VIGNAN'S UNIVERSITY

SCHEME OF VALUATION-NOV/DEC 2018

Course	B.Tech				
Regulation	R16	Year	III	Semester	I
Branch/ Specialization	ECE	Course Code	16EC302/1		
Subject Name	MPMC				
Scheme Prepared by	M.SIVA SRINIVASA RAO				
Signature	[illegible]				
Date	19-11-2018				

Section A

1) 2^{20} Bytes

2) ADDRESS LATCH ENABLE: signal indicates the availability of the valid address on the multiplexed lines and is connected to the strobe input of the address latches.
It is a HIGH pulse active during clock low of T_1 of any bus cycle. Note that ALE is never floated.
READY is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The RDY signal from memory or I/O is synchronized by the 8284 clock generator to form READY. This signal can be used by only slowly operated external devices. If the logic level on this pin is equal to 0 then the processor will insert WAIT states between T_3 and T_4. If the logic level on this pin=1, the external device is ready to communicate with the processor.

3)

S.NO	Micro-processor	Micro-controller
1	Micro-processor contains ALU, GPRs, PC, SP, Control and Timing circuit, and interrupt circuit.	Micro-controller contains the CPU, in built ROM, RAM, I/O ports, and Timer &Counter circuits.
2	Microprocessor based system requires more hardware.	Microcontroller based system requires less hardware.
3	Microprocessor based system is more flexible in design point of view	Microcontroller based system is less flexible in design point of view
4	CPU on a single chip is called Microprocessor.	Computer on a single chip is called Microcontroller.

4) **PC (Program Counter):** The program counter points to the address of the next instruction to be executed.
DPTR (Data Pointer): - The DPTR register is made up of two 8-bit registers, named DPH and DPL, that are used to hold memory addresses for internal and external code access and external data access.

5)

TF	Timer Overflow flag. Set when timer rolls from all ones to zero. Cleared when processor vectors to execute interrupt service routine
TR	Timer run control bit. Set to 1 by program to enable timer to count; cleared to 0 by program to halt timer. Does not reset timer.

6)
THE POWER MODE CONTROL (PCON) SPECIAL FUNCTION REGISTER

Bit	Symbol	Function
7	SMOD	Serial baud rate modify bit. Set to 1 by program to double baud rate using timer 1 for modes 1, 2, and 3. Cleared to 0 by program to use timer 1 baud rate.

7)
Advanced RISC Machine:

T	⇨	Thumb
D	⇨	On-Chip Debug support
M	⇨	Enhanced Multiplier
I	⇨	Embedded ICE (In Circuit Emulator)

8) There are three instruction sets: ARM, Thumb, and Jazelle. The Jazelle *J* and Thumb *T* bits in the *cpsr* reflect the state of the processor. When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions. When the T bit is 1, then the processor is in Thumb state and executes *Thumb 16-bit instruction set.*

9)
Register *r13* is traditionally used as the stack pointer (*sp*) and stores the head of the stack in the current processor mode.

▪ Register *r14* is called the link register (*lr*) and is where the core puts the return address whenever it calls a subroutine.

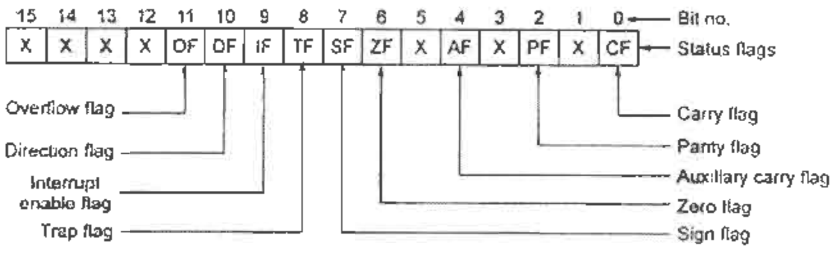
10) save multiple registers increment after/ load multiple registers increment after
Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction. The transfer occurs from a base address register *Rn* pointing into memory.

Section B

11)

Flag Register format – 2 marks
Eacg flag explanation- 0.5 marks ----- 6*0.5 = 3 Marks

Flag Register: -Flags Register determines the current state of the processor. They are modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program. It is also called PSW of 8086.



Status Flags of Intel 8086

- X--- Unused bit positions(total 7)
- Remaining 9 are active bit positions. Active flags of 8086.
- OV=overflow flag DF=direction flag TF=trap flag IF=interrupt flag SF=sign flag
- ZF=zero flag AC=auxiliary carry flag PF=parity flag CY=carry flag
- 9 active flags can be divided into 2 groups.
  a) Conditional flags----6 b) Control flags----3

a) Conditional Flags: -

- Also called status flags.
- Gives the information of conditions of the result produced by the ALU.
- Gives the status of ALU after completion of arithmetic/logical operations.
- All the flags in LSB side are status flags along with OV in MSB side.

CF (Carry flag): When the microprocessor performs the addition of two 8/16-bit numbers, then the result obtained will be of 9/17 bits. The last carry generated i.e. 9/17th bit is stored in CF.

Similarly, when the microprocessor performs the subtraction of two 8/16-bit numbers X - Y, then:

- If X> = Y, then additional borrow required to perform the subtraction X- Y is zero, so CF = 0.
- If X <Y, Men additional borrow required to perform the subtraction X -Y is one, so CF = 1.

PF (Parity flag): When the microprocessor performs any arithmetic or logical operations, then the status of only 8 Least Significant Bits of the result is stored in PF. The parity flag is set to 1, if the result contains even no. of 1's in its LSByte. i.e.

- If the count of numbers of 1s bit is 0/2/4/6/8 in the 8 LSBs of the result, then the result is of even parity, so PF = 1.
- If the count of numbers of 1s bit is 1/3/5/7 in the 8 LSBs of the result, then the result is of odd parity, so PF = 0.

AC (Auxiliary carry flag): When the microprocessor performs the addition of two 8/16-bit numbers, then the carry bit generated after adding 4 Least Significant Bits (nibble) is stored in AC flags. Similarly, when the microprocessor performs the subtraction of two 8/16 bit-numbers then the borrow required to perform subtraction of 4 LSBs is stored in AC flags.

This is not a general-purpose flag; it is used internally by the processor to perform Binary to BCD conversion
ZF (Zero flag): When the microprocessor performs any arithmetic or logical operations of the 8-bit number, then all the 8 LSBs of the result are zero, i.e. 00h, then ZF = 1 otherwise ZF = 0. Similarly, when the microprocessor performs any arithmetic or logical operations of the 16-bit number, then all the 16 LSBs of one result are zero 0000h, then ZF = 1 otherwise ZF = 0.

SF (Sign flag): When the microprocessor performs any arithmetic or logical operations of the 8/16-bit number, then Most Significant Bits of the 8–bit result/16-bit result are directly stored into sign flags. If the result obtained is correct binary number then the sign flags bit will give correct sign of the result. If SF = 0/1, then the result is +ve/-ve respectively.

OF (Overflow flag): When the microprocessor performs any arithmetic or logical operations of the 8/16-bit sign binary numbers and when the sign binary result of the 8/16-bit is out of the range, i.e. incorrect, then OF = 1, so sign flag bit will also be incorrect. If the 8/16-bit sign binary result is within the range, i.e. the result is correct sign binary number, then OF = 0, so sign flag bit will give correct sign of the result.

When the microprocessor performs the addition of two 8/16-bit numbers, then:

(a) If carry into the MSBs is equal to the carry out of the MSBs, then the sign result is correct and OF = 0.
(b) If carry into the MSBs is not equal to the carry out of the MSBs, then the sign result is incorrect and OF = 1.
Similarly, when the microprocessor performs the subtraction of 8/16-bit numbers, then:

(a) If borrow into the MSBs is equal to the borrow out of the MSBs, then the sign result is correct and OF = 0.
(b) If borrow into the MSBs is not equal to the borrow out of the MSBs, then sign result is incorrect and OF = 1.

12)

```
ASSUME CS:CODE, DS:DATA

CODE SEGMENT

MOV AX,DATA
MOV DS,AX

MOV CX,0008H
LEA SI, ARRAY
LEA DI,RESULT

XOR DX,DX
XOR AX,AX
XOR BX,BX
L1:ADD AX,[SI]
JNC L2
INC DX
L2: INC SI
INC SI
INC SI
```

7Mark

```
INC SI
LOOP L1

MOV CX,0008H
LEA SI, ARRAY
INC SI
INC SI
L4:ADD DX,[SI]
JNC L3
INC BX
L3: INC SI
INC SI
INC SI
INC SI
LOOP L4

MOV [DI],AX
MOV [DI+2],DX
MOV [DI+4],BX
INT 03
CODE ENDS

DATA SEGMENT

ARRAY DD 08 DUP(0)
RESULT DW 03 DUP(0)
DATA ENDS
END
```

2m

2m

13)

Memory Segmentation-	3Marks
Advantages-	2Marks

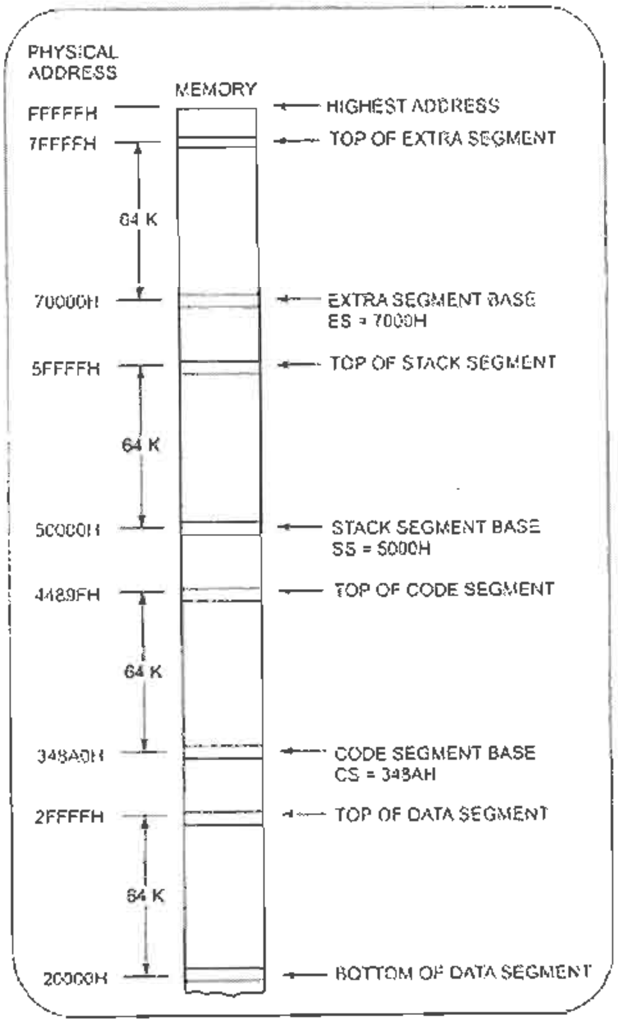
Memory segmentation:

- The 8086 BIU sends out 20-bit addresses, so it can address any of 2^{20} or 1,048,576 bytes in memory. However, at any given time the 8086 works with only four 65,536 byte (64-Kbyte) segments within this 1,048,576 byte (1-Mbyte) range.
- Four segment registers in the BIU are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with at a particular time. The four segment registers are the code segment (CS) register, the stack segment (SS) register, the extra segment (ES) register, and the data segment (DS) register.
- Figure shows how these four segments might be positioned in memory at a given time. The four segments can be separated as shown, or, for small programs which do not need all 64 Kbytes in each segment, they can overlap.

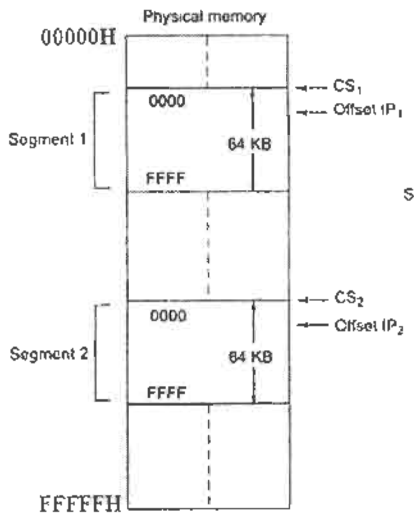
Advantages due to Memory Segmentation: -

- It allows the memory addressing capacity to be 1 Mbyte even though the address associated with individual instruction is only 16-bit.
- It allows instruction code, data, stack, and portion of program to be more than 64 KB long by using more than one code, data, stack segment, and extra segment.
- It facilitates use of separate memory areas for program, data and stack.

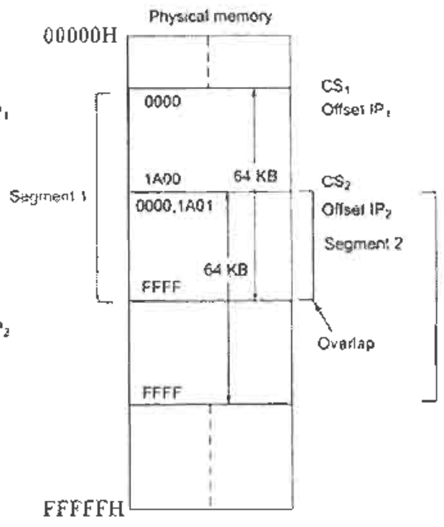
- It permits a program or its data to be put in different areas of memory, each time the program is executed i.e. program can be relocated which is very useful in multiprogramming.
- Program can work on several data sets by reloading the DS register.



One way four 64-Kbyte segments might be positioned within the 1-Mbyte address space of an 8086.



Non-overlapping Segments



Overlapping Segments

- As mentioned a segment register is used to hold the upper 16 bits of the starting address for each of the segments. The code segment register, for example, holds the upper 16 bits of the starting address for the segment from which the BIU is currently fetching instruction code bytes.
- The BIU always inserts zeros for the lowest 4 bits (nibble) of the 20-bit starting address for a segment. If the code segment register contains 348AH, for example, then the code segment will start at address 348A0H. In other words, a 64-Kbyte segment can be located anywhere within the 1-Mbyte address space, but the segment will always start at an address with zeros in the lowest 4 bits.
- The part of a segment starting address stored in a segment register is often called the segment base.

- An alternate way of representing a 20-bit physical address is the Segmentbase : offsetform
- Ex:- CS:IP
  DS:SI
- **Physical Address = Base Address * 10H + Offset Address**

14)

```
MOV CX,000FH
MOV SI,2000H
MOV DI,3000H
```

}1mark

```
L1:MOV AX,[SI]
MOV [DI],AX
```

2marks

```
ADD SI,02
ADD DI,02
LOOP L1
INT 03
END
```

} 2marks

15)

OnChip memory Identification-	1 Mark
RAM organisation-	2 Marks
ROM organisation-	2Marks

MEMORY ORGANISATION

The 8051 devices have 4 Kbytes of on-chip program memory and 128 bytes of on-chip data random access memory.

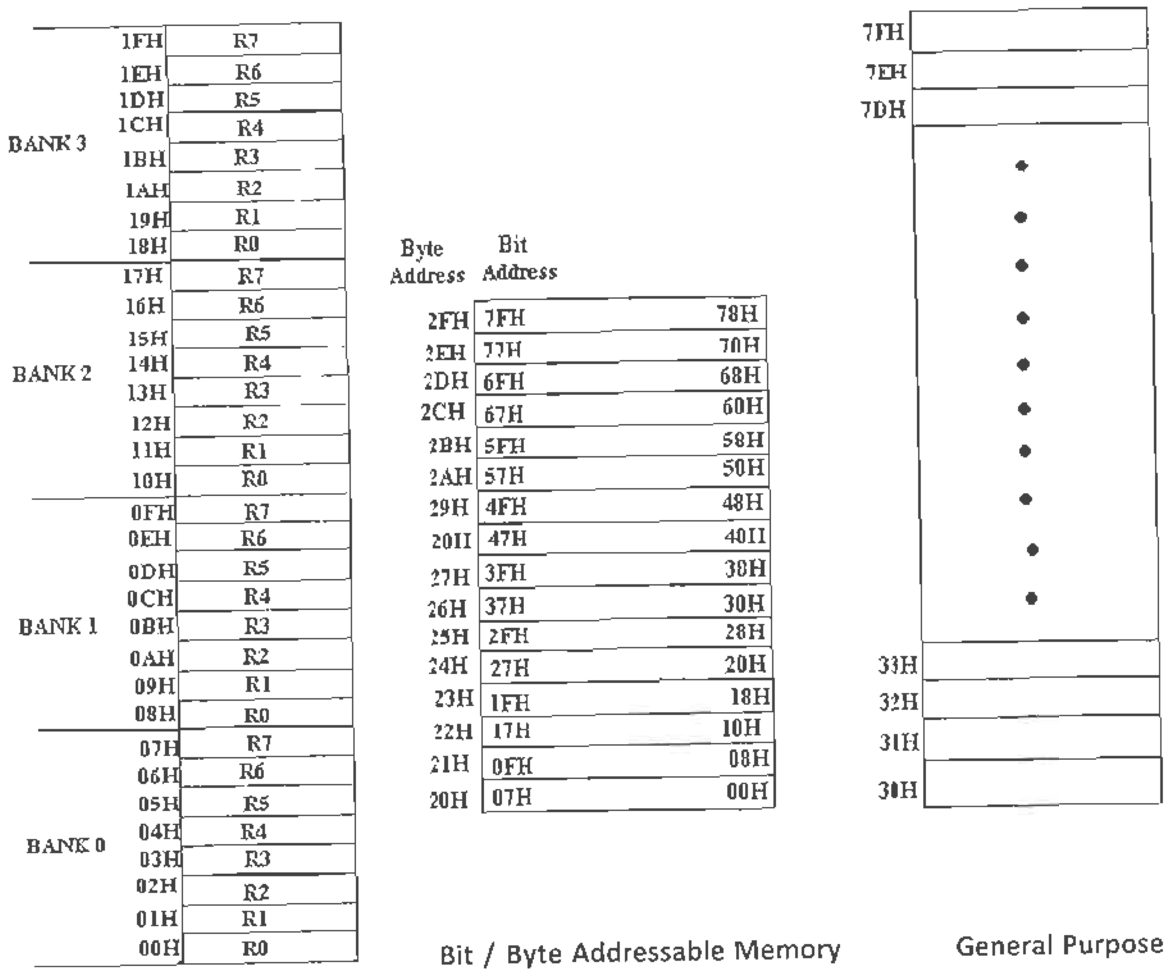
Internal RAM: - The 8051 microcontroller has 128 bytes of internal RAM, its address range from 00H to 07FH. From 80H to 0FFH address space are assigned to SFRs (Special Function Registers). The internal RAM 128Bytes can divide into three parts. Those are

1. Register Banks – 32 Bytes (00H – 1FH)
2. Bit/Byte addressable memory – 16 Bytes (20H – 2FH)
3. User memory or General purpose memory—80 Bytes (30H – 7FH)

A. Thirty two bytes addresses from 00H to 1FH that make up 32 working registers organized as four register banks of eight registers each. The register banks are numbered from 0 to 3 and are made up of eight registers named R0 to R7. Each register can be called in to the program by its name or by its RAM address. The register bank selection bits RS1 & RS0 in the PSW decides which register bank currently used for the operation of the microcontroller. The register bank-0 is selected when the microcontroller is reset.

B. The Bit / Byte addressable area of 16-Bytes occupies RAM addresses from 20H-2FH forming a total of 128 addressable bits. A bit addressable bit can be specified by its bit addresses from 00H to 7FH or 8- bits may form a byte address from 20H-2FH. Addressable bits are useful when the program need only remember a binary equivalent.

C. A general purpose RAM area starts from 30H-7FH. This is addressable as bytes.



Bit / Byte Addressable Memory General Purpose

Memory

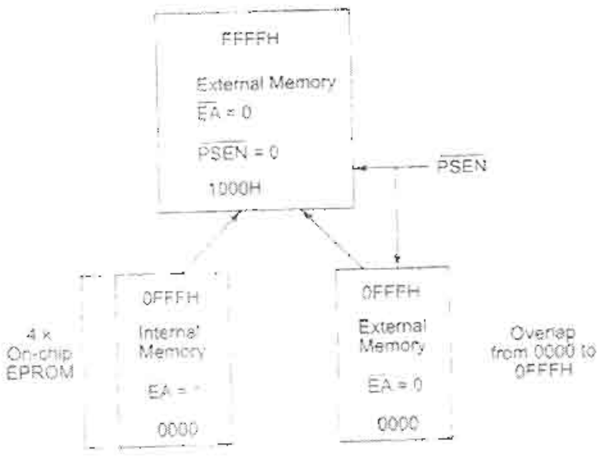
Register Banks Memory

FIG: INTERNAL RAM ORGANISATION

Internal ROM: -The 8051 microcontroller has an internal ROM of 4Kbytes, its addresses from 0000H to 0FFFH. This is used to store the system files or system code information.

The program addresses higher than the 0FFFH, which exceeds the internal ROM capacity, will cause the 8051to automatically fetch the program codes from external program memory.

Code bytes can also be fetched exclusively from the external program memory addresses from 0000H to FFFFH, by connecting the EA (External Active or External Enable) pin to the ground.



Program Memory Map of an 8051 System

16)

Finding Count –	1Mark
Finding Mode of operation-	1Mark
Program-	3Marks

Look at the following steps.

(a) T = 1 / f = 1 / 1 kHz = 1000 μs the period of square wave.

(b) 1 / 2 of it for the high and low portion of the pulse is 500 μs.

(c) Assume XTAL = 11.0592 MHz, The timer works with a clock frequency of 1/12 of the XTAL frequency; therefore, we have 11.0592 MHz / 12 = 921.6 kHz as the timer frequency. As a result, each clock has a period of T = 1/921.6kHz = 1.085us.

500 μs / 1.085 μs = 462 and 65536 – 462 = 65075 which in hex is FE33H.

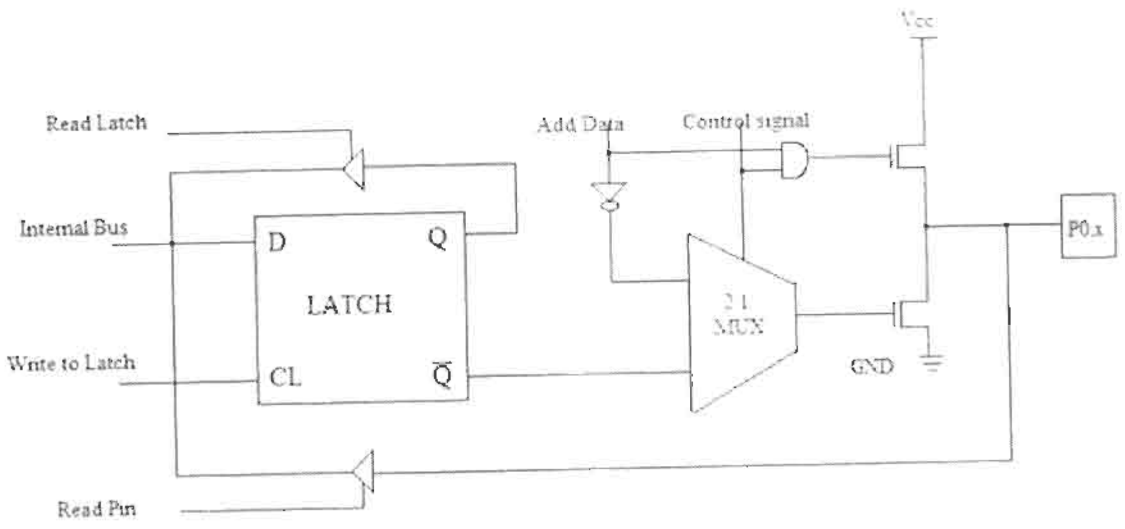
(d) TL = 33 and TH = FE, all in hex. The program is as follow.

```
            MOV TMOD,#01            ;Timer 0, 16-bitmode
AGAIN:      MOV TL1,#33H            ;TL1=33, low byte of timer
            MOV TH1,#0FEH           ;TH1=FE, the high byte
            SETB TR1                ;Start timer 1
BACK:       JNB TF1,BACK            ;until timer rolls over
            CLR TR1                 ;Stop the timer 1
            CPL P1.3                ;Complement port1.3 pin
            CLR TF1                 ;Clear timer 1 flag
            SJMP AGAIN              ;Reload timer
```

17)

Port 0 Pin configuration-	3 Marks
Port 3 other function-	2 Marks

Port-0 Pin configuration



- When a pin is to be used as an input, a 1 must be written to the corresponding port 0 latch by the program, thus turning both of the output transistors off, which in turn causes the pin to "float" in a high-impedance state, and the pin is essentially connected to the input buffer.
- When used as an output, the pin latches that are programmed to a 0 will turn on the lower FET, grounding the pin. All latches that are programmed to a 1 still float; thus, external pullup resistors will be needed to supply logic high when using port 0 as an output.

(RXD) P3.0
(TXD) P3.1
($\overline{\text{INT0}}$) P3.2
($\overline{\text{INT1}}$) P3.3
(T0) P3.4
(T1) P3.5
($\overline{\text{WR}}$) P3.6
($\overline{\text{RD}}$) P3.7

18)

Addressing modes 5*1 = 5 Marks

Addressing Modes of 8051 Microcontroller:-

An Addressing Mode indicates how the data is represented in the instruction. 8051 supports 6 types of Addressing Modes, those are

1. Immediate Addressing mode
2. Register Addressing Mode
3. Register Indirect Addressing Mode
4. Direct Addressing Mode
5. Indexed Addressing Mode

1.Immediate Addressing Mode: -In these addressing mode instructions the data is directly placed in the source operand field of the instruction, and a '#' symbol must prefix for the data.

Ex: - MOV A, #38H
ADD A, #67H

2. Register Addressing Mode: -In these addressing mode instructions the data is directly placed in the operand field of the instruction through a GPR (General Purpose Register).

EX: - MOV A, R6
ADD A, R5

3. Register Indirect Addressing Mode: -In these addressing mode instructions the address of the data is indirectly placed in the operand field of the instruction through a GPR (General Purpose Register) R0 or R1.

Ex: - MOV A, @R0
ADD A, @R1

4. Direct Addressing Mode: - In these addressing mode instructions the address of the data is directly placed in the operand field of the instruction. The address is the internal RAM or internal SFR (Special Function Register).

Ex: - MOV A, 20H
MOV R1, 70H

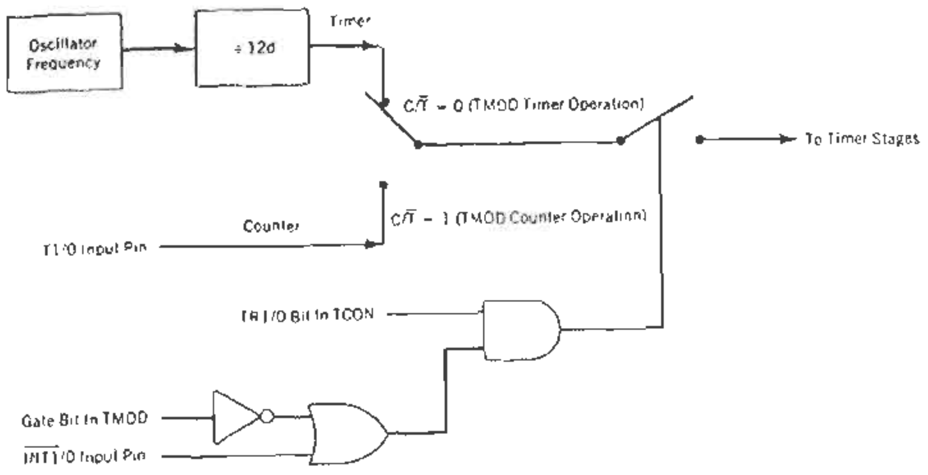
5. Indexed Addressing Mode: -In these addressing mode instructions the address of the data is indirectly placed in the operand field of the instruction through combination 'A' register PC or DPTR.

Ex: - MOVC A, @A+DPTR
MOVC A, @A+PC

SECTION C

19)

Timer Logic -	2 Marks
Timer modes of operation-	4 Marks
Program-	4 Marks


Oscillator Frequency
÷ 12d
Timer
C/$\overline{T}$ = 0 (TMOD Timer Operation)
To Timer Stages
Counter
C/$\overline{T}$ = 1 (TMOD Counter Operation)
T1/0 Input Pin
TR1/0 Bit In TCON
Gate Bit In TMOD
$\overline{INT1}$/0 Input Pin

Timer/Counter Control Logic

13-Bit Timer Mode (Mode 0)

Mode 0 is a 13-bit timer mode that provides compatibility with the 8051's predecessor, the 8048. It is not generally used in new designs. (See Figure) The timer high-byte (THx) is cascaded with the five least-significant bits of the timer low-byte (TLx) to form a 13-bit timer. The upper three bits of TLx are not used.

16-Bit Timer Mode (Mode 1)

Mode 1 is a 16-bit timer mode and is the same as mode 0, except the timer is operating as a full 16-bit timer.

The clock is applied to the combined high and low timer registers (TLx/THx).

As clock pulses are received, the timer counts up: 0000H, 0001H, 0002H, etc.

An overflow occurs on the 0FFFFH-to-0000H transition of the count and sets the timer overflow flag. The timer continues to count.

The overflow flag is the TFx bit in TCON that is read or written by software.

The most-significant bit (MSB) of the value in the timer registers is THx bit 7, and the least-significant bit (LSB) is TLx bit 0.

The LSB toggles at the input clock frequency divided by 2, while the MSB toggles at the input clock frequency divided by 65,536 (i.e., 2^{16}.) .

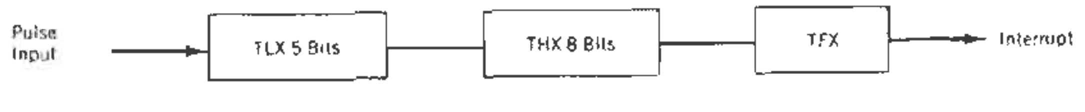
The timer registers (TLx/THx) may be read or written at any time by software.

8-Bit Auto-Reload Mode (Mode 2)

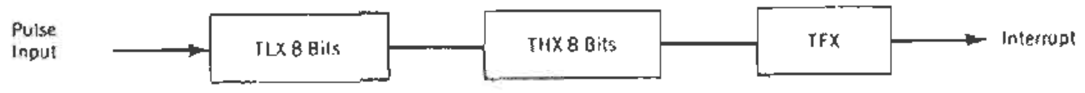
Mode 2 is 8-bit auto-reload mode. The timer low-byte (TLx) operates as an 8-bit timer while the timer high-byte (THx) holds a reload value.

When the count overflows from 0FFH, not only is the timer flag set, but the value in THx is loaded into TLx; counting continues from this value up to the next 0FFH overflow, and so on. If TLx contains 4FH, for example, the time counts continuously from 4FH to 0FFH.

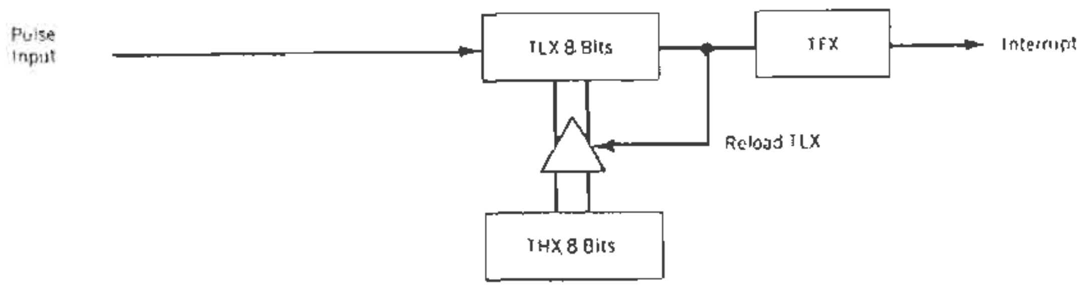
FIGURE Timer 1 and Timer 0 Operation Modes



Timer Mode 0 13 - Bit Timer/Counter



Timer Mode 1 16 - Bit Timer/Counter



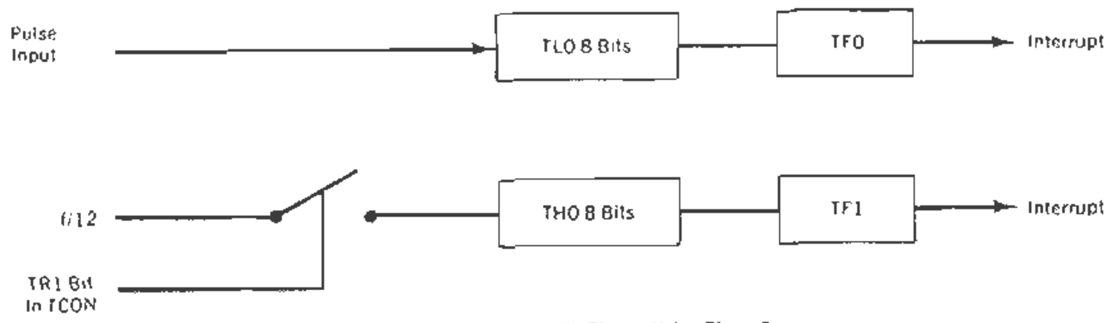
Timer Mode 2 Auto - Reload of TL from TH

Split Timer Mode (Mode 3)

Mode 3 is the split timer mode and is different for each timer.

Timer 0 in mode 3 is split into two 8-bit timers. TL0 and TH0 act as separate timers with overflows setting the TF0 and TF1 bits, respectively.

Timer 1 is stopped in mode 3 but can be started by switching it into one of the other modes. The only limitation is that the usual Timer 1 overflow flag, TF1, is not affected by Timer 1 overflows, since it is connected to TH0.



Timer Mode 3 Two 8 - Bit Timers Using Timer 0 program that continuously get 8-bit data from P0 and sends it to P1 while simultaneously creating a square wave of 200 µs period on pin P2.1. Use timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

We will use timer 0 in mode 2 (auto reload). TH0 = 100/1.085 µs = 92

```
                                    ;--upon wake-up go to main, avoid using
                                    ;memory allocated to Interrupt Vector Table

ORG 0000H
LJMP MAIN                           ;by-pass interrupt vector table
                                    ;--ISR for timer 0 to generate square wave
ORG 000BH                           ;Timer 0 interrupt vector table
CPL P2.1                            ;toggle P2.1 pin
RETI                                ;return from ISR
                                    ;--The main program for initialization
ORG 0030H                           ;after vector table space
MAIN: MOV TMOD,#02H                 ;Timer 0, mode 2
          MOV P0,#0FFH              ;make P0 an input port
          MOV TH0,#-92              ;TH0=A4H for -92
          MOV IE,#82H               ;IE=10000010 (bin) enable ;Timer 0
          SETB TR0                  ;Start Timer 0
BACK: MOV A,P0                      ;get data from P0
          MOV P1,A                  ;issue it to P1
          SJMP BACK                 ;keep doing it loop unless interrupted by TF0
          END
```

21)

Move instructions	2Marks
Arithmetic	2Marks
Logical	2Marks
Comparison	2Marks
Multiply	2Marks

- Data processing inst manipulate data within registers
  1. Move instructions
  2. Arithmetic
  3. Logical
  4. Comparison
  5. Multiply
  - ➢ Most of these instructions can process one of their operands using the barrel shifter

Move Inst

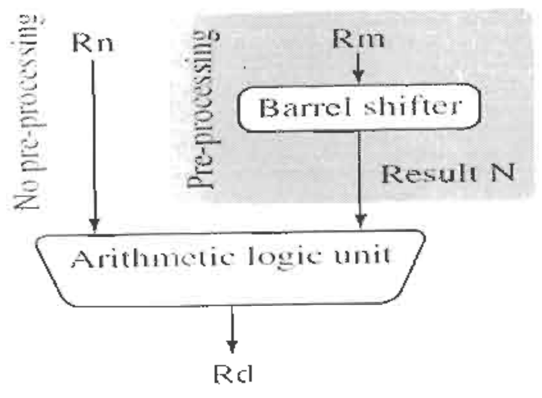
- Useful for settings initial values and transferring data between registers.
- *Syntax: <instruction>{<condition>}{S} Rd, N*
- *The operand N is a register or a constant preceded by #.*
- MOV: move a 32-bit value into a register Rd=N
- MVN: Rd=~N
- If suffix S is used in inst, then it updates flags in the cpsr.
- Move and logical inst update the C, N, and Z.

Example:

```
PRE   r3=8
         r4=2
mov  r3, r4;
POST  r3=2
         r4 =2
```

Barrel Shifter

- In syntax N need not be number it may be register also
- 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 the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- This shift increases the power and flexibility of many data processing operations.
- Pre-processing or shift occurs within the cycle time of the instruction.
- useful for loading constants into a register
- fast multiplies or division by a power of 2.



Barrel shifter and ALU.

Barrel shifter operations.

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

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

Barrel shift operation syntax for data processing instructions.

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

Example

```
PRE cpsr = nzcvqiFt_USER
r0 = 0x00000000
r1 = 0x80000004
MOVS  r0, r1, LSL #1
POST cpsr = nzCvqiFt_USER
r0 = 0x00000008
r1 = 0x80000004
```

Arithmetic Instructions

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:

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
```

- 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
```

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

```
PRE r0 = 0x00000000
    r1 = 0x00000005
    ADD r0, r1, r1, LSL #1
POST r0 = 0x0000000f
    r1 = 0x00000005
```

Logical Instructions

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

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

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

- This example shows a logical OR operation between registers *r1 and r2. r0 holds the result.*

```
PRE r0 = 0x00000000
    r1 = 0x02040608
    r2 = 0x10305070
    ORR r0, r1, r2
POST r0 = 0x12345678
```

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.*
- You do not need to apply the S suffix for comparison instructions to update the flags.
- Syntax: <instruction>{<cond>} Rn, N

Example

```
    PRE cpsr = nzcvqiFt_USER
    r0 = 4
    r9 = 4
    CMP r0, r9
    POST cpsr = nZcvqiFt_USER
```

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 $Rn \& N$

Multiply Instructions

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

Syntax:

- MLA{<cond>}{S} Rd, Rm, Rs, Rn
- MUL{<cond>}{S} Rd, Rm, Rs

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

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

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

Example:

PRE r0 = 0x00000000

r1 = 0x00000002

r2 = 0x00000002

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

POST r0 = 0x00000004

r1 = 0x00000002

r2 = 0x00000002

Example:

```
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
```

22)

Load Store instruction syntax-	2Marks
Types	2Marks
Addressing modes	3Marks
Examples-	3Marks

<u>Single register transfer</u>

- Single data item
- Data types

  signed and unsigned words (32 –bit)

  Half words (16-bit)

  Bytes

Syntax:

```
<LDR/STR> {<cond>} {B} Rd, addressing
<LDR> {<cond>} SB|H|SH Rd, addressing
<STR> {<cond>} H Rd, addressing
```

Instructions Mnemonics

• LDR	Rd← mem32[address]
• STR	Rd → mern32[address]
• LDRB	Rd← mem8[address]
• STRB	Rd → mem8[address]
• LDRH	Rd← mem16[address]
• STR H	Rd → mem16[address]
• LDRSB	Rd← signextend(mem32[address])
• LTRSH	Rd← signextend(mem32[address])

Single register Load – Store Addressing modes -Indexing methods

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.

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`

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

Table 3.6 Examples of LDR instructions using different addressing modes.

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

Table 3.8 Variations of STRH instructions.

	Instruction	Result	*r1* + =
Preindex with writeback	STRH r0,[r1,#0x4]!	mem16[r1+0x4]=r0	0x4
	STRH r0,[r1,r2]!	mem16[r1+r2]=r0	r2
Preindex	STRH r0,[r1,#0x4]	mem16[r1+0x4]=r0	*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

23) ARM core dataflow model-	2 Marks
Register Set -	3 Marks
CPSR format-	2 Marks
Modes of operation -	3 Marks

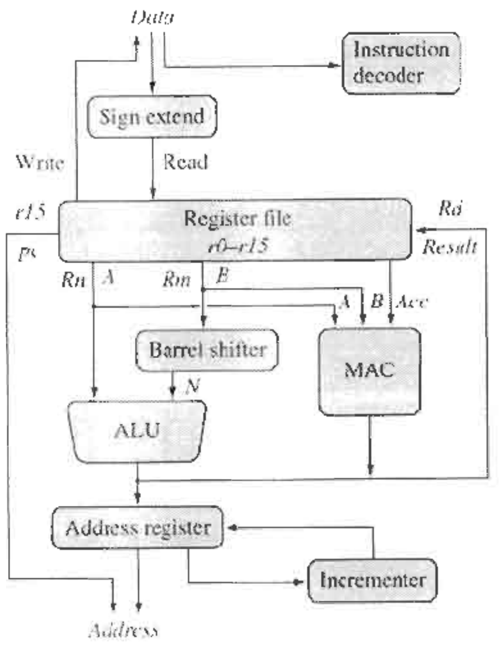


Figure 2.1 ARM core dataflow model.

General-purpose registers

General-purpose registers hold either data or an address. They are identified with the letter *r* prefixed to the register number. For example, register 4 is given the label *r4*.Figure 2.2 shows the active registers available in *user* mode—a protected mode normally used when executing applications. The processor can operate in seven different modes, which we will introduce shortly. All the registers shown are 32 bits in size. There are up to 18 active registers: 16 data registers and 2 processor status registers. The data registers are visible to the programmer as *r0* to *r15*.The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*. They are frequently given different labels to differentiate them from the other registers.

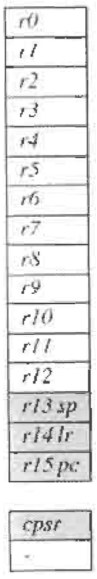


Figure 2.2 Registers available in *user* mode.

■ Register r13 is traditionally used as the stack pointer (sp) and stores the head of the stack in the current processor mode.
■ Register r14 is called the link register (lr) and is where the core puts the return address whenever it calls a subroutine.
■ Register r15 is the program counter (pc) and contains the address of the next instruction to be fetched by the processor.

Of those 37 registers in the register file, 20 registers are hidden from a program at different times. These registers are called banked registers and are identified by the shading in the diagram. They are available only when the processor is in a particular mode; for example, abort mode has banked registers r13_abt, r14_abt and spsr_abt. Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or _mode.
Every processor mode except user mode can change mode by writing directly to the mode bits of the cpsr. All processor modes except system mode have a set of associated banked registers that are a subset of the main 16 registers. A banked register maps one-to one onto a user mode register. If you change processor mode, a banked register from the new mode will replace an existing register.
For example, when the processor is in the interrupt request mode, the instructions you execute still access registers named r13 and r14. However, these registers are the banked registers r13_irq and r14_irq. The user mode registers r13_usr and r14_usr are not affected by the instruction referencing these registers. A program still has normal access to the other registers r0 to r12.
The processor mode can be changed by a program that writes directly to the cpsr (the processor core has to be in privileged mode) or by hardware when the core responds to an exception or interrupt. The following exceptions and interrupts cause a mode change: reset, interrupt request, fast interrupt request, software interrupt, data abort, prefetch abort, and undefined instruction. Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location. Figure 2.5 illustrates what happens when an interrupt forces a mode change. The figure shows the core changing from user mode to interrupt request mode, which happens when an interrupt request occurs due to an external device raising an interrupt to the processor core. This change causes user registers r13 and r14 to be banked. The user registers are replaced with registers r13_irq and r14_irq, respectively. Note r14_irq contains the return address and r13_irq contains the stack pointer for interrupt request mode. Figure 2.5 also shows a new register appearing in interrupt request mode: the saved program status register (spsr), which stores the previous mode's cpsr. You can see in the diagram the cpsr being copied into spsr_irq. To return back to user mode, a special return instruction is used that instructs the core to restore the original cpsr from the spsr_irq and bank in the user registers r13 and r14. Note that the spsr can only be modified and read in a privileged mode. There is no spsr available in user mode. Another important feature to note is that the cpsr is not copied into the spsr when a mode change is forced due to a program writing directly to the cpsr. The saving of the cpsr only occurs when an exception or interrupt is raised. Figure 2.3 shows that the current active processor mode occupies the five least significant bits of the cpsr. When power is applied to the core, it starts in supervisor mode, which is privileged. Starting in a privileged mode is useful since initialization code can use full access to the cpsr to set up the stacks for each of the other modes. Table 2.1 lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the cpsr.

Table 2.1 Processor mode.

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

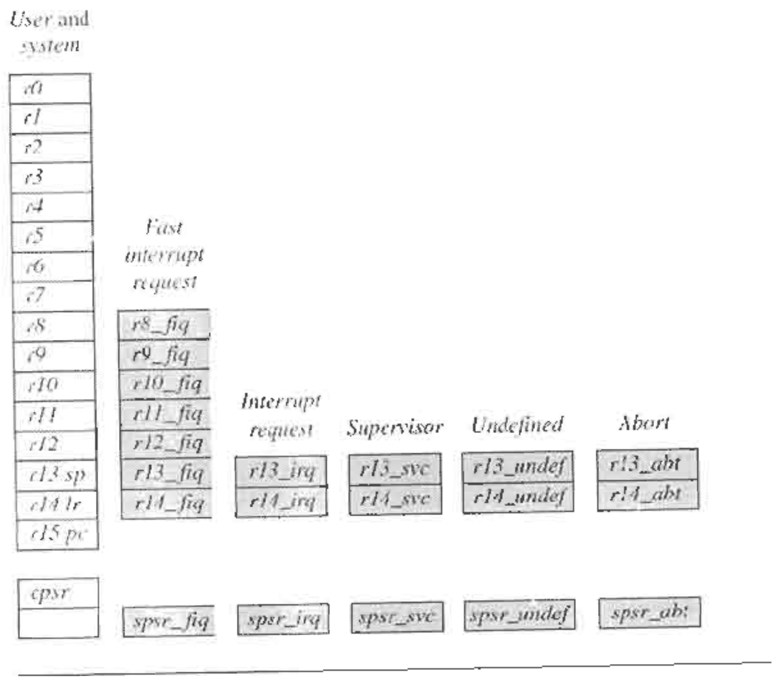


Figure 2.4 Complete ARM register set.

Processor Modes

The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or nonprivileged: A privileged mode allows full read-write access to the *cpsr*. Conversely, a nonprivileged mode only allows read access to the control field in the *cpsr* but still allows read-write access to the condition flags.

There are seven processor modes in total: six privileged modes (*abort*, *fast interruptrequest*, *interrupt request*, *supervisor*, *system*, and *undefined*) and one nonprivileged mode (*user*).

The processor enters *abort* mode when there is a failed attempt to access memory. *Fast interrupt request* and *interrupt request* modes correspond to the two interrupt levels available on the ARM processor. *Supervisor* mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in. *System* mode is a special version of *user* mode that allows full read-write access to the *cpsr*. *Undefined* mode is used when the processor encounters an instruction that is undefined or not supported by the implementation. *User* mode is used for programs and applications.

Current Program Status Register

The ARM core uses the *cpsr* to monitor and control internal operations. The *cpsr* is a dedicated 32-bit register and resides in the register file. Figure 2.3 shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.
The *cpsr* is divided into four fields, each 8 bits wide: flags, status, extension, and control. In current designs the extension and status fields are reserved for future use. The control field contains the processor mode, state, and interrupt mask bits. The flags field contains the condition flags.
Some ARM processor cores have extra bits allocated. For example, the *J* bit, which can be found in the flags field, is only available on Jazelle-enabled processors, which execute

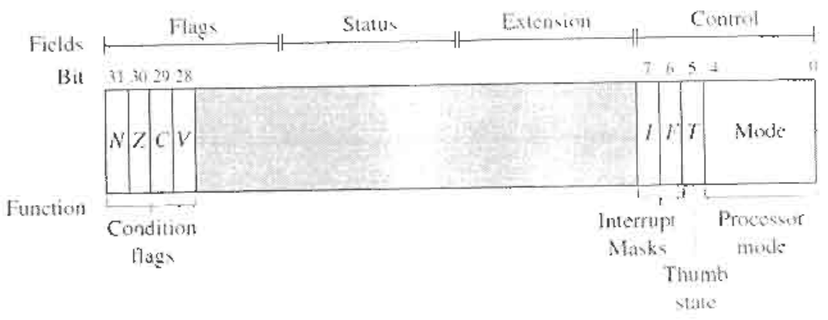


Figure 2.3 A generic program status register (*psr*).

8-bit instructions. We will discuss Jazelle more in Section 2.2.3. It is highly probable that future designs will assign extra bits for the monitoring and control of new features.
For a full description of the *cpsr*, refer to Appendix B.

24)

Evolution-	**2 Marks**
ARM design Philosophy –	**4 Marks**
RISC VS CISC –	**4 Marks**

The ARM Design Philosophy
There are a number of physical features that have driven the ARM processor design. First, portable embedded systems require some form of battery power. The ARM processor has been specifically designed to be small to reduce power consumption and extend battery operation—essential for applications such as mobile phones and personal digital assistants (PDAs).

High code density is another major requirement since embedded systems have limited memory due to cost and/or physical size restrictions. High code density is useful for applications that have limited on-board memory, such as mobile phones and mass storage devices.

In addition, embedded systems are price sensitive and use slow and low-cost memory devices. For high-volume applications like digital cameras, every cent has to be accounted for in the design. The ability to use low-cost memory devices produces substantial savings.

Another important requirement is to reduce the area of the die taken up by the embedded processor. For a single-chip solution, the smaller the area used by the embedded processor, the more available space for specialized peripherals. This in turn reduces the cost of the design and manufacturing since fewer discrete chips are required for the end product.

ARM has incorporated hardware debug technology within the processor so that software engineers can view what is happening while the processor is executing code. With greater visibility, software engineers can resolve issues faster, which has a direct effect on the time to market and reduces overall development costs.

The ARM core is not a pure RISC architecture because of the constraints of its primary application—the embedded system. In some sense, the strength of the ARM core is that it does not take the RISC concept too far. In today's systems the key is not raw processor speed but total effective system performance and power consumption.

ARM Processor Families

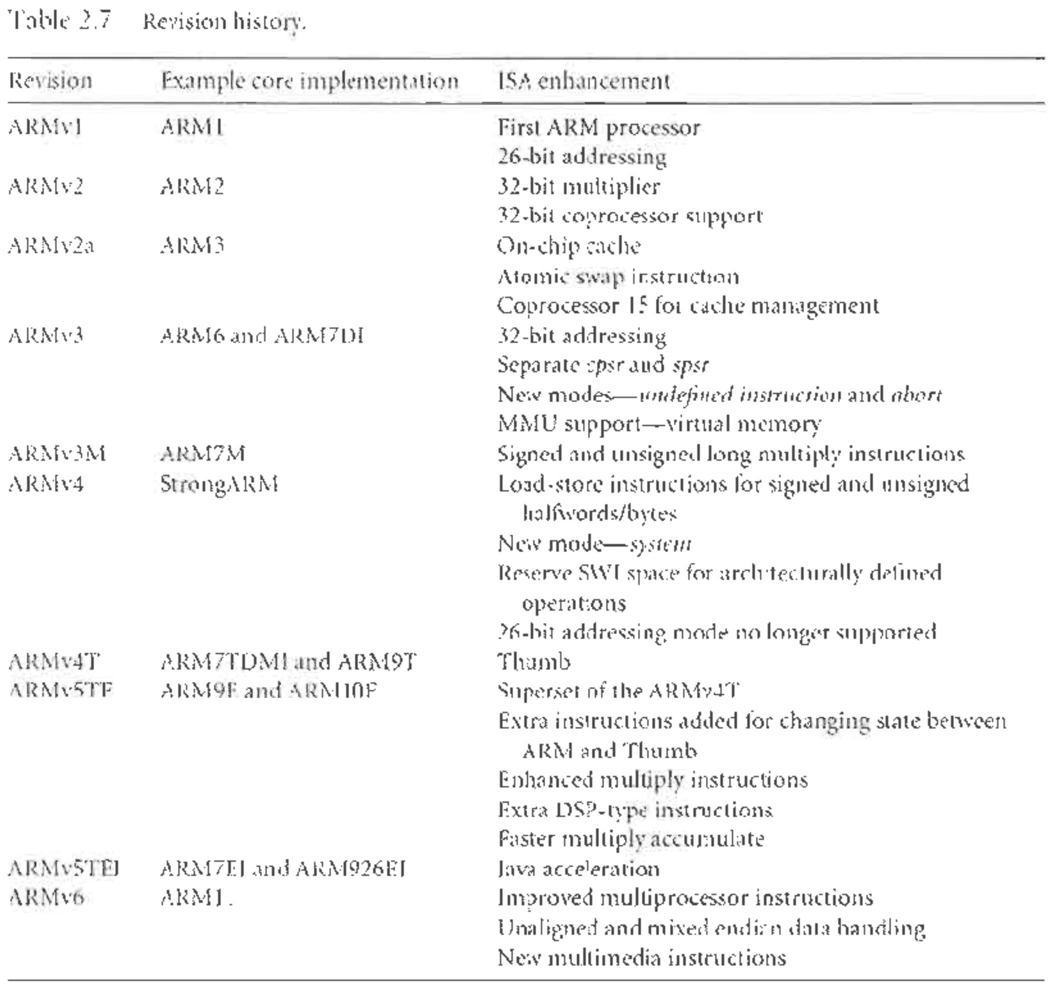


Table 2.7 Revision history.

Revision	Example core implementation	ISA enhancement
ARMv1	ARM1	First ARM processor 26-bit addressing
ARMv2	ARM2	32-bit multiplier 32-bit coprocessor support
ARMv2a	ARM3	On-chip cache Atomic swap instruction Coprocessor 15 for cache management
ARMv3	ARM6 and ARM7DI	32-bit addressing Separate *cpsr* and *spsr* New modes—*undefined instruction* and *abort* MMU support—virtual memory
ARMv3M	ARM7M	Signed and unsigned long multiply instructions
ARMv4	StrongARM	Load-store instructions for signed and unsigned halfwords/bytes New mode—*system* Reserve SWI space for architecturally defined operations 26-bit addressing mode no longer supported
ARMv4T	ARM7TDMI and ARM9T	Thumb
ARMv5TE	ARM9E and ARM10E	Superset of the ARMv4T Extra instructions added for changing state between ARM and Thumb Enhanced multiply instructions Extra DSP-type instructions Faster multiply accumulate
ARMv5TEJ	ARM7EJ and ARM926EJ	Java acceleration
ARMv6	ARM11	Improved multiprocessor instructions Unaligned and mixed endian data handling New multimedia instructions

RISC vs CISC

1. Instructions—RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (for example, a divide operation) by combining several simple instructions. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction. In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.

2. Pipelines—The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage. There is no need for an instruction to be executed by a mini program called microcode as on CISC processors.

3. Registers—RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data processing operations. In contrast, CISC processors have dedicated registers for specific purposes.

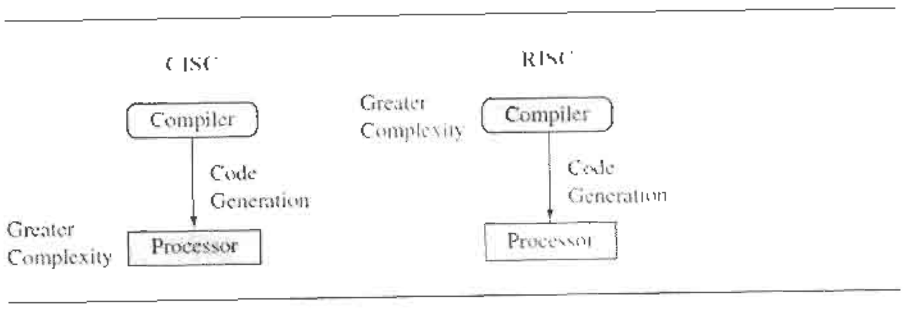


Figure 1.1 CISC vs. RISC. CISC emphasizes hardware complexity. RISC emphasizes compiler complexity.

4. Load-store architecture—The processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory. Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses. In contrast, with a CISC design the data processing operations can act on memory directly.

These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies. In contrast, traditional CISC processors are more complex and operate at lower clock frequencies. Over the course of two decades, however, the distinction between RISC and CISC has blurred as CISC processors have implemented more RISC concepts.

20)
Interfacing Diagram-5 Marks
Program- 5 Marks

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
org 0h

RAM_ADDR       EQU 30H
;ASCI_RSULT    EQU 50H
COUNT          EQU 3

hundreds    equ 30h
tens        equ 31h
ones        equ 32h


            ACALL  DELAY_200M
            ACALL  ONETIME_LCDINIT

 STT:
     ACALL  ADC
     ACALL  Bin2Dec
```

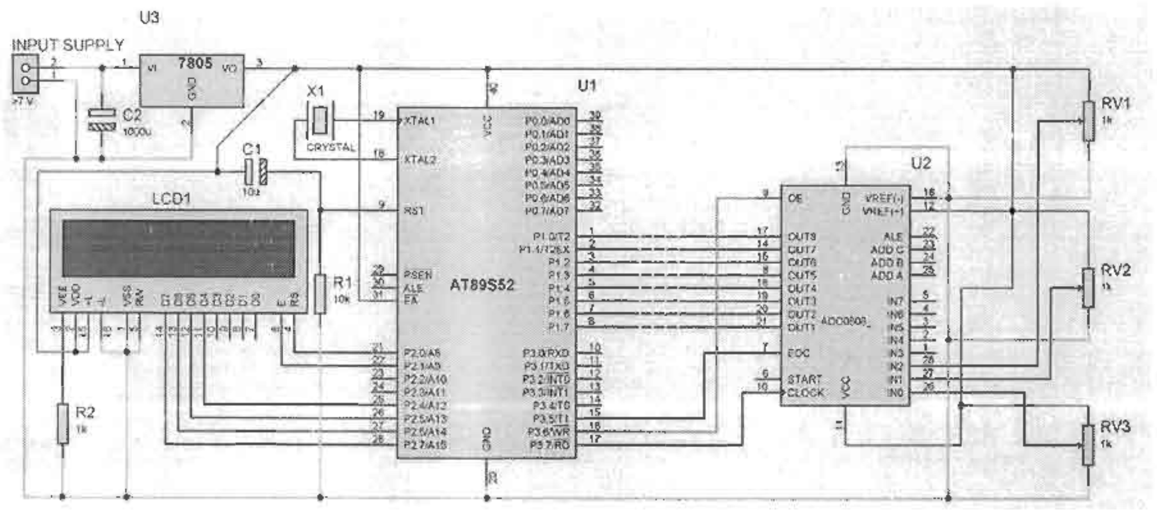
```
        ACALL   DEC_ASCI_CONVRT
        ACALL   DELAY_200M
        ACALL   DELAY_200M
        ACALL   DELAY_200M
        sjmp    STT

;======================================================================
;this subroutine is used to take data from ADC and
;keep to Accumulator
;======================================================================
ADC:    mov A,P0
       nop
       nop
       ret
Bin2Dec:
       mov b,#100d
       div ab
       mov hundreds,a
       mov a,b
       mov b,#10d
       div ab
       mov tens,a
       mov ones,b
       ret

DEC_ASCI_CONVRT:
        MOV R0,#RAM_ADDR
        MOV R2,#3
BACKk:  MOV A,@R0
        ORL A,#30H
        ACALL   DATA_DISPLAY
        INC R0
        DJNZ R2,BACKk
        RET

ONETIME_LCDINIT:
        MOV     A,#38H
        ACALL   COMMAND
        MOV     A,#0EH
        ACALL   COMMAND
        MOV     A,#01H
        ACALL   COMMAND
        MOV     A,#06H
        ACALL   COMMAND
        MOV     A,#85H
        ACALL   COMMAND
        RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
COMMAND:
        ACALL   READY
        MOV     P1,A
        CLR     P2.0
        CLR     P2.1
```

```
    SETB   P2.2
    CLR    P2.2
    RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DATA_DISPLAY:
    ACALL  READY
    MOV    P1,A
    SETB   P2.0
    CLR    P2.1
    SETB   P2.2
    ACALL  DELAY
    CLR    P2.2
    RET
READY: SETB   P1.7
    CLR    P2.0
    SETB   P2.1
BACK:  CLR    P2.2
    ACALL  DELAY
    SETB   P2.2
    JB     P1.7,BACK
    RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DELAY_200M:
    MOV    R7,#30H
L1_DELAY:
    MOV    TMOD,#10
    MOV    TH1,#3CH
    MOV    TL1,#0B0H
    SETB   TR1
AGAIN: JNB    TF1,AGAIN
    CLR    TR1
    CLR    TF1
    DJNZ   R7,L1_DELAY
    RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DELAY: MOV    R3,#50
HERE3: MOV    R4,#255
HERE2: DJNZ   R4,HERE2
    DJNZ   R3,HERE3
    RET
    END
```


U3
INPUT SUPPLY
>7 V
7805
C2
1000u
X1
CRYSTAL
C1
10u
LCD1
R1
10k
R2
1k
U1
AT89S52
U2
ADC0808
RV1
RV2
RV3
1k

[illegible]