easy to shift the direction of rotation by just switching the terminals to which the coils are connected. The figure illustrate the working of a two-



phase unipolar stepper motor.

The coils are represented as A, B, C, D. Coil A and C carry current in opposite directions for phase 1. Similarly, coil B and D carry current in opposite direction for phase 2. (only 1 of them will be carrying current at a time.

Bipolar: A bipolar stepper motor contains single winding per phase. For reversing the motor rotation, the current flow through the windings is reversed dynamically. It requires complex circuitry for current flow reversal. The stator winding details for a two phase unipolar stepper motor is shown in the below figure.



The stepper motor can be implemented in different ways by changing the sequence of activation od stator windings. The different stepping modes supported by stator windings are discussed as below.

2 Phase Unipolar Stepper Motor – Stator Winding

Full Step:

In the full step mode both the phases are energized simultaneously. The coils A, B, C and D are energized in the order.

Step	Coil A	Coil B	Coil C	Coil D
1	H	H	L	L
2	L	H	H	L
3	L	L	H	H
4	H	L	L	H

- Only one winding of a phase is energized at a time

Wave Step:

- Only one phase is energized at a time and each coils of the phase are energized alternatively. The coils A, B, C and D are energized in the order.
- Only one winding of a phase is energized at a time

Step	Coil A	Coil B	Coil C	Coil D
1	H	L	L	L
2	L	H	L	L
3	L	L	H	Lz
4	L	L	L	H

Half Step:

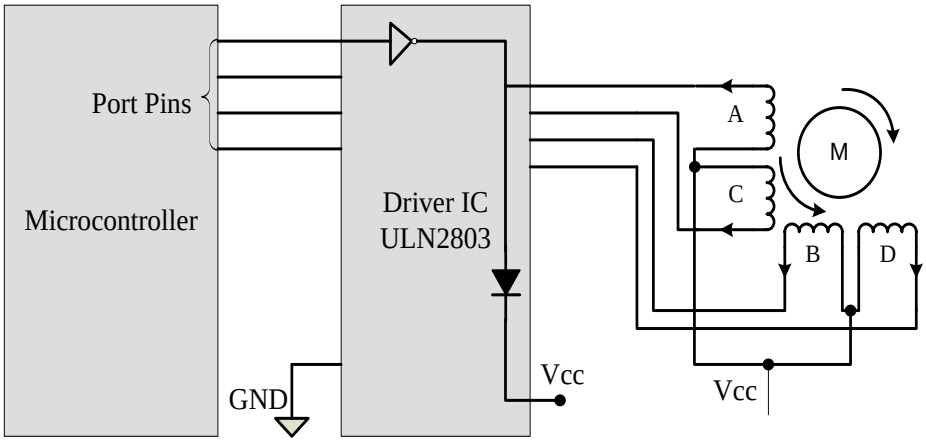
Half step uses the combination of wave and full step. It has the highest torque and stability. The coils A, B, C and D are energized in the order

Step	Coil A	Coil B	Coil C	Coil D
1	H	L	L	L
2	H	H	L	L
3	L	H	L	L
4	L	H	H	L
5	L	L	H	L
6	L	L	H	H
7	L	L	L	H
8	H	L	L	H

The rotation of the stepper motor can be reversed by reversing the order in which the coil is energized

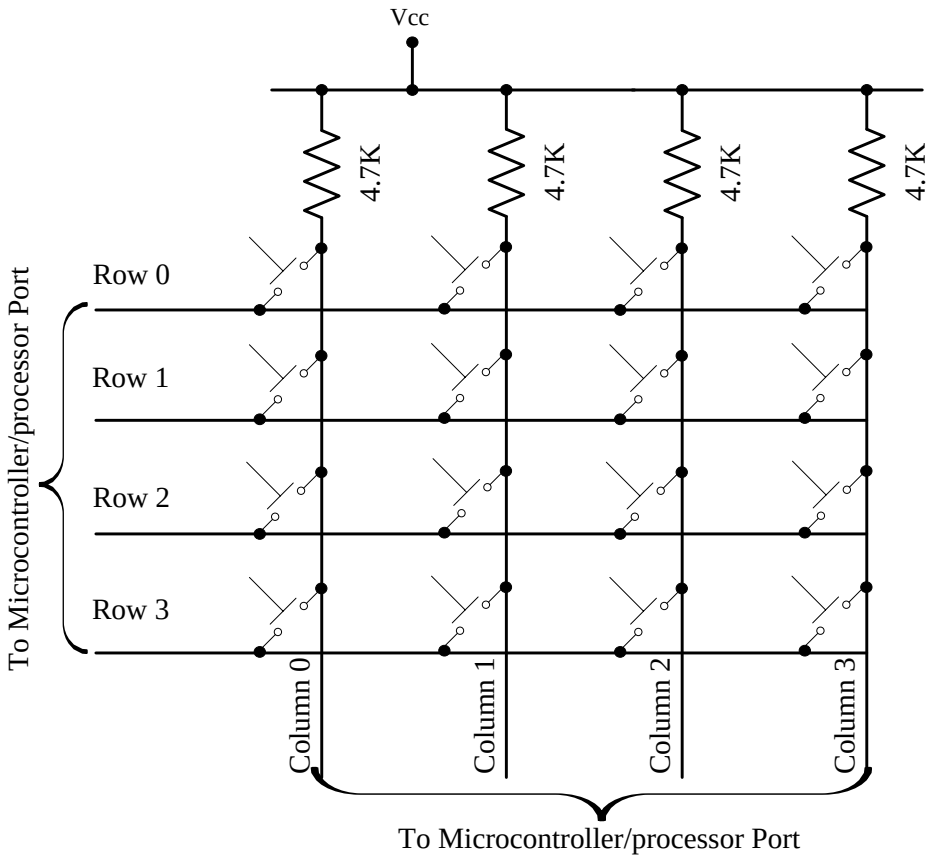
Phase Unipolar Stepper Motor – Interfacing

- Depending on the current and voltage requirements, special driving circuits are required to interface the stepper motor with microcontroller/processors.
- Stepper motor driving ICs like ULN2803 or simple transistor-based driving circuit can be used for interfacing stepper motors with processor/controller.
- The following figure illustrate the interfacing of a stepper motor through a Driver circuit connected to the port pins of a microcontroller/processor.



3.8.3.2 Keyboard

- Keyboard is an input device for user interfacing.
- If the number of keys required is very limited, push button switches can be used and they can be directly interfaced to the port pins for reading.
- Matrix keyboard is an optimum solution for handling large number of key requirements.
- Matrix keyboard greatly reduces the number of interface connections.
- Matrix keyboard connects the keys in a row column fashion
- For example, for interfacing 16 keys, in the direct interfacing technique 16 port pins are required, where as in the matrix keyboard only 4 columns and 4 rows are required for interfacing 16 keys.
- The 16 keys are arranged in a 4*4 matrix. The following figure illustrates the connection of keys in a matrix keyboard.

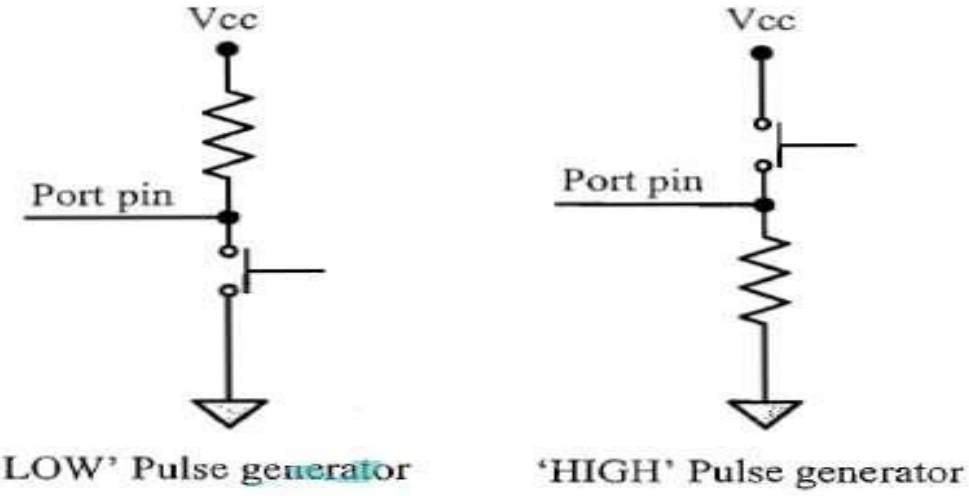


- The key press in matrix keyboard is identified with row-column scanning technique where each row of the matrix is pulled low and the columns are read.
- After reading the status of each columns corresponding to a row is pulled high and the next row is pulled low and the status of the column are read.
- When a row is pulled low and if a key connected to the row is pressed, reading the column to which the key is connected will give logic 0.
- Since the keys are mechanical devices, there is a possibility for de-bounce issues, which may give triple key press effect for a single key press.
- The techniques to prevent from this de-bouncing issues is:
    1. Hardware key de-bouncer
    2. Software key de-bouncer: on detection of a key press the key is read again after de-bounce delay. If the key press is genuine one the static key press will remain as "pressed".

3.8.4 Push button switch

- It is an input device. Push button switch comes in two configurations, namely 'Push to Make' and 'Push to Break'.

- **In the 'Push to Make' configuration**, the switch is normally in the open state and it makes a circuit contact when it is pushed or pressed.
- **In the 'Push to Break' configuration**, the switch is normally in the closed state and it breaks the circuit contact when it is pushed or pressed.
- In the embedded application push button is generally used as reset and start switch.
- The Push button is normally connected to the port pin of the host processor/controller. Depending on the way in which the push button interfaced to the controller, it can generate either a „HIGH" pulse or a „LOW" pulse.
- Figure illustrates how the push button can be used for generating "LOW" and "HIGH" pulses.



3.9 Communication Interface (onboard and external types)

- Communication interface is essential for communicating with various subsystems of the embedded system and with the external world.
- For an embedded product, the communication interface can be viewed in two different perspectives; namely;
    1. Device/board level communication interface (Onboard Communication Interface)
    2. Product level communication interface (External Communication Interface)
- Embedded product is a combination of different types of components (chips/devices) arranged on a Printed Circuit Board (PCB).
- The communication channel which interconnects the various components within an embedded product is referred as Device/board level communication interface (Onboard Communication Interface).

- Serial interfaces like I2C, SPI, UART, 1-Wire etc and Parallel bus interface are examples of “Onboard Communication Interface”.
- The “Product level communication interface” (External Communication Interface) is responsible for data transfer between the embedded system and other devices or modules.
- The external communication interface can be either wired media or wireless media and it can be a serial or parallel interface. Infrared (IR), Bluetooth (BT), Wireless LAN (Wi-Fi), Radio Frequency waves (RF), GPRS etc are examples for wireless communication interface.
- RS-232C/RS-422/RS 485, USB, Ethernet (TCP-IP), IEEE 1394 port, Parallel port, CF-II Slot, SDIO, PCMCIA etc are examples for wired interfaces.

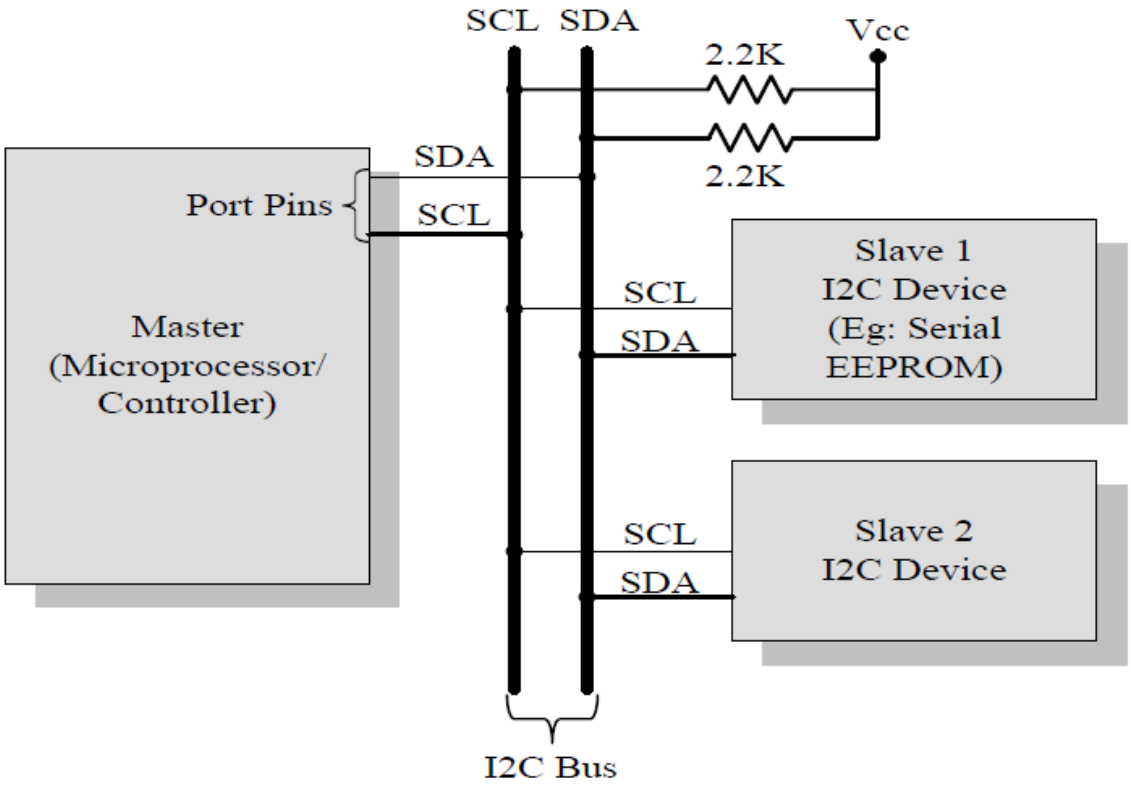
3.9.1 On-board Communication Interfaces

On-board Communication Interface refers to the different communication channels/buses for interconnecting the various integrated circuits and other peripherals within the embedded system. The following section gives an overview of the various interfaces for on-board communication.

3.9.1.1 Inter Integrated Circuit (I2C) Bus

- The Inter Integrated Circuit Bus (I2C-Pronounced „I square C„) is a synchronous bidirectional half duplex (one-directional communication at a given point of time) two wire serial interface bus.
- The concept of I2C bus was developed by „Philips semiconductors“ in the early 1980s.
- The original intention of I2C was to provide an easy way of connection between a microprocessor/micr0controller system and the peripheral chips in television sets.
- The I2C bus comprise of two bus lines. Namely; Serial Clock SCL and Serial Data SDA.
- SCL line is responsible for generating synchronization clock pulses and SDA is responsible for transmitting the serial data across devices.
- I2C bus is a shared bus system to which many number of I2C devices can be connected. Devices connected to the I2C bus can act as either 'Master“ device or “Slave“ device.

- The "Master' device is responsible for controlling the communication by initiating/terminating data transfer. Sending data and generating necessary synchronization clock pulses.
- "Slave" devices wait for the commands from the master and respond upon receiving the command, "Master" and "Slave" devices can act as either transmitter or receiver. Regardless whether a master is acting as transmitter or receiver, the synchronization clock signal is generated by the „Master" device only.
- I2C supports multi masters on the same bus. The following bus interface diagram shown in Fig. illustrates the connection of master and slave devices on the I2C bus.



I2C Bus Interfacing

- The I2C bus interface is built around an input buffer and an open drain or collector transistor. When the bus is in the idle state, the open drain/collector transistor will be in the floating state and the output lines (SDA and SCL) switch to the "High Impedance" state. For proper operation of the bus, both the bus lines should be pulled to the supply voltage (+5V for TTL family and +3.3V for CMOS family devices)

using pull-up resistors. The typical value of resistors used in pull-up is 2.2K. With pull-up resistors, the output lines of the bus in the idle state will be “HIGH“.

- The address of a I2C device is assigned by hardwiring the address lines of the device to the desired logic level. The address to various I2C devices in an embedded device is assigned and hardwired at the time of designing the embedded hardware.
- The sequence of operations for communicating with a I2C slave device is listed below:
    1. The master device pulls the clock line (SCL) of the bus to „HIGH“.
    2. The master device pulls the data line (SDA) „LOW“, when the SCL line is at logic „HIGH“ (This is the „Start“ condition for data transfer).
    3. The master device sends the address (7 bit or 10 bit wide) of the „slave“ device to which it wants to communicate, over the SDA line. Clock pulses are generated at the SCL line for synchronizing the bit reception by the slave device. The MSB of the data is always transmitted first. The data in the bus is valid during the „HIGH“ period of the clock signal.
    4. The master device sends the Read or Write bit (Bit value = 1 Read operation; Bit value 0 Write Operation) according to the requirement.
    5. The master device waits for the acknowledgement bit from the slave device whose address is sent on the bus along with the Read/Write operation command.
    6. Slave devices connected to the bus compares the address received with the address assigned to them. The slave device with the address requested by the master device responds by sending an acknowledge bit (Bit value = 1) over the SDA line.
    7. Upon receiving the acknowledge bit, the Master device sends the 8bit data to the slave device over SDA line, if the requested operation is “Write to device„. If the requested operation is „Read from device', the slave device sends data to the master over the SDA line.
    8. The master device waits for the acknowledgement bit from the device upon byte transfer complete for a write operation and sends an acknowledge bit to the Slave device for a read operation.

9. The master device terminates the transfer by pulling the SDA line „HlGI-I“ when the clock line SCL is at logic „HIGH“ (Indicating the „STOP“ condition).

- I2C bus supports three different data rates. They are: Standard mode (Data rate up to 100kbits/sec (100 kbps)), Fast mode (Data rate up to 400kbits sec (400 kbps)) High Speed mode (Data rate up to 3.4 Mbps). The first generation I2C devices were designed to support data rates only up to 100kbps. The new generation I2C devices are designed to operate at data rates up to 3.4Mbits/sec.

3.9.1.2 Serial peripheral Interface (SPI) Bus

- The Serial Peripheral Interface Bus (SPI) is a synchronous bi-directional full duplex four-wire serial interface bus. The concept of SPI was introduced by Motorola.
- SPI is a single master multi-slave system. It is possible to have a system where more than one SPI device can be master, provided the condition only one master device is active at any given point of time, is satisfied.
- SPI requires four signal lines for communication. They are:

  Master Out Slave in (MOSI): Signal line carrying the data from master to slave device. It is also known as Slave Input/Slave Data in (SI/SD1)

  Master in Slave out (MISO): Signal line carrying the data from slave to master device. It is also known as Slave Output (SO/SDO)

  Serial Clock (SCLK): Signal line carrying the clock signals

  Slave Select (SS): Signal line for slave device select. It is an active low signal
- The bus interface diagram shown in Figure illustrates the connection of master and slave devices on the SPI bus.

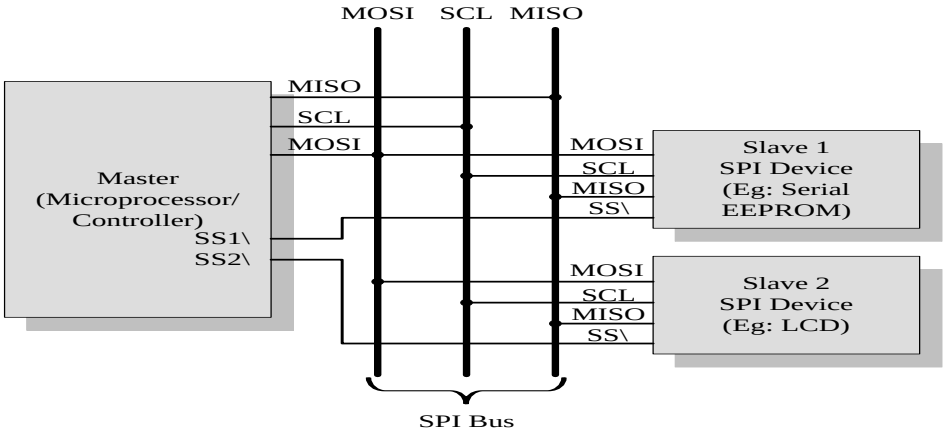


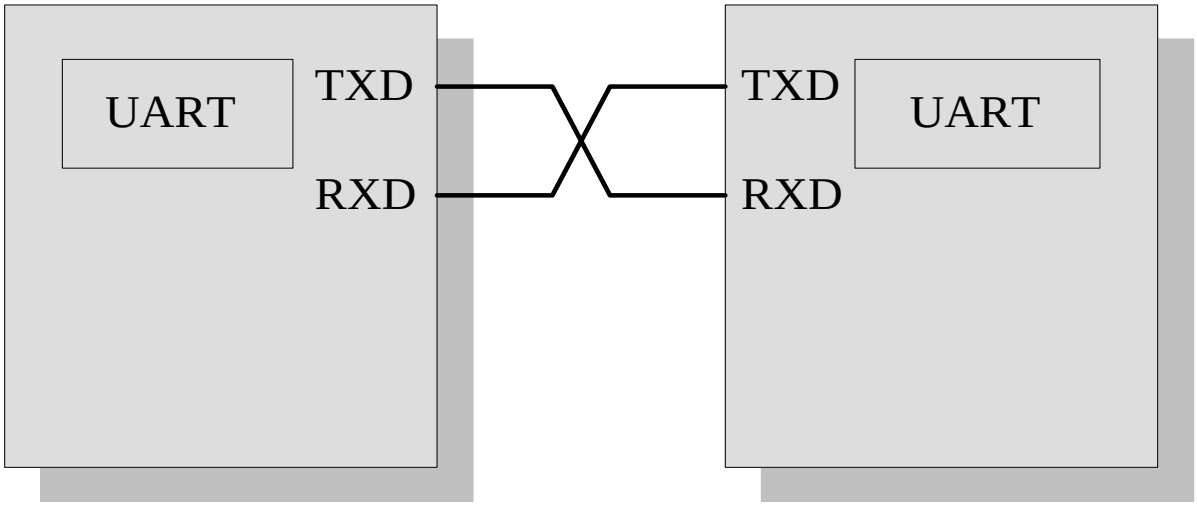
Fig: SPI bus Interfacing

- The master device is responsible for generating the clock signal. It selects the required slave device by asserting the corresponding slave device‟s slave select signal „LOW‟. The data out line (MISO) of all the slave devices when not selected floats at high impedance state.
- SP1 works on the principle of „Shift Register‟. The master and slave devices contain a special shift register for the data to transmit or receive. The size of the shift register is device dependent. Normally it is a multiple of 8.
- During transmission from the master to slave, the data in the master‟s shift register is shifted out to the M0SI pin and it enters the shift register of the slave device through the M0SI pin of the slave device. At the same time the shifted out bit from the slave device's shift register enters the shift register of the master device through MISO pin. In summary, the shift registers of "master‟ and "slave‟ devices form a circular buffer. For some devices, the decision on whether the LS/MS bit of data needs to be sent out first is configurable through configuration register (e.g. LSBF bit of the SP1 control register for Motorola‟s 68HC12 controller).
- When compared to 12C. SPI bus is most suitable for applications requiring transfer of data in "streams". The only limitation is SPI doesn't support an acknowledgement mechanism.

3.9.1.3 Universal Asynchronous Receiver Transmitter (UART)

- Universal Asynchronous Receiver Transmitter (UART) based data transmission is an asynchronous form of serial data transmission.
- UART based serial data transmission doesn't require a clock signal to synchronize the transmitting end and receiving end for transmission. Instead it relies upon the pre-defined agreement between the transmitting device and receiving device. The serial communication settings (Baud rate, number of bits per byte. parity, number of start bits and stop bit and flow control) for both transmitter and receiver should be set as identical.
- The start and stop of communication is indicated through inserting special bits in the data stream. While sending a byte of data, a start bit is added first and a stop bit is added at the end of the bit stream. The least significant bit of the data byte follows the "start‟ bit.

- The “start“ bit informs the receiver that a data byte is about to arrive. The receiver device starts polling its received line“ as per the baud rate settings.
- If parity is enabled for communication, the UART of the transmitting device adds a parity bit.
- The UART of the receiving device calculates the parity of the bits received and compares it with the received parity bit for error checking. The UART of the receiving device discards the “Start“, “Stop“ and “Parity' bit from the received serial bit data to a word.
- For proper communication, the “Transmit line“ of the sending device should be connected to the “Receive line“ of the receiving device. Figure illustrates the same.



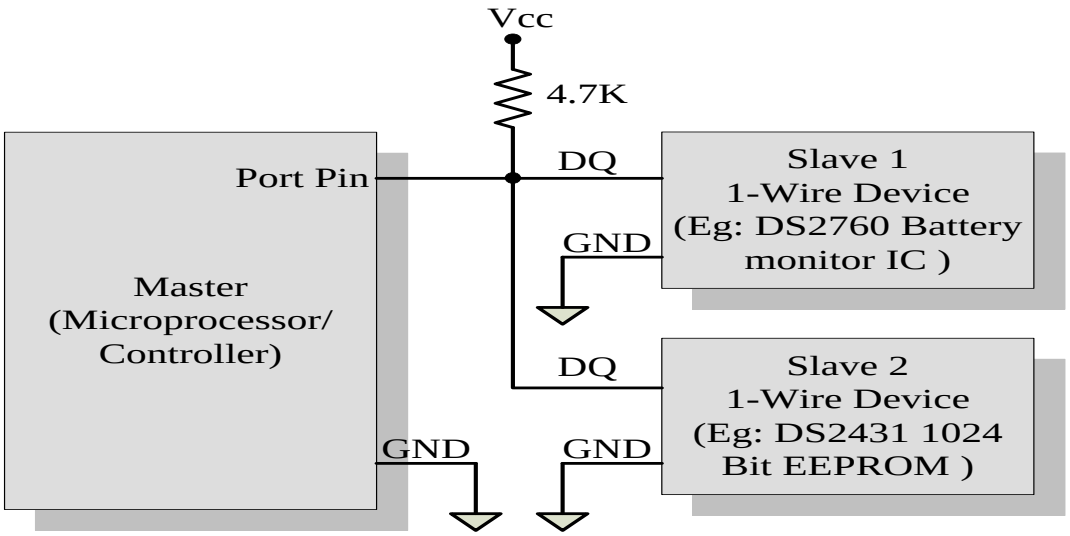
TXD: Transmitter Line
RXD: Receiver Line

- In addition to the serial data transmission function, UART provides hardware handshake signal support for controlling the serial data now.

3.9.1.3 1–Wire Interface

- 1-wire interface is an asynchronous half-duplex communication protocol developed by Maxim Dallas Semiconductor. It is also known as Dallas 1-Wire® protocol.
- It makes use of only a single signal line (wire) called DQ for communication and follows the master-slave communication model.
- One of the key features of l-wire bus is that it allows power to be sent along the signal wire as well. The 12C slave devices incorporate internal capacitor (typically of the order of 800 pF) to power the device from the signal line. The l-wire interface supports a Single master and one or more slave devices on the bus.

- The bus interface diagram shown in Figure illustrates the connection of master and slave devices on the l-wire bus.



- Every l-wire device contains a globally unique 64bit identification number stored within it. The unique identification number can be used for addressing individual devices present on the bus in case there are multiple slave devices connected to the 1-wire bus.
- The identifier has three parts: an 8bit family code, a 48bit serial number and anS bit CRC computed from the first 56 bits.
- The sequence of operation for communicating with a 1-wire slave device is listed below:

  1. The master device sends a „Reset“ pulse on the l-wire bus.

  2. The slave device(s) present on the bus respond with 3 „Presence“ pulse.

  3. The master device sends a ROM command (Net Address Command followed by the 64bit address of the device). This addresses the slave device(s) to which it wants to initiate a communication.

  4. The master device sends a read/write function command to read/write the internal memory or register of the slave device.

  5. The master initiates a Read data/Write data from the device or to the device

- All communication over the l-wire bus is master initiated. The communication over the l-wire bus 3, divided into timeslots of 60 microseconds. The „Reset“ pulse occupies 8 time slots.
- For starting a communication, the master asserts the reset pulse by pulling the 1-wire bus “LOW“ for at least 8 time slots “slave“ device is present on the bus and is ready

for communication it should respond to the master with a “Presence“ pulse, within 60us of the release of the “Reset“ pulse by the master.

- The slave device(s) responds with a “Presence“ pulse by pulling the l-wire bus “LOW“ for a minimum of 1 time slot (60448).
- For writing a bit value of 1 on the l-wire bus, the bus master pulls the bus for l to l5 bus and then releases the bus for the rest of the time slot. A bit value of „0“ is written on the bus by master pulling the bus for a minimum of 1 time slot (60us) and a maximum of 2 time slots.
- To Read a bit from the slave device, the master pulls the bus “LOW“ for l to 15us. If the slave wants to send a bit value “1“ in response to the read request from the master, it simply releases the bus for the rest of the time slot. If the slave wants to send a bit value “0“, it pulls the bus “LOW“ for the rest of the time slot.

Parallel Interface

- The on-board parallel interface is normal used for communicating with peripheral devices which are memory mapped to the host of the system.
- The host processor/controller of the embedded system contains a parallel bus and the device which supports parallel bus can directly connect to this bus system.
- The communication through the parallel bus is controlled by the control signal interface between the device and the host.
- The “Control Signals“ for communication includes “Read/ Write“ signal and device select signal. The device normally contains a device select line and the device becomes active only when this line is asserted by the host processor.
- The direction of data transfer (Host to Device or Device to Host) can be controlled through the control signal lines for “Read“ and “Write“. Only the host processor has control over the “Read“ and “Write“ control signals.
- The device is normally memory mapped to the host processor and a range of address is assigned to it. An address decoder circuit is used for generating the chip select signal for the device. When the address selected by the processor is within the range assigned for the device, the decoder circuit activates the chip select line and thereby the device becomes active.
- The processor then can read or write from or to the device by asserting the corresponding control line (RD and WR respectively). Strict timing characteristics are followed for parallel communication. As mentioned earlier, parallel communication is

host processor initiated. If a (device wants to initiate the communication, it can inform the same to the processor through interrupts. For this, the interrupt line of the device is connected to the interrupt line of the processor and the core, responding interrupt is enabled in the host processor. The width of the parallel interbank is determined by the data bus width of the host processor. It can be 4bit, 8bit, 16bit, 32bit or 64bit etc. The bus width supported by the device should be same as that of the host processor. The bus interface diagram shown in Figure illustrates the interfacing of devices through parallel interface.

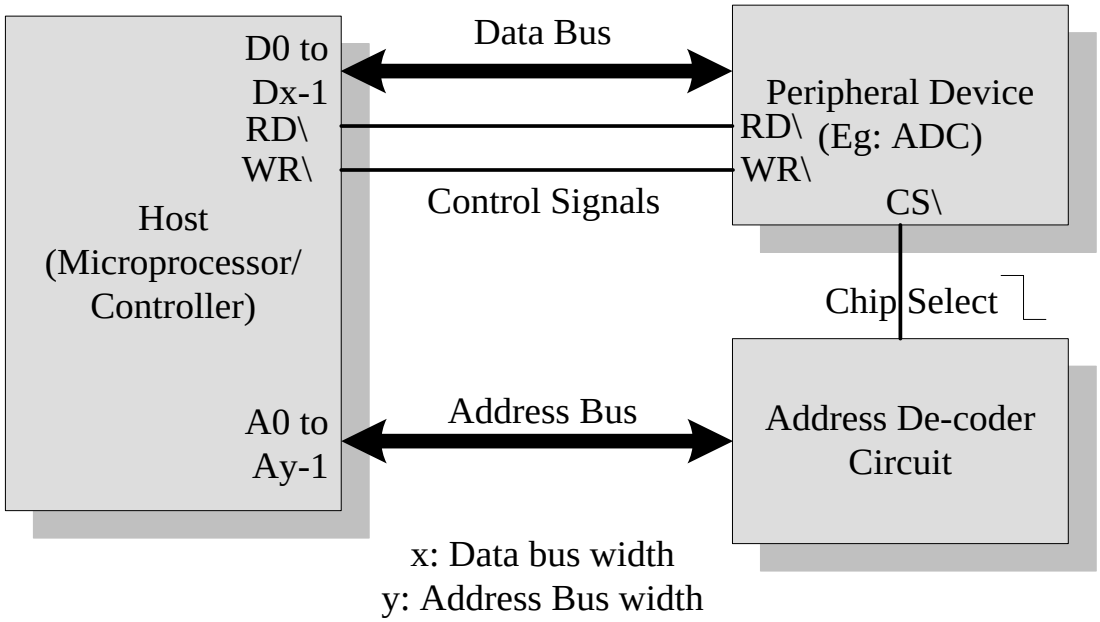


Fig: Interfacing of devices through parallel interface

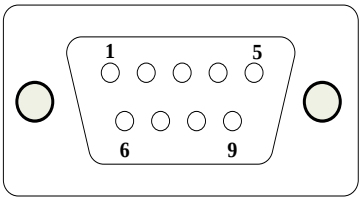
3.9.2 External Communication Interfaces

The External Communication Interface refers to the different communication channels/buses used by the embedded system to communicate with the external world. The following section gives an overview of the various interfaces for external communication.

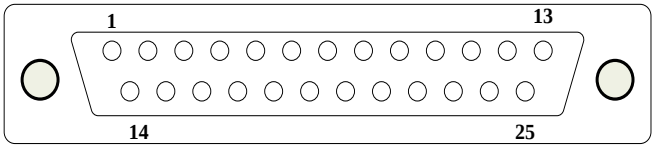
3.9.2.1 RS-232 & RS 485:

- RS-232 C (Recommended Standard number 232, revision C from the Electronic Industry Association) is a legacy, full duplex, wired, asynchronous serial communication interface.
- The RS-232 interface is developed by the Electronics Industries Association (EIA) during the early 1960s. RS-232 extends the UART communication signals for external data communication.

- UART uses the standard TTL/CMOS logic (Logic „High‟ corresponds to bit value 1 and Logic “Low” corresponds to hit value 0) for bit transmission whereas RS-232 follows the EIA standard for bit transmission.
- As per the EIA standard, a logic „0‟ is represented with voltage between +3 and +25V and a logic “l‟ is represented with voltage between -3 and -25V. In EIA standard, logic “0‟ is known as “Space‟ and logic “1‟ as “Mark‟. The RS-232 interface defines various handshaking and control signals for communication apart from the “Transmit‟ and “Receive' signal lines for data communication.
- RS-232 supports two different types of connectors, namely; DB-9: 9-Pin connector and DB-25: 25-Pin connector. Figure illustrates the connector details for DB-9 and DB-25.



DB-9



DB-25

- The pin details for the DB-9 connectors are explained in the following table:

Pin Name	Pin No: (For DB-9 Connector)	Description
TXD	3	Transmit Pin. Used for Transmitting Serial Data
RXD	2	Receive Pin. Used for Receiving serial Data
RTS	7	Request to send.
CTS	8	Clear To Send
DSR	6	Data Set ready
GND	5	Signal Ground
DCD	1	Data Carrier Detect

DTR	4	Data Terminal Ready
RI	9	Ring Indicator

- RS-232 is a point-to-point communication interface and the devices involved in RS-232 communication are called “Data Terminal Equipment (DTE)” and “Data Communication Equipment (DCE)”.
- If no data flow control is required. Only TXD and RXD signal lines and ground line (GND) are required for data transmission and reception. The RXD pin of DCE should be connected to the TXD pin of DTE and vice versa for proper data transmission.
- If hardware data how control is required for serial transmission, various control signal lines of the RS-232 connection are used appropriately. The control signals are implemented mainly for modem communication and some of them may not be relevant for other type of devices.
- The Request to Send (RTS) and Clear to Send (CTS) signals co-ordinate the communication between DTE and DCE. Whenever the DTE has a data to send, it activates the RTS line and if the DCE is ready to accept the data, it activates the CTS line.
- The Data Terminal Ready (DTR) signal is activated by DTE when it is ready to accept data.
- The Data Set Ready (DSR) is activated by DCE when it is ready for establishing a communication link. DTR should be in the activated state before the activation of DSR.
- The Data Carrier Detect (DCD) control signal is used by the DCE to indicate the DTE that a good signal is being received.
- Ring Indicator (RI) is a modem specific signal line for indicating an incoming call on the telephone line.
- The 25 pin DB connector contains two sets of signal lines for transmit, receive and control lines. Nowadays DB-25 connector is obsolete and most of the desktop systems are available with DB-9 connectors only.
- As per the EIA standard RS-232 C supports baud rates up to 20Kbps (Upper limit 19.2 Kbps) The commonly used baud rates by devices are 300bps, l200bps, 2400bps, 9600bps, 11.52Kbps and 19.2Kbps. 9600 is the popular baud rate setting used for PC

CTS 8 Clear To Send DSR 6 Data Set ready GND 5 Signal Ground DCD 1 Data Carrier Detect DTR 4 Data Terminal Ready RI 9 Ring Indicator communication. The maximum operating distance supported by RS-232 is 50 feet at the highest supported baud rate.

- Embedded devices contain a UART for serial communication and they generate signal levels conforming to TTL CMOS logic.
- A level translator lC like MAX 232 from Maxim Dallas semiconductor is used for converting the signal lines from the UART to RS-232 signal lines for communication.
- On the receiving side the received data is converted back to digital logic level by a converter IC. Convener chips contain converters for both transmitter and receiver.
- Though RS-232 was the most popular communication interface during the olden days, the advent of other communication techniques like Bluetooth, USB, Fire wire, etc. are pushing down RS-232 from the scenes. Still RS-232 is popular in certain legacy industrial applications.
- RS-232 supports only point-to-point communication and not suitable for multi-drop communication. It uses single ended data transfer technique for signal transmission and thereby more susceptible to noise and it greatly reduces the operating distance.
- RS-422 is another serial interface standard from EIA for differential data communication. It supports data rates up to 100 Kbps and distance up to 400 ft. The same RS-232 connector is used at the device end and an RS-232 to RS-422 converter is plugged in the transmission line. At the receiver end the conversion from RS-422 to RS-232 is performed. RS-422 supports multi-drop communication with one transmitter device and receiver devices up to 10.
- RS-485 is the enhanced version of RS-422 and it supports multi-drop communication with up to 32 transmitting devices (drivers) and 32 receiving devices on the bus. The communication between devices in the bus uses the „addressing‟ mechanism to identify slave devices.

3.9.2.2 Universal Serial Bus (USB):

- Universal Serial Bus (USB) is a wired high-speed serial bus for data communication. The first version of USB (USB1.0) was released in 1995 and was created by the USB core group members consisting of Intel, Microsoft, IBM, Compaq, Digital and Northern Telecom.

- The USB communication system follows a star topology with a USB host at the center and one or more USB peripheral devices/USB hosts connected to it.
- A USB host can support connections up to 127, including slave peripheral devices and other USB hosts.
- Figure illustrates the star topology for USB device connection.

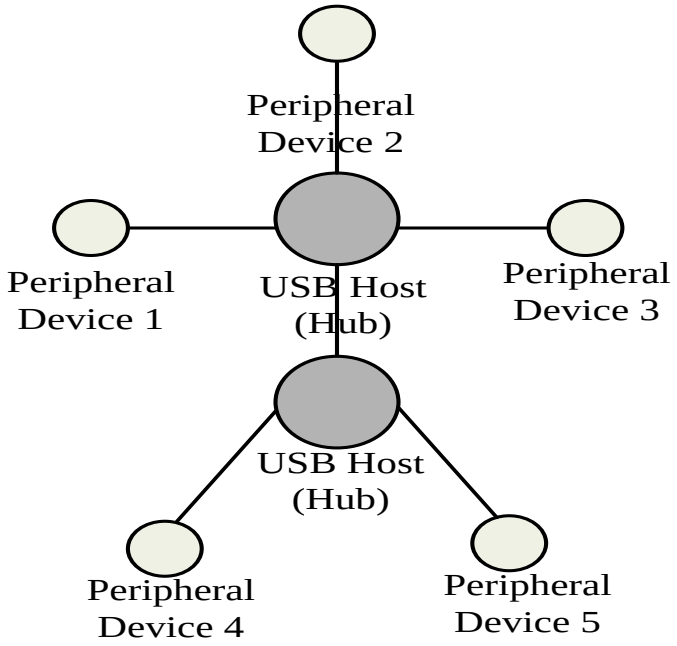


Fig: USB Device Connection Topology

- USB transmits data in packet format. Each data packet has a standard format. The USB communication is a host initiated one. The USB host contains a host controller which is responsible for controlling the data communication, including establishing connectivity with USB slave devices, packetizing and formatting the data.
- There are different standards for implementing the USB Host Control interface; namely Open Host Control Interface (OHCI) and Universal Host Control Interface (UHCI).
- USB uses differential signals for data transmission. It improves the noise immunity. USB interface has the ability to supply power to the connecting devices. Two connection lines (Ground and Power) of the USB interface are dedicated for carrying power. It can supply power up to 500 mA at 5 V. It is sufficient to operate low power devices. Mini and Micro USB connectors are available for small form factor devices like portable media players.
- The pin details for connectors are listed below:

Pin No:	Pin Name	Description
1	V_{BUS}	Carries power (5V)
2	D-	Differential data carrier line
3	D+	Differential data carrier line
4	GND	Ground signal line

- Each USB device contains a Product ID (PID) and a Vendor ID (VID). The PID and VID are embedded into the USB chip by the USB device manufacturer. The VID for a device is supplied by the USB standards forum. PID and VID are essential for loading the drivers corresponding to a USB device for communication.
- USB supports four different types of data transfers, namely; Control, Bulk, Isochronous and Interrupt.
- Control transfer is used by USB system software to query, configure and issue commands to the USB device. 8qu transfer is used for sending a block of data to a device. Bulk transfer supports error checking and correction. Transferring data to a printer is an example for bulk transfer.
- Isochronous data transfer is used for real-time data communication. In Isochronous transfer, data is transmitted as streams in real-time. Isochronous transfer doesn't support error checking and re-transmission of data in case of any transmission loss. All streaming devices like audio devices and medical equipment for data collection make use of the isochronous transfer.
- Interrupt transfer is used for transferring small amount of data. Interrupt transfer mechanism makes use of polling technique to see whether the USB device has any data to send.
- The frequency of polling is determined by the USB device and it varies from 1 to 255 milliseconds. Devices like Mouse and Keyboard, which transmits fewer amounts of data, uses interrupt transfer.
- Presently USB supports four different data rates namely; Low Speed (1.5Mbps), Full Speed (l2Mbps), High Speed (480Mbps) and Super Speed (4.8Gbps). The Low Speed and Full Speed specifications are defined by USB 1.0 and the High-Speed specification is defined by USB 2.0. USB 3.0 defines the specifications for Super Speed. USB 3.0 is expected to be in action by year 2009.

3.9.2.3 IEEE 1394 (Fire wire):

- IEEE 1394 is a wired, isochronous high speed serial communication bus. It is also known as High Performance Serial Bus (HPSB). The research on 1394 was started by Apple Inc. in 1985 and the standard for this was coined by IEEE.
- 1394 supports peer-to-peer connection and point-to-multipoint communication allowing 63 devices to be connected on the bus in a tree topology. 1394 is a wired serial interface and it can support a cable length of up to 15 feet for interconnection.
- The 1394 standard has evolved a lot from the first version IEEE 1394-1995 released in 1995 to the recent version IEEE 1394-2008 released in June 2008. The 1394 standard supports a data rate of 400 to 3200Mbits/second.
- The IEEE 1394 uses differential data transfer and the interface cable supports 3 types of connectors, namely; 4-pin connector, 6-pin connector (alpha connector) and 9 pin connector (beta connector).
- The table given below illustrates the pin details for 4, 6 and 9 pin connectors.

Pin Name	Pin No: (4-Pin Connector)	Pin No: (6-Pin Connector)	Pin No: (9-Pin Connector)	Description
Power		**1**	**8**	**Unregulated DC supply. 24 to 30V**
Signal Ground		**2**	**6**	**Ground connection**
TPB-	**1**	**3**	**1**	**Differential Signal line for Signal Line B**
TPB+	**2**	**4**	**2**	**Differential Signal line for Signal Line B**
TPA-	**3**	**5**	**3**	**Differential Signal line for Signal Line A**
TPA+	**4**	**6**	**4**	**Differential Signal line for Signal Line A**
TPA(S)			**5**	**Shield for the differential signal line A. Normally grounded**
TPB(S)			**9**	**Shield for the differential signal line B. Normally grounded**
NC			**7**	**No connection**

- There are two differential data transfer lines A and B per connector. In a 1394 cable, normally the differential lines of A are connected to B (TPA+ to TPB+ and TPA-to TPB~) and vice versa.
- 1394 is a popular communication interface for connecting embedded devices like Digital Camera, Camcorder, and Scanners to desktop computers for data transfer and storage.
- Unlike USB interface (Except USB OTG), IEEE 1394 doesn't require a host for communicating between devices. For example, you can directly connect a scanner with a printer for printing.

3.9.2.4 IrDA (Infrared)

- Infrared (IrDA) is a serial, half duplex, line of sight based wireless technology for data communication between devices. It is in use from the olden days of communication and you may be very familiar with it. The remote control of your TV, VCD player, etc. works on infrared data communication principle.
- Infrared communication technique uses infrared waves of the electromagnetic spectrum for transmitting the data.
- IrDA supports point-point and point-to-multipoint communication, provided all devices involved in the communication are within the line of sight.
- The typical communication range for IrDA lies in the range 10 cm to 1 m. The range can be increased increasing the transmitting power of the IR device. IR supports data rates ranging from 9600bits/second to 16Mbps.
- Depending on the speed of data transmission IR is classified into Serial IR (SIR), Median1 IR (MIR), Fast IR (FIR), Very Fast IR (VFIR) and Ultra-Fast IR (UFIR).
- SIR supports transmissio1 rates ranging from 9600bps to 115.2kbps. MIR supports data rates of 0.576Mbps and 1.152Mbps. FIR supports data rates up to 4Mbps. VFIR is designed to support high data rates up to 16Mbps. The UFIR specs are under development and it is targeting a data rate up to l00Mbps.
- IrDA communication involves a transmitter unit for transmitting the data over IR and a receiver for receiving the data.
- Infrared Light Emitting Diode (LED) is the IR source for transmitter and at the receiving end a photodiode acts as the receiver. Both transmitter and receiver unit will be present in each device supporting IrDA communication for bidirectional data transfer. Such IR units are known as "Transceiver".

- Certain devices like a TV remote control always require unidirectional communication and so they contain either the transmitter or receiver unit (The remote control unit contains the transmit. per unit and TV contains the receiver unit).
- The IrDA control protocol contains implementations for Physical Layer (PHY), Media Access Control (MAC) and Logical Link Control (LLC). The Physical Layer defines the physical characteristics of communication like range, data rates, power, etc.

3.9.2.5 Bluetooth (BT)

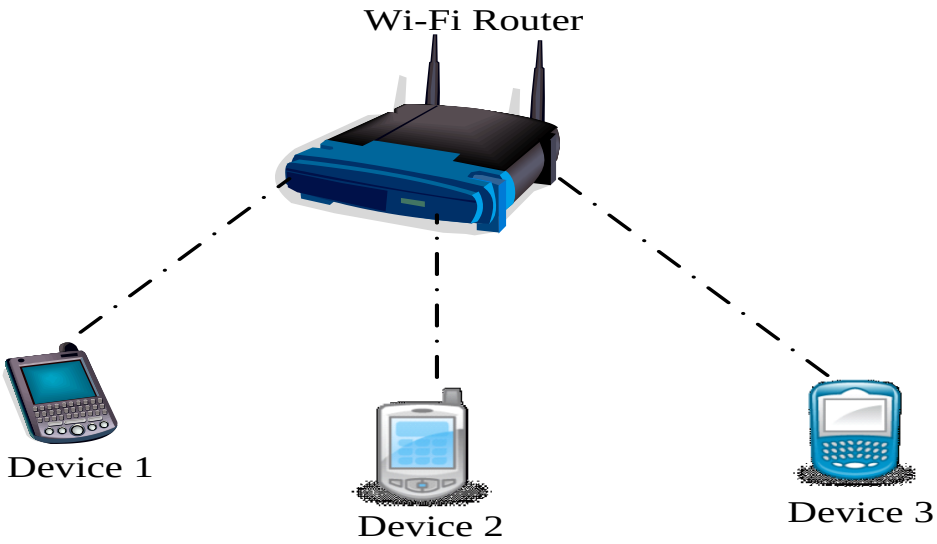
- Bluetooth is a low cost, low power, short range wireless technology for data and voice communication. Bluetooth was first proposed by "Ericsson" in 1994.
- Bluetooth operates at 2.4GHz of the Radio Frequency spectrum and uses the Frequency Hopping Spread Spectrum (FHSS) technique for communication.
- It supports a data rate of up to 1Mbps and a range of approximately 30 feet for data communication.
- Bluetooth communication also has two essential parts; a physical link part and a protocol part. The physical link is responsible for the physical transmission of data between devices supporting high Bluetooth communication and protocol part is responsible for defining the rules of communication. The physical link works on the wireless principle making use of RF waves for communication. Bluetooth enabled devices essentially contain a Bluetooth wireless radio for the transmission and reception of data. The rules governing the Bluetooth communication is implemented in the "Bluetooth protocol stack". The Bluetooth communication IC holds the stack. Each Bluetooth device will have a 48 bit unique identification number. Bluetooth communication follows packet used data.
- Bluetooth supports point-to-point (device to device) and point-to-multipoint (device to multiple device broadcasting) wireless communication. The point-to-point communication follows the master slave relationship.
- A Bluetooth device can function as either master or slave. When a network is formed with one Bluetooth device as master and more than one device as slaves, it is called a Piconet/ A Pico net supports a maximum of seven slave devices.
- Bluetooth is the favorite choice for short range data communication in handheld embedded devices. Bluetooth technology is very popular among cell phone users as

they are the easiest communication channel for transferring ringtones, music files, pictures, media files, etc. between neighboring Bluetooth enabled phones.

- The Bluetooth standard specifies the minimum requirements that a Bluetooth device must support for a specific usage scenario.
- The Generic Access Profile (GAP) defines the requirements for detecting a Bluetooth device and establishing a connection with it. All other specific usage profiles are based on GAP.
- Serial Port Profile (SPP) for serial data communication, File Transfer Profile (FTP) for file transfer between devices, Human Interface Device (HID) for supporting human interface devices like keyboard and mouse are examples for Bluetooth profiles.
- The specifications for Bluetooth communication is defined and licensed by the standards body "Bluetooth Special interest Group (SIG)".

3.9.2.6 WI-FI

- Wi-Fi or Wireless Fidelity is the popular wireless communication technique for networked communication of devices.
- Wi-Fi follows the IEEE 802.11 standard. Wi-Fi 1s intended for network communication and it supports Internet Protocol (IP) based communication it is essential to have device identities in a multipoint communication to address specific devices for data communication.
- In a 1P based communication each device is identified by an IP address, which is unique to each device on the network. Wi-Fi based communications require an intermediate agent called Wi-Fi router/Wireless Access point to manage the communications.
- The Wi-Fi router is responsible for restricting the access to a network, assigning IP address to devices on the network, routing data packets to the intended devices on the network.
- Wi-Fi enabled devices contain a wireless adaptor for transmitting and receiving data in the form of radio signals through an antenna. The hardware part of it is known as Wi-Fi Radio.
- Wi-Fi operates at 2.4GHz or 5GHz of radio spectrum and they co-exist with other ISM band devices like Bluetooth. Figure illustrates the typical interfacing of devices in a Wi- Fi network.



- For communicating with devices over a Wi-Fi network, the device when its Wi-Fi radio is turned ON, searches the available Wi-Fi network in its vicinity and lists out the Service Set Identifier (SSID) of the available networks.
- If the network is security enabled, a password may be required to connect to a particular SSID. Wi-Fi employs different security mechanisms like Wired Equivalency Privacy (WEP) Wireless Protected Access (WPA), etc. for securing the data communication.
- Wi-Fi supports data rates ranging from 1Mbps to 150Mbps.
- Wi-Fi offers a range of 100 to 300 feet.

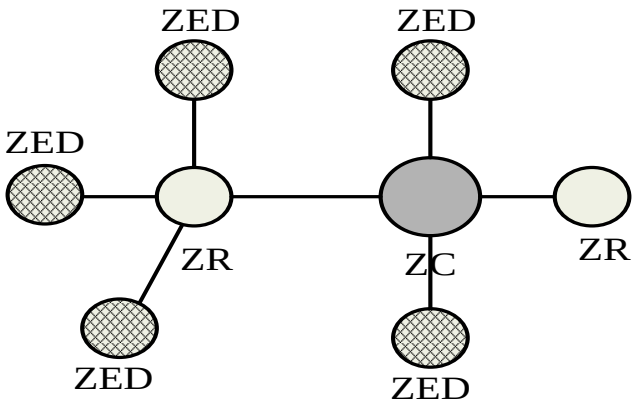
3.9.2.7 ZigBee:

- ZigBee is a low power, low cost, wireless network communication protocol based on the IEEE 802.15.4-2006 standard.
- ZigBee is targeted for low power, low data rate and secure applications for wireless Area Networking (W PAN).
- The ZigBee specifications support a robust mesh network containing multiple nodes. This networking strategy makes the network reliable by permitting messages to travel through a number of different paths to get from one node to another.
- ZigBee operates worldwide at the unlicensed bands of Radio spectrum, mainly at 2.400 to 2.484 GHZ, 902 to 928 MHz and 868.0 to 868.6MHz.
- ZigBee Supports an operating distance of up to 109 meters and a data rate of 20 to 250Kbps.
- In the ZigBee terminology, each ZigBee device falls under any one of the following ZigBee device category.

1.ZigBee Coordinator (ZC)/Network Coordinator: The ZigBee coordinator acts as the root of the ZigBee network. The ZC is responsible for initiating the ZigBee network and it has the capability to Store information about the network.

2.ZigBee Router (ZR)/Full function Device (FFD): Responsible for passing information from device to another device or to another ZR.

3.ZigBee End Device (ZED)/Reduced Function Device (RFD): End device containing ZigBee functionality for data communication. It can talk only with a ZR or ZC and Doesn't have the capability to act as a mediator for transferring data from one device to another.



3.10 Embedded firmware

Embedded firmware refers to the control algorithm (Program instructions) and or the configuration settings that an embedded system developer dumps into the code (Program) memory of the embedded system. It is an un-avoidable part of an embedded system. There are various methods available for developing the embedded firmware. They are listed below.

1. Write the program in high level languages like Embedded C/C++ using an Integrated Development Environment (The IDE will contain an editor, compiler, linker, debugger, simulator, etc. IDES are different for different family of processors/controllers.

   Example: Keil micro vision3 IDE is used for all family members of 8051 microcontroller, since it contains the generic 8051 compiler C51).
2. Write the program in Assembly language using the instructions supported by your application's target processor controller.

3.11 Other system components

3.11.1 Reset Circuit

- The Reset circuit is essential to ensure that the device is not operating at a voltage level where the device is not guaranteed to operate, during system power ON.

- The Reset signal brings the internal registers and the different hardware systems of the processor/controller to a known state and starts the firmware execution from the reset vector (Normally from vector address 0x0000 for conventional processors/controllers.
- The reset vector can be relocated to an address for processors/controllers supporting bootloader.
- The reset signal can be either active high (The processor undergoes reset when the reset pin of the processor is at logic high) or active low (The processor undergoes reset when the reset pin of the processor is at logic low).
- Figure illustrates a Resistor capacitor based passive reset circuit for active high and low configuration:

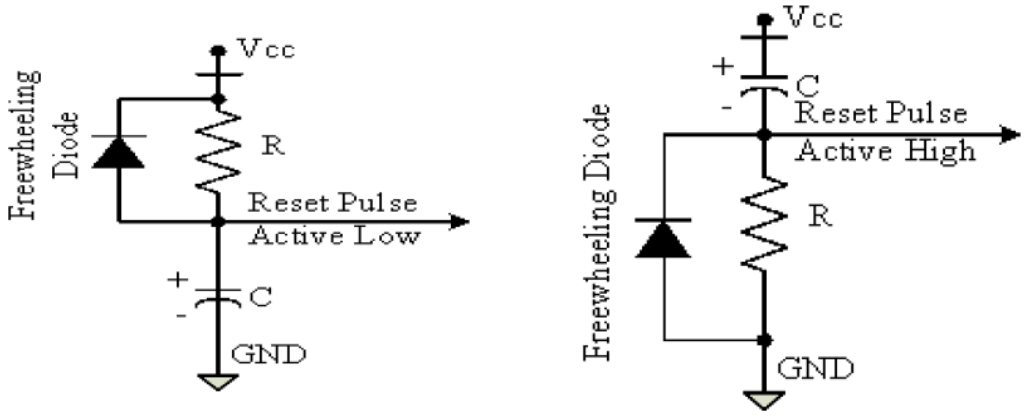
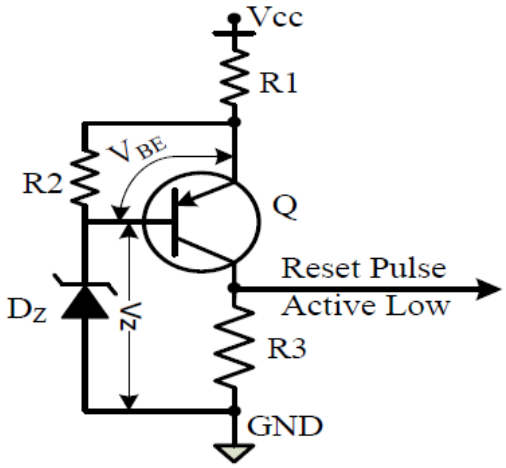


Fig:RC based Reset Circuit

3.11.2 Brown-out Protection Circuit

- Brown-out protection circuit prevents the processor/controller from unexpected program execution behavior when the supply voltage to the processor/controller falls below a specified voltage.
- The processor behavior may not be predictable if the supply voltage falls below the recommended operating voltage. It may lead to situations like data corruption.
- A brown-out protection circuit holds the processor/controller in reset state, when the operating voltage falls below the threshold, until it rises above the threshold voltage.
- Certain processors/controllers support built in brown-out protection circuit which monitors the supply voltage internally.
- If the processor/controller doesn‟t integrate a built-in brown-out protection circuit, the same can be implemented using external passive circuits or supervisor ICs.

- Figure illustrates a brown-out circuit implementation using Zener diode and transistor for processor/controller with active low Reset Logic.



Brown-out Protection circuit using active low output

The Zener diode Dz and transistor Q forms the heart of this circuit. The transistor conducts always when the supply voltage Vcc is greater than that of the sum of VBE and Vz (Zener voltage). The transistor stops conducting when the supply voltage falls below the sum of Var. and VzSelect the Zener diode with required voltage for setting the low threshold value for Vcc. The values of R1, R2, and R3 can be selected based on the electrical characteristics (Absolute maximum current and voltage ratings) of the transistor in use. Microprocessor Supervisor like D81232 from Maxim Dallas (www.maximigcom) also provides Brown-out protection.

3.11.3 Oscillator Unit

- A microprocessor/microcontroller is a digital device made up of digital combinational and sequential circuits.
- The instruction execution of a microprocessor/controller occurs in sync with a clock signal.
- The oscillator unit of the embedded system is responsible for generating the precise clock for the processor.
- Certain processors/controllers integrate a built-in oscillator unit and simply require an external ceramic resonator/quartz crystal for producing the necessary clock signals.

- Certain processor/controller chips may not contain a built-in oscillator unit and require the clock pulses to be generated and supplied externally. Quartz crystal Oscillators are example for clock pulse generating devices.
- Figure illustrates the usage of quartz crystal/ceramic resonator and external oscillator chip for clock generation.

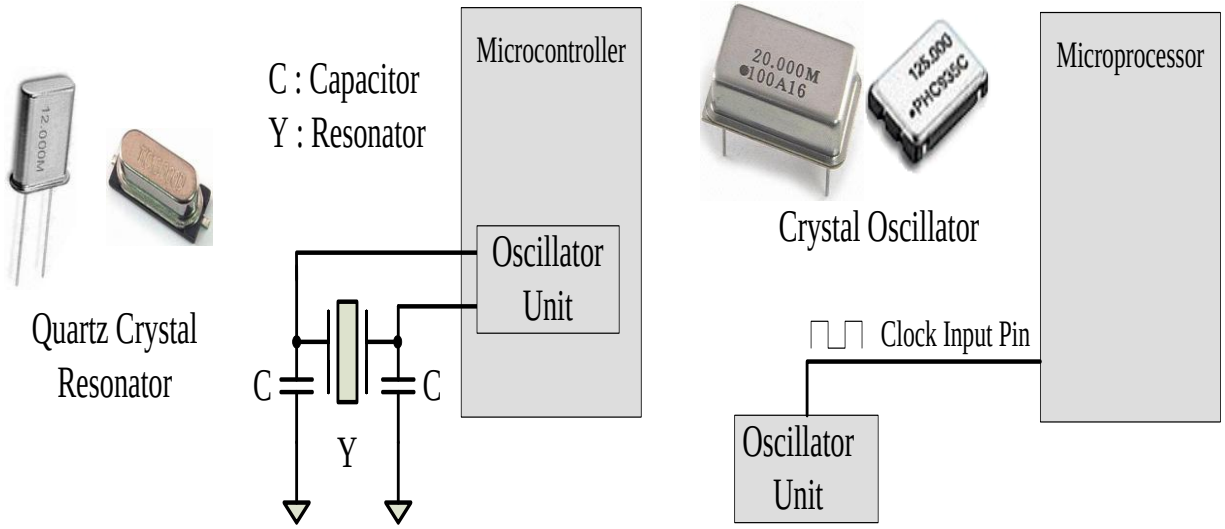


Fig: Oscillator circuitry using quartz crystal and quartz crystal oscillator

3.11.4. Real-Time Clock (RTC)

- The system component responsible for keeping track of time. RTC holds information like current time (In hour, minutes and seconds) in 12 hour /24 hour format, date, month, year, day of the week etc and supplies timing reference to the system.
- RTC is intended to function even in the absence of power. RTCs are available in the form of Integrated Circuits from different semiconductor manufacturers like Maxim/Dallas, ST Microelectronics etc.
- The RTC chip contains a microchip for holding the time and date related information and backup battery cell for functioning in the absence of power, in a single IC package.
- The RTC chip is interfaced to the processor or controller of the embedded system.
- For Operating System based embedded devices, a timing reference is essential for synchronizing the operations of the OS kernel. The RTC can interrupt the OS kernel by asserting the interrupt line of the processor/controller to which the RTC interrupt line is connected. The OS kernel identifies the interrupt in terms of the Interrupt Request (IRQ) number generated by an interrupt controller.

- One IRQ can be assigned to the RTC interrupt and the kernel can perform necessary operations like system date time updation, managing software timers etc when an RTC timer tick interrupt occurs.

3.11.5 Watchdog Timer

- The watchdog timer is a timing device that resets the system after a predefined timeout. It is activated within the first few clock cycles after power-up.
- It helps in rescuing the system if a fault develops and program gets stuck. On restart, the system functions normally.
- The watchdog timer reset is a required feature in control applications.
- Assume that we anticipate that a set of tasks must finish in 100 ms interval.
- The watchdog timer is disabled and stopped by the program instruction in case the tasks finish within 100 ms interval.
- In case task does not finish (not disabled by the program instruction), watchdog timer generates interrupts after 100 ms and executes a routine, which is programmed to run because there is failure of finishing the task in anticipated interval.
- An application in mobile phone is that display is off in case no GUI interaction takes place within a watched time interval.
- The interval is usually set at 15 s, 20 s, 25 s, 30 s in mobile phone. This save power.
- A timer unit for monitoring the firmware execution
- Depending on the internal implementation, the watchdog timer increments or decrement a free running counter with each clock pulse and generates a reset signal to reset the processor if the count reaches zero for a down counting watchdog, or the highest count value for an up counting watchdog.
- If the watchdog counter is in the enabled state, the firmware can write a zero (for up counting watchdog implementation) to it before starting the execution of a piece of code (subroutine or portion of code which is susceptible to execution hang up) and the watchdog will start counting. If the firmware execution doesn't complete due to malfunctioning, within the time required by the watchdog to reach the maximum count, the counter will generate a reset pulse and this will reset the processor.
- If the firmware execution completes before the expiration of the watchdog timer the WDT can be stopped from action.

- Most of the processors implement watchdog as a built-in component and provides status register to control the watchdog timer (like enabling and disabling watchdog functioning) and watchdog timer register for writing the count value. If the processor/controller doesn‟t contain a built in watchdog timer, the same can be implemented using an external watchdog timer IC circuit.

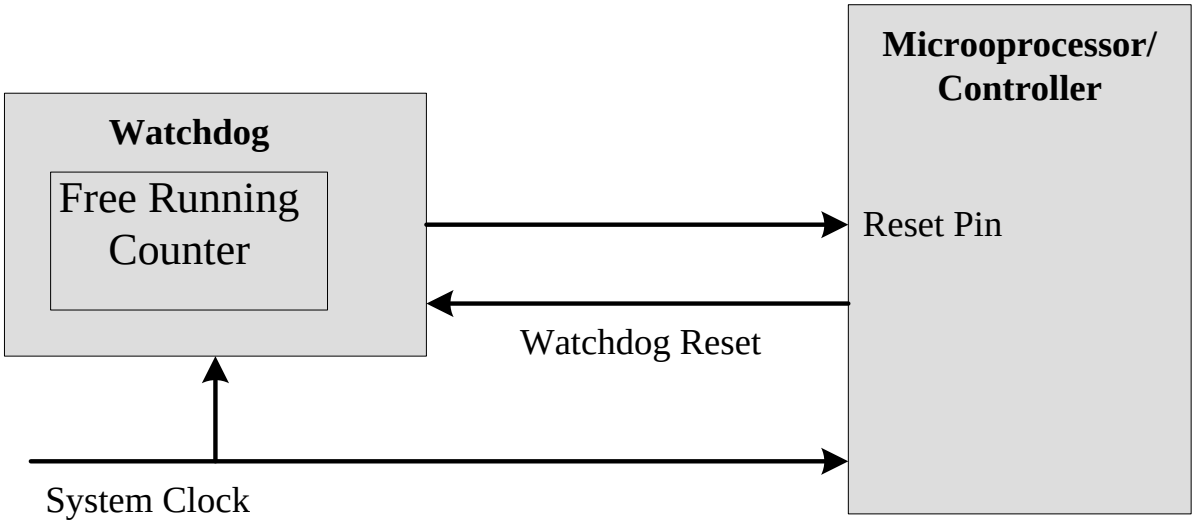


Fig: Watch Dog timer for firmware execution supervision

**********End**********

MCQ

1. Embedded systems are

(a) General purpose (b) Special purpose

2. Embedded system is

(a) An electronic system (b) A pure mechanical system

(c) An electro-mechanical system (d) (a) or (c)

3. Which of the following is not true about embedded systems?

(a) Built around specialized hardware (b) Always contain an operating system

(c) Execution behavior may be deterministic (d) All of these

(e) None of these

4. Which of the following is (are) an intended purpose(s) of embedded systems?

(a) Data collection (b) Data processing (c) Data communication

(d) All of these (e) None of these

5. Which of the following is an (are) example(s) of embedded system for data communication?

(a) USB Mass storage device (b) Network router

(c) Digital camera (d) Music player

(e) All of these (f) None of these

6. A digital multi meter is an example of an embedded system for

(a) Data communication (b) Monitoring

(c) Control (d) All of these

(e) None of these

Question bank

1. What is an embedded system? Explain the different applications of embedded systems. (5 Marks).
2. Difference between embedded systems and general computing systems (5 Marks)
3. Describe the various purposes of embedded systems. Explain any two in detail with illustrative examples. (10 Marks).
4. Explain the 6 purposes of embedded systems with an example for each.Explain the components of a typical embedded system with a neat diagram.
5. Differentiate between
   a. General Computing Systems and Embedded Systems.
   b. RISC and CISC architectures.
   c. microprocessor and microcontroller
   d. Hardvard vs von Neumann
   e. Big endian vs Little endian
6. Explain the 3 classifications of embedded systems based on complexity and performance.
7. Mention the applications of embedded systems with an example for each.
8. What are the core used in embedded system? Discuss in detail.
9. What are the different types of memories used in Embedded system design? Explain the role of each.
10. Brifly explain PLDs and type of PLDs
11. Write a note on embedded firmware.
12. Discuss the I2c communication interface with neat diagram.
13. Explain the sequence of operations for communicating with an I2C slave device.
14. Explain operation of UART. Compare UART and USB.
15. Compare serial and parallel communication.
16. Explain USB protocol with a neat diagram also discuss different data transferred in USB.
17. Write a note on 1-wire bus. Explain its advantage and disadvantages.

18. Explain the reset and brown out protection circuits their significance and application in embedded system.
19. Mention the role of watch dog timer in embedded system with relevant examples.
20. Explain the features of the following: (i)IrDA (ii)WiFi