MAILAM ENGINEERING COLLEGE
Mailam (po),Villupuram(dt.) Pin:604 304
(Approved by AICTE, New Delhi, Affiliated to Anna University Chennai& Accredited by TCS)
Department of Electrical & Electronics Engineering

SUB CODE/NAME : EE6008 / MICROCONTROLLER BASED SYSTEM DESIGN
YEAR / SEC : IV / A & B

SYLLABUS

UNIT I INTRODUCTION TO PIC MICROCONTROLLER **9**

Introduction to PIC Microcontroller–PIC 16C6x and PIC16C7x Architecture–PIC16cxx–- Pipelining - Program Memory considerations – Register File Structure - Instruction Set - Addressing modes – Simple Operations.

UNIT II INTERRUPTS AND TIMER **9**

PIC micro controller Interrupts- External Interrupts-Interrupt Programming–Loop time subroutine -Timers -Timer Programming– Front panel I/O-Soft Keys– State machines and key switches– Display of Constant and Variable strings.

UNIT III PERIPHERALS AND INTERFACING **9**

I^2C Bus for Peripherals Chip Access– Bus operation-Bus subroutines– Serial EEPROM—Analog to Digital Converter–UART-Baud rate selection–Data handling circuit–Initialization - LCD and keyboard Interfacing -ADC, DAC, and Sensor Interfacing.

UNIT IV INTRODUCTION TO ARM PROCESSOR **9**

ARM Architecture –ARM programmer's model –ARM Development tools- Memory Hierarchy –ARM Assembly Language Programming–Simple Examples–Architectural Support for Operating systems.

UNIT V ARM ORGANIZATION **9**

3-Stage Pipeline ARM Organization– 5-Stage Pipeline ARM Organization–ARM Instruction Execution- ARM Implementation– ARM Instruction Set– ARM coprocessor interface– Architectural support for High Level Languages – Embedded ARM Applications.

TEXT BOOKS:

1. Peatman,J.B., "Design with PIC Micro Controllers"PearsonEducation,3rd Edition, 2004.
2. Furber,S., "ARM System on Chip Architecture" Addison Wesley trade Computer Publication, 2000.

Unit	Part - A				Part - B				Part - C		Month/Year
	Q. No	Page No.	Q. No	Page No.	Q. No	Page No.	Q. No	Page No.	Q. No	Page No.	
01	2	2	30	6	2	9	6	21			Nov'17
					8	23					
	7	3	41	7	2	9	6	21			Apr'18
					15	27					
02	3	29	22	32	1	33	7	41			Nov'17
	10	30	25	32	1	33	2	35			Apr'18
					5	37	10	45			
03	6	60	19	62	1	64	4	77			Nov'17
	21	63									
	8	61	21	63	6	82	31	116			Apr'18
04	23	123			1	124	9	148			Nov'17
	7	120	37	125	3	131	4	135			Apr'18
					16	167	17	168			
05	3	170	35	175	17	215	18	217			Nov'17
	4	170	13	171	2	178	5	190			Apr'18
					11	195					

Prepared By
S. Ganesan
Mr. S. Ganesan, AP/EEE
R. Rajendirakumar
Mr. R. Rajendirakumar, AP/EEE

Checked By
15/6/18
Dr. G. Prem Sunder

Verified By
15/6/18
HOD/EEE

Approved By
Principal

UNIT I PART A

1. What is PIC?

- The PIC [Programmable Interface Controller] was develop by the semiconductor division of General Instruments Inc.
- The first PICs were a programmable, high output current, input/output controller built around RISC (Reduced Instruction Set Code) architecture.
- The first PICs ran efficiently at one instruction per internal clock cycle and the clock cycle was derived from the oscillator divided by 4.

2. Difference between microcontroller and PIC microcontroller. *[Nov'17]*

Microcontroller	**PIC (Peripheral Interface Controller)**	**AVR (Alf & Vegard's RISC)**	**ARM (Advanced RISC Machine)**
Very basic controller (used for the simple applications)	Used to interface more advanced peripherals such as microSD card, RFID scanner etc.		Most advanced controller family (used for Real Time Applications)
Harvard architecture (separate memory spaces for RAM and program memory)			ARM has von Neumann architecture (program and RAM in the same space)
8-bit architecture			ARM has a 16 and/or 32 bit architecture
Limited stack space - limited to 128 bytes	As little as 8 words or less for PIC	8-bits stack space	Stack that a function uses depends on various factors (number and type of arguments to the function, local variables in the function)
Can directly address all available RAM	PIC can only directly address 256 bytes and must use bank switching to extend it	Can directly address all available RAM	
Need multiple clock cycles per instruction		Execute most instructions in a single clock cycle	
Writing a C compiler for these architectures must have been challenging and compiler choice is limited		Have best compiler and application support, including free GCC (GNU) compilers.	

3.Give the various comparison of PIC families.

PIC Family	**Stack Size (words)**	**Instruction word size**	**Number of instructions**	**Interrupt Vectors**
12CXXX/12FXXX	2	12 or 14 bit	33	None
16C5XX/16F5XX	2	12 bit	33	None
16CXXX/16FXXX	8	14 bit	35	1
17CXXX	16	16 bit	58 including hardware multiply	4

4. Give the architecture of PIC 16C6X series. *(Jan'12)*

- PIC16C6X series of microcontroller uses Harvard architecture.
- It uses separate buses for accessing code and data that is four set of buses two each (address and data) for data and code.

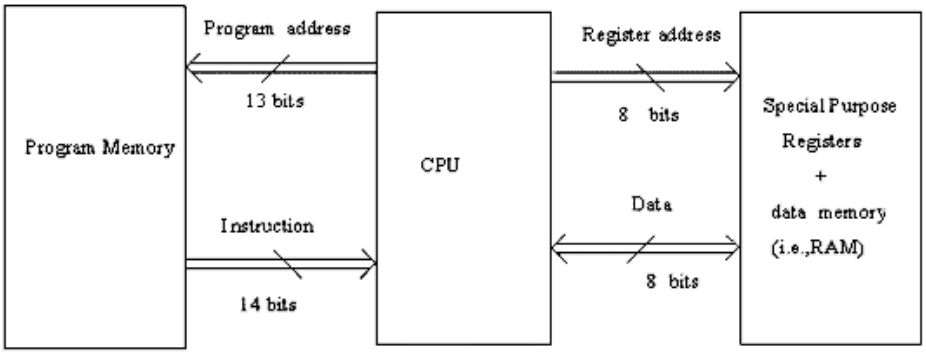
5. Give the Core Features of PICI6CXX Microcontroller.

- High performance RISC CPU.
- Only 35 single word instructions to learn.
- All single cycle instructions except for program branches which are two cycle.
- Operating speed: DC - 20 MHz clock input DC - 200 ns instruction cycle.
- Up to 8K x 14 words of Program Memory, up to 368 x 8 bytes of Data Memory (RAM).
- Direct, indirect, and relative addressing modes.

6. Give the Peripheral Core Features of PIC16CXX Microcontroller.

- Timer0: 8 bit timer/counter with 8 bit prescaler
- Timer1: 16 bit timer/counter with prescaler, can be incremented during sleep through external crystal/clock
- Timer2: 8 bit timer/counter with 8 bi period register, prescaler and postscaler.
- Capture Module is 16 bit maximum resolution is 12.5 ns
- Compare Module is 16 bit maximum resolution is 200 ns
- PWM Module maximum resolution is 10 bit
- 8 bit multichannel Analog to Digital converter
- Synchronous Serial Port (SSP) with SPI and I^2C.

7. Illustrate the CPU —Harvard architecture of P/C microcontroller. [OR] **Draw the program memory organization of PIC16C6x microcontroller.** *[Apr'18]*



8. What are the features of Harvard architecture?

- The separate instruction and data buses of the Harvard architecture 14-bit wide instruction word with the separate 8-bit wide data.
- The two sanction pipeline allows all instructions to execute in a single cycle, (eg. for program branches which require two cycles).
- A total of 35 instructions (reduced instruction set) are available

.

9. Define instruction pipelining.

- In many CPUs these two steps are done one after the other, first the CPU fetches and then it executes.

- However, program memory has its own address and data bus, separate from data memory (a Harvard structure).
- Then there is no reason why a CPU cannot be designed so that while it is executing one instruction, it is already fetching the next.
- This is called pipelining.

10. Differentiate between Harvard architecture and von Neumann architecture.

- PICI6CXX uses Harvard architecture, in which, program and data are accessed from separate memories using separate buses.
- This improves bandwidth over traditional von Neumann architecture in program and data are fetched from the same memory using the same bus.

11. How P1C16CXX is accessed for data memory or register files?

- The PICI6CXX can directly or indirectly address its register files or memory.
- All special function registers, including the program counter, mapped in the data memory.
- The P1C16CXX has an orthogonal (symmetrical instruction set that makes it possible to carry out any operation on any register using addressing mode.

12. Why PICI6CXX is said to be an orthogonal?

- The PICI6CXX has an orthogonal (symmetrical) instruction set that makes it possible to carry out any operation on any register using any addressing mode.
- This symmetrical nature and lack of 'special optimal situations' makes programming with the PIC I6CXX simple yet efficient.

13. What is the difference between the presence or absence of A in the microcontroller?

- The presence of A indicates the brown-out reset feature, which causes a reset of the PIC when the Power Supply voltage drops below 4.0v.

14. What is the usage of working register?

- The working register (W) is used by many instructions as the source of an operand.
- It may also serve as the destination for the result of the instruction execution, similar to that of the accumulator in many other microcontroller.

15. Define the register file structure in Microcontroller. Classify it?

- Register file is used to denote the location than an instruction can access through an address. Register file is classified into
  - General – purpose register file
  - Special purpose register file.

16. What is general purpose register file?

- The general purpose register file is another name for the microcontroller's RAM.
- Data can be written to each 8-bit location, updated and retrieved any number of times.

17. What is special purpose register file?

- The special purpose register file contains input and output ports as well as the control registers used to establish each bit of a port as either an input or e output.
- It contains registers that provide the data input and data output to variety of resources on the chip, such as the timers, the serial ports, and 1 analog to digital converter.

18. Write the significant of brown out reset [BOR] mode? *[Apr'17]*

- When the power supply falls below a certain voltage, it causes PIC to reset. This is called brown out to reset mode.

19. Explain about watchdog timer? *(Jan'11)*

- It is a timer circuit which monitors the continuous functioning of processor with respect to time and prevents the endless loop hanging condition.

20. What is meant by PCLATH? Give its use.

- It is program counter latch.
- Any write to PCL will cause the contents of PCLATH to be transferred to PC higher locations.

21. Give the importance of MCLR?

- It is called master clear pin.
- This pin is used to reset the PIC.

22. What are register present in the CPU register in the microcontroller

The PIC PIC16F877 Microcontroller has the following registers.

- Working Register-W (Similar to Accumulator)
- Status Register
- FSR – File Select Register (Indirect Data memory address pointer)
- INDF
- Program Counter

23. Write about the Status Register of PIC Microcontroller. *[Nov'16]*

It indicates the program status after the arithmetic logic unit calculation are carried out, with its bit position in the status register.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C

D0-carry, D1-Digit carry, D2-Zero, D3-power down
D4-time out D5-Register bank select D6- Direct addressing mode
D7-Inidrect addressing mode.

24. What is the role of status register?

- STATUS register contains arithmetic status ALU (C, DC, Z), RESET status (TO, PD) and bits for selecting of memory bank (IRP, RP1, RPO).

25. What is the use of RP0 and RP1 in Register file?

- RP0 is used as selection of bit in register bank.
- If the RP0 is in STATUS register is 0, then the effective address feeds the H14 in to the register in the Bank 0.
- If the RP0 bit is 1, then the effective address feeds the H94 in to the register in the Bank 1.

26. What are the components of Program memory Organization 16CXX?

- The Program Counter,
- The Stack
- The actual program memory.

27. What is the need of FSR?

- FSR (File Select Register) is used as a pointer in the indirect addressing mode.

28. What is RISC? *(Jun'14, Jul'13)*

- RISC [*Reduced Instruction Set Computer]* is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

29. What are the benefits of having RISC architecture? *[Apr'17]*

- The performance of RISC processors is often two to four times than that of CISC (Complex Instruction Set Computing) processors because of simplified instruction set.
- This architecture uses less chip space due to reduced instruction set. This makes to place extra functions like floating point arithmetic units or memory management units on the same chip.
- The per-chip cost is reduced by this architecture that uses smaller chips consisting of more components on a single silicon wafer.
- RISC processors can be designed more quickly than CISC processors due to its simple architecture.
- The execution of instructions in RISC processors is high due to the use of many registers for holding and passing the instructions as compared to CISC processors.

30. List out all the addressing Modes in PIC Microcontroller. *(Jan '12, Nov'16, Nov'17)*

Addressing is defined as how the operands are specified in the instruction. They are

- Immediate addressing mode
- Register operand addressing mode
- Memory operand addressing mode
  1. Direct Addressing Mode
  2. Indirect Addressing mode

31. What do you mean by direct addressing mode?

- It uses 7 bits of the instruction and the 8th bit from RP.
- It directly give the address where the data is present that is the address of the operand is given in the instruction.

32. What is instruction pipelining? *(Jan '11)*

- It allows the CPU to fetch and execute at the same time while executing one instruction, CPU will fetch next instruction to be executed.

33. What is indirect addressing mode?

- Indirectly addressing the memory used in FSR and INDF instruction.
- Here the operand is specified indirectly in the instruction.

34. What is Access bank in PIC18?

- It is the default bank which is invoked when power up.
- It is divided into two equal sections of 128 bytes, which is given to GPR and SFR.

35. What is I/O port of PIC? *(Jan'14)*

- I/O port is used to get and send the data from/to external devices.
- Some I/O pins have multi-functions.

36. Mention the guidelines suggest by Microchip Technology will be used for writing assembly language code.

- Write the instruction mnemonics in lowercase (eg. xorwf)
- Write special register names, RAM variables names and bit names in uppercase (eg. STATUS, RP0)
- Write instruction and subroutine labels in mixed case (eg. Mainline, LoopTime)

37. What are the groups of instruction set in PIC micro controller? *(Jan'13)*

1. Bit oriented Instructions
2. Instructions using a literal value
3. Byte oriented instructions
4. Table read and write instructions
5. Control instructions using branch and call.

38. Write any four instructions of PIC microcontroller and state in a line the operation performed. *(Jan'14)*

MOVLW 25H; WREG=25
ADDLW 0X34; ADD 34H TO WREG
ADDLW; ADD 11H TO WREG
ADDLW; W=W+12H=7CH

39. Using the instruction of PIC microcontroller, convert BCD to Hex. *(Jan'13)*

MOVLW 0X54
ADDLW 0X87
DAA.

40. How is the internal RAM in PIC microcontroller accessed by indirect addressing? *(Jun'12)*

LFSR 0, 0x30
LFSR 1, 0x40

41. Write the operation carried out when these instructions executed by PIC.
BTFSS f, b
BCF f, b. *[Apr'18]*

Operations carried out,

- BTFSS f, b – Test b of f
- BCF f, b – to set bit b in location f

PART B

1. Briefly explain the features present in PIC microcontroller?

- The PIC [Peripheral Interface Controller] microcontroller have been introduced by Microchip Technology, USA.

- Originally this was developed as a supporting device for PDP computers to control its peripheral devices and therefore named as PIC, Peripheral Interface Controller.
- The focus will be on the PIC16C6x/7x family.

PIC16C6x and PIC16C7x Microcontroller:

- PIC16C7x family has an enhancement of Analog to Digital converter capability.
- These microcontroller are available with a range of capabilities packaged in both dual in-line (DIP) packages and surface - mount packages.
- These are available in 28 pin DIP, 40 pin DIP, 44 pin surface mount package.
- Some of PIC controllers contain the letter A in their number.
- The presence of A indicates the brown-out reset feature, which causes a reset of the PIC when the Power Supply voltage drops below 4.0v.
- PIC16C7x is a family with low cost, high performance, CMOS, fully static 8 – bit microcontroller with Integrated ADC.
- The PIC16C7x has enhanced core features of eight – level deep stack and multiple internal – external interrupt sources.
- It has special features to reduce external components thus reducing cost, enhancing system reliability and reducing power conception.
- The PIC16C74A devices has 192 bytes of RAM and 4K x 14 EPROM.
- It has 33 I/O pins and the other peripheral features as: 3 Timer/Counters, 2 Capture/Compare/PWM [CCP] modules and 2 serial ports.
- The Synchronous Serial Port [SSP] can be configures as either a 3 – wire Serial Peripheral Interface [SPI] or the two – wire Inter – Integrated Circuit [I^2C] bus.
- The USART is also known as the Serial Communication Interface [SCI].
- An 8 – bit parallel slave port and 8 – channel high speed with 8 – bit A/D converter are provided.

PIC 16C6X/7X family members:

Part Number	Pins	I/O Pins	EPROM (K bytes)	RAM (bytes)	ADC Channels	USART	CCP Modules
PIC 16C62A	28	22	2	128	0	0	1
PIC 16C63	28	22	4	192	0	1	2
PIC 16C64A	40/44	33	2	128	0	0	1
PIC 16C65A	40/44	33	4	192	0	1	2
PIC 16C72	28	22	2	128	5	0	1
PIC 16C73A	28	22	4	192	5	1	2
PIC 16C74A	40/44	33	4	192	8	1	2

SALIENT FEATURES

- **Speed:**
  When operated at its maximum clock rate a PIC executes most of its instructions in 0.2 μs or five instructions per microsecond.

- **Instruction set Simplicity:**
  The instruction set is so simple that it consists of only just 35 instructions.

- **Integration of operational features:**
  Power-on-reset (POR) and brown-out protection ensure that the chip operates only when the supply voltage is within specifications. A watch dog timer resets the PIC if the chip malfunctions or deviates from its normal operation at any time.

- **Programmable timer options:**
  Three timers can characterize inputs, control outputs and provide internal timing for the program execution.

- **Interrupt control:**
  Up to 12 independent interrupt sources can control when the CPU deal with each sources.

- **Powerful output pin control:**
  A single instruction can select and drive a single output pin high or low in its 0.2μs instruction execution time. The PIC can drive a load of up to 25μA.

- **I/O port expansion:**
  With the help of built in serial peripheral interface the number of I/O ports can be expanded. EPROM/DIP/ROM options are provided.

2. Explain clearly about the architecture of PIC Microcontroller? *[Nov'16, Apr'17]* [OR] **Draw and explain about the architecture of PIC microcontroller.** *[Nov'17]* [OR] **Explain the architecture of PIC16C6x microcontroller with neat block diagram.** *[Apr'18]*

- PIC 16C74A is a member of PIC 16C7X family of 8 – bit microcontroller.
- It is fabricated with CMOS technology and packed in 40 – pin DIP or 44 – pin surface mounted packages.

- **High speed performance RISC CPU**
  - Only 35 single word instructions.
  - Operating speed: DC – 20 MHz clock input
    DC – 200 ns instruction cycle
  - Addressing modes: Direct, Indirect and relative
  - 4K X 14 words of program memory (EPROM)
  - 192 X 8 bytes of Data memory (RAM)
  - In – circuit Serial Programming.
  - 12 Interrupt sources.
  - Eight level deep hardware stack

- **Peripheral features**
  - Watchdog Timer (WDT) with its own on-chip RC oscillator for reliable operation.
  - 3 Timers - Timer0, Timer 1 and Timer 2.
    Timer0 (TMR0) – 8 bit timer/counter with 8 bit prescaler.
    Timer1 (TMR1) – 16 bit timer/counter with prescaler can be incremented during sleep through external crystal/Clock.
    Timer2 (TMR2) – 8 bit timer/counter with 8 bit period register, prescaler and post scaler.
  - Power – up Timer (PWRT) and Oscillator Start-up Timer (OST)
  - 8 - bit multi-channel Analog-to-Digital converter
  - One USART /SCI port with 9-bit address detection.
  - 8 bit Parallel Slave Port (PSP)
  - Synchronous Serial Port (SSP) with SPI (Master mode) and I^2C (Master/Slave)
  - Capture, Compare, PWM (CCP) modules

- **Special Microcontroller features**
  - Power – on – Reset (POR)
  - Device Reset Timer
  - Selectable oscillator options
  - Programmable code-protection
  - Power saving SLEEP mode

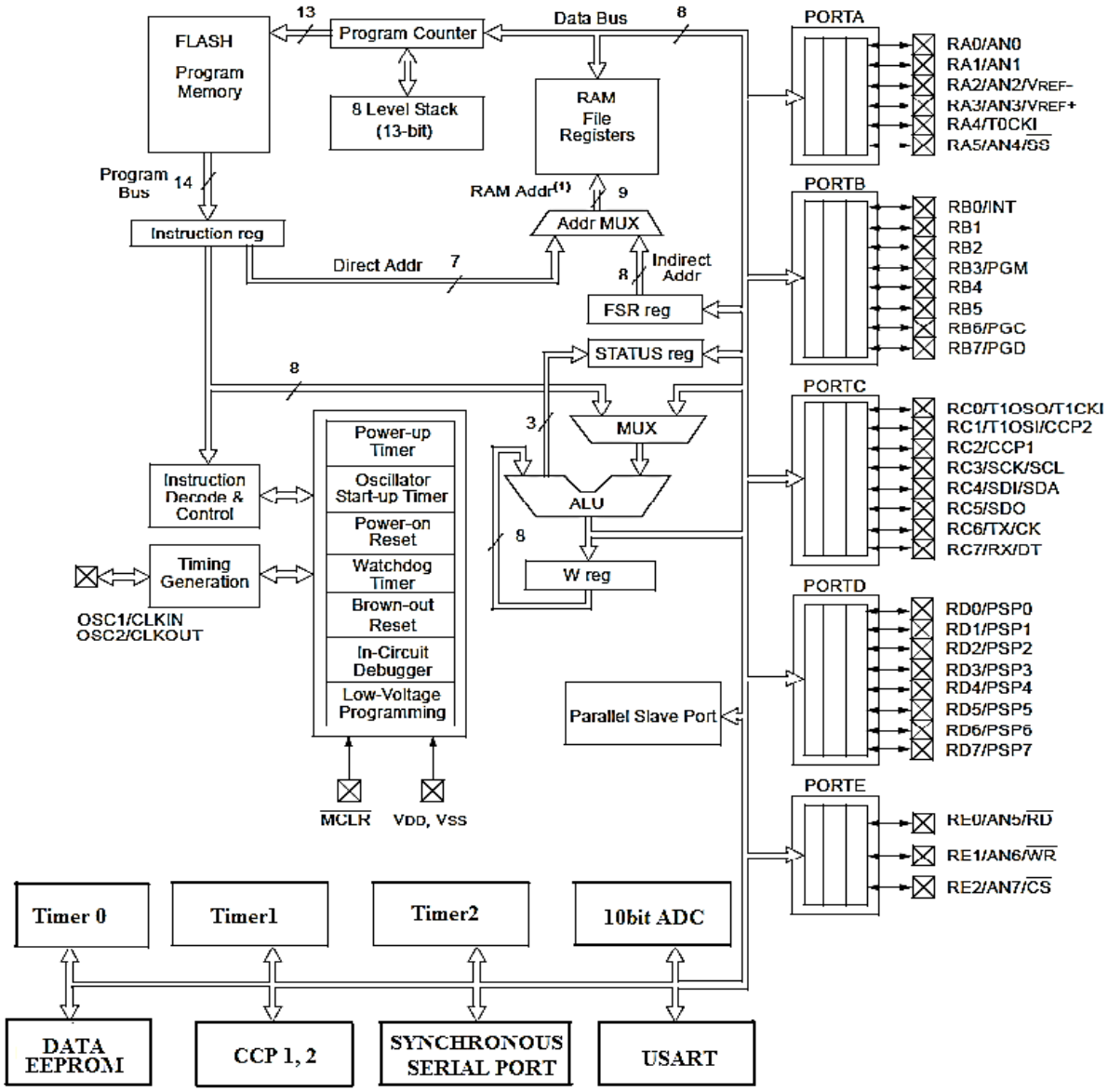
- **CMOS Technology**
  - Low-power, high-speed CMOS EPROM/ROM (FLASH) technology
  - Fully static design
  - Low – power consumption: < 2 mA at 5V, 4MHz,
    15 μA typical at 3V, 32 kHz
  - Low – power Standby Mode: <0.6 μA
  - Wide operating voltage range: 2.5V to 6.0V
  - Commercial, Industrial and Extended temperature ranges.

Pin Configuration:

- PIC 16C74A has 40 – pins DIP.
- Out of 40 pins
  - 33 pins are used as I/O pins.

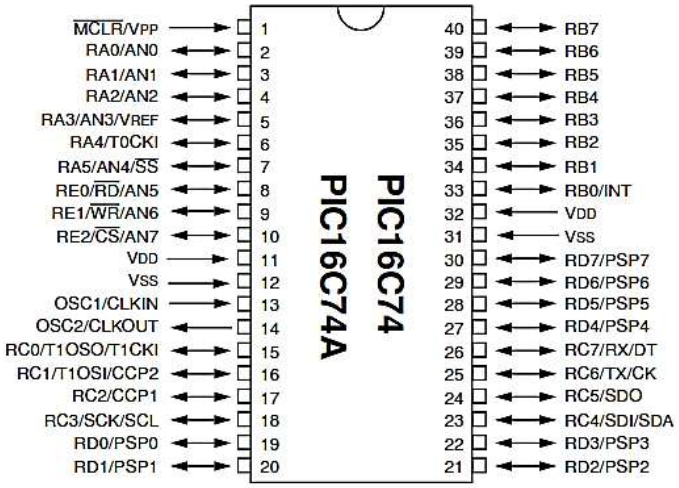
Port	Alternative Use	I/O pins
PORT A	A/D converter inputs	6
PORT B	External interrupt sources	8
PORT C	Serial Ports, Timer I/O	8
PORT D	Parallel Slave port	8
PORT E	A/D converter inputs	3

  - 4 pins are needed to supply power [V_{SS}, V_{DD}, V_{SS}, V_{DD}]
  - 2 pins are connected to a 4 MHz crystal [OSC 1, OSC 2]
  - 1 Master Clear [MCLR] pin.



Block diagram of PIC 16F87X Microcontroller

Pin Layout:



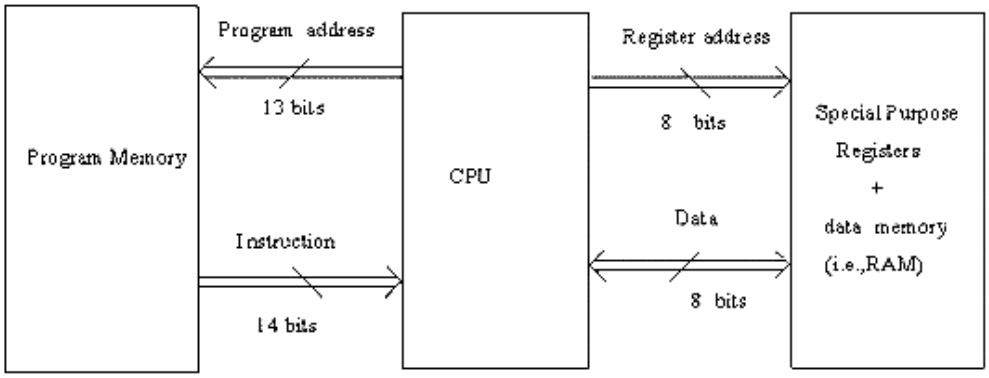
Pin Description:

Pin Name	DIP Pin#	PLCC Pin#	QFP Pin#	I/O/P Type	Buffer Type	Description
OSC1/CLKIN	13	14	30	I	ST/CMOS(4)	Oscillator crystal input/external clock source input.
OSC2/CLKOUT	14	15	31	O	—	Oscillator crystal output. Connects to crystal or resonator in crystal oscillator mode. In RC mode, OSC2 pin outputs CLKOUT which has 1/4 the frequency of OSC1, and denotes the instruction cycle rate.
$\overline{\text{MCLR}}$/VPP	1	2	18	I/P	ST	Master clear (reset) input/programming voltage input. This pin is an active low reset to the device.
						PORTA is a bi-directional I/O port.
RA0/AN0	2	3	19	I/O	TTL	Analog input0
RA1/AN1	3	4	20	I/O	TTL	Analog input1
RA2/AN2	4	5	21	I/O	TTL	Analog input2
RA3/AN3/VREF	5	6	22	I/O	TTL	Analog input3/VREF
RA4/T0CKI	6	7	23	I/O	ST	Can also be selected to be the clock input to the Timer0 timer/counter. Output is open drain type.
RA5/AN4/SS	7	8	24	I/O	TTL	Analog input4 can also be the slave select for the synchronous serial port.
						PORTB is a bi-directional I/O port. PORTB can be software programmed for internal weak pull-up on all inputs.
RB0/INT	33	36	8	I/O	TTL/ST(1)	RB0/INT can also be selected as an external interrupt pin.
RB1	34	37	9	I/O	TTL	
RB2	35	38	10	I/O	TTL	
RB3	36	39	11	I/O	TTL	
RB4	37	41	14	I/O	TTL	Interrupt on change pin.
RB5	38	42	15	I/O	TTL	Interrupt on change pin.
RB6	39	43	16	I/O	TTL/ST(2)	Interrupt on change pin. Serial programming clock.
RB7	40	44	17	I/O	TTL/ST(2)	Interrupt on change pin. Serial programming data.
						PORTC is a bi-directional I/O port.
RC0/T1OSO/T1CKI	15	16	32	I/O	ST	RC0/T1OSO/T1CKI can also be selected as a Timer1 oscillator output or a Timer1 clock input.
RC1/T1OSI/CCP2	16	18	35	I/O	ST	RC1/T1OSI/CCP2 can also be selected as a Timer1 oscillator input or Capture2 input/Compare2 output/ PWM2 output.
RC2/CCP1	17	19	36	I/O	ST	RC2/CCP1 can also be selected as a Capture1 input/ Compare1 output/PWM1 output.
RC3/SCK/SCL	18	20	37	I/O	ST	RC3/SCK/SCL can also be selected as the synchronous serial clock input/output for both SPI and I2C modes.
RC4/SDI/SDA	23	25	42	I/O	ST	RC4/SDI/SDA can also be selected as the SPI Data In (SPI mode) or data I/O (I2C mode).
RC5/SDO	24	26	43	I/O	ST	RC5/SDO can also be selected as the SPI Data Out (SPI mode).
RC6/TX/CK	25	27	44	I/O	ST	RC6/TX/CK can also be selected as Asynchronous Transmit or USART Synchronous Clock.
RC7/RX/DT	26	29	1	I/O	ST	RC7/RX/DT can also be selected as the USART Synchronous Data.

RD0/PSP0 RD1/PSP1 RD2/PSP2 RD3/PSP3 RD4/PSP4 RD5/PSP5 RD6/PSP6 RD7/PSP7	19 20 21 22 27 28 29 30	21 22 23 24 30 31 32 33	38 39 40 41 2 3 4 5	I/O I/O I/O I/O I/O I/O I/O I/O	ST/TTL**(3)** ST/TTL**(3)** ST/TTL**(3)** ST/TTL**(3)** ST/TTL**(3)** ST/TTL**(3)** ST/TTL**(3)** ST/TTL**(3)**	PORTD is a bi-directional I/O port or parallel slave port when interfacing to a microprocessor bus.
RE0/RD/AN5 RE1/WR/AN6 RE2/CS/AN7	8 9 10	9 10 11	25 26 27	I/O I/O I/O	ST/TTL**(3)** ST/TTL**(3)** ST/TTL**(3)**	PORTE is a bi-directional I/O port. RE0/RD/AN5 read control for parallel slave port, or analog input5. RE1/WR/AN6 write control for parallel slave port, or analog input6. RE2/CS/AN7 select control for parallel slave port, or analog input7.
VSS	12, 31	13, 34	6, 29	P	—	Ground reference for logic and I/O pins.
VDD	11, 32	12, 35	7,28	P	—	Positive supply for logic and I/O pins.
NC	—	1,17,28,40	12, 13, 33, 34		—	These pins are not internally connected. These pins should be left unconnected.

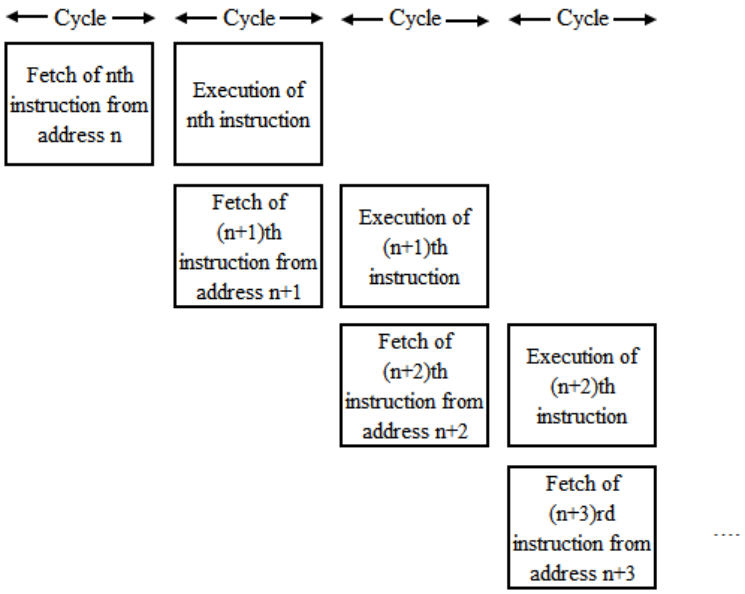
3. Explain in detail about the pipeline structure of PIC controller?

- The PIC16C6X/7X family of microcontroller uses a Harvard architecture to achieve an exceptionally fast execution speed for a given clock rate.
- The figure shows, instructions are fetched from program memory using buses that are distinct from buses used for accessing variables in data memory, I/O ports, etc.



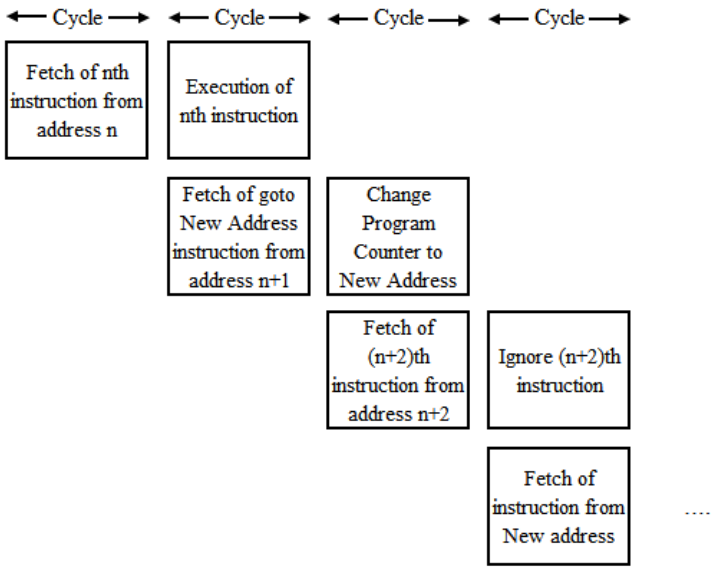
Pipeline structure

- Every instruction is coded as a single 14 bit word and fetched over a 14 bit wide bus.
- Consequently, instructions are fetched from successive program memory locations, a new instruction is fetched every cycle.
- The CPU executes each instruction during the cycle following its fetch, pipelining instruction fetches and instruction executions to achieve the execution of one instruction every cycle.
- The figure shows, that each instruction requires two cycles (a fetch cycle followed by an execute cycle).



Pipelining of instructions fetched from successive address

- The overlapping of the execute cycle of one instruction with the fetch cycle of the next instruction leads to the execution of a new instruction every cycle.
- This lockstep progression can be overcome by branch operation as shown in the figure.
- An instruction is fetched during the second cycle, goto New Address whose work is to change the normal flow of instruction fetches from one address to the next address.
- During the third cycle, the CPU carries out the sequential fetch from the address n+2.
- At the end of that third cycle, the CPU executes the goto New Address instruction by changing the program counter to New Address instead of simply incrementing it to n+3.
- At fourth cycle, it fetches the instruction at New Address, it ignores the instruction automatically fetched from address n+2 (this instruction is located in the program immediately after the $(n+1)^{th}$ goto New Address instruction.
- But it will be never executed immediately after the execution of that $(n+1)^{th}$ goto New Address instruction.

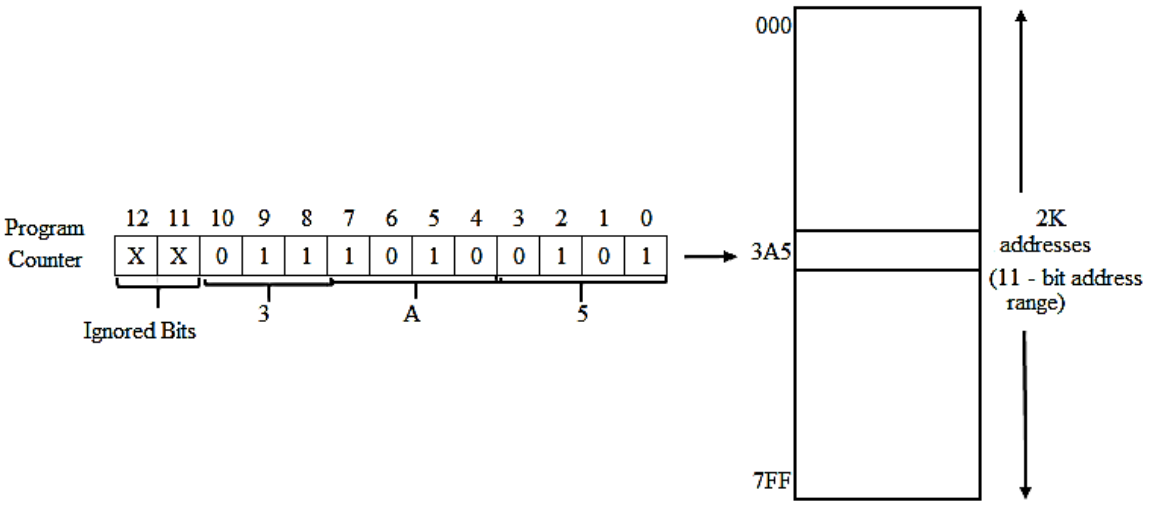


Introduction of extra cycle for a goto instruction

4. Explain various Memory Organization of PIC microcontroller. *[Nov'16, Apr'17]*

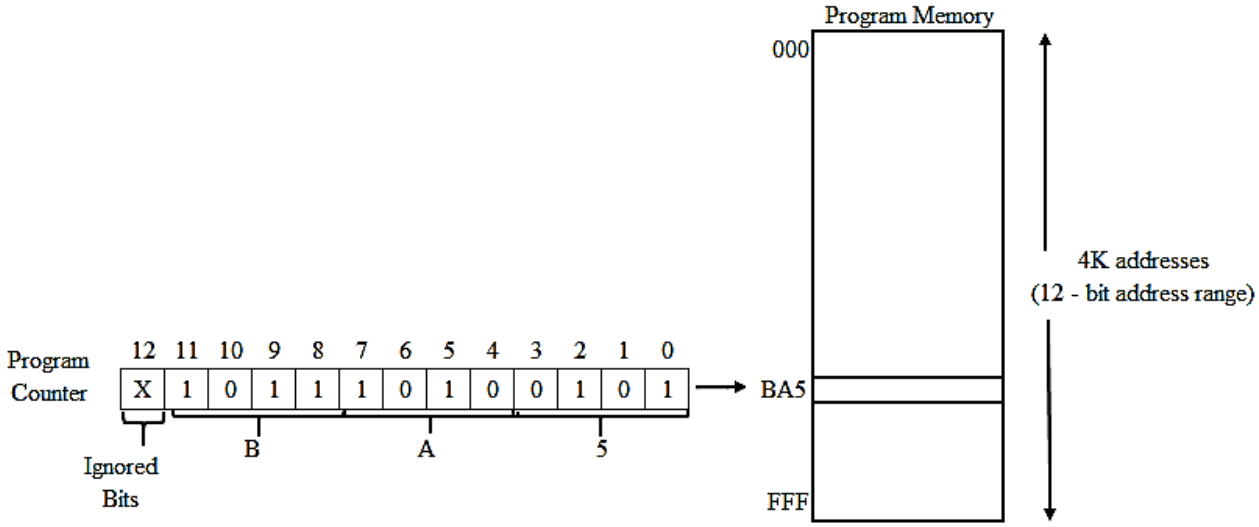
Program Memory:

- Each members of the PIC16C6X/7X family of microcontrollers includes either 2K (2048) or 4K (4096) address of program memory.
- As shown in figure a program memory of 2K addresses needs only an 11 – bit program counter to access any address (2^{11} = 2048 = 2K).



Program Memory accesses for PIC parts having 2K of program memory

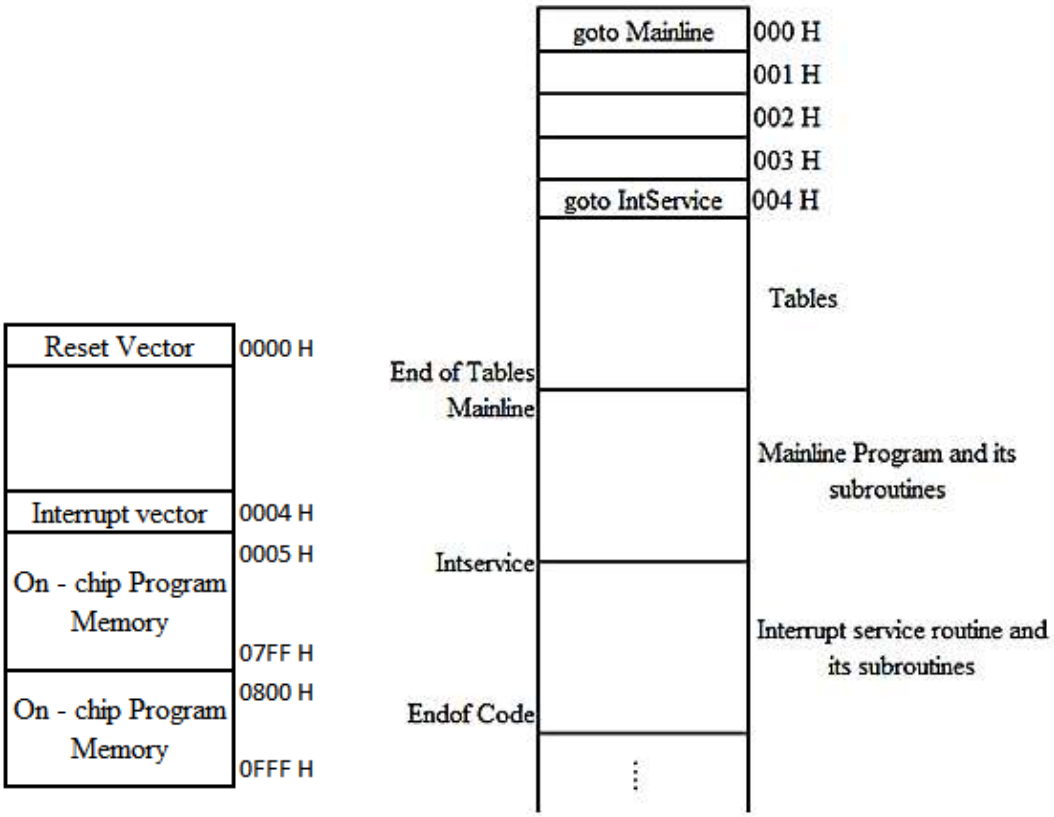
- A program memory of 4K addresses needs a 12 – bit program counter.



Program memory accesses for PIC parts having 4K of program memory

- This PIC family actually uses a 13 – bit program counter allowing for extending the family to an 8K program memory without changing the CPU structure.
- For the 4K and 2K parts, the upper bit or are simply ignored during fetches from program memory.
- Two addresses in the program memory space are treated in CPU, the CPU starts up from its reset state and its program counter is automatically cleared to zero.
- This is illustrated in figure above with the content of address 000H being a goto Mainline instruction.
- The second special address 004H is automatically loaded into the program counter when an interrupt occurs.

- A goto IntService instruction can be assigned to this address to cause the CPU to jump to the beginning of the interrupt service routine, located anywhere in memory space.
- The program code can be simplified somewhat if any tables that are created are assigned to addresses in the range 005H – 0FFH.
- For most of the applications these 250 locations provide more enough.



Program memory allocation

- The Mainline program begins immediately following the tables.

Program Structure:

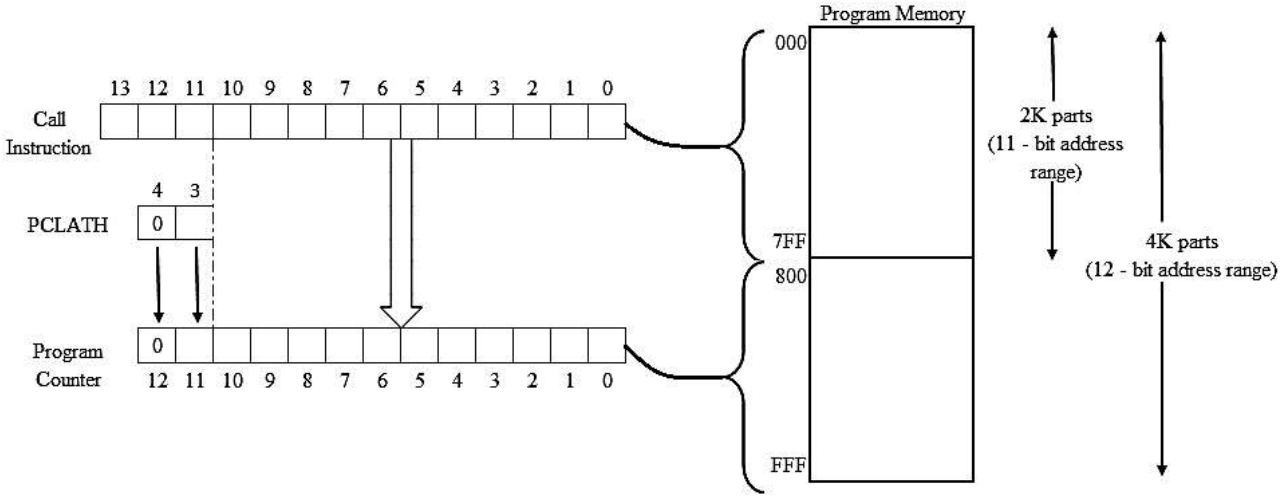
```
Mainline
        call Initial        ; Initialize everything
Mainloop
        call Task1          ; Deal with Task1
        call Task2          ; Deal with Task2
        .
        .
        .
        call  LoopTime      ; Force looptime to a fixed value
        goto MainLoop       ; Repeat
```

- PIC's timers will be used to make the time it take to traverse MainLoop a fixed amount of time for 10 ms.
- If Task1 is supposed to toggle an LED indicator lamp for half second, then it need only increment a scale of 50 counter once per call of Task1.

$$\frac{500}{10} = 50$$

- The LED is toggled for each complete cycle of this counter.
- The mainline program begins execution when the PIC comes out of reset.
- It continues running until one of the PIC interrupt sources requests service.

- At that point the execution of the MainLine code is temporarily suspended.
- The CPU begins the execution of the interrupt services routine by automatically loading the program counter with 004H.
- At the completion of the interrupt service routine, the CPU returns to where it left off in the mainline program.
- If all the program code for the tables, the mainline program, its subroutines, the interrupt service routine and its subroutines take up less than 2K words of instruction.
- This 2K word constraint on code length is that EndofCode ≤ 2047D = 7FFH.
- It is a result of having a one word subroutine call instruction.



- As shown in the figure, bits 10 to 0 of the call instruction are loaded into the program counter.
- Bit 4 and 3 of a special register called PCLATH (Program Counter Latch) are loaded into bits 12 and 11 of the program counter.
- If the program memory is less than 2048 (2K), PCLATH can be left initialized to 00H and then the 11 address bits in the call instruction will identify the starting address of any subroutine located up to address 7FF H.
- For program larger than this, bit 3 of PCLATH is set or cleared each time a subroutine is called.
- The goto instruction is also has an 11 – bit address field requires an identical treatment.

Data Memory:

- The data memory of PIC 16F8XX is partitioned into multiple banks which contain the general purpose registers and the Special function Registers (SFRs).
- The bits RP1 and RP0 bits of the status register are used to select these banks.
- Each bank extends upto 7FH (128 Bytes).
- The lower bytes of the each bank are reserved for the Special Function Registers.
- Above the SFRs are general purpose registers implemented as static RAM.

5. Write short notes on register file structure in PIC controller?

In PIC Microcontrollers the Register File consists of two parts namely

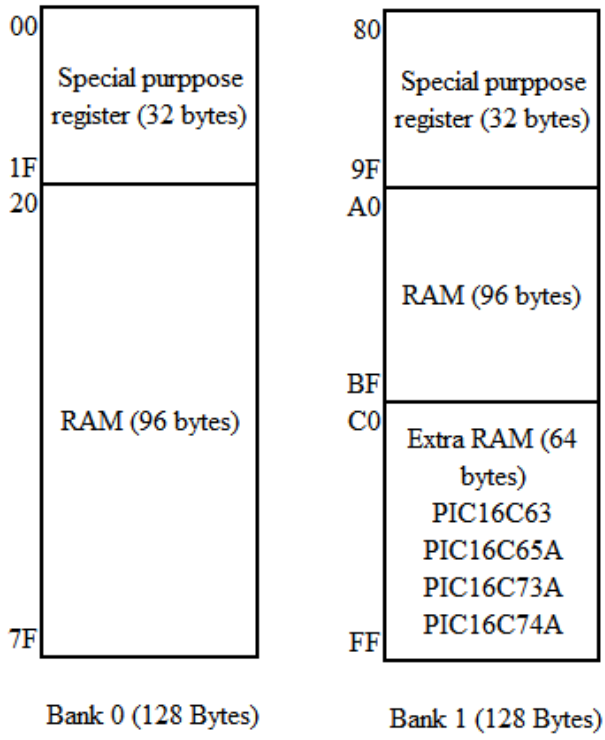
a) General Purpose Register File
b) Special Purpose Register File

General Purpose Register File:

- The general purpose register file is also said to be microcontroller's RAM.
- Data can be written to each 8-bit location, updated and retrieved any number of times.

Special Purpose Register File:

- The special purpose register file consists of input ports, output ports and control registers used to configure each 8-bit port either as input or output.
- It contains registers that provide the data input and data output to a chip resources like Timers, Serial Ports and Analog to Digital converter.
- Also the registers that contains control bits for selecting the mode of operation of a chip resource and also enabling or disabling its operation.
- It has registers containing status bits, which denote the state of one of these chip resources say serial data transfer complete.
  - The register file structure is shown in figure with addresses that the 8 – bit range from 00H – FFH.



RP0	RP1	Bank
0	0	Bank 0
0	1	Bank 1

CPU Registers:

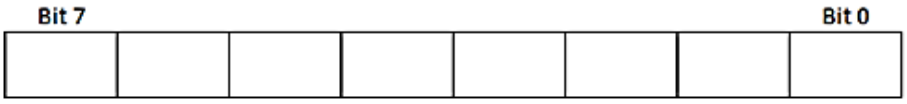
- The CPU registers are used in the execution of the instruction of the PIC microcontroller.
- The **PIC16F877** Microcontroller has the following registers.
  1. Working Register-W (Similar to Accumulator)
  2. Status Register
  3. FSR – File Select Register (Indirect Data memory address pointer)
  4. INDF
  5. Program Counter

Working Register (W):

- Working Register is used by many instructions as the source of an operand.
- It also serves as the destination for the result of instruction execution and it is similar to accumulator in other microcontrollers and microprocessors.

Status Register: [Location: 03H, 83H]

- This is an 8-bit register which denotes the status of ALU after any arithmetic operation
- Also it RESET status and the bank select bits for the data memory.



Work Register (W)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C

Status Register

C: Carry/ $\overline{\text{borrow}}$ bit

- The 'C' bit is set when two 8 bit operands are added or subtracted, a 9-bit result can occurs. This 9th bit is placed in the Carry/ $\overline{\text{borrow}}$ bit.

DC: Digit Carry/ $\overline{\text{borrow}}$ bit

- It is the auxiliary carry bit.
- DC is set (DC = 1) as a result of the carry from the bit – 3 to the bit – 4 position.
- This digit carry bit is useful when adding BCD encoded numbers.

Z: Zero bit

- The Z is affected by the execution of arithmetic or logic instructions.
- If the result is zero, Z = 1
  Otherwise, Z = 0.

$\overline{PD}$: Reset Status bit (Power-down mode bit)

- $\overline{PD}$ = 1; coming out of a power – on reset or after the execution of a CLRWDT instruction.
- $\overline{PD}$ = 0; after wakeup from the SLEEP condition.

$\overline{TO}$: Reset Status bit (time- out bit)

- $\overline{TO}$ = 1; coming out of a power – on reset or after the execution of a CLRWDT instruction or after a non – watch dog timer wakeup from SLEEP condition.
- $\overline{TO}$ = 0; after a watchdog timer reset has occurred.

RP0: Register bank Select

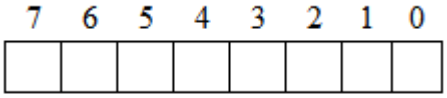
- The Register bank select bit RPO is used to select either bank 0 or bank 1.
- When RP0 = 0, select Bank 0.
  RP0 = 1, select Bank 1.

RP1: Direct addressing mode
IRP: Indirect addressing mode.

```
Example:  bcf STATUS, RPO      ;     Select bank 0
          bsf STATUS, RPO      ;     Select bank 1.
```

FSR – (File Select Register):

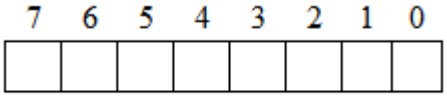
- It is the pointer used for indirect addressing.
- In the indirect addressing mode the 8-bit register file address is first written into FSR.
- It is a special purpose register that serves as an address pointer to any address throughout the entire register file.



FSR [Location: 04H,84H]

INDF – (Indirect File):

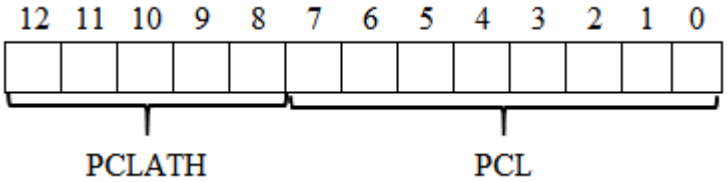
- It is not a physical register addressing, but this INDF will cause indirect addressing.
- Any instruction using the INDF register actually access the register pointed to by the FSR.



INDF [Location: 00H, 80H]

Program Counter:

- PIC PIC16F877A has a 13 bit program counter.
- PCL (Program Counter Lower) is the lower 8-bits of the PC. [Location: 02H,82H]
- PCLATH (Program Counter Latch) is the write buffer for the upper 5 bits of the PC. [Location: 0AH,8AH]



- PCLATH can be read from or written to without affecting the PC.
- The upper 3 bits of PCLATH remain zero.
- It is only when PCL is written to that PCLATH is automatically written into the PC at the same time.

Parallel I/O Ports:

- Most of the PIC16CX/7X family controllers have 33 Input/ Output (I/O) lines.
- Five I/O ports - PORT A, PORT B, PORT C, PORT D and PORT E.

PORT A:

- Port A is a 6-bit wide bi-directional port.
- Its data direction register is TRISA.
- If TRISA bit = 1; will make the corresponding PORT A Pin an input.
  TRISA = 0; a bit will make the corresponding pin as an output.

PORT B:

- Port B is an 8-bit wide, bi-directional port.
- Four of the PORT B pins (RB_7 – RB_4) have an interrupt – on – change feature.
- Only the pins configured as inputs, can cause this interrupt to occur.

PORT C:

- Port C is an 8-bit wide, bidirectional port. Bits of the TRISC Register determine the function of its pins. Similar to other ports, a logic one 1 in the TRISC Register configures the appropriate port pin as an input.

PORT D:

- Port D is an 8-bit wide bi-directional port.
- In addition to I/O port, Port D also works as 8-bit parallel slave port or microprocessor port.
- When control bit PSPMODE (TRISE: 4) is set.

PORT E:

- It is a 3-bit bi-directional port.
- Port E bits are multiplexed with analog inputs of ADC and they serve as control signals ($\overline{RD}$,$\overline{WR}$,$\overline{CS}$) for parallel slave port mode of operation.

6. **Explain the different PIC addressing modes?** *[Apr'17, Nov'17]* [OR] **With examples, explain the addressing modes of PIC16C6x microcontroller.** *[Apr'18]*

The PIC microcontroller supports

- Immediate addressing mode
- Register operand addressing mode
- Memory operand addressing mode
  1. Direct Addressing Mode
  2. Indirect Addressing mode

Immediate addressing mode:

- In this mode, the operand is a number or constant, not an address as MOVLW43H.
- The operand here is a data not an address.
- So in this addressing mode of PIC microcontroller data is direct transfer and data is immediately after opcode.

Register operand addressing mode:

- In this mode the operand is a register which holds the data to be execute.
- Register operand addressing mode deals with registers like W, FSR (File Select Register), INDT (Indirect Register)

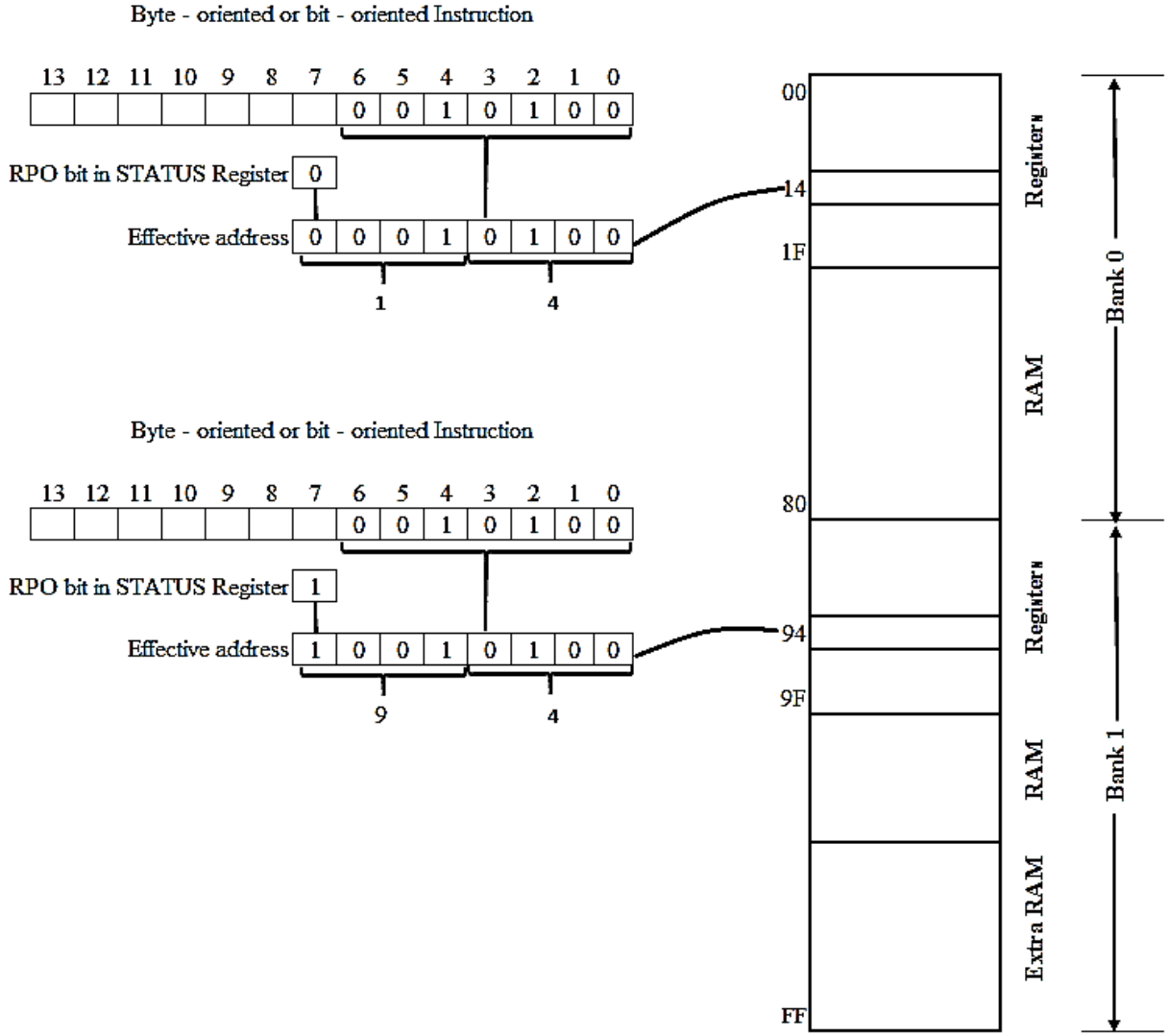
Memory Operand addressing mode:

- In this mode, the operand is an address of memory location which holds the data to be execute.
- The memory operand addressing mode is classified into,
  - Direct addressing mode
  - Indirect addressing mode

Direct Addressing Mode:

- In direct addressing mode 7 bits (0-6) of the instruction identify the register file address and the 8^{th} bit of the register file address register bank select bit (RP0).
- The figure illustrates direct addressing being used to access register file address 14H or 94H depending on the value of RP0.
- If the RP0 bit in STATUS register is 0, then the effective address feeds the 14H in to the Register in the Bank 0.

- If the RP0 bit is 1, then the effective address feeds the 94H in to the register in the Bank 1.

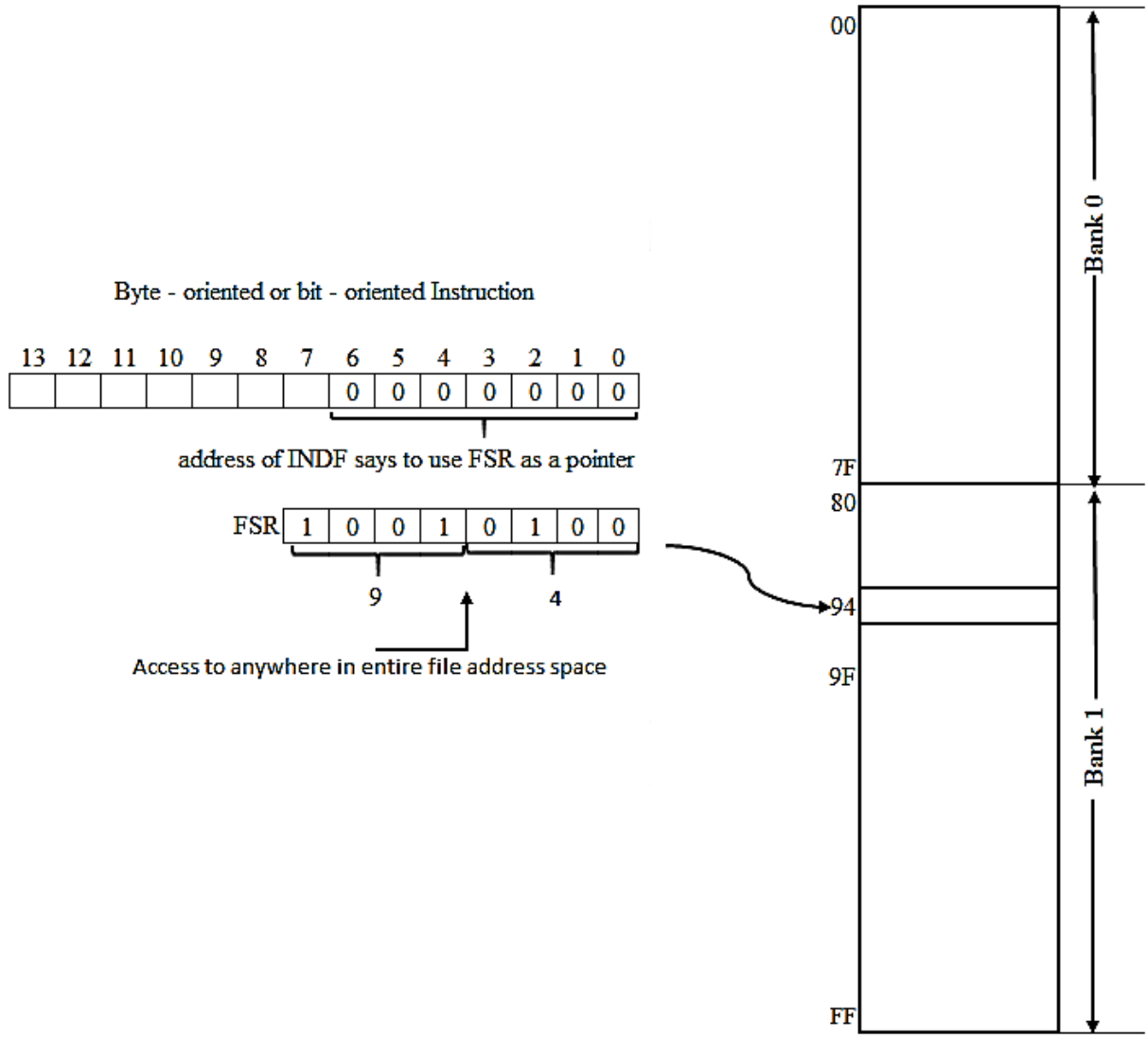


Direct Addressing Mode

Indirect Addressing Mode:

- Every instruction that can employ the direct addressing mode can employ the indirect addressing mode.
- In the indirect addressing mode the full 8-bit register file address is first written into a File Select Register (FSR) using direct addressing.
- FSR serves as an address pointer to any address throughout the entire register file.
- A subsequent direct access of INDF will actually access the register file using the content of FSR as a pointer to the desired location of the operand.
- This is possible by making the value of the register bank select bit RP0, the FSR register is accessed at either address 14H or 94H.

14 H = 00010100 94 H = 10010100



Indirect Addressing Mode

8. Mention the guidelines and instruction sets used in PIC controller? *[Nov'16]* [OR] **Explain about the instruction set of PIC microcontroller.** *[Nov'17]*

While writing the instructions the following guidelines are followed.

- Write the instructions mnemonics in lower case (example: xorwf)
- Write special Register names, RAM variable names and bit names in upper case (example: STATUS, RPO….)
- Write instruction and subroutine labels in mixed case (example: Mainline, LoopTime.)

The instruction set of PIC is divided into three basic categories.
They are

(a) Byte oriented Instructions
(b) Bit oriented Instructions
(c) Literal and Control Instructions

Byte Oriented Instructions

- In a byte oriented Instructions **'f'** represents a file register and **'d'** represents destination register.

- The destination specifies where the result of operation is to be placed.
- If d = 0, the result is placed in W register (Accumulator)
- If d = 1, the result is placed in the file register specified in the instruction.

```
addwf f, d      ;  Add  W and  f
clrf  f         ; Clear f
movwf f ,d      ; Move f
nop             ; No operation
subwf f ,d      ; Subtract W from f
```

Bit Oriented Instruction

- In bit oriented instructions, **'b'** represents a bit field designator which selects the number of the bit affected by the operation
- And **'f'** represents the number of the file in which the bit is located**.**

```
bcf f, b        ; Bit clear f
bsf f, b        ; Bit set f
btfsc f, b      ; Bit test f, skip if set
```

Literal and Control Instructions

- In literal and control instructions **'k'** represents an 8 or 11 bit constant or literal value.

```
addlw   k       ; Add literal and W
andlw   k       ; AND literal with W
call k          ; Call subroutine
movlw k         ; Move literal to W
sublw k         ; Subtract W from literal
```

CLASSIFICATION OF INSTRUCTIONS: [Based on Operation]

- All the instructions of the PIC microcontroller are classified into nearly 9 groups. They are given below with examples.

Single-bit manipulation		Operation
bcf	PORTB, 0	;Clear bit 0 of PORTB
bsf	STATUS, C	;Set the carry bit
Clear/move		
Clrw		;Clear the working register, W
clrf	TEMP1	;Clear temporary variable TEMP1
movlw	5	;Load 5 into W
movlw	10	;Load D '10' or H '10' into W
		;depending upon default representation
movwf	TEMP1	;Move W into TEMP1
movwf	TEMP1, F	;Incorrect syntax
movf	TEMP1, W	;Move TEMP1 into W
swapf	TEMP1, F	;Swap 4-bit nibbles of TEMP1
swapf	TEMP1, W	;Move TEMP1 to W, swapping nibbles
		;and leave TEMP1 unchanged
Increment/decrement/complement		
incf	TEMP1, F	;Increment TEMP1
incf	TEMP1, W	;W < - TEMP1 + 1; TEMP1 unchanged
decf	TEMP1, F	;Decrement TEMP1
comf	TEMP1, F	;Change 0s to 1s and 1s to 0s

Multiple-bit manipulation	
andlw B'00000111' andwf TEMP1, F andwf TEMP1, W iorlw B'00000111' iorwf TEMP1, F xorlw B'00000111' xorwf TEMP1, W	;Force upper 5 bits of W to zero ;TEMP1 < - TEMP1 and W ;W < - TEMP1 AND W ;Force lower 3 bits of W to one ;TEMP1 < - TEMP1 or W ;Complement lower 3 bits of W ;W < - TEMP1 XOR W
Addition/ Subtraction	

addlw 5
addwf TEMP1, F
sublw 5
subwf TEMP1, F
	;Add 5 to W
	;TEMP1 < - TEMP1 + W
	; W< - 5 – W (not W < - W – 5!)
	;TEMP1 < - TEMP1 – W

Rotate	
rlf TEMP1, F rrf TEMP1, W	;Nine-bit left rotate through C ;(C < - TEMP1, 7; TEMP1, I+1 < - TEMP1, I ; TEMP1, 0 < - C) ; Leave TEMP1 unchanged ;copy to W and rotate W right through C
Conditional branch	
btfsc TEMP1, 0 btfss STATUS, C decfsz TEMP1 , F incfsz TEMP1, W	;Skip the next instruction if bit 0 of ;TEMP1 equals zero ;Skip if C = 1 ;Decrement TEMP1; skip if zero ;Leave TEMP1 unchanged; skip if ;TEMP1 = H'FF'; W< - TEMP1 + 1
Goto/call/return/return from interrupt	
goto There call Task1 return retlw 5 retfie	;Next instruction to be executed is ; labeled "There" ;Pushed return address; next instruction ;to be executed is labeled "Task1" ;Pop return address off of stack ;Pop return address; W < -5 ;Pop return address; returnable interrupts
Miscellaneous	
clrwdt sleep nop	;If watchdog timer is enabled, this; instruction will reset it (before it,;resets the CPU) ;Stop clock; reduce power; wait,;for watchdog timer or external signal;to begin program execution again ; Do nothing; wait one clock cycles

Simple Program:

9. Write a program alternately loads Port B with 55 and AA.

```
        Movlw   0X0
        movwf   TRISB       ; make the Port B an output port.
L1:     movlw   0X55        ; WREG = 55
        movwf   PORTB       ; Move 55 into Port B.
        call    DELAY
        movlw   0X AA
        movwf   PORTB       ; Move AA into Port B
        call    DELAY
        goto    L1
```

10. Write a program for Port B and Port C are used to transfer the data continuously.

```
        movlw   B '00000000'  ; WREG = 00000000(Binary)
        movwf   TRISB         ; Port B an out port
        movlw   B '11111111'  ;WREG = 11111111 (Binary)
        movwf   TRISC         ; Port C input Port
L2:     movf    PORTC, W      ; Move data from Port C to WREG.
        addlw   5             ; Add literal 5 to it.
        movwf   PORTB         ; send it to Port B
        goto    L2            ; Continue the loop.
```

11. Write a program to conform that TRIS bits are activated by putting a 1, the data will not be transferred to WREG from the port pins of PORT C

```
        clrf    TRISB       ; Clear TrisB ( Port B is made outport)
        setf    TRISC       ; Set TRISC, (Port C is made Input port)
L2:     movf    PORT C, W   ; Get data from Port C.
        addlw   5           ;  Add  literal 5
        movwf   PORT B      ; Send it to Port B
        bra     L2          ;  Branch to Loop L2
```

12. Write a program by using Either – or – sequence

Assume that an instruction that affects the Z bit has just been executed. Then depending on the result, one instruction sequence or another is to be executed, continuously on after either case.

```
        btfsc   STATUS, Z       ; Test Z bit, skip if clear
        goto    Zset
Zclear
          .                     ; Instructions to execute
          .                     ; If z = 0
          .
        Goto    Zdone
Zset
         .                      ; Instructions to execute
          .                     ; If z = 1
          .
Zdone                           ; CarryOn
```

13. Write a program to Decrement a 16 – bit counter

Assume that the upper byte of the counter is called COUNTH and the lower byte is called COUNTL.

```
movf    COUNTL, F       ; Set Z if lower byte = 0
btfsc   STATUS, Z       ; If so, decrement COUNTH
decf    COUNTH, F
decf    COUNTL, F       ; In either case decrement COUNTL
```

14. Write a program to Test a 16 – bit variable for zero

```
movf    COUNTL, F       ; Set Z if lower byte = 0
btfsc   STATUS, Z       ; If not, then done testing
movf    COUNTH, F       ; Set Z if upper byte = 0
btfsc   STATUS, Z       ; If not, then done
goto    BothZero        ;Branch if 16 – bit variable = 0
CarryOn
```

15. Write PIC microcontroller assembly language program to arrange the given array having byte type data in ascending order. *[Apr'18]*

```
ARR DB 15, 12, 18, 13, 19, 16, 14, 20, 11, 17
LEN DW $-ARR
DATAENDS
CODESEGMENT
ASSUMEDS: DATACS: CODE
START:
movax.DATA
movds.AX
movcx.LEN-1
OUTER:
LEASI.ARR
movbx.0
INNER:
incBX
moval.ARR[SI]
incSI
cmpal.ARR[SI]
jb SKIP
xchgal.ARR[SI]
movARR[SI-1]AL
SKIP
cmpbx.CX
jl INNER
loop OUTER
movah.4CH
int21H
CODEENDS
END START
```

ANNA UNIVERSITY QUESTIONS

PART A

1. Difference between microcontroller and PIC microcontroller. *[Nov'17]*
2. Give the architecture of PIC 16C6X series. *(Jan'12)*
3. Draw the program memory organization of PIC16C6x microcontroller. *[Apr'18]*
4. Write the significant of brown out reset [BOR] mode? *[Apr'17]*
5. Explain about watchdog timer? *(Jan'11)*
6. Write about the Status Register of PIC Microcontroller. *[Nov'16]*
7. What is RISC? *(Jun'14, Jul'13)*
8. What are the benefits of having RISC architecture? *[Apr'17]*
9. List out all the addressing Modes in PIC Microcontroller. *(Jan '12, Nov'16, Nov'17)*
10. What is instruction pipelining? *(Jan '11)*
11. What is I/O port of PIC? *(Jan'14)*
12. What are the groups of instruction set in PIC micro controller? *(Jan'13)*
13. Write any four instructions of PIC microcontroller and state in a line the operation performed. *(Jan'14)*
14. Using the instruction of PIC microcontroller, convert BCD to Hex. *(Jan'13)*
15. How is the internal RAM in PIC microcontroller accessed by indirect addressing? *(Jun'12)*
16. Write the operation carried out when these instructions executed by PIC.
    BTFSS f, b
    BCF f, b. *[Apr'18]*

PART B

1. Explain clearly about the architecture of PIC Microcontroller? *[Nov'16, Apr'17]*
2. Draw and explain about the architecture of PIC microcontroller. *[Nov'17]*
3. Explain the architecture of PIC16C6x microcontroller with neat block diagram. *[Apr'18]*
4. Explain various Memory Organisation of PIC microcontroller. *[Nov'16, Apr'17]*
5. Explain the different PIC addressing modes? *[Apr'17, Nov'17]*
6. With examples, explain the addressing modes of PIC16C6x microcontroller. *[Apr'18]*
7. Mention the guidelines and instruction sets used in PIC controller? *[Nov'16]*
8. Explain about the instruction set of PIC microcontroller. *[Nov'17]*
9. Write PIC microcontroller assembly language program to arrange the given array having byte type data in ascending order. *[Apr'18]*

UNIT – II PART A

1. **What is interrupt service routine (ISR)?**
   - If an interrupt has occurred and both its local enable and global enable are set.
   - The interrupt is therefore detected by the CPU and it executes a special section of program called the Interrupt Service Routine (ISR).

2. **Mention the interrupts available in PIC microcontroller?** *[Apr'17]*
   - Hardware interrupts
   - Software interrupts

3. **Define subroutine.** *[Nov'17]*
   - Many applications requires the software to perform signed and unsigned multiprecision multiplication and division.
   - These operations must be implemented as programs.
   - To be reusable in other applications, these programs should be written as subroutines.

4. **Write the loop time subroutine program for interrupt.**

```
LoopTime
        btfss   SCALER, 7       ; wait for 00000000 to 11111111 change
        goto    LoopTime
        movlw   5               ; Add 5 to SCALER
        addwf   SCALER, F
        return
```

5. **What is the role of timer module in PIC16CXX?**
   - The PIC16C7X each have three timer modules.
   - Each module can generate an interrupt to indicate that an event has occurred (time overflow).

6. **What is the minimum and maximum clock frequency of PIC 16cxx?** *[Nov'16]*
   - Minimum clock frequency - 4MHZ
   - Maximum clock frequency - 40MHZ

7. **What is the role of TRISx register in I/O port management?** *[Nov'16]*
   To set the PORT (A to E) as input and output values in PIC microcontroller.

8. **Write an ALP to initialize the PORTA using PIC microcontroller.** *[Apr'17]*

```
Org0
        Bcf     STATUS.RP0
        clrf    PORTA
        bsf     STATUS.RP0
        movlw   00010000H
        movwf   TRISA
End
```

9. Mention the working of Timer 0.

- Internal clock source of OSC/4 is an external clock source, if selected can be applied at RA4/T0CAI input at PORTA

10. What is the necessity of prescaler in the timer operation? *[Apr'18]*

- Prescaler is a mechanism for generating clock for timer by CPU clock.
- Every CPU has a clock source and the frequency of this source decides the rate at which instructions are executed by the processor.
- Clocks of several frequencies such as 1 MHz, 8 MHz, 12 MHz and 16 MHz (max).
- The prescaler is used to divide the clock frequency and produce a clock for TIMER.

11. Mention the usage of prescaler of Timer 0.

- Prescaler is a programmable divided by N counter that divides the available clock by a prespecified number, before applied to the timer zero counter.
- It can be used either timer 0 or timer 1.

12. What is the feature of Timer 0?

- Timer 0 is a simple 8 – bit overflow counter.
- It has a programmable prescaler option
- It can increment at the following rates: 1:1 (when prescaler assigned to watchdog timer), 1:2, 1:4, 1:8, 1:16, 1:32, 1:64, 1:128 and 1:256.

13. Explain the functions of Timer 1.

- Timer 1 is a 16 bit timer/counter consisting of two 8 bit register namely TMR1H and TMR1L which are readable and writable nature.
- The clock source can be either the internal system clock (Fosc/4), an external clock or an external crystal.
- The TMR1 register pair is increments from 0000H to FFFFH and rolls over to 0000H at the following rates 1:1, 1:2, 1:4 and 1:8.
- If enable it generates on overflow by setting the interrupt flag bit TMR1F in the PIR1 register.

14. How timer 1 performs in timer mode?

- The timer mode is selected by clearing the TMR1CS bit.
- In this mode, the input clock to the timer is FOSC/4.
- The synchronize control bit T1SYNC has no effect since the internal clock is always in synchronized.

15. How timer 1 performs in counter mode?

- Counter mode is selected by setting bit TMR1CS.
- In this mode the timer increments on every rising edge of clock input on pin RC1/T1OSI/CCP2.
- When bit T1OSCEN is set or pin RC0/T1OSO/T1CKI when bit T1OSCEN is cleared.

16. What are the features of timer 2?

- Timer 2 is an 8 – bit timer with a programmable prescaler and postscaler as well as an 8 – it period register (PR2).

- Timer 2 can be used with the CCP1 module (PWM mode) as well as the Baud rate generator for the Synchronous Serial Port (SSP).
- The postscaler allows the TMR2 register to match the period register (PR2) a programmable number of times before generating an interrupt.

17. Write usage of Timer 2.

- Timer2 is an 8-bit timer with a prescaler, a postscaler, and a period register.
- Using the prescaler and postscaler at their maximum settings, the overflow time is the same as a 16-bit timer.
- Timer2 is the PWM time-base when the CCP module(s) is used in the PWM mode.

18. Elaborate the CCP module of PIC. *(Jun'12)*

- The expansion of CCP is Capture / Compare/ PWM module used in Timer 1 / Counter.
- It contains a 16 – bit register which can operate as a 16 – bit capture register or as a 16 – bit compare register or as a PWM master/Slave Duty Cycle register.

19. Draw the INTCON register of configuration of PIC.

D7	D6	D5	D4	D3	D2	D1	D0
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF

Where, GIE - Global Interrupt Enable.
PEIE – Peripheral Interrupt Enable
TMR0IE – Timer 0 Interrupt Enable
INTE – External Timer Enable
RBIE – Port Interrupt Enable
TMR0IF – Timer 0 Interrupt Flag
INTF – External Timer Flag
RBIF – Port Interrupt Flag

20. List out the code templates in PIC.

- Explanatory remarks and a program hierarchy
- Assembler directives
- Equates to give names to numbers
- variable definitions
- Reset and Interrupt vectors
- A table
- Mainline code and its subroutines
- Initialization code
- Interrupt service routine
    - Systematic handling of W and STATUS
    - Polling routine
    - Specific interrupt handling subroutine
- LoopTime control

21. Draw the optional register configuration of PIC.

7	6	5	4	3	2	1	0
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

Where,

If PSA is 0 - Prescaler is assigned to the Timer0 module

PSA is 1 - Prescaler is assigned to the watch dog

If PS0 is 1 - Prescaler rate bits

If PS1 is 1 - are set to "111"

If PS2 is 1 - which means divide by 256

If T0SE is 0 - Timer0 Set Enable

T0CS is 1 - Timer0 clock source

22. What do you mean by state machine? *[Nov'17]*

- State machines are simple constructs used to perform several activities, usually in a sequence. Many real-life systems fall into this category.
- State machines create a structured, easily maintained approach to programming.
- They can easily be used in projects, to help prevent bugs, to stop infinite loops hogging the processor, and to ease debugging.

23. What is softkeys?

- A softkey or soft key is a button flexibly programmable to invoke any of a number of functions rather than being associated with a single fixed function or a fixed set of functions.
- A softkey is often located alongside a display device of a portable device such as a front panel, where the button invokes a function described by the text at that moment shown adjacent to the button on the display.

24. What are applications of front panel softkeys?

- Softkeys are generally found on
  - keyboards (e.g. the F keys),
  - cellular phones,
  - Automated Teller Machines,
  - Primary Flight and Multi-Function Displays

25. How to display constant strings? *[Apr'18]*

- Constant strings arise in several ways.
- The labels associated with softkeys represented one applications.
- The units (kHz) associated with a variable represent another.
- A DisplayC subroutine that makes use of display strings stored in program memory will be developed.
- Each byte of each string is accessed through a "retlw" instruction in the process of returning from a DisplayC_Table subroutine.

- The source code form of a display string stored in program memory can be, "Row4coll"

PART B

1. **Explain in detail about the PIC Microcontroller Interrupts?** *[Nov'16, Apr'17, Apr'18]*
   [OR] **Explain the interrupt structure of PIC microcontroller with neat diagram.** *[Nov'17]*

- Whenever any device needs the microcontroller's services, the device notifies it by sending an Interrupt signal.
- Upon receiving an interrupt signal, the microcontroller stops whatever process and servers the device.
- The program associated with the interrupt is called the Interrupt Service Routine (ISR) or Interrupt handler.

Interrupt Service Routine:

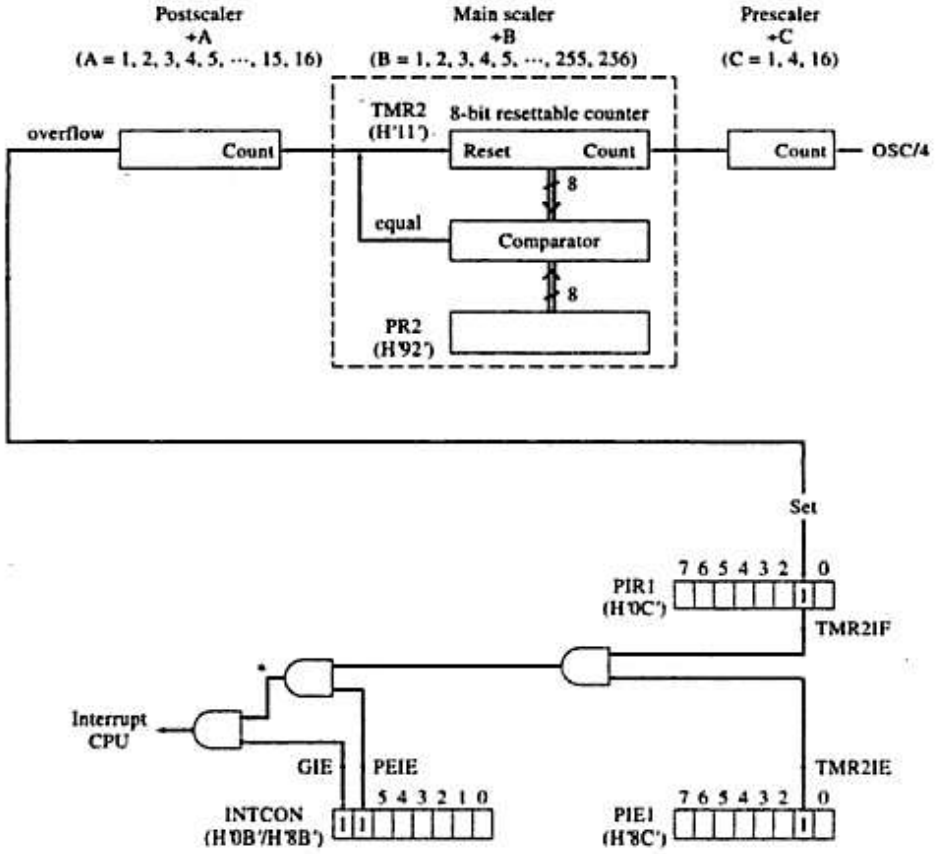
- In microcontroller, for every interrupt there is a fixed location in memory that holds the address of its ISR.
- The group of memory set aside to hold the address of ISRS is called the interrupt vector table.
- In PIC there are only two locations for the interrupt vector table, location 0008 and 0018.

Interrupt	POM Location (Hex)
Power – ON Reset	0000
High Priority Interrupt	0008 (Default upon power ON reset)
Low Priority Interrupt	0018

Execution of an Interrupt:

- It finishes the instruction it is executing and saves the address of the next instruction (Program Counter) on the stack.
- It jumps to a fixed location in memory called Interrupt Vector Table, which directs the microcontroller to the address of the Interrupt Service Routine.
- The microcontroller starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine RETFIE (Return from Interrupt Exit).
- Upon executing the RETFIE instruction, the microcontroller returns to the place where was interrupted.
- First it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PIC. Then it start to execute from that address.

Interrupt Logic:



- From the figure when the postscaler rolls over, it sets a Flag called TMR21F in the P1R1 register.
- If the TMR21E bit in the P1E1 register and PE1E bit in the INTCON register, both been initialized at one.
- Then an interrupt signal will make it all the way to the point labelled with an asterisk (*) in the figure at GIE bit in the INTCON register can be set and cleared under the program control with.

```
bsf     INTCON, GIE;    Global Interrupt Enable

bcf     INTCON, GIE;    Global Interrupt clean/Disable.

retfie          ;       Return from the interrupt service routine
```

- Within the interrupt service routine, TMR21F flag that caused the interrupt must be cleared with

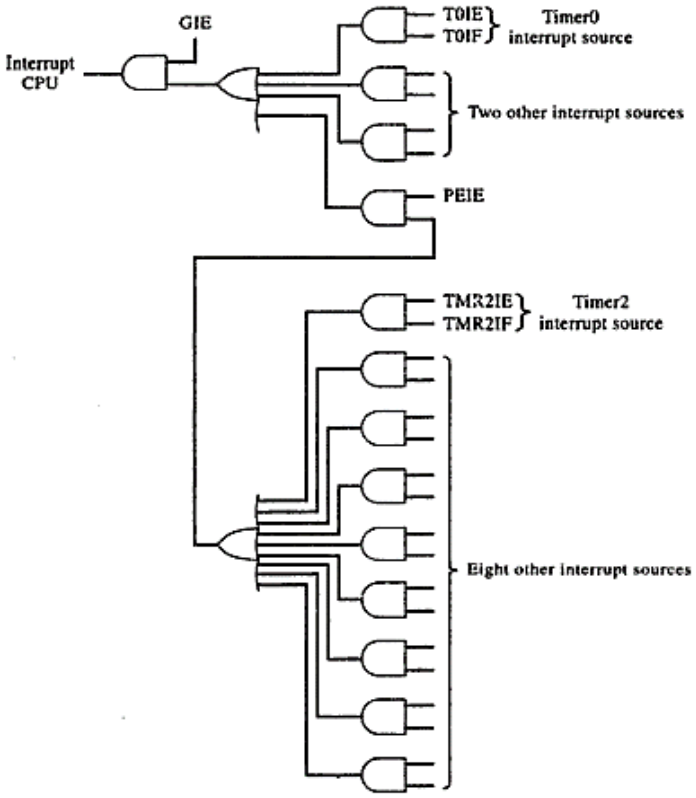
```
bcf     P1R1, TMR21F
```

- Otherwise, upon execution of

```
retfie
```

- The CPU will re- enable interrupts, Timer 2 interrupt still pending and vector off to service it again.

- The CPU will never execute another instruction in the mainline program.
- When the PC chip is reset, GIE, PE1E and TMR21F are cleared.
- Consequently they must be the set in the initial subroutine to turn on the interrupts that will be used to control the loop time.
- The figure shows the gating path used by one specific interrupt source.



PIC16C74A Interrupt Logic

- The larger picture of PIC interrupt logic is shown in the figure above to handle up to 12 interrupt source in the PIC16C74A.
- The other chips in this family have somewhat fewer interrupt sources, but the structure is the same.

2. Explain how Intservice Interrupt Service Routine is achieved in PIC. *[Nov'16, Apr'17, Apr'18]*

- Whenever an interrupt occurs, the CPU automatically pushes the return address in the program counter onto the stack and clear the GIE bit, disabling further interrupts.
- Consequently the intservice is to set aside the content of W and STATUS.
- Then they can be restored at the end of the interrupts service routine to exactly the same state where the interrupt occurred, as required for the proper execution of the mainline code.
- This setting aside of W and STATUS is illustrated in the first these instructions of the below.

```
Intservice
; Set aside W and STATUS
        movwf     W_TEMP          ; Copy W to RAM
        swapf     STATUS, W       ; Move STATUS to W without affecting z
```

```
                                        bit
        movwf   STATUS_TEMP     ; Copy to RAM (with nibbles swapped)
; Execute polling routine
        btfsc   PIR1, TMR2IF    ; check for Timer 2 interrupt
        call    Timer2          ; If ready, service it
        btfsc   ….....          ; check another interrupt source
        call    ….....          ; If ready, service it
        btfsc   ….....          ; check another interrupt source
        call    ….....          ; If ready, service it
; Restore STATUS and W and return from interrupt
        swapf   STATUS_TEMP, W  ; Restore STATUS bits
        movwf   STATUS          ; without affecting z bit
        swapf   W_TEMP, F       ; swap temporary into W
        swapf   W_TEMP, W       ; swap again into W without affecting z bit
        retfie                  ; Return to mailnine code
Timer2
        bcf     PIR1, TMR2IF    ; clear interrupt flag (Bank 0)
        decf    SCALER, F
        return
```

- The three instructions for setting aside W and STATUS use;

  swapf STATUS, W

- to move the content of STATUS to W instead of:

  movf STATUS, W

- The swapf instruction does not affect the z bit in the STATUS register when it makes this move. Whereas, the movf instruction may corrupt the z bit, so it must not be used here.

  swapf W_Temp, W

- Copies W_ Temp to W without affecting the z bit, it is preceded by

  swapf W_ Temp, F

  which swapf the two values of W_ TEMP so that the following swapf instruction will swap them again, ending up with every bit of W just there it was when the interrupt occurred.
- The central code of IntService is a sequence of btfsc, call instruction pairs.
- Each pairs tests the flag of an enabled interrupt source. If the flag is set, the source's interrupt service routine is called that provides the desired response and cleans the flag.

3. Write short notes on Looptime Subroutine.

- With the help of the Timer2 subroutine in Intservice, the Looptime subroutine that is called from within the mainline loop is able to make the time around the loop take exactly 10 ms.
- For the LoopTime subroutine to work correctly, the worst case execution remainder code in the mainline loop plus time for all interrupt service routines within a 10 ms interval must be less than 10 ms.
- This mainline overrun condition is easily avoided, if not during one loop the next looptime will be shortened or compensated.
- As a consequence, successive execution of some tasks may occur less than 10 ms apart.
- On the other hand, even if this mainline overrun condition does occur, the looptime subroutine will be accurate as long as no counts of SCALER are ever lost.

- Once the CPU enters the Looptime subroutine, it waits on the Timer2 subroutine in Intservice, which waits on successive interrupts from Timer2.
- When it occurs, finally decrements SCALER down from 00H to FFH, the goto Looptime instruction will be skipped, five will be added to SCALER (resulting in SCALER= 4) and the CPU will return from the interrupt service routine.
  LoopTime

```
btfss   SCALER, 7       ; wait for 00000000 to 11111111 change
goto    LoopTime
movlw   5               ; Add 5 to SCALER
addwf   SCALER, F
return
```

4. How Enabling and Disabling of an Interrupt is achieved in PIC controller

- Upon reset, all interrupts are disabled (masking), that is none will be responded to by microcontroller if they are activated.
- The interrupts must be enabled (unmasked) by software.
- The D7 bit in of the INTCON (Interrupt Control) register is responsible for enabling and disabling the interrupts globally. The GIE bit makes the job of disabling all the interrupts by

```
bcf     INTCON, GIE
```

  we can make the GIE = 0, during the operation of a critical task.

D7	D6	D5	D4	D3	D2	D1	D0
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF

Where, GIE - Global Interrupt Enable.
PEIE – Peripheral Interrupt Enable
TMR0IE – Timer 0 Interrupt Enable
INTE – External Timer Enable
RBIE – Port Interrupt Enable
TMR0IF – Timer 0 Interrupt Flag
INTF – External Timer Flag
RBIF – Port Interrupt Flag

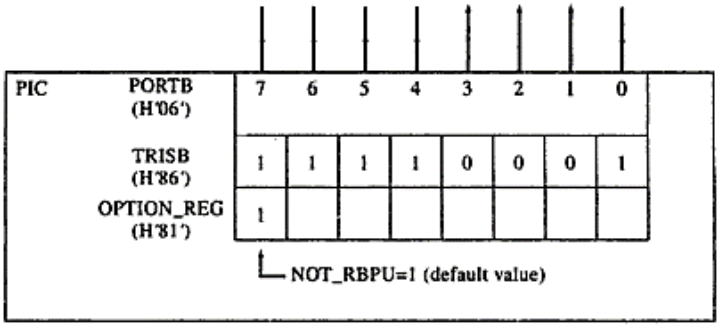
- These bits along with GIE, must be set high for an interrupt to be responded to upon activation of the interrupt.
- The GIE bit is cleared by the PIC itself to make sure another interrupts cannot interrupt the microcontroller while it is servicing the current one and to allow another interrupts to come in PEIE (Peripheral Interrupt Enable).

5. What is meant by External Interrupts and explain how RB0/ INT External Interrupt Input is connected with PIC. *[Apr'18]*

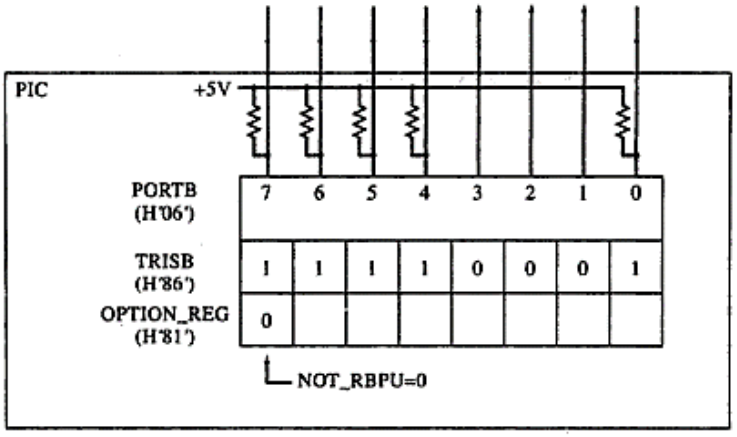
- The response of a PIC microcontroller to an external event can be initiated by a mainline subroutine that looks for a change on an input port pin.
- This is satisfactory for events that are slow relative to the 10 ms looptime, but not for faster events.
- Faster – occurring events can be made to trigger an interrupt response through a change on a specific input pin.

RB0 / INT External Interrupt Input:

- The PIC controller has one pin, RB0/INT serves as its primary external interrupts input. This pin is bit 0 of PORTB.
- Before initializing the interrupt, PORTB is initialized.
- The bits of the Bank 1 register TRISB set up the corresponding bits of PORTB as either inputs or outputs.
- All of the pins that are set up as inputs can include an optional weak pullup resistor by clearing the NOT_RBPU bit of OPTION_REG.
- This provides a useful input for a pushbutton switch or for any array of keyswitches in the keypad.

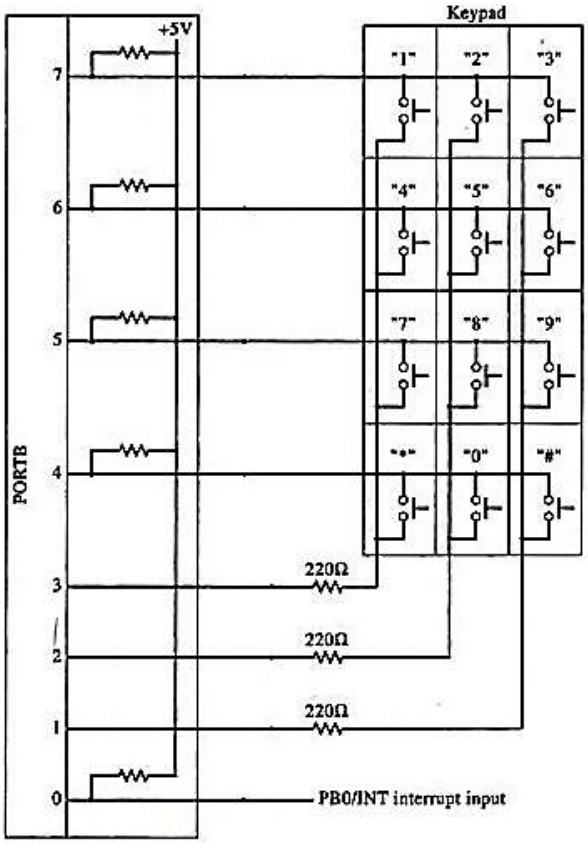


Initialization of PORTB with three outputs and five high – impedance inputs



Initialization of PORTB so that its inputs include weak pullup resistors

- The internal pullups of the circuit shown hold each input pin high until one of the three keyswitches attached to it is closed and the corresponding column driver output has been driven low.
- For example, bit 7 of PORTB will be driven low if the keyswitch labelled "1" is pressed and if the bit "3" output from PORTB is driven low. Otherwise the internal pullup resistor pulls the bit 7 input high.



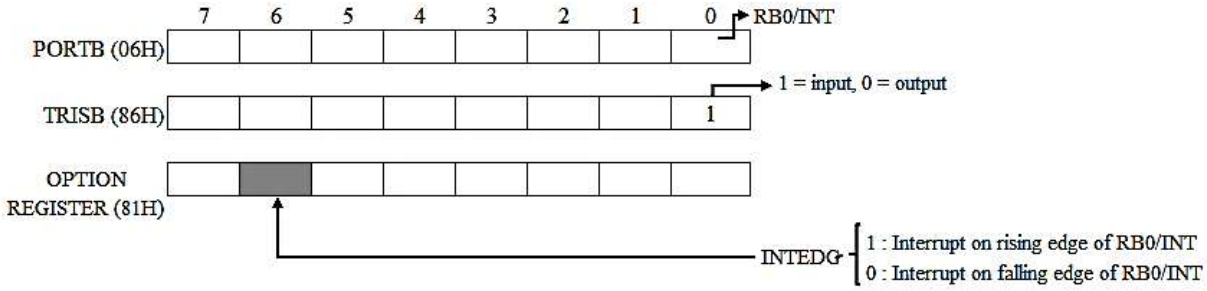
Use of internal pullup resistors with an array of keyswitches

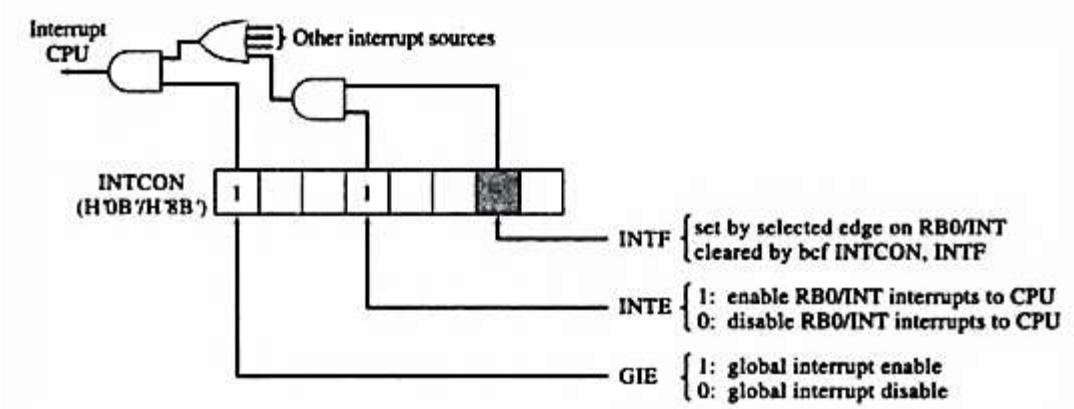
- The figure shows the use of bit 0 of PORTB as an interrupt input that can be used independently.

1	2	3
4	5	6
7	8	9
*	0	#

3 x 4 keypad

- The presence or absence of the weak pullup resistor on all PORTB inputs is irrelevant to bit 0 input.
- The INTEDG bit of OPTION_REG permits input to generate an interrupt on either a rising or a falling edge.
- When used as an interrupt input, this PB0/INT pin is automatically configured as a Schmitt – trigger input, triggering on the input edge regardless of its rise or fall time.





Initialization for RB0/INT Interrupts

- The INTCON register must be initialized with one of its INTE (RB0/INT Interrupt Enable) bit as well as in its GIE bit.
- When the interrupt occurs, there is no need to read PORTB. Just poll the INTF bit of INTCON to determine if an edge occurring on this pin, is the source of the interrupt.
- Then the clear flag by

```
bcf     INTCON, INTF
```

- Then service the interrupt and go back to INTService's polling routine to look for any other pending interrupts.

6. Write short note on CODE TEMPLATES in PIC.

- Code templates contain all of the elements of a source file for a general application:
  1) Explanatory remarks and a program hierarchy
  2) Assembler directives
  3) Equates to give names to numbers
  4) variable definitions
  5) Reset and Interrupt vectors
  6) A table
  7) Mainline code and its subroutines
  8) Initialization code
  9) Interrupt service routine
     1) Systematic handling of W and STATUS
     2) Polling routine
     3) Specific interrupt handling subroutine
  10) Looptime control

7. How many timers are present in PIC16F8XX timer modules? Explain them? *[Nov'16]*
[OR] **In detail give an account on Timer programming. RAM/ROM allocation in PC.** *[Nov'17]*

- There are three completely independent Timers available in PIC 16F8XX Microcontrollers. They are
  - Timer 0
  - Timer1 and
  - Timer2

Timer 0:

- The Timer 0 module is a simple 8-bit overflow counter.
- The clock source can be either the internal system clock (Fosc/4) or an external clock.
- When the clock source is an external clock, the Timer0 module can be selected to increment on either the rising or falling edge.
- The Timer 0 module also has a programmable prescaler option.
- This prescaler can be assigned to either the Timer0 module or the Watchdog Timer.
- Bit PSA assigns the prescaler and bits PS2: PSO determine the prescaler value.
- TMR0 can increment at the following rates: 1:1 when the prescaler is assigned to Watchdog Timer, 1:2, 1:4, 1:8, 1:16, 1:32, 1:64, 1:128 and 1:256.
- Synchronization of the external clock occurs after the prescaler.
- When the prescaler is used, the external clock frequency may be higher than the device's frequency.
- The maximum frequency is 50 MHz, given the high and low time requirements of the clock.

Timer 1

- Timer1 is a 16-bit timer/counter.
- The clock source can be either the internal system clock (Fosc/4), an external clock, or an external crystal.
- Timer1 can operate as either a timer or a counter.
- When operating as a counter (external clock source), the counter can either operate synchronized to the device or asynchronously to the device.
- Asynchronous operation allows Timer1 to operate during sleep, which is useful for applications that require a real-time clock as well as the power savings of SLEEP mode.
- Timer 1 also has a prescaler option, which allows TMR1 to increment at the following rates: 1:1, 1:2, 1:4 and 1:8 TMR1 can be used in conjunction with the Capture/Compare/PWM module.
- When used with a CCP module, Timer1 is the time-base for 16-bit capture or 16-bit compare and must be synchronized to the device.

Timer 2

- Timer 2 is an 8-bit timer with a programmable prescaler and a programmable postscaler, as well as an 8-bit Period Register (PR2).
- Timer 2 can be used with the CCP module (in PWM mode) as well as the Baud Rate Generator for the Synchronous Serial Port (SSP).
- The prescaler option allows Timer2 to increment at the following rates: 1:1, 1:4 and 1:16.
- The post scaler allows TMR2 register to match the period register (PR2) a programmable number of times before generating an interrupt.

- The postscaler can be programmed from 1:1 to 1:16 (inclusive).

CCP (Capture-Compare –PWM)

- The CCP module(s) can operate in one of three modes 16-bit capture, 16-bit compare, or up to 10-bit Pulse Width Modulation (PWM).

Capture mode:

- It captures the 16-bit value of TMR1 into the CCPRxH:CCPRxL register pair.
- The capture event can be programmed for either the falling edge, rising edge, fourth rising edge, or sixteenth rising edge of the CCPx pin.

Compare mode:

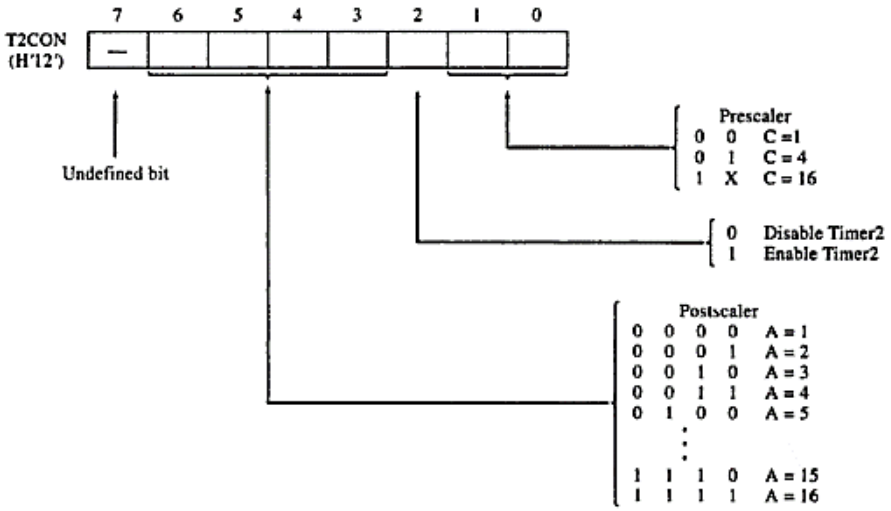
- It compares the TMR1H:TMR1L register pair to the CCPRxH:CCPRxL register pair.
- When a match occurs, an interrupt can be generated and the output pin CCPx can be forced to a given state (High or Low) and Timer1 can be reset.
- This depends on control bits CCPxM3:CCPxM0.

PWM mode:

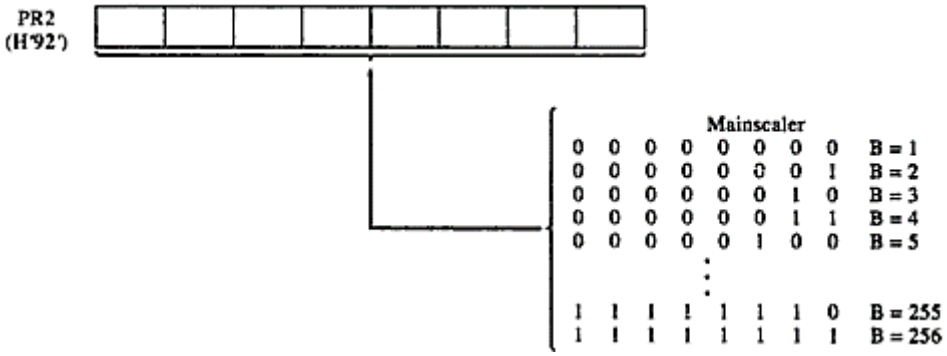
- It compares the TMR2 register to a 10-bit duty cycle register (CCPRxH:CCPRxl<5:4>) as well as to an 8-bit period register (PR2).
- When the TMR2 register = Duty Cycle register, the CCPx pin will be forced low.
- When TMR2=PR2, TMR2 is cleared to 00h, an interrupt can be generated, and the CCPx pin (if an output) will be forced high.

8. Explain how TIMER2 scaler initialization is made in PIC controller?

- The T2CON and PR2 registers controls the divide by scale of Timer2.
- T2CON sets up the prescaler and postscaler as shown in figure.
- The number represented by the 4 bits that determine the postscaler value must be one less than the desired divider value (That is to get a scale – of – five – divider, the number loaded into the bits 6, 5, 4, 3 is 4).
- The number loaded into PR2 must be one less the desired divider value for the main scaler (that is load 249 to get a scale – of – 250 divider).



T2CON Register



PR2 Register

- When Timer2 can implement a scaler having a period of 10 ms directly using a 4 MHz crystal with A = 10, B = 250, C = 4.
- The scaling for 2 ms interrupt interval is shown in the table.
- Timer2 can directly implement a scaler with a period of 2 ms using any of the three crystal frequencies.

OSC	Internal clock rate (OSC/4)	Internal clock period	A	B	C	Resulting scale
4 MHz	1 MHz	1μs	2	250	4	2 x 250 x 4 = 2000
10 MHz	2.5 MHz	0.4 μs	5	250	4	5 x 250 x 4 = 5000
20 MHz	5 MHz	0.2 μs	10	250	4	10 x 250 x 4 = 10000

- Timer2 interrupts occurring at every 2 ms and decrementing a RAM variable each time this interrupt occurs, the mainline program can keep track of this variable, when 10 ms have passed.
- The mainline program can reinitialize this variable, thereby creating a scale of five scaler to get 10 ms intervals for the mainline code while using 2 ms intervals for the interrupts.
- Initialization of T2CON can be generalized by

$$(4 \times freq) - 3$$

Where freq – 4, 10, 20.

  - For Initialization of scaler for 4 MHz, T2CON – B'00001101', PR2 – D'249'
  - For Initialization of scaler for 10 MHz, T2CON – B'00100101', PR2 – D'249'
  - For Initialization of scaler for 20 MHz, T2CON – B'01001101', PR2 – D'249'
- All conditions required by an interrupt will have been set up before the first interrupt actually occurs, the initial subroutine should be

```
bsf     INTCON, GIE
return
```

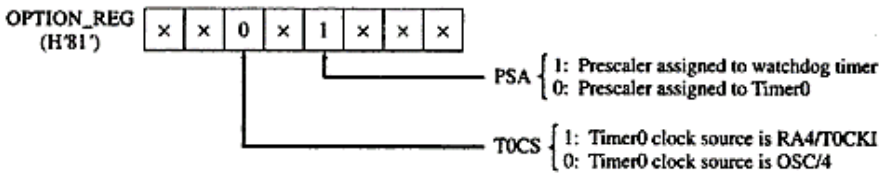
- The looptime subroutine and its interrupt service routine program will be

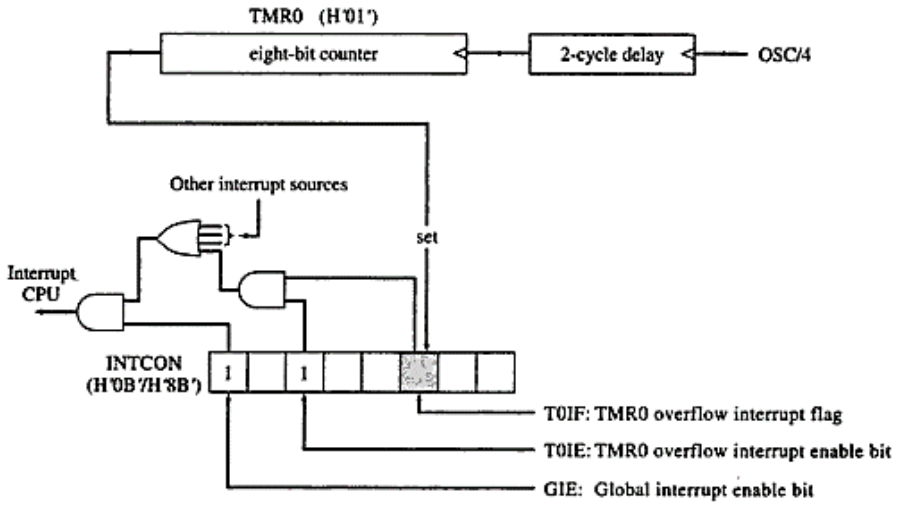
```
// add to equates block
        Freq    equ     4               ; crystal frequency in MHz (4, 10, 20)
// add to variables block
        w_temp                          ; Temporary storage for w
        STATUS_temp                     ; Temporary storage for STATUS
        SCALER                          ; Scale – of – 5 scaler used by Looptime
                                          subroutine.
```

```
// New vectors block
        org     H'000'                  ; Reset vector
        goto    Mainline                ; Branch past tables
        org     H'004'                  ; Interrupt vector

        goto    Instservice             ; Branch to interrupt service routine
// add to initial subroutine
; New bank 0 initializations
        movlw (4*freq)-3                ; Set up Timer 2
        movwf T2CON
        movlw 4                         ; Set up Timer 2 subroutine
        movwf SCALER
        bsf     INTCON, PEIE            ; Enable interrupt path
; New bank 1 initializations
        movlw 249                       ; Set up Timer 2
        movwf PR2
        bsf     PIEI, TMR2IE            ; Enable Timer 2 interrupt source
; Final bank 0 initializations
        bsf     INTCON, GIE             ; Enable global interrupts
        return
```

9. Write about the working of Timer0 in PIC controller.

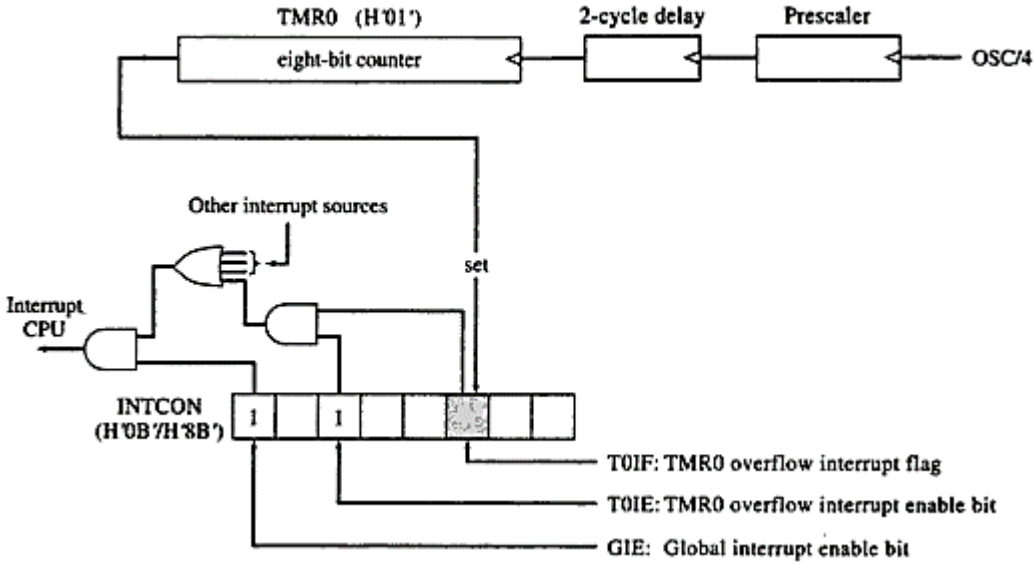
- Timer0 consists of an 8 – bit counter, TMR0, will be in read or write form.
- The counter sets a flag, T0IF, when it overflows and can cause an interrupt at the time interrupt enable T0IE = 1.
- Timer0 can be assigned an 8 – bit prescaler that can divide the counter's input by 2, 4, 8, ….256.
- Timer0 or its prescaler can be connected to either
    - OSC/4, the PIC's internal clock
    - RA4/T0CKI, the input connected to bit 4 of PORTA
- If the prescaler is bypassed and the internal clock used, the circuit is shown below.
- The two cycles delay is a result due to the need of synchronize the external clock T0CKI with the internal clock.
- Because of this two – cycles delay, 256 – 10 = 246 is written to TMR0, the T0IF flag will be set in 12 cycles, but not as 10 cycles expected.





Minimal use of Timer0.

- From the prescaler diagram shown below, when OPTION_REG is initialized TIMER0 will overflow every 2.048 ms for 4 MHz.
- This provides an accurate alternative to the use of the Timer2 circuitry for obtaining interrupts every 2 ms and a looptime of 10 ms.



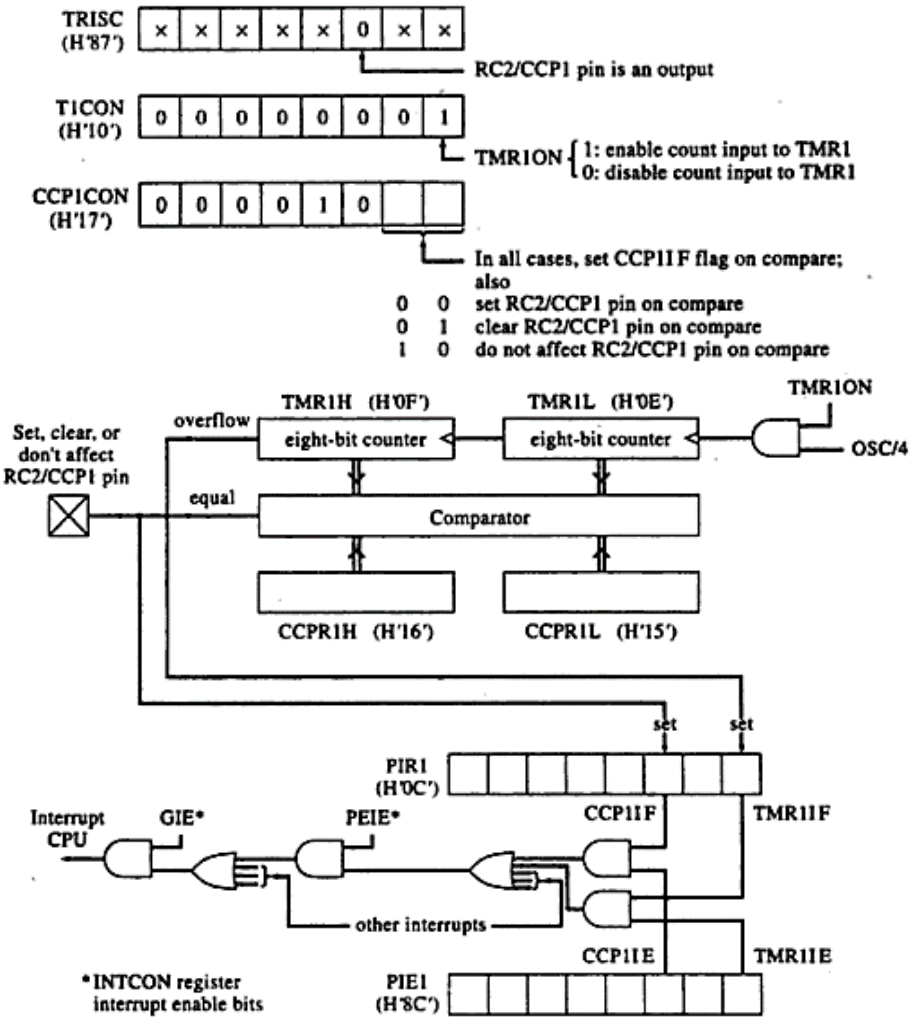
- Timer0 can serve as a powerful adjacent to use a rotary pulse generator (RPG).
- The prescaler value can be set to divide by 256, so TMR0 may count through less than a complete cycle between RPG interrupts.

10. Explain the working of Timer 1 in CCP (Capture/Compare/PWM) mode. [OR] **Explain the modes of Timer 1 of PIC16C6x microcontroller with block diagram. Also explain the function of associated registers.** *[Apr'18]*

Compare Mode:

- Timer 1 is a 16 – bit counter that, together with a CCP (Capture/Compare/PWM) module, can drive a pin high or low at a precisely controlled time, independent of CPU's time.

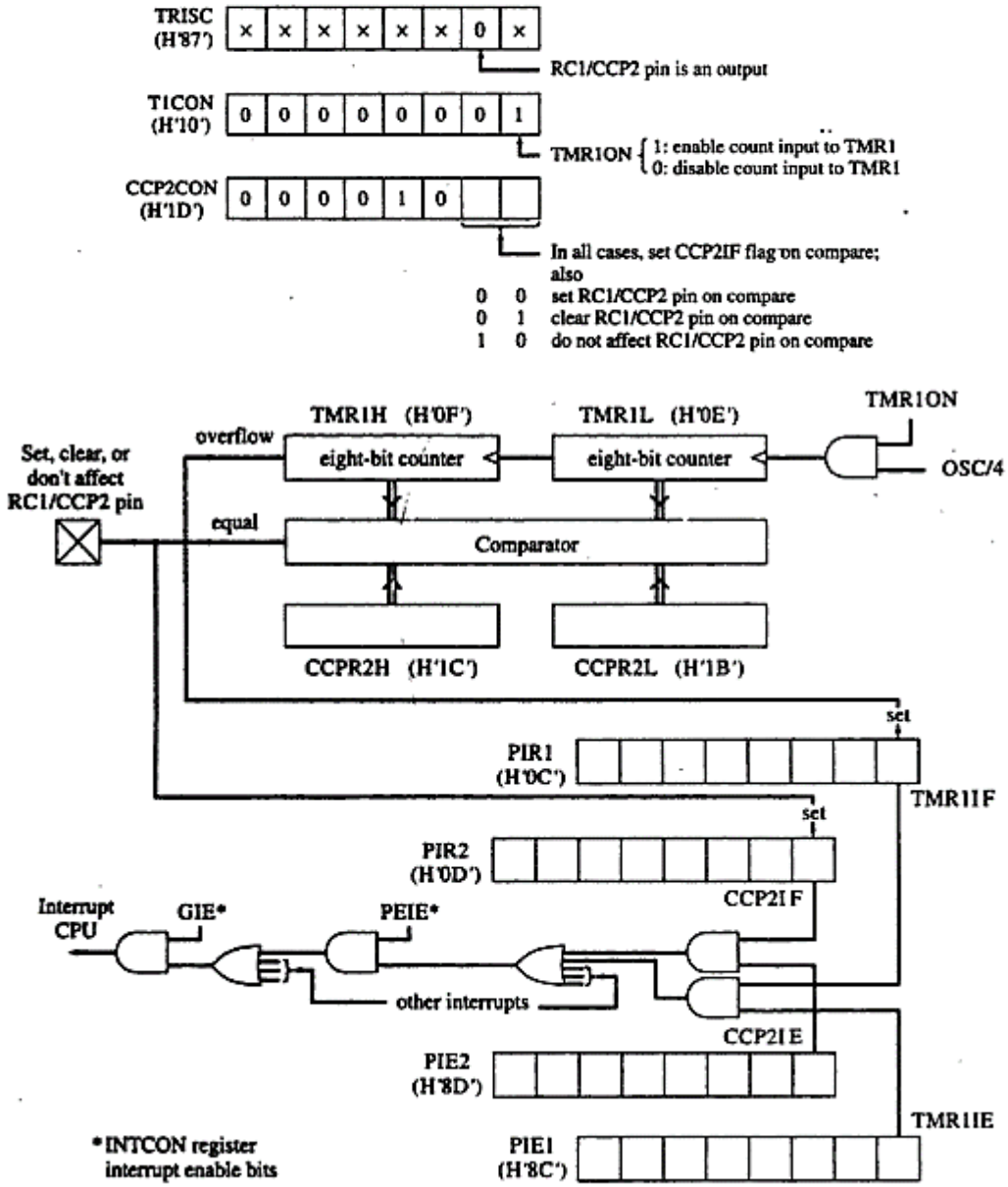
- In the seven PIC families, all have two CCP modules except the PIC16C62A, PIC16C64A and PIC16C72 provides with only one which precise control the output timing on the RC2/CCP1 pin.
- Those with two can control by RC2/CCP2 pin.
- Timer 1 includes prescaler to divide the internal clock by 1, 2, 4, 8 the choice of divide – by – one gives the finest resolution in setting the time of the output edge.
- For OSC = 4 MHz the timing of the edges of a pulse can be controlled with a resolution of 1 μs.
- The initialization for the CCP compare mode is shown in the figure.



CCP1 Compare Mode

- For the PIC having two modules, both being used for either compare function or capture with sharing TMR1.
- For this TMR1 should never be changed by writing to it.
- IF TMR1 is being used in one role only, its use is simplified by being able to stop its clocking, clearing TMR1, setting up CCPR1 and then starting the clocking of TMR1 again.
- IF TMR1 is being used for another function, in addition to the compare function, then TMR1 cannot be stopped and changed.
- For this TMR1 must be read and its value added in order to obtain the values for writing into CCPR1 for starting and stopping the pulse.

- Reading the 2 bytes of TMR1 while it is being clocked every cycle raises the potential for obtaining an erroneous value if the lower byte of TMR1 rolls over between the two reads.
- For reliable operation, read the upper byte first and then to tread it again after reading the lower byte.
- If these two values differ then do it again.
- For PIC chips having two CCP modules, the circuitry and registers for CCP2 are shown below.

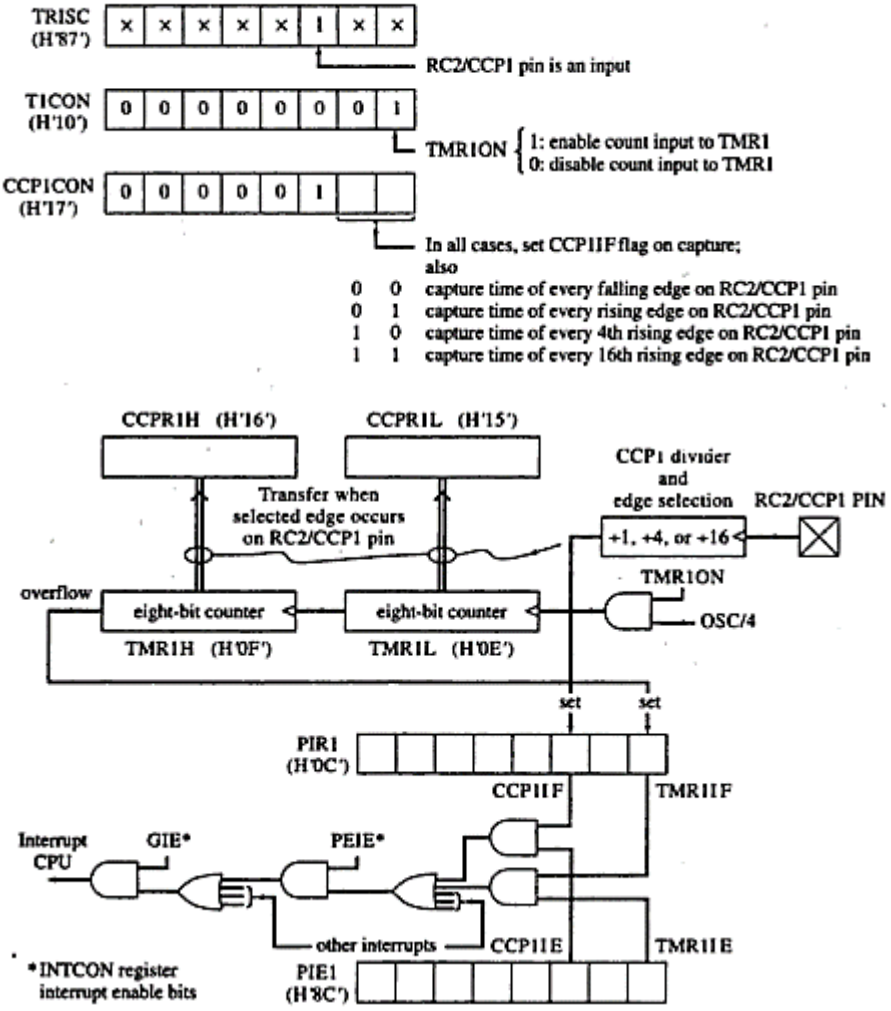


CCP2 compare mode.

Capture Mode:

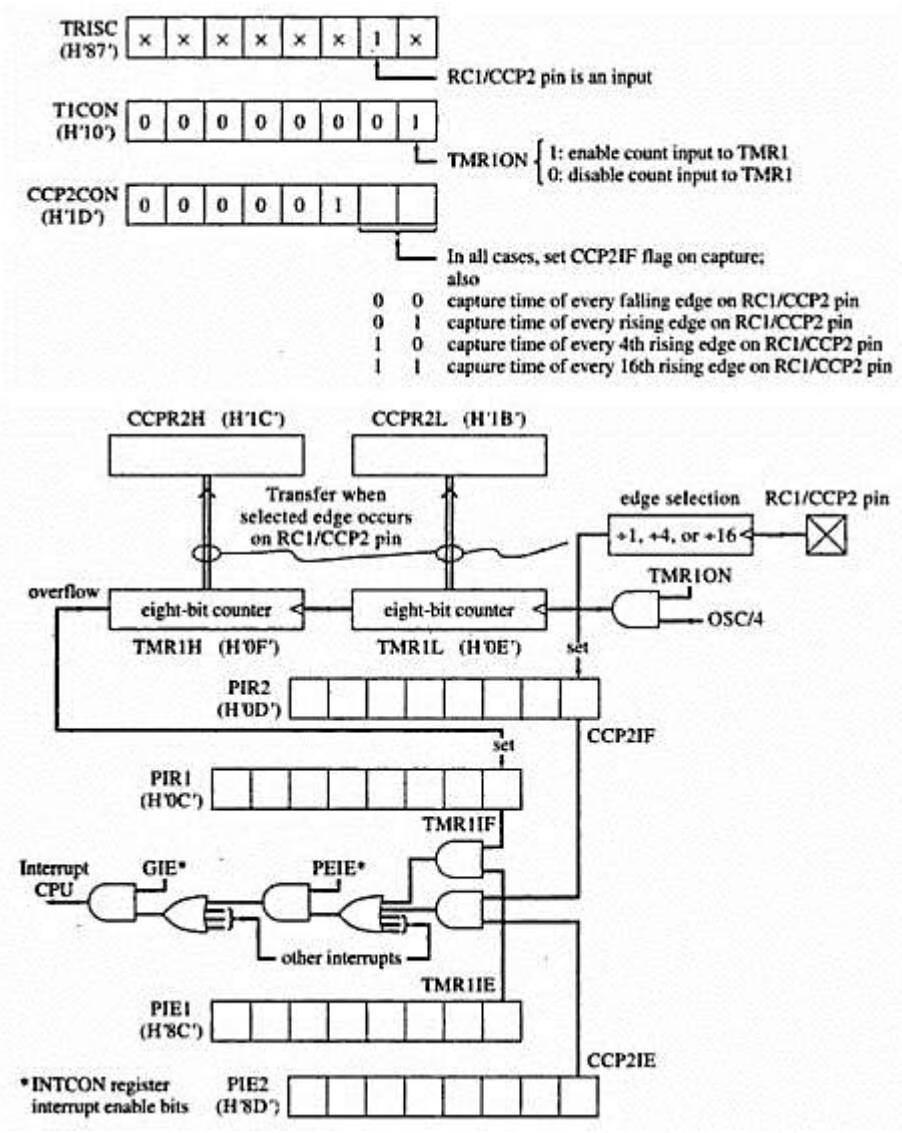
- The combination of Timer 1 and either the CCP1 or the CCP2 module permits a PIC chips to be used to determine the time of occurrence of an input edge.
- Timer 1 can be used with its prescaler to 16 bit count range measure longer intervals directly.

- But the finest resolution in the measurement will occur if prescaler is bypassed, so as an input edge will be ascertained to within 1 μs.



CCP1 Capture Mode

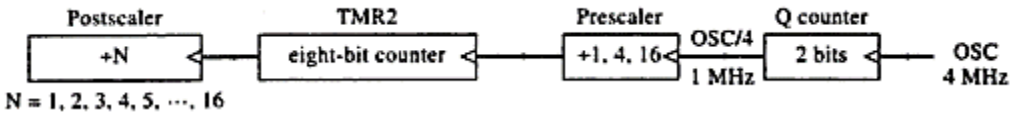
- The time between two input edges is determined by making two captures and subtracting the one time from the other.
- The circuit shown in the figure can be set up initially to capture the time of the raising edge by loading CCP1CON with 05H.
- Then the CCP1IF flag in PIR1 should be cleared and the CCP1IE bit in PIE1 set to enable CCP1 interrupts.
- GIE and PEIE interrupt enable bits in INTCON have been set in the initial subroutine.
- When the interrupt occurs on the RC2/CCP1 pin, the CCP1IF flag is cleared.
- The bit 0 of CCP1CON is cleared to set up to capture time of occurrence of the falling edge of the input.
- Finally the 2 – byte register CCPR1 is copied to a 2 – byte RAM.
- When the second interrupt occurs, the first captured value is subtracted from the newly captured value to give the pulse width.
- The CCP1IE interrupt enable bit in PIE1 can finally be cleared since the measurement has been completed.
- The circuitry for CCP2 capture mode is shown below and is similar to CCP1 capture mode.



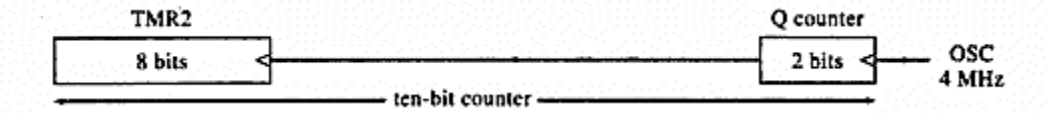
CCP2 Capture Mode

PWM Mode:

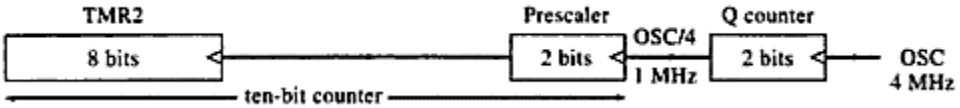
- The heart of the PIC's PWM circuit is a 10 – bit counter formed from the Timer2's 8 bit TMR2 counter from its upper bits and whatever 2 – bit counter drives it.
- These latter 2 bits depend on the prescaler setting.
- If the prescaler is bypassed (set to divide by one), then the PWM circuitry actually reaches into the 2 bit Q counter, which divides the crystal clock frequency by four to obtain the internal clock frequency.



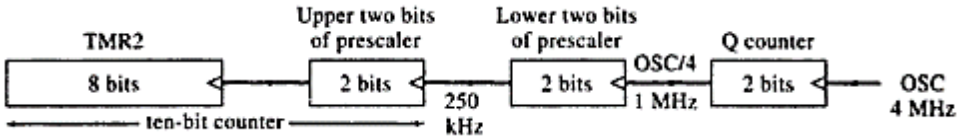
Timer2 Counting Chain



Prescaler set to divide by one



Prescaler set to divide by four



Prescaler set to divide by sixteen

- The figure shows that the period is controlled by two things: the value initialized for the Timer 2 prescaler and the value initialized to PR2.
- In addition to controlling the PWM frequency, the value loaded into PR2 controls the duty cycle resolution.
- When the circuitry turns on to the PWM mode, other circuitry will turns OFF at any desired count of the 10 bit counter.
- If PR2 is initialized to 63, the full 10 – bit counter will count with a scale of 64 x 4 = 256 or 8 – bit resolution.

Period = (PR2 +1) x [(Prescaler Value)/OSC] x 4

= PR2 + 1 μs for OSC = 4 MHz and + 1 Prescaler

= 4 (PR2 + 1) μs for OSC = 4 MHz and + 4 Prescaler

= 16 (PR2 + 1) μs for OSC = 4 MHz and + 16 Prescaler

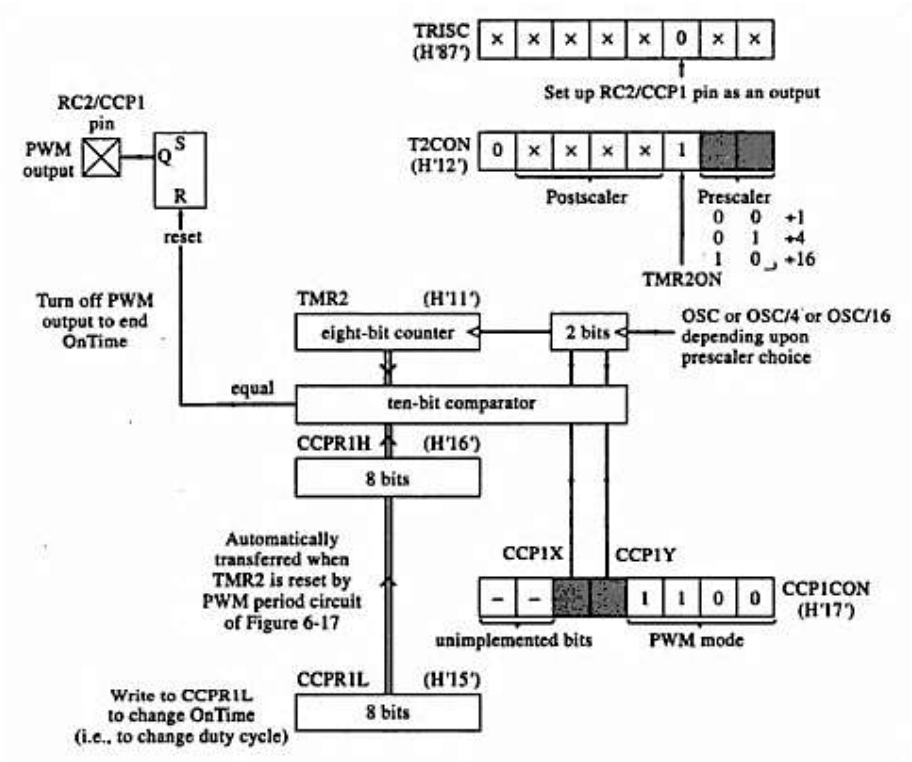
Duty Cycle Resolution	Ten – bit counter scale	PR 2 value	Prescaler = 1	Prescaler = 4	Prescaler = 16
10 bit	1024	255	3.91 kHz	0.98 kHz	244 Hz
~ 10 bit	1000	249	4 kHz	1 kHz	250 Hz
8 bit	256	63	15.6 kHz	3.91 kHz	0.98 kHz
6 bit	64	15	62.5 kHz	15.6 kHz	3.91 kHz

- From the RC2/CCP1 output, the 10 bit value is formed that turns off the PWM output, thus controlling the duty cycle value.
- The upper 8 bits of the 10 – bit value are loaded under program control into CCPRIL.
- The PWM circuitry automatically transfer this value to CCPR1H as TMR2 is reset to start each PWM period.
- This double buffering value that is actually used in the comparison is designed to help to prevent the glitch that would occur if CCPR1H were changed from H'50' to H'40'.
- In this event the PWM output would not go low at all until the next period.

Duty cycle calculation:

For Numerator ≤ Denominator, Duty cycle = $\frac{(4 \times CCPR1L) + (2 \times CCP1X) + CCP1Y}{4\,(PR2 + 1)}$

For Numerator ≥ Denominator, Duty cycle = 1.0



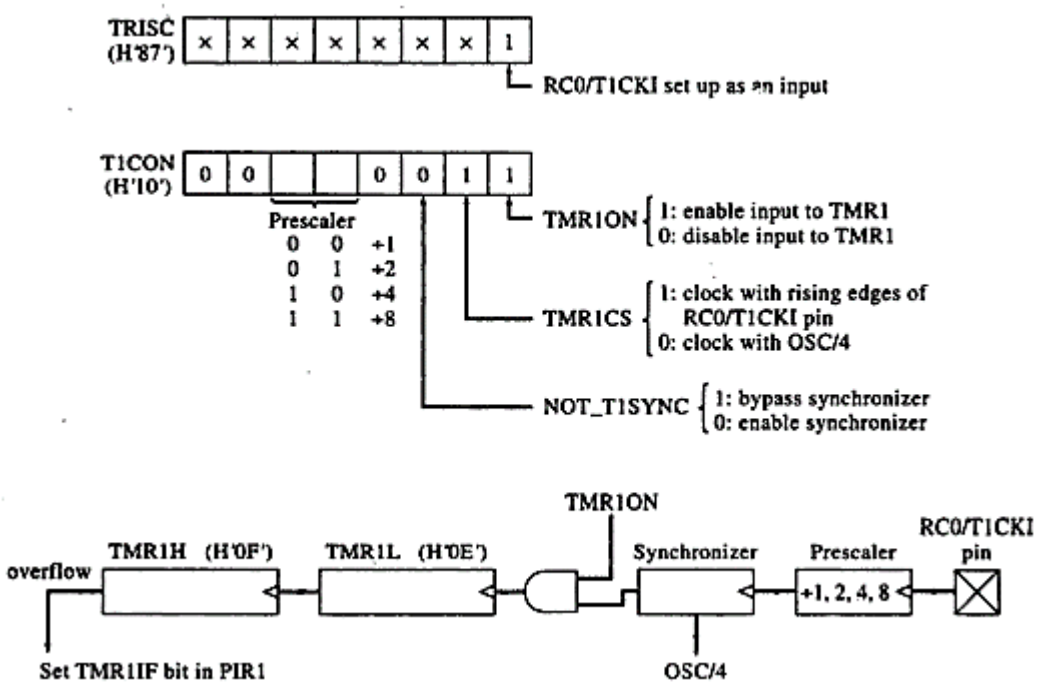
Control of PWM duty cycle on RC2/CCP1 output

- The lower 2 bit of the 10 – bit value are loaded under the program control into the bits 5 and 4 of the CCP1CON.
- If a 10 – bit value is loaded into a 2 – byte RAM variable PWM, then the code in the program will transfer it to CCPR1L and CCP1CON appropriately to vary the output over the full duty cycle range from zero to one.

```
PWM update
            rrf         PWMH, F         ; rotate bit 8
            rrf         PWML, F         ; into PWML (7)
            rrf         PWMH, F         ; rotate bit 0 into PWMH (7)
            rrf         PWML, F         ; and bit 9 into PWML (7)
            rrf         PWMH, F         ; and bits 1, 0 into PWMH (7:6)
; Upper 8 bits are now in PWML
; Lower 2 bits are in PWMH (7:6)
            rrf         PWMH, F         ; Move bits 1, 0 to align with CCP1CON
            rrf         PWMH, W         ; move to W
            xorwf       CCP1CON, w      ; Toggle if CCP1X : CCP1Y differ
            andlw       B' 00110000'    ; Force other bits to zero
            xorwf       CCP1CON, F      ; change bits that differ
            movf        PWML, W         ; Move upper eight bits
            movwf       CCPR1L
            return                      ; fourteen cycles
```

11. Explain how Timer 1 used in External event counter.

- Timer 1 is like Timer 0 can be used to count external events.
- When used with one of the CCP modules, it can generate a CCP interrupt after every N^{th} input edge, for any integer value of N up to 65, 536.
- With Timer 1's prescaler, this can be extended to every multiple of 8 up to 524,288.

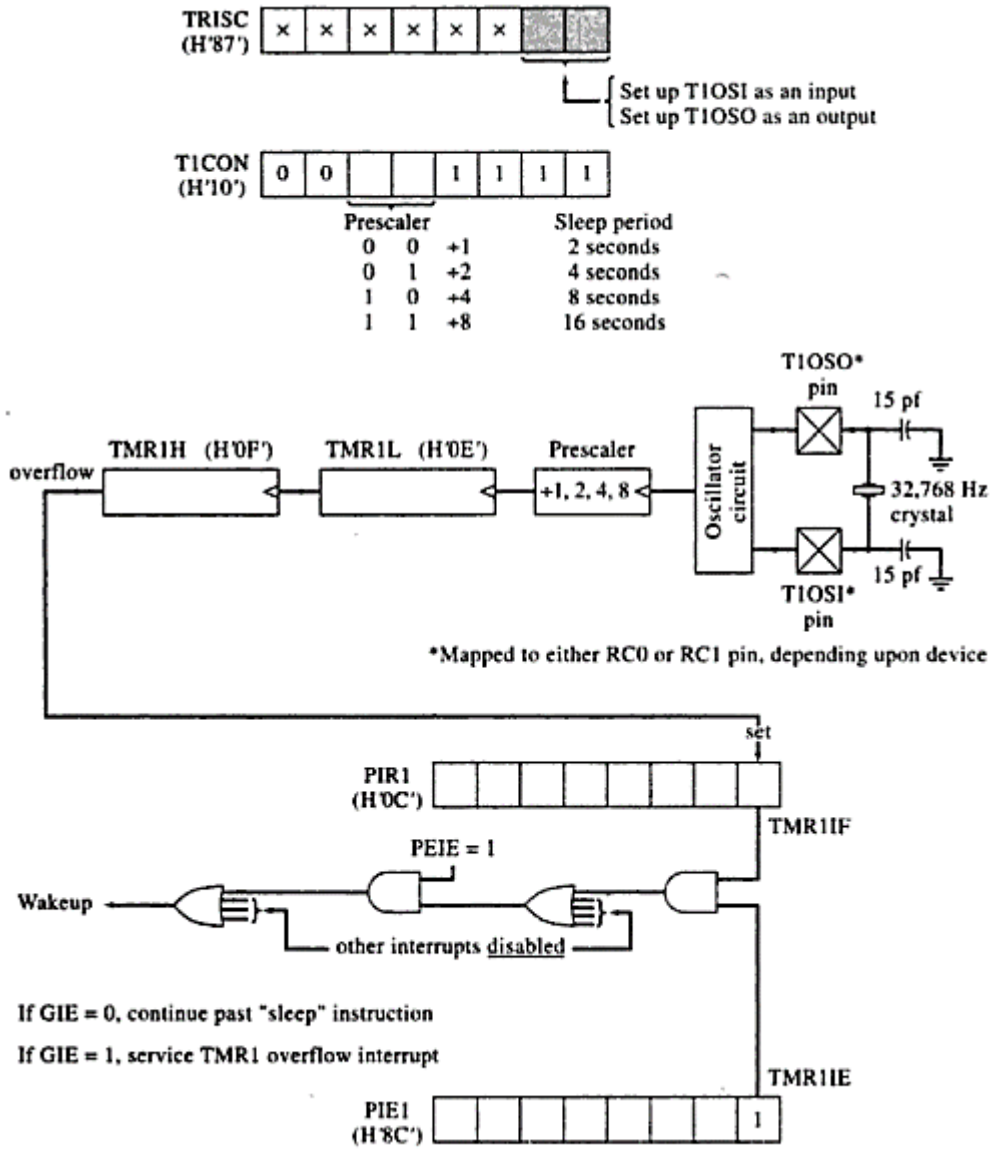


Timer External event counter

- This is all that is needed to count N evens.
- With T1CON set to H'02', the prescaler is bypassed and the input from the pin 0 of PORTC, the external clock input to Timer 1 (T1CKI) is blocked by TMRION = 0.
- To count N input rising edges, TMR1 is present to 65,536 – N, the TMR1IF flag in the PIR1 register is cleared and TMR1IE interrupt enable bit in the PIE1 register is set.
- Finally, counting is begun by setting the TMR1ON bit in the T1CON register.
- After N rising edges on the input pin, RC0/ T1CKI, Timer 1 will generate an interrupt permitting the desired action to be taken at that time.
- The synchronizer, synchronizes the input to the internal clock.
- It is an optional features, controlled by the NOT_T1SYNC bit (bit2) of the T1CON register.
- Synchronizing the external input to the internal clock permits TMR1 to be read from and written to even as the counter is counting.
- Synchronization is also vital to proper operation when /timer 1 is used with one of the CCP modules while counting external events.
- The state of the NOT_T1SYNC bit is ignored and the synchronizer bypassed when TMR1CS = 0 selects the internal clock, OSC/4.

12. Explain how Timer 1 is acting as Sleep Mode.

- One option that PIC chips make available to users is the ability to stop the internal clock (OSC/4), reducing power consumption and yet have an accurate internal time base.
- Timer 1 includes the pins and the oscillator circuit to allow 32768 Hz crystal to serve as its external clock source.
- Since the synchronizer will not produce output pulses with the OSC/4 internal clock stopped, the synchronizer must be bypassed.
- TMR1 will overflow at 2 – second, 4 – second, 8 – second or 16 – second intervals depending on which prescaler value is used.



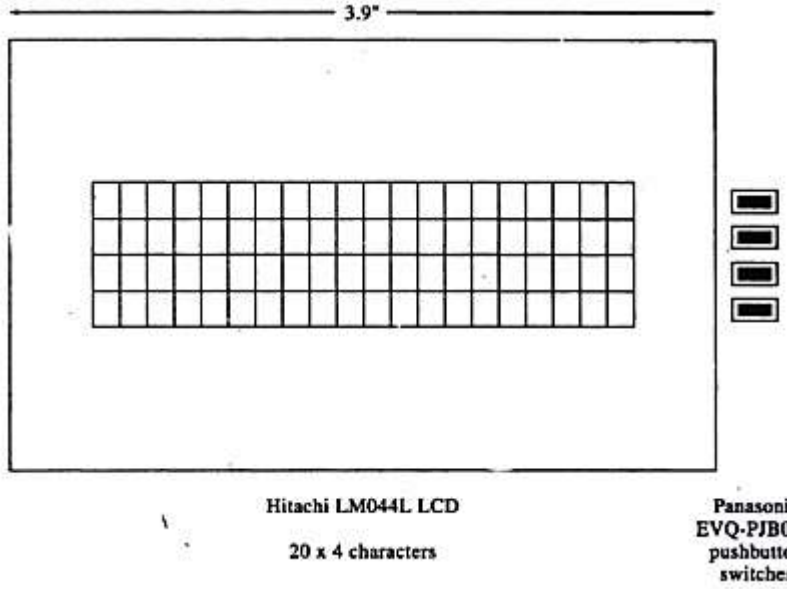
Timer1 + 32,768 Hz crystal clock source to sleep, waking up periodically.

- Each time that a TMR1 overflow occurs the CPU initiates the startup of the internal clock which may take as long as 1000 internal clock cycles before the next instruction executed.
- If GIE bit had been cleared before the sleep instruction had been executed, then about a millisecond after TMR1 overflows, the CPU will continue execution with the code that follows the sleep instruction.
- If GIE = 1, then about a millisecond after TMR1 overflows, the CPU will execute the one instruction that follows the sleep instruction and the vector to the interrupt service routine at address H'004'.

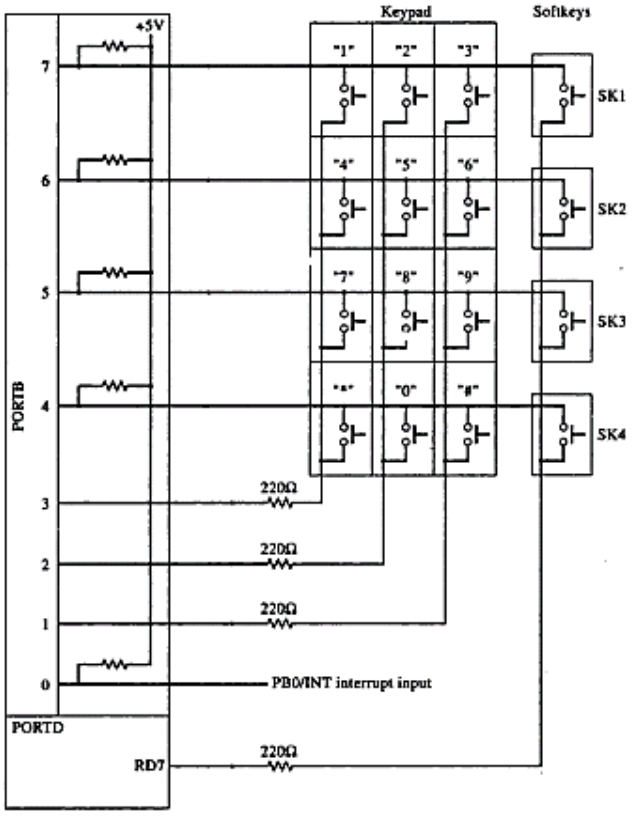
13. Write short notes on Softkeys?

- A multiple – line LCD display used with miniature pushbutton switches that are aligned to the right of each line of the display.
- The label for each switch can be displayed and changed at the right edge of the row corresponding to that switch.
- For example, if a device measures the temperature at six points in a test bed, then it might permit a user to cycle among these by repeatedly pressing the bottom softkeys, displaying

"Temp1=", "Temp2=", … "Temp6=" as its softkey label and displaying corresponding two digit temperature to the right of this on the bottom row.



- The circuit shows how the four softkeys can be treated in the same manner as the 12 keys of a keypad.
- In fact keyswitches are generally grouped into an array such as whether or not they physically grouped together in a keypad.
- The figure shows a pin from a separate port bit 7 of PORTD, being used to drive the column of softkeys.
- If Bit 0 of PORTB is not used as an interrupt input, then it actually makes sense to use that pin to drive the column of softkeys.



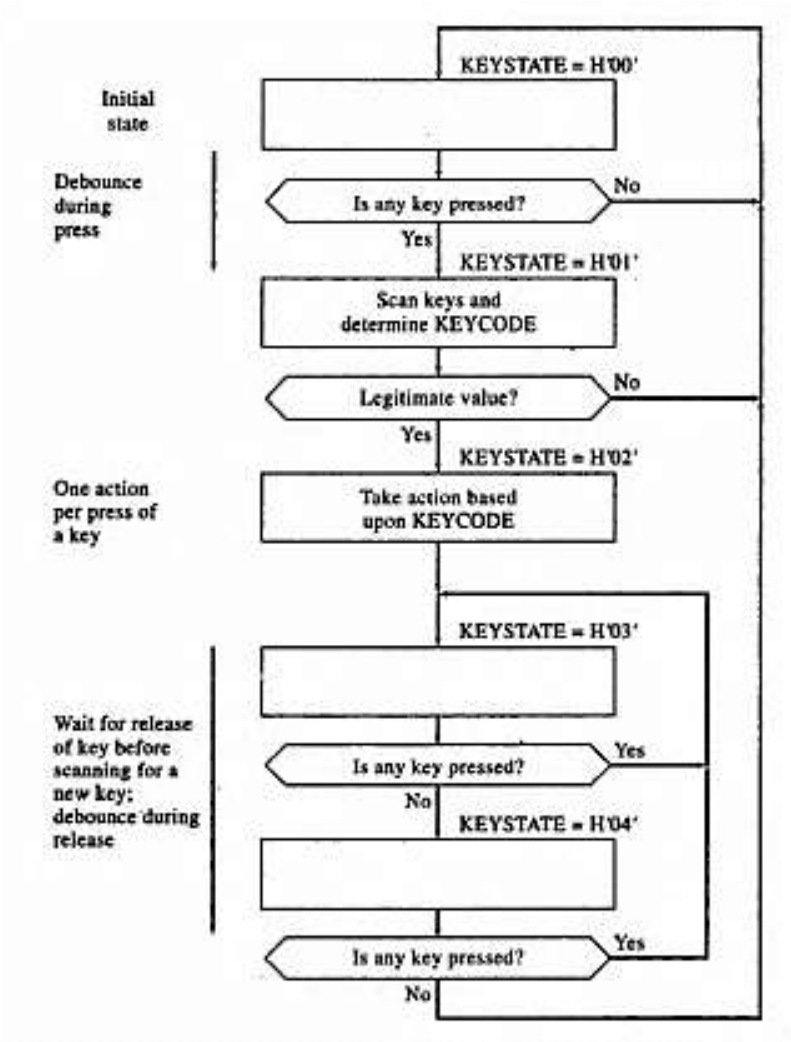
14. Mention the working of State Machines And Keyswitches in PIC

- Because keyswitches are not changed very fast, they can be checked once each time around main in loop in a keyswitch subroutine.
- Recall that a looptime of 10ms was selected because the maximum keybounce times of most mechanical keyswitches is less than 10ms.
- Consequently if keyswitch across that key is newly pressed, it can be assured that the next time it is called 10ms later, any bouncing of the key contacts will have settled out, with the contacts family closed.
- The keyswitch algorithm of fig shows the STATUS register's Z bit upon returning from the Any key Subroutine.
- If Z=1, a return from the keyswitch subroutine occurs.
- On the other hand if Z=0, a key is newly pressed so KEYSTATE is incremented to H'01 before returning from the subroutine.
- Ten millisecond later the keyswitch subroutine is recommended this time with KEYSTATE =H'01 a key press was detected last time.
- By now any keybounce has settled out.
- A scankey subroutine is called.
- It returns with Z=1 and a KEYCODE RAM variable loaded with a value thata identifies the pressed key.
- If for any return it could not identify the pressed key, it returns with Z=0 in the STATUS register.
- A table driven implantation of the SCANKEY subroutine is listed in fig.
- It seems that the keys in the order of their KEYCODE value 0, 1, 2, 3 …15.
- For each value, a corresponding table entry is taken from Scankeys table.
- The lower 4 bits of the table entry used to drive the column of the selected keys low and the other column high.
- In this way the only keys that can drive one of the 4 upper bits of PORTB low are the four keys in the column.
- The Scankeys subroutine matches what is read back from the upper 4 bits of PORTB against the upper 4 bits of the table entry.
- If a match does not occur, the next key is checked up, if a match does occur for any of the 16 keys, the subroutine occurs with Z=0.
- The 220Ω resistor in fig are there to protect the PIC'S output drives during the execution of Scankeys if two keyswitches in the same row are pressed simultaneously.
- In this aberrant case, two output drivers will be shorted together.
- A high output will be shorted to a low output during the testing of half of the keys.
- The 220Ω resistors limit the current to less than 10mA when this happens.
- Upon the return from the scankeys subroutine, the keyswitch subroutine tests the STATUS register Z bit.
  - If Z=1, it increments KEYSTATE and returns, prepared to act on the pressed key in 10ms.
  - If Z=0 then somehow Scankeys failed to identify a pressed key.
  - This might occur if, for example two keys in the same column are pressed simultaneously.
  - In that case, there will be no entry in Scankeys_table that matches what is read from PORTB.
  - However this failure occurs, KEYSTATE is cleared to zero, starting over again in the hunt for a pressed key.
  - When keyswitch is called with KEYSTATE =H'02', it increment KEYSTATE.

- If it does a

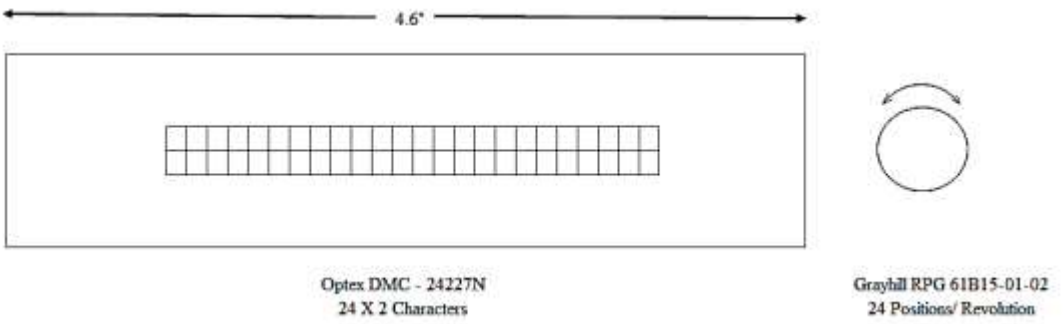
  Goto keyaction

- Instruction, the job of this keyaction subroutine is to carry out the proper response for the pressed key.
- The return from keyaction will pop the return address and actually execute a return from Keyswitch.
- The last two states of the keyswitch algorithm require the pressed key to be released during two successive passes around the mainline loop.
- This overcomes any potential problem with keybounces time during the release of a key.



15. Mention how display plus RPG used in PIC

- The Grayhill rotary pulse generator differs from Hewlett – Packeard RPG in two ways.
- It features 24 detected positions per revolution, so it click from one position to next, giving the feel of rotary switch.
- It also features an integral momentary – action pushbutton switch.
- When used with an alphanumeric display, the RPG's knob can be pushed and released to cycle the display among the instrument's setup parameter.
- Stopping at a specific setup parameter, the RPG can be rotated to increase or decrease the value of the parameter.

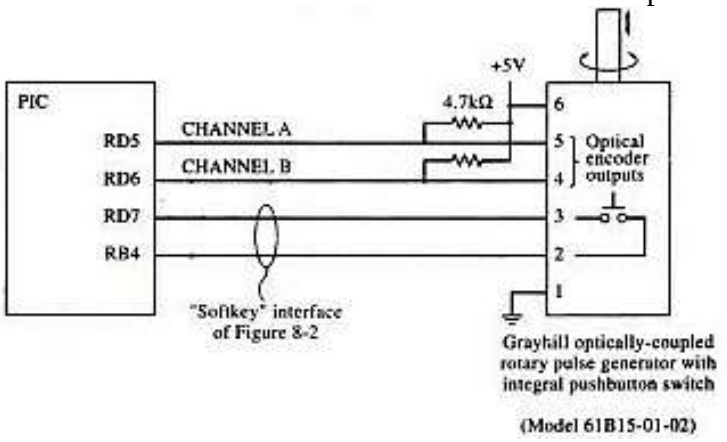


Front Panel Appearance

Row: \ Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90	91	92	93	94	95	96	97
2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D2	D2	D3	D4	D5	D6	D7

LCD display's cursor – positioning codes
LCD display plus RPG with integral pushbutton switch.

- The PIC interface for the Grayhill RPG is shown in the figure below.
- The momentary action pushbutton switch can be treated in the same way as one of the softkeys.
- Then it will be checked, debounced and handled at regular 10 ms intervals.
- The two RPG outputs must be treated differently.
- Hewlett RPG used one of the RPG outputs as a PB0/INT interrupt input.
- In Grayhill RPG, action is needed whenever either output changes, not just one of the outputs produces a rising edge.
- The two RPG outputs can be checked in an RPG subroutine called from the mainline loop. The two inputs are compared with their values 10 ms.
- The selected parameter can be incremented or decremented if the inputs have changed.



- If the number of 10 ms intervals, since the last change are counted in an RPG_INVERVAL RAM variable, the selected parameter can be changed in large steps when the RPG is turned quickly and incremented or decremented when it is turned slowly.
- For this application, the incrementing of RPG_INTERVAL must be stopped when it reaches H'FF' and reset to zero each time the RPG output changes.

16. Briefly explain the Display of Variable Strings in PIC controller? *[Apr'17]*

- When entering the setup parameters with either a keypad or an RPG and when displaying the results of an instruments measurement, it is necessary to write a string of ASCII – coded characters to the display, since the character are taken from the RAM.
- The assignment of a string variables, VSTRING to RAM with Microchip's MPASM assembler is
  - Cursor – positioning code
  - ASCII string of characters to be displayed
  - End – of – string designator.
- Given strings having this format, a DisplayV subroutine can be written to send the first character to the LCD display as a command (RS = 0) and subsequent characters as displayable characters (RS = 1) and to stop when H'00' is assessed.
- If the maximum number of characters ever to be sent to the display from the string is 10, then 12 RAM locations must be reversed for the 10 displayable characters, the leading cursor – positioning code and the trailing end – of – string designator.
- This reservation of RAM for a string called VSTRING,

```
VSTRINGlength     equ    12      ; Maximum number of character in
                                   VSTRING.

Cblock          BANK0RAM
-
-
-                                 ; Variables
-
VSTRING:  VSTRINGlength
Marker
endc
if              Marker > H'80'
        error "RAM use exceeds Bank  0"
endif
```

- The MPASM assembler permits code writers to create error messages to warn of error – producing conditions.
- If the number of variables eventually added to the cblock…..endc construct plus those reserved for VSTRING push the reserved memory past the end of Bank0, this will not happen without notice being given to the code writer.

ANNA UNIVERSITY QUESTIONS
PART A

1. Mention the interrupts available in PIC microcontroller?*[Apr'17]*
2. Define subroutine. *[Nov'17]*
3. What is the minimum and maximum clock frequency of PIC 16cxx ? *[Nov'16]*
4. What is the role of TRISx register in I/O port management? *[Nov'16]*
5. Write an ALP to initialize the PORTA using PIC microcontroller. *[Apr'17]*
6. What is the necessity of prescaler in the timer operation? *[Apr'18]*
7. Elaborate the CCP module of PIC. *(Jun'12)*
8. What do you mean by state machine? *[Nov'17]*
9. How to display constant strings? *[Apr'18]*

PART B

1. Explain in detail about the PIC Microcontroller Interrupts? *[Nov'16, Apr'17, Apr'18]*
2. Explain the interrupt structure of PIC microcontroller with neat diagram. *[Nov'17]*
3. Explain how Intservice Interrupt Service Routine is achieved in PIC. *[Nov'16, Apr'17, Apr'18]*
4. What is meant by External Interrupts and explain how RB0/ INT External Interrupt Input is connected with PIC. *[Apr'18]*
5. How many timers are present in PIC16F8XX timer modules? Explain them? *[Nov'16]*
6. In detail give an account on Timer programming. RAM/ROM allocation in PC. *[Nov'17]*
7. Explain the modes of Timer 1 of PIC16C6x microcontroller with block diagram. Also explain the function of associated registers. *[Apr'18]*
8. Briefly explain the Display of Variable Strings in PIC controller? *[Apr'17]*

UNIT – III PART-A

1. What are the two modes of operations of SSP?

- Serial Peripheral Interface (SPI)
- Inter – Integrated Circuit (I^2C)

2. What is the use of I^2C bus?

- The Inter – Integrated Circuit (I^2C) bus is two wire serial interface developed by the Philips Corporation.
- This device will communicate with both standard and fast mode devices if attached to the same bus.

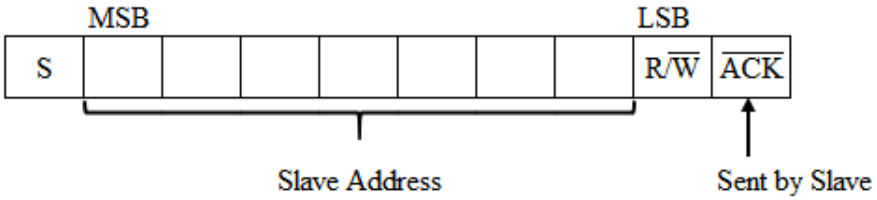
3. What is the role of master and slave operation performed by the I^2C?

- The I^2C interface uses two bi – directional lines meaning that any devices could drive either line.
- The master and slave are always in opposite modes of operation during a data transfer.
- They can be brought into two modes of operation:
  - Master – transmitter and Slave – receiver
  - Slave – transmitter and Master – receiver

4. What is the function performed by the I^2C?

- The SSP module in I^2C mode fully implements all slave functions.
- Except general call support and provides interrupts on start and stop bits in hardware to facilitate firmware implementations of the master functions.

5. Illustrate I^2C seven bit address format.



S – Start Condition

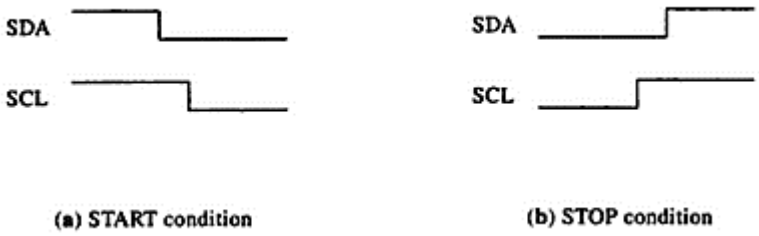
6. Difference between bus operation and subroutine operation. *[Nov'17]*

Bus Operation	Subroutine Operation
The open – drain outputs for SCL and SDA pins are achieved in a way that could use any one of the PIC's I/O pins	The SCL pin must have open – drain output while SDA pin must be either an input or have an open – drain output.
To change the output from 0V to a high impedance output, instead of wring 1 to PORT C bit, 1 is written to corresponding TRISC bit.	I^2C bus subroutines will repeatedly access TRISC, data direction register for PORTC.
TRISC is located at Bank 1 which cannot be accessed by direct addressing by first executing the instruction	Indirect pointer FSR, with the address of TRISC can be loaded first and then required bit setting and bit clearing of TRISC bits can be do indirectly.

7. Define START and STOP signals of I^2C?

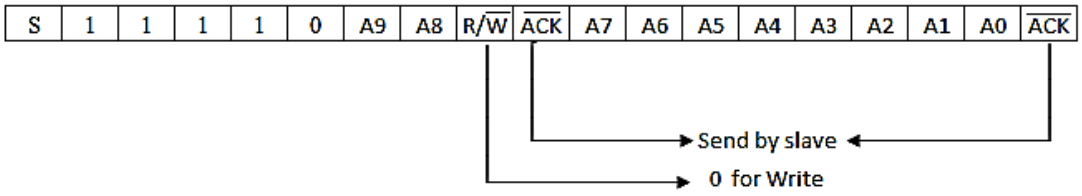
- START Condition (S) – SCL =1, SDA Falling edge.
- STOP Condition (P) – SCL =1, SDA Rising edge.

8. Draw the start and stop conditions of I^2C. *[Apr'18]*



I^2C START and STOP condition

9. Illustrate I^2C 10 bit address format.



S – Start Condition
R/$\overline{W}$ – Read / Write Pulse
$\overline{ACK}$ - Acknowledge

10. Mention the two conditions for utilization of FSR?

- If these I^2C subroutines are executed from the mainline program, then any interrupt service routine that also used FSR must set it.
- Any use of indirect addressing to access a sequence of addresses in the PIC's RAM when used in conjunction with these I^2C subroutines must swap pointers in and out of FSR.

11. What is EEPROM?

- Electrically Erasable Programmable Read – Only Memory (EEPROM) is a nonvolatile storage technology of variables to a PIC controlled device.
- The variables stored in an EEPROM will remain there even if power supply has to be turned OFF and then ON again.

12. Mention the PIC microcontroller RAM and EEPROM size.

RAM size:

- The PIC microcontroller RAM size is also important as it stores all your variables and intermediate data.
- For example don't use floating point alter it to use a different variable type e.g. you can use long integers with fixed point operation to avoid floating point.

EEPROM size:

- Electrically Erasable ROM is used to store data that must be saved between powers up and power down.
- This area is readable and writable and has a much longer life than the main program store i.e. it has been designed for more frequent use.

13. What is ADC?

- The A/D allows conversion of an analog input signal to a corresponding 8 – bit digital number.
- The analog – to digital (A/D) converter module has the capabilities of five inputs to eight for the different varieties of PIC16CXX.

- An adjustable sampling rate.
- Internal or external reference voltage option.
- 8 – bit conversion
- Interrupt capability at the end of conversion.

14. Using PIC micro controller how is analog signal converted into digital. *(Jan'13)*

- ADC to translate the analog signals to digital numbers.
- So, that the microcontroller can read and process them.

15. List the various registers used in A/D conversion.

- A/D result register (ADRES)
- A/D Control Register 0 (ADCON0)
- A/D Control Register 1 (ADCON1)

16. What is the function of TRISA pin?

- Setting TRISA bit will configure PORTA as input and resetting will configure as output port.

17. Write a program to initialize portA.

```
Org0
Bcf     STATUS.RP0
clrf    PORTA
bsf     STATUS.RP0
movlw   00010000H
movwf   TRISA
End
```

18. Explain about UART?

- Universal asynchronous receiver transmitter.
- UART is useful for receiving and transmission of datas in asynchoronous mode.

19. List the register associated with UART? *(Nov'16, Nov'17)*

Synchronous UART:

- TRISC Register
- TXSTA: Transmit status and control register.
- RCSTA: Receive status and control register
- SPBRG: Baud Rate generation register.
- INTCON Register

Asynchronous UART:

- PIR1
- RCSTA
- RCREG
- TXREG
- PIE1
- TXSTA
- SPBRG

20. What is synchronous and asynchronous transmission?

- Asynchronous – start and stop bit allowed for transmission of data.
- Synchronous – no start and stop bit only block header data

21. Define baud rate. *[Nov'17, Apr'18]*

- A number related to the speed of data transmission in a system. The rate indicates the number of electrical oscillations per second that occurs within a data transmission. The higher the baud rate, the more bits per second that are transferred.

22. What is baud rate in asynchronous mode?

- The baud rate in asynchoronous mode is given by

$$BR = \frac{F_{osc}}{64\,(x+1)} \text{ for low speed}$$

$$BR = \frac{F_{osc}}{16\,(x+1)} \text{ for high speed}$$

20. What is the value to be loaded into SPBRG register if we want 19200 baud rate with 10MHz clock source. *(Nov'16)*

$$\text{SPBRG} = \frac{F_{osc}}{16 \times baud\ rate} - 1 = \frac{10x10^6}{16x19200} - 1$$

= 32.5~32

21. How do you configure the ports as input and output?

- Any ports can be made as input by setting the port bits and they can be set as output by resetting the port bits.

22. How can the LCD be tested whether it is ready or not to receive a command or data? *(Jun'12)*

The steps that has to be done for initializing the LCD display is given below and these steps are common for almost all applications.

- Send 38H to the 8 bit data line for initialization
- Send 0FH for making LCD ON, cursor ON and cursor blinking ON.
- Send 06H for incrementing cursor position.
- Send 01H for clearing the display and return the cursor.

Sending data to the LCD.

The steps for sending data to the LCD module are given below. It is the logic state of these pins that make the module to determine whether a given data input is a command or data to be displayed.

- Make R/W low.
- Make RS=0 if data byte is a command and make RS=1 if the data byte is a data to be displayed.
- Place data byte on the data register.
- Pulse E from high to low.
- Repeat above steps for sending another data.

23. What is interfacing? *(Jan'14)*

- Interfacing means connecting microcontroller with external devices.
- The external devices are Input devices, output devices, Memory chip and External applications.

24. While programming for LCD display, what initialization has to be done? *(Jan'13)*

- When D7=0, the LCD is ready to receive new information.

25. What is the need for D/A converter? *(Apr'11)*

- D/A converter (Digital to Analog Converter) is used to interface the microprocessor output with the external device.

26. Microcontroller based control is advantageous than conventional control-Justify. *(Apr'17)*

- Flexibility
- Fast speed of Execution
- Inexpensive
- Rigid
- Labour saving

27. How is temperature sensor interfaced with PIC microcontroller? *(Apr'17)*

- The temperature sensor is interfaced in LM chip includes a thermal watchdog that can be setup to interrupt PIC on its RBO/INT Edge –triggered interrupt input when the temperature rises above a programmable TOS.

PART B

1. What is meant by I^2C module? Explain how I^2C is interfaced with PIC microcontroller. *[Nov'16]* [OR] **Exhibit the operation of I2C bus and develop embedded C program to transmit data using I^2C bus.** *[Nov'17]*

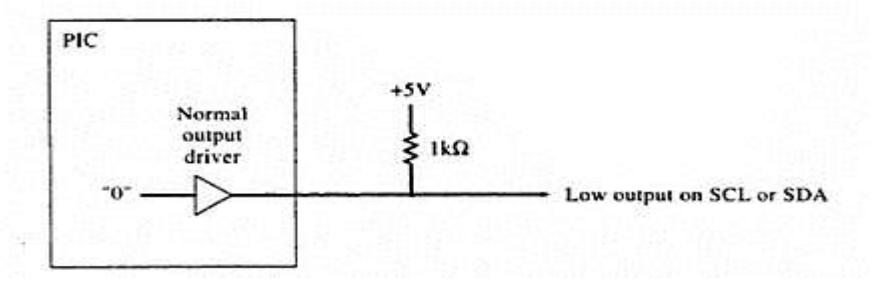
I^2C Bus for Peripheral chip Access:

- The I^2C bus, developed by Philips Semiconductor, provides a two wire bidirectional interface to a variety of chip that can serve as powerful adjuncts to a PIC.
- It can also serve as the means for connecting a master PIC to one or more slave PICs using only two wires for the connections.
- The I^2C bus will be seen through its connection to three small (eight pin), low cost parts:
  - A dual 8-bit digital to analog converter.
  - A 9 bit temperature sensor
  - A 128 byte serial EEPROM
- To Write data to one of these parts, the PIC will bit-bang the two I^2C pins on PORTC, transferring out
  - A peripheral chip address and read/write bit designating that the peripheral chip is to read successive bytes.
  - A peripheral internal register or address byte.
  - Data to write into one or more consecutive internal addresses
- To read data to one of these parts, the PIC
  - Sends out a peripheral chip address and a read/write bit designating that the peripheral chip is to send successive bytes beginning at a previously selected internal register or address.
  - Reads back one or more bytes of data
- Inspite of its relatively slow speed, the I^2C bus interface is widely used for many applications where its speed is still much faster than an application requires.

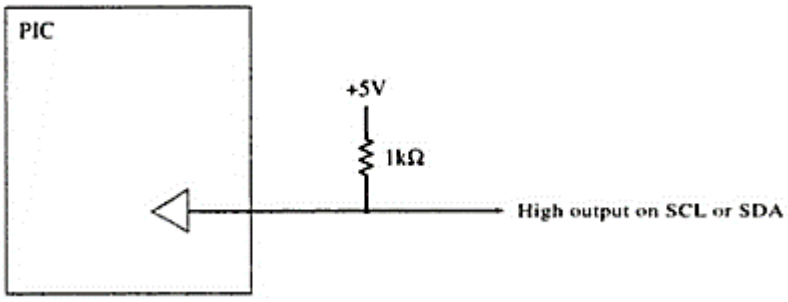
- Example: reading the temperature from a transducer having thermal time constant measured in second.
- Also, once the bit-banging subroutine have been written accessing an additional I^2C device simply requires attaching it to the same two I^2C lines going to all other I^2C peripheral chips and then calling the same bit banging sub-routine used with other chips.

I^2C Bus Operation:

- It requires two open –drain I/O pins.
- These two pins, called SCL (serial clock) and SDA (serial data) are implemented on the PIC chips as the two multipurpose pinRC3/SCK/SCL and RC4/SDI/SDA respectively.
- The open - drain outputs for SCL and SDA pins are achieved in a way that could use any of the PICs I/O pins as shown in fig

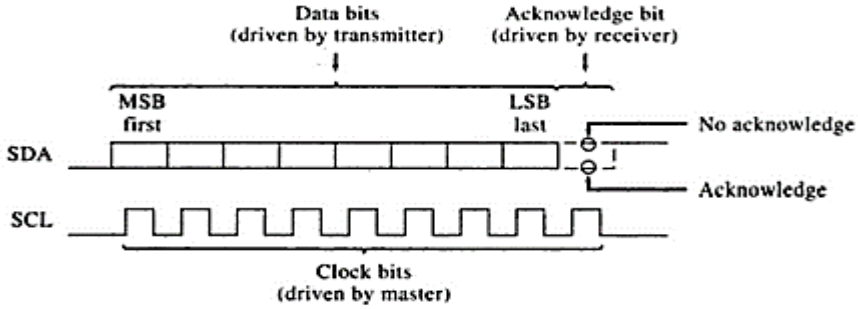


I/O pin set to be an output with "0" written to it



I/O pin set to be an input - Implementation of open-drain output

- To change the output from 0V (fig I/O pin set to be an output with "0" written to it) to a high impedance output, instead of writing a one to the PORTC bit, a one is written to the corresponding TRISC bit, thereby obtaining the high impedance by turning the pin into a high –impedance input pin.
- Whereas any of the PICs I/O pins could be used to implement the SCL and SDA pins.
- There are two good reasons to use the RC3/SCK/SCL and RC4/SDI/SDA pins:
  - The I^2C circuitry controls the slope of the output changes on these pins to meet the I^2C bus specification
  - If an application utilizes this PIC as a slave to another PIC, these are the same two pins that the I^2C slave mode uses automatically.
- The I^2C bus protocol includes a variety of features that are not needed for peripherals chip access. Eg. Multi - master control.
- Transfer on the I^2C bus take place 9 bits at a time as shown in fig. below.
- The clock line, SCL is driven by the PIC chip which serves as bus master.



Byte transfer plus acknowledge

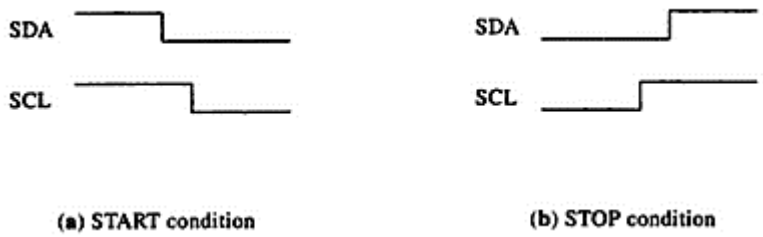
- The open drive features of every chip's bus drive can be used by the receiver to hold the clock line low, thereby signalling the transmitter to pause until the clock line is released by the receiver.
- The open drain features is also needed if this PIC will ever become an I^2C slave to another PIC, in which case it must relinquish control of the SCL line
- Fig illustrates that the first eight data bits on the SDA line are sent by the transmitter, whereas the ninth acknowledge bit is a response by the receiver.
- For example, when the PIC sends out a chip address, it is the transmitter while every other chip on the I^2C bus is a receiver.
- During the acknowledge bit time ,the addressed chip is the only one that drives the SDA line, pulling it low in response to the master's pulse on SCL, acknowledging the reception of its chip address.
- When the byte transfer represents data being returned to the PIC from a peripheral chip. it is the peripheral chip that drives the eight data bits in response to the clock pulses from the PIC.
- In this case, the acknowledge bit is driven in a special way by the PIC, which is serving as receiver but also as bus master.
- If the peripheral chip is one that can send the contents of successive internal addresses back to the PIC (a serial EEPROM) then the PIC completes reception of each byte and signals a for next byte by pulling the SDA line low in acknowledgement.
- After any number of bytes has been received in this way from the peripheral the PIC can signal the peripheral to stop any further low in acknowledgement.

Byte transfer plus acknowledge:

- Figure also illustrates that the data bits on the SDA line must be stable during the high period of the clock.
- When the slave peripheral is driving the SDA line, either as transmitter or acknowledger, it initiates the new bit in response to the falling edge of SCL, after a specified hold time.
- It maintains that bit on the SDA line until the next falling edge of SCL, again after a specified hold time.
- When the PIC master is driving the SDA line, it must meet the same hold time specification when it changes SDA after driving SCL low.
- In addition, it must meet several other timing specification as it changes SCL and SDA**.**

I^2C START and STOP conditions:

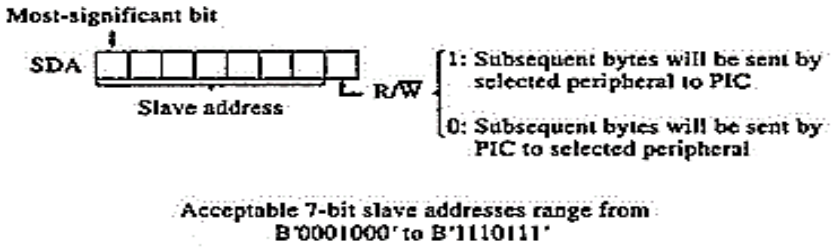
- I^2C bus transfer consist of number of byte transfer framed between START condition and either another START condition or STOP condition.
- When bus transfer are not taking place, both the SDA and the SCL lines are released by all drivers and float high.



I²c START and STOP condition

- The PIC (I^2C bus controller) initiates a transfer with the START condition.
- It first pulls SDA low and then it pulls SCL low, as shown in fig a.
- Likewise, the PIC terminates a multiple –byte transfer with the STOP condition. With both SDA and SCL initially low, it releases SCL and SDA, as shown in fig b.
- Both of these occurrences are easily recognized by the I^2C hardware in each peripheral chip since they both consists of a change in the SDA line while SCL is high, a condition that never happens in the middle of a byte transfer.

First byte of a message string:



First byte of a massage string

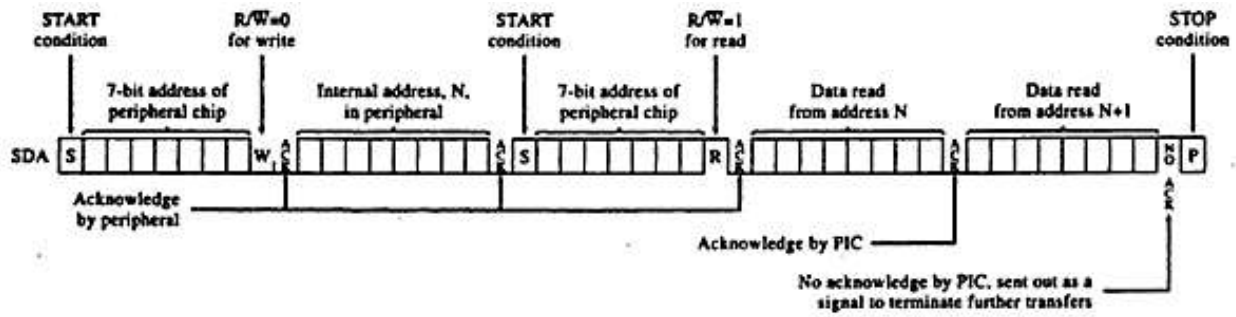
- The PIC I^2C bus master generates the first byte after the START condition. It consists of a 7-bit slave address followed by an R/W bit as shown in fig.
- If the R/W bit is low, subsequent bytes transmitted on the bus will be written by the PIC to the selected peripheral.
- If the R/W bit is high, Subsequent bytes is sent by the selected peripheral and read by the PIC.
- The I^2C bus standard was augmented with the definition of 10 bit addresses that begin with what looks like a nonstandard 7 bit address, B'11110xx'.
- The last two bits of this 7bit address plus a second 8 bit address byte form the 10 bit address in the augmented standard.

I^2C typical message string formats:

- The functions of the bytes that follows the first or control byte are defined by the needs of the peripheral chip.
- For the peripheral chip that contains more than one internal register or memory address, the PIC will typically write a second byte to the chip to set a pointer to the selected internal register or address.
- Subsequent bytes in the message string will typically be written to that address and then to the consecutive addresses that follow it .This is given in fig. a.
- A message string for reading internal peripheral register or addresses is shown in fig. b.

- It begins with a 2 –byte message string that selects the internal address of the selected peripheral chip

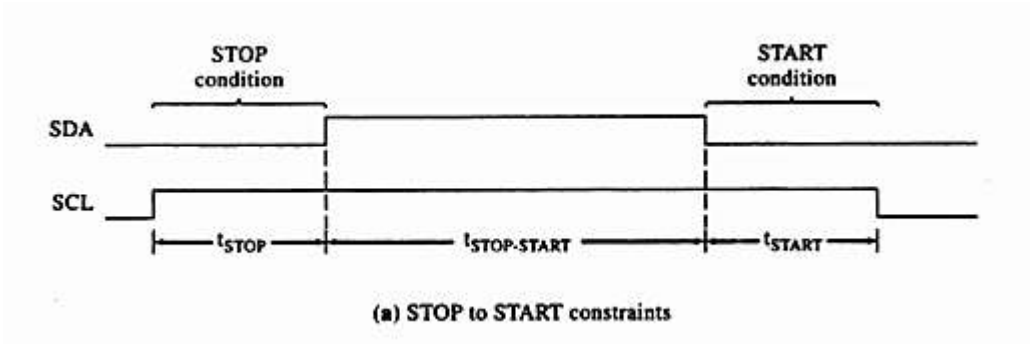
General format to write to several peripheral internal registers or address



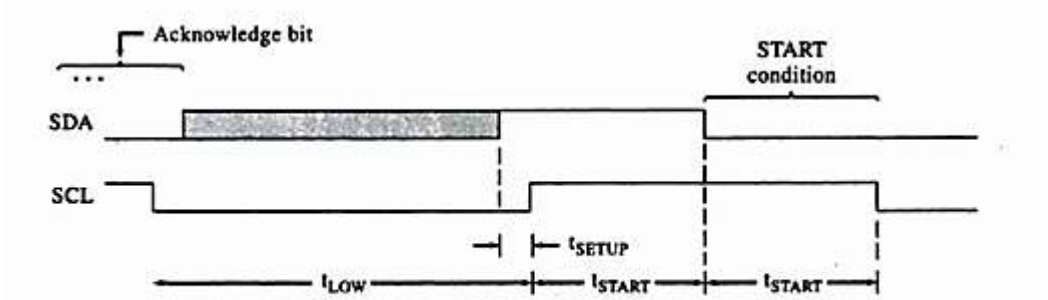
General format to read from several peripheral internal registers or address

- Then a second START condition initiates a new message string.
- The first byte of this new message string again selects the same peripheral chip but signals that the subsequent byte are to consists of reads from successive addresses in the peripheral chip.

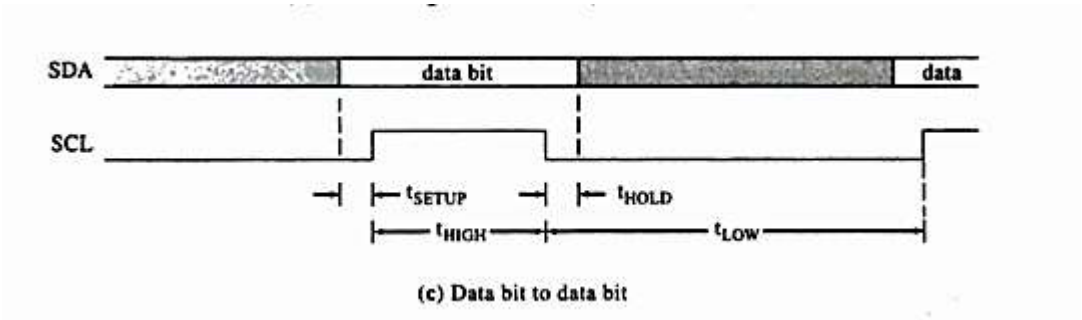
I^2C bus fast-mode timing constraints:



(a) STOP to START constraints



(b) Acknowledge bit to START (restart condition)



(c) Data bit to data bit

Parameter	Constraint	Cycles required to meet constraint		
		OSC = 4 MHz Period = 1µs	OSC = 10 MHz Period = 0.4µs	OSC = 20 MHz Period = 0.2µs
t_{START}	>0.6µs	1	2	3
t_{SETUP}	>0.1µs	1	1	1
t_{HIGH}	>0.6µs	1	2	3
t_{HOLD}	>0µs	1	1	1
t_{LOW}	>1.3µs	2	4	7
t_{STOP}	>0.6µs	1	2	3
$t_{STOP-START}$	>1.3µs	2	4	7

(d) Cycles required

I^2C bus fast –mode timing constraints

- I^2C bus specification includes the timing constraints for older chips designed for the maximum bit rate of 100kbit/s.
- It is also includes the constraints for newer fast-mode 400 kbit/s parts.
- The timing diagram of fig. a through fig. c defines timing parameters.
- The worst case value and translates these to internal clock cycles for PIC operating at any one of the three crystal frequencies.
- These are the values needed when code is written to generate the I^2C waveforms.

Embedded C program to transmit data using I^2C bus:

MASTER FUNCTIONS

Initialize I^2C Module as Master

```
void I2C_Master_Init(const unsigned long c)
{
  SSPCON = 0b00101000;          //SSP Module as Master
  SSPCON2 = 0;
  SSPADD = (_XTAL_FREQ/(4*c))-1; //Setting Clock Speed
  SSPSTAT = 0;
  TRISC3 = 1;                   //Setting as input as given in datasheet
  TRISC4 = 1;                   //Setting as input as given in datasheet
}
```

For Waiting

```
void I2C_Master_Wait()
{
  while ((SSPSTAT & 0x04) || (SSPCON2 & 0x1F)); //Transmit is in progress
}
```

Start Condition

```
void I2C_Master_Start()
{
  I2C_Master_Wait();
  SEN = 1;            //Initiate start condition
}
```

Repeated Start

```
void I2C_Master_RepeatedStart()
{
  I2C_Master_Wait();
  RSEN = 1;           //Initiate repeated start condition
}
```

Stop Condition

```
void I2C_Master_Stop()
{
  I2C_Master_Wait();
  PEN = 1;           //Initiate stop condition
}
```

Write Data

```
void I2C_Master_Write(unsigned d)
{
  I2C_Master_Wait();
  SSPBUF = d;         //Write data to SSPBUF
}
```

Read Data

```
unsigned short I2C_Master_Read(unsigned short a)
{
  unsigned short temp;
  I2C_Master_Wait();
  RCEN = 1;
  I2C_Master_Wait();
  temp = SSPBUF;      //Read data from SSPBUF
  I2C_Master_Wait();
  ACKDT = (a)?0:1;    //Acknowledge bit
  ACKEN = 1;          //Acknowledge sequence
  return temp;
}
```

SLAVE FUNCTIONS

Initialize module as slave

```
void I2C_Slave_Init(short address)
{
  SSPSTAT = 0x80;
  SSPADD = address; //Setting address
  SSPCON = 0x36;    //As a slave device
  SSPCON2 = 0x01;
  TRISC3 = 1;       //Setting as input as given in datasheet
  TRISC4 = 1;       //Setting as input as given in datasheet
```

```
  GIE = 1;         //Global interrupt enable
  PEIE = 1;        //Peripheral interrupt enable
  SSPIF = 0;       //Clear interrupt flag
  SSPIE = 1;       //Synchronous serial port interrupt enable
}
```

On Receive interrupt

```
void interrupt I2C_Slave_Read()
{
 if(SSPIF == 1)
 {
  SSPCONbits.CKP = 0;

  if ((SSPCONbits.SSPOV) || (SSPCONbits.WCOL)) //If overflow or collision
  {
   z = SSPBUF; // Read the previous value to clear the buffer
   SSPCONbits.SSPOV = 0; // Clear the overflow flag
   SSPCONbits.WCOL = 0; // Clear the collision bit
   SSPCONbits.CKP = 1;
  }
 }

 if(!SSPSTATbits.D_nA && !SSPSTATbits.R_nW) //If last byte was Address + Write
 {
  z = SSPBUF;
  while(!BF);
  PORTD = SSPBUF;
  SSPCONbits.CKP = 1;
 }
 else if(!SSPSTATbits.D_nA && SSPSTATbits.R_nW) //If last byte was Address + Read
 {
  z = SSPBUF;
  BF = 0;
  SSPBUF = PORTB ;
  SSPCONbits.CKP = 1;
  while(SSPSTATbits.BF);
 }

 SSPIF = 0;
 }
}
```

2. **Explain briefly the concept of I^2C subroutines. Illustrate with suitable example how I^2 communication is carried out in PIC microcontroller.** *(Apr'17)*
   - Because the SCL pin must have an open drain output while the SDA pin must be either an input or have an open drain output, the I^2C bus subroutines will repeatedly access TRISC, the data direction register for PORTC.
   - However TRISC is located at the bank 1 address, H'87', which cannot be accessed by direct addressing without first executing without first executing the instruction.

```
        baf     STATUS, RPO
```

- Then changing the required bit of TRISC and finally reverting back to bank 0 with

```
bcf     STATUS, RPO
```

- Instead of doing this load the indirect pointer, FSR with the address of TRISC and then do the required bit setting and bit clearing of TRISC bits indirectly.
- For example ,with the following definitions

```
SCL     equ 3
```

And

```
SDA     equ 4
```

Then

```
bsf     INDF, SDA
```

- Will release the SDA line, letting the external pullup resistor of fig, b pull it high or some other I^2C chip pull it low.
- This use of FSR raises two conditions:
  - If these I^2C subroutines are executed from the mainline program, then any interrupt service routine that also uses FSR must set it aside upon entry and restore it upon exit.
  - Any use of indirect addressing to access a sequence of addresses in the PIC's RAM when used in conjunction with these I^2C subroutines must swap pointers in and out of FSR.

Delay macro:

- The timing requirement of fig. will be handled by inserting a number of nop instructions between the instruction that changes SDA and SCL.
- The number of nop instructions required depends on the crystal clock rate.
- The delay macro defines in fig.a. Uses the equate of frequency to 4, 10 or 20 to insert a number of nop instruction equal to first, second or third macro parameter.
- The equates and variable needed for the I^2C subroutine are listed in fig.8. DEVADD is the selected peripheral chip's 7-bit address on the I^2C bus shifted left one place to align it for use as a control byte.
- INTADD is a selected register or memory address inside the selected peripheral chip.
- DATAOUT is used to hold the byte of data to be sent to the selected register in the selected peripheral chip by anI^2C output subroutine, I^2Cin, from selected register in the selected peripheral chip.
- The I^2Cout subroutine of fig. calls a start subroutine to generate the START condition, and calls a TX subroutine three times to send DEVADD, INTADD and DATAOUT out on the I^2C bus.
- Finally it calls a STOP subroutine to generate STOP condition.
- The TX subroutine takes the byte passed to it in W, uses a TXBUF variable to extract the bits one by one and transmits each bit using a Bit out subroutine.
- TX reads the acknowledge bit by calling a BitIn subroutine, setting Z if ACK occurs.

```
Noexpand
Delay   macro   freq4, freq10, freq20
        If      freq==4
                Fill    (nop), freq4
        endif
        If      freq==10
```

```
        Fill    (nop), freq10
    Endif
    If      freq==20
        Fill    (nop), freq20
    endif
    endm
```

a) macro definition
   delay 0,1,2

b) Example of macro invocation which will insert:
   0 nop for OSC = 4MHZ (i.e., for freq equ 4),
   1 nop for OSC = 10MHZ (i.e., for freq equ 10),
   2 nops for OSC = 20MHZ (i.e. for freq equ 20),

Delay macro

```
Freq  equ   4  ;set to 4,10,or 20 for 4MHZ,10MHZ,or 20MHZ
SDA   equ   4  ; I2C serial data bit for PORTC
SCL   equ   3  ; I2C serial clock bit for PORTC
```

Equates

```
Cblock
    .
    .
    DEVADD      ; Device's I2C address x2
    INTADD      ; Internal address
    DATAOUT     ; Data to be written into INTADD during a write
    DATAIN      ; Data to be read into INTADD during a read
    TXBUFF      ; Buffer for each byte sent by TX
    RXBUFF      ; Buffer for each byte received by RX
    .
    .
endc
```

Variables

- The I^2Cin subroutine of fig is similar to the I^2Cout subroutine.
- It calls the START subroutine and TX subroutine twice to send DEVADD (plus R/W=0) and INTADD.
- Then it calls the START subroutine to send DECADD (plus R/W=1), the RX subroutine to read back a byte (with NOACK) and finally the STOP subroutine.

I^2C subroutine:

; The I^2Cout subroutine transfer out three bytes: DEVADD, INTADD and DAATAOUT.

I^2Cout

```
    Call    Start       ; Generate START condition
    Movf    DEVADD,W    ; Send peripheral address with R/W=0(write)
    Call    TX
    Movf    INTADD,W    ; Send peripheral 's internal address
```

```
        Call    TX
        Movf    DATAOUT,W ; Send data to write to peripheral
        Call    TX
        Call    Stop        ; Generate STOP condition
        return
```

; The I^2Cin subroutine transfer out DEVADD (with R/W=0) and INTADD, restarts; transfer out DEVADD (with R/W=1) and reads one byte back into DATAIN

```
I2Cin
        Call    start       ; Generate START condition
        Movf    DEVADD,W    ; Send peripheral address with R/W=0(write)
        Call    TX
        Movf    INTADD,W    ; Send peripheral'
        Call    TX
        Call    Restart     ; Restart
        Movf    DEVADD,W    ; Send peripheral address
        Iorlw   B'00000001' ; with R/W=2(read)
        Call    TX
        bsf     TXBUFF,7    ; NOACK the following read of the one byte
        call    RX          ; Read byte
        movwf   DATAIN      ; into DATAIN

        call    Stop        ; Generate STOP condition
        return
```

; The Start subroutine initializes the I^2C bus and then generates the START condition on the I^2C bus

; The Restart entry point bypass the initialization of the I^2C bus

```
Start
        Movlw       B'00111011  ;Enable I2C master mode
        Movwf       SSPCON
        bcf         PORTC,SDA   ;Drive SDA low when it is an output
        bcf         PORTC,SCL   ;Drive SCL low when it is an output
        movlw       TRISC
        movwf       FSR
Restart
        bsf         INDF,SDA    ;make sure SDA is high
        bsf         INDF,SCL    ; make sure SCL is high
        delay       0,1,2       ;t:START
        bcf         INDF,SDA
        delay       0,1,2
        bcf         INDF,SCL
        return
```

; The Stop subroutine generates the STOP condition on the I^2C bus

```
Stop
        bcf         INDF,SDA    ;return SDA low
        bsf         INDF,SCL    ;Drive SCL high
        delay       0,1,2       ;t:STOP
        bsf         INDF,SDA    ;and then drive SDA high
```

```
        return
```

I²C subroutine

```
; The TX subroutine sends out the byte passed to it in W
; It returns with z=1 if ACK occurs
; It returns with z=0 if NOACK occurs
  TX
        movwf   TXBUFF      ;save parameter in TXBUFF
        bsf     STATUS,C    ;Rotate a one through TXBUFF to count bits
  TX_1
        rlf     TXBUFF,F    ;Rotate TXBUFF left,through carry
        movf    TXBUFF,F    ;Set Z bit when all eight bits have been transferred
        btfss   STATUS,Z    ;Until Z=1
        call    BitOut      ;Send Carry bit,then clear carry bit
        btfss   STATUS,Z    ;
        goto    TX_1        ;then do it again
        call    BitIn       ;Read acknowledge bit into bit 0 of RXBUFF
        movlw   B'0000001'  ;Check ackknowlege bit
        andwf   RXBUFF,W    ;Z=1 if ACK;Z=0 if NOACK
        return
```

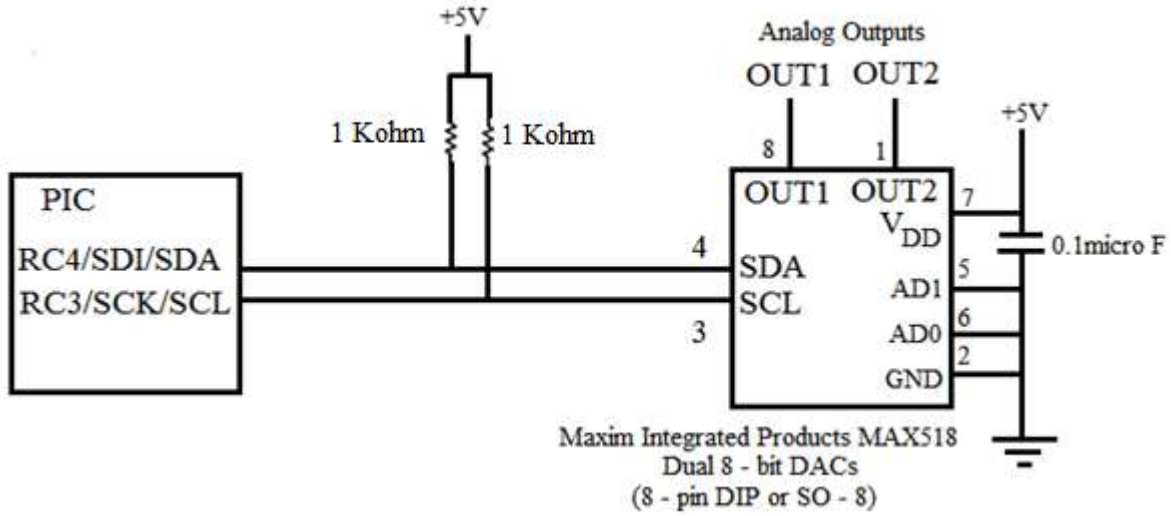
```
; The RX subroutine receives a byte from the I2C bus into w,using RXBUFF buffer
; Call RX with bit 7 of TXBUFF clear for ACK
; Call RX with bit 7 of TXBUFF set for NOACK
  RX
       movlw    B'00000001   ; Rotate a one through RXBUFF to the carry bit to count
                               bits
        movwf   RXBUFF
  RX_1
        rlfss   RXBUFF,F    ;Shift previous bits left

        call    BitIn       ;Read a bit from SDA into bit 0 of RXBUFF
        btfss   STATUS,C    ;C=1 yet;
        goto    RX_1        ;No,do it again
        rlf     TXBUFF,F    ;move bit 7 of TXBUFF to carry bit
        call    BitOut      ;and from there to SDA as acknowledgement
        movf    RXBUFF,W    ;Put received byte into W
        return
```

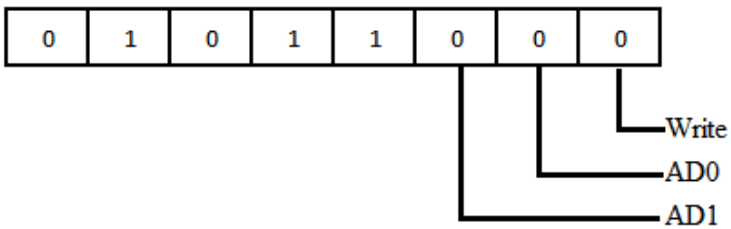
```
; The Bitout subroutine transmits, then clears, the carry bit

  BitOut
        bcf     INDF,SDA        ;copy carry bit to SDA
        btfsc   STATUS,C
        bsf     INDF,SDA
        bsf     INDF,SCL        ;pulse clock line
        delay   0,1,2           ;t: HIGH
        bcf     INDF,SCL
        bcf     STATUS,C        ;clear carry bit
        return
```

```
; The BitIn subroutine receives one bit into bit 0 of RXBUFF
  BitIn
        bsf     INDF,SDA        ;Release SDA line
        bsf     INDF,SCL        ;Drive Clock line high
        bcf     RXBUFF,0        ;copy SDA to bit 0 of RXBUFF
        btfsc   PORTC,SDA
        bsf     RXBUFF,0
        bcf     INDF,SCL        ;Drive clock line low again
  return

  END of I2C subroutine
```

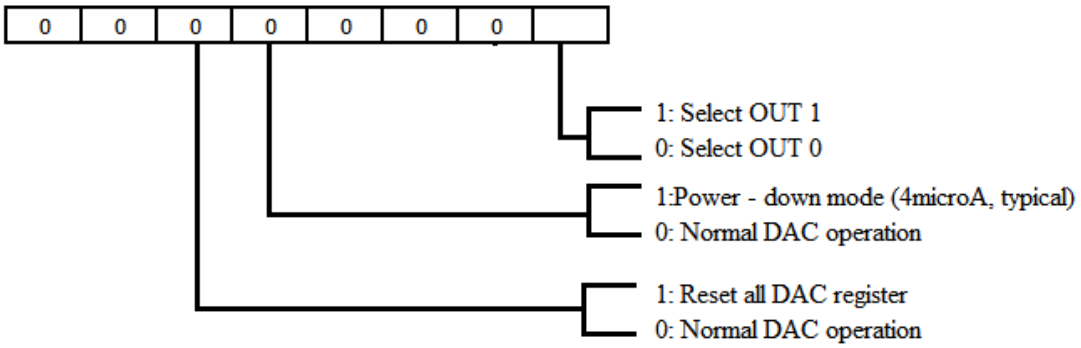
3. Explain the operation of DAC OUTPUT with neat diagram.

- Two digital to analog converter output are easily added to a PIC with the MAX518 eight – pin DIP or SO -8 surface –mount part shown in fig 10.





First byte of message String



Second Byte

$$\text{Analog output voltage} = V_{DD} \times \frac{B}{256}$$

- Each output channel produces an output voltage that ranges from 0V upto 255/256th of the power supply voltage, giving roughly 20-mV output increments.
- An output of 2.50V will be appear on the OUT0 pin if the following three bytes are sent to the chip:
  B'01011000' B'00000000' B'10000000'
- An output of 1.25V will appear on the OUT1 pin following
  B'01011000' B'00000001' B'01000000'
- The MAX518 chip includes a power –on reset circuit that drives the two outputs to 0v initially.
- Because the MAX518 may come out of reset after the PIC chip comes out of reset, the MAX518 may ignore commands sent to it immediately after the PIC comes out of reset.

- The two address input AD1 and AD0, provide an adjustable part of the chip's I^2C address. With 5 bts fixed at 01011 and two adjustable bits, it is possible to connect four MAX518 chips to a PIC.
- Each chip must have its AD1 and AD0 pins tied to a different combination of +5v and GND.
- The four 7-bit addresses become **B'0101100', B'0101101', B'0101110',and B'0101111'.**

4. With neat sketches, show the PIC interfacing with peripherals that includes ADC's with sensors. [OR] **Explain the operation of ADC interfacing with PIC microcontroller.** *(Nov'16, Apr'17, Nov'17)*

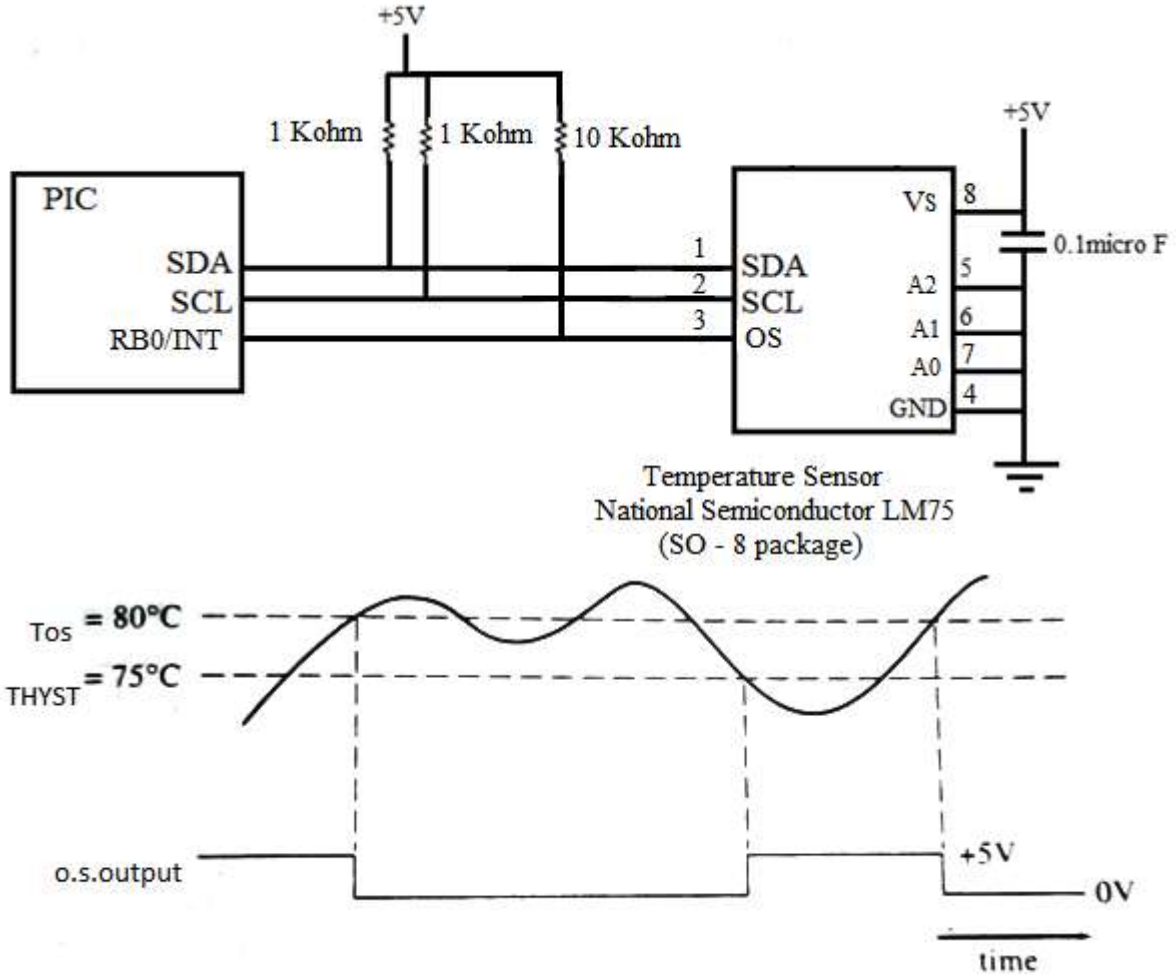
- The combination of an analog temperature transducer, an analog-to-digital converter and an I^2C bus interface all in a tiny SO-8 surface mount package represents a significant contribution to designers.
- The analog voltage from the internal temperature transducer passes to a converter located in a close physical proximity that all of the potential problems of noise and ground voltage offsets are handled inside the chip, once and for all.
- National semiconductor 's LM75 chip converter temperatures over the range of -25° to +100°C with ±2°C accuracy.
- The same part delivers ±3°C accuracy for temperature down to -55°C and upto +125°.
- For many application, an even more important features is its fine 0.5°C resolution, obtained with the support of a 9 bit ADC.

Temperature	Digital Output	
	Binary	Decimal
+125ºC	0 1111 1010	250
+25ºC	0 0011 0010	50
+0.5ºC	0 0000 0001	1
0 ºC	0 0000 0000	0
-0.5 ºC	1 1111 1111	512 – 1 = 511
-25 ºC	1 1100 1110	512 – 50 = 462
-55 ºC	1 1001 0010	512 – 110 = 402

LM75 output coding of temperature

- Fig Illustrates the two's complement form of the output.
- This 0.5°C resolution means that small temperature difference are measured within 0.5°C.

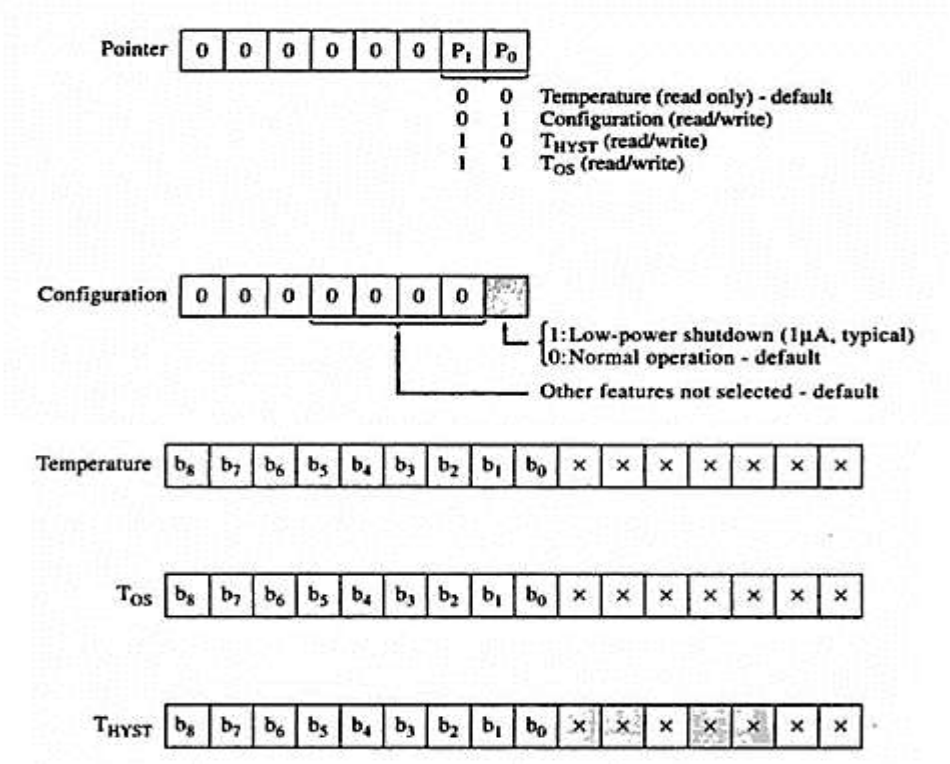
- The LM75 chip also includes a thermal watchdog that can be set upto interrupt the PIC on its RB0/INT edge –triggered interrupt input when the temperature rises past a programmable set point, T_{os}(where OS stands for one temperature shutdown).
- It includes programmable hysteresis so that the temperature must dip down below the set point's T_{os} threshold to a lower T_{HYST} Threshold before rising again past the T_{os} set point to generate another output edge.



Default performance of O.S (over temperature shutdown) output

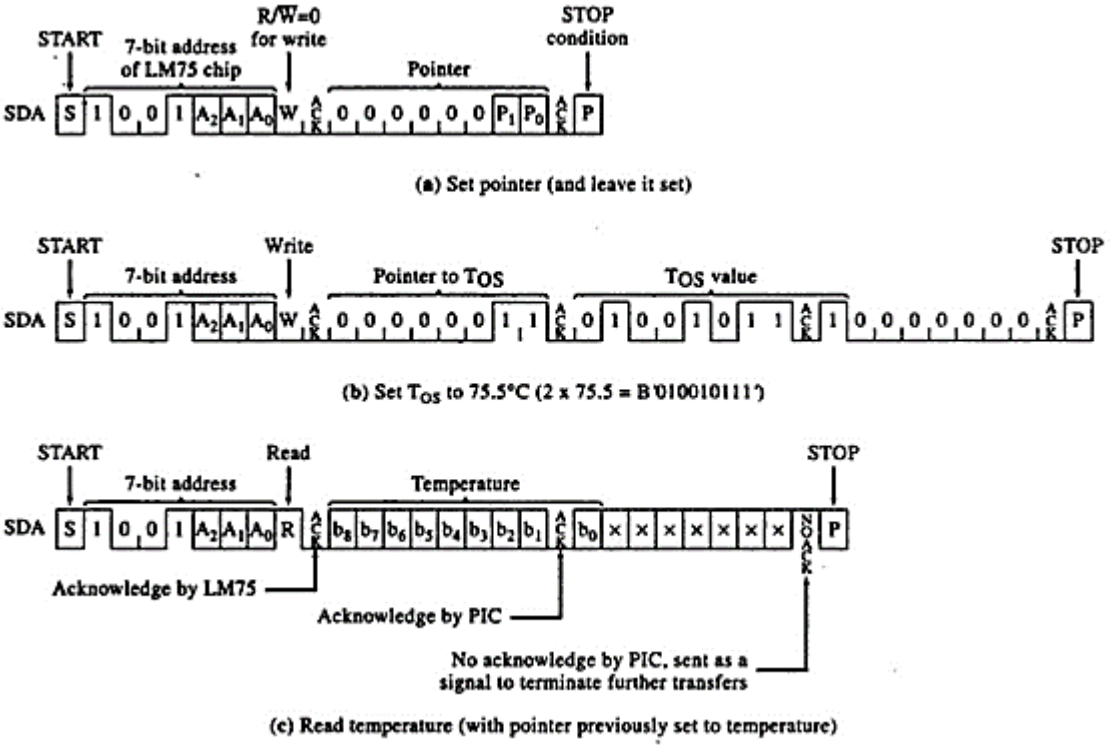
LM75 temperature inputs

- The chip includes a power –on reset circuit that default to the operation shown in fig. b.
- At power-on time the PIC may come out of reset first; therefore it is necessary to insert a delay before initializing the LM75's thermal watchdog circuitry.
- Otherwise the PIC's commands to the LM75 may go unnoticed.



LM75 internal register

- The register structure of the LM75 is shown in fig when a "write" message string is transmitted to the chip, the first byte selects the chip for a write and the second byte loads the pointer register.
- The write message string can stop there (as in fig a) or it can continue with a 2-byte write of 100° F=75.5° C to the T_{os} register (as in fig.b).

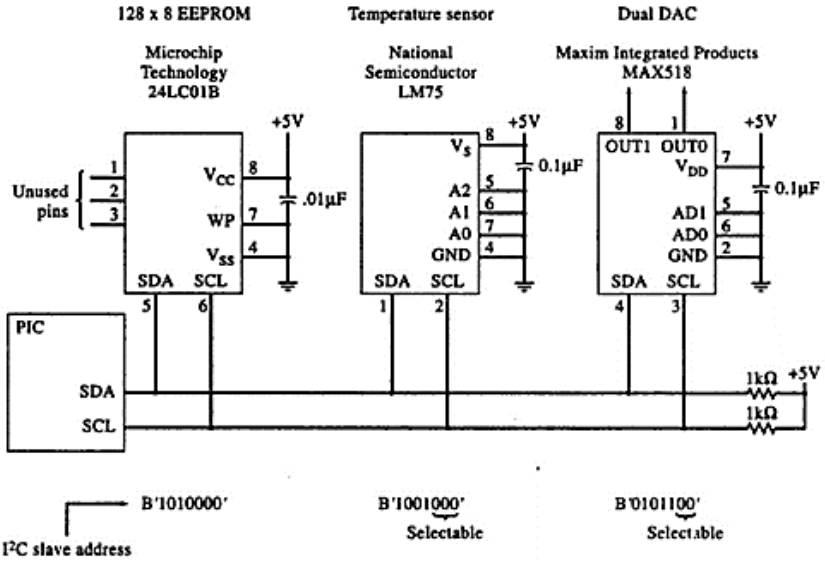


LM75 message string

- Once the pointer has been set, any of these register can be read, reading 2 byte for temperature T_{os} or T_{HYST} (as in fig. c) or reading just 1 byte for the configuration register.
- If the thermal watchdog function of the LM75 is not used, the advantage can be taken of the power –on default clearing of the pointer and configuration register.
- In this case the interactions with the chip need to be no more than successive reads of the temperature, as in fig. c

5. Discuss detail about SERIAL EEPROM:

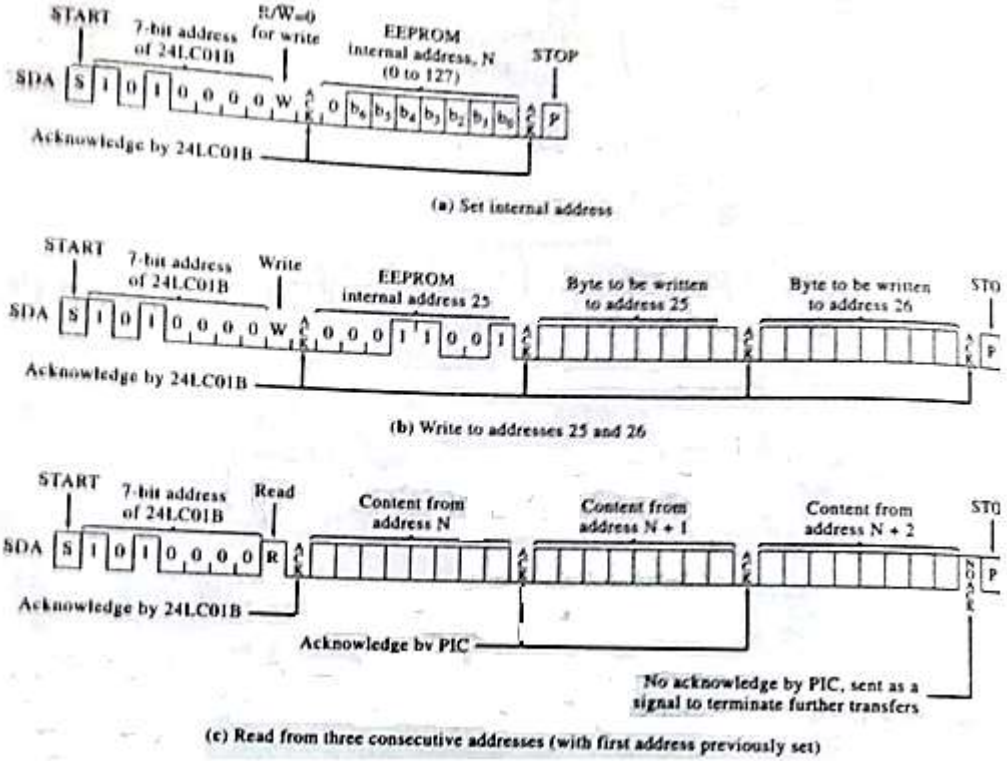
- EEPROM (Electrically Erasable, Programmable Read-Only Memory) technology supplies nonvolatile storage of variables to a PIC Controlled device or instrument.
- That is variables, stored in an EEPROM will remain there even after power has been turned off and then on again.
- Some instruments use an EEPROM to store calibration data during manufacture.
- In this way, each instruments is actually custom built, with customization that can be easily automated.
- Other instruments use an EEPROM to allow a user several sets of setup information.
- For an instrument requiring a complicated setup procedure, this permits a user to retrieve the setup required for any one of several very different measurements.
- Still other devices use an EEPROM in a way that is transparent to a user, providing backup of setup parameters and thereby bridging over power outages.
- An EEPROM with Controlled an I^2C serial interface such as microchip technology's 24LC01B provides designers with a convenient solution to a need for nonvolatile data storage.
- It holds 128 bytes of data.
- It is packaged in a tiny eight –pin DIP or surface-mount package.
- It requires only a +5V supply and will operate on the "fast-mode" (400 kbit/s) I^2C bus.
- It draw less than 3mA of supply current during programming, 1mA during reading, and 0.1mA during standby.
- It times its own write cycle and automatically erases a byte before writing into it.
- A block of up to 8 byte can be written to the chip at one time and the chip will program then all simultaneously in less than 10ms (our mainline looptime).
- The manufacturer guarantees successful writes for up to 10,000,000 erase/write cycles and data retention beyond 200 years over an operating temperature range of 0°C to +70°C (or -40°C to +85°C for industrial version).



I^2C bus connection to EEPROM chip and to other device.

- The device with its interface circuit is illustrated in fig. the WP (write protect) pin permits a manufacturer to program a part with calibration constant (with WP low) and then to permit only reads thereafter (with WP tied high).
- The I^2C devices in this part has the single, fixed 7-bit address.

  1010xxx
- That is, any read from or write to the slave address B'1010000' or B'1010001' or ….or B'1010111' will access the EEPROM chip.
- The EEPROM makes use of an internal address pointer that is set during the second byte of a "write" message string, as in fig.



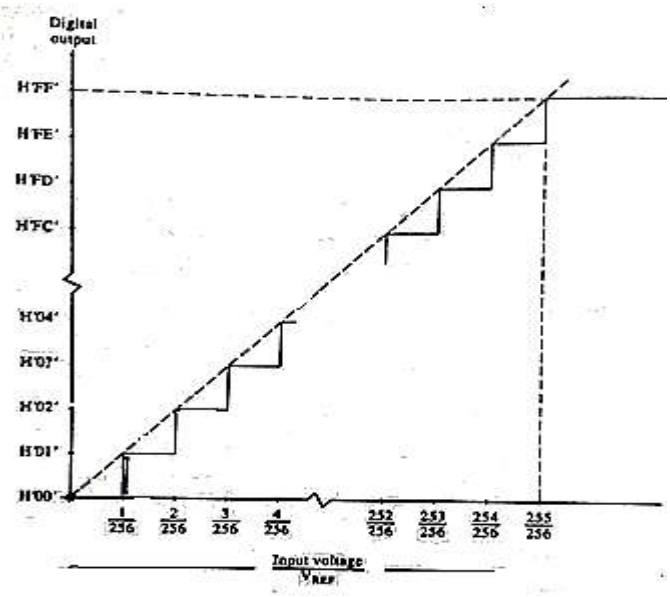
24LCO1B EEPROM message string

- If further bytes are transmitted before the STOP condition, as in fig b. they will be accepted as the data to be written into the selected internal address.
- The reception of the STOP condition triggers the programming of these bytes into the selected addresses.
- While the EEPROM is doing its autonomous programming operation, it will not acknowledge another write command.
- Because of this, the acknowledge bit can be used as a flag to determine when the programming operation has been completed.
- Simply send out the slave address with the write bit low and check whether the ACK bit is pulled low by the EEPROM.
- Until it does get pulled low in acknowledgement, the START condition followed by the same byte can be sent repeatedly and the ACK bit tested.
- With a typical programming with a typical programming time of 2ms, programming of many bytes can take place as rapidly as possible, faster than simply allowing the 10ms worst case write time to expire.
- This EEPROM includes a page-write buffer for writing upto 8 bytes simultaneously with the single write message string shown in fig.b.
- Within 10ms after the STOP condition is received by the EEPROM, all of the transmitted bytes will be programmed.

- However, all eight addresses are constrained to have the same upper 5 bits.
- That is, only the lower 3 bits of the EEPROM's internal address counter are incremented when more than one data byte is included in a write command sequence.
- For example, if the EEPROM address sent in the second byte of the write command is B'00010110' and example, if the EEPROM address sent in the second byte of the write command is B'00010110'and if that address byte is followed by three data bytes and the STOP condition,
- Then the first of the three data bytes will be written into address B'00010110',the second into B'00010111', and the third into B'00010000'(and not into B'00011000',as intended).
- Reading any number of bytes of data from selected EEPROM addresses requires that a starting address first sent to the EEPROM with the write message string of fig. a
- This string is followed by the message string of fig. c, consisting of the START condition, a read command and then a read of data from consecutive addresses sent back by the EEPROM.
- The PIC signals the EEPROM to send no further bytes by not pulling the SDA line low during the last acknowledge bit time.
- The sending of the STOP condition by the PIC completes the message string.

6. Draw and explain the architecture of on chip ADC of PIC microcontroller and write a suitable assembly language program for configuring the ADC. *[Apr'18]*

- The PIC analog to digital converter has the idealized transfer function shown in fig.1 .It converts an input voltage to an 8 bit number.
- The input voltage is scaled against a reference voltage, $V_{REF,}$ producing the 8 bit output shown.



Idealized PIC ADC transfer function

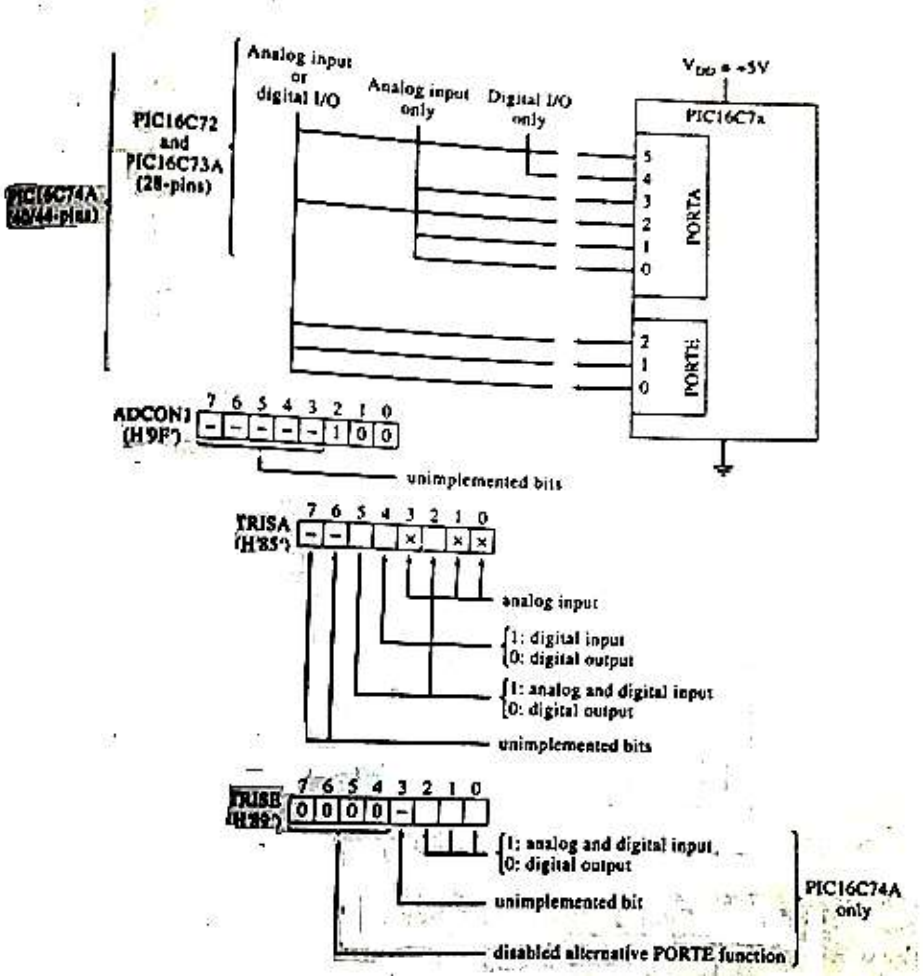
- The reference voltage that is best suited for many applications is the PIC supply voltage, V_{DD}, selected as one option and connected to the ADC internally, thereby avoiding the need to dedicate a pin to this role.
- For a transducer whose output is proportional to its own supply voltage and that uses the PIC supply voltage as its own supply voltage, making V_{REF} equal to V_{DD} is an ideal choice.
- For other application ,using V_{DD} as the reference voltage offers the largest possible analog voltage input range, since proper ADC operation requires

$$0V \leq V_{INPUT} \leq V_{REF}$$

- And also

$$3V \leq V_{REF} \leq V_{DD}$$

- Given this choice of $V_{Ref} = V_{DD}$, the PIC parts can assign upto eight pins to serve as analog inputs to the ADC, using the pins and the register initializations shown in fig.



ADC input with $V_{Ref} = V_{DD}$

- Any pin that is assigned to serve as an "analog and digital input" to the ADC can be read as a digital input by reading the appropriate port pin or used as an analog input by selecting it as the input channel to the ADC.
- The default power –on state of ADCON1 (H'00') powers down the digital I/O circuitry for the five pins labeled "Analog input or digital I/O", thereby making them "Analog input only" pins.
- For some application, the use of an external voltage reference of 3.0V provide the greatest possible resolution in the output.
- This is particularly useful for voltage difference measurement.

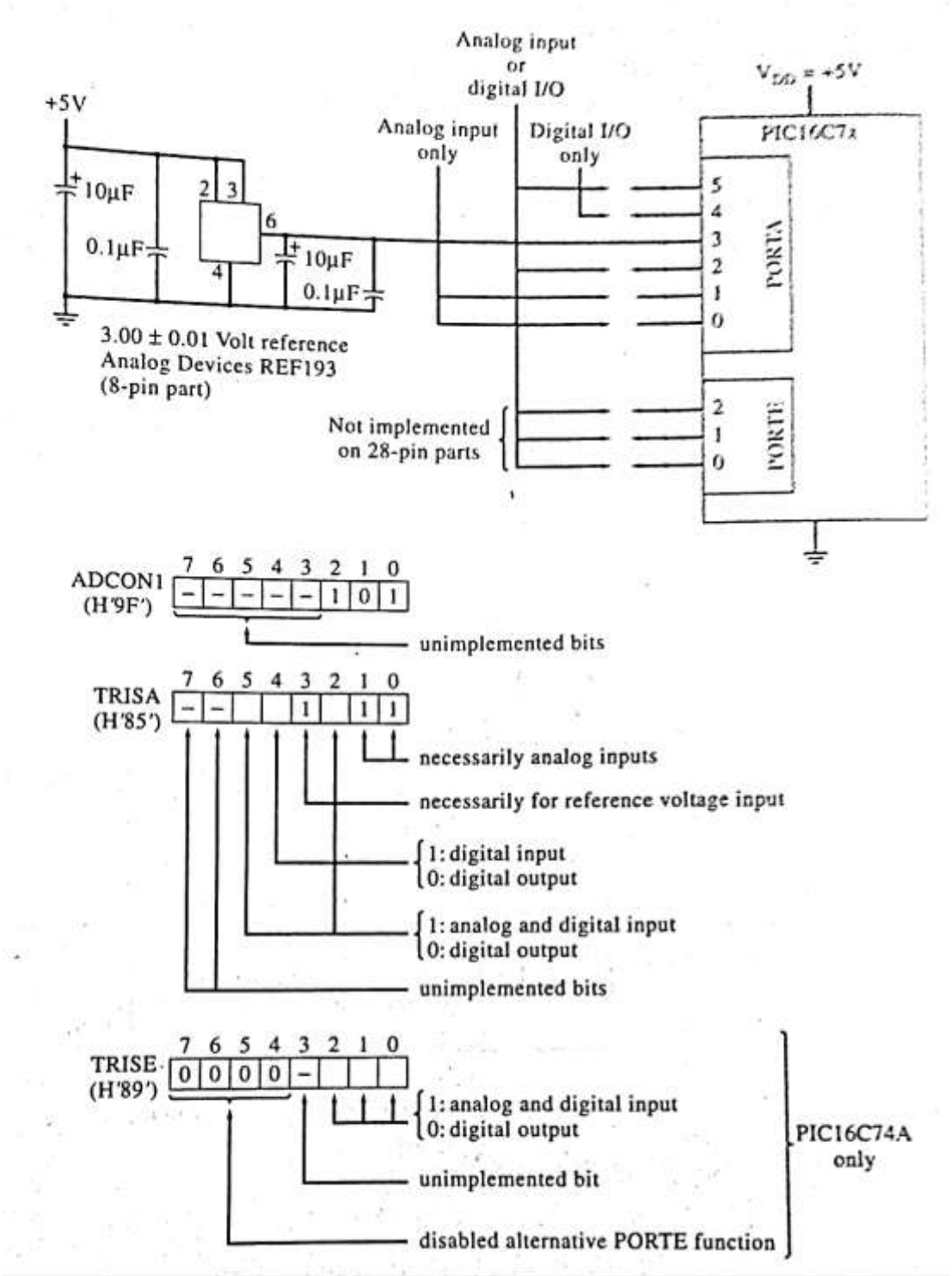


Fig. ADC input with external V_{REF}

- Fig. illustrates the connection of an external voltage reference to the PIC.
- The 10µF Capacitors on the input and output of the voltage reference part suppress the effects of power supply ripple.
- The 0.1µF capacitors suppress RF switching transients.

Voltage reference	Internal: $V_{REF} = V_{DD}$(the PIC power supply voltage) External :$3.0V \leq V_{REF} \leq V_{DD}+0.3V$
Error from idealized characteristic of fig.1	Internal voltage reference: Error < ± $V_{DD}/256$ for V_{DD} = 5V±10% External voltage reference: Error < ± $V_{REF}/256$ for $V_{REF} \geq 3.0V$
Power supply current drawn by ADC	180µA typical
Minimum sample time	12µs for R_{SOURCE}<10KΩ. This is time after an input channel has been selected and before a conversion is initiated
Conversion time	15µs for OSC=20MHz 30µs for OSC=10MHz 19µs for OSC=4MHz
Automatic sample rate control for conversion of analog input from one channel	Use CCP2's"special event mode" to set period
Interrupt when conversion is complete	Use PIE1 register's ADIE bit to enable "ADC conversion complete" interrupts Use PIR1 register's ADIF interrupt flag

ADC performance characteristics

- The performance characteristics of the analog – to - digital converter are listed in fig.
- In carrying out conversion, it is important to allow for the sample time listed in fig. and expanded on in fig.
- When a new input channel is selected, the analog multiplexer's sampling switch connects the input pin to C_{HOLD}, must be allowed time to charge until it equals V_{SOURCE} to within one –half of one of the voltage steps of Fig.1.
- The larger the resistance of the source being measured, the longer this charging time will become. The relationship is shown in fig.5b.A high source resistance can be converted to a low source resistance with the help of the op amp "follower" circuit of fig. c.
- After waiting out the sample time, a conversion can be initiated.
- The ADC circuit will open the sampling switch and carry out the conversion of the input voltage as it was at the moment the switch was opened.
- Upon completion of the conversion, the sampling switch is closed and V_{HOLD} again tracks V_{SOURCE}.
- If the ADC is used to sample a single channel at equally spaced intervals, this can be done automatically under interrupt control. The timer's CCP (capture/compare/PWM) module can be used with Timer1 to initiate periodic ADC conversions of the selected channel.
- In addition, the ADC is setup to generate an interrupt when the conversion has been completed.
- After this process has been set up begun, the CPU simply deals with each sample as it becomes available.

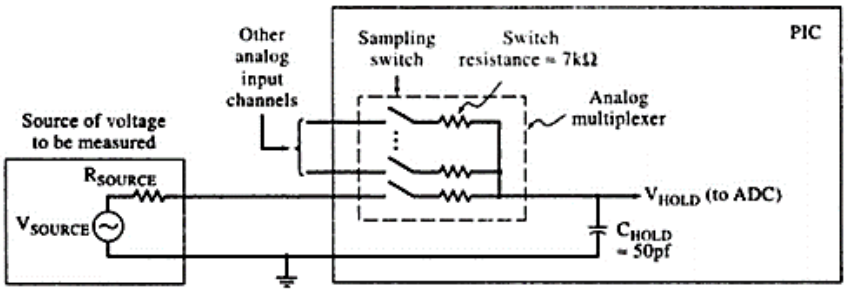
Assembly language program for configuring the ADC:

A/D conversion with Interrupt:

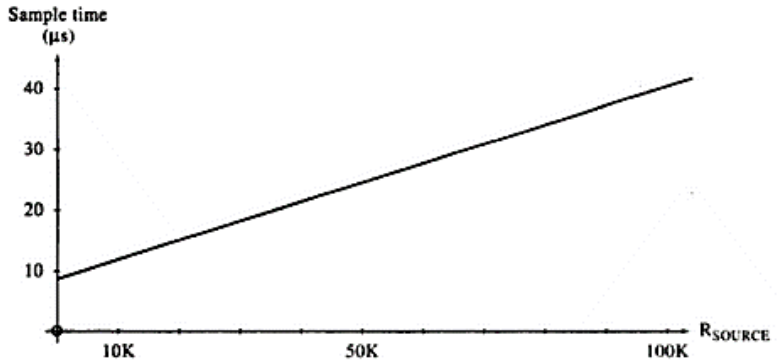
```
bsf STATUS, RP0          ; Select Bank 1
clrf ADCON 1      ; Configure A/D input
bsf PIE1, ADIE    ; Enable A/D interrupt
bcf STATUS, RP0; Select Bank 0
movlw 0811+       ; Select fosc/32, channel 0, A/D on movwf ADCON0
bcf PIR1, ADIF
bsf INTCON, PEIE
```

```
bsf INTCON, GIE
                ; Ensure that the required sampling time for the selected input channel
                  has elapsed
                ; Then the conversion may be started.
bsf ADCON0, GO  ; Start A/D conversion
                ; AD/F bit will be set and GO/DONE bit is cleared upon completion
                  of A/D conversion.
```

7. Explain briefly ADC USE:

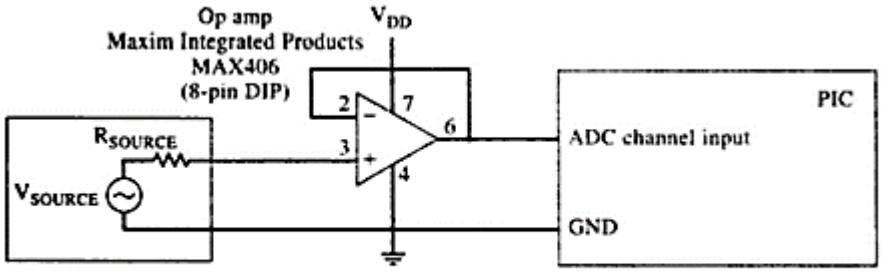


(a) Equivalent circuit



Sample time consideration

- Register ADCON1, TRISA and TRISE must be initialized to select the reference voltage and the input channel desired, as described by fig.2 and fig.3.
- Then ADCONO is initialized with the steps listed in fig.6.The first step selects the ADC clock source from among four choices (OSC/2, OSC/8, OSC/32 and RC).
- The choices shown in fig.6a provide the highest rate consistent with the constraint the ADC clock period must be 1.6 µs or greater .The "RC" choice is designed for use with a PIC being clocked by a relatively slow clock .
- It lets the ADC run at a nominal 250-kHz rate.



(c) Reduction of source impedance seen by ADC input

Sample time considerations.

- If just one analog input is to be used, it can be selected once and for all by combining the channel selection in fig.6b with the ADC clock period selection of fig. a.
- If several analog channels are to be used, it is important to remember to wait for the sample time in fig.

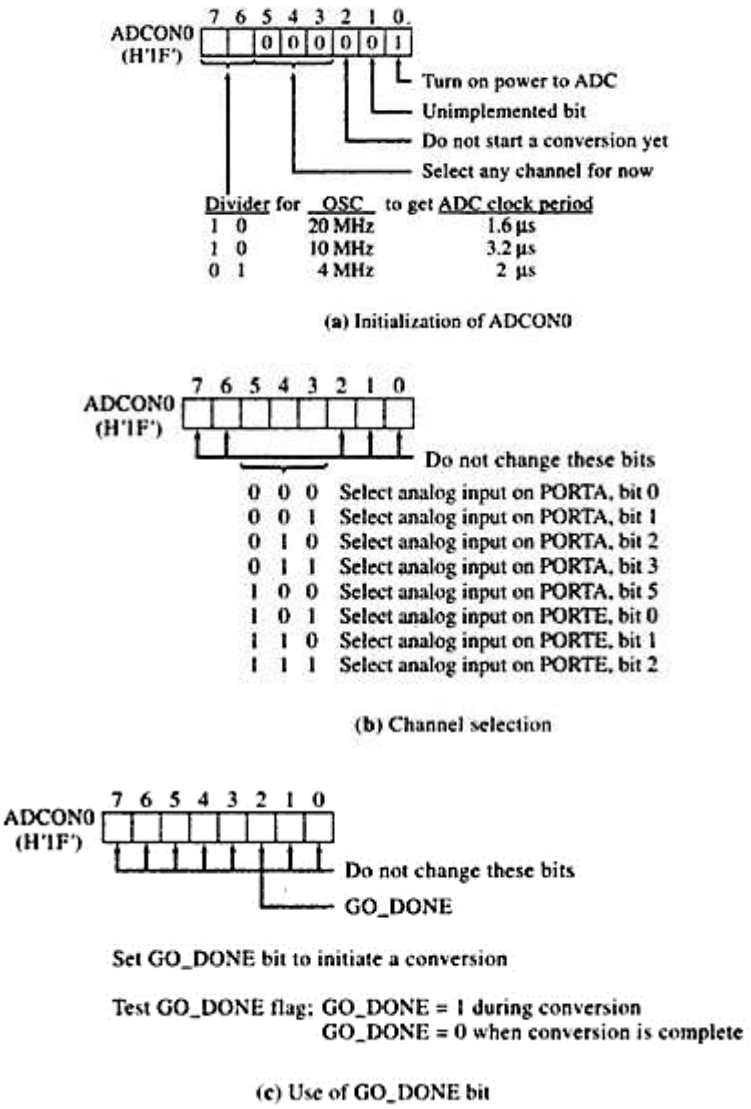
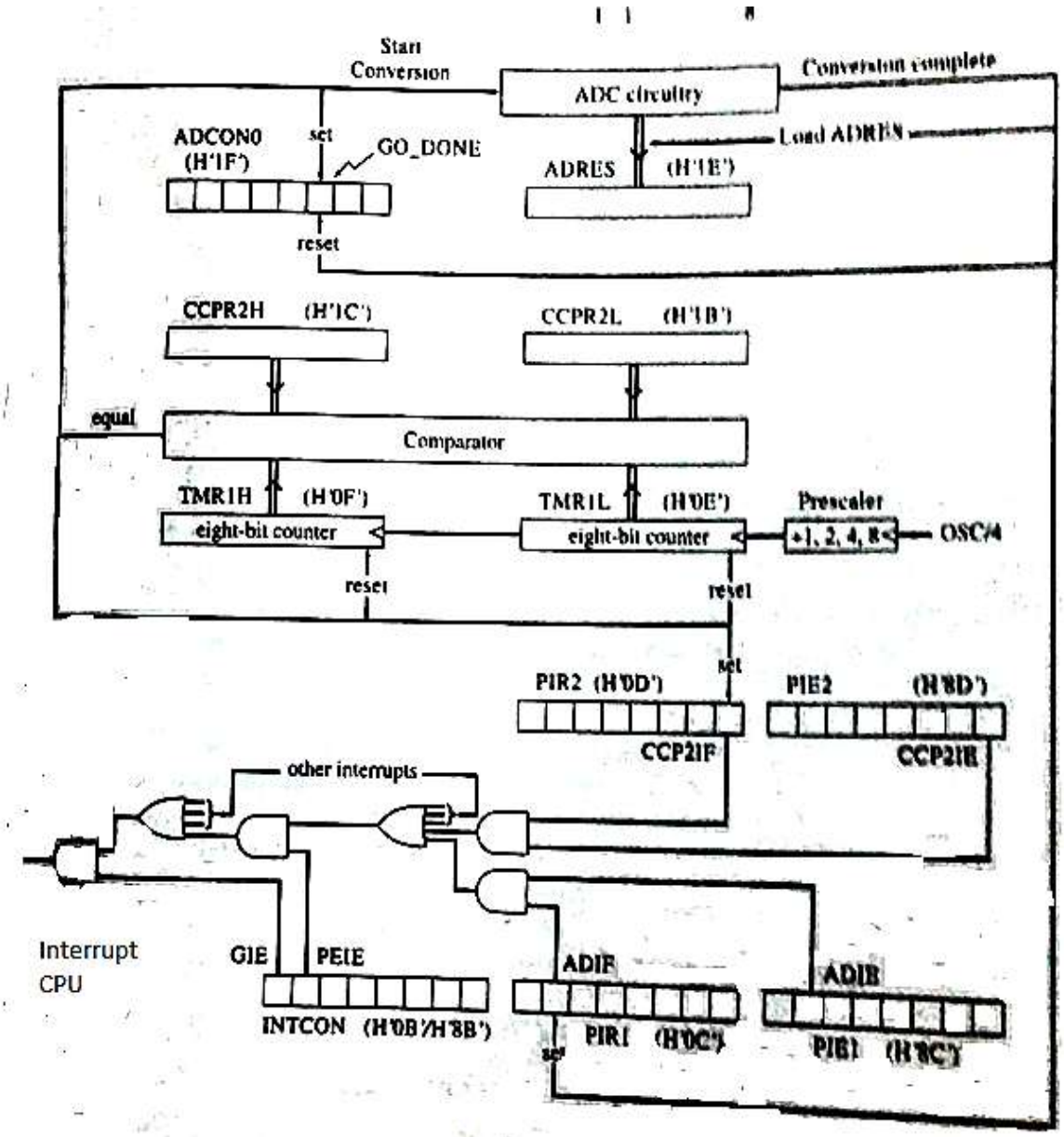


Fig.6.Setup and use of the ADCONO register.

- That is select the channel ,wait out the required sample time and then initiate the conversion by setting the GO_DONE bit of ADCONO .When GO_DONE=0 again, the conversion is complete; read the result from ADRES(H'1E), the ADC result register.

```
CALL    COMMAND   ; issue command
CALL    READY     ; Is LCD ready?
MOVLW   Ox86      ; cursor: line1, pos.6
CALL    COMMAND   ; command subroutine
CALL    READY     ; Is LCD ready?
MOVLW   A'N       ; displayletter 'N'
CALL    DATADISPLAY
CALL    READY     ; Is LCD ready?
MOVLW   A'O       ; displayletter 'O'
CALL    DATADISPLAY
HERE    BRA
```

```
HERE                    ; STAYHERE
```



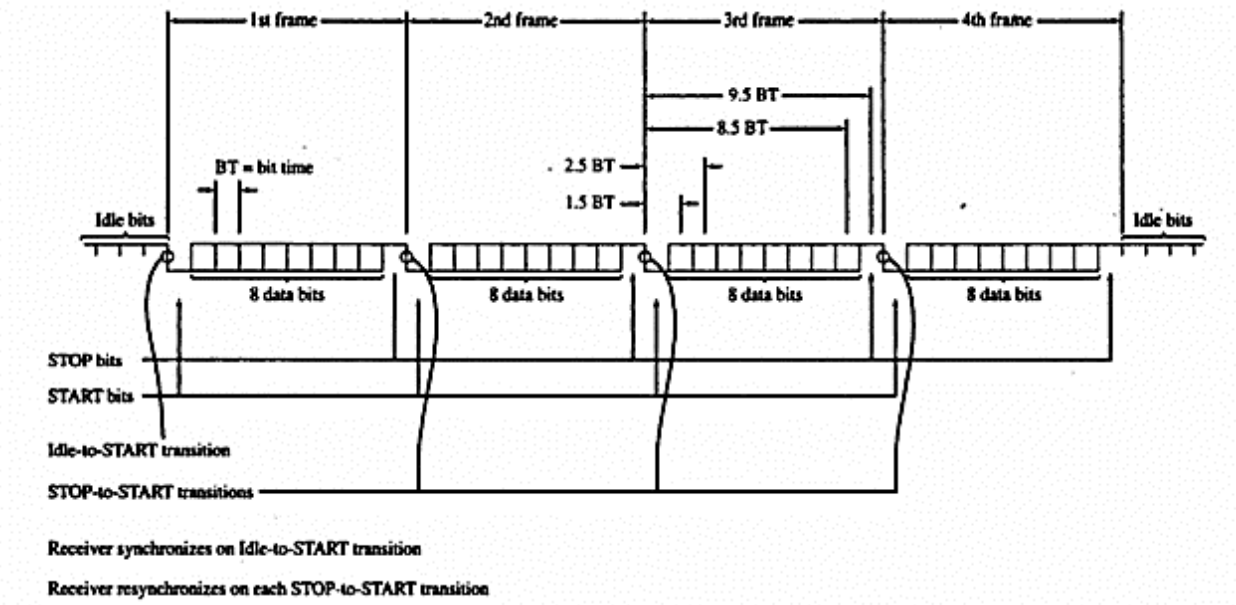
Timer1/CCP2 control of ADC

8. Explain UART waveform briefly.

UART:

- UART (Universal Asynchronous Receiver Transmitter) are one of the basic interfaces which provide a cost effective simple and reliable communication between one controller to another controller or between a controller and PC.
- It is a module included in the following parts: PIC 16C63, PIC 16C65A, PIC 16C73A and PIC 16C74A.It is omitted from the following: PIC 16C62A, PIC 16C64A, PIC 16C72

UART waveform:



Four data frames having a serial protocol of one START bit, eight data bit and one STOP bit

- When serial data is transmitted asynchronously the data stream is generated with the transmitter's clock. The receiver must synchronize the incoming data stream to the receiver's clock.
- An example of the transmission of 4bytes is shown in fig. Each 8-bit byte is framed by a START bit and a STOP bit. For transmission at 9600Bd, each of these bits lasts for the bit time (BT) of 1/9600 second.
- Before the first frame is transmitted, the line from the transmitter's TX output to the receiver's RX input idles high.
- The receiver monitors its RX input, waiting for the line to drop low because of the transmission of the (low) START bit.
- The receiver synchronizes on this high-to-low transition.
- Then the receiver reads the 8 bits of serial data by sampling the RX input at
  1.5 BT, 2.5 BT, 3.5 BT, 4.5 BT, 5.5 BT, 6.5 BT, 7.5 BT and 8.5 BT
- As shown in fig. It checks that the framing of the byte has been interpreted correctly by reading what should be a high STOP bit at 9.5 BT.
- If the RX line is actually low at this time, for whatever reason, the receiver sets a flag to indicate a framing error.
- Regardless of whether or not a framing error occurs, the receiver then begins again, resynchronizing upon the next high-to-low transition of the RX line.
- Because of this resynchronization, the receiver can generate its own baud-rate clock that only approximates the transmitter's baud rate clock and yet the receiver can recover the serial data perfectly.

9. Describe BAUD-RATE SELECTION with diagram

Baud-rate selection:

- A desired baud rate can be approximated by the UART's baud rate generator.
- If the crystal clock rate were selected to be a carefully chosen multiple of the desired baud rate, then the baud-rate generator would produce the desired baud-rate exactly.
  - The clock rates used by Microchip to characterize the three speed grades of their parts 4MHz, 10MHz and 20MHz
- Do not provide exact multiples of the popular 9,600 Bd and 19,200 Bd rates commonly used by personal computer serial ports.

- However, the flexibility of the baud-rate generator circuitry permits close approximations to both 9,600 Bd and 19,200 Bd with any of the standard crystal clock rates.
- The baud rate is derived from the crystal rate using an 8-bit presettable divider and a fixed divider of either 16 or 64 as shown in fig. b.
- The result are tabulated in fig. a. Even in the worst case, the percent error of the approximate baud rate is only one third of the percent error that cannot be tolerated by the UART.

Nominal Baud rate	OSC=4MHz			OSC = 10MHz			OSC = 20MHz		
	BRGH	SPBRG	%error	BRGH	SPBRG	%error	BRGH	SPBRG	%error
9,600 baud	1(high)	25	+0.16%	1(high)	64	+0.16%	1(high)	129	+0.16%
19,200 baud	1(high)	12	+0.16%	1(high)	32	-1.4%	1(high)	64	+0.16%

a) Register contents and accuracy of approximated baud rate

For BRGH=1(high speed baud rate)

$$\text{Baud rate} = \frac{\text{OSC}}{16(\text{SPBRG}+1)}$$

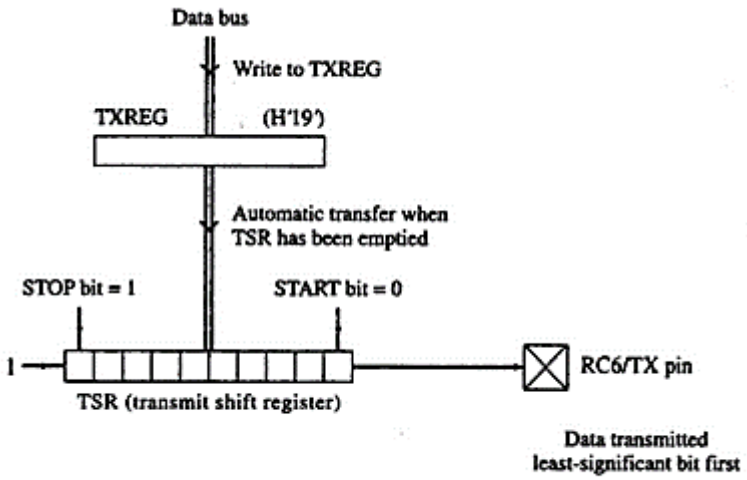
For BRGH=0(low speed baud rate)

$$\text{Baud rate} = \frac{\text{OSC}}{64(\text{SPBRG}+1)}$$

b) Relationship between OSC,BRGH,SPBRG and baud rate

Setup for 9,600 Bd and 19,200 baud

UART DATA HANDLING CIRCUITRY:



(a) Transmit data circuit

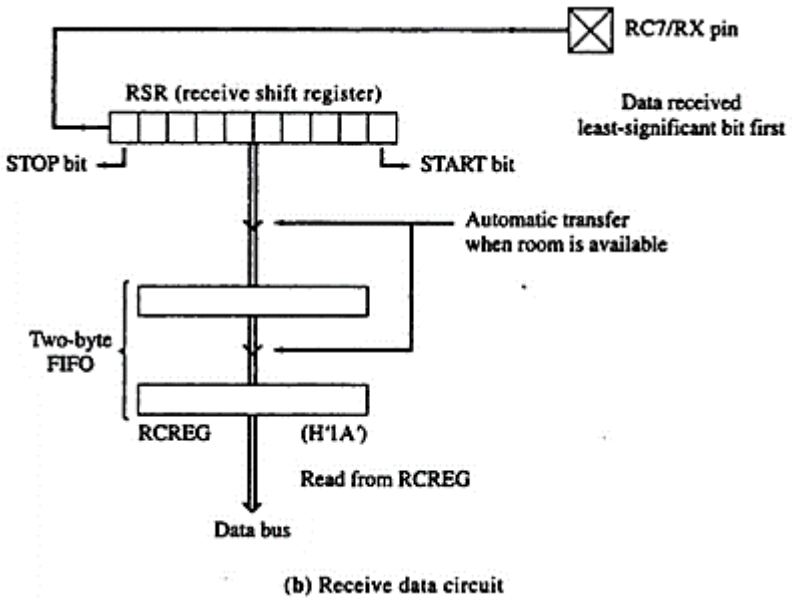
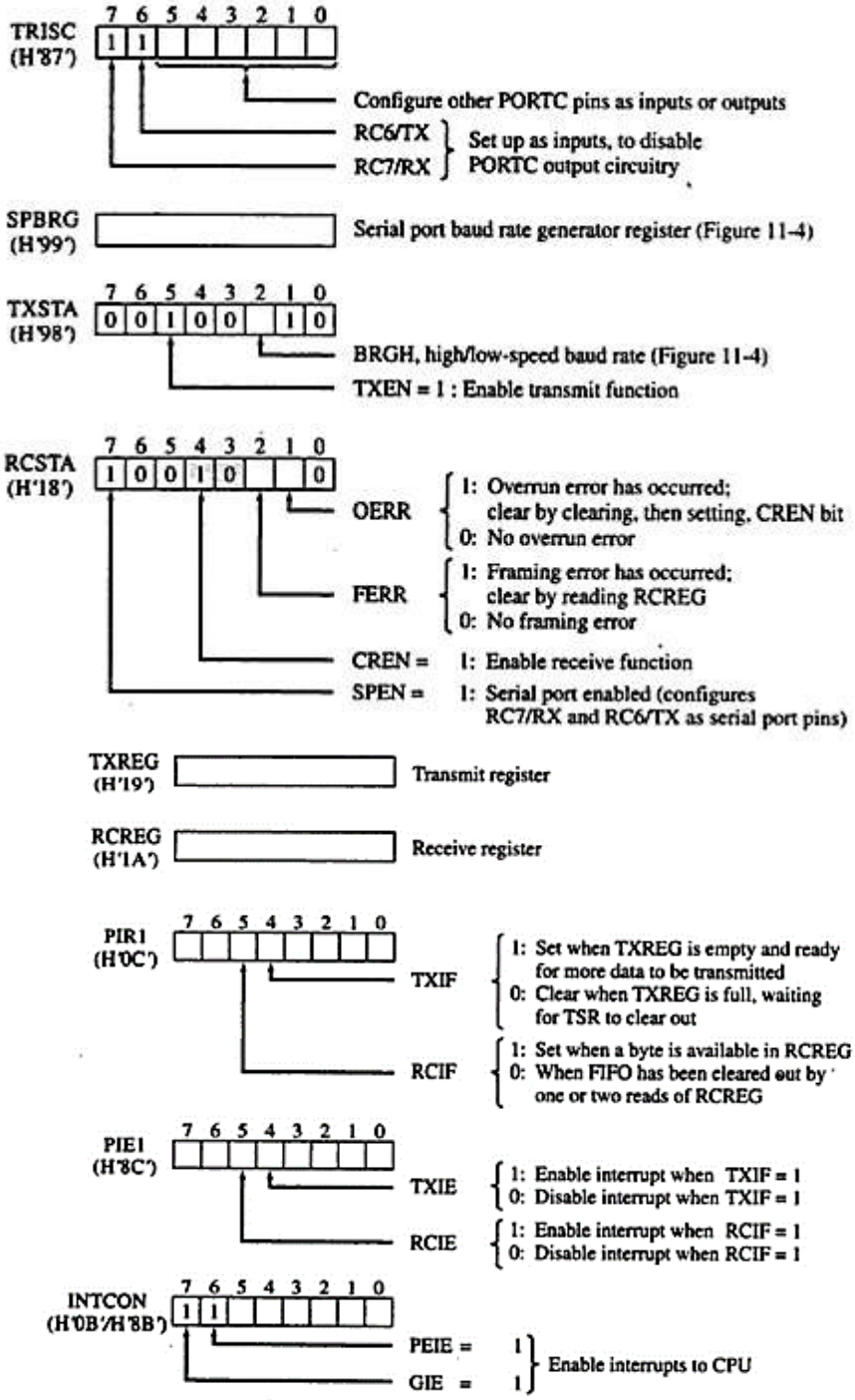


Fig. UART 'S data –handling circuitry

- The transmit data circuit is shown in fig 5a. To transmit a byte of data serially from the TX pin, the byte is written to the TXREG register.
- Assuming there is not already data in the TSR (transmit shift register)the content of TXREG will be automatically transferred to the TSR, making TXREG available for a second byte even as the first byte is being shifted out of the TX pin, framed by START and STOP bits.
- The receive data circuit is similar, with received data shifted into the RSR (receive shift register).
- When it is in place, the STOP bit is checked and an error flag is set if the STOP bit does not equal one.
- In any case, the received byte is automatically transferred into a 2 byte FIFO (first in first out memory).
- If the FIFO was initially empty, the received byte will fall through to the RCREG (receive register) virtually immediately, where it is ready by the CPU.
- If the CPU is slow in reading the RCREG, a second byte can be received at the RX pin.
- When it is in place in the RSR, it will follow the first byte into the 2-byte FIFO.
- At that point, the FIFO is full. If a third byte enters the RX pin and is shifted all the way across the RSR before at least one of the two bytes in the FIFO has been read, then the new byte will be lost.
- An overflow error flag will be set, alerting the receiver software of the loss of a byte of data.
- At 9,600 Bd it takes 10/9,600 second or just a little longer than a millisecond, to receive each byte.
- If the received bytes are handled under interrupt control, each byte should be easily handled in a timely fashion, well before an overrun error can ever occur.
- No, other interrupt handler should be permitted to lock out this or any other interrupt source from anywhere near a millisecond.

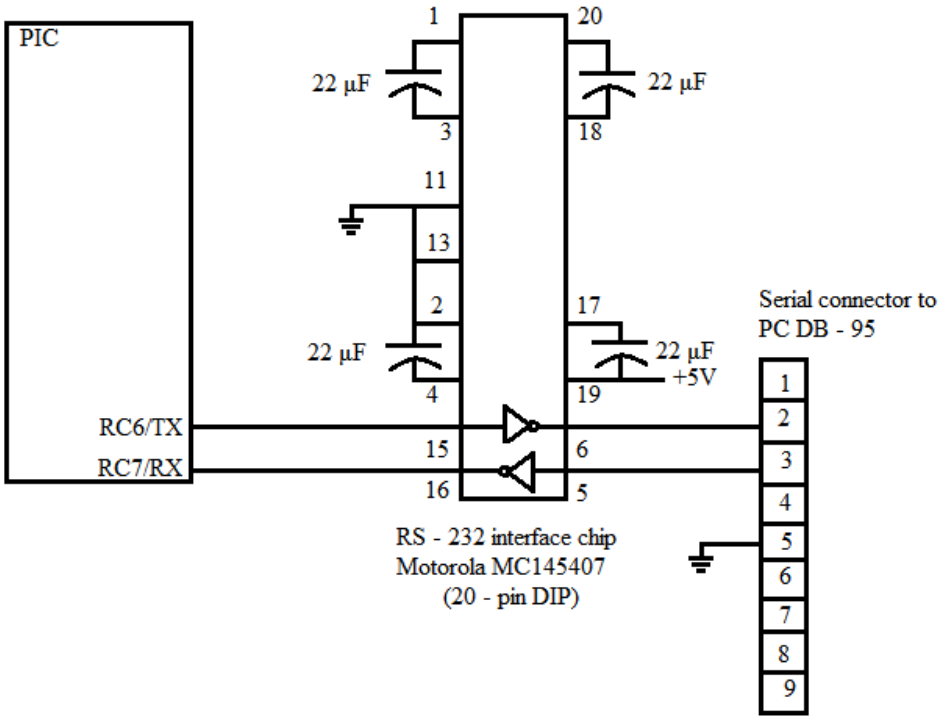
10. What is UART initialization

- The register involved with UART use are shown in fig. The data direction bits associated with the RC6/TX pin and the RC7/RX pin must both be set up as inputs, with ones in bits 6 and 7 of the TRISC register.
- The setting of these two bits disables the general I/O port output circuitry associated with these two pins.



UART registers

- These handling of these bits of TRISC stands in contrast to the clearing of bits 3 and 5 of TRISC in support of the serial peripheral interface output pins, as shown in fig.

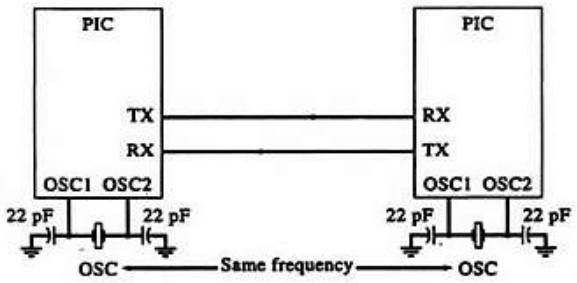


PIC UART interface to a PC

- The UART's baud rate and it transmit and receive functions are initialized by writes to SPBRG, TXSTA and RCSTA, as shown in fig 4 and fig.6.
- At 9,600 Bd, each transfer takes about a millisecond, so sending or receiving a string of characters is best carried out under interrupt control.
- The flag and interrupt enable bits of the PIR1, PIE1 and INTCON register control the timing of the CPU's interaction with the UART.

UART USE:

- A major application for the PIC's UART is to provide a two –wire (plus ground) serial interface to a personal computer.
- The circuit of fig.8.1 uses a Motorola chip to translate between the 0V and +5V logic level signal swings on the PIC's RX and TX pins and± 10V signal swings that support the RS-232 interface requirements.
- Both the PIC and the PC should be set up for the same baud rate (eg. 9,600 Bd) and for one start bit, eight data bits, one stop bit and no parity.
- Given this setup the PIC will respond to RCIF interrupt by reading each byte from the RCREG register sent by the PC.
- The RCIF flag will clear itself when the byte from RCREG leaves the receive circuit's FIFO empty.
- The PIC sends out string of bytes by writing them, one by one under interrupt control, to TXREG.
- The TXIF flag takes care of itself, clearing automatically when TXREG is written to and setting again as the data written to TXREG are automatically transferred to the transmit shift register .
- At the completion of sending the string of bytes to the pc, the TXIE bit in the PIE1 register is cleared to disable further "transmit" interrupts until another string needs to be sent to the PC.
- Another application of the PIC UART is to couple is to couple two PIC's together. In this way some of the work that would be done by one PIC is off-loaded to a second PIC.



(a) Circuit

	OSC = 4 MHz	OSC = 10 MHz	OSC = 20 MHz
Baud Rate	250 kbaud	625 kbaud	1.25 Mbaud
Time to transfer one byte	40 μs	16 μs	8 μs

Setup for maximum transfer rate

BRGH = 1, SPBRG = 00H, Baud rate OSC/16

UART interconnection of two PICs

- Fig.8. 2 shows this connection of two PIC using the maximum possible baud rate to obtain fast coupling between the two PIC's.
- Within 40 internal clock cycle what is written into one PIC's TXREG register appears in the other PIC'S RCREG REGISTER.
- Carrying out transfer at this fast rate calls for some precautions if overrun errors are to be avoided, given PIC's that are trying to carry out tasks in addition to monitoring the UART 's RCREG register.
- A PIC can only receive 2 bytes into its FIFO without reading them immediately. Any further bytes received will be discarded until the earlier bytes are read out of the FIFO making room for new bytes.
- One application that can easily bypass this limitation is illustrated in fig.
- The slave PIC is used a phoneme generator chip and a speaker to speak any word that is included in its dictionary of N words.
- The code representing each word is used to access a string of phoneme codes in the slave PIC's program area and sent to the phoneme generator chip, one by one, producing a vocalization of the desired dictionary word.
- Given this scenario, the master PIC can send a 1-byte code to the slave PIC to initiate the vocalization of any word in a dictionary of upto 256 words.
- When the slave PIC has the time to read the received word, it can respond by sending a byte of acknowledge back to the master PIC.
- This handshake procedure ensures that no byte will ever be lost because of an overrun error.
- With room for 2 bytes in the FIFO, the dictionary can easily be extended to more than 256 words by using a 2-byte code extended to more than 256 words by using a 2-byte code to identify each of the words.
- The slave PIC in this application can go one step further by letting the master PIC quickly download complete sentences of words.
- In this way, the master PIC can avoid getting tied up with a slow sequence of transfer dictated by the rate at which the phoneme chip can generate the vocalization of each word.
- The slave PIC need only handshake for each received byte and then put it in a queue.
- As the vocalization of each word is completed, the slave PIC goes to this queue for the next word to be spoken.

11. Discuss in detail about interfacing the keyboard and 4x4matrixkeypad connected to PORTB.

KEYBOARD INTERFACING

- Keyboards and LCDs are the most widely used input/output devices and a basic understanding of them is essential.

Interfacing the keyboard to the PIC18:

- At the lowest level, keyboards are organized in a matrix of rows and columns.
- The CPU accesses both rows and columns through ports; therefore, with two 8-bit ports, an 8 x 8 matrix of keys can be connected to a microprocessor.
- When a key is pressed, a row and a column make a contact; otherwise, there is no connection between rows and columns.
- In IBM PC keyboards, a single microcontroller takes care of hardware and software interfacing of the keyboard.
- In such systems, programs stored in the ROM of the microcontroller scan the keys continuously, identify which one has been activated, and present it to the motherboard.

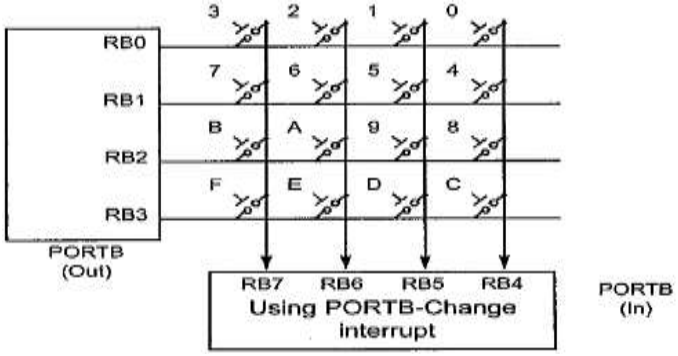
Inprogramming for keypad interfacing we must have two processes:

(a) Keypress detection
- The interrupt method
- The scanning method.

(b) Key identification.

- In the PIC18, the PORTB - Change interrupt can be used to implement the interrupt - based keypress detection.

Interrupt method of keypress detection:



4x4 matrix keypad connected to PORTB.

- The rows are connected to PORTB.
- Low (RB3-RBO) and the columns are connected to PORTB.
- High (RB7-RB4), which is the PORTB-Change interrupt.
- Any changes on the RB7-RB4 pins will cause an interrupt indicating a key press.

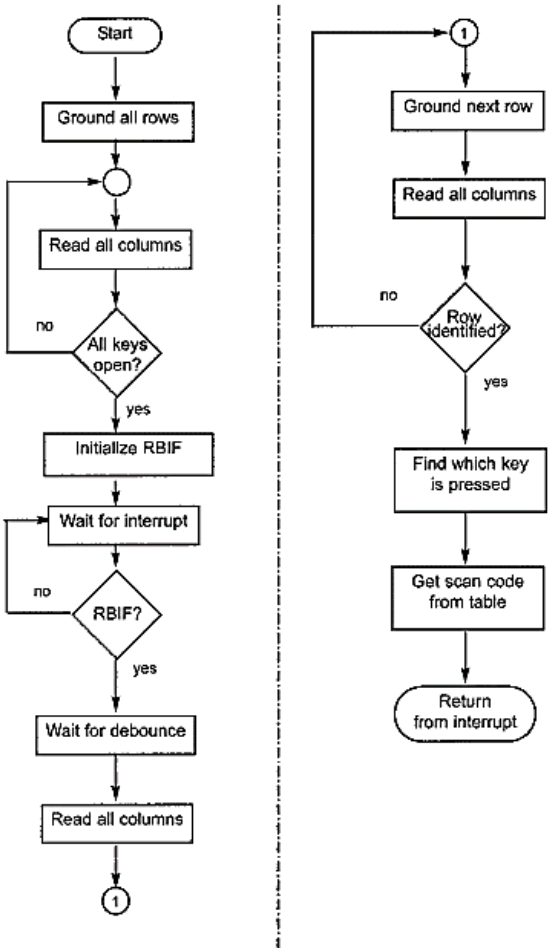
Examine Program 9, which goes through the following stages:

1. To make sure that the proceeding key has been released, O are output to all rows at once, and the columns are read and checked repeatedly until all the columns are HIGH. When all columns are found to be HIGH, the program waits for a short amount of time before it goes to the next stage of waiting for a key to be pressed.
2. To see if any key is pressed, the columns are connected to the PORTB-Change interrupt. Therefore, any key press will cause an interrupt and the microcontroller will execute the ISR. The JSR must do two things:
   (a) Ensure that the first key press detection was not erroneous due to spike noise,

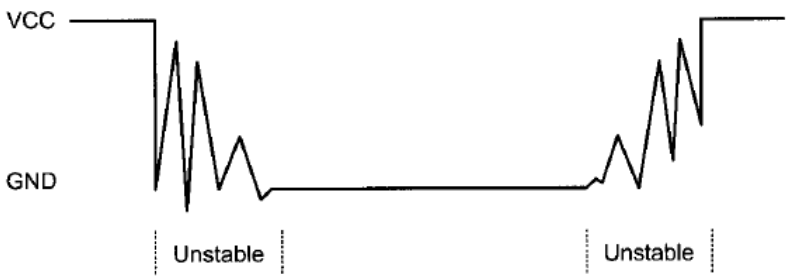
(b) Wait 20 ms to prevent the same key press from being interpreted as multiple key presses. See Figure for keyboard debounce.

3. To detect which row the key press belongs to, the microcontroller grounds one row at a time, reading the columns each time. If it finds that all columns are HIGH, this means that the key press cannot belong to that row; therefore, it grounds the next row and continues until it finds the row the key press belongs to. Upon finding the row that the key press belongs to, it sets up the starting address for the look-up table holding the scan codes (or the ASCII value) for that row and goes to the next stage to identify the key.
4. To identify the key press, the microcontroller rotates the column bits, one bit at a time, into the carry flag and checks to see if it is LOW. Upon finding the zero, it pulls out the ASCII code for that key from the look-up table; otherwise, it increments the pointer to point to the next element of the look-up table. Figure flowcharts this process.

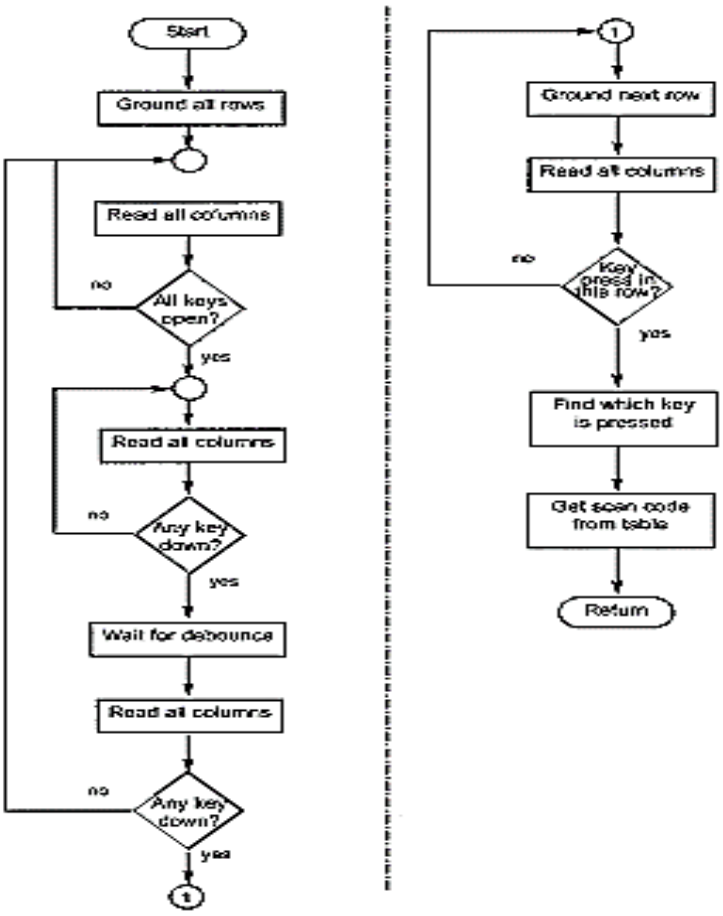
- The look-up table method shown in Program can be modified to work with any matrix up to 8 x 4.
- Figure provides the flowchart for Program 12-4 for scanning and identifying the pressed key.
- Examine Program 9-4. Notice in that program that the interrupt detects the key press. Then it is the job of the ISR to identify to which key the key press belongs (key identification).
- Program 9-4C is a C18 version of Program 9-4. In the Assembly version (12-4), the character is placed on PORTO, while in the Cl8 version (12-4C), it is sent to the serial port to be displayed on the monitor.



Flowchart for program 4



Keyboard debounce



Flowchart of scanning method of keypress detection

Scanning method for keypress detection:

- Another method for key press detection is by scanning. In this method, to detect a pressed key, the microcontroller grounds all rows by providing 0 to the output latch, then it reads the columns.
- If the data read from the columns are equal to 1111, no key has been pressed and the process continues until a key press is detected.
- If one of the column bits has a zero, however, this means that a key press has occurred. After a key press is detected, the microcontroller will go through the process of identifying the key.
- Starting with the top row, the microcontroller grounds it by providing a LOW to the first row only; then it reads the columns.
- If the data read is all 1 s, no key in that row is activated and the process is moved to the next row.
- It grounds the next row, reads the columns, and checks for any zero. This process continues until the row is identified.

- After identification of the row in which the key has been pressed, the next task is to find out which column the pressed key belongs to.
- This should be easy since the microcontroller knows at any time which row and column are being accessed.
- Figure 9-9 shows the flowchart for this method. The program implementation is left to the reader.
- Some IC chips, such as National Semiconductor's MM74C923, incorporate keyboard scanning and decoding all in one chip. Such chips use combinations of counters and logic gates (no microcontroller) to implement the underlying concepts presented in this section.

12. Explain in detail about LCD operation

- In recent years the LCD has been finding widespread use replacing LEDs (seven-segment LEDs or other multi-segment LEDs).
- This is due to the following reason:

The declining prices of LCDs

- The ability to display numbers, characters, and graphics.
- This is in contrast to LEDs, which are limited to numbers and a few characters.
- Incorporation of a refreshing controller into the LCD, thereby relieving the CPU of the task of refreshing the LCD. In contrast, the LED must be refreshed by the CPU (or in some other way) to keep displaying the data.
- Ease of programming for characters and graphics.

LCD pin descriptions

- The LCD discussed in this section has 14 pins. The function of each pin is given in for various LCDs.

Vcc, Vss, and VEE:

- While Vcc and Vss provide +5V and ground, respectively, VEE is used for controlling LCD contrast.

RS, register select:

- There are two very important register inside the LCD.
- The RS pin is used for their selection as follows.
- If RS=0, the instruction command code register is selected, allowing the user to send a command code register is selected, allowing the user to send a command such as clear display, cursor at home, and so on.
- If RS=1 the data register is selected, allowing the user to send data to be displayed on the LCD.

R/W, read/write:

- R/W input allows the user to write information to the LCD or read information from it. R/W=1 when reading; R/W=0 when writing.

E, enable:

- The enable pin is used by the LCD to latch information presented to its data pins.
- When data is supplied to data pins, a high-to-low pulse must be applied to the En pin in order for the LCD to latch in the data present at the data pins.
- This pulse must be a minimum of 450 ns wide. In this book we call this delay the SDELAY (short delay) to distinguish it from other delays.

D0-07:

- The 8-bit data pins, DO-D7, are used to send information to the LCD or read the contents of the LCD's internal registers.
- To display letters and numbers, we send ASCII codes for the letters A-Z, a-z, and numbers 0-9 to these pins while making RS= 1.
- There are also instruction command codes that caution command codes that can be sent to the LCD to clear the display or force the cursor to the home position or blink the display or force the cursor to the home position or blink the cursor.
- Table 10-2 lists the instruction command codes. To send any of the commands listed in Table 10-2 to the LCD, make pin RS = 0.
- For data make RS = 1. Then send a high-to-low pulse to the E pin to enable the internal latch of the LCD.
- There are two ways to send characters (command /data) to the LCD: (1) use a delay before sending the next one, (2) use the busy flag to see if the LCD is ready for the next one.

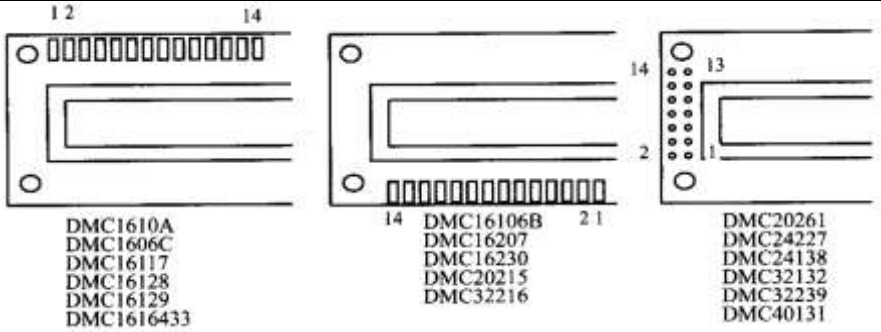
Table 10.1: Pin Descriptions for LCD

P_{in}	Symbol	I/O	Description
1	V_{SS}	-	Ground
2	V_{CC}	-	+5Vpower supply
3	V_{EE}	-	Power supply to control contrast
4	RS	I	RS=0 to select command register,RS=1 to select data register
5	R/W	I	R/W=0 for write,R/W=1 to select data register
6	E	I/O	The 8-bit data bus
7	DBO	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 8-bit data bus
12	DB5	I/O	The 8-bit data bus
13	DB6	I/O	The 8-bit data bus
14	DB7	I/O	The 8-bit data bus

Table 10.2 LCD Command Codes

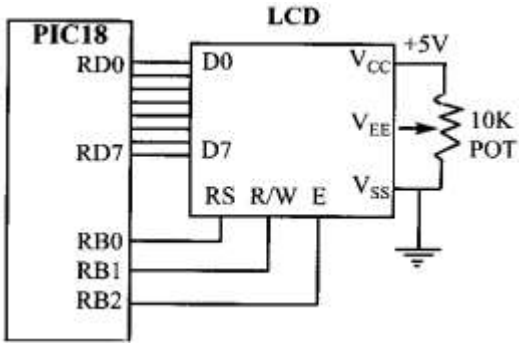
Code(Hex)	Command to LCD instruction Register
1	Clear display screen
2	Return home
4	Decrement cursor(shift cursor to left)
6	increment cursor(shift cursor to right)
5	Shift display right
7	Shift display left
8	Display off, cursor off
A	Display off, cursor on

C	Display on, cursor off
E	Display on, cursor blinking
F	Display on, cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift the entire display to left
1C	Shift the entire display to right
80	Focus cursor to beginning of 1st line
CO	Focus cursor to beginning of 2nd line
38	2lines and 5x7 matrix



Sending commands and data to LCDs with a time delay:

- Program 10-1 shows how to send characters (command/data) to the LCD without checking the busy flag. Notice that we need to wait 5-10 ms (DELAY) between issuing each character to the LCD.
- We call this delay simply DELAY. In programming an LCD, we also need a long delay for the power-up process.
- We call it LDELAY (long delay). SDELAY (short delay) is used to make the En signal wide enough for the LCD's enable input.



The LCD connections to the microcontroller.

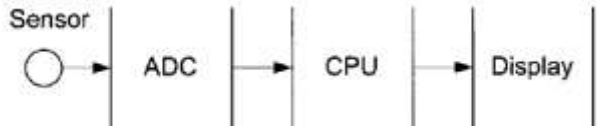
Sending command or data to the LCD using busy flag

- We use RS= 0 to read the busy flag bit to see if the LCD is ready to receive information.
- The busy flag is D7, and can be read when RfW=1 and RS = 0, as follows: if RIW = 1, RS= 0.
- When D7 = 1 (busy flag= 1), the LCD is busy taking care of internal operations and will not accept any new information.
- When D7= 0, the LCD is ready to receive new information

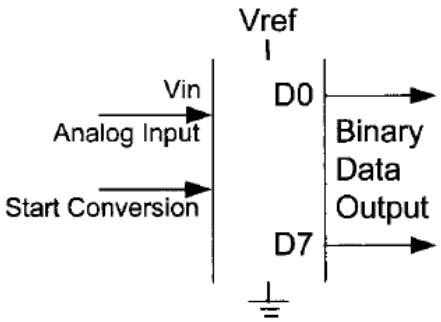
13. Discuss the ADC (analog to digital converter) section of the PIC18 chip.

ADC devices

- Analog-to-digital converters are among the most widely used devices for data acquisition.
- Digital computers use binary (discrete) values, but in the physical world everything is analog (continuous).
- Temperature, pressure (wind or liquid), humidity, and velocity are a few examples of physical quantities that we deal with every day.
- A physical quantity is converted to electrical (voltage, current) signals using a device called a transducer.
- Transducers are also referred to as sensors.
- Sensors for temperature, velocity, pressure, light, and many other natural quantities produce an output that is voltage (or current).
- Therefore, we need an analog to-digital converter to translate the analog signals to digital numbers so that the microcontroller can read and process them. See Figures 13-1 and 13-2.



Microcontroller connection to sensor via ADC



An 8-bit ADC block diagram

Resolution versus step size for ADC (V_{ref} = 5V)

n-bit	Number of steps	Step size(mV)
8	256	5/256=19.53
10	1,024	5/1,024=4.88
12	4,096	5/4,096=1.2
16	65,536	5/65,536=0.076

Characteristics of ADC:

Resolution:

- ADC has n-bit resolution, where n can be 8, 10, 12, 16, or even 24 bits.
- The higher-resolution ADC provides a smaller step size, where step size is the smallest change that can be discerned by an ADC.
- Some widely used resolutions for ADCs are shown in Table 11-1.
- Although the resolution of an ADC chip is decided at the time of its design and cannot be changed, we can control the step size with the help of what is called Vref.

Conversion time

- In addition to resolution, conversion time is another major factor in judging an ADC.
- Conversion time is defined as the time it takes the ADC to convert the analog input to a digital (binary) number.
- The conversion time is dictated by the clock source connected to the ADC in addition to the method used for data conversion and technology used in the fabrication of the ADC chip such as MOS or TTL technology.

Vref

- V ref is an input voltage used for the reference voltage.
- The voltage connected to this pin, along with the resolution of the ADC chip, dictate the step size.
- For an 8-bit ADC, the step size is Vref 256 because it is an 8-bitADC, and 2 to the power of 8 gives us 256 steps. See Table 11-1.
- For example, if the analog input range needs to be 0 to 4 volts, Vref is connected to 4 volts.
- That gives 4 V/256 =15.62 m V for the step size of an 8-bit ADC.
- In another case, if we need a step size of 10 mV for an 8-bit ADC, then Vref = 2.56 V, because 2.56 V/256 = 10 mV.
- For the 10-bit ADC, if the Vref = 5V, then the step size is 4.88 mV as shown in Table 11-1. Tables 11-2 and 11-3 show the relationship between the Vref and step size for the 8- and 10-bit ADCs, respectively.
- In some applications, we need the differential reference voltage where Vref= Vref (+) - Vref (-).
- Often the Vref(•) pin is connected to ground and the Vref (+) pin is used as the Vref.

Table Vref Relation to V_{in} Range for an 8-bitADC

V_{ref}(v)	V_{in}(v)	Step size(mV)
5.00	0 to 5	5/256=19.53
4.0	0 to 4	4/256=15.62
3.0	0 to 3	3/256=11.71
2.56	0 to 2.56	2.56/256=10
2.0	0 to 2	2/256=7.81
1.28	0 to 1.28	1.28/256=5
1	0 to 1	1/256=3.90

Table V_{ref} relation to V_{in} range for an 10-bit ADC

V_{ref}(v)	V_{in}(v)	Step size(mV)
5.00	0 to 5	5/1,024=4.88
4.096	0 to 4.096	4.096/1,024=4
3.0	0 to 3	3/1,024=2.93
2.56	0 to 2.56	2.56/1,024=2.5
2.048	0 to 2.048	2/1,024=2
1.28	0 to 1.28	1/1,024=1.25
1.024	0 to 1.024	1.024/1,024=1

Digital data output

- In an 8-bit ADC we have an 8-bit digital data output of D0 - D7 while in the10-bit ADC the data output is D0 - D9.
- To calculate the output voltage, we use the following formula:

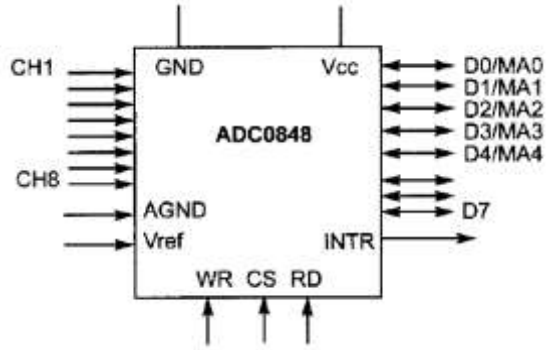
$$Dout = V_{in} / \text{step size}$$

where Dout = digital data output (in decimal), Vin = analog input voltage, and step size (resolution) is the smallest change, which is Vref/256 for an 8-bit ADC.

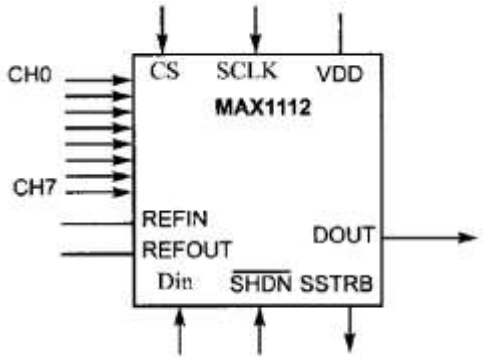
- See Example 11-1. This data is brought out of the ADC chip either one bit at a time (serially), or in one chunk, using a parallel line of outputs.

Parallel versus serial ADC:

- The ADC chips are either parallel or serial. In parallel ADC, we have 8 or more pins dedicated to bringing out the binary data, but in serial ADC we have only one pin for data out.
- That means that inside the serial ADC, there is a parallel-in-serial-out shift register responsible for sending out the binary data one bit at a time.
- The D0-07 data pins of the 8-bit ADC provide an 8-bit parallel data path between the ADC chip and the CPU.



ADC 0848 parallel ADC Block Diagram



MAX1112 Serial ADC Block Diagram

- In the case of the 16-bit parallel ADC chip, we need 16 pins for the data path.
- In order to save pins, many 12- and 16-bitADCs use pins DO-D7 to send out the upper and lower bytes of the binary data.
- In recent years, for many applications where space is a critical issue, using such a large number of pins for data is not feasible.

- For this reason, serial devices such as the serial ADC are becoming widely used.
- While the serial ADCs use fewer pins and their smaller packages take much less space on the printed circuit board, more CPU time is needed to get the converted data from the ADC because the CPU must get data one bit at a time, instead of in one single read operation as with the parallel ADC.
- ADC848 is an example of a parallel ADC with 8 pins for the data output, while the MAX 1112 is an example of a serial ADC with a single pin for Dout.
- Figures 11-3 and 11.4 show the block diagram for ADC848 and MAXI 112, respectively.

Analog input channels:

- Many data acquisition applications need more than one ADC.
- For this reason, we see ADC chips with 2, 4, 8, or even 16 channels on a single chip. Multiplexing of analog inputs is widely used as shown in the ADC848 and MAXI 112.
- In these chips, we have 8 channels of analog inputs, allowing us to monitor multiple quantities such as temperature, pressure, heat, and so on.
- PIC18 microcontroller chips come with 5 to 15 ADC channels, depending on the family member. The PIC18 ADC feature is discussed in the next section.

Start conversion and end- of - conversion signals

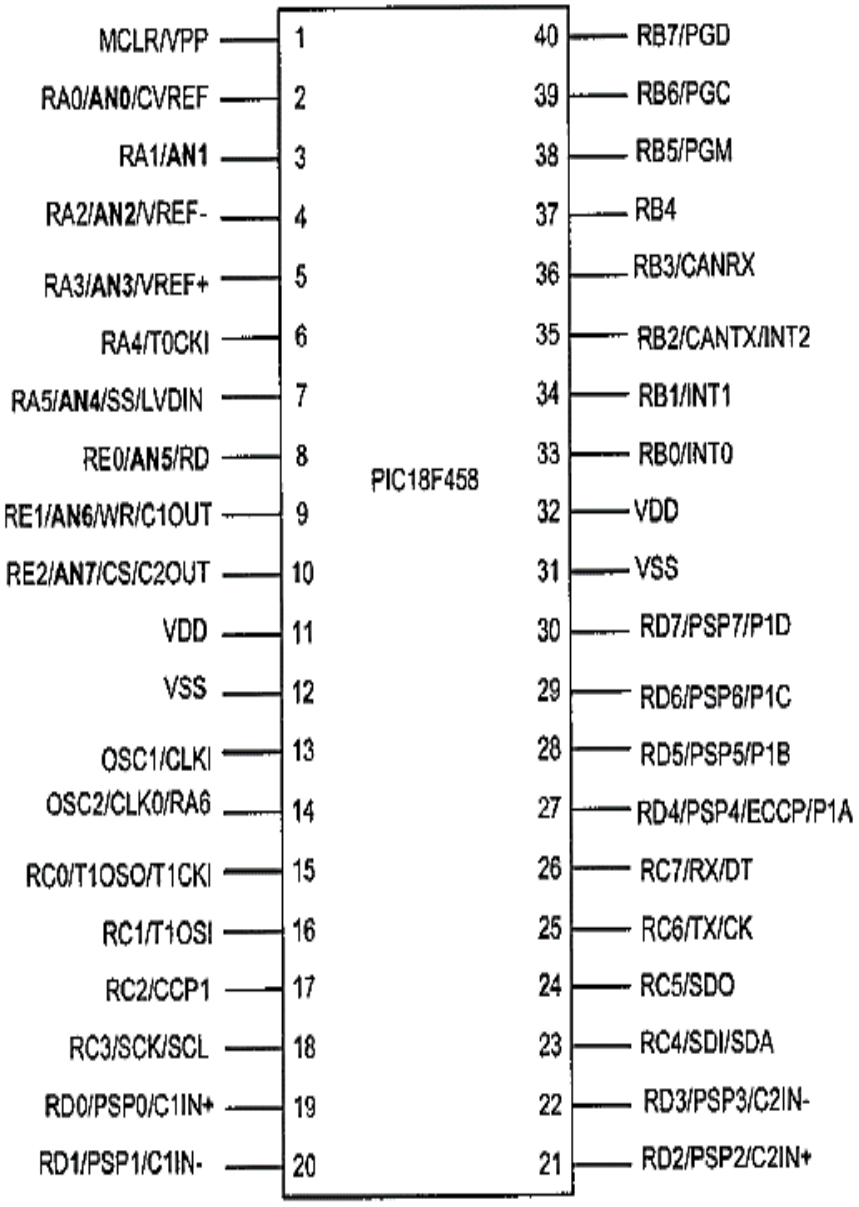
- The fact that we have multiple analog input channels and a single digital output register makes it necessary for start conversion (SC) and end-of-conversion (EOC) signals.
- When SC is activated, the ADC starts converting the analog input value of Vin to an n-bit digital number. The amount of time it takes to convert varies depending on the conversion method.
- When the data conversion is complete, the end-of-conversion signal notifies the CPU that the converted data is ready to be picked up.
- From the discussion we conclude that the following steps must be followed for data conversion by an ADC chip:
  1. Select a channel.
  2. Activate the start conversion (SC) signal to start the conversion of analog input.
  3. Keep monitoring the end-of-conversion (EOC) signal.
  4. After the EOC has been activated, we read data out of the ADC chip.

14. Derive PIC18F452/458 ADC features with corresponding programming

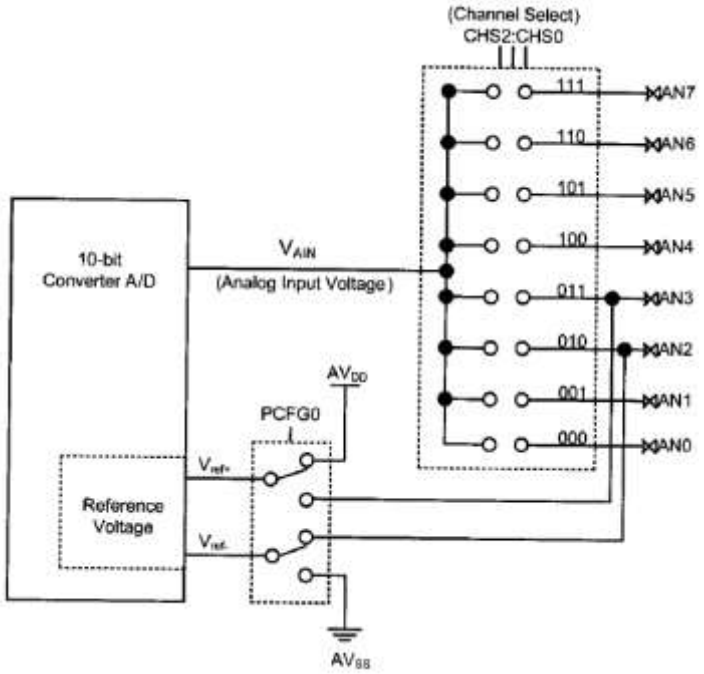
The ADC peripheral of the PIC I 8 has the following characteristics:

- It is a 10-bit ADC.
- It can have 5 to 15 channels of analog input channels, depending on the family member.
- In PIC 18452/458, pins RAO-RA 7 of PORTA are used for the 8 analog channels. See Figures 13-5A and 13-SB.
- The converted output binary data is held by two special function registers called ADRESL (AID Result Low) and ADRESH (AID Result High).
- Because the ADRESH: ADRESL registers give us I 6 bits and the ADC data out is only 10 bits wide, 6 bits of the 16 are unused.
- We have the option of making either the upper 6 bits or the lower 6 bits unused.
- We have the option of using Vdd (Vcc), the voltage source of the PIC 18 chip itself, as the Vref or connecting it to an external voltage source for the Vref.

- The conversion time is dictated by the Fosc of crystal frequency connected to the OSCs pins. While the Fosc for PIC18can be as high as 40 MHz, the conversion time cannot be shorter than 1.6 ms.
- It allows the implementation of the differential Vref voltage using Vref(+) and Vref(-) pins, where Vref= Vref (+)- Vref (-).



PIC18F458 pins with ADC channels shown in BOLD



PIC18 ADC channel and Reference selection

ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	--	ADON

ADCS2(from ADCON1)	ADCS1	ADCS0	Conversion clock source
0	0	0	Fosc/2
0	0	1	Fosc/8
0	1	0	Fosc/32
0	1	1	Internal RC used for clock source
1	0	0	Fosc/4
1	0	1	Fosc/16
1	1	0	Fosc/64
1	1	1	Internal RC used for clock source

CHS2	CHS1	CHS0	CHANNEL SELECTION
0	0	0	CHANO(AN0)
0	0	1	CHANO(AN1)
0	1	0	CHANO(AN2)
0	1	1	CHANO(AN3)
1	0	0	CHANO(AN4)
1	0	1	CHANO(AN5) not implemented on 28-pin PIC18
1	1	0	CHANO(AN6) not implemented on 28-pin PIC18
1	1	1	CHANO(AN7) not implemented on 28-pin PIC18

GO/DONE: A/D conversion status bit.

- 1=A/D conversion is in progress. This is used as start conversion, which means after the conversion is complete, it will go LOW to indicate the end-of-conversion.
- 0=A/D conversion is complete and digital data is available in registers ADRESH and ADRESL

ADON: A/D on bit

- 0 =A/D part of the PIC 18 is off and consumes no power. This is the default and we should leave it off for applications in which ADC is not used.
- 1 = A/D feature is powered up.

ADFM	ADCS2	--	--	PCFG3	PCFG2	PCFG1	PCFG0

ADCONO (AID Control Register 0)

15. Write in detail about ADCON1 register

ADCON1 register

- Another major register of the PIC18's ADC feature is ADCONI.
- The ADCON1 register is used to select the Vref voltage among other things.
- It is shown in Figure 13-7. After the AID conversion is complete, the result sits in registers ADRESL (AID Result Low Byte) and ADRESH (A/D Result High Byte).
- The ADFM bit of the ADCON1 is used for making it right-justified or left-justified because we need only 10 bits of the 16. See Figure 13-8.

ADFM: AID Result format select bit

- 1 = Right justified: The 10-bit result is in the ADRESL register

ADRESH: That means the 6 most significant bits of the ADRESH register are all Os.

- 0 =Left justified: The l O-bit result is in the ADRESL register and the upper 2 bits of ADRESL**.** That means the 6 least significant bits of the ADRESL register are all Os.

ADCS2: AID Clock Select bit 2.

- This bit along with the ADCS 1 and ADCSO bits of the ADCONO register decide the conversion clock for the ADC.
- The default value for ADCS2 is 0, which means setting the ADCSO and ADCS 1 values of ADCONO can give us clock conversion of Fosc/2, Fosc/8, and Fosc/32. See the ADCONO register.

PCFGs: A/D Port Configuration Control bits:

PCFGs	AN7	AN76	AN5	AN4	AN3	AN2	AN1	AN0	V_{ref+}	V_{ref-}	C/R
0000	A	A	A	A	A	A	A	A	V_{dd}	V_{SS}	8/0
0001	A	A	A	A	V_{ref+}	A	A	A	AN3	V_{SS}	7/1
0010	D	D	D	A	A	A	A	A	V_{dd}	V_{SS}	5/0
0011	D	D	D	A	V_{ref+}	A	A	A	AN3	V_{SS}	4/1
0100	D	D	D	D	A	D	A	A	V_{dd}	V_{SS}	3/0
0101	D	D	D	D	V_{ref+}	D	A	A	AN3	V_{SS}	2/1
011x	D	D	D	D	D	D	D	D	-	-	0/0
1000	A	A	A	A	V_{ref+}	V_{ref-}	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	V_{dd}	V_{SS}	6/0

1010	D	D	A	A	V_{ref+}	A	A	A	AN3	V_{SS}	5/1
1011	D	D	A	A	V_{ref+}	V_{ref-}	A	A	AN3	AN2	4/2
1100	D	D	D	A	V_{ref+}	V_{ref-}	A	A	AN3	AN2	3/2
1101	D	D	D	D	V_{ref+}	V_{ref-}	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	V_{dd}	V_{SS}	1/0
1111	D	D	D	D	V_{ref+}	V_{ref-}	D	A	AN3	AN2	1/2

- A = Analog input,
- D = Digital I/O
- CIR = # of analog input channels I # of pins used for AID voltage reference

- The default is option 0000, which gives us 8 channels of analog input and uses the Vdd of PIC 18 as Vref.

ADCONl (AID Control Register 1)

ADFM Bit and ADRES x Registers

- The port configuration for A/D channels is handled by the PCFG (A/D port configuration) bits.
- While in chips such as the PIC 18452/458, we can have up to8 channels of analog input, not all applications need that many ADC inputs.
- The PORTA pins ofRA~RA3 and RAS and REO~RE2 of PORTE are used for the analog input channels.
- With PCFG = 0110, we can use all the pins of the PORTA as the digital 1/0. The default is PCFG = 0000, which allows us to use all 8 pins for analog inputs.
- In that case Vref = Vdd, the same voltage source used by the PIC18 chip itself.
- In many applications we need Vref other than V dd. The AN3 pin can be used as an external source of voltage for Vref.
- For example, option PCFG = 0101 allows us to use two channels for analog inputs, AN3 = Vref, and the other 5 pins for the digital 110.
- In this case the Vss (Gnd) pin of the PIC18 is used for the Vref (-).See Examples 13-2 and 13-3.

16. For an 8-bit ADC, we have Vref = 2.56 V. Calculate the DO--D7 output if the analog input is: (a) 1.7 V, and (b) 2.1 V.

Solution:

Because the step size is 2.56/256 = 10 mV, we have the following:

(a) D_{out} = 1.7 V/10 mV = 170 in decimal, which gives us 10101011 in binary for D7-DO.

(b) D_{out} = 2.1 V/10 mV = 210 in decimal, which gives us 11010010 in binary for D7-DO.

17. For a PIC18-based system, we have Yref = Vdd = 5 V. Find (a) the step size, and (b) the ADCONl value if we need 3 channels. Assume that the ADRESH: ADRESL registers are right justified.

Solution:

(a) The step size is 5/1,024 = 4.8 mV.

(b) ADCONl = lxOOOlOO because option 100 gives us 3 analog input channels. The x= ADCS2 is decided by the conversion speed.

18. For a PIC18-based system, we have Vref = 2.56 V. Find (a) the step size, and (b) the ADCONl value if we need 3 channels. Assume that the ADRESH: ADRESL registers are right justified.

Solution:

(a) The step size is 2.56/1024 = 2.5 mV.

(b) ADCON 1 =1xOOOO11 because option 0011 gives us 3 analog input channels where x = ADCS2 is decided by the conversion speed

19. A PIC18 is connected to the 10 MHz crystal oscillator. Calculate the conversion time for all options of ADCS bits in both the ADCONO and ADCONI registers.

Solution:

The options for the conversion clock source for both ADCONO and ADCONl are as follows:

(a) For Fosc/2, we have 10MHz /2 = 5 MHz.
Tad = 1 /5 MHz = 200 ns. Invalid because it is faster than 1.6 µs.

(b) For Fosc/4, we have 10 MHz / 4 = 2.5 MHz.
Tad = 1 / 2.5 MHz = 400 ns. Invalid because it is faster than 1.6 µs.

(c) For Fosc/8, we have 10 MHz I 8 = 1.25 MHz.
Tad= 1/ 2.5 MHz= 800 ns. Invalid because it is faster than 1.6 µs.

(d) For Fosc/16, we have 10 MHz/ 16 = 625 kHz.
Tad =1 / 625 kHz = 1.6 µs. The conversion time = 12 x 1.6 µs = 19.2 µs

(e) For Fosc/32, we have 10 MHz/ 32 = 312.5 kHz.
Tad= 1/312.5 kHz = 3.2 µs. The conversion time = 12 x 3.2 µs = 38.4 µs

(f) For Fosc/64, we have 10 MHz I 64 = 156.25 kHz.
Tad= 1/156.25 kHz = 6.4 µs. The conversion time = 12 x 6.4 µs = 76.8 µs

- Notice that for the Fosc/4, Fosc/16, and Fosc/64 selections, we must use the ADSC2 bit in the ADCONl register, in addition to the ADCS bits in the ADCONO register.

23. A PIC 18 is connected to the 4 MHz crystal oscillator. Calculate the conversion time if we want to use only the ADCS bits of the ADCONO register.

Solution:

The options for the conversion clock source available in the ADCONO register are as follows:

(a) For Fosc/2, we have 4 MHz /2 = 2 MHz.

Tad= 1/2 MHz = 400 ns. Invalid because it is faster than 1.6 µs.

(b) For Fosc/8, we have 4 MHz /8 = 500 kHz.

Tad= 1/500 kHz = 2 µs. The conversion time= 12 x 2 µs = 24 µs

(c) For Fosc/32, we have 4 MHz /32 = 125 kHz.

Tad = 1/125 kHz = 8 µs. The conversion time = 12 x 8 µs = 96 µs

24. Find the values for the ADCONO and ADCONI registers for the following options: (a) channel ANO as analog input, (b) Vref+ = V dd, Vref- = Vss, (c) Fosc/64, (d) AID result is right justified, and (e) AID module is on.

Solution:

We have ADCONO = 1 0 0 0 0 X 1. With x = 0 we have 10000001.
We have ADCONl = 1 1 X X 1 1 1 0. With x = 0 we have 11001110.

25. How to Calculating A/D conversion time.

- By using the ADCS (ND clock source) bits of both the ADCONO and ADCON 1 registers we can set the ND conversion time.
- The conversion time is defined in terms of Tad, where Tad is the conversion time per bit.
- To calculate the Tad, we can select a conversion clock source of Fosc/2, Fosc/4, Fosc/8, Fosc/t 6, Fosc/32, or Fosc/64, where Fosc is the speed of the crystal frequency connected to the PIC18 chip.
- For the PIC18, the conversion time is 12 times the Tad. Notice that the Tad cannot be faster than 1.6 ms.
- We can also use the the internal RC oscillator for the conversion clock source, instead of the Fosc of the external crystal oscillator. In that case the Tad is typically 4-6 µs and conversion time is 12 x 6 µs = 72 µs.
- Another timing factor that we must pay attention to is the acquisition time (Tacq).
- After an ND channel is selected, we must allow some time for the sample-and-hold capacitor (C hold) to charge fully to the input voltage level present at the channel.
- It is only after the elapsing of this acquisition time that the ND conversion can be started.
- Although many factors (e.g., Vdd and temperature) affect the duration of Tacq, we can use a typical value of 15 µs.
- In some newer generations of the PIC18, we have the option of controlling the exact time of Tacq by programming the internal register ADCON2.
- In the PIC l 8F452/458, we have only the ADCONO and ADCON 1 registers.

Steps in programming the AID converter using polling

To program the AID converter of the PIC 18, the following steps must be taken:

Step1: Turn on the ADC module of the PIC18 because it is disabled upon power-on reset to save power. We can use the "BSF ADCONO, ADON" instruction.

Step 2: Make the pin for the selected ADC channel an input pin. We use "BSFTRI SA, x." or "BSF TRI SE, x" where x is the channel number.

Step 3: Select voltage reference and A/C input channels. We use registers ADCONO and ADCONl.

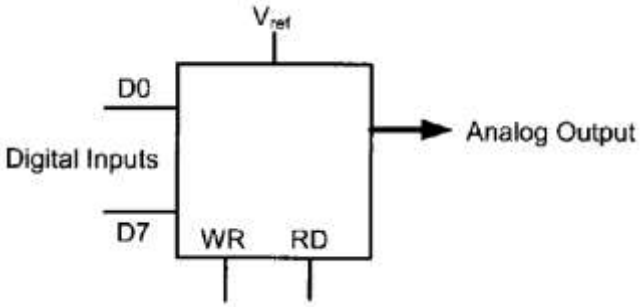
Step 4: Select the conversion speed. We use registers ADCON0 and ADCONl.

Step 5: Wait for the required acquisition time.
Step 6: Activate the start conversion bit of GO/DONE.
Step 7: Wait for the conversion to be completed by polling the end-of- conversion (GO/DONE) bit.
Step 8: After the GO/DONE bit has gone LOW, read the ADRESL and ADRESH registers to get the digital data output.
Step 9: Go back to step 5

26. Describe DAC INTERFACING of PIC18.

Digital-to-analog converter (DAC)

- The digital-to-analog converter (DAC) is a device widely used to convert digital pulses to analog signals interfacing a DAC to the PICI8.
- Two methods of creating a DAC: binary weighted and R/2R ladder.
- The vast majority of integrated circuit DA Cs, including the MC 1408 (DAC0808) used in this section, use the R/2R method because it can achieve a much higher degree of precision.
- The first criterion for judging a DAC is its resolution, which is a function of the number of binary inputs.
- The common ones are 8, 10, and 12 bits. The number of data bit inputs decides the resolution of the DAC because the number of analog output levels is equal to 2n, where n is the number of data bit inputs.
- Therefore, an 8-input DAC such as the DAC0808 provides 256 discrete voltage (or current) levels of output.
- Similarly, the 12-bit DAC provides 4,096 discrete voltage levels. There are also 16-bit DACs, but they are more expensive.



DAC Block Diagram

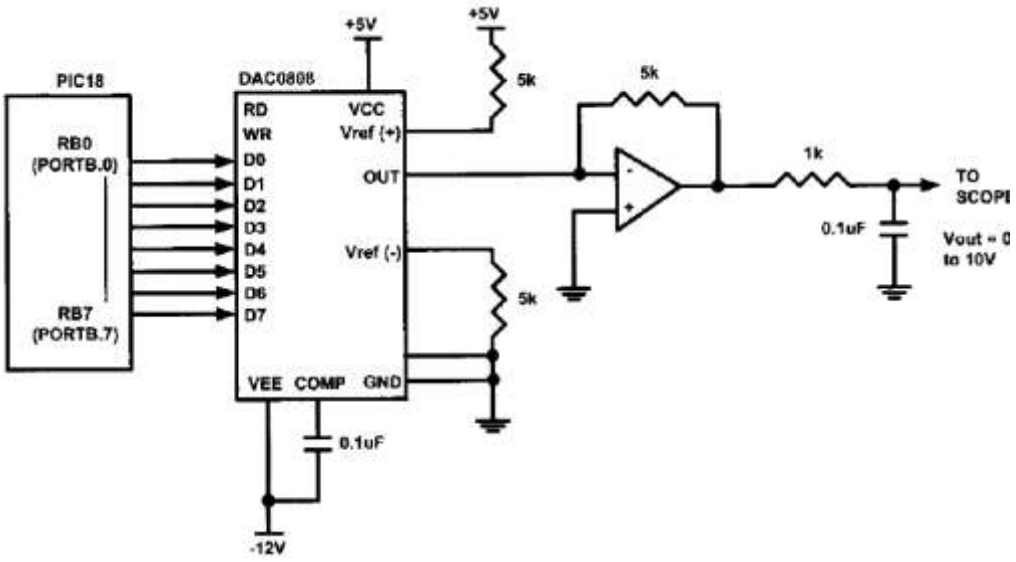
MC1408 DAC (or DAC0808)

- In the MC1408 (DAC0808), the digital inputs are converted to current (I_{out}), and by connecting a resistor to the lout pin, we convert the result to voltage.
- The total current provided by the lout pin is a function of the binary numbers at the DO-D7 inputs of the DAC0808 and the reference current I_{ref}), and is as follows:

$$I_{out} = I_{ref}\left(\frac{D_7}{2}+\frac{D_6}{4}+\frac{D_5}{8}+\frac{D_4}{16}+\frac{D_3}{32}+\frac{D_2}{64}+\frac{D_1}{128}+\frac{D_0}{256}\right)$$

where D0 is the LSB, D7 is the MSB for the inputs, and Iref is the input current that must be applied to pin 14. The Iref current is generally set to 2.0 mA.

- Figure shows the generation of current reference (setting ref = 2 mA) by using the standard 5 V power supply.
- Now assuming that Iref = 2 mA, if all the inputs to the DAC are high, the maximum output current is 1.99 mA



PIC18 Connection to DAC0808

Converting I_{out} to voltage in DAC0808

- Ideally we connect the output pin I_{out} to a resistor, convert this current to voltage, and monitor the output on the scope.
- In real life, however, this can cause inaccuracy because the input resistance of the load where it is connected will also affect the output voltage.
- For this reason, the I_{ref} current output is isolated by connecting it to an op-amp such as the 741 with R_f = 5 kOhms for the feedback resistor. Assuming that R = 5 kOhms, by changing the binary input, the output voltage changes.

27. Assuming that R = 5 kOhms and I_{ref} = 2 mA, calculate V_{out} for the following binary inputs: (a) 10011001 binary (99H) (b) 11001000 (C8H)

Solution:

(a) I_{out}= 2 mA (153/256) = 1.195 mA and Vout = 1.195 mA X 5K = 5.975 V

(b) l_{out} = 2 mA (200/256) = 1.562 mA and Vout = 1.562 mA X 5K = 7 .8125 V

28. In order to generate a stair-step ramp, set up the circuit in Figure and connect the output to an oscilloscope. Then write a program to send data to the DAC to generate a stair-step ramp.

Solution:

```
CLRF TRI SB          ; PORTB as output
CLRF PORTB           ; clear PORTB
```

```
AGAIN:  INCF PORTB,F      ;count from 0 to FFH send it to DAC
RCALL DELAY               ; let DAC recover
BRA AGAIN
```

- Table shows the angles, the sine values, the voltage magnitudes, and the integer values representing the voltage magnitude for each angle (with 30-degree increments).
- To generate Table 15-5, we assumed a full-scale voltage of 10 V for DAC output (as designed in Figure 15-11).
- Full-scale output of the DAC is achieved when all the data inputs of the DAC are HIGH. Therefore, to achieve the full-scale 10 V output, we use the following equation.

$$Vout = 5\ V + (5 \times \sin 0)$$

V out of DAC for various angles is calculated and shown in Table.

29. Verify the values given for the following angles: (a) 30° (b) 60°.

Solution:

(a) $Y_{out} = 5\ V + (5\ V \times \sin 0)$
$= 5\ V + 5 \times \sin 30°$
$= 5\ V + 5 \times 0.5$
$= 7.5\ V$
DAC input value = $7.5\ V \times 25.6$
$= 192$ (decimal)

(b) $Yout = 5\ V + (5\ V \times \sin 0)$
$= 5\ V + 5 \times \sin 60°$
$= 5\ V + 5 \times 0.866$
$= 9.33\ V$
DAC input value = $9.33\ V \times 25.6$
$= 238$ (decimal)

Angle versus voltage magnitude for sine wave

Angle Θ (degrees)	**Sin Θ**	**Vout (voltage magnitude) 5V + (5V X sinΘ)**	**Values sent to DAC (decimal) (voltage mag. X 25.6)**
0	0	5	128
30	0.5	7.5	192
60	0.866	9.33	238
90	1.0	10	255
120	0.866	9.33	238
150	0.5	7.5	192
180	0	5	128
210	-0.5	2.5	64
240	-0.866	0.669	17
270	-1.0	0	0
300	-0.866	0.669	17
330	-0.5	2.5	64
360	0	5	128

- To find the value sent to the DAC for various angles, we simply multiply the Vout voltage by 25.60 because there are 256 steps and full-scale Vout is 10 volts.
- Therefore, 256 steps I 10 V = 25.6 steps per volt.

Programming DAC in C:

```
//Program This is the C version of Program 15-3.
#include <p18F458.h>
ram canst unsigned char WAVEVALUE[12]  ={128,192,238,255,238,192,128,64,17,0,17,64};
void main {}
{
unsigned char x; TRISB=O;
while (l)
{
for (x~O;x<l2;x++}
PORTE=  WAVEVALUE(x);
}
}
```

30. Discuss Temperature sensors of PIC18

- Transducers convert physical data such as temperature, light intensity, flow, and speed to electrical signals.
- Depending on the transducer, the output produced is in the form of voltage, current, resistance, or capacitance.
- For example, temperature is converted to electrical signals using a transducer called a thermistor.
- A thermistor responds to temperature change by changing resistance, but its response to not linear.
- The complexity associated with writing software for such nonlinear devices has led many manufacturers to market a linear temperature sensor.
- Simple and widely used linear temperature sensors include the LM34 and LM35 series from National Semiconductor Corp.

Thermistor Resistance vs. Temperature

Temperature(c)	Tf(K ohms)
0	29.490
25	10.000
50	3.893
75	1.700
100	0.817

LM34 and LM35 temperature sensors

- The sensors of the LM34 series are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Fahrenheit temperature.
- The LM34 requires no external calibration because it is internally calibrated.
- It outputs 10 m V for each degree of Fahrenheit temperature. Table is a selection guide for the LM34.
- The LM35 series sensors are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Celsius (centigrade) temperature.

- The LM35 requires no external calibration because it is internally calibrated.
- It outputs 10mV for each degree of centigrade temperature.

LM34 Temperature Sensor Series Selection Guide

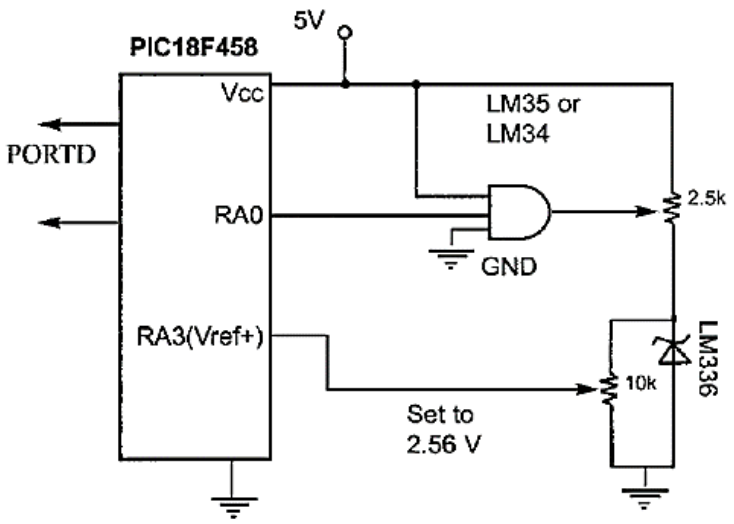
Part scale	Temperature Range	Accuracy	Output
LM34A	-50F TO +300F	+2.0F	10m V/F
LM34	-50F TO +300F	+3.0F	10m V/F
LM34CA	-40F TO +230F	+2.0F	10m V/F
LM34C	-40F TO +230F	+3.0F	10m V/F
LM34D	-32F TO +212F	+4.0F	10m V/F

LM35 Temperature Sensor Series Selection Guide

Part scale	Temperature Range	Accuracy	Output
LM35A	-55C TO +150C	+1.0C	10m V/C
LM35	-55C TO +150C	+1.5C	10m V/C
LM35CA	-40C TO +110C	+1.0C	10m V/C
LM35C	-40C TO +110C	+1.5C	10m V/C
LM35D	0C TO +100C	+2.0C	10m V/C

Signal conditioning and interfacing the LM35 to the PIC

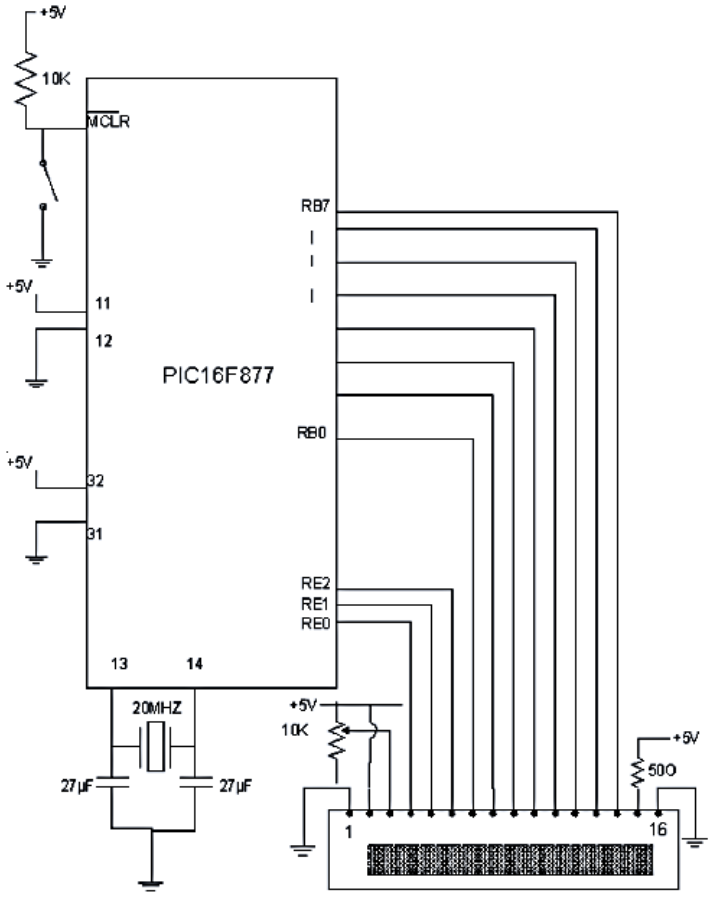
- Figure shows the connection of a temperature sensor to the PIC18F458. Notice that we use the LM336-2.5 zener diode to fix the voltage across the 1 OK pot at 2.5 volts.
- The use of the LM336-2.5 should overcome any fluctuations in the power supply



Reading and displaying temperature

31. Write PIC microcontroller assembly language program to display the characters '2018' in the first row of 2 lines x 20 characters LCD. *[Apr'18]*

Circuit Diagram:



Assembly language program:

```
        ORG 0X000           ; Reset address
        GOTO RESETFUN
        ORG 0X007
RESETFUN
        BSF STATUS,RP0      ; Now in BANK 1
        BCF STATUS,RP1
        BSF PCON, 0
        BSF PCON, 1
        GOTO START
        ORG 0X020
START
        BCF STATUS,RP0       ;Bank0
        CLRF PORTB          ; PORTB used as data pins
        CLRF PORTE          ; PORTE as control pins
        BSF STATUS,RP0      ;Bank1
MOVLW 0X00
MOVWF TRISB                 ; Initializing PORTB as output pins
MOVWF TRISE                 ; Initializing PORTE as output pins
```

```
MOVLW 0X86
MOVWF ADCON1
        NOP
        BCF STATUS,RP0          ;Bank0
MOVLW 0XFF
MOVWF PORTB
MOVLW 0XDC
        CALL LCDDELAY           ; Power up delay of 67msec
MOVLW 0X3F
MOVWF PORTB
        NOP
        CALL LCDENABL
MOVLW 0XA3                      ; 50msec delays between each enable
        CALL LCDDELAY
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
        NOP
MOVLW 0X3B                      ; Function Set
MOVWF PORTB
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
MOVLW 0XOC                      ; Display ON\OFF
MOVWF PORTS
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
MOVLW 0X01                      ; Display CLEAR
MOVWF PORTB
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
MOVLW 0X06                      ; Entry Mode Set
MOVWF PORTE
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
MOVLW 0X01
MOVWF PORTS                     ; Display clear
        NOP
```

```
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
MOVLW 0X80                  ; Set DDRAM address as Ox0OH
MOVWF PORTB
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY       ; Start writing data from here
        BSF PORTE, 0
MOVLW 0X07
        CALL LCDDELAY
MOVLW 0X5D                  ; 2
MOVWF PORTB
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
MOVLW 0X5B                  ; 0
MOVWF PORTS
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
MOVLW 0X5C                  ; 1
MOVWF PORTB
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
MOVLW 0X63                  ; 8
MOVWF PORTB
        NOP
        CALL LCDENABL
MOVLW 0XA3
        CALL LCDDELAY
OVER
        GOTO OVER           ; Infinite loop

LCDENABL
        BSF PORTE, 2
        NOP
        NOP
        BCF PORTE, 2
        MOVLW 0X07
        CALL LCDDELAY
        RETURN
LCDDELAY
        MOVWF 0X025
NEST1
```

```
        MOVLW 0XFF
        MOVWF 0X027
NEST2
        DECFSZ 0X027
        GOTO NEST2
```

ANNA UNIVERSITY QUESTIONS

PART - A

1. Difference between bus operation and subroutine operation. *[Nov'17]*
2. Draw the start and stop conditions of I^2C. *[Apr'18]*
3. Using PIC micro controller how is analog signal converted into digital. *(Jan'13)*
4. List the register associated with UART? *(Nov'16, Nov'17)*
5. Define baud rate. *[Apr'18]*
6. What is the value to be loaded into SPBRG register if we want 19200 baud rate with 10MHz clock source. *(Nov'16)*
7. How can the LCD be tested whether it is ready or not to receive a command or data? *(Jun'12)*
8. What is interfacing? *(Jan'14)*
9. While programming for LCD display, what initialization has to be done? *(Jan'13)*
10. What is the need for D/A converter? *(Apr'11)*
11. Microcontroller based control is advantageous than conventional control-Justify. *(Apr'17)*
12. How is temperature sensor interfaced with PIC microcontroller? *(Apr'17)*

PART - B

1. What is meant by I^2C module? Explain how I^2C is interfaced with PIC microcontroller. *[Nov'16]*
2. Exhibit the operation of I^2C bus and develop embedded C program to transmit data using I^2C bus. *[Nov'17]*
3. Explain briefly the concept of I^2C subroutines. Illustrate with suitable example how I^2 communication is carried out in PIC microcontroller. *(Apr'17)*
4. Explain the operation of ADC interfacing with PIC microcontroller. *(Nov'16, Apr'17, Nov'17)*
5. Draw and explain the architecture of on chip ADC of PIC microcontroller and write a suitable assembly language program for configuring the ADC. *[Apr'18]*
6. Write PIC microcontroller assembly language program to display the characters '2018' in the first row of 2 lines x 20 characters LCD. *[Apr'18]*

UNIT – IV PART A

1. What is ARM?

- An ARM processor is one of a family of CPUs based on the **RISC** (reduced instruction set computer) architecture developed by **Advanced RISC Machines** (ARM).
- ARM makes 32-bit and 64-bit **RISC** multi-core processors.

2. What are the features of the ARM architecture?

The main features of the ARM architecture are:

- A large set of registers, all of which can be used for most purposes.
- A load-store architecture.
- 3-address instructions (that is, the two source operand registers and the result register are all independently specified).
- Conditional execution of every instruction.

3. Why the ARM delayed branches were not used?

- The problem with delayed branches is that they remove the atomicity of individual instructions.
- They work well on single issue pipelined processors, but they do not scale well to super-scalar implementations and can interact badly with branch prediction mechanisms.
- On the original ARM delayed branches were not used because they made exception handling more complex; in the long run this has turned out to be a good decision since it simplifies re-implementing the architecture with a different pipeline.

4. Why the simplicity of the ARM may be more apparent?

- The simplicity of the ARM may be more apparent in the hardware organization and implementation than it is in the instruction set architecture.
- From the programmer's perspective it is perhaps more visible as a conservatism in the ARM instruction set design which, while accepting the fundamental precepts of the RISC approach, is less radical than many subsequent RISC designs.

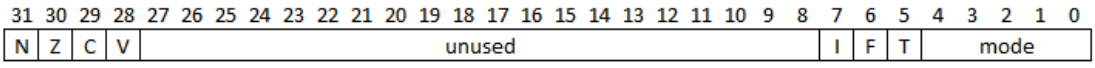
5. Define processor's instruction set

- A processor's instruction set defines the operations that the programmer can use to change the state of the system incorporating the processor.
- This state usually comprises the values of the data items in the processor's visible registers and the system's memory.
- Each instruction can be viewed as performing a defined transformation from the state before the instruction is executed to the state after it has completed.

6. What is CPSR?

- The Current Program Status Register (CPSR) is used in user-level programs to store the condition code bits.
- These bits are used, for example, to record the result of a comparison operation and to control whether or not a conditional branch is taken.

7. Write the CPSR format of ARM Processor. *[Apr'18]*



N: Negative; the last ALU operation which changed the flags produced a negative result (the top bit of the 32-bit result was a one).

Z: Zero; the last ALU operation which changed the flags produced a zero result (every bit of the 32-bit result was zero).
C: Carry; the last ALU operation which changed the flags generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.
V: oVerflow; the last arithmetic ALU operation which changed the flags generated an overflow into the sign bit.

8. What are the three types of ARM instructions?

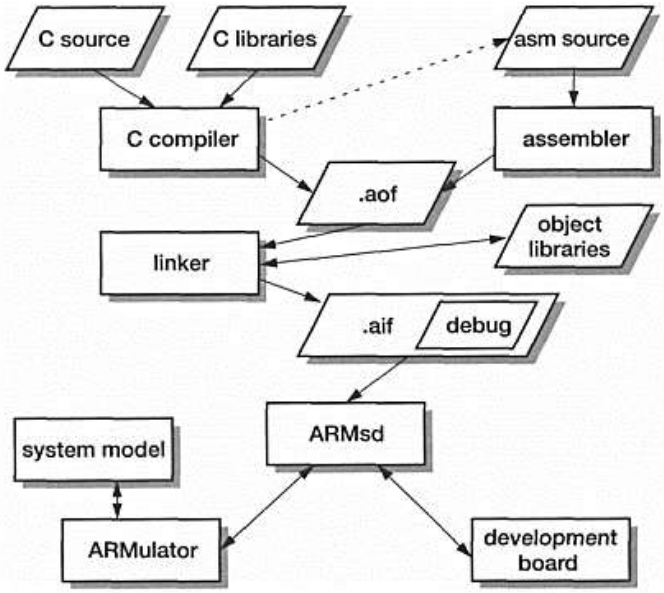
- Data processing instructions
- Data transfer instructions
- Control flow instructions

9. Explain most notable features of the ARM instruction set.

The most notable features of the ARM instruction set are:

- The load-store architecture
- 3-address data processing instructions (that is, the two source operand registers and the result register are all independently specified)
- Conditional execution of every instruction
- The inclusion of very powerful load and store multiple register instructions
- The ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle

10. What is the structure of the ARM cross-development toolkit?



11. Write various levels of accuracy the ARMulator can be operated.

It can operate at various levels of accuracy:

- **Instruction-accurate modelling** gives the exact behaviour of the system state without regard to the precise timing characteristics of the processor.
- **Cycle-accurate modelling** gives the exact behaviour of the processor on a cycle by-cycle basis, allowing the exact number of clock cycles that a program requires to be established.
- **Timing-accurate modelling** presents signals at the correct time within a cycle, allowing logic delays to be accounted for.

12. What are the On-chip RAM benefits?

In many embedded systems simple on-chip RAM is preferred to cache for a number of reasons:

- It is simpler, cheaper, and uses less power.
- Cache memory carries a significant overhead in terms of the logic that is required to enable it to operate effectively. It also incurs a significant design cost if a suitable off-the-shelf cache is unavailable.
- It has more deterministic behaviour.
- Cache memories have complex behaviours which can make difficult to predict how well they will operate under particular circumstances.

13. What are the function of MMU?

An MMU performs two primary functions:

- It translates virtual addresses into physical addresses.
- It controls memory access permissions, aborting illegal accesses.

14. What is the reason for including SWAP in the ARM instruction set?

- The ARM 'SWAP' instruction is just such an instruction which is included in the instruction set for exactly this purpose.
- A register is set to the 'busy' value, then this register is swapped with the memory location containing the Boolean. If the loaded value is 'free' the process can continue; if it is 'busy' the process must wait, often by **spinning** (repeating the test until it gets the 'free' result) on the lock.
- Note that this is the only reason for including SWAP in the ARM instruction set. It does not contribute to the processor's performance and its dynamic frequency of use is negligible. It is there just to provide this functionality.

15. Differentiate between RISC and CSIC.

- RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed.
- The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware.
- In contrast, the traditional complex instruction set computer (CISC) relies more on the hardware for instruction functionality, and consequently the CISC instructions are more complicated.

16. List the features of RISC architecture.

- A large uniform register file.
- A load/store architecture, where data-processing operations only operate on register contents, not directly on memory contents.
- Simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only.
- Uniform and fixed-length instruction fields, to simplify instruction decode.

17. What is 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.

18. Give the features of ARM processor.

- Load/store architecture.
- An orthogonal instruction set.
- Mostly single-cycle execution.
- Enhanced power-saving design.
- 64 and 32-bit execution states for scalable high performance.
- Hardware virtualization support.

19. What the components are of ARM 7 functional architecture?

- Instruction Pipeline and Read data register
- Instruction Decoder and control logic
- Address Register
- Address Incrementer
- Register Bank
- Booth's Multiplier
- Barrel shifters ALU
- ALU

20. What are the ARM programmer's models?

- ARM registers (Stack pointer, Link register, Program counter
- Current Program Status Register
- Processor Modes
- Barrel shifters

21. What are the different classes of ARM exception?

- Reset
- Attempted execution of an undefined instruction.
- Software interrupt (SWI) instructions, can be used to make a call to an operating system.
- Pre-fetch Abort, an instruction fetch memory abort.
- Data Abort, a data access memory abort.
- IRQ, normal interrupt FIQ fast interrupts.

22. What are the different classes of ARM instruction set?

The ARM instruction set can be divided into six broad classes of instruction:

- Branch instructions
- Data-processing instructions
- Status register transfer instructions
- Load and store instructions
- Coprocessor instructions
- Exception-generating instructions

23. What are the various ARM development tools? [OR] **List out some of ARM development tools.** *[Nov'16, Nov'17]*

- ARM ASSEMBLER
- The linker
- ARMsd
- ARMulator

24. List the various Memory hierarchy.

- Multiple types of memory
- Caches
- Write buffers
- Virtual memory and other memory remapping techniques.

25. What are the classifications of memory behaviour?

- Strongly ordered
- Device
- Normal.

These basic types can be further qualified by cacheable and shared attributes as well as access mechanisms.

26. What is memory hierarchy?

- Where a system is designed with different types of memory in a layered model, this is referred to as a memory hierarchy.
- Systems can employ caches at multiple levels.

27. What is L1 cache?

- The entries in the cache do not need to be cleaned and/or invalidated by software for different virtual to physical mappings
- Aliases to the same physical address may exist in memory regions.

28. What is L2 cache?

- L1 caches are always tightly coupled to the core, but L2 caches can be either:
- Tightly coupled to the core
- Implemented as memory mapped peripherals on the system bus.

29. What is the use of write buffers?

- The term write buffer can cover a number of different behaviours.
- The effects of these behaviours on different uses of memory mapped space needs to be understood by the programmer to avoid unexpected results.

30. What is Tightly Coupled Memory (TCM)?

- The Tightly Coupled Memory (TCM) is an area of memory that can be implemented alongside the L1 cache, as part of the level 1 memory subsystem.
- The TCM is physically addressed, with each bank occupying a unique part of the physical memory map.

31. Define addressing modes of ARM processor.

- There are different ways to specify the address of the operands for any given operations such as load, add or branch.
- The different ways of determining the address of the operands are called addressing modes.

32. List the different addressing modes used in the ARM processor.

- Register direct,
- Immediate Register indirect,
- Register indirect with offset,

- Register direct with offset,
- Register indirect pre-incrementing,
- Register indirect post-increment,
- Register indirect Register indexed,
- Register indirect indexed with scaling

33. List the various conditional flags used in ARM.

Flag	Flag name	Set when
Q	Saturation	The result causes an overflow and /or saturation
V	Overflow	The result causes a signed overflow
C	Carry	The result causes an unsigned carry
Z	Zero	The result is zero frequently used to indicate equality
N	Negative	Bit 31of the result is a binary 1

34. What is the purpose of program counter? *(Nov'16)*

- A program counter is a register in a computer processor that contains the address of the instruction being executed at the current time.
- As each instruction gets fetched, the program counter increased its stored value by 1.

35. Define Context Switching? *(Apr'17)*

- A **context switch** is the process of storing and restoring the state (more specifically, the execution context) of a process or thread so that execution can be resumed from the same point at a later time.
- This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system.

36. State the function of ARMulator and define its operations at various levels of accuracy? *(Apr'17)*

- Provides an environment for the development of ARM-targeted software on arange of non-ARM-based host systems.
- Allows benchmarking of ARM-targeted software (though its performance is somewhat slow compared to real hardware)
- Supports the simulation of prototype ARM-based systems, ahead of the availability of real hardware, so that software and hardware development canproceed in parallel.

It can operate at various levels of accuracy:

- **Instruction-accurate modelling** gives the exact behaviour of the system state without regard to the precise timing characteristics of the processor.
- **Cycle-accurate modelling** gives the exact behaviour of the processor on a cycle by-cycle basis, allowing the exact number of clock cycles that a program requires to be established.
- **Timing-accurate modelling** presents signals at the correct time within a cycle, allowing logic delays to be accounted for.

37. Differentiate little – endian and big – endian memory organizations. *[Apr'18]*

Little – endian mode:

The lowest order byte residing in the low – order bits of the word.

Big – endian mode:

The lowest – order byte stored in the highest bits of the word.

PART B

1. **With neat sketch explain the functional block diagram ARM architecture.***(Nov'16, Nov'17)*[OR]**Write short notes on ARM MMU architecture.***(Apr'17)*
   - The first ARM processor was developed at Acorn Computers Limited, of Cambridge, England, between October 1983 and April 1985.
   - At that time, and until the formation of Advanced RISC Machines Limited (which later was renamed simply ARM Limited) in 1990, ARM stood for Acorn **RISC Machine.**
   - An ARM processor is one of a family of CPUs based on the **RISC** (reduced instruction set computer) architecture developed by **Advanced RISC Machines** (ARM).
   - ARM makes 32-bit and 64-bit **RISC** multi-core processors.
   - ARM which licensees use to create micro controllers (MCUs) and CPUs based on those cores.
   - ARM7TDMI is the most successful implementation of ARM with hundreds of millions sold.

CORE:
- Two main blocks Data path and Decoder.
- Two read ports to register banks from A-Bus and B-bus and one write port from ALU.

Barrel Shifter:
- Shift/rotate 2nd operand by any number of bits

ALU:
- Perform arithmetic/logic functions

Address Register and Address incrementer:
- It holds either PC address or operand address.

Data register:
- It holds read/write data from/to memory

Instruction decoder:
- decodes machine code to control signals
- In single cycle, data values are read on bus A & B and the result from ALU is written to registers.

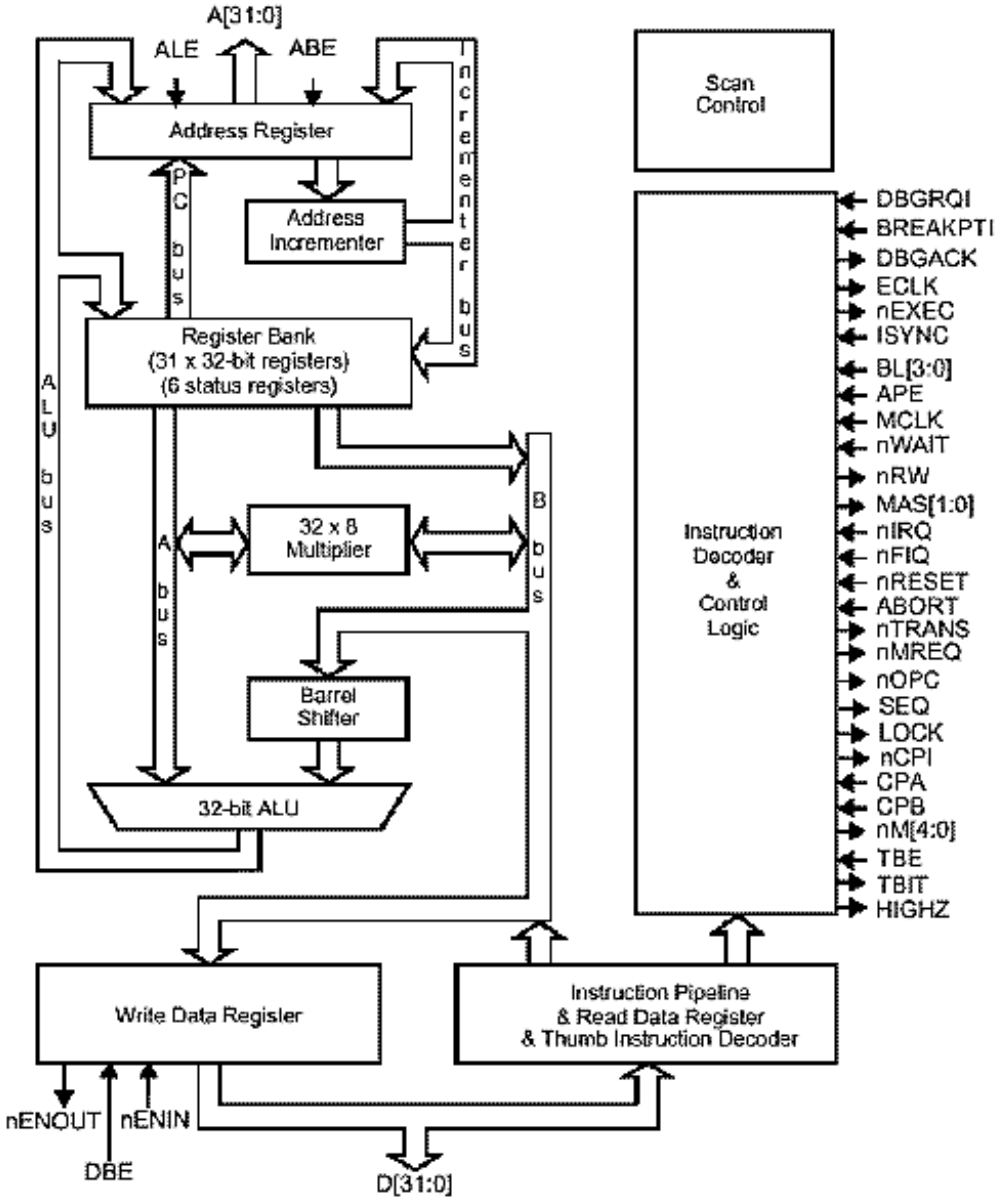
PIPELINING
- Usually ARM instructions are executed in 3 stages :
  - Fetch: fetch instruction from memory to pipeline.
  - Decode: decode the instruction to ARM
  - Execute : ALU result written to destination registers
- With latest processor adding two more stages as
  - Memory access
  - Write back.
- Portion of hardware which does fetching of instruction will be idle while decode and execute phase of instruction, this leaves the room for starting the next instruction's fetch before first instruction finishes decode or execute phase.

- So in optimized way, when first instruction is getting executed, second instruction can be decoded and a third instruction can be fetched. This is said to be pipelining.

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
Instruction 1	Fetch	Decode	Execute		
Instruction 2		Fetch	Decode	Execute	
Instruction 3			Fetch	Decode	Execute

REGISTERS

- ARM has 37, 32-bit long registers:
  - 30 – General purpose
  - 5 – SPSR (Saved process status register)
  - 1 – CPSR (Current process status register)
  - 1 – PC (program counter)



General purpose registers:

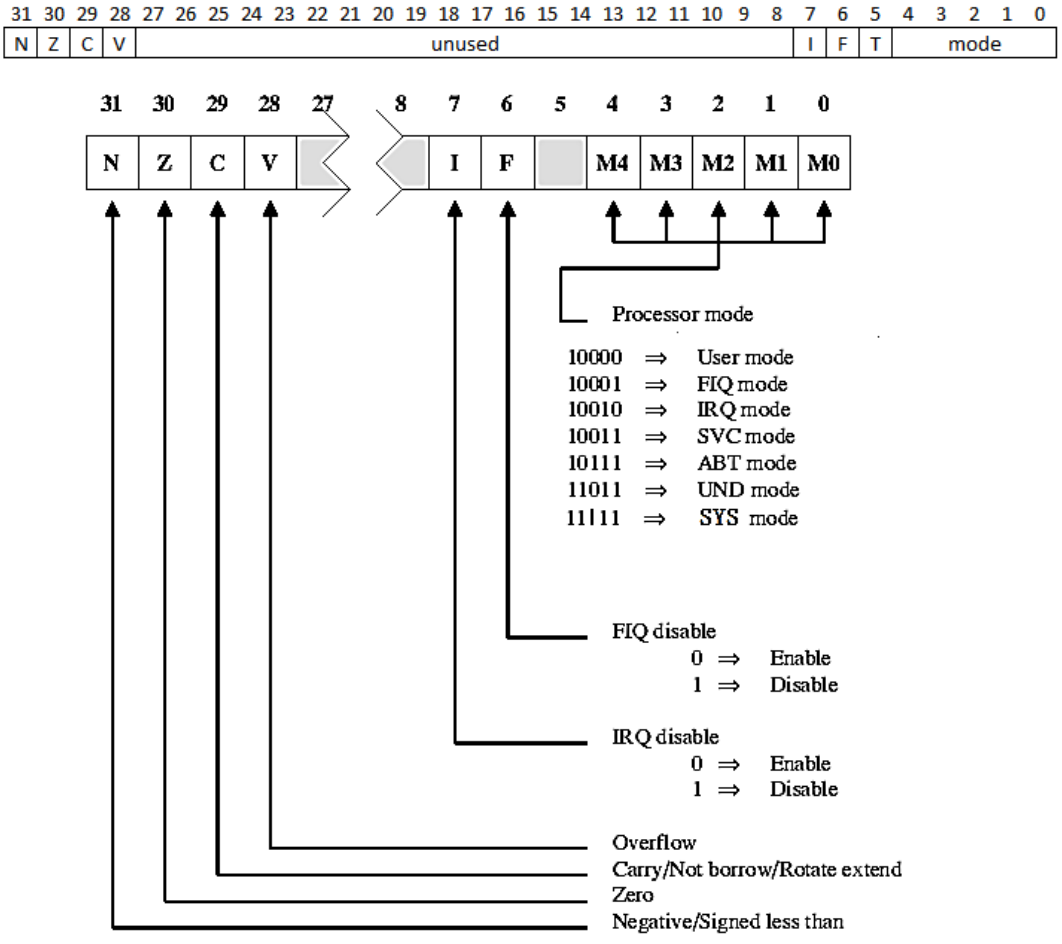
- 15 registers are visible at max in one mode (in USER mode) naming R0 to R14.
- R0 to R7 are unbanked registers (same physical address across all the modes)
- R8 to R14 are banked registers (separate copy of these registers in different mode if they exist).
- R13 is used as stack pointer commonly known as SP.
- R14 is used as link register to store the return address for exception/sub-routine. [If there are multiple nested levels, the previous return address goes to stack, pointed by R13, and the last address is kept in R14].

Program Counter:

- R15 is known as PC. PC contains the address of the instruction being executed at the current time.
- As each instruction gets fetched, the program counter increments by 4 bytes in ARM state and 2 bytes in THUMB state.
- Due to pipelining, current executing instruction is typically PC-8 for ARM and PC-4 for thumb.
- For ARM state bits 1 & 0 are always 0 or ignored.
- For THUMB state bit 0 is always 0 and ignored.

CPSR (Current process status register):

- CPSR holds the information of current process.



SPSR (Saved Process status Register):

- Used to store CPSR when an exception occurs, each exception mode has its separate SPSR.
- USER mode and SYSTEM mode doesn't have SPSR as they need not to execute exception handlers.

Thumb State:

- It is a subset of ARM state.
- In thumb state there is no access to R8 to R12.

EXCEPTIONS

- As the processor enters in to an exception mode, some registers are automatically switched depending on the type of mode.
- This ensure that task state is not corrupted by occurrence of exception.
- When an exception occurs ARM completes its current instruction, then :
  Step 1 : Saves the PC to LR (R14)
  Step 2 : Saves CPSR in new mode's SPSR
  Step 3 : Changes the mode corresponding to the exception
  Step 4 : Disable the exceptions of lower priority
  Step 5 : Load the new mode's instruction to PC (exception handler or ISR)

- A unique address is predefined for each exception handler, address to which processor is forced to branch is called exception/ interrupt vector.
- Once the exception is handled by the exception handler, mode is changed back to USER mode and the user task is resumed.
- Handler program must restore the user state exactly as it was before exception.
- Any modified register must be restored from the handler stack.
- CPSR must be restored from its SPSR.
- PC must be changed back to what it was executing, LR (R14) will help here.
- In case multiple exception occurs at same time, depending on their priority they will be serviced.

2. Explain ARM architectural inheritance with neat diagram.

- The main features of the ARM architecture are:
  - A large set of registers;
  - A load-store architecture;
  - 3-address instructions (that is, the two source operand registers and the result register are all independently specified);
  - Conditional execution of every instruction;
  - The inclusion of very powerful load and store multiple register instructions;
  - The ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle;
  - Open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model.
  - If the Thumb instruction set is considered part of the ARM architecture, we could also add:
  - 16-bit compressed representation of the instruction.

Architectural inheritance:

- At the time the first ARM chip was designed, the only examples of RISC architectures were the Berkeley RISC I and II and the Stanford MIPS (r **Microprocessor without Interlocking Pipeline** Stages)

Features used:

- The ARM architecture incorporated a number of features from the Berkeley RISC design.
    - A load-store architecture;
    - Fixed-length 32-bit instructions;
    - 3-address instruction formats.

Features rejected:

- The features that were employed on the Berkeley RISC designs which were rejected by the ARM designers were:

1. **Register windows**

- The register banks on the Berkeley RISC processors incorporated a large number registers, 32 of which were visible at any time.
- Procedure entry and exit instructions moved the visible 'window' to give each procedure access to new registers, thereby reducing the data traffic between the processor and memory resulting from register saving and restoring.
- The problem with register windows is the large chip area occupied by the large number of registers.
- This feature was therefore rejected on cost grounds, although the shadow registers used to handle exceptions on the ARM are not too different in concept.

2. **Delayed branches**

- Branches cause pipelines problems since they interrupt the smooth flow of instructions.
- Most RISC processors ameliorate the problem by using delayed branches where the branch takes effect after the following instruction has executed.
- The problem with delayed branches is that they remove the atomicity of individual instructions.
- They work well on single issue pipelined processors, but they do not scale well to super-scalar implementations and can interact badly with branch prediction mechanisms.
- On the original ARM delayed branches were not used because they made exception handling more complex; in the long run this has turned out to be a good decision since it simplifies re-implementing the architecture with a different pipeline.

3. **Single-cycle execution of all instructions:**

- Although the ARM executes most data processing instructions in a single clock cycle, many other instructions take multiple clock cycles.
- The rationale here was based on the observation that with a single memory for both data and instructions
- Even a simple load or store instruction requires at least two memory accesses (one for the instruction and one for the data).

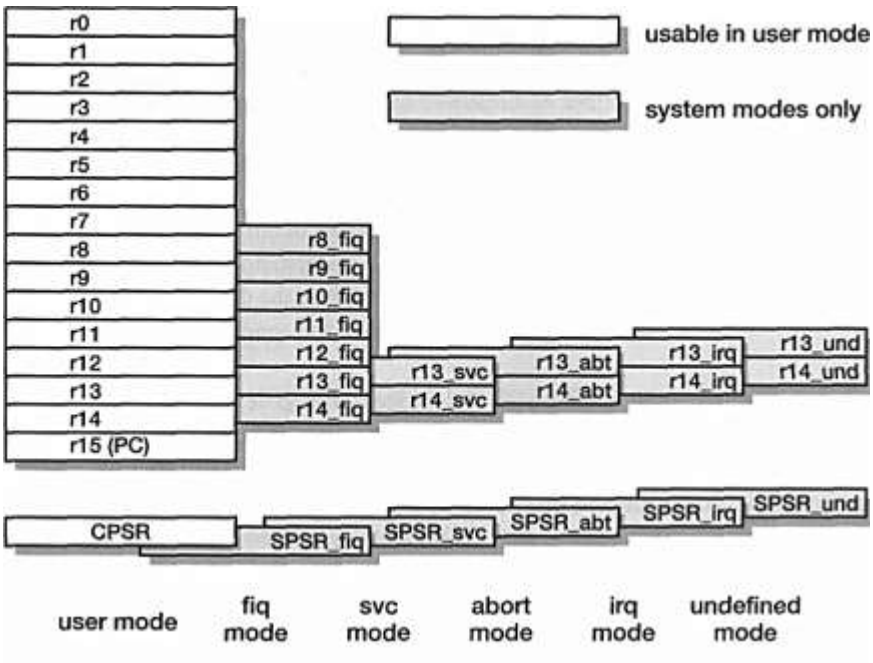
- Therefore single cycle operation of all instructions is only possible with separate data and instruction memories, which were considered too expensive for the intended ARM application areas.
- Instead of single-cycle execution of all instructions, the ARM was designed to use the minimum number of cycles required for memory accesses.
- This was greater than one, the extra cycles were used, to do support the auto-indexing addressing modes.
- This reduces the total number of ARM instructions required to perform any sequence of operations, improving performance and code density.

4. Simplicity:

- An overriding concern of the original ARM design team was the need to keep the design simple.
- Before the first ARM chips, Acorn designers had experience only of gate arrays with complexities up to around 2,000 gates, so the full-custom CMOS design medium was approached with some respect.
- The simplicity of the ARM may be more apparent in the hardware organization and implementation than it is in the instruction set architecture.
- From the programmer's perspective it is perhaps more visible as a conservation in the ARM instruction set design which, while accepting the fundamental precepts of the RISC approach, is less radical than many subsequent RISC designs.
- The combination of the simple hardware with an instruction set that is grounded in RISC ideas but retains a few key CISC features, and thereby achieves a significantly better code density than a pure RISC, has given the ARM its power-efficiency and its small core size.

3. Describe ARM programmer's model in detail. [OR] **Explain the various operating modes programmers model in ARM processor.** *[Nov'16]* [OR] **Explain the ARM programmer's model in detail, with supporting diagram.** *[Apr'17]* [OR] **Draw and explain the visible registers in an ARM processor.** *[Apr'18]*

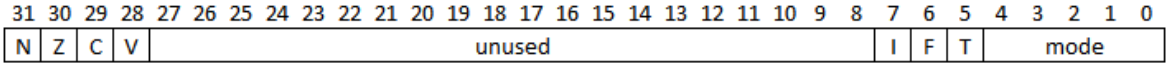
- A processor's instruction set defines the operations that the programmer can use to change the state of the system incorporating the processor.
- This state usually comprises the values of the data items in the processor's visible registers and the system's memory.
- Each instruction can be viewed as performing a defined transformation from the state before the instruction is executed to the state after it has completed.
- The visible registers in an ARM processor are shown in Figure.
- When writing user-level programs, only the 15 general-purpose 32-bit registers (r0 to r14), the program counter (r15) and the current program status register (CPSR) need be considered.
- The remaining registers are used only for system-level programming and for handling exceptions (for example, interrupts).



ARM's visible registers

The Current Program Status Register (CPSR)

- The CPSR is used in user-level programs to store the condition code bits. These bits are used, to record the result of a comparison operation and to control whether or not a conditional branch is taken.
- The user-level programmer need not usually be concerned with how this register is configured, but for completeness the register is illustrated in figure below.
- The bits at the bottom of the register control the processor mode, instruction set and interrupt enables and are protected from change by the user-level program.



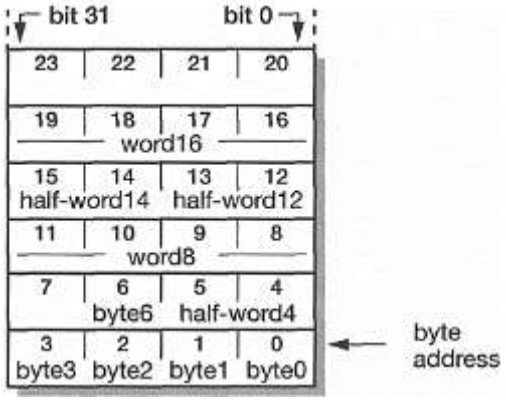
ARM CPSR format

- N: Negative; the last ALU operation which changed the flags produced a negative result (the top bit of the 32-bit result was a one).
- Z: Zero; the last ALU operation which changed the flags produced a zero result (every bit of the 32-bit result was zero).
- C: Carry; the last ALU operation which changed the flags generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.
- V: Overflow; the last arithmetic ALU operation which changed the flags generated an overflow into the sign bit.

The memory system

- In addition to the processor register state, an ARM system has memory state.
- Memory may be viewed as a linear array of bytes numbered from zero up to 23.
- Data items may be 8-bit bytes, 16-bit half-words or 32-bit words.
- Words are always aligned on 4-byte boundaries (that is, the two least significant address bits are zero) and half-words are aligned on even byte boundaries.

- The memory organization is illustrated in figure below. This shows a small area of memory where each byte location has a unique number.
- A byte may occupy any of these locations.
- A word-sized data item must occupy a group of four byte locations starting at a byte address which is a multiple of four.
- Half-words occupy two byte locations starting at an even byte address.



ARM memory organization.

Load-store architecture

- In common with most RISC processors, ARM employs a load-store architecture.
- This means that the instruction set will only process (add, subtract, and so on) values which are in registers (or specified directly within the instruction itself), and will always place the results of such processing into a register.
- The only operations which apply to memory state are ones which copy memory values into registers(load instructions) or copy register values into memory (store instructions).
- *CISC processors typically allow a value from memory to be added to a value in a register, and sometimes allow a value in a register to be added to a value in memory.*
- ARM does not support such 'memory-to-memory' operations.

- Therefore all ARM instructions fall into one of the following three categories:
  - **Data processing instructions**.
    - These use and change only register values.
    - For example, an instruction can add two registers and place the result in a register.
  - **Data transfer instructions**.
    - These copy memory values into registers (load instructions) or copy register values into memory (store instructions).
    - An additional form, useful only in systems code, exchanges a memory value with a register value.
  - **Control flow instructions**.
    - Control flow instructions cause execution to switch to a different address, either permanently (branch instructions) or saving a return address to resume the original sequence (branch and link instructions) or trapping into system code (supervisor calls).

Supervisor mode

- The ARM processor supports a protected supervisor mode.
- The protection mechanism ensures that user code cannot gain supervisor privileges without appropriate checks being carried
- Out to ensure that the code is not attempting illegal operations.
- The upshot of this for the user-level programmer is that system-level functions can only be accessed through specified supervisor calls.
- These functions generally include any accesses to hardware peripheral registers, and to widely used operations such as character input and output.
- User-level programmers are concerned with devising algorithms to operate on the data 'owned' by their programs, and rely on the operating system to handle all transactions with the world outside their programs.
- The instructions which request operating system functions are covered in 'Supervisor calls'

ARM instruction set

- All ARM instructions are 32 bits wide and are aligned on 4-byte boundaries in memory.
- The features of the ARM instruction sets are:
    - The load-store architecture;
    - 3-address data processing instructions (the two source operand registers and the result register are all independently specified);
    - Conditional execution of every instruction;
    - The inclusion of very powerful load and store multiple register instructions;
    - The ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle;
    - Open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model;
    - A very dense 16-bit compressed representation of the instruction set in the Thumb architecture.
- For the small embedded systems that most ARM processors are used in, this code density advantage outweighs the small performance penalty incurred by the decode complexity.
- Thumb code extends this advantage to give ARM better code density than most CISC processors.

The I/O system

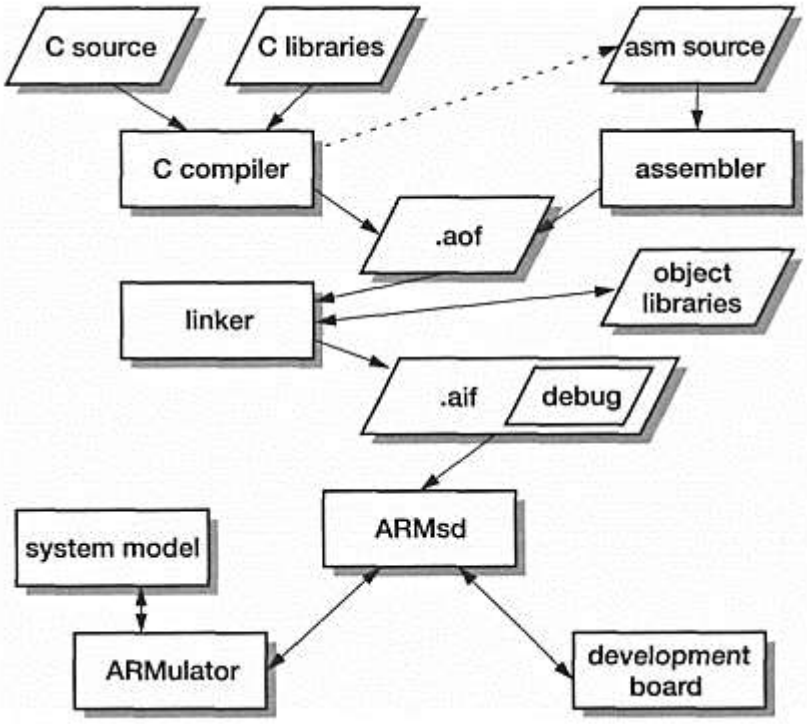
- The ARM handles I/O (input/output) peripherals (such as disk controllers, network interfaces, and so on) as memory-mapped devices with interrupt support.
- The internal registers in these devices appear as addressable locations within the ARM's memory map and may be read and written using the same (load-store) instructions as any other memory locations.
- Peripherals may attract the processor's attention by making an interrupt request using either the normal interrupt (IRQ) or the fast interrupt (FIQ) input.
- Both interrupt inputs are level-sensitive and mask able.
- Normally most interrupt sources share the IRQ input, with just one or two time-critical sources connected to the higher-priority FIQ input.
- Some systems may include direct memory access (DMA) hardware external to the processor to handle high-bandwidth I/O traffic.

ARM exceptions

- The ARM architecture supports a range of interrupts, traps and supervisor calls, all grouped under the general heading of exceptions.
  1. The current state is saved by copying the PC into *rl4_exc* and the CPSR into SPSR_exc (where *exc* stands for the exception type).
  2. The processor operating mode is changed to the appropriate exception mode.
  3. The PC is forced to a value between 0016 and 1C16, the particular value depending on the type of exception.
- The instruction at the location the PC is forced to (the *vector address)* will usually contain a branch to the exception handler.
- The exception handler will use rl3_exc, which will normally have been initialized to point to dedicated stack in memory.
- The return to the user program is achieved by restoring the user registers and then using an instruction to restore the PC and the CPSR automically.
- This may involve some adjustment of the PC value saved in *rl4_exc* to compensate for the state of the pipeline when the exception arose.

4. Explain the structure of the ARM cross – development tool kit. *[Apr'18]*

- Since the ARM is widely used as an embedded controller where the target hardware will not make a good environment for software development, the tools are intended for **cross-development** from a platform such as a PC running Windows or a suitable UNIX workstation.
- The overall structure of the ARM cross-development toolkit is shown in figure below.



Structure of the ARM cross-development toolkit

- C or assembler source files are compiled or assembled into ARM object format (.aof) files, which are then linked into ARM image format (.aif) files.
- The image format files can be built to include the debug tables required by the ARM symbolic debugger (ARMsd which can load, run and debug programs either on hardware

such as the ARM Development Board or using a software emulation of the ARM (the ARMulator).
- The ARMulator has been designed to allow easy extension of the software model to include system features such as caches, particular memory timing characteristics, etc.

ARM C compiler
- The ARM C compiler is compliant with the ANSI (American National Standards Institute) standard for C and is supported by the appropriate library of standard functions.
- It uses the ARM Procedure Call Standard for all externally available functions.
- It can produce assembly source output instead of ARM object format, so the code can be inspected or even hand optimized, and then assembled subsequently.
- The compiler can also produce Thumb code.

ARM assembler
- The ARM assembler is a full macro assembler which produces ARM object format output that can be linked with output from the C compiler.

Linker
- The linker takes one or more object files and combines them into an executable program.
- It resolves symbolic references between the object files and extracts object modules from libraries as needed by the program.
- It can assemble the various components of the program in a number of different ways, depending on whether the code is to run in RAM or ROM.
- Normally the linker includes debug tables in the output file.
- If the object files were compiled with full debug information, this will include full symbolic debug.
- The linker can also produce object library modules that are not executable but are ready for efficient linking with object files in the future

ARMsd
- The ARM symbolic debugger is a front-end interface to assist in debugging programs running either under emulation (on the ARMulator) or remotely on a target system such as the ARM development board.
- The remote system must support the appropriate remote debug protocols either via a serial line or through a JTAG test interface.
- Debugging a system where the processor core is embedded within an application-specific system chip is a complex issue.
- At its most basic, ARMsd allows an executable program to be loaded into the ARMulator or a development board and run.

ARMulator
- The ARMulator (ARM emulator) is a suite of programs that models the behaviour of various ARM processor cores in software on a host system.
- It can operate at various levels of accuracy:
  - **Instruction-accurate modelling** gives the exact behaviour of the system state without regard to the precise timing characteristics of the processor.
  - **Cycle-accurate modelling** gives the exact behaviour of the processor on a cycle by-cycle basis, allowing the exact number of clock cycles that a program requires to be established.

- **Timing-accurate modelling** presents signals at the correct time within a cycle, allowing logic delays to be accounted for.
- It allows the number of clock cycles the program using compiler C takes to execute to be measured exactly, so the performance of the target system can be evaluated.
- At its most complex, the ARMulator can be used as the centre of a complete, timing-accurate, C model of the target system, with full details of the cache and memory management functions added, running an operating system.
- The ARMulator comes with a set of model prototyping modules including a rapid prototype memory model and coprocessor interfacing support.
- The ARMulator can also be used as the core of a timing-accurate ARM behavioural model in a hardware simulation environment based around a language such as VHDL.

ARM development board

- The ARM Development Board is a circuit board incorporating a range of components and interfaces to support the development of ARM-based systems.
- It can support both hardware and software development before the final application-specific hardware is available.
- It includes an ARM core (ARM7TDMI), memory components which can be configured to match the performance and bus-width of the memory in the target system, and electrically programmable devices which can be configured to emulate application-specific peripherals.

Software Toolkit

- The Toolkit CD-ROM includes a PC version of the toolset that runs under most versions of the Windows operating system and includes a full Windows-based project manager.
- The ARM Project Manager is a graphical front-end for the tools described above.
- It supports the building of a single library or executable image from a list of files that make up a particular project.
- These files may be:
    - source files (C, assembler, and so on);
    - object files;
    - library files.
- The source files may be edited within the Project Manager, a dependency list created and the output library or executable image built.
- There are many options which may be chosen for the build, such as:
    - Whether the output should be optimized for code size or execution time.
    - Whether the output should be in debug or release form.
    - Which ARM processor is the target
- The CD-ROM also contains versions of the tools that run on a Sun or HP UNIX host, where a command-line interface is used.

JumpStart

- The JumpStart tools from VLSI Technology, Inc., include the same basic set of development tools but present a full X-windows interface on a suitable workstation rather than the command-line interface of the standard ARM toolkit.

5. Explain briefly about Memory Hierarchy in ARM.

Memory size and speed

- A typical computer memory hierarchy comprises several levels, each level having a characteristic size and speed.
- The processor registers can be viewed as the top of the memory hierarchy.

Memory	Speed
A RISC processor (128 bytes)	Few nanoseconds
On-chip cache (8 to 32 Kbytes)	10 nanoseconds
Desktop systems (Few hundred Kbytes)	Few 10 of nanoseconds
Main memory – RAM (Megabytes to tens of megabytes)	100 nanoseconds
Backup store - hard disk (Hundreds of Mbytes up to a few Gbytes)	Few tens of milliseconds

- The data which is held in the registers is under the direct control of the compiler or assembler programmer, but the contents of the remaining levels of the hierarchy are usually managed automatically.
- The caches are effectively invisible to the application program, with blocks or 'pages' of instructions and data migrating up and down the hierarchy under hardware control.
- Paging between the main memory and the backup store is controlled by the operating system, and remains transparent to the application program.
- Since the performance difference between the main memory and the backup store is so great, much more sophisticated algorithms are required here to determine when to migrate data between the levels.
- An embedded system will not usually have a backing store and will therefore not exploit paging. However, many embedded systems incorporate caches, and ARM CPU chips employ a range of cache organizations.

Memory cost:

- Fast memory is more expensive per bit than slow memory.

On-chip memory

- Some form of on-chip memory is essential if a microprocessor is to deliver its best performance.
- On-chip memory can support zero wait state access speeds, and it will also give better power-efficiency and reduced electromagnetic interference than off-chip memory

On-chip RAM benefits

- It is simpler, cheaper, and uses less power.
- Cache memory carries a significant overhead in terms of the logic that is required to enable it to operate effectively.
- It also incurs a significant design cost if a suitable off-the-shelf cache is unavailable.
- It has more deterministic behaviour.
- Cache memories have complex behaviours which can make difficult to predict how well they will operate under particular circumstances.
- In particular, it can be hard to guarantee interrupt response time.

Drawback

- With on-chip RAM *vis-d-vis*cache is that it requires explicit management by the programmer, whereas a cache is usually transparent to the programmer.

Application

- A cache is usually preferred in any general-purpose system where the application mix is unknown.

Advantage

- On-chip RAM is that it enables the programmer to allocate space in it using knowledge of the future processing load.
- A cache left to its own devices has knowledge only of past program behaviour, and it can therefore never prepare in advance for critical future tasks.

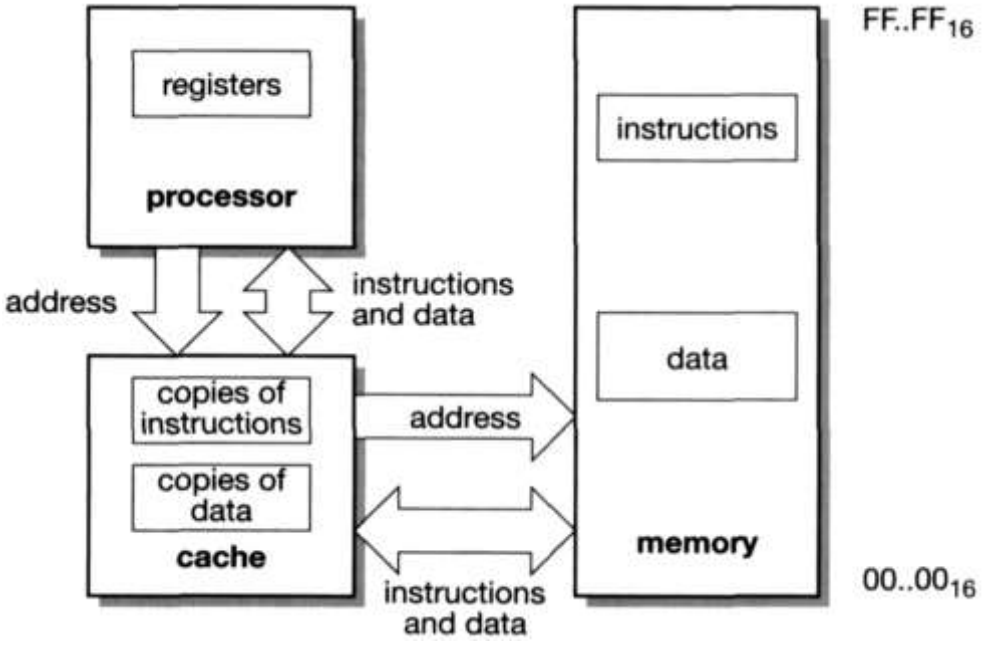
6. Explain in detail about Caches

Processor and memory speeds:

- A cache memory is a small, very fast memory that retains copies of recently used memory values.
- It operates transparently to the programmer, automatically deciding which values to keep and which to overwrite.
- If the processor is so much faster than the memory, it can only deliver its full performance potential with the help of a **cache** memory.
- Caches work because programs normally display the property of **locality,** which means that at any particular time they tend to execute the same instructions many times on the same areas of data.

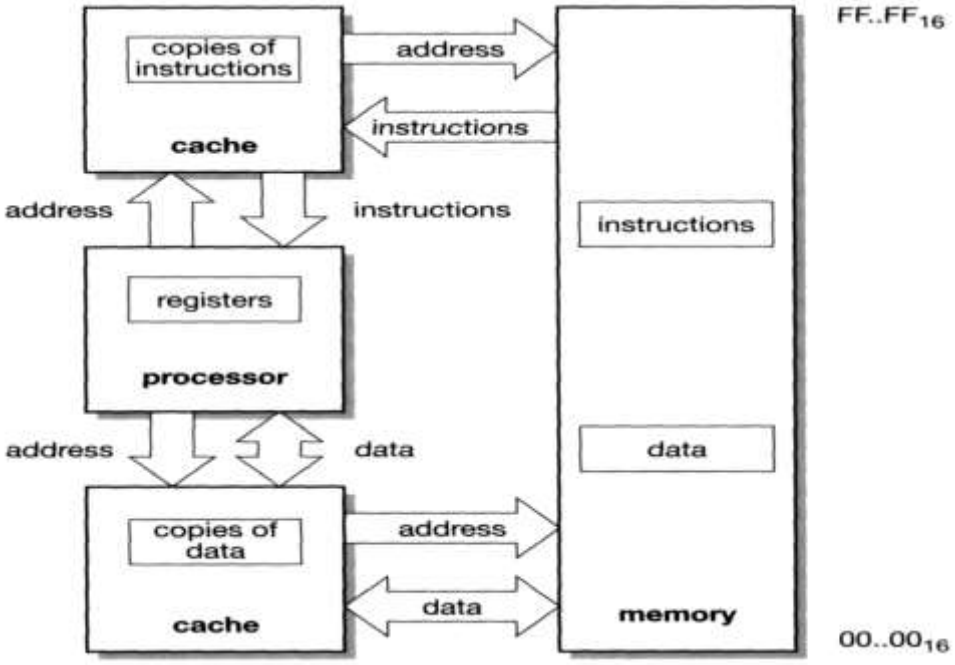
Unified and Harvard caches

- Organizations:
  - A unified cache.
    - The unified cache automatically adjusts the proportion of the cache memory used by instructions according to the current program requirements, giving a better performance than a fixed partitioning.



A unified instruction and data cache

- Separate instruction and data caches.
    - Separate caches allow load and store instructions to execute in a single clock cycle.



Separate data and instruction caches

Cache performance metrics

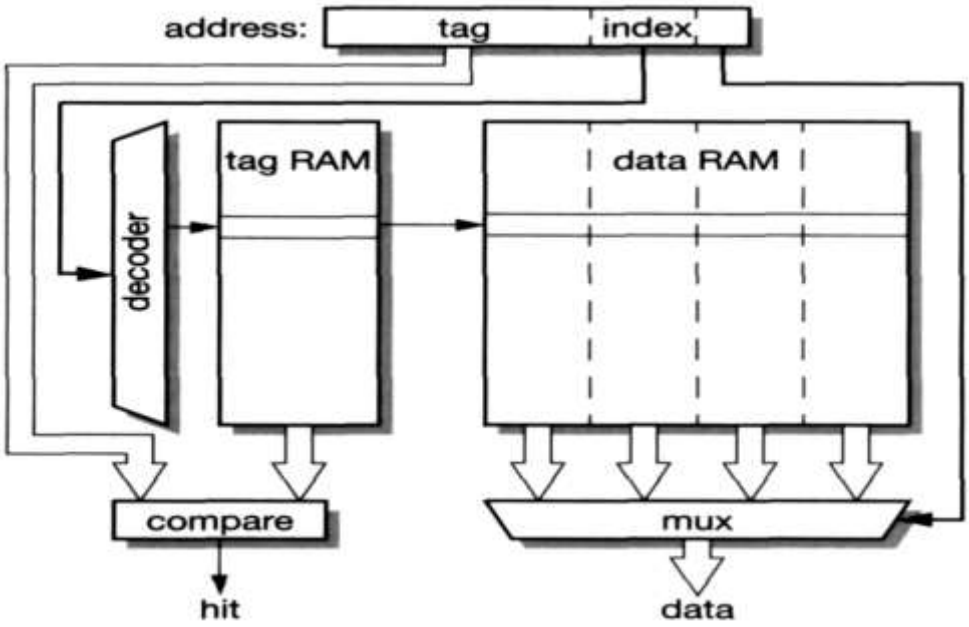
- Since the processor can operate at its high clock rate only when the memory items it requires are held in the cache.
- The overall system performance depends strongly on the proportion of memory accesses which cannot be satisfied by the cache.
- An access to an item which is in the cache is called a hit.
- An access to an item which is not in the cache is a **miss.**
- The proportion of all the memory accesses that are satisfied by the cache is the **hit rate (%).**
- The proportion that are not is the **miss rate.**
- The miss rate depends on a number of cache parameters, including its size (the number of bytes of memory in the cache) and its organization.

Cache organization

- Since a cache holds a dynamically varying selection of items, it must have storage for both the data and the address at which the data is stored in main memory.

The direct-mapped cache:

- In the direct-mapped cache a **line** of data is stored along with an address **tag** in a memory addressed by some portion of the memory address (the **index).**

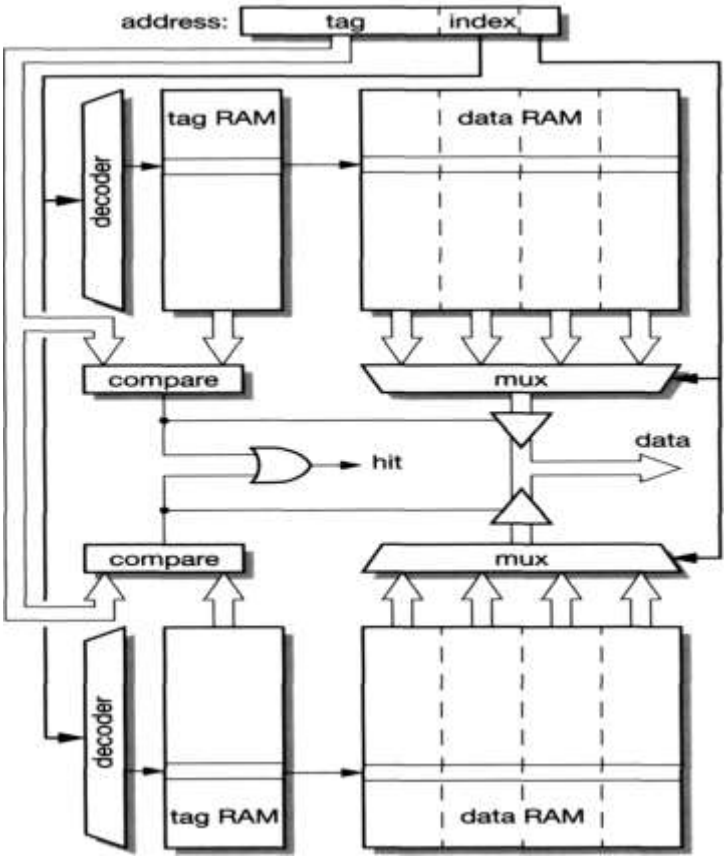


Direct-mapped cache organization

- To check whether or not a particular memory item is stored in the cache, the index address bits are used to access the cache entry.
- The top address bits are then compared with the stored tag; if they are equal, the item is in the cache.
- The lowest address bits can be used to access the desired item within the line.
- Properties:
    - A particular memory item is stored in a unique location in the cache [two items with the same cache address field will contend for use of that location].
    - Only those bits of the address that are not used to select within the line or to address the cache, RAM need be stored in the tag field.
    - The tag and data access can be performed at the same time.
    - Since the tag RAM is typically a lot smaller than the data RAM, its access time is shorter, allowing the tag comparison to be completed within the data access time.
- A typical direct-mapped cache might store 8 Kbytes of data in 16-byte lines [512 lines].
- A 32-bit address would have four bits to address bytes within the line and nine bits to select the line, leaving a 19-bit tag which requires just over one Kbyte of tag store.
- When data is loaded into the cache, a **block** of data is fetched from memory.
- If the block size is smaller than the line size, the tag store must be extended to include a valid bit for each block within the line.
- Choosing the line and block sizes to be equal results in the simplest organization.

Set-associative cache:

- The set-associative cache aims to reduce the problems of complexity due to contention by enabling a particular memory item to be stored in more than one cache location.
- A 2-way set-associative cache is illustrated in Figure.



Two-way set-associative cache organization

- As the figure suggests, this form of cache is effectively two direct-mapped caches operating in parallel.
- An address presented to the cache may find its data in either half, so each memory address may be stored in either of two places.
- Each of two items which were in contention for a single location in the direct-mapped cache may now occupy one of these places, allowing the cache to hit on both.
- The 8 Kbyte cache with 16 byte lines will have 256 lines in each half of the cache, so four bits of the 32-bit address select a byte from the line and eight bits select one line from each half of the cache.
- The address tag must therefore be one bit longer, at 20 bits.
- The access time is only very slightly longer than that of the direct-mapped cache, the increase being due to the need to multiplex the data from the two halves.
- When a new data item is to be placed in the cache, a decision must be taken as to which half to place it in.
- There are several options here, the most common being:

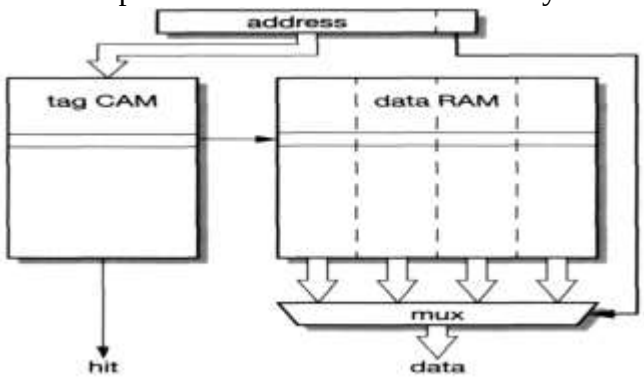
Random allocation

- The decision is based on a random or pseudo-random value.
    - Least recently used (LRU).
- The cache keeps a record of which location of a pair was last accessed and allocates the new data to the other one.
    - Round-robin (also known as 'cyclic').
- The cache keeps a record of which location of a pair was last allocated and allocates the new data to the other one.

- The set-associative approach extends beyond 2-way up to any degree of associativity, but in practice the benefits of going beyond 4-way associatively are small and do not warrant the extra complexity incurred.

Fully associative cache:

- Rather than continuing to divide the direct-mapped cache into ever smaller components, the tag store is designed differently using content addressed memory (CAM) using VLSI technology.
- A CAM cell is a RAM cell with an inbuilt comparator, so a CAM based tag store can perform a parallel search to locate an address in any location.
- Since there are no address bits implicit in the position of data in the cache, the tag must store all the address bits apart from those used to address bytes within the line.



Fully associative cache organization

Write strategies

- The above schemes operate in an obvious way for read accesses:
    - When presented with a new read address the cache checks to see whether it holds the addressed data; if it does, it supplies the data; if it does not, it fetches a block of data from main memory, stores it in the cache in some suitable location and supplies the requested data to the processor.
- There are more choices to make when the processor executes a write cycle.
- **Write-through**
    - All write operations are passed to main memory; if the addressed location is currently held in the cache, the cache is updated to hold the new value.
    - The processor must slow down to main memory speed while the write takes place.

- **Write-through with buffered write.**
    - Here all write operations are still passed to main memory and the cache updated as appropriate, but instead of slowing the processor down to main memory speed the write address and data are stored in a **write buffer** which can accept the write information at high speed.

- **Copy-back** (also known as **write-back).**
    - A copy-back cache is not kept coherent with main memory.
    - Write operations update only the cache, so cache lines must remember when they have been modified (usually using a **dirty** bit on each line or block).

- If a dirty cache line is allocated to new data it must be copied back to memory before the line is reused.

7. Explain in detail about Cache design with an example.

Choice on the performance of the cache:

- The size of the cache,
- The degree of associativity,
- The line and block sizes,
- The replacement algorithm,
- The write strategy.

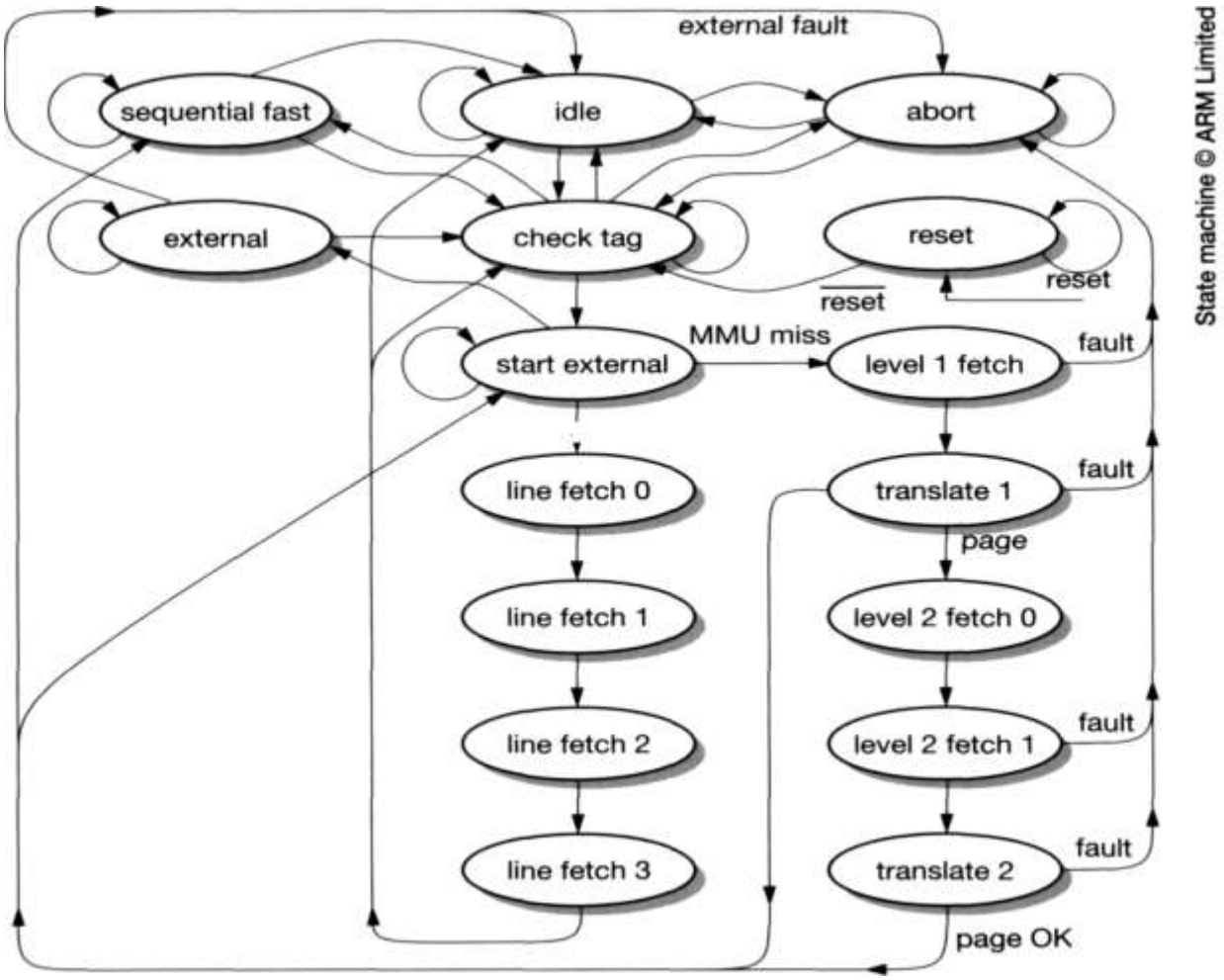
- Detailed architectural simulations are required to analyse the effects of these choices on the performance of the cache

Arms Cache

- These studies used specially designed hardware to capture address traces while running several benchmark programs on an ARM2;
- Software was then used to analyse these traces to model the behaviour of the various organizations.
- A 'perfect' cache, which always contains the requested data, was modelled to set this bound.
- Any real cache is bound to miss some of the time, so it cannot perform any better than one which always hits.
- Three forms of perfect cache were modelled using realistic assumptions about the cache and external memory speeds (which were 20 MHz and 8 MHz respectively):
  - caches which hold either just instructions,
  - mixed instructions and data,
  - Just data.
- Instructions are the most important values to hold in the cache, but including data values as well can give a further 25% performance increase.

8. Explain about Memory management

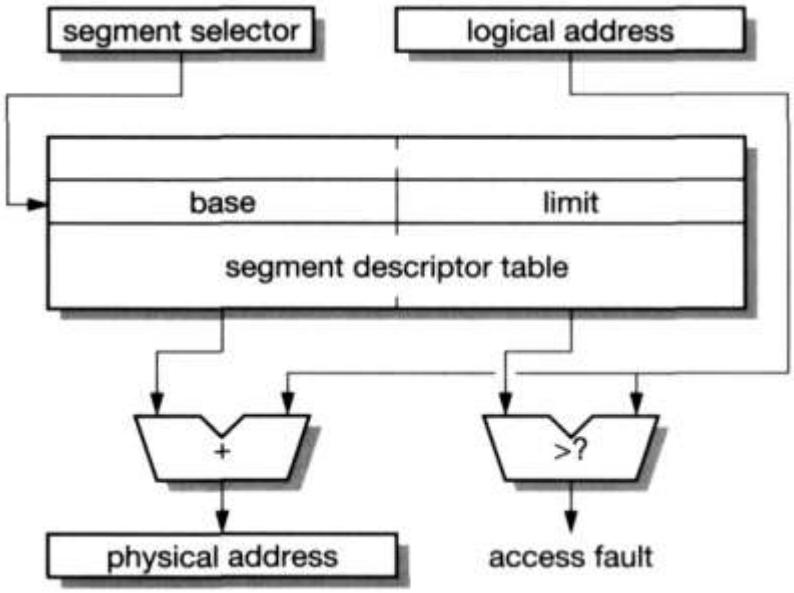
- Modern computer systems typically have many programs active at the same time.
- A single processor can, only execute instructions from one program at any instant, but by switching rapidly between the active programs they all appear to be executing at once, at least when viewed on a human timescale.
- The rapid switching is managed by the operating system,
- So the application programmer can write his or her program as though it owns the whole machine.
- The mechanism used to support this illusion is described by the term **memory management unit** (MMU).
- There are two principal approaches to memory management, called
  - **segmentation and**
  - **paging**



ARM600 cache control state machine

Segments

- The simplest form of memory management allows an application to view its memory as a set of segments, [each segment contains a particular sort of information].
- For instance, a program may have a code segment containing all its instructions, a data segment and a stack segment.
- Every memory access provides a segment selector and a logical address to the MMU.
- Each segment has a base address and a limit associated with it.
- The logical address is an offset from the segment base address and must be not greater than the limit or else an access violation will occur [causing an exception].
- Segments may also have other access controls, [code segment may be read-only and if attempt to write to it will also cause an exception].

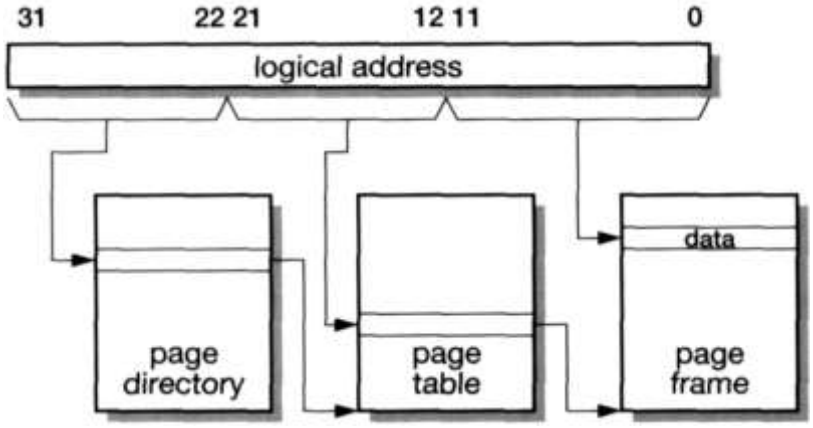


Segmented memory management scheme

- Segmentation allows a program to have its own private view of memory and to coexist transparently with other programs in the same memory space.
- Since the segments are of variable size, the free memory becomes fragmented over time and a new program may be unable to start, because the free memory is all in small pieces.
- The crisis can be alleviated by the operating system moving segments around in memory to join the free memory into one large piece, but this is inefficient, and most processors now incorporate a memory mapping scheme based on fixed-size chunks of memory called **pages.**
- Some architectures include segmentation and paging, but many, including the ARM, just support paging without segmentation

Paging:

- In a paging memory management scheme both the logical and the physical address spaces are divided into fixed-size components called pages.
- Usually few kilobytes in size, but different architectures use different page sizes.
- The relationship between the logical and physical pages is stored in **page tables,** which are held in main memory.
- Most paging systems use two or more levels of page table.



Paging memory management scheme

- The minimum overhead for a small system is 4 Kbytes for the page directory plus 4 Kbytes for one page table; this is sufficient to manage up to 4 Mbytes of physical memory.
- A fully populated32 gigabyte memory would require 4 Mbytes of page tables, but this overhead is probably acceptable with this much memory to work in.

Virtual memory:

- One possibility with either memory management scheme is to allow a segment or page to be marked as absent and an exception to be generated whenever it is accessed.
- Operating system which has run out of memory to allocate can transparently move a page or a segment out of main memory into backup store, a hard disk, and mark it as absent.
- The physical memory can then be allocated to a different use.
- If the program attempts to access an absent page or segment the exception is raised and the operating system can bring the page or segment back into main memory, then allow the program to retry the access.
- When implemented with the paged memory management scheme, this process is known as **demand-paged virtual memory.**
- Over-exploiting this facility causes the operating system to switch pages in and out of memory at a high rate.
- This is described as **thrashing,** and will adversely affect performance.

Restartable instructions:

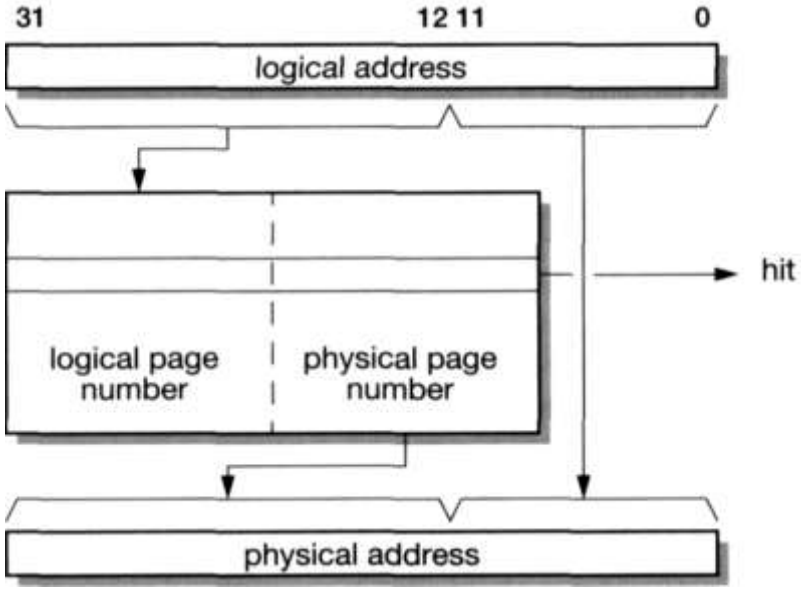
- An important requirement in a virtual memory system is that any instruction that can cause a memory access fault must leave the processor in a state that allows the operating system to page-in the requested memory and resume the original program as though the fault had not happened.
- This is often achieved by making all instructions that access memory **restartable**.
- The processor must retain enough state to allow the operating system to recover enough of the register values so that, when the page is in main memory, the faulting instruction is retried with identical results to those that would have been obtained had the page been resident at the first attempt.

Translation look-aside buffers:

- The paging scheme gives the programmer complete freedom and transparency in the use of memory, but this has been achieved at considerable cost in performance since each memory access to have incurred an overhead of two additional memory accesses, one to the page directory and one to the page table, before the data itself is accessed.
- This overhead is usually avoided by implementing a translation look-aside buffer (TLB), which is a cache of recently used page translations.
- The line and block sizes usually equate to a single page table entry, and the size of a typical TLB is much smaller than a data cache at around 64 entries.
- The locality properties of typical programs enable a TLB of this size to achieve a miss rate of a percent or so. The miss incur the table-walking overhead of two additional memory accesses.

Virtual and physical caches

- When a system incorporates both an MMU and a cache, the cache may operate either with virtual (pre-MMU) or physical (post-MMU) addresses.



The operation of a translation look-aside buffer

- A virtual cache has the advantage that the cache access may start immediately the processor produces an address, and, indeed, there is no need to activate the MMU if the data is found in the cache
- The drawback is that the cache may contain **synonyms,** which are duplicate copies of the same main memory data item in the cache.
- Synonyms arise because address translation mechanisms generally allow overlapping translations.
- If the processor modifies the data item through one address route it is not possible for the cache to update the second copy, leading to inconsistency in the cache.
- A physical cache avoids the synonym problem since physical memory addresses are associated with unique data items.
- A physical cache arrangement that neatly avoids the sequential access cost.
- Exploits the fact that a paging MMU only affects the high-order address bits.
- The cache is accessed by the low-order address bits.
- Provided these sets do not overlap, the cache and MMU accesses can proceed in parallel.
- The physical address from the MMU arrives at the right time to be compared with the physical address tags from the cache, hiding the address translation time behind the cache tag access.
- This optimization is not applicable to fully associative caches, and only works if the page size used by the MMU is larger than each directly addressed portion of the cache.

9. Explain the various data operations involved in ARM. Illustrate the concept of data operations in ARM processor. *[Nov'17]*

ARM Assembly Language Programming

Data processing instructions:

- ARM data processing instructions enable the programmer to perform arithmetic and logical operations on data values in registers.
- These instructions typically require two operands and produce a single result, though there are exceptions to both of these rules.
- Rules:
  - All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself.

- Each of the operand registers and the result register are independently specified in the instruction.

Simple register operands:

- A typical ARM data processing instruction is written in assembly language as shown below:

ro, r1, r2

ADD r0, r1, r2; r0:=r1+r2

- The semicolon in this line indicates it is a comment and should be ignored by the assembler.
- This example simply takes the values in two registers (r1 and r2), adds them together, and places the result in a third register (r0).
- The addition may produce a carry-out or, in the case of signed 2's-complement values, an internal overflow into the sign bit, but in either case this is ignored.

- The different instructions are available:

Arithmetic operations.

- These instructions perform binary arithmetic on two 32-bitoperands.

```
ADD   r0, r1, r2      ; r0 := r1+r2          ('ADD' is simple addition)
ADC   r0, r1, r2      ;r0 := r1+r2+C         ('ADC' is add with carry)
SUB   r0, r1, r2      ;r0 := r1-r2           ('SUB' is subtract)
SBC   r0, r1, r2      ;r0 := r1-r2+C-1       ('SBC' is subtract with carry)
RSB   r0, r1, r2      ;r0 := r1-r2           ('RSB' is reverse subtraction)
RSC   r0, r1, r2      ;r0 := r2-r1+C-1       ('RSC' reverse subtract with carry)
```

Bit-wise logical operations.

- These instructions perform the specified Boolean logic operation on each bit pair of the input operands, so in the first case r0[i]:= r1[i] AND r2[i] for each value of i from 0 to 31 inclusive, where r0[i] is the i^{th} bit of r0.

```
AND   r0, r1, r2      ; r0 := r1 and r2
ORR   r0, r1, r2      ;r0 := r1 or r2
EOR   r0, r1, r2      ;r0 := r1 xor r2
BIC   r0, r1, r2      ;r0 := r1 and not r2    ('BIC' - bit clear)
```

Register movement operations.

- These instructions ignore the first operand, which is omitted from the assembly language format, and simply move the second operand to the destination.

```
MOV   r0, r2   ; r0 := r2
MVN   r0, r2   ;r0 := not r2          ('MVN' - 'move negated')
```

Comparison operations.

- These instructions do not produce a result but just set the condition code bits (N, Z, C and V) in the CPSR according to the selected operation.

```
CMP CMN   r1, r2      ; set cc on r1 – r2     (‘CMP’ - Compare)
TST TEQ   r1, r2      ; set cc on r1 + r2     (‘CMN’ - compare negated)
```

```
                r1, r2          ; set cc on r1 and r2    ('TST' - Test)
                r1, r2          ; set cc on r1 xor r2    ('TEQ' - Test Equal)
```

Immediate operands:

- Instead of adding two registers, add a constant to a register replace the second source operand with an immediate value, with a literal constant, preceded by '#':

```
ADD #1      r3, r3      ; r3 ;=  r3 + 1
AND  #&ff   r8, r7      ; r8;=  r7[7:0]
```

- The second example shows that the immediate value may be specified in hexadecimal (base 16) notation by putting '&' after the '#'.
- Since the immediate value is coded within the 32 bits of the instruction, it is not possible to enter every possible 32-bit value as an immediate.
- The values which can be entered correspond to any 32-bit binary number where all the binary ones fall within a group of eight adjacent bit positions on a 2-bit boundary.
- Most valid immediate values are given by:

$$immediate = (0 \rightarrow 255) \times 2^{2n}$$

where $0 < n < 12$.

Shifted register operands:

- A data operation is similar to the first, but allows the second register operand to be subject to a shift operation before it is combined with the first operand. For example:

```
ADD   r3, r2, r1, LSL #3 ;      r3 := r2 + 3 x  r1
```

- Here 'LSL' indicates 'logical shift left by the specified number of bits', which in this example is 3.
- Shift operations are:
  - **LSL:** logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.
  - **LSR:** logical shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros.
  - **ASL:** arithmetic shift left; this is a synonym for LSL.
  - **ASR:** arithmetic shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros, for positive or ones with negative.
  - **ROR:** rotate right by 0 to 32 places; the bits which fall off the least significant end of the word are used,
  - **RRX:** rotate right extended by 1 place; the vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right.

```
ADD   r5, r5, r3,      LSL r2;          r5:= r5 + r3 x 2^r2
```

Setting the condition codes

- Any data processing instruction can set the condition codes (N, Z, C and V)
- At the assembly language level this request is indicated by adding an's' to the opcode, standing for 'Set condition codes'.
- A 64-bit addition of two numbers held in r0-r1 and r2-r3, using the C condition code flag to store the intermediate carry:

```
ADD   r2, r2, r0        ; 32 – bit carry out → C and added into high word
```

```
ADC     r3, r3, r1
```

- An arithmetic operation (which here includes CMP and CMN) sets all the flags according to the arithmetic result.
- A logical or move operation does not produce value for C or V, so these operations set N and Z according to the result but preserve V, and either preserve C or set C to the value of the last bit to fall off the end of the shift.

Multiplies

- A special form of the data processing instruction supports multiplication:

```
MUL r4, r3, r2                ;  r4 := (r3 x r2) [31:0]
```

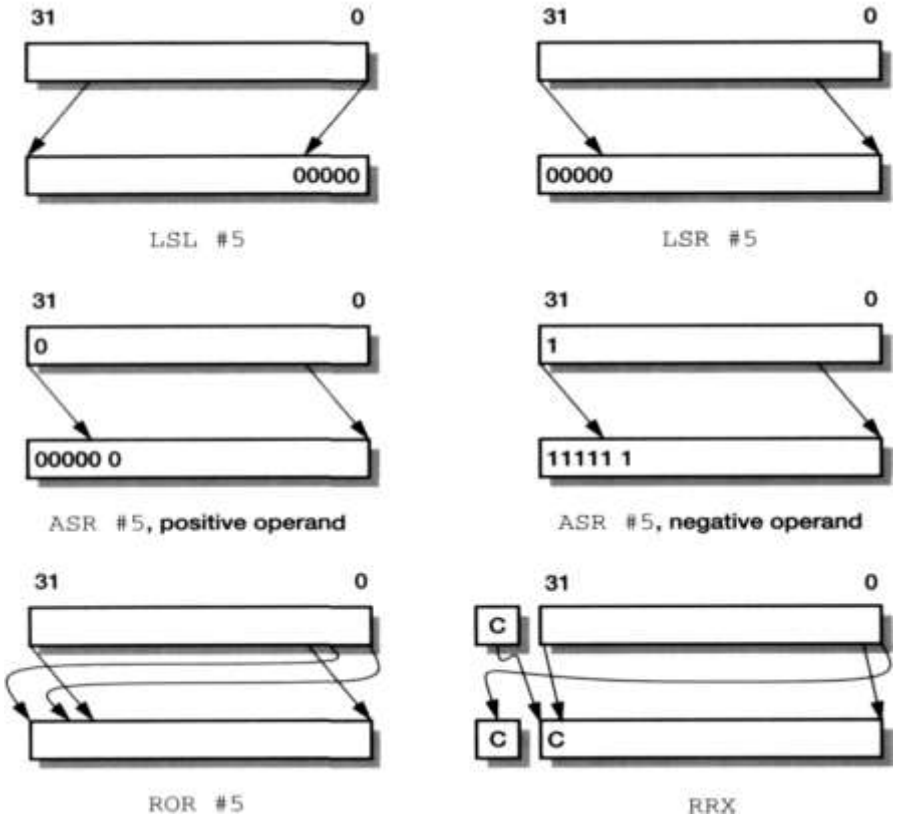
- Multiplying two 32-bit integers gives a 64-bit result, the least significant 32 bits of which are placed in the result register and the rest are ignored.
- Adds the product to a running total.
- The multiply-accumulate instruction:

```
MLA r4, r3, r2, r1            ;  r4 := (r3 x r2 + r1) [31:0]
```

- Multiplication by a constant can be implemented by loading the constant into a register rand then using one of these instructions, but it is usually more efficient to use a short series of data processing instructions using shifts and adds or subtracts.

- For example, to multiply r0 by 35:

```
ADDRSB  r0, r0, r0, r0 , r0, r0,  LSL      r0' := 5 x
                                  #2; LSL   r0, r0"   := 7 (=35 x r0)
                                  #3;              x r0'
```



ARM shift operations

Data transfer instructions:

- Data transfer instructions move data between ARM registers and memory.
  - **Single register load and store instructions.**
    - These instructions provide the most flexible way to transfer single data items between an ARM register and memory.
    - The data item may be a byte, a 32-bitword, or a 16-bit half-word.

      LDR r0, [r1] ; r0 := mem_{32} [r1]

      STR r0, [r1] mem_{32} [r1] := r0
  - **Multiple register load and store instructions.**
    - These instructions are less flexible than single register transfer instructions, but enable large quantities of data to be transferred more efficiently.
    - They are used for procedure entry and exit, to save and restore workspace registers, and to copy blocks of data around memory.
  - **Single register swap instructions.**
    - These instructions allow a value in a register to be exchanged with a value in memory, effectively doing both a load and a store operation in one instruction.

Register-indirect addressing:

- The ARM data transfer instructions are all based around register-indirect addressing, with modes that include base-plus-offset and base-plus-index addressing.
- Register-indirect addressing uses a value in one register (the **base** register) as a memory address and either **loads** the value from that address into another register or **stores** the value from another register into that memory address.
- These instructions are written in assembly language as follows:

  LDR r0, [r1] ; r0 := mem_{32} [r1]

  STR r0, [r1] mem_{32} [r1] := r0

Initializing an address pointer:

- To load or store from or to a particular memory location, an ARM register must be initialized to contain the address of that location.
- In the case of single register transfer instructions, an address within 4 Kbytes of that location.
- If the location is close to the code being executed it is often possible to exploit the fact that the program counter, r15, is close to the desired address.
- A data processing instruction can be employed to add a small offset to r15.
- ARM assemblers have an inbuilt 'pseudo instruction', ADR, which makes this easy.
- A pseudo instruction looks like a normal instruction in the assembly source code but does not correspond directly to a particular ARM instruction.
- As an example, consider a program which must copy data from TABLE1 to TABLE2, both of which are near to the code:

```
COPY    ADR   r1, TABLE1        ; r1 points to TABLE1
        ADR   r2, TABLE2        ; r2 points to TABLE2
        ….
TABLE1                          ; < Source of data>
        ….
TABLE2                          ; <destination>
```

- The first ADR pseudo instruction causes r1 to contain the address of the data that follows TABLE1; the second ADR likewise causes r2 to hold the address of the memory starting at TABLE2.

- Copy the first word from one table to the other:

```
COPY    ADR   r1, TABLE1        ; r1 points to TABLE1
        ADR   r2, TABLE2        ; r2 points to TABLE2
        LDR   r0, [r1]          ; load first value…
        STR   r0, [r2]          ; and store it in TABLE2
        ….
TABLE1                          ; < Source of data>
        ….
TABLE2                          ; < destination>
```

- Use data processing instructions to modify both base registers ready for the next transfer:

```
COPY    ADR   r1, TABLE1        ; r1 points to TABLE1
        ADR   r2, TABLE2        ; r2 points to TABLE2
LOOP    LDR   r0, [r1]          ; get TABLE1 1st word
        STR   r0, [r2]          ; copy into TABLE2
        ADD   r1, r1, #4        ; step r1 on 1 word
        ADD   r2, r2, #4        ; step r2 on 1 word
                                ; if more go back to LOOP
        ….
TABLE1                          ; < Source of data>
        ….
TABLE2                          ; < destination>
```

Base plus offset addressing: [Pre – indexed addressing mode]

- If the base register does not contain exactly the right address, an offset of up to4 Kbytes may be added (or subtracted) to the base to compute the transfer address:

```
LDR r0, [r1, #4]    ;    r0    := men32 [r1 + 4]
```

- It allows one base register to be used to access a number of memory locations which are in the same area of memory.
- Sometimes it is useful to modify the base register to point to the transfer address by using pre-indexed addressing with **auto-indexing,** and allows the program to walk through a table of values:

```
LDR r0, [r1, #4]!   ;    r0    := men32 [r1 + 4];
                         r1    := r1 + 4
```

- The exclamation mark indicates that the instruction should update the base register after initiating the data transfer.
- **Post-indexed** addressing, allows the base to be used without an offset as the transfer address, after which it is auto-indexed:

```
LDR r0, [r1], #4    ;    r0    := mem32 [r1]
                         r1    := r1 + 4
```

- Here the exclamation mark is not needed, since the only use of the immediate offset is as a base register modifier.
- Using the last of these forms we can now improve on the table copying program example introduced earlier:

Example:

```
COPY    ADR   r1, TABLE1        ; r1 points to TABLE1
        ADR   r2, TABLE2        ; r2 points to TABLE2
LOOP    LDR   r0, [r1] , #4     ; get TABLE1 1st word
        STR   r0, [r2] , #4     ; copy into TABLE2
        ???                     ; if more go back to LOOP
        ….
TABLE1                          ; < Source of data>
        ….
TABLE2                          ; < destination>
```

- The size of the data item which is transferred may be a single unsigned 8-bit byte instead of a 32-bit word. This option is selected by adding a letter B onto the opcode:

```
LDRB r0, [r1]        ; r0 : = mem8 [r1]
```

- In this case the transfer address can have any alignment and is not restricted to a4-byte boundary, since bytes may be stored at any byte address.
- The loaded byte is placed in the bottom byte of r0 and the remaining bytes in r0 are filled with zeros.

Multiple register data transfers:

- A simple example of this instruction class is:

```
LDMIA   r1, (r0, r2, r5)     ; r0 : = mem32 [r1]
                             ; r2 : = mem32 [r1 + 4]
                             ; r5 : = mem32 [r1 + 8]
```

- Since the transferred data items are always 32-bit words, the base address (r1) should be word-aligned.
- The transfer list, within the curly brackets, may contain any or all of r0 to r15. The order of the registers within the list is insignificant and does not affect the order of transfer or the values in the registers after the instruction has executed.

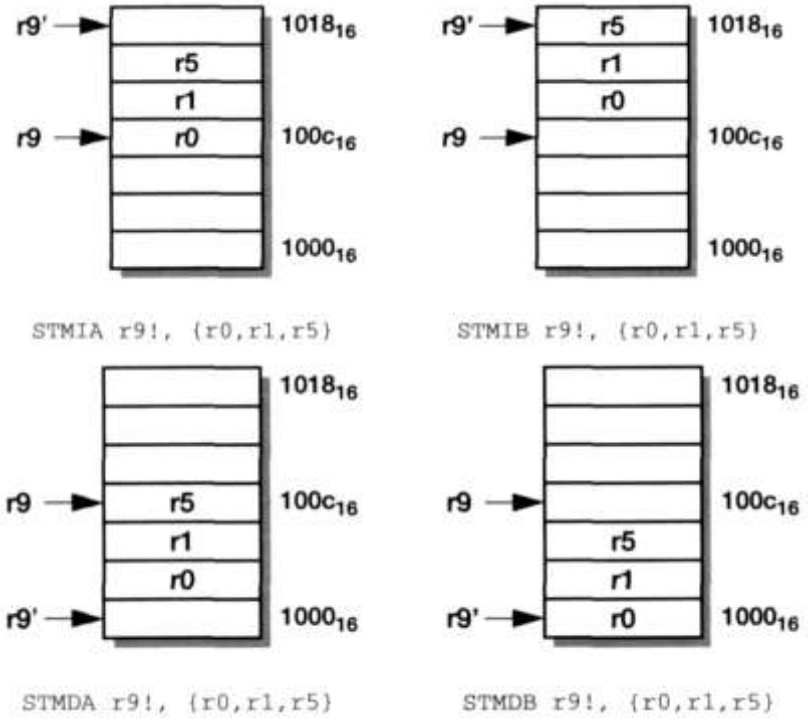
Stack addressing:

- A stack is a form of last-in-first-out store which supports memory allocation where the address to be used to store a data value is not known at the time the program is compiled or assembled.
- A stack is usually implemented as a linear data structure which grows up (an **ascending** stack) or down (a **descending** stack) memory as data is added to it and shrinks back as data is removed.
- A **stack pointer** holds the address of the current top of the stack, either by pointing to the last valid data item pushed onto the stack (a **full** stack), or by pointing to the vacant slot where the next data item will be placed (an **empty** stack).
- The ARM multiple register transfer instructions support all four forms of stack:
    - Full ascending: the stack grows up through increasing memory addresses and the base register points to the highest address containing a valid item.

- Empty ascending: the stack grows up through increasing memory addresses and the base register points to the first empty location above the stack.
- Full descending: the stack grows down through decreasing memory addresses and the base register points to the lowest address containing a valid item.
- Empty descending: the stack grows down through decreasing memory addresses and the base register points to the first empty location below the stack.

Block copy addressing:

- The block copy view is based on
  - whether the data is to be stored above or below the address held in the base register
  - Whether the address incrementing or decrementing begins before or after storing the first value.
  - The mapping between the two views depends on whether the operation is a load or a store.



Multiple register transfer addressing modes

- The block copy which given below shows how each variant stores three registers into memory and how the base register is modified if auto-indexing is enabled.
- The base register value before the instruction is r9, and after the auto-indexing it is r9'.
- Here are two instructions which copy eight words from the location r0 points to the location r1 points to:

```
LDMIA   r0!,   [r2-r9]
STMIA   r1,    [r2-r9]
```

- After executing these instructions r0 has increased by 32 since the '!' causes it to auto-index across eight words, whereas r1 is unchanged. If r2 to r9 contained useful values, we could preserve them across this operation by pushing them onto a stack:

```
STMFD   r13!,  [r2 – r9]    save regs onto stack
LDMIA   r0!,   [r2 – r9]
STMIA   r1,    [r2 – r9]
LDMFD   r13!,  [r2 – r9]    restore from stack
```

- Here the 'FD' postfix on the first and last instructions signifies the full descending stack address mode.
- **Table shows** the mapping between the stack and block copy views of the load and store multiple instructions.

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMED	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

10. Discuss in detail about Control flow instructions

- This third category of instructions neither processes data nor moves it around; does it simply determine which instructions get executed next.

Branch instructions:

- The most common way to switch program execution from one place to another is use the branch instruction:

```
To
B                   LABEL
…
…
LABEL
```

- The processor normally executes instructions sequentially, but when it reaches the branch instruction it proceeds directly to the instruction at LABEL instead of executing the instruction immediately after the branch.

Conditional branches:

- Sometimes you will want the processor to take a decision whether or not to branch.
- A typical loop control sequence might be:

```
MOV         r0, #0          ; initialize counter LOOP
ADD         r0, r0, #1      ; increment loop counter
CMP BNE     r0, #10         ; Compare with limit
            LOOP            ; repeat if not equal
                            ; Else fall through
```

Table - Branch conditions

Branch	Interpretation	Normal uses
B BAL	Unconditional	Always take this branch
	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set Higher	Arithmetic operation gave carry-out
BHS	or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Conditional execution:

- An unusual feature of the ARM instruction set is that conditional execution applies not only to branches but to all ARM instructions.
- A branch which is used to skip a small number of following instructions may be omitted altogether by giving those instructions the opposite condition. For example, consider the following sequence:

```
        CMP     r0, #5
        BEQ     BYPASS          ; if (r0 != 5)
        ADD     r1, r1, r0      ; r1 := r1 + r0 – r2
        SUB     r1, r1, r2      ;
BYPASS  …...
```

- This may be replaced by:

```
        CMP     r0, #5          ; if (r0 != 5)
        ADDNE   r1, r1, r0      ; r1 := r1 + r0 – r2
        SUBNE   r1, r1, r2      ;
```

Branch and link instructions:

- A common requirement in a program is to be able to branch to a subroutine in a way which makes it possible to resume the original code sequence when the subroutine has completed.
- This requires that a record is kept of the value of the program counter just before the branch is taken.

```
        BL      SUBR    ; branch to SUBR
        ….              ; return to here
SUBR    ….              ; Subroutine entry point
        MOV     pc, r14 ; return
```

- The normal mechanism used here is to push r14 onto a stack in memory.
- Since the subroutine will often require some work registers, the old values in these registers can be saved at the same time using a store multiple instruction:

```
        BL      SUB1        ; branch to SUBR
SUB1    STMFD   r13! (r0 – r2, r14) BL      ; save work & link regs
        SUB2
        ….
SUB2
```

Subroutine return instructions:

- To get back to the calling routine, the value saved by the branch and link instruction in r14 must be copied back into the program counter.

```
SUB2
        MOV     pc, r14             ; copy r14 into r15 to return
```

- The program counter as r15 means that any of the data processing instructions can be used to compute a return address, though the 'MOV' form.
- Where the return address has been pushed onto a stack, it can be restored along with any saved work registers using a load multiple instruction:

```
SUB1    STMFD   r13!,   (r0 – r2, r14)  ; save work regs & link BL
        SUB2
        …..
        LDMFD   r13!,   (r0 – r2, pc)   ; restore work regs & return
```

Supervisor calls

- Whenever a program requires input or output, for instance to send some text to the display, it is normal to call a supervisor routine.
- The supervisor is a program which operates at a privileged level, which means that it can do things that a user-level program cannot do directly.
- The supervisor provides trusted ways to access system resources which appear to the user-level program rather like special subroutine accesses.
- The instruction set includes a special instruction to call these functions, SWI **'Software Interrupt' or 'Supervisor Call'**)
- The most useful of these is a routine which sends the character in the bottom byte of r0 to the user display device:

```
        SWI     SWI_WriteC      ; output r0 [7:0]
```

- Another useful call returns control from a user program back to the monitor program:

```
        SWI     SWI_Exit        ; return to monitor
```

Jump tables:

- The idea of a jump table is that a programmer sometimes wants to call one of a setof subroutines, the choice depending on a value computed by the program.

```
        BL      JUMPTAB

JUMPTAB CMP     r0, #0
        BEQ     SUB0 r0,
        CMP     1, SUB1
```

```
        BEQ     r0, #2
        CMP     SUB2
        BEQ
```

- A solution which is more efficient in this case exploits the visibility of the program counter in the general register file:

```
        BL      JUMPTAB

JUMPTAB ADR     R1, SUBTAB          ; r1→ SUBTAB
        CMP     r0, #SUBMAX         ; check for overrun…if OK,
        LDRLS   pc, [r1, r0, LSL #2] ; table jump…..otherwise
        B       ERROR               ; Signal error
SUBTAB  DCD     SUB0                ; Table of subroutine
        DCD     SUB1                ; Entry points
        DCD     SUB2
```

- The 'DCD' directive instructs the assembler to reserve a word of store and to initialize it to the value of the expression to the right, which in these cases is just the address of the label.
- The overrun check is implemented by making the load into the PC conditional, so the overrun case skips the load and falls into the branch to the error handler.
- The only performance cost of checking for overrun is the comparison with the maximum value.

```
CMP     r0, # SUBMAX            ; Check for overrun
BHI     ERROR                   ; if overrun call error
LDR     pc, [r1, r0, LSL #2]    ; else table jump
```

10. Write a simple assembly language program to display "Hello World".

- To sort of simple test program is often referred to as a Hello World program because all it does is print 'Hello World' on the display before terminating.
- Here is an ARM assembly language version:

```
        AREA    Hello, CODE, READONLY   ; declare code area
SWI_WriteC      EQU &0                  ; output character in r0
SWI_Exit        EQU &11                 ; finish program
        ENTRY                           ; Code entry point
        START   ADR   r1, text          ; r1 → "Hello World"
LOOP    LDRB    r0, [r1], #1            ; get the next byte
        CMP     r0, #0                  ; Check for next end
        SWINE   SWI_WriteC              ; if not end print…
        BNE     LOOP                    ; …. And Loop back
        SWI     SWI_Exit                ; end of execution
TEXT    =       "Hello World", &0a, &0d, 0
        END                             ; end of program source
```

- This program illustrates a number of the features of the ARM assembly language and instruction set:
  - The declaration of the code 'AREA', with appropriate attributes.
  - The definitions of the system calls which will be used in the routine. (In a larger program these would be defined in a file which other code files would reference.)

- The use of the ADR pseudo instruction to get an address into a base register.
- The use of auto-indexed addressing to move through a list of bytes.
- Conditional execution of the SWI instruction to avoid an extra branch.

11. Write a simple assembly language program to display "Hello World" using word loads and stores to copy the table.

- This program uses word loads and stores to copy the table, which is why the tables must be word-aligned.
- It then uses byte loads to print out the result using a routine which is the same as that used in the 'Hello World' program.

```
            AREA        BlkCpy, CODE, READONLY
SWI_WriteC              EQU   &0                ; output character in r0
SWI_Exit                EQU   &11               ; finish program
            ENTRY                               ; Code entry point
            ADR         r1, TABLE1              ; r1→ TABLE1
            ADR         r2, TABLE2              ; r2→ TABLE2
            ADR         r3, T1END               ; r3→ T1END
LOOP1       LDRB        r0, [r1], #4            ; get TABLE1 1st word
            STR         r0, [r2], #4            ; Copy into TABLE2
            CMP         r1, r3                  ; Finished?
            BLT         LOOP1                   ; if not, do more
            ADR         r1, TABLE2              ; r1→ TABLE2
LOOP2       LDRB        r0, [r1], #1            ; get next byte
            CMP         r0, #0                  ; Check for next end
            SWINE       SWI_WriteC              ; if not end, print
            BNE         LOOP2                   ; and loop back
            SWI         SWI_EXIT                ; Finish
TABLE1      =           "This is the Right string!", &0a, &0d, 0
T1END
            ALIGN                               ; ensure word alignment
TABLE2      =           "This is the Wrong string!" 0
            END
```

12. Write a simple assembly language program to Print out r1 in hexadecimal.

- This is a useful little routine which dumps a register to the display in hexadecimal (base16) notation.
- It can be used to help debug a program by writing out register values and checking that algorithms are producing the expected results

```
            AREA Hex_Out,       CODE, READONLY
SWI_WriteC      EQU   &0                    ; output character in r0
SWI_Exit    EQU   &11                       ; finish program
                ENTRY                       ; code entry point
                LDR         r1, VALUE       ; get value to print
                BL          HexOut          ; call hexadecimal output
                SWI         SWI_Exit        ; finish
    VALUE       DCD         &12345678       ; test value
    HexOut      MOV         r2, #8          ; nibble count = 8
    LOOP        MOV          r0, r1, LSR #28 ; get top nibble
                CMP         r0, #9          ; 0-9 or A-F?
```

```
ADDGT    r0, r0, #"A"-10    ; ASCII alphabetic
ADDLE    r0, r0, #"0"       ; ASCII numeric
SWI      SWI_WriteC         ; print character
MOV      r1, r1, LSL #4     ; shift left one nibble
SUBS     r2, r2, #1         ; decrement nibble count
BNE      LOOP               ; if more do next nibble
MOV      pc, r14            ; return
END
```

13. Write a subroutine to output a text string immediately following the call instruction using ARM processor? *(Apr'17)*

- It is often useful to be able to output a text string without having to set up a separate data area for the text.

```
        BL      TextOut
        =       "Test String", &0a, &0d, 0
ALIGN
….                                      ; Return to here
```

- The issue here is that the return from the subroutine must not go directly to the value put in the link register by the call.

```
        AREA    Text_Out, CODE, READONLY
SWI_WriteC      EQU  &0                 ; output character in r0
SWI_Exit        EQU  &11                ; finish program
        ENTRY                           ; Code entry point
        BL      TextOut                 ; print the following string
        =       "Test String", &0a, &0d, 0
        ALIGN
        SWI     SWI_Exit                ; Finish
TextOut LDRB    r0, [r14], #1           ; get next character
        CMP     r0, #0                  ; Test for end mark
        SWINE   SWI_WriteC              ; if not end, print….
        BNE     TextOut                 ; ….. and loop
        ADD     r14, r14, #3            ; pass nest word boundary
        BIC     r14, r14, #3            ; round back to boundary
        MOV     pc, r14                 ; return
        END
```

14. Briefly explain the Architectural Support for Operating Systems.

- The role of an operating system is to present a uniform and clean interface between the underlying hardware resources of the machine and the application programs that run on it.
- The most sophisticated operating systems are those that provide facilities for multiple general-purpose programs run by several different users at the same time.

MULTI - USER SYSTEMS:

- It is very inconvenient if a multi-user system requires each program to make allowances for the presence of other programs in the same machine.
- Since the number and type of concurrent programs is unknown and will vary from one run to the next.

- Therefore a multi-user operating system presents each program with a complete **virtual machine** in which to operate.
- Each program can be written as though it is the only program running at the time.
- Although several programs may be present in the machine at one time, the processor has only one set of registers, so only one program is executing at any particular time.
- The apparent concurrency is achieved by **time-slicing.**
- Since the processor operates at very high speeds by human standards, the effect is that over a period of.
- The operating system is responsible for **scheduling** (deciding which program runs when), and it may give each program an equal share of the CPU time or it may use **priority** information to favour some programs over others.
- A program is switched out of the processor either because the operating system is invoked by a timer interrupt and decides the program had enough time for now, or because the program has requested a slow peripheral access (such as a disk access) and cannot do any more useful work until it gets a response.
- Rather than leave the program idling in the processor, the operating system switches it out and schedules another program that can make useful progress.

Memory management:

- In order to create the virtual machine in which a program runs the operating system, must establish an environment where the program has access to its code and data at the memory locations where it expects to find them.
- Since one program's expectations of the addresses it will use may conflict with another's.
- The operating system uses memory translation to present the **physical** memory locations where it has loaded the code and data to the program at appropriate **logical** addresses.

Protection:

- Where several users are running the programs on the same machine it is highly desirable to ensure that an error in one user's program cannot interfere with the operation of any of the other programs.
- The memory-mapping hardware which gives each program its own virtual machine can also ensure that a program cannot see any memory belonging to another program.
- It is not efficient to enforce this too far, however, sharing areas of memory that contain, libraries useful.
- A solution here is to make these areas read-only or execute-only so one program cannot corrupt code that will be used by another.
- An obvious route for a malicious user to cause damage to another is to overcome the protection afforded by the memory-management system by assuming operating system status and then changing the translation tables.
- Designing a computer system to be secure against malicious attacks by clever individuals is a complex issue which requires some architectural support.
- On the ARM this support is provided by privileged processor modes with controlled access and various forms of memory protection in the memory management units.

Resource allocation

- Two programs which are running concurrently may place conflicting demands on system resources.
- For example, one program may request data from one part of a disk.

- It will be switched out while the disk drive seeks the data, and the program that gets switched in may immediately request data from a different part of the disk
- If the disk drive responds directly to these requests a situation can easily arise where the programs alternately have control and the disk drive oscillates between the two seeks, never having long enough to find either data area, and the system will live-lock until the disk drive wears out.
- In order to avoid this sort of scenario, all requests for input/output activity are channelled through the operating system.
- It will accept the request from the first program and then queue up the request from the second program to receive attention once the first has been satisfied.

SINGLE - USER SYSTEMS

- Where a system serves a single user, still possibly running several programs at the same time, much of the above continues to apply.
- Although the threat of a malicious user sharing the same machine is removed, it is still very useful for each program to run in its own space so that an error in one program does not cause errors in another.
- The simplification that arises from removing the concern about the malicious user is that it is no longer necessary to make it impossible for a program to assume system privileges.
- However, desktop machines that appear to belong to one user are increasingly being connected to computer networks that allow other users to run programs on them remotely.
- Such machines should clearly be viewed as multi-user and incorporate appropriate levels of protection.

Embedded systems:

- An embedded system is quite different from the single- and multi-user general-purpose system.
- It typically runs a fixed set of programs and has no mechanism for introducing new programs.
- Presumably, then, the problem of the malicious user has been removed.
- The operating system continues to play a similar role, giving each active program a clean virtual machine in which to run.
- Many embedded systems operate within real-time constraints which must be allowed to determine scheduling priorities.
- Cost issues has led to the development of the real-time operating system **(RTOS)** which provides the scheduling and hardware interface facilities required by an embedded system using just a few kilobytes of memory.
- Smaller embedded systems may not even be able to bear this cost, or they may have such simple scheduling requirements
- A simple 'monitor' program suffices, providing a few system functions such as sanitized interfaces to input/ output functions.
- Such systems almost certainly dispense with the memory management hardware and use the processor's logical address to access memory directly.
- If an embedded system includes a cache memory some mechanism is required to define which areas of memory are cacheable

Chapter structure:

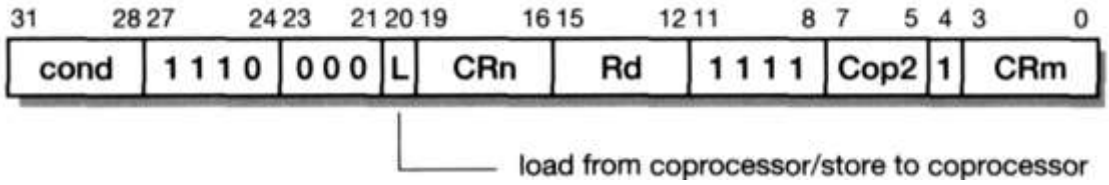
- The ARM system control coprocessor and the memory management systems it controls, which include a full MMU with address translation and a simpler 'protection unit' for embedded systems that do not require address translation.

The ARM system control coprocessor

- The ARM system control coprocessor is an on-chip coprocessor, using logical coprocessor number 15, which controls the operation of the on-chip cache or caches, memory management or protection unit, write buffer, pre-fetch buffer, branch target cache and system configuration signals.

CP15 instructions

- The control is affected through the reading and writing of the CP15 registers. The registers are all 32 bits long and access is restricted to MRC and MCR instructions which must be executed in supervisor mode.
- Use of other coprocessor instructions or any attempted access in user mode will cause the undefined instruction trap to be taken.
- In most cases the CRm and Cop2 fields are unused and should be zero, though they are used in certain operations.



CP15 register transfer instructions

Protection unit:

- ARM CPUs which are used in embedded systems with fixed or controlled application programs do not require a full memory management unit with address translation capabilities. For such systems a simpler protection unit is adequate.

MMU:

- ARM CPUs for use in general-purpose applications where the range and number of application programs is unknown at design time will usually require a full memory management unit with address translation

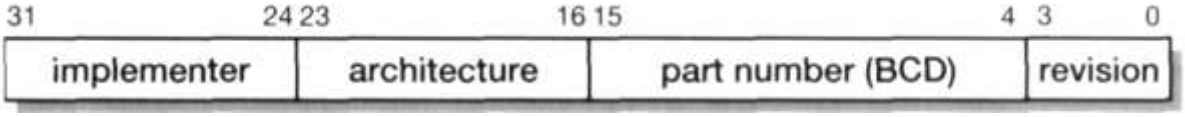
CP15 protection unit registers

- The protection unit register structure is illustrated in table below. The registers are read and written using the CP15 instruction shown in figure, with CRn specifying the register to be accessed.

Table CP15 protection unit register structure.

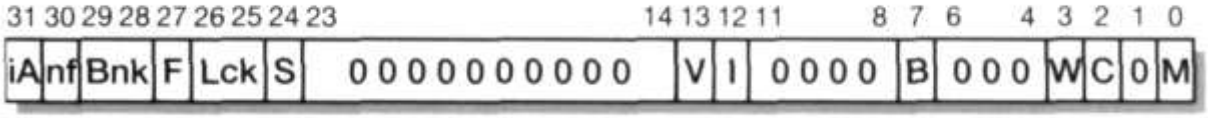
Register	Purpose
0	ID Register
1	Configuration
2	Cache control
3	Write Buffer control
5	Access Permissions
6	Region Base and Size
7	Cache Operations
9	Cache Lock Down
15	Test
4, 8, 10-14	UNUSED

Register 0 (which is read-only) returns device identification information



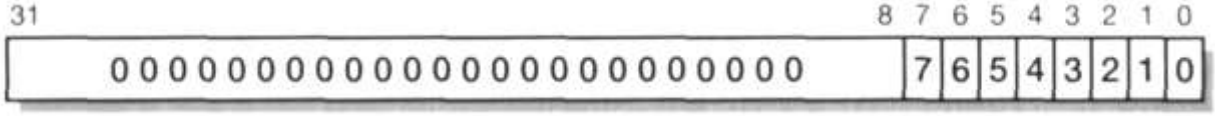
- Bits [3:0] contain a revision number,
- Bits [15:4] contain a 3-digit part number in binary-coded decimal,
- Bits [23:16] contain the architecture version (0 for version3, 1 for version 4, 2 for version 4T, 4 for version 5T)
- Bits [31:24] contain the ASCII code of an implementer's trademark (ASCII 'A' = 4116 indicates ARM Limited, 'D' = 44I6 indicates Digital, and so on).

Register 1 (which is read-write) contains several bits of control information which enable system functions and control system parameters.



- All bits are cleared on reset.
    - M enables the protection unit.
    - C enables the data or unified cache,
    - W enables the write buffer,
    - B switches from little-to big-endian byte ordering,
    - I enables the instruction cache when this is separate from the data cache,
    - V causes the exception vectors to move to near the top of the address space,
    - S, Lck, F and Bnk are used to control the cache (on theARM740T),
    - nf and iA control various clock mechanisms (on theARM940T).

Register 2 (which is read-write) controls the cache ability of the eight individual protection regions.



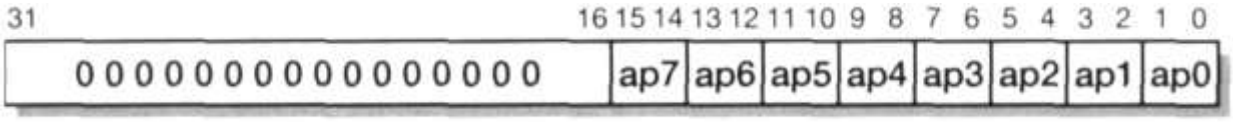
- Bit 0 enables the cache for loads within region 0, bit 1 likewise for region 1, and so on.
- The ARM940T has separate protection units on its instruction and data ports and Cop2 is used to determine which unit is accessed: Cop2 = 0 gives access to the protection unit on the data port.

Register 3 (which is read-write)

- It defines whether the write buffer should be used for each of the protection regions.
- Its format is the same as that for register 2, but ARM940T instruction port is read-only, the write buffer can only be enabled for the data port and so Cop2 should always be zero.

Register 5 (which is read-write):

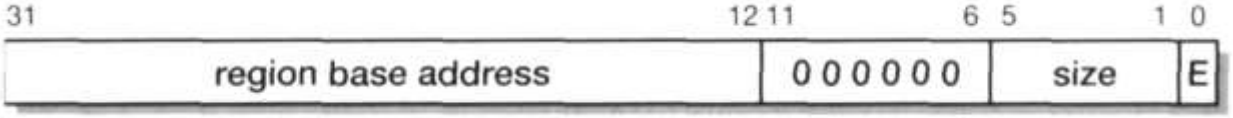
- It gives the access permissions for each of the protection regions.



  - The access permissions cover no access (00),
  - privileged modes only (01),
  - privileged full access and user read only (10)
  - Full access (11).
- Again theARM940T uses the Cop2 field to differentiate the instruction (1) and data (0) protection units.

Register *6* (which is read-write):

- It defines the start address and size of each of the eight regions.



- The region base address must be a multiple of the size. The encoding of the size field is shown. E enables the region.
- The particular region is specified in the CRm field which should be set from 0 to 7.
- For a Harvard core such as the ARM940T there are separate region registers for the instruction and data memory ports and Cop2 specifies which memory port is to be addressed as described above for register 2.

Register 7:

- It controls various cache operations and its operation is different for theARM740T and the ARM940T.

Register 9:

- It is used in the ARM940T to lock down areas of the cache. (TheARM740T uses certain bits in register 1 for this purpose.)

Register 15 is used in the ARM940T to modify the cache allocation algorithm from random to round-robin. This is intended for use only during silicon production testing.

15. Why, on the ARM, can user-level code not disable interrupts?

- To allow a user to disable interrupts would make building a protected operating system impossible.
- The following code illustrates how a malicious user could destroy all the currently active programs:

```
        MSR   CPSR f, #&co          ; disable IRQ and FIQ
HERE    B     HERE                  ; loop forever
```

- Once interrupts are disabled there is no way for the operating system to regain control, so the program will loop forever.
- The only way out is a hard reset, which will destroy all currently active programs.
- If the user cannot disable interrupts the operating system can establish a regular periodic interrupt from a timer, so the infinite loop will be interrupted and the operating system can schedule other programs.
- This program will either time-out, if the operating system has an upper limit on the amount of CPU time it is allowed to consume, or it will continue to loop whenever it gets switched in, running up a large billon a system with accounting.

16. Write ARM assembly language program to multiply two 32 – bit binary numbers. *[Apr'18]*

```
.Model small

.data
      mult1 dw 2521h
            dw 3206h
mult2 dw 0a26h
            dw 6400h
      ans   dw 0,0,0,0

.code
      mov ax,@data
      mov ds,ax

;     lea si,ans

      mov ax,mult1
      mul mult2
      mov ans,ax
      mov ans+2,dx

      mov ax,mult1+2
      mul mult2
      add ans+2,ax
      adc ans+4,dx
      adc ans+6,0

      mov ax,mult1
      mul mult2+2
      add ans+2,ax
      adc ans+4,dx
```

```
        adc ans+6,0

        mov ax,mult1+2
        mul mult2+2
        add ans+4,ax
        adc ans+6,dx

     mov ax,4c00h
int 21h
End
```

13. Write a subprogram which copies a string of bytes from one memory location to another. The start of the source string will be passed in r_1, the length (in bytes) in r_2 and the start of the destination string in r_3. *[Apr'18]*

```
AREA    StrCopy, CODE, READONLY
        ENTRY                       ; Mark first instruction to execute
start
        LDR     r1, =srcstr         ; Pointer to first string
        LDR     r0, =dststr         ; Pointer to second string
        BL      strcopy             ; Call subroutine to do copy
stop
        MOV     r0, #0x18           ; angel_SWI reason_Report Exception
        LDR     r1, =0x20026        ; ADP_Stopped_Application Exit
        SVC     #0x123456           ; ARM semi hosting (formerly SWI)
strcopy
        LDRB    r2, [r1],#1         ; Load byte and update address
        STRB    r2, [r0],#1         ; Store byte and update address
        CMP     r2, #0              ; Check for zero terminator
        BNE     strcopy             ; Keep going if not
        MOV     pc,lr               ; Return

        AREA    Strings, DATA, READWRITE
srcstr  DCB     "First string - source",0
dststr  DCB     "Second string - destination",0
        END
```

ANNA UNIVERSITY QUESTIONS

PART A

1. Write the CPSR format of ARM Processor. *[Apr'18]*
2. What are the various ARM development tools? [OR] List out some of ARM development tools. *[Nov'16, Nov'17]*
3. What is the purpose of program counter? *(Nov'16)*
4. Define Context Switching? *(Apr'17)*
5. State the function of ARMulator and define its operations at various levels of accuracy? *(Apr'17)*
6. Differentiate little – endian and big – endian memory organizations. *[Apr'18]*

PART B

1. With neat sketch explain the functional block diagram ARM architecture. *(Nov'16, Nov'17)*
2. Write short notes on ARM MMU architecture. *(Apr'17)*
3. Explain the various operating modes programmers model in ARM processor. *[Nov'16]*

4. Explain the ARM programmer's model in detail, with supporting diagram. *[Apr'17]*
5. Draw and explain the visible registers in an ARM processor. *[Apr'18]*
6. Explain the structure of the ARM cross – development tool kit. *[Apr'18]*
7. Explain the various data operations involved in ARM. Illustrate the concept of data operations in ARM processor. *[Nov'17]*
7. Write a subroutine to output a text string immediately following the call instruction using ARM processor? *(Apr'17)*
8. Write ARM assembly language program to multiply two 32 – bit binary numbers. *[Apr'18]*
9. Write a subprogram which copies a string of bytes from one memory location to another. The start of the source string will be passed in r_1, the length (in bytes) in r_2 and the start of the destination string in r_3. *[Apr'18]*

UNIT – V PART A

1. What is instruction pipeline?

- A pipeline is the mechanism a RISC processor uses to execute instructions.
- Using a pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed.
- One way to view the pipeline is to think of it as an automobile assembly line, with each stage carrying out a particular task to manufacture the vehicle.

2. What is the need of pipeline in ARM?

- The ARM uses a pipeline in order to increase the speed of the flow of instructions to the processor.
- Allow several operations to be undertaken simultaneously, rather than serially.
- Rather than pointing to the instruction being executed, the PC points to the instruction being fetched.

3. What are the principal components of 3 stage pipeline? [OR] **What is the three stage pipelining in ARM processor?** *[Nov'17]*

The principal components are:

- The register bank, which stores the processor state.
- The barrel shifter, which can shift or rotate one operand by any number of bits.
- The ALU, which performs the arithmetic and logic functions required by the instruction set.
- The address register and incrementer, which select and hold all memory addresses and generate sequential addresses when required.
- The data registers, which hold data passing to and from memory

4. What is ARM datapath timing? *[Apr'18]*.

- The datapath cycles is the sum of:
    - The register read time
    - The shifter delay
    - The ALU delay
    - The register write set up time
    - The phase 2 to phase 1 non – overlap time.

5. What are the ARM 3 stage pipelines?

ARM processors employ a simple 3 stage pipeline with the following pipeline stages:

- **Fetch**:
    - The instruction is fetched from memory and placed in the instruction pipeline.
- **Decode**:
    - The instruction is decoded and the datapath control signals prepared for the next cycle. In this stage the instruction 'owns' the decode logic but not the datapath.
- **Execute:**
    - The instruction 'owns' the datapath; the register bank is read, an operand shifted, the ALU result generated and written back into a destination register.

6. Define data processing Instructions executions.

- The execution of an ARM instruction can best be understood by start with a simple data processing instruction.
- Basically it needs two operands, one of which is always a register and the other is either also a register or an immediate value.

7. Define data transfer Instructions executions.

- The way of identification of a memory address for computing results in a data transfer instruction (load or store) is similar to the way of a data processing instruction.
- A register is used as the base address, to which is added or subtracted an offset which again may be another register or an immediate value.

8. What are the 5 stages of pipeline? [OR] **What is five stage pipeline in ARM Processor?** *(Nov'16)*

The ARM processors which use a 5-stage pipeline have the following pipeline stages:

- Fetch
- Decode
- Execute
- Buffer/data
- Write-back

9. Define branch instructions executions

- Branch instructions compute the target address in the first cycle.
- A 24-hit immediate field is extracted from the instruction and then shifted left two bit positions to give a word-aligned offset which is added to the PC.

10. What is class of ARM implementation?

- In ARM implementation the design is divided into a data path section that is described in register transfer level (RTL) notation and a control section that is observed as a finite state machine (FSM).

11. List the various ARM implementations.

- Adder design
- ALU functions
- ARM6 carry-select adder ARM6 ALU structure
- Carry arbitration Adder
- The barrel shifter

12. List at least 4 instruction set used in ARM processor.

- Data Processing Instructions
- Logical Instructions
- Comparison Instructions
- Branch Instructions

13. Write the operation carried out when CLZ instruction executed. *[Apr'18]*

- CLZ (Count Leading Zeros) instruction returns the number of 0 bits at the most significant end of its operand before the first 1 bit is encountered.
- It is used to locate the highest priority bit in a bit mask

- It is used to determine how many bits the operand should be shifted left to normalize it, so that its most significant bit is 1.

14. What is load-Store Instructions?

- Load store instruction transfer data between memory and processor registers.
- There are three types of load-store instructions:
    - single –register transfer,
    - multiple register transfer
    - swap

15. What is Stack Operations?

- The ARM architecture uses the load-store multiple instructions to carry out stack operations.
- The pop operation (removing data from a stack) uses a load multiple instruction; similarly, the push operation (placing data onto the stack) uses a store multiple instruction.

16. What is the use of software Interrupt Instruction?

- A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.

17. What is conditional Execution?

- Most ARM instructions are conditionally executed; we can specify that the instruction only executes if the condition code flags pass a given condition or test.
- By using conditional execution instructions you can increase performance and code density.

18. What are the most important features of coprocessor architecture?

- Support up to maximum of 16 logical coprocessors
- Each coprocessor can have up to 16 private registers of any reasonable size; they are not limited to 32 bits.
- Coprocessors use load-store architecture, with instructions to perform internal operations on registers, instructions to load and save registers from and to memory, and instructions to move data to or from an ARM register.

19. What are the two implications of ARM9 core?

- Coprocessor pipeline operates in step with the ARM core
- Coprocessor pipeline one cycle behind the ARM core.

20. List the coprocessor Instructions.

- Coprocessor registers
- Coprocessor Data operations
- Coprocessor Data transfers
- Coprocessor register transfers
- Coprocessor interface (ARM7TDMI)

21. List the different way of handling coprocessor handshake signals:

- The ARM may decide not to execute it, either because it falls in a branch shadow or fails its condition code test.
- The ARM may decide to execute it, but no present coprocessor can take.
- ARM decides to execute the instruction and a coprocessor accepts it, but cannot execute it yet.
- ARM decides to execute the instruction and a coprocessor accepts it for immediate execution.

22. What is the need of architectural support for high-level language?

- In general high-level languages allow a program to be expressed in terms of concepts such as data types, structures, procedures, functions, and so on.
- Since the RISC approach represents a movement away from instruction sets that attempt to support these high-level concepts directly.
- In this case use C as the example high-level language and the ARM instruction set as the architecture that the language is compiled onto.

23. Differentiate between ARM assembly-level languages and High-level languages.

- A programmer who writes at the assembly programming level directly with the raw machine instruction set, expressing the program in terms of instructions, addresses, registers, bytes and words.
- A high-level language allows the programmer to think in terms of abstractions that are above the machine level; indeed, the programmer may not even know on which machine the program will ultimately run.

24. What are the characteristics of data types?

- Generally a computer data type can be characterized as:
  - The number of bits required;
  - The ordering of bits;
  - The uses to which the group of bits is put.

25. What are the simplest way to view breaks in the ARM pipeline

- All instructions occupy the datapath for one or more adjacent cycles.
- For each cycle that an instruction occupies the datapath, it occupies the decode logic in the immediately preceding cycle.
- During the first datapath cycle each instruction issues a fetch for the next instruction but one.
- Branch instructions flush and refill the instruction pipeline.

26. What is the role of a Coprocessor? *(Apr'17)*[OR]**What are the important features of Coprocessor architecture?**

- Support for up to 16 logical coprocessors.
- Each coprocessor can have up to 16 private registers of any reasonable size; they are not limited to 32 bits.
- Coprocessors use a load-store architecture, with instructions to perform internal operations on registers, instructions to load and save registers from and to memory, and instructions to move data to or from an ARM register.

27. What are three signals of coprocessor interface?

- **CPI (from ARM to all coprocessors).**
  This signal, which stands for 'Coprocessor Instruction', indicates that the ARM has identified a coprocessor instruction and wishes to execute it.
- **CPA (from the coprocessors to ARM).**
  This is the 'Coprocessor Absent' signal which tells the ARM that there is no coprocessor present that is able to execute the current instruction.
- **CPB (from the coprocessors to ARM).**
  This is the 'Co-Processor Busy' signal which tells the ARM that the coprocessor cannot begin executing the instruction yet.

28. How the datatypes of ARM characterized?

- A computer data type can therefore be characterized by:
  - The number of bits it requires;
  - The ordering of those bits;
  - The uses to which the group of bits is put.

29. What are the different Data types in ARM?

- Numbers
- Roman numerals:
- Decimal numbers:
- Binary coded decimal:
- Binary notation
- Hexadecimal notation
- Number ranges
- Signed integers

30. What are the ANSI C basic data types?

- Signed and unsigned **characters** of at least eight bits.
- Signed and unsigned **short integers** of at least 16 bits.
- Signed and unsigned **integers** of at least 16 bits.
- Signed and unsigned **long integers** of at least 32 bits.
- **Floating-point, double** and **long double** floating-point numbers.
- **Enumerated** types.
- **Bitfields.**

31. Explain ANSI C derived data types

- **Arrays** of several objects of the same type.
- **Functions** which return an object of a given type.
- **Structures** containing a sequence of objects of various types.
- **Pointers** (which are usually machine addresses) to objects of a given type.

32. Write FPA10 pipeline stages?

- The FPA10 arithmetic unit operates in four pipeline stages:
  - Prepare: align operands.
  - Calculate: add, multiply or divide.
  - Align: normalize the result.
  - Round: apply appropriate rounding to the result.

33. What are the Conditional statements used in ARM programs.

- if...else
- Switches

34. Different types of loop structure used in program.

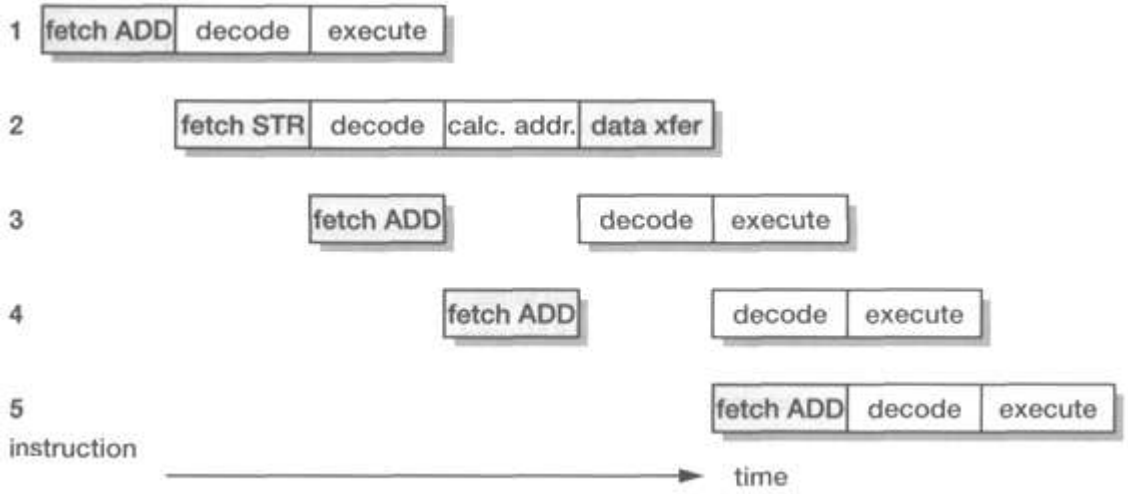
- for loop
- while loop
- do..while loops

- The C language supports three forms of loop control structure:
  - for (el;e2;e3){..}
  - while (el) {..}
  - do {..} while (el)
- Here el, e2 and e3 are expressions which evaluate to 'true' or 'false' and {..} is the body of the loop which is executed a number of times determined by the control structure.

35. List few Embedded ARM Applications for ARM Processor. *(Nov'16)* [OR] **Give the details about the real time embedded ARM applications.** *[Nov'17]*

- The VLSI Ruby II Advanced Communication Processor
- The VLSI ISDN Subscriber Processor
- The One C™ VWS22100 GSM chip
- The Ericsson-VLSI Bluetooth Baseband Controller
- The ARM7500 and ARM7500FE
- TheARM7100
- TheSA-1100

36. Draw the structure of multicycle instructions of three stage pipeline operation? *(Apr'17)*

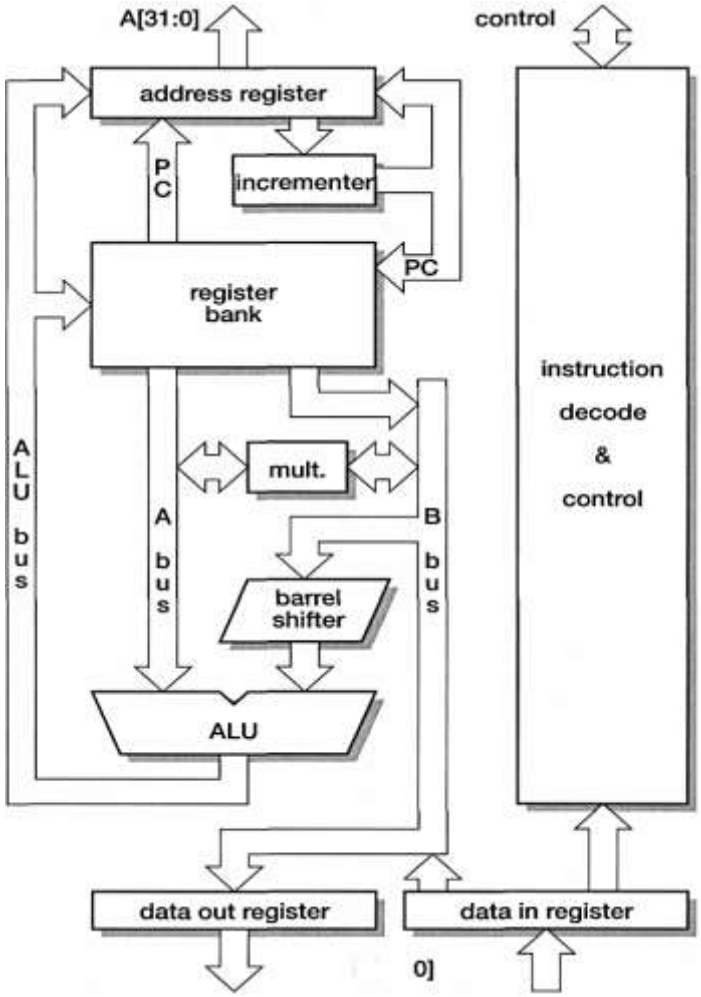


PART B

1. Explain briefly about 3-stage pipeline ARM organization.

- The organization of an ARM with a 3-stage pipeline is illustrated in figure below. Three Stages are:

1. **Fetch 2.Decode 3. Execute**.



3-stage pipeline ARM organization.

Pipeline stages:

1. **Fetch:**

The instruction is fetched from memory and placed in the instruction pipeline.

2. Decode:

The instruction is decoded and the datapath control signals prepared for the nextcycle. In this stage the instruction 'owns' the decode logic but not the datapath.

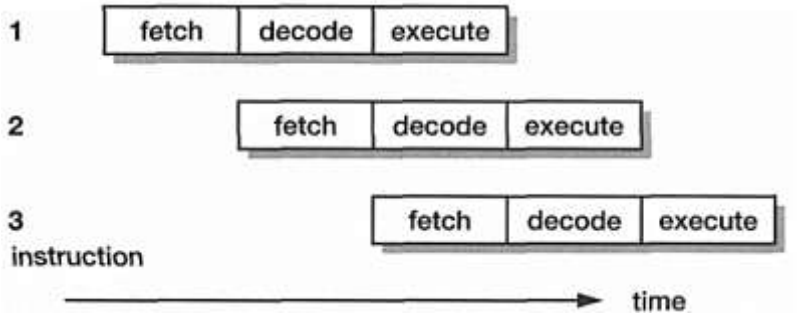
3. Execute:

The instruction 'owns' the datapath; the register bank is read, an operand shifted,the ALU result generated and written back into a destination register.

- At any one time, three different instructions may occupy each of these stages, so the hardware in each stage has to be capable of independent operation.
- When the processor is executing simple data processing instructions the pipeline enables one instruction to be completed every clock cycle.
- An individual instruction takes three clock cycles to complete, so it has a three-cycle latency, but the throughout is one instruction per cycle.

Single-cycle instructions

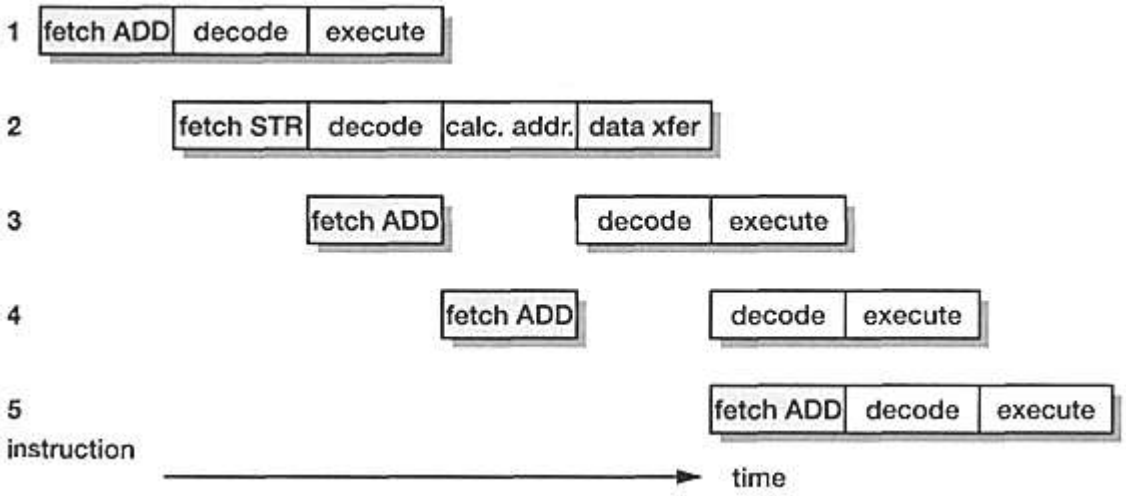
- The 3-stage pipeline operation for single-cycle instructions is shown in figure.



ARM single-cycle instruction 3-stage pipeline operation.

Multi-cycle instructions

- When a multi-cycle instruction is executed the flow is less regular, as illustrated in figure below.
- It shows a sequence of single-cycle ADD instructions with a data store instruction, STR, occurring after the first ADD.
- The cycles that access main memory are shown with light shading so it can be seen that memory is used in every cycle.



ARM multi-cycle instruction 3-stage pipeline operation

- The datapath is likewise used in every cycle, being involved in all the execute cycles, the address calculation and the data transfer.
- The decode logic is always generating the control signals for the datapath to use in the next cycle, so in addition to the explicit decode cycles it is also generating the control for the data transfer during the address calculation cycle of the STR.
- Thus, in this instruction sequence, all parts of the processor are active in every cycle and the memory is the limiting factor, denning the number of cycles the sequence must take.

The simplest way to view breaks in the ARM pipeline is to observe that:

- During the first datapath cycle each instruction issues a fetch for the next instruction but one.
- Branch instructions flush and refill the instruction pipeline.
- All instructions occupy the datapath for one or more adjacent cycles.

- For each cycle that an instruction occupies the datapath, it occupies the decode logic in the immediately preceding cycle.

PC behaviour:

- The program counter, which is visible to the user as r15, must run ahead of the current instruction.
- Instructions fetch the next instruction but one during their first cycle, this suggests that the PC must point eight bytes (two instructions) ahead of the current instruction.
- The programmer who attempts to access the PC directly through r15 must take account of the exposure of the pipeline here.
- However, for most normal purposes the assembler or compiler handles all the details.
- Even more complex behaviour is exposed if r15 is used later than the first cycle of an instruction.
- The instruction will itself have incremented the PC during its first cycle.
- Such use of the PC is not often beneficial so the ARM architecture definition specifies the result as 'unpredictable' and it should be avoided.

2. Explain the 5 – stage pipeline ARM cross – development tool kit. *[Apr'18]*

- All processors have to develop to meet the demand for higher performance. The 3-stage pipeline used in the ARM cores up to the ARM is very cost-effective, but higher performance requires the processor organization to be thought.
- The time, T , required to execute a given program is given by:

$$T_{prog} = \frac{N_{inst} \times CPI}{f_{clk}}$$

- Where N_{inst} is the number of ARM instructions executed in the course of the program, CPI is the average number of clock cycles per instruction and F_{clk} is the processor's clock frequency.
- Since N_{inst} is constant for a given program, there are only two ways to increase performance to Increase the clock rate, f_{clk}
- It requires the logic in each pipeline stage to be simplified and, therefore, the number of pipeline stages to be increased.
- Reduce the average number of clock cycles per instruction, [CPI].
- This requires either that instructions which occupy more than one pipeline slot in a 3-stage pipeline ARM are re-implemented to occupy fewer slots, or that pipeline stalls caused by dependencies between instructions are reduced, or a combination of both.

Memory bottleneck:

- The fundamental problem with reducing the CPI relative to a 3-stage core is related to the von Neumann bottleneck - any stored-program computer with a single instruction and data memory will have its performance limited by the available memory bandwidth.
- A 3-stage ARM core accesses memory on (almost) every clock cycle either to fetch an instruction or to transfer data.
- To get a significantly better CPI the memory system must deliver more than one value in each clock cycle either by delivering more than 32 bits per cycle from a single memory or by having separate memories for instruction and data accesses.
- The above issues, higher performance ARM cores employ a 5-stage pipeline and have separate instruction and data memories.

- Breaking instruction execution down into five components rather than three reduces the maximum work which must be completed in a clock cycle, and hence allows a higher clock frequency to be used.
- The separate instruction and data memories allow a significant reduction in the core's CPI.
- A typical 5-stage ARM pipeline is that employed in the ARM9TDMI. The organization of the ARM9TDMI is illustrated in Figure below

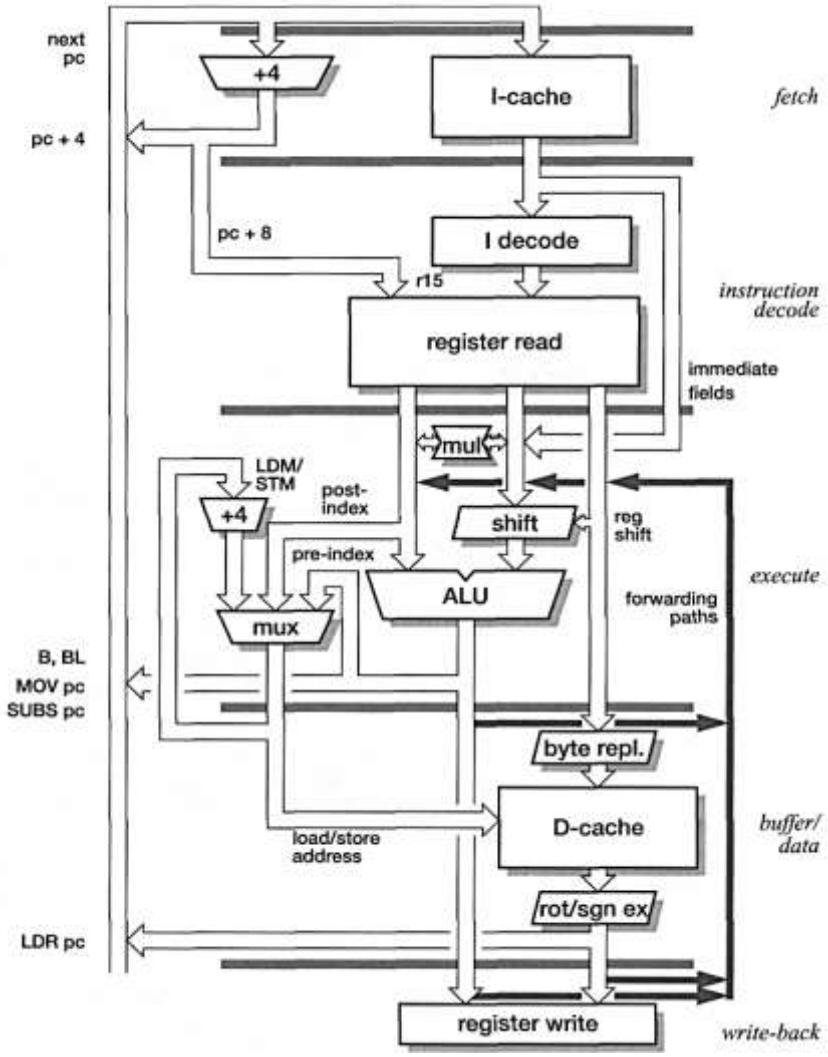


Figure 4.4 ARM9TDMI 5-stage pipeline organization.

The 5-stage pipeline:

The ARM processors which use a 5-stage pipeline have the following pipeline stages:

1. Fetch:

- The instruction is fetched from memory and placed in the instruction pipeline.

2. Decode:

- The instruction is decoded and register operands read from the register file.
- There are three operand read ports in the register file, so most ARM instructions can source all their operands in one cycle.

3. Execute:

- An operand is shifted and the ALU result generated.
- If the instruction is a load or store the memory address is computed in the ALU.

4. Buffer/data:

- Data memory is accessed if required.
- Otherwise the ALU result is simply buffered.
- For one clock cycle to give the same pipeline flow for all instructions

5. Write-back:

- The results generated by the instruction are written back to the register file, including any data loaded from memory.

- This 5-stage pipeline has been used for many RISC processors and is considered to be the 'classic' way to design such a processor.
- The principal concession store the ARM instruction set architecture in the organization shown in Figure, are the three source operand read ports and two write ports in the register file (where a 'classic' RISC has two read ports and one write port), and the inclusion of address incrementing hardware in the execute stage to support load and store multiple instructions.

Data forwarding:

- A major source of complexity in the 5-stage pipeline (compared to the 3-stage pipeline) is that, because instruction execution is spread across three pipeline stages, the only way to resolve data dependencies without stalling the pipeline is to introduce *forwarding* paths.
- Data dependencies arise when an instruction needs to use the result of one of its predecessors before that result has returned to the register file.
- Forwarding paths allow results to be passed between stages as soon as they are available, and the 5-stage ARM pipeline requires each of the three source operands to be forwarded from any of three intermediate result registers as shown in Figure.
- Consider the following code sequence:

```
LDR   rN, [. . .]          ; load rN from somewhere
ADD   r2, r1, rN           ; and use it immediately
```

- The processor cannot avoid a one-cycle stall as the value loaded into rN only enters the processor at the end of the buffer/data stage and it is needed by the following instruction at the start of the execute stage.
- The only way to avoid this stall is to encourage the compiler (or assembly language programmer) not to put a dependent instruction immediately after a load instruction.
- Since the 3-stage pipeline ARM cores are not adversely affected by this code sequence, existing ARM programs will often use it.
- Such programs will run correctly on 5-stage ARM cores, but could probably be rewritten to run faster by simply reordering the instructions to remove these dependencies.

PC generation

- The behaviour of r15 is based on the operational characteristics of the 3-stage ARM pipeline.
- The 5-stage pipeline reads the instruction operands one stage earlier in the pipeline, and would naturally get a different value (PC+4 rather than PC+8).
- As this would lead to unacceptable code incompatibilities, however, the 5-stage pipeline ARMs all 'emulate' the behaviour of the older 3-stage designs.

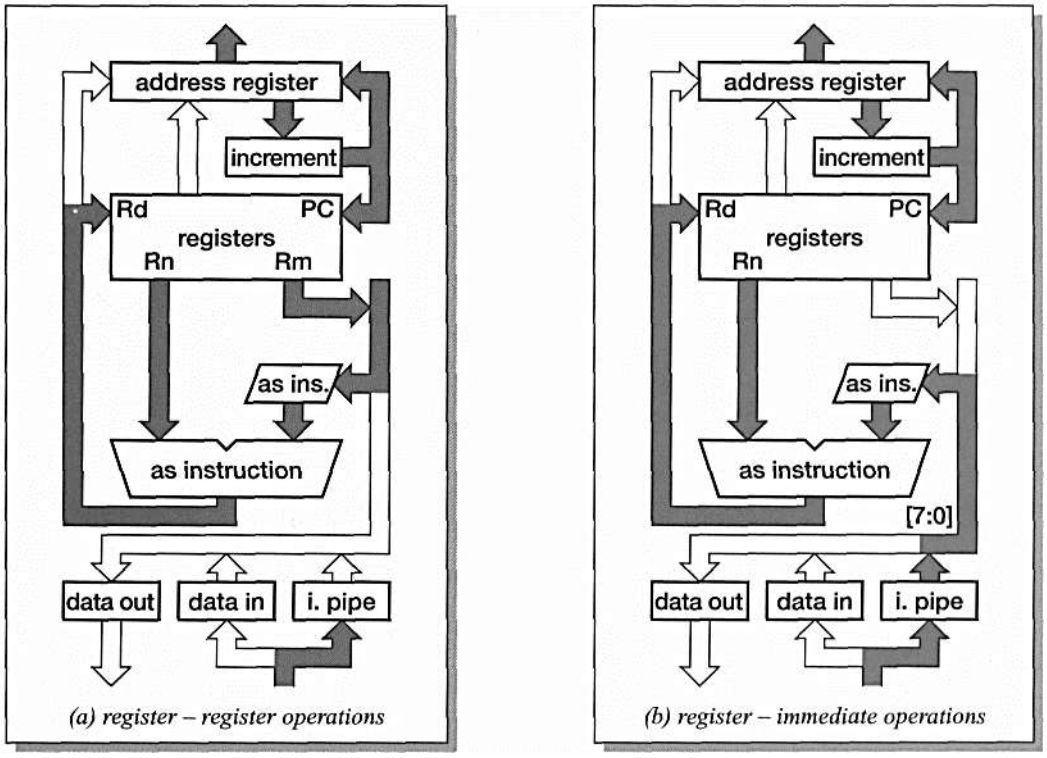
- The incremented PC value from the fetch stage is fed directly to the register file in the decode stage, bypassing the pipeline register between the two stages.
- PC+4 for the next instruction is equal to PC+8 for the current instruction, so the correct r15 value is obtained without additional hardware.

3. Explain briefly about ARM instruction execution.

- The execution of an ARM instruction can best be understood by reference to the datapath organization.
- We start with a simple data processing instruction:

Data processing instructions

- A data processing instruction requires two operands, one of which is always a register and the other is either a second register or an immediate value.
- The second operand is passed through the barrel shifter where it is subject to a general shift operation.
- Then it is combined with the first operand in the ALU using a general ALU operation.
- The result from the ALU is written back into the destination register.
- All these operations take place in a single clock cycle as shown in figure.



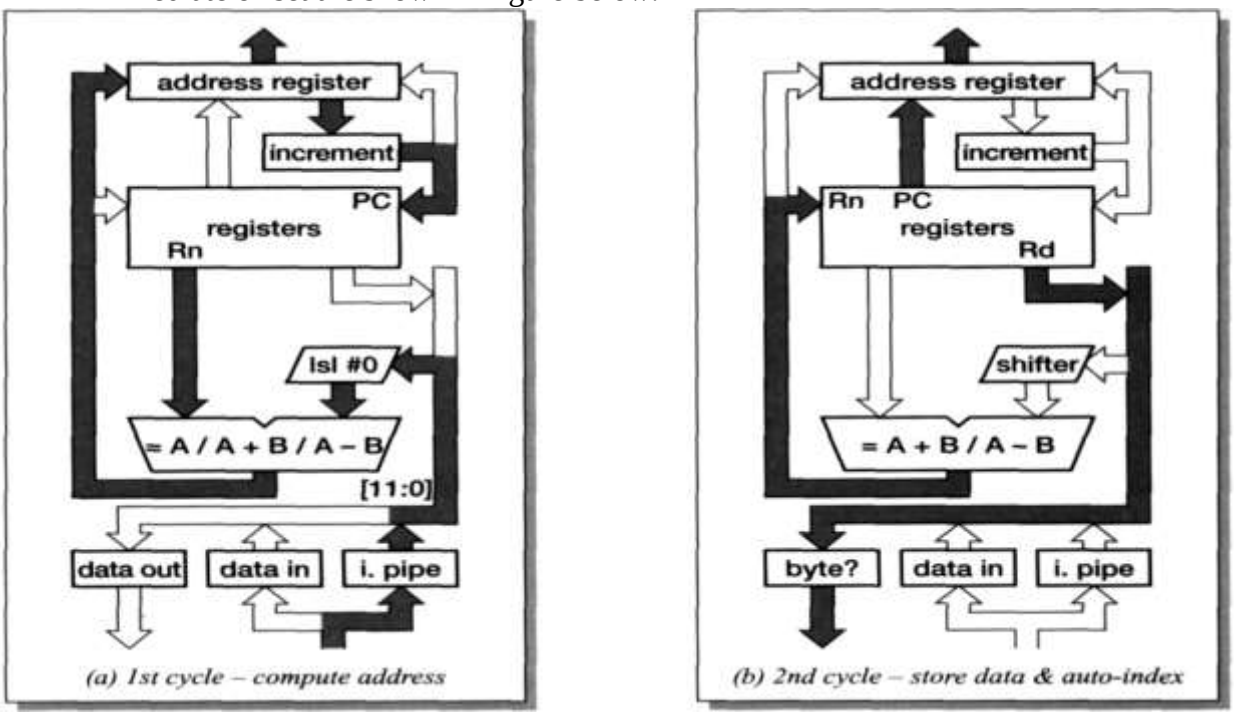
Data processing instruction datapath activity

- For data processing instructions only the bottom eight bits (bits [7:0]) of the instruction are used in the immediate value.

Data transfer instructions:

- A data transfer (load or store) instruction computes a memory address in a very similar to a data processing instruction computes its result.
- A register is used as the base address, to which is added and another which again may be another register or an immediate value.
- A 12-bit immediate value is used without a shift operation rather than a shifted 8-bit value.

- The address is sent to the address register, and in a second cycle the data transfer takes place.
- The ALU holds the address components from the first cycle and is available to compute an auto-indexing modification to the base register if this is required. (If auto-indexing is not required the computed value is not written back to the base register in the second cycle.)
- The datapath operation for the two cycles of a data store instruction (SIR) with an immediate offset are shown in figure below.



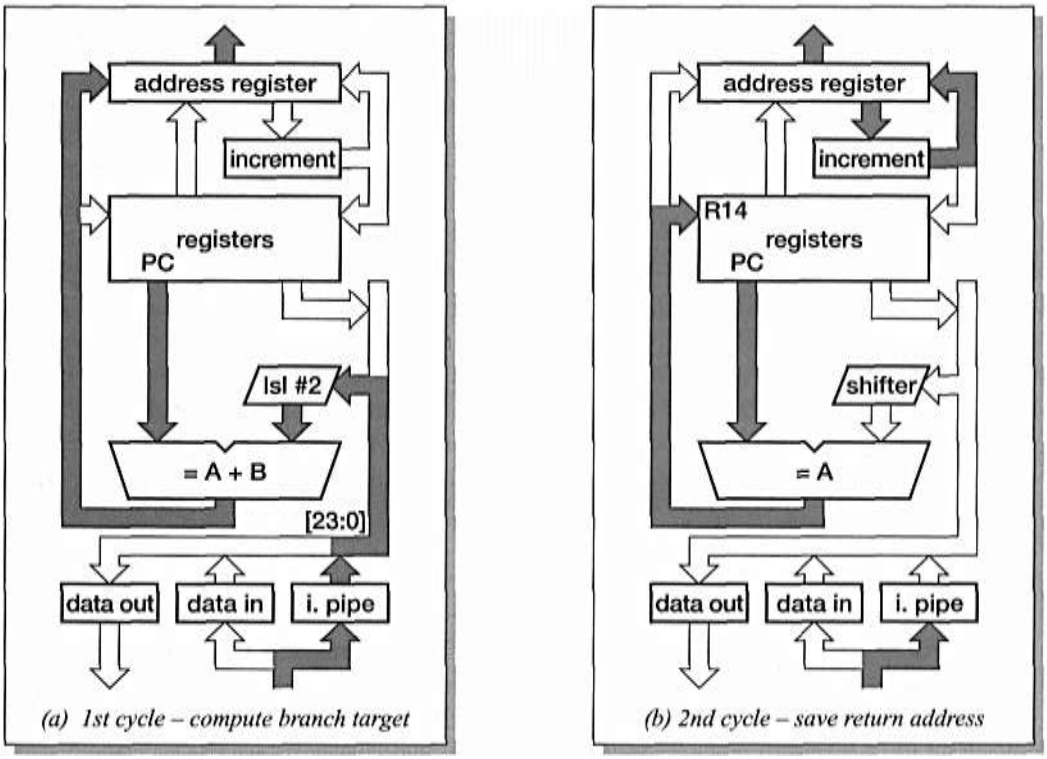
SIR (store register) datapath activity

- The address register is, in effect, a pipeline register between the processor datapath and the external memory.
- The address register can produce the memory address for the next cycle a little before the end of the current cycle, moving responsibility for the pipeline delay out into the memory when this is desired.
- This can enable some memory devices to operate at higher performance.
- When the instruction specifies the store of a byte data type, the 'data out' block extracts the bottom byte from the register and replicates it four times across the 32-bit data bus.
- External memory control logic can then use the bottom two bits of the address bus to activate the appropriate byte within the memory system.
- Load instructions follow a similar pattern except that the data from memory only gets as far as the 'data in' register on the second cycle and a third cycle is needed to transfer the data from there to the destination register.

Branch instructions

- Branch instructions compute the target address in the first cycle as shown in figure below.
- A 24-bit immediate field is extracted from the instruction and then shifted left two bit positions to give a word-aligned offset which is added to the PC.

- The result is issued as an instruction fetch address, and while the instruction pipeline refills the return address is copied into the link register (r14).



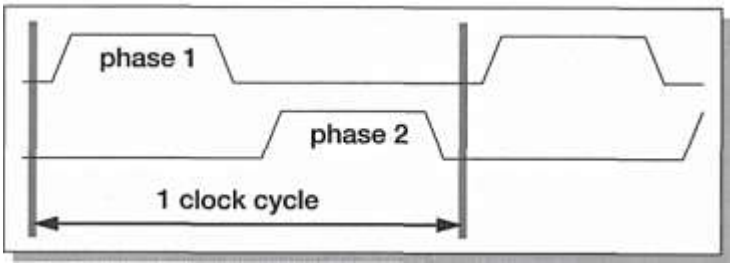
The first two (of three) cycles of a branch instruction.

- The third cycle, which is required to complete the pipeline refilling, is also used to make a small correction to the value stored in the link register in order that it points directly at the instruction which follows the branch.
- This is necessary because r15 contains pc + 8 whereas the address of the next instruction is pc + 4.

4. Explain ARM implementation.

Clocking scheme:

- Most ARMs do not operate with edge-sensitive registers; instead the design is based around 2-phase non-overlapping clocks, as shown in figure.

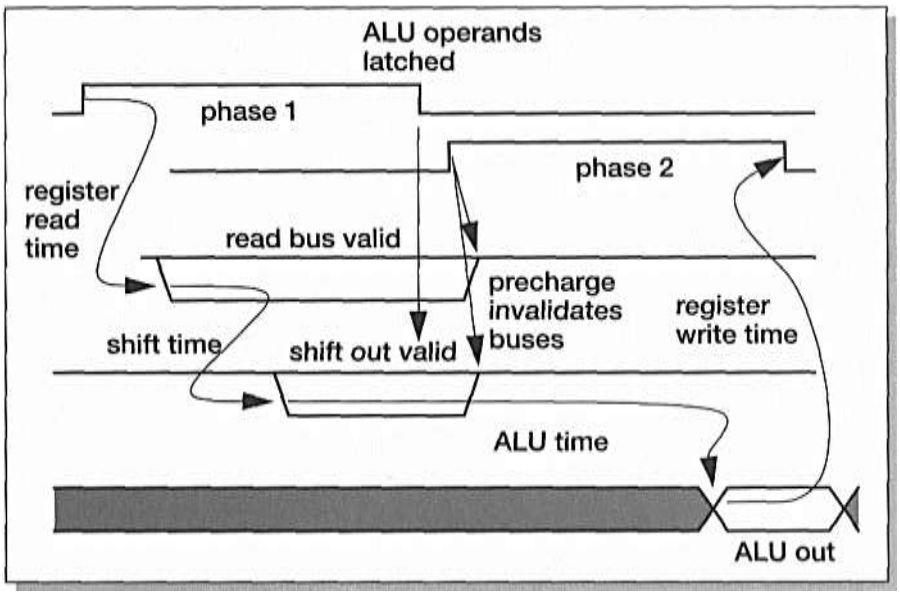


2-phase non-overlapping clock scheme.

- They are generated internally from a single input clock signal.
- This scheme allows the use of level-sensitive transparent latches.
- Data movement is controlled by passing the data alternately through latches which are open during phase 1 and phase 2.
- The non-overlapping property of the phase 1 and phase 2 clocks ensures that there are no race conditions in the circuit.

Datapath timing:

- The normal timing of the datapath components in a 3-stage pipeline is illustrated in figure.



ARM datapath timing (3-stage pipeline)

- The register read buses are dynamic and are precharged during phase 2.
- When phase 1 goes high, the selected registers discharge the read buses which become valid early in phase 1.
- One operand is passed through the barrel shifter, [dynamic techniques] and the shifter output becomes valid a little later in phase 1.
- The ALU has input latches which are open during phase 1, allowing the operands to begin combining in the ALU as soon as they are valid.
- But they close at the end of phase 1 so that the phase 2 precharge does not get through to the ALU.
- The ALU then continues to process the operands through phase 2, producing a valid output towards the end of the phase which is latched in the destination register at the end of phase 2.

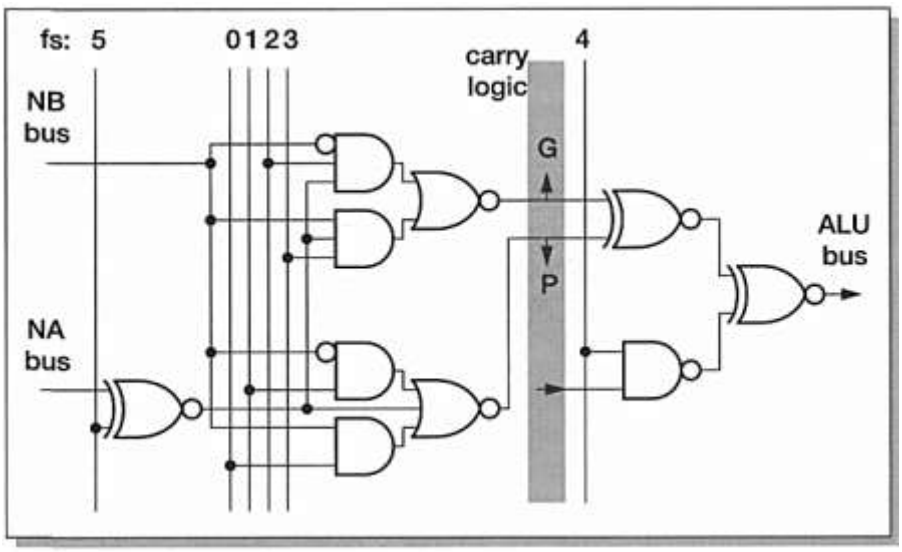
The minimum datapath cycle time is therefore the sum of:

- The register read time;
- The shifter delay;
- The ALU delay; [Highly variable, depending on the operating performance]
- The register write set-up time;
- The phase 2 to phase 1 non-overlap time.

- Logical operations are relatively fast, since they involve no carry propagation.
- Arithmetic operations (addition, subtraction and comparisons) involve longer logic paths as the carry can propagate across the word width.

ALU functions

- The ALU does not only add its two inputs. It must perform the full set of data operations defined by the instruction set, including address computations for memory transfers, branch calculations, bit-wise logical functions, and so on.
- The full ARM2 ALU logic is illustrated in figure below.



The ARM2 ALU logic for one result bit.

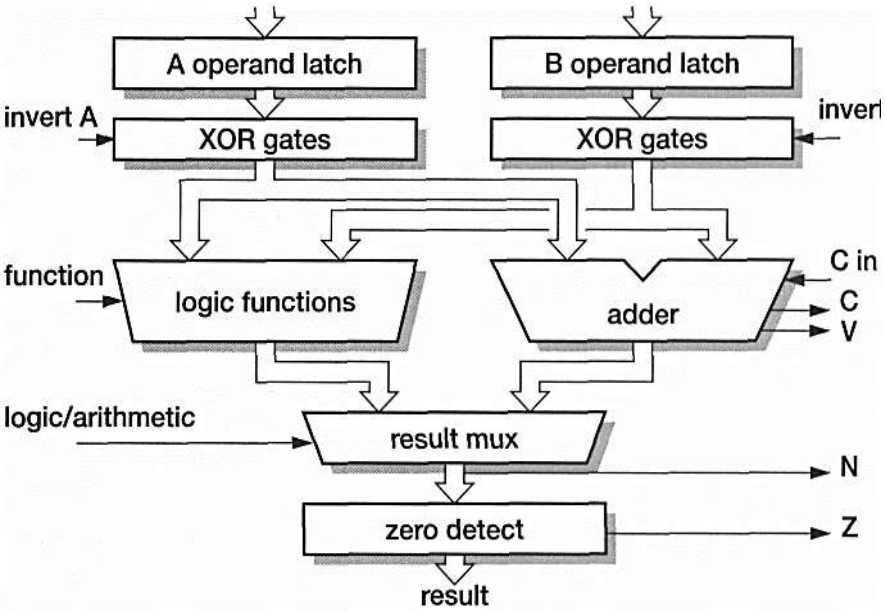
- The set of functions generated by this ALU and the associated values of the ALU function selects are listed in table below.

Table ARM2 ALU function codes

fs5	fs4	fs3	fs2	fs1	fs0	ALU output
0	0	0	1	0	0	A and B
0	0	1	0	0	0	A and not B
0	0	1	0	0	1	A xor B
0	1	1	0	0	1	A plus not B plus carry
0	1	0	1	1	0	A plus B plus carry
1	1	0	1	1	0	Not A plus B plus carry
0	0	0	0	0	0	A
0	0	0	0	0	1	A or B
0	0	0	1	0	1	B
0	0	1	0	1	0	Not B
0	0	1	1	0	0	Zero

ARM6 ALU structure:

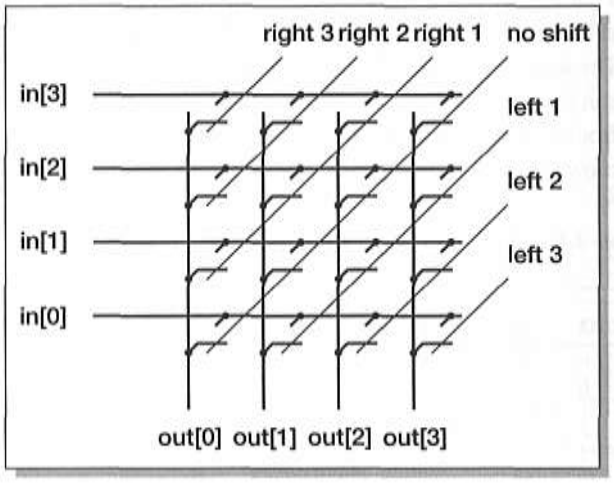
- A separate logic unit runs in parallel with the adder, and a multiplexer selects the output from the adder or from the logic unit as required.
- The overall ALU structure is shown in figure below.
- The input operands are each selectively inverted, then added and combined in the logic unit, and finally the required result is selected and issued on the ALU result bus.
- The *C* and *V* flags are generated in the adder
- The *N* flag is copied from bit 31 of the result and the *Z* flag is evaluated from the whole result bus.



The ARM6 ALU organization.

The barrel shifter:

- The ARM architecture supports instructions which perform a shift operation in series with an ALU operation.
- The shifter performance is therefore critical since the shift time contributes directly to the datapath cycle time.
- In order to minimize the delay through the shifter, a cross-bar switch matrix is used to steer each input to the appropriate output.
- The principle of the cross-bar switch is illustrated in figure below, where a 4 x 4 matrix is shown. (The ARM processors use a 32 x 32 matrix).



The cross-bar switch barrel shifter principle

- Each input is connected to each output through a switch.
- If pre-charged dynamic logic is used, as it is ON the ARM datapaths, each switch can be implemented as a single NMOS transistor.

.

The shifting functions are implemented by wiring switches along diagonals to a common control input:

- For a left or right shift function, one diagonal is turned on. This connects all the input bits to their respective outputs where they are used.

- In the ARM the barrel shifter operates in negative logic ' 1', potential near ground and a '0' a potential near the supply.
- Precharging sets all the outputs to a logic '0', so those outputs that are not connected to any input during a particular switching operation remain at '0' giving the zero filling required by the shift semantics.
- For a rotate right function, the right shift diagonal is enabled together with the complementary left shift diagonal.

Multiplier design

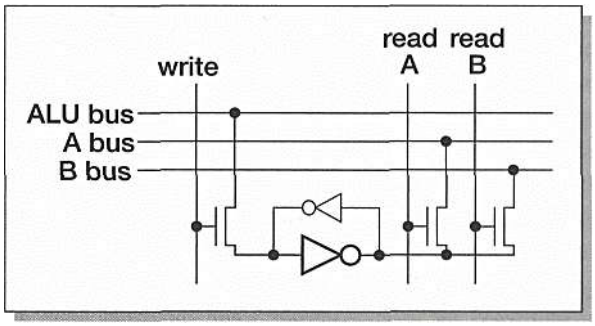
- All ARM processors apart from the first prototype have included hardware support for integer multiplication.
- Two styles of multiplier have been used:
  - Older ARM cores include low-cost multiplication hardware that supports only the 32-bit result multiply and multiply-accumulate instructions.
  - Recent ARM cores have high-performance multiplication hardware and support the 64-bit result multiply and multiply-accumulate instructions.
- The low-cost support uses the main data path iteratively, employing the barrel shifter and ALU to generate a 2-bit product in each clock cycle.
- The control settings for the N^{th} cycle of the multiplication are shown in Table below.
- Since this multiplication uses the existing shifter and ALU, the additional hardware it requires is limited to a dedicated two-bits-per-cycle shift register for the multiplier and a few gates for the Booth's algorithm control logic.

Table - 2-bit multiplication algorithm, N^{th} cycle

Carry - in	Multiplier	Shift	ALU	Carry - out
0	x 0	LSL #2N	A + 0	0
	x 1	LSL #2N	A + B	0
	x 2	LSL #(2N+1)	A – B	1
	x 3	LSL #2N	A – B	1
1	x 0	LSL #2N	A + B	0
	x 1	LSL #(2N+1)	A + B	0
	x 2	LSL #2N	A – B	1
	x 3	LSL #2N	A + 0	1

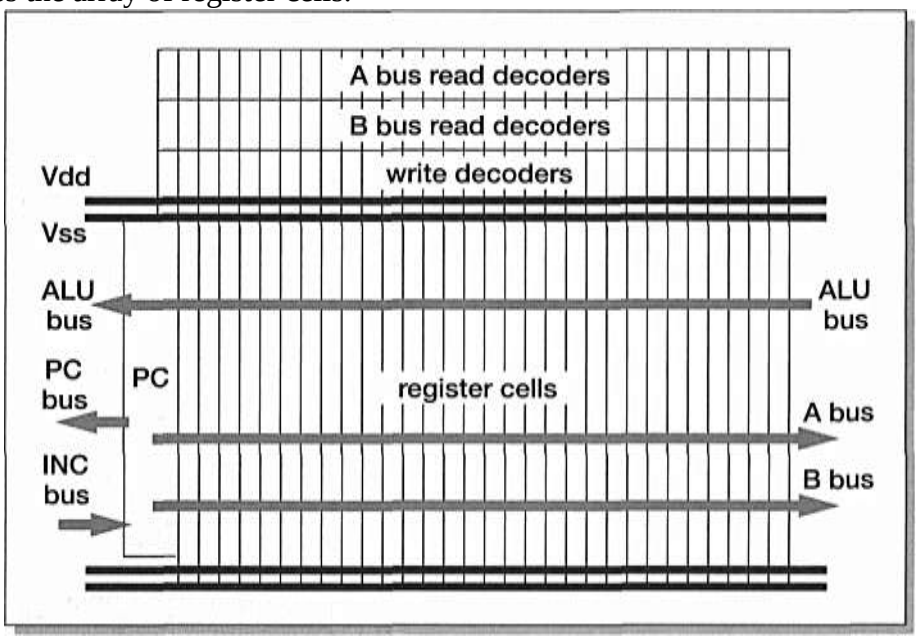
The register bank:

- The last major block on the ARM datapath is the register bank. This is where all the user-visible state is stored in 31 general-purpose 32-bit registers, amounting to around 1 Kbits of data altogether.
- The transistor circuit of the register cell used in ARM cores up to the ARM6 is shown in figure below.



ARM6 register cell circuit.

- The storage cell is an asymmetric cross-coupled pair of CMOS inverters which is overdriven by a strong signal from the ALU bus when the register contents are changed.
- The feedback inverter is made weak in order to minimize the cell's Resistance to the new value.
- The *A* and *B* read buses are precharged to *Vdd* during phase 2 of the clock cycle, so the register cell need only discharge the read buses, which it does through n-type pass transistors when the read-lines are enabled.
- This register cell design works well with a 5 volt supply, but writing a ' 1' through the n-type pass transistor becomes difficult at lower supply voltages.
- These register cells are arranged in columns to form a 32-bit register, and the columns are packed together to form the complete register bank.
- The decoders for the read and write enable lines are then packed above the columns as shown in figure below, so the enables run vertically and the data buses horizontally across the array of register cells.

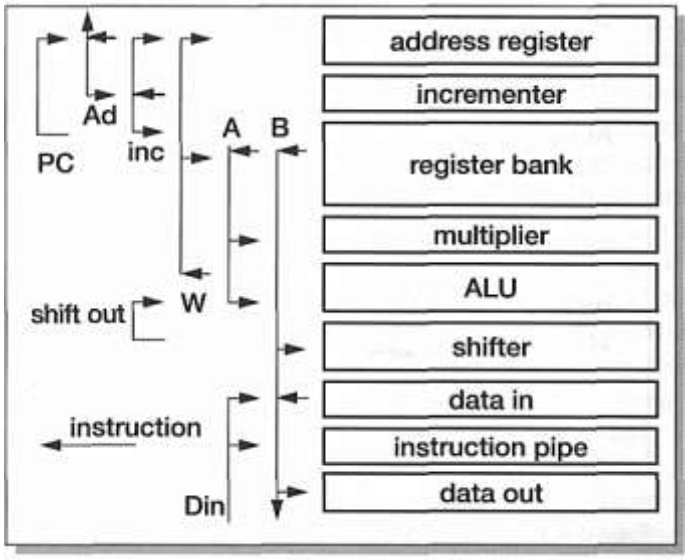


ARM register bank floor plan.

- Since a decoder is logically more complex than the register cell itself, but the horizontal pitch is chosen to suit the cell, the decoder layout can become very tight and the decoders themselves have to be tall and thin.

Datapath layout

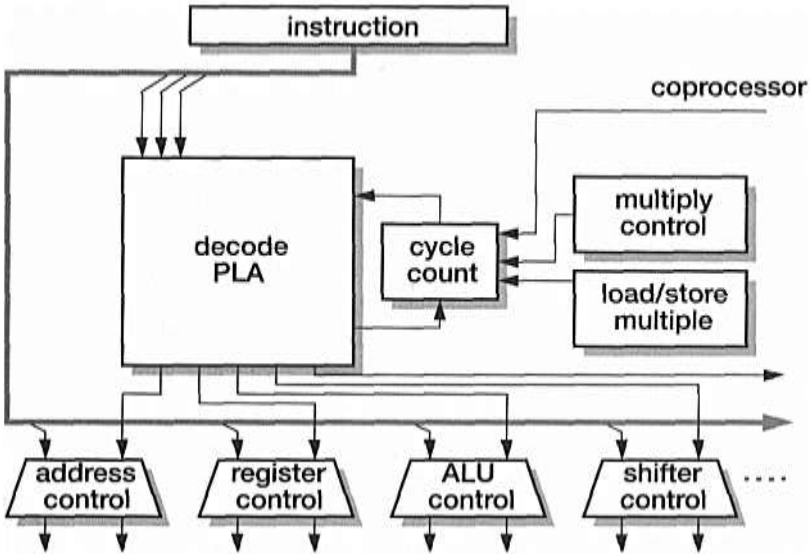
- The ARM datapath is laid out to a constant pitch per bit. The pitch will be a compromise between the optimum for the complex functions (such as the ALU) which are best suited to a wide pitch and the simple functions.
- The order of the function blocks is chosen to minimize the number of additional buses passing over the more complex functions.



ARM core datapath buses.

Control structures:

- The control logic on the simpler ARM cores has three structural components which relate to each other as shown in Figure below.



ARM control logic structure.

1. An instruction decoder PLA (programmable logic array).
2. Distributed secondary control associated with each of the major datapath function blocks.
3. Decentralized control units for specific instructions that take a variable number of cycles to complete

- The main decoder PLA has around 14 inputs, 40 product terms and 40 outputs, the precise number varying slightly between the different cores.
- ARM cores it is implemented as two PLAs: a small, fast PLA which generates the time critical outputs and a larger, slower PLA which generates all the other outputs.
- The 'cycle count' block distinguishes the different cycles of multi-cycle instructions so that the decode PLA can generate different control outputs for each cycle.

- It determines when the current instruction is about to complete and initiates the transfer of the next instruction from the instruction pipeline, including aborting instructions at the end of their first cycle if they fail their condition test.

5. Explain the instruction set of ARM Processor. *(Nov'16)* [OR] **Discuss on coprocessor data transfer instruction of ARM processor.** *[Apr'18]*

The different ARM architecture supports different instruction .They are:

1. Data Processing instruction
    a. Move instruction
    b. Barrel instruction
    c. Arithmetic
    d. Barrel shifter with arithmetic instruction
2. Logical instruction
3. Comparison instruction
4. Multiply instruction
5. Branch instruction
6. Load store instruction
7. Stack operation
8. Swap instruction

Mnemonics	**ARM ISA**	**Description**
ADC	vl	add two 32 bit values & carry
ADD	vl	add two 32 bit values
AND	vl	logical bitwise AND of two 32 bit values
B	vl	branch relative +/- 32MB
BIC	vl	logical bit clear of two 32 bit values
BKPT	v5	breakpoint instructions
BL	vl	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 zeroes
CMN	vl	compare negative two 32-bit values
CMP	vl	compare two 32 bit values
EOR	vl	logical EOR of two 32 bit values
LDCLDC2	v2 v5	load to coprocessor single or multiple 32 — bit values
LDM	vl	load multiple 32 — bit words from memory to ARM registers
LDR	vl v4 v5E	load a single value from a virtual address in memory
MCRMCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register to registers
MLA	v2	multiply & accumulate 32 bit values
MOV	vl	move a 32 bit value into a register
MRCMRC2 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	vl	move the logical NOT of 32 bit value into a register
ORR	vl	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 sub.
QSUB	v5E	32 bit values reverse subtract
RSB	vl	Reverse subtract of two 32 bit values
RSC	vl	Sub with carry of two 32 bit integers
SBC	vl	subtract with carry of two 32 bit values
SMLAxy	v5E	signed multiply accumulate instruction ((16 x 16) + 32 = 32 — bit)
SMLAL	V3M	signed multiply accumulate long ((32 x 32) + 64 = 64 — bit)
SMLALxy	v5E	signed multiply accumulate long ((32 x 16) + 64 = 64 — bit)
SMLAWy	v5E	signed multiply accumulate instruction ((32 x 16) » 16 +32 = 32 — bit)
SMULL		

6. Explain the ARM coprocessor interface [OR] **Explain how does the coprocessor interface of the ARM work.** *(Nov'16)* [OR] **Write short on coprocessor data and register transfer.** *(Apr'17)*

- The ARM supports a general-purpose extension of its instruction set through the addition of hardware coprocessors, and it also supports the software emulation of these coprocessors through the undefined instruction trap.

Coprocessor architecture:

- Its most important features are:
  - Support for up to 16 logical coprocessors.
  - Each coprocessor can have up to 16 private registers of any reasonable size; they are not limited to 32 bits.
  - Coprocessors use a load-store architecture, with instructions to perform internal operations on registers, instructions to load and save registers from and to memory, and instructions to move data to or from an ARM register.
- High clock speeds make board-level interfacing very difficult, so the higher-performance ARMs restrict the coprocessor interface to on-chip use.

ARM7TDMI coprocessor interface:

- The ARM7TDMI coprocessor interface is based on 'bus watching'.
- The coprocessor is attached to a bus where it copies the instructions into an internal pipeline that mimics the behaviour of the ARM instruction pipeline.
- As each coprocessor instruction begins execution there is a 'hand-shake' between the ARM and the coprocessor to confirm that they are both ready to execute it.
- The handshake uses three signals:
  1. **cpi -** Coprocessor Instruction **(from ARM to all coprocessors).**
     The ARM has identified a coprocessor instruction and wishes to execute it.
  2. **cpa -** Coprocessor Absent **(from the coprocessors to ARM).**
     ARM that there is no coprocessor present that is able to execute the current instruction.
  3. **cpb -** CoProcessor Busy **(from the coprocessors to ARM).**
     ARM that the coprocessor cannot begin executing the instruction yet.

- The timing is such that both the ARM and the coprocessor must generate their respective signals autonomously.
- The coprocessor cannot wait until it sees cpi before generating cpa and cpb.

Handshake outcomes:

- Once a coprocessor instruction has entered the ARM7TDMI and coprocessor pipelines, there are four Possible ways it may be handled depending on the handshake signals:
  1. The ARM may decide not to execute it.
  2. The ARM may decide to execute it.
  3. ARM decides to execute the instruction and a coprocessor accepts it, but cannot execute it yet.
  4. ARM decides to execute the instruction and a coprocessor accepts it for immediate execution, cpi, cpa and cpb are all taken low and both sides commit to complete the instruction.

Data transfers:

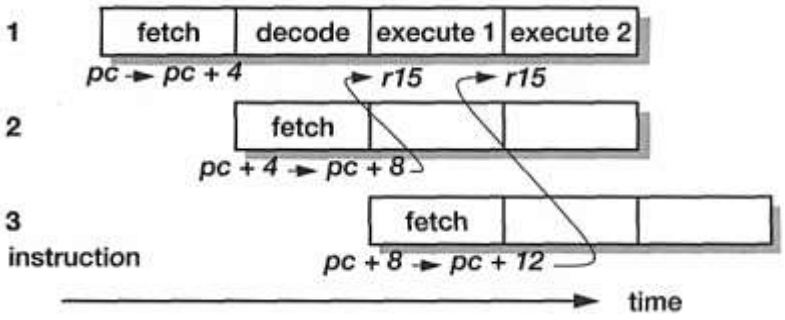
- If the instruction is a coprocessor data transfer instruction the ARM is responsible for generating an initial memory address but the coprocessor determines the length of the transfer.
- ARM will continue incrementing the address until the coprocessor signals completion.
- The cpa and cpb:
  - Handshake signals are also used for this purpose.
  - Since the data transfer is not interruptible once it has started, coprocessors should limit the maximum transfer length to 16 words

Pre-emptive execution

- A coprocessor may begin executing an instruction as soon as it enters its pipeline so long as it can recover its state if the handshake does not ultimately complete. All activity must be id em potent (repeatable with identical results) up to the point of commitment.

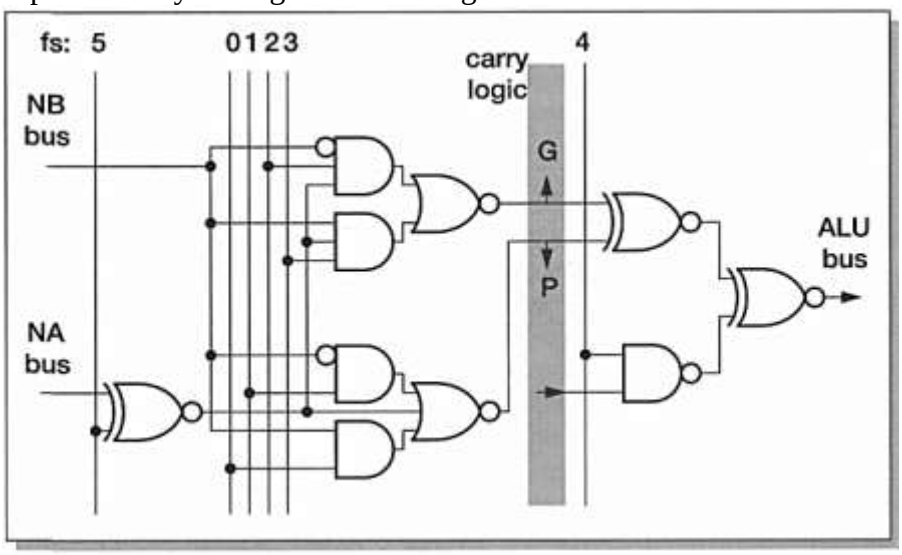
7. Why does r15 give pc + 8 in the first cycle of an instruction and pc + 12 in subsequent cycles on an ARM7?

- This is the ARM pipeline being exposed to the programmer.
- The pc value was incremented once when the current instruction (' 1' in the figure below) was fetched and once when its successor ('2') was fetched, giving pc + 8 at the start of the first execute cycle.
- During the first execute cycle a third instruction ('3') is fetched, giving pc + 12 in all subsequent execute cycles.
- While multi-cycle instructions interrupt the pipeline flow they do not affect this aspect of the behaviour.
- An instruction always fetches the next-instruction-but-one during its first execute cycle, so r15 always progresses from pc + 8 at the start of the first execute cycle to pc + 12 at the start of the second (and subsequent) execute cycle(s).



8. Complete the ARM2 4-bit carry logic circuit outlined.

- The 4-bit carry look-ahead scheme uses the individual bit carry generate and propagate Signals produced by the logic shown in figure below.



The ARM2 ALU logic for one result bit

- Denoting these by G[3:0] and P[3:0], the carry-out from the top bit of a 4-bit group is given by:

  Cout = G[3] + P[3].(G[2] + P[2].(G[1] + P[1].(G[0] + P[0].Cin)))

- Therefore the group generate and propagate signals, G4 and P4, as used in figure below are given by:

  G4 = G [3] + P [3]. (G [2] + P [2]. (G [1] + P [1].G [0]))

  P4 = P [3].P [2].P [1].P [0]

- These two signals are independent of the carry-in signal and therefore can be set up ready for its arrival, allowing the carry to propagate across the 4-bit group in just one AND-OR-INVERT gate delay.

9. Describe Architectural Support for High-level Languages

- There are many ways to implement the ARM architecture.
- If a program works correctly on one implementation, it should work correctly on them all.

Assembly-level abstraction:

- A programmer who writes the assembly programming level works directly with the raw machine instruction set, expressing the program in terms of instructions, addresses, registers, bytes and words.
- A good programmer, faced with a non-trivial task, will begin by determining higher levels of abstraction that simplify the program design.

- The rest of the program can then work just in terms of these end coordinates.
- Abstraction is important, then, at the assembly programming level, but all the responsibility for supporting the abstraction and expressing it in terms of the machine primitives rests with the programmer.

High-level languages

- A high-level language allows the programmer to think in terms of abstractions that are above the machine level; indeed, the programmer may not even know on which machine the program will ultimately run.
- Parameters such as the number of registers vary from architecture to architecture, so clearly these must not be reflected in the design of the language.
- The job of supporting the abstractions used in the high-level language on the target architecture falls upon the compiler.
- Compilers are themselves extremely complex pieces of software, and the efficiency of the code they produce depends to a considerable extent on the support that the target architecture offers them to do their job.

10. Explain data types in ARM.

A computer data type can therefore be characterized by:

- The number of bits it requires;
- The ordering of those bits;
- The uses to which the group of bits is put.

- Some data types, such as addresses and instructions, exist principally to serve the purposes of the computer, whereas others exist to represent information in a way that is accessible to human users.

Numbers: 0 to 9.
Roman numerals: I, V, X, L, M, C
Decimal numbers: 90.86, 0.016
Binary coded decimal: 0 to 9 [0000 to 1001]
Binary notation: 0 and 1
Hexadecimal notation: 0 to 9, A to F
Number ranges:

- The ARM deals efficiently with 32-bit quantities, so the first data type that the architecture supports is the 32-bit (unsigned) integer, which has a value in the range:

0 to 4 294 967 29510 = 0 to FFFFFFFF16

Signed integers:

- In many cases it is useful to be able to represent negative numbers as well as positive ones.
  -2 147 483 64810 to +2 147 483 64710 = 8000000016 to 7FFFFFFF, 6

Real numbers:

- 'Real' numbers are used to represent fractions and transcendental values that are useful when dealing with physical quantities.

Printable characters:

- The normal characters such as the upper and lower case alphabet, decimal digits from 0 to 9, punctuation marks and a number of special characters such as £, $, %, and etc.

ASCII:

- The 7-bit ASCII (American Standard for Computer Information Interchange) code, which includes these printable characters and a number of control codes

ARM support for characters:

- The support in the ARM architecture for handling characters is the unsigned byte load and store instructions.

Byte ordering:

- A character output routine might print characters at successive increasing byte addresses, in which case, with 'little-endian' addressing, it will print '5991'.
- The order of bytes within words in memory, (ARMs can operate with either little- or big-endian addressing.)

High-level languages:

- A high-level language defines the data types that it needs in its specification, usually without reference to any particular architecture that it may run on.

ANSI C basic data types:

- American National Standards Institute (ANSI), C' language are of the following basic data types:
    - Signed and unsigned **characters** of at least eight bits.
    - Signed and unsigned **short integers** of at least 16 bits.
    - Signed and unsigned **integers** of at least 16 bits.
    - Signed and unsigned **long integers** of at least 32 bits.
    - **Floating-point, double** and **long double** floating-point numbers.
    - **Enumerated** types. [variables have one value out of a specified set of possible values]
    - **Bitfields.** [Set of Boolean variables]

ANSI C derived data types:

- The ANSI C standard defines derived data types:
    - **Arrays** of several objects of the same type.
    - **Functions** which return an object of a given type.
    - **Structures** containing a sequence of objects of various types.
    - **Pointers** (which are usually machine addresses) to objects of a given type.
    - Unions which allow objects of different types to occupy the same space at different times.

ARM architectural support for C data types:

- The ARM integer core provides native support for signed and unsigned 32-bit integers and for unsigned bytes, covering the C integer, long integer and unsigned character types.
- Pointers are implemented as native ARM addresses and are therefore supported directly.

11. Explain the ARM floating point architecture. *[Apr'18]*

- Floating-point numbers attempt to represent real numbers with uniform accuracy. A generic way to represent a real number is in the form:

$$R = a \times b^n$$

Where n is chosen so that a falls within a defined range of values

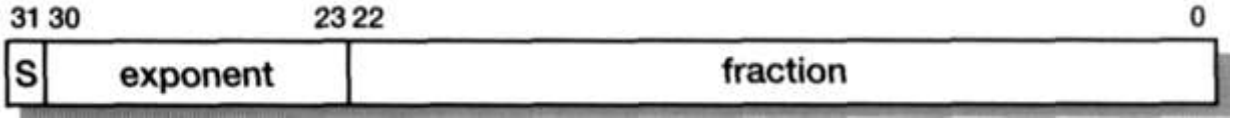
b is usually implicit in the data type and is often equal to 2.

IEEE 754:

- There are many complex issues to resolve with the handling of floating-point numbers in computers to ensure that the results are consistent when the same program is run on different machines.
- The consistency problem was aided by the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/ IEEE Standard 754-1985, referred as IEEE 754).

- 32-bit 'single precision' format:

Single precision:



IEEE 754 single precision floating-point number format.

- The number is made up from a **sign bit** ('S'),
- an **exponent** which is an unsigned integer value with a 'bias' of+127 (for normalized numbers)
- A **fractional** component.
    - The binary representation of 1995, 11111001011.
    - This is a positive number, so the S bit will be zero.

Normalized numbers:

- The first step is to **normalize** the number

$$1 < a < 2 \text{ and } b = 2$$

a can be constrained within this range by inserting a 'binary point', after the first' 1'.

- The normalized representation of 1995 is:

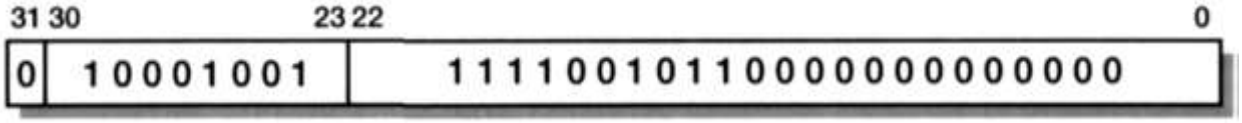
1995 = 1.111100101 Ix210

Where *a* and *n* are both in binary notation.

- When any number is normalized, the bit in front of the binary point in *a* will be a ' 1' (otherwise the number is not normalized).

Exponent bias:

- The bias (+127 for single precision normalized numbers) is added to the exponent value. Hence 1995 is represented as:



IEEE 754 single precision representation of '1995'.

- The exponent is 127+10 = 137; the fraction is zero-extended to the right to fill the 23-bit field.

Normalized value

- In general, the value of a 32-bit normalized number is given by:

value (norm) = (-l)S x l . fraction x 2(exponent"127)

- Although this format represents a wide range of values efficiently.
- It has one rather glaring problem: there is no way to represent zero. Therefore the IEEE 754 standard reserves numbers where the exponent is either zero or 255 to represent special values:
    - Zero is represented by a zero exponent and fraction
    - Plus or minus infinity are represented by the maximum exponent value with a zero fraction and the appropriate sign bit.
    - NaN (Not a Number) is indicated by the maximum exponent and a non-zero fraction; 'quiet' NaNs have a T in the most significant fraction bit position and 'signalling' NaNs have a '0' in that bit
    - Denormalized numbers, which are numbers that are just too small to normalize within this format, have a zero exponent, a non-zero fraction.

value (denorm) = (-l)s x O.fractionx2(-126)

Double precision:

- Greater accuracy may be achieved by using the double precision format which uses 64 bits to store each floating-point value.
- The exponent bias for normalized numbers is +1023:

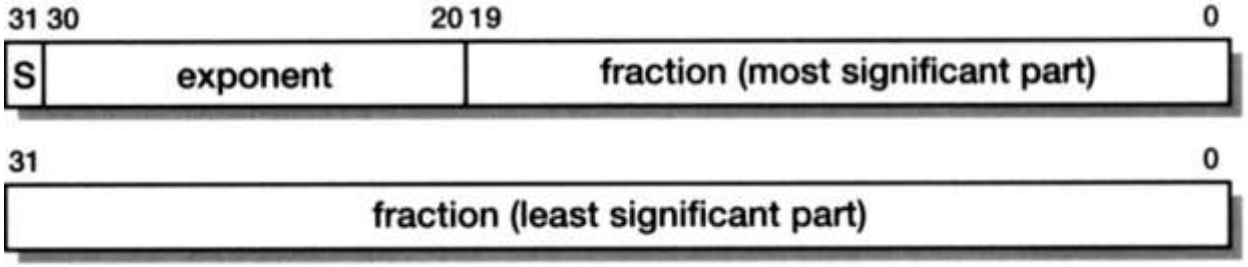


Figure 6.3 IEEE 754 double precision floating-point number format

Double extended precision:

- Even greater accuracy is available from the double extended precision format, which uses 80 bits of information spread across three words.
- The exponent bias is 16383, and the J bit is the bit to the left of the binary point (and is a T for all normalized numbers):

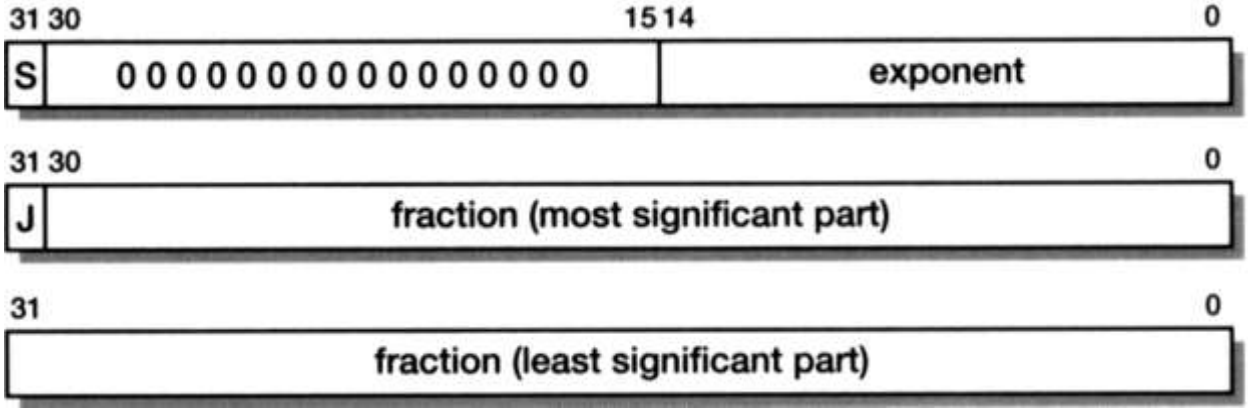
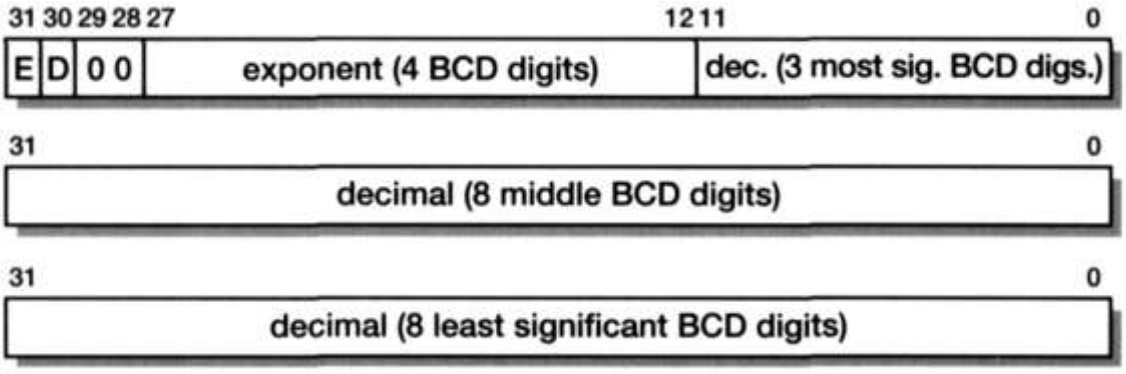


Figure 6.4 IEEE 754 double extended precision floating-point number format.

Packed decimal:

- In these packed formats *b* is 10 and *a*
- *n* are stored in a binary coded decimal format as described in 'Binary coded decimal'.
- The number is normalized so that 1 < a < 10.

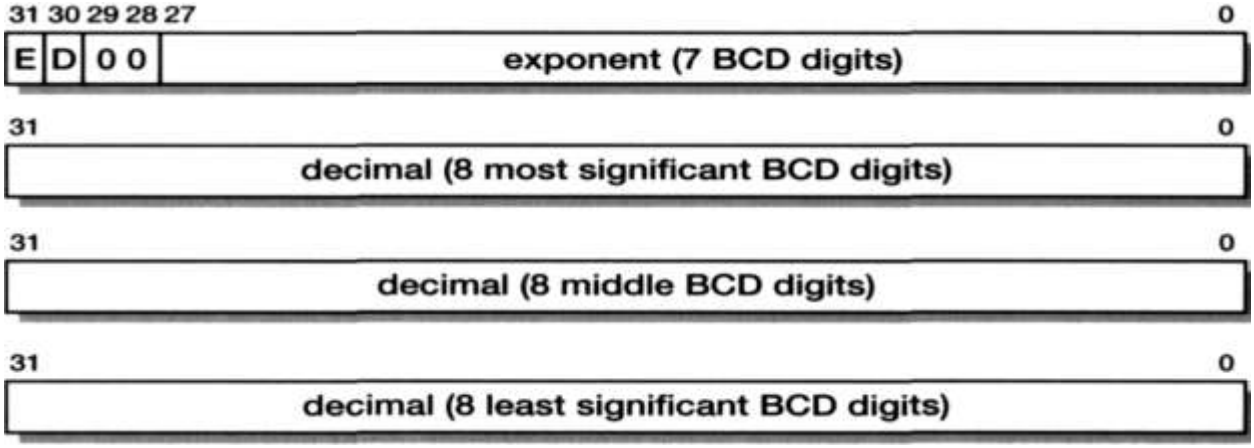


IEEE 754 packed decimal floating-point number format.

- The sign of the exponent is held in bit 31 of the first word ('E') and the sign of the decimal in bit 30 ('D'). The value of the number is:
  value (packed) = (-1)S x decimal x 1 o'"1>*x exponent)

Extended packed decimal:

The extended packed decimal format occupies four words to give higher precision.



IEEE 754 extended packed decimal floating-point number format

ARM floating point instructions:

- These instructions are normally implemented entirely in software through the undefined instruction trap but a subset may be handled in hardware by the FPA10 floating-point coprocessor.

ARM floating- point library:

- As an alternative to the ARM floating-point instruction set (and the only option for Thumb code), ARM Limited also supplies a C floating-point library which supports IEEE single and double precision formats.

12. The ARM floating-point architecture

The ARM floating-point architecture presents:

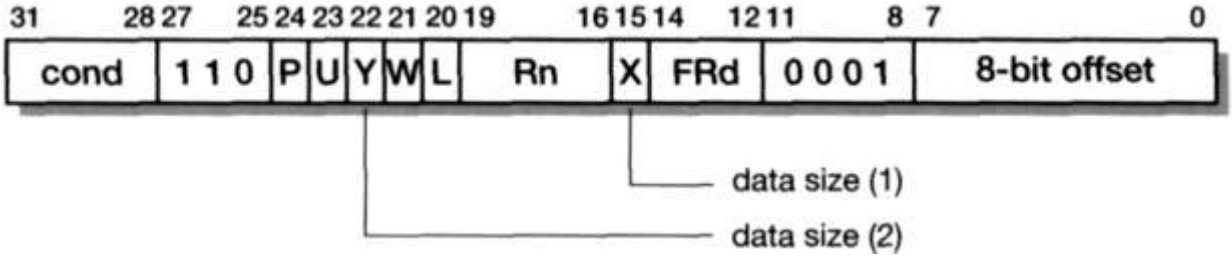
- An interpretation of the coprocessor instruction set when the coprocessor number is 1 or 2. (The floating-point system uses two logical coprocessor numbers.)
- Eight 80-bit floating-point registers in coprocessors 1 and 2 (the same physical registers appear in both logical coprocessors).
- A user-visible floating-point status register (FPSR) which controls various operating options and indicates error conditions.
- A floating-point control register (FPCR) which is user-invisible and should be used only by the support software specific to the hardware accelerator.

FPA10 data types

- The ARM FPA10 hardware floating-point accelerator supports single, double and extended double precision formats. Packed decimal formats are supported only by software.
- The coprocessor registers are all extended double precision, and all internal calculations are carried out in this.
- However loads and stores between memory and these registers can convert the precision as required.

Load and store floating instructions

- There are only eight floating-point registers, the register specifier field in the coprocessor data transfer instruction has a spare bit which is used here as an additional data size specifier:

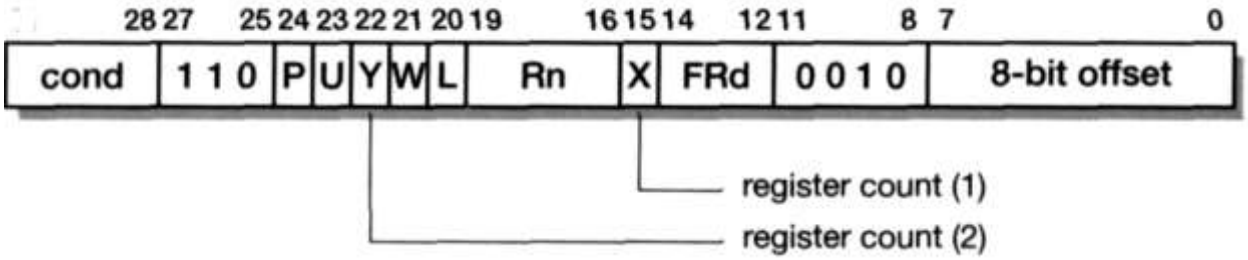


Load and store floating binary encoding.

- The X and Y bits allow one of four precisions to be specified, choosing between single, double, double extended and packed decimal.
- The choice between packed decimal and extended packed decimal is controlled by a bit in the FPSR.

Load and store multiple floating:

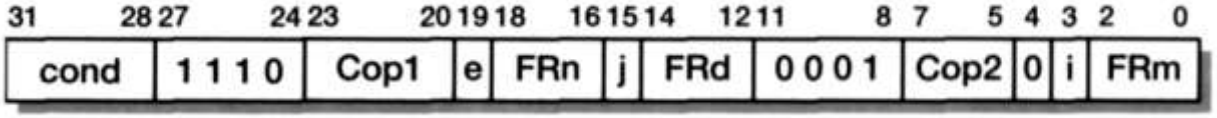
- The load and store multiple floating-point registers instructions are used to save and restore the floating-point register state.
- Each register is saved using three memory words and the precise format is not defined; it is intended that the only use for the saved values will be to reload them using the equivalent load multiple floating instruction to restore the context.
- 'FRd' → the first register to be transferred.
- 'X' and 'Y' encode the number of registers transferred which can be from one to four.



Load and store multiple floating binary encoding.

Floating-point data operations:

- The floating-point data operations perform arithmetic functions on values in the floating point registers; their only interaction with the outside world is to confirm that they should complete through the ARM coprocessor handshake.

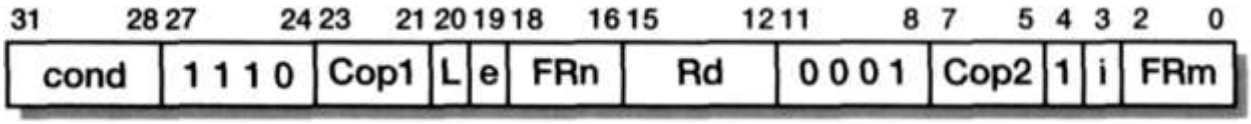


Floating-point data processing binary encoding.

- 'i' selects between a register ('FRm') or one of eight constants for the second operand.
- 'e' and 'Cop2' control the destination size and the rounding mode.
- 'j' selects between monadic (single operand) and dyadic (two operand) operations.

Floating-point register transfers:

- The register transfer instructions accept a value from or return a value to an ARM register. This is generally combined with a floating-point processing function.



Floating-point register transfer binary encoding.

- The floating-point compare instructions are special cases of this instruction type, where Rd is r15.
- Two floating-registers are compared and the result of the comparison is returned to the N, Z, C and V flags in the ARM CPSR where they can directly control the execution of conditional instructions:
  - N indicates 'less than'.
  - Z indicates equality.
  - C and V indicate more complex conditions, including 'unordered' comparison results which can arise when an operand is a 'NaN' (not a number).

Floating-point instruction frequencies:

❖ The design of the FPAIO is guided by the typical frequencies of the various floating point instructions.

Table Floating-point instruction frequencies

Instruction	Frequency (%)
Load/Store	67
Add	13
Multiply	10.5
Compare	3
Fix and float	2
Divide	1.5
Others	3

13. Write short note a Run-time environment.

- A C program requires an environment in which to operate; this is usually provided through a library of functions that the C program can call.
- In a PC or workstation a C programmer can expect to find the full ANSI C library, giving access to a broad range of functions such as file management, input and output (print f ()), the realtime clock, and so on.

Minimal run-time library:

❖ In a small embedded system such as a mobile telephone, most of these functions are irrelevant.

❖ ARM Limited supplies a minimal stand-alone run-time library which, once ported to the target environment, allows basic C programs to run. This library therefore reflects the minimal requirements of a C program.

❖ It comprises:

• Division and remainder functions.

Since the ARM instruction set does not include divide instructions, these are implemented as library functions.

• Stack-limit checking functions.

A minimal embedded system is unlikely to have memory management hardware available for stack overflow detection; therefore these library functions are needed to ensure programs operate safely.

• Stack and heap management.

All C programs use the stack for (many) function calls, and all but the most trivial create data structures on the heap.

• Program start up.

Once the stack and heap are initialized, the program starts with a call to main ().

• Program termination.

Most programs terminate by calling _exit (); even a program which runs forever should terminate if an error is detected.

- The total size of the code generated for this minimal library is 736 bytes, and it is implemented in a way that allows the linker to omit any unreferenced sections to reduce the library image to around half a kilobyte in many cases. This is a great deal smaller than the full ANSI C library.

Examples and exercises

Example 6.1 Write, compile and run a 'Hello World' program written in C.

The following program has the required function:

```
/* Hello World in C */
#include <stdio.h>
int main() {
printf( "Hello World\n" );
return
( 0 ); }
```

The principal things to note from this example are:

• The '#include' directive which allows this program to use all the standard input and output functions available in C.

• The declaration of the 'main' procedure. Every C program must have exactly one of these as the program is run by calling it.

• The 'printf (. .}' statement calls a function provided in stdio which sends output to the standard output device. By default this is the display terminal.

- As with the assembly programming exercises, the major challenge is to establish the flow through the tools from editing the text to compiling, linking and running the program.
- Once this program is working, generating more complex programs is fairly straightforward (at least until the complexity reaches the point where the design of the program becomes a challenge in itself).
- Using the ARM software development tools, the above program should be saved as 'HelloW.c'. Then a new project should be created using the Project Manager and this file added (as the only file in the project).
- A click on the 'Build' button will cause the program to be compiled and linked, then the 'Go' button will run it on the ARMulator, hopefully giving the expected output in the terminal window.

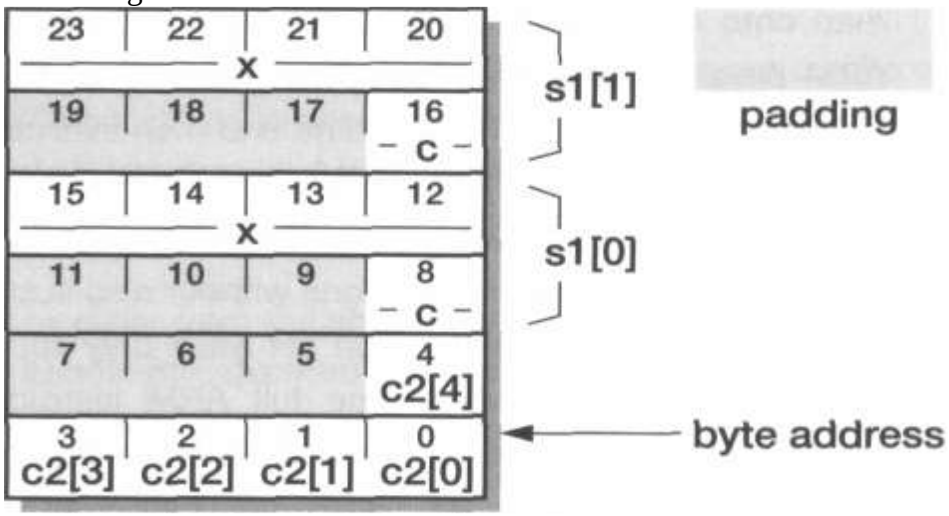
Example 6.2 Write the number 2001 in 32-bit binary, binary-coded decimal, ASCII and single-precision floating-point notation.

```
Binary: 2001    = 1024 + 512 + 256 + 128 + 64 + 16 + 1
                = 00000000000000000000011111010001₂
BCD:    2001    = 0010 0000 0000 0001
ASCII:  2001    = 00110010 00110000 00110000 00110001
F-P:    2001    = 1.1111010001 x 2^1010
                = 01001001 11110100 01000000 00000000
```

Example 6.3Show how the following data is organized in memory:

```
struct SI {char c; int x;}; struct
    S2 {
        char c2[5];
        SI si [2]; }
        example;
```

- The first structure statement only declares a type, so no memory is allocated. The second establishes a structure called 'example' comprising an array of five characters followed by an array of two structures of type SI.
- The structures must start on a word boundary, so the character array will be padded out to fill two words and each structure will also occupy two words. The memory organization is therefore as shown below:



16.List various Embedded ARM Applications in detail. Or Elaborate the working principle of the VLSI ISDN Subscriber Processor in detail.(Apr/may 17)

13.1 The VLSI Ruby II Advanced Communication Processor
13.2 The VLSI ISDN Subscriber Processor
13.3 The OneC™ VWS22100 GSM chip
13.4 The Ericsson-VLSI Bluetooth Baseband Controller
13.5 The ARM7500 and ARM7500FE
13.6 TheARM7100
13.7 TheSA-1100

16.1 The VLSI Ruby II Advanced Communication Processor

❖ VLSI Technology, Inc., were the first ARM semiconductor partner and were instrumental, along with Acorn Computers Limited and Apple Computer, Inc., in setting up ARM Limited as a separate company.

❖ Their relationship with the ARM predates the existence of ARM Limited, since they fabricated the very first ARM processors in 1985 and licensed the technology from Acorn Computers in 1987.

❖ VLSI have manufactured many standard ARM-based chips for Acorn Computers and produced ARM610 chips for Apple Newtons.

❖ They have also produced several ARM-based designs for customer specific products and a number which they have made available as standard parts.

❖ The Ruby II chip is one such standard part which is intended for use in portable communications devices.

Ruby IIorganization

❖ The organization of Ruby II is illustrated in Figure 13.1. The chip is based around an ARM core and includes 2 Kbytes of fast (zero wait state) on-chip SRAM. Critical routines can be loaded into the RAM under application control to get the best performance and minimum power consumption.

❖ There is a set of peripheral modules which share a number of pins, including a PCMCIA interface, four byte-wide parallel interfaces and two UARTs. A mode select block controls which combination of these interfaces is available at any time, and byte-wide FIFO buffers decouple the processor from having to respond to every byte which is transferred.

❖ A synchronous communications controller module supports a range of standard serial communication protocols, and a serial controller module provides a software-controlled data port which can be used to implement various serial control protocols such as the I2C bus defined by Philips which enables a range of serial devices such as battery-backed RAM, real-time clock, E2PROM and audio codecs to be connected.

❖ The external bus interface supports devices with 8-, 16- and 32-bit data buses and has flexible wait state generation.

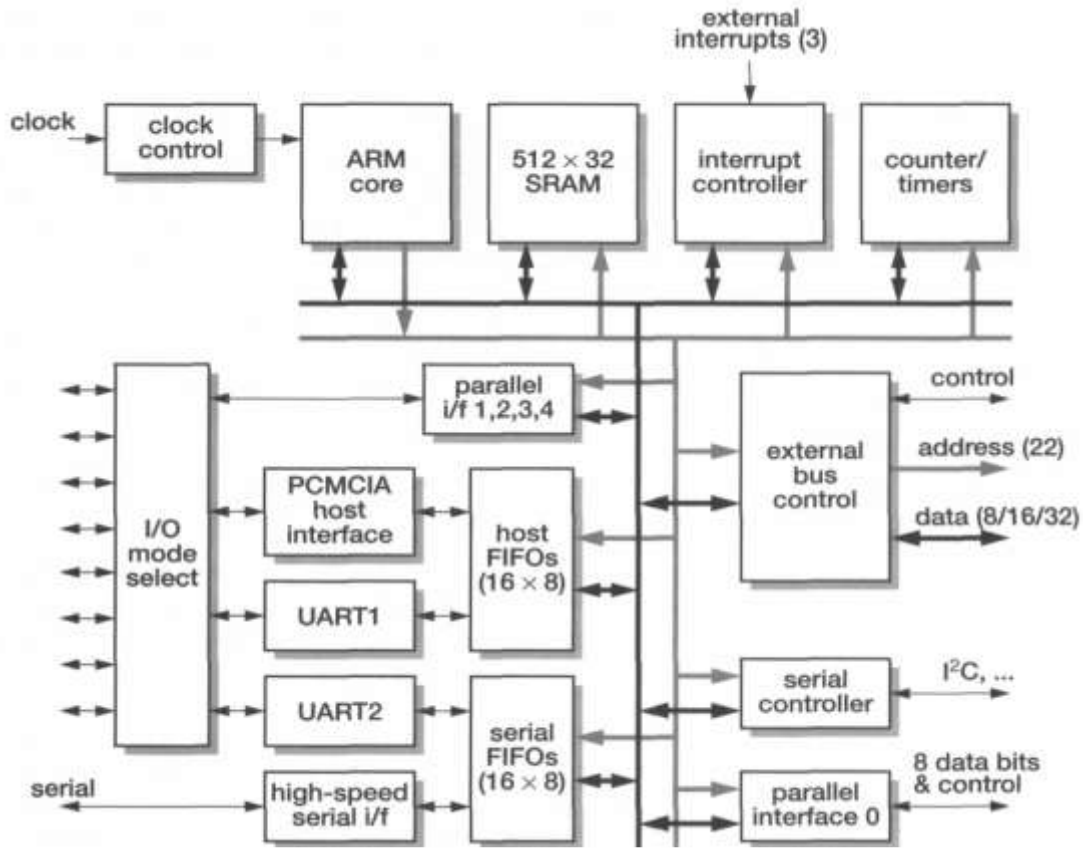
❖ The counter-timer block has three 8-bit counters connected to a 24-bit prescaler, and an interrupt controller gives programmable control of all on- and off-chip interrupt sources.

❖ The chip has four power-management modes:

1. On-line - all circuits are clocked at full speed.
2. Command - the ARM core runs with 1 to 64 wait states but all other circuitry runs at full speed. An interrupt switches the system into on-line mode immediately.
3. Sleep - all circuitry is stopped apart from the timers and oscillators. Particular interrupts return the system to on-line mode.
4. Stopped - all circuits (including the oscillators) are stopped. Particular interrupts return the system to on-line mode.

Packaging

❖ The Ruby II is available in 144- and 176-pin thin quad flat packs and can operate at up to 32 MHz at 5 volts. At 20 MHz using 32-bit 1 wait state memory the chip consumes 30 mA in on-line mode, 7.9 mA in command mode, 1.5 mA in sleep mode and 150 uA in stop mode.



13.2 The VLSI ISDN Subscriber Processor

- The VLSI ISDN Subscriber Processor (VIP) is a programmable engine for **ISDN** (Integrated Services Digital Network; a digital telephony standard) subscriber communications.
- The design was developed by Hagenuk GmbH for use in their ISDN product range and subsequently licensed back to VLSI Technology for sale as an **ASSP** (Application Specific Standard Part).
- It incorporates most of the circuitry required to implement a full-feature ISDN terminal, supporting voice, data and video services down the same digital line. The sort of applications it is targeted at include:

• ISDN terminal equipment, such as domestic and digital PABX telephones, H.320 videophones and integrated PC communications.
• ISDN to **DECT** (Digital European Cordless Telephone) controllers, allowing a number of cordless telephones to link to each other and to an ISDN line for domestic and business use.
• ISDN to PCMCIA communication cards.

- The VIP chip incorporates the specialized interfaces required to connect to the ISDN SO-interface, support for telephony interfaces such as a numeric keypad, a number display, a microphone and an earphone, digital links for external signal processors or codecs, and power management features such as a programmable clock and an analogue to digital converter to monitor the battery state.
- The ARM6 core performs general control and ISDN protocol functions. A 3 Kbyte on-chip RAM operates without wait states at the full processor clock rate of 36.864MHz. Critical code routines can be loaded into this RAM as required, for example, the signal processing routines required to support hands-free operation.

VIP organization

- The organization of the VIP chip is illustrated in Figure 13.2 and a typical system configuration is shown in Figure 13.3

Memory interface

- The external memory interface supports 8-, 16- and 32-bit off-chip static RAMs and ROMs and 16- and 32-bit dynamic RAMs.
- The addressable memory is divided into four ranges, each of which operates with a programmable number of wait states (where the minimum is one wait state giving a 54 ns access time).

SO-interface

- The on-chip ISDN SO-interface allows connection to an SO-interface bus via isolating transformers and surge protection. The on-chip functions include a phase-locked loop for data and clock recovery, framing, and low-level protocols.
- The 192 Kbit/s raw data rate includes two 64 Kbit/s B channels and one 16 Kbit/s D channel.
- In telephony applications the B channels carry 8-bit speech samples at an 8 KHz sample rate and the D channel is used for control purpose

Codec

- The G.711 codec includes an on-chip analogue front end that allows direct connection to both a telephone handset and a hands-free microphone and speaker.
- The input and output channels have independently programmable gains. The amplification stages have power-down modes to save power when they are inactive.

ADCs

- The on-chip analogue to digital converters are based upon timing how long it takes to discharge a capacitor to the input voltage level.
- This is a very simple way to measure slowly varying voltages, requiring little more than an on-chip comparator, an output to charge the capacitor at the start of the conversion and a means of measuring the time from the start of the conversion to the point where the comparator switches.
- Typical uses would be to measure the voltage from a volume control potentiometer or to check the battery voltage in a portable application.

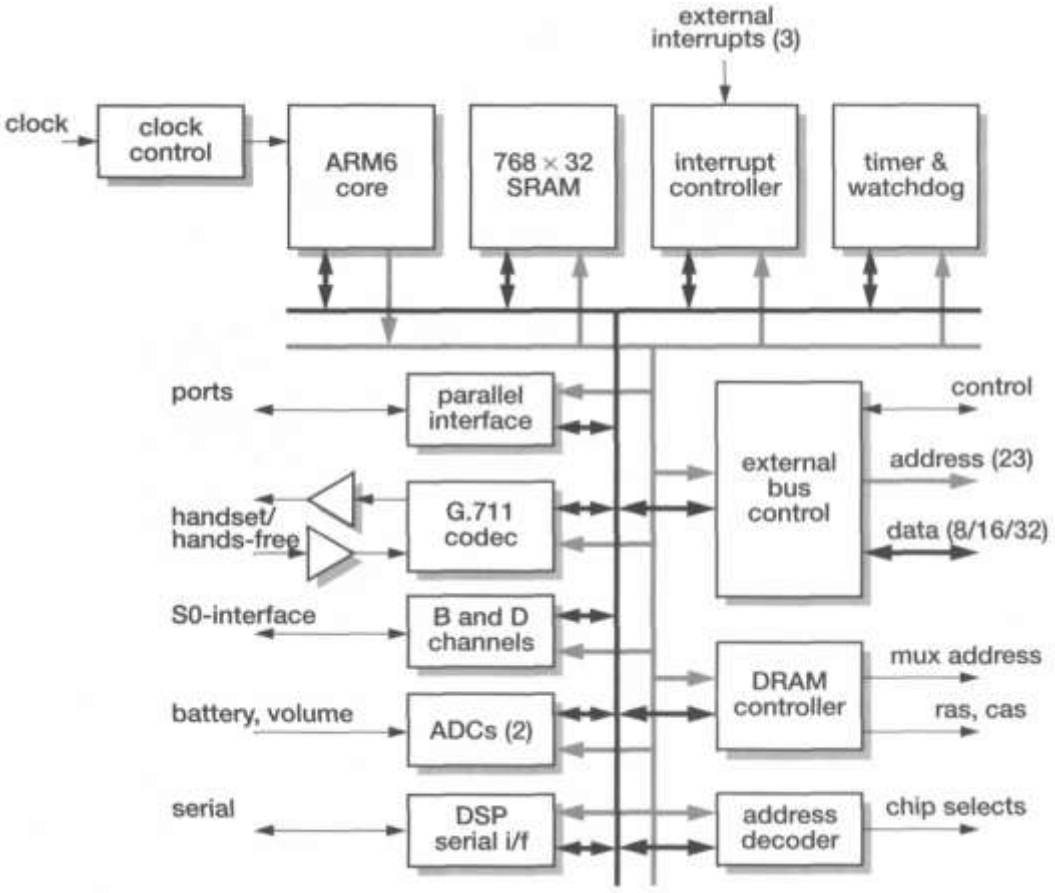


Figure 13.2 VIP organization.

Keypad interface

- The keyboard interface uses parallel output ports to strobe the columns of the keypad and parallel input ports with internal pull-down resistors to sense the rows.
- An OR gate on the inputs can generate an interrupt. If all the column outputs are active, any key press will generate an interrupt whereupon the ARM can activate individual columns and sense individual rows to determine the particular key pressed.

Clocks and Timers

- The chip has two clock sources. Normal operation is at 38.864 MHz, with 460.8 KHz used during power-down. A watchdog timer resets the CPU if there is no activity for 1.28 seconds, and a 2.5 ms tinier interrupts the processor from sleep mode for DRAM refresh and multitasking purposes.

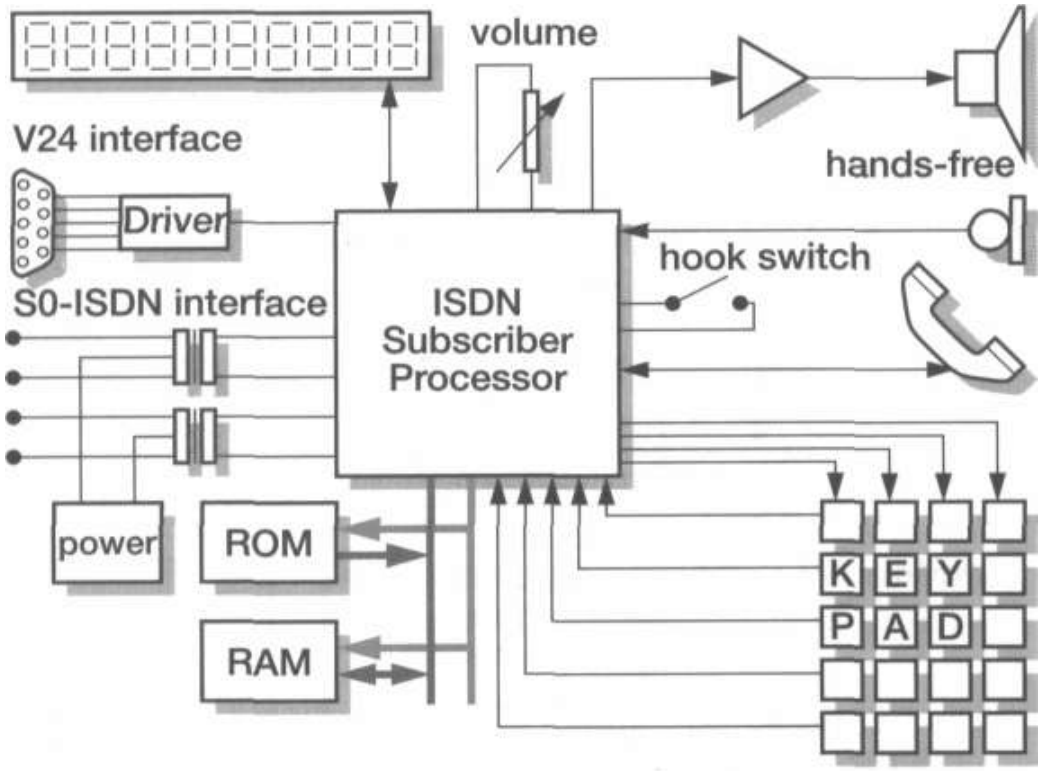


Figure 13.3 Typical VIP system configuration

13.5 The ARM7500 and ARM7500FE

- The ARM7500 is a highly integrated single-chip computer which combines the major components of the Acorn Rise PC (apart from the memory) onto a single chip.
- It was the first such system chip to employ as its processor macrocell a full ARM CPU, including cache and memory management, rather than just a basic integer processor core.
- The ARM7500FE adds the FPA10 floating-point coprocessor as an on-chip macrocell, and incorporates a number of other minor changes.
- Both of these devices are well suited to consumer multimedia applications such as set-top boxes, Internet appliances and games consoles, but have many other potential application areas.
- The principal macrocells on the ARM7500 and ARM7500FE chips are:
  - the ARM CPU core;
  - the FPA10 floating-point coprocessor (on the ARM7500FE only);
  - the video and sound macrocell;
  - the memory and I/O controller.

The ARM CPU CPU, the Core

- The ARM CPU core contains most of the functionality of the ARM710 only compromise to allow room for the other macrocells being a reduction in the size of the cache from 8 Kbytes to 4 Kbytes.
- The CPU is based around the ARM7 integer core (a forerunner of the ARM7TDMI, without Thumb and embedded debug support), with a 4 Kbyte 4-way set-associative mixed instruction and data cache, a memory management unit based on a 2-level page table with a 64-entry translation look-aside buffer and a write buffer.

The FPA10 floating-point unit

- The FPA10 was described in some detail in Section 6.4. It provides a significant enhancement of the ability of the system to handle floating-point data types (which, without the coprocessor, would be handled using the ARM floating-point library routines).
- The floating point performance is up to 6 MFLOPS (measured using the Lin-pack benchmark running double-precision floating point code) at 40 MHz.

The video and sound macrocell

- The video controller can generate displays using a pixel clock of up to 120 MHz (which is generated by a simple off-chip voltage controlled oscillator using an on-chip phasecomparator).
- It includes a 256-entry colour palette with on-chip 8-bit digital-to-analogue converters for each of the red, green and blue outputs and additional control bits for external mixing and fading.
- A separate hardware cursor is supported, and the output can drive a high-resolution colour monitor or a single- or double-panel grey-scale or colour liquid crystal display. The display timing is fully programmable.
- The ARM7500 sound system can generate eight independent channels of 8-bit (logarithmic) analogue stereo sound, played through an on-chip exponential digital-to-analogue converter.
- Alternatively, 16-bit sound samples can be generated through a serial digital channel and an external CD-quality digital-to-analogue converter. The ARM7500FE only supports the latter 16-bit sound system.
- The data channels for the video, cursor and sound streams are generated using the DMA controllers in the memory and I/O controller.

The memory andI/O controller

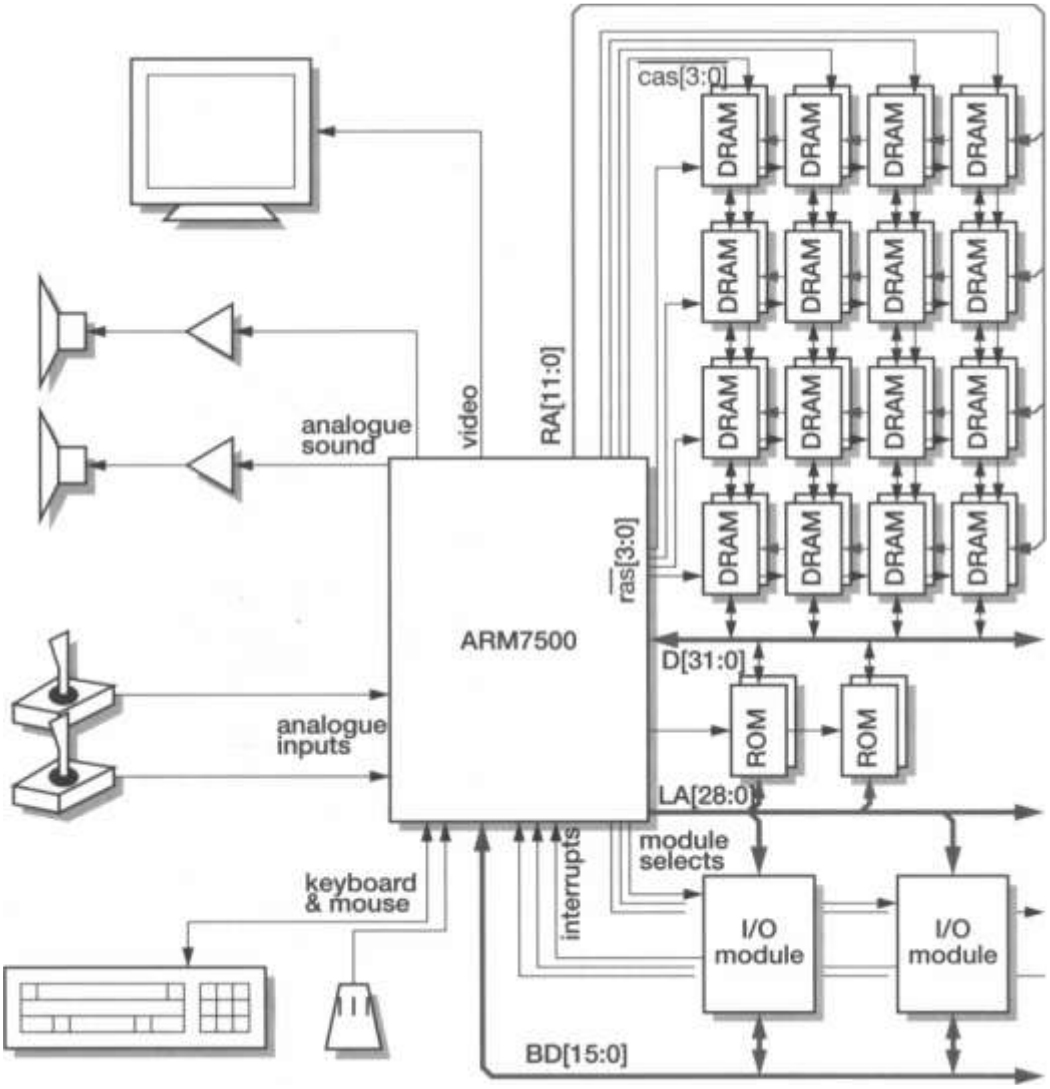
- The memory controller supports the direct connection of up to four banks of DRAM and two banks of ROM. Each bank can be programmed to be 16 or 32 bits wide and the memory controller will make double accesses for 32-bit quantities in 16-bit banks.
- The DRAM controller uses page mode accesses for sequential cycles in bursts of up to 256 transfers and supports a range of DRAM refresh modes, and the ROM controller also supports burst-mode where suitable devices are used.
- Three DMA controllers handle data streams for the video, cursor and sound channels.
- The I/O controller manages a 16-bit off-chip I/O bus (which is expandable to 32 bits using external buffers) and a number of on-chip interfaces.
- The off-chip bus supports simple peripheral devices, intelligent peripheral modules, PC-style peripherals and an interface for PCMCIA cards.
- The on-chip interfaces include four analogue comparators which can be used to support four analogue input channels, two serial ports intended for keyboards and/or mice, counter-timers, eight general-purpose open-drain I/O lines and a programmable interrupt controller. Power management facilities are also available.

System diagram

There are many variations on how these chips could be used, but a typical system organization is illustrated in Figure 13.10

Applications

- The ARM7500 has been used in low-cost versions of the Acorn Rise PC and in the Online Media interactive video set-top box.
- Its principal limitation compared with the original Rise PC chip set is restricting the video data stream to normal DRAM, whereas high-resolution displays on the standard Rise PC use VRAM.
- This precludes the use of large numbers of colours on displays with high resolutions, but with LCD displays or monitors of television, VGA or super-VGA resolution the restriction is not apparent.
- It is only at resolutions of 1,280 x 1,024 and above that the number of colours becomes restricted because of the bandwidth limitations of standard DRAM.



- Since interactive video and games machines use TV quality displays, and portable computers use liquid crystal displays at VGA (640 x 480 pixel) resolution, the ARM7500 is ideally suited to these applications. Its high integration and power-saving features make it suitable for hand-held test equipment, and its high quality sound and graphics are good characteristics for multimedia applications.

ARM7500 silicon

- A photograph of an ARM7500FE die is shown in Figure 13.11 and the characteristics of the ARM7500 are summarized in Table 13.2. Note the ARM7 core in the upper left-hand corner of the die occupying only 5% of the die area.

TheARM7100

Figure 13.12 The Psion Series 5MX.

- The ARM 100 is a highly integrated microcontroller suited to a range of mobile applications such as smart mobile phones and palm-top computers.
- It is the basis of the Psion Series 5 range of palm-top computers.
- A photograph of the Psion Series 5MX (which uses a later version of the 7100 with a modified architecture and a more advanced process technology)

ARM7100 organization

- The organization ARM710a CPU incorporates an ARM? processor core (a fore runner of the ARM7TDMI, without Thumb support), an ARM memory management unit, an 8 Kbyte 4-way associative quad-word line cache and a 4-address 8-data word write buffer.
- The use of a cache memory in this class of product is primarily to improve power-efficiency, which it does by reducing the number of off-chip memory accesses.
- The incorporation of a memory management unit is required to support the class of operating system required to run the general-purpose application mix.
- The system-on-chip organization is based around the AMBA bus. The peripherals include an LCD controller, serial and parallel I/O ports, an interrupt controller, and a 32-bit external bus interface to give efficient access to off-chip ROM, RAM and DRAM. A DRAM controller provides the necessary multiplexed address and control signals needed by this type of memory.

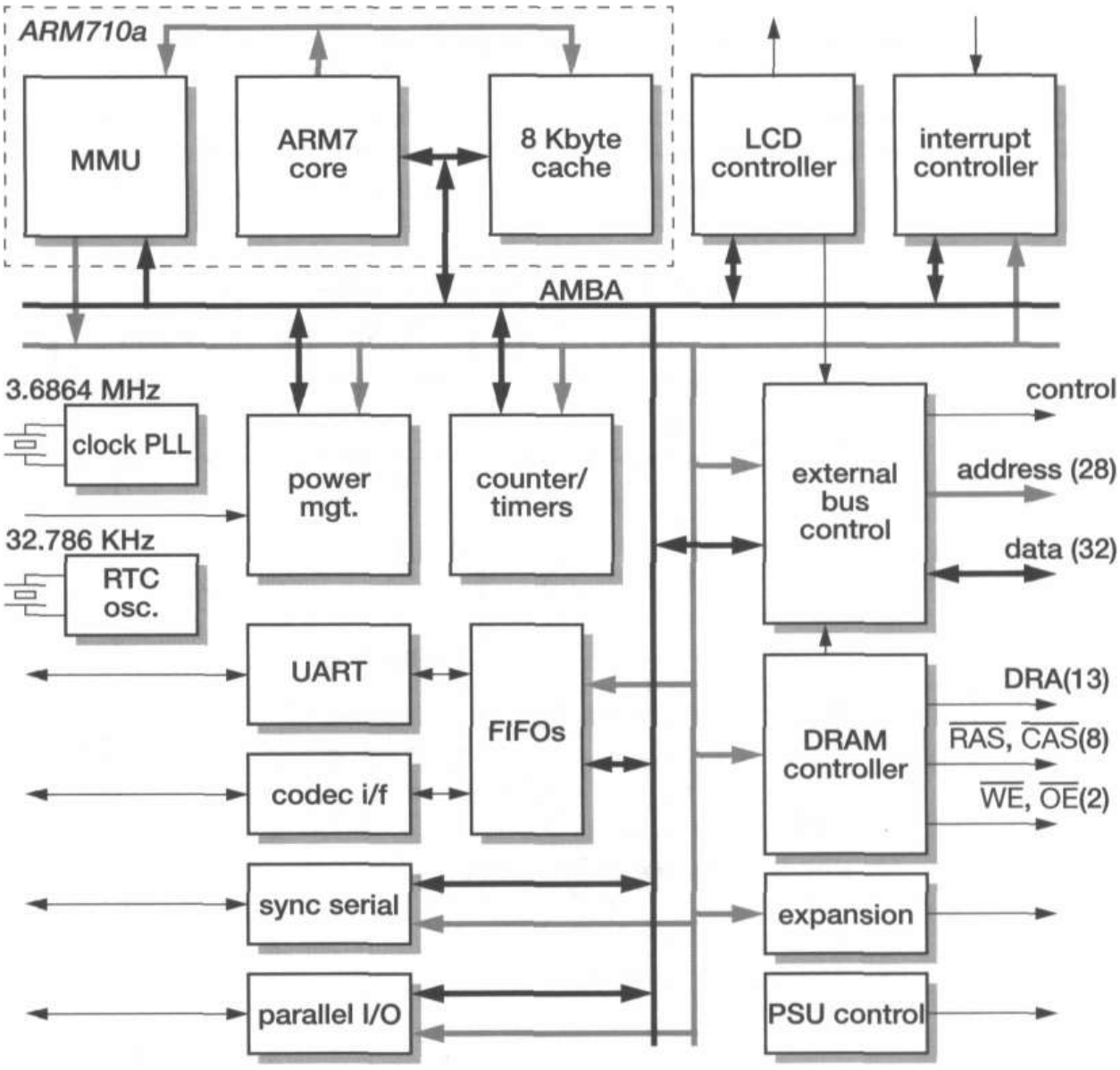


Figure 13.13 ARM7100 organization.

Power management :

- The ARM 100 is intended for use in battery-powered portable equipment where it is expected to deliver high performance in response to user input, but to operate at very low power consumption levels when awaiting user input.
- To address these requirements the chip has three levels of power management:

•in full operational mode the ARM CPU delivers around 14 MIPS while consuming 24mA at 3.0 V;
•in idle mode with the CPU stopped but other systems running it consumes 33 mW;
•in standby mode, with only the 32 kHz clock running, it consumes 33 uW.

- Other features to enhance power-efficiency include support for self-refresh DRA Memories which will retain their contents with no intervention from the ARM 100.

The Psion Series 5:

- The organization of the Psion Series 5 is illustrated in Figure 13.14.
- This shows how the various user interfaces connect to the ARM 100, giving a system of considerable architectural sophistication with minimal complexity at the chip level.

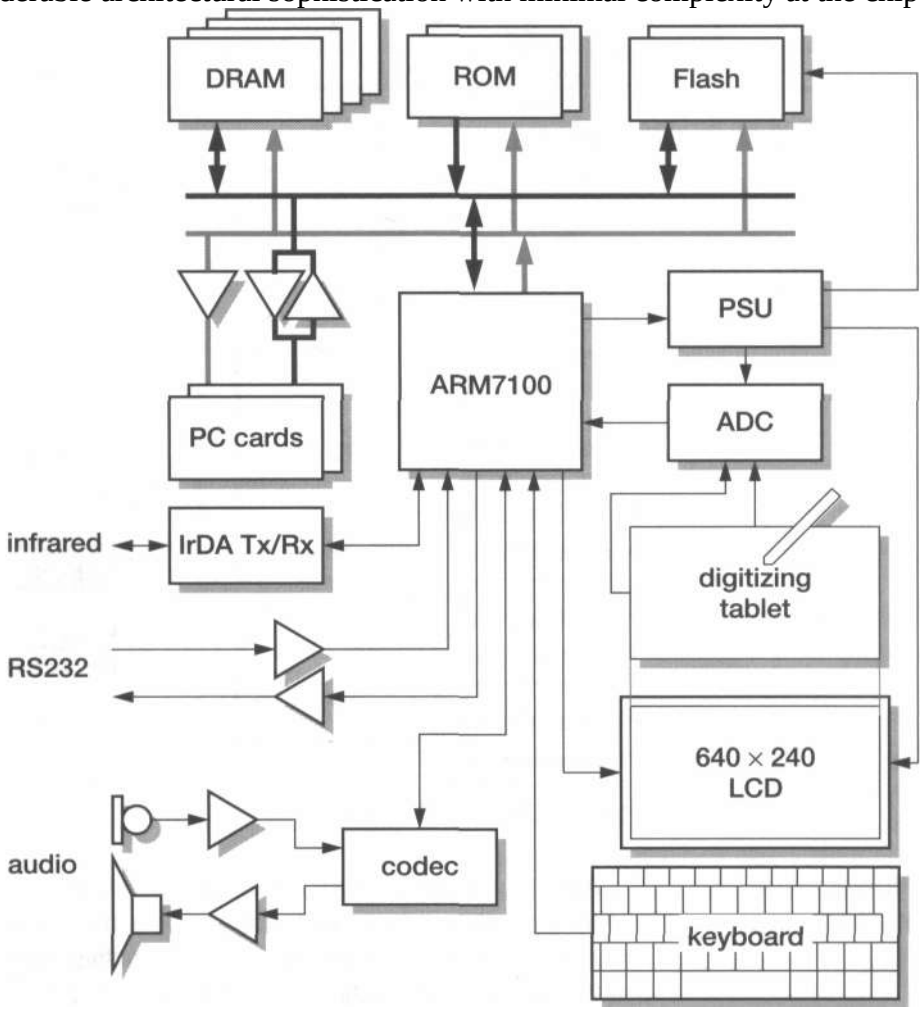
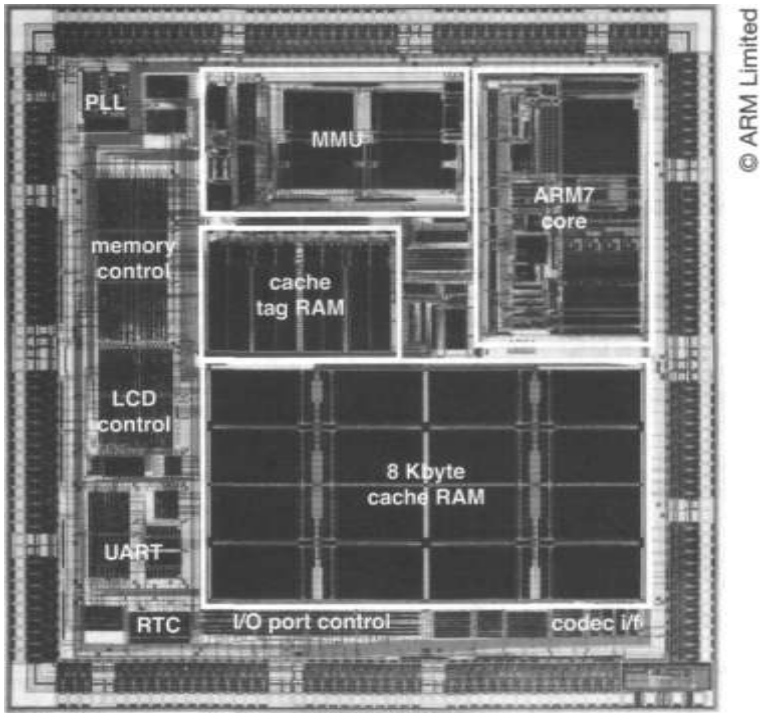


Figure 13.14 The Psion Series 5 hardware organization.

- The principal user input devices are the keyboard and the stylus pointing device.
- The former is connected simply to parallel I/O pins; the latter uses a transparent digitizing tablet overlaid on the LCD display and interfaced via an analogue-to-digital converter (ADC).
- Communications devices include an RS232 serial interface for wired connection and an IrDA compliant infrared interface for wireless connection to printers, modems and host PCs (for software loading and backup).
- An audio codec allows sound to be digitized from a built-in microphone, stored in the memory, and played back later through the small built-in loud-speaker.
- Hardware expansion is catered for through PCcard slots.

ARM7100 silicon:

- A plot of the ARM? 100 silicon layout is shown in Figure 13.15 and its principal characteristics are summarized in Table 13.3.
- It can be seen from Figure 13.15 that the silicon area is dominated by the CPU core which occupies all but the leftmost quarter of the chip core area.
- The 8 Kbyte cache RAM occupies the lower half of the CPU core area, with the cache tag store above it to the left, the MMU above that, and the ARM? core at the top right. All of the periph-erals and the AMBA bus are in the leftmost quarter.



ARM7100 die plot.

Comparison with ARM7500FE

- The balance here is different from the ARM7500FE (see Figure 13.11).
- The high-speed colour look-up table required to drive a CRT monitor on the ARM7500FE occupies more area than does the LCD controller on the ARM 100, and the floating-point hardware on the ARM7500FE also occupies a considerable area.
- To compensate the ARM7500FE has a half-size cache, but it still occupies a greater area.

TheSA-1100

- The SA-1100 is a high-performance integrated system-on-chip based around a modified version of the SA-110 StrongARM CPU core described in Section12.3
- It is intended for use in mobile phone handsets, modems, and other handheld applications that require high performance with minimal power consumption.
- The organization of the chip is illustrated in Figure 13.16

CPU core:

- The CPU core on the SA-1100 uses the same SA-1 processor core as the SA-110 with a small modification to support the exception vector relocation mechanism required by Windows CE.
- The instruction cache is also similar, being a 16 Kbyte cache using a 32-way associative CAM-RAM structure with 8-word lines.
- The memory management systems are unchanged apart from the inclusion of the ProcessID mechanism, again as required by Windows CE.
- The major differences from the SA-110 are on the data cache side, where the original 16 Kbyte cache has been replaced by an 8 Kbyte 32-way associative cache in parallel with a 512 byte 2-way set-associative cache.
- The memory management tables are used to determine which data cache a particular memory location Should be mapped into (if any).

- The objective of the second 'mini-cache' is to allow large data structures to be cached without causing Major pollution of the main data cache.
- The other difference on the data cache side is the addition of a read buffer. This unit can be used to pre-load data before the processor requests it so that when it is requested it is available with much reduced latency.
- The read buffer is software controlled via coprocessor registers.
- The final extension to the CPU core is the addition of hardware breakpoint and watchpoint registers, again programmed via coprocessor registers.

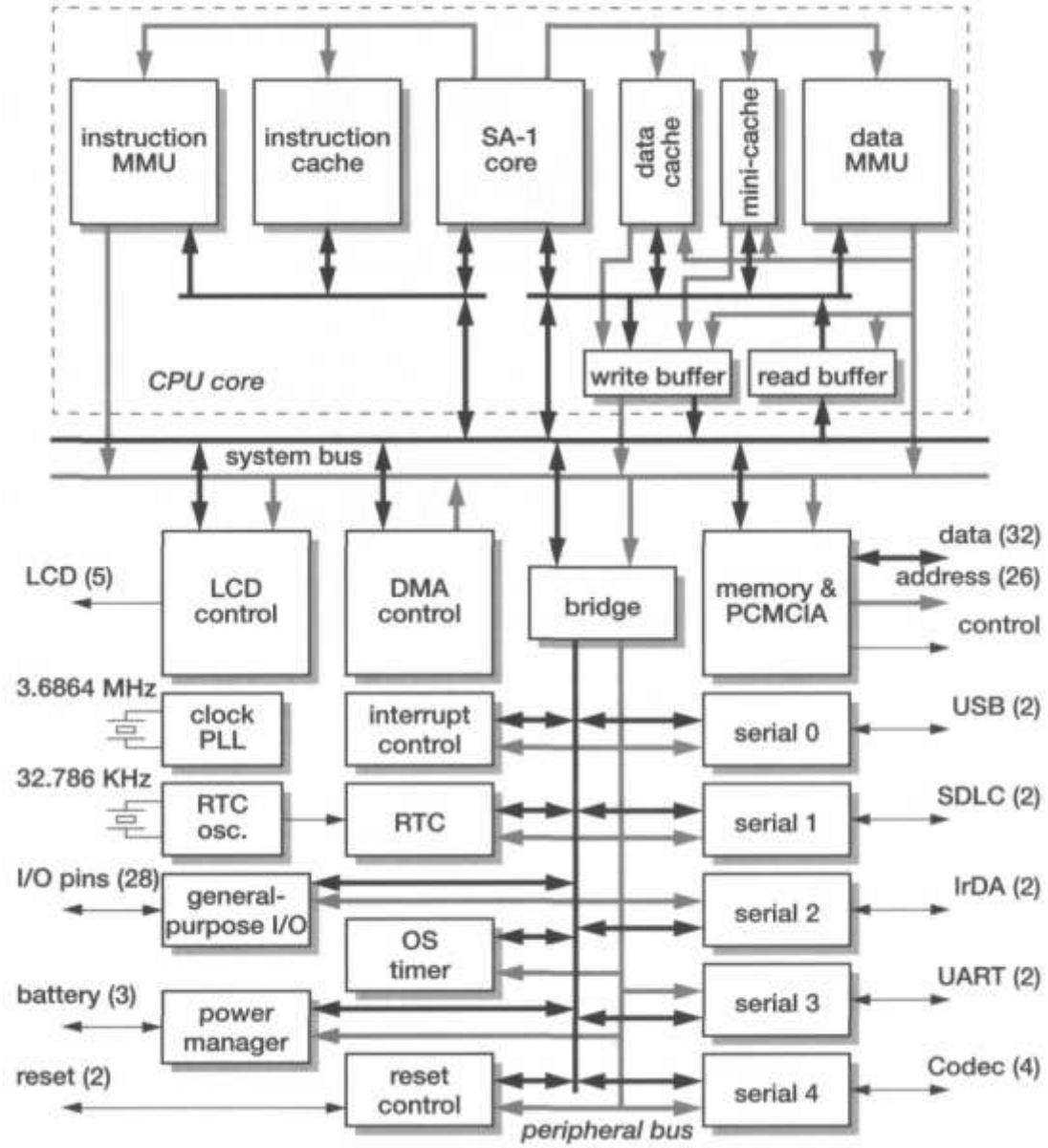


Figure 13.16 SA-1100 organization.

Memory controller

- The memory controller supports up to four banks of 32-bit off-chip DRAM, which may be conventional or of the 'extended data out' (EDO) variety. ROM, flash memory and SRAM are also supported.

- Further memory expansion is supported through the PCMCIA interface (which requires some external 'glue' logic), where two card slots are supported.

System control

The on-chip system control functions include:

•a reset controller;

•a power management controller that handles low-battery warnings and switches the system between its various operating modes;

•an operating system timer block that supports general timing and watchdog functions;

•an interrupt controller;

•a real-time clock that runs from a 32 KHz crystal source;

•28 general-purpose I/O pins.

Peripherals:

- The peripheral subsystem includes an LCD controller and specialized serial ports for USB, SDLC, IrDA, codec and standard UART functions.
- A 6-channel DMA controller releases the CPU from straightforward data transfer responsibilities.

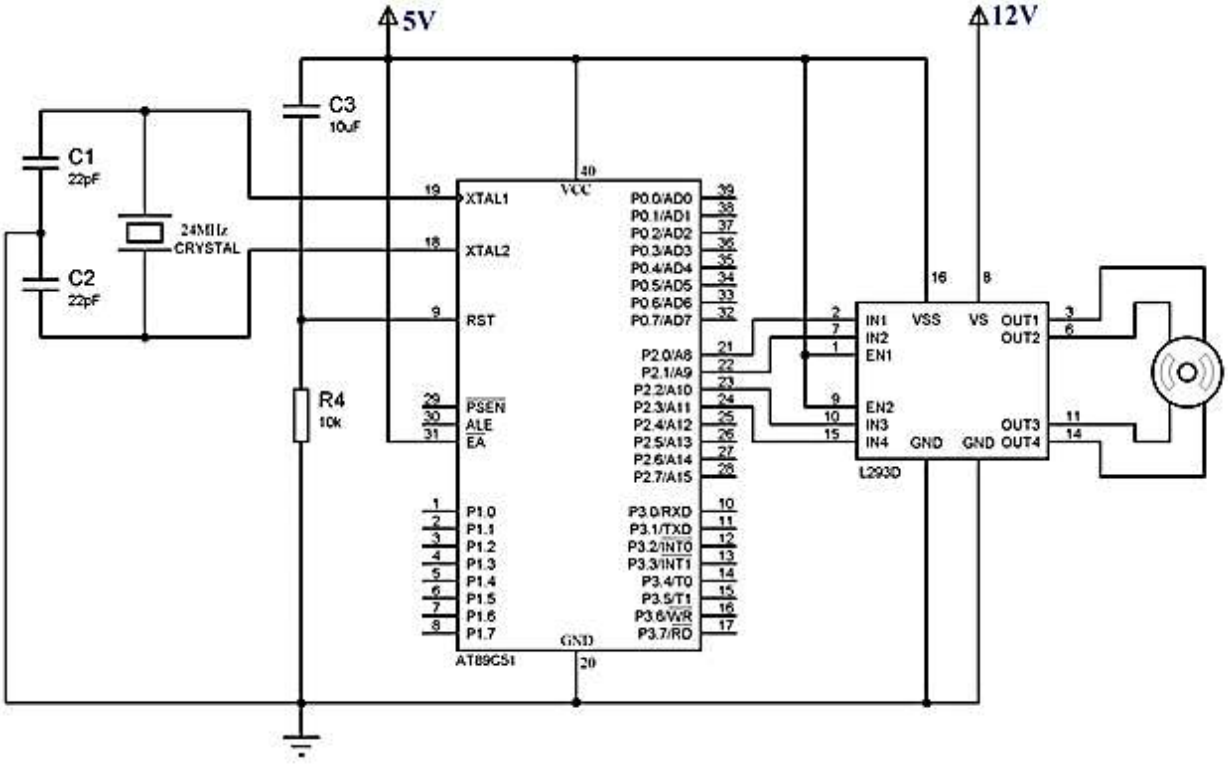
Bus structure:

- As can be seen from Figure 13.16, the SA-1100 is built around two buses connected through a bridge:
  - •The system bus connects all the bus masters and the off-chip memory.
  - •The peripheral bus connects all the slave peripheral devices.
- This dual bus structure is similar to the AMBA ASB-APB (or AHB-APB) split. It minimizes the size of the bus that has a high duty cycle and also reduces the complexity and cost of the bus interface that must be built into all of the peripherals.

Applications

- A typical SA-1100 application will require a certain amount of off-chip memory, probably including some DRAM and some ROM and/or flash memory.
- All that is then required is the necessary interface electronics for the various peripheral interfaces, display, and so on.
- The resulting system is very simple at the PCB level, yet very powerful and sophisticated in terms of its processing capability and system architecture.

17. Write a embedded C program to control the speed of the stepper motor and interface stepper motor with 8051. *[Nov'17]*

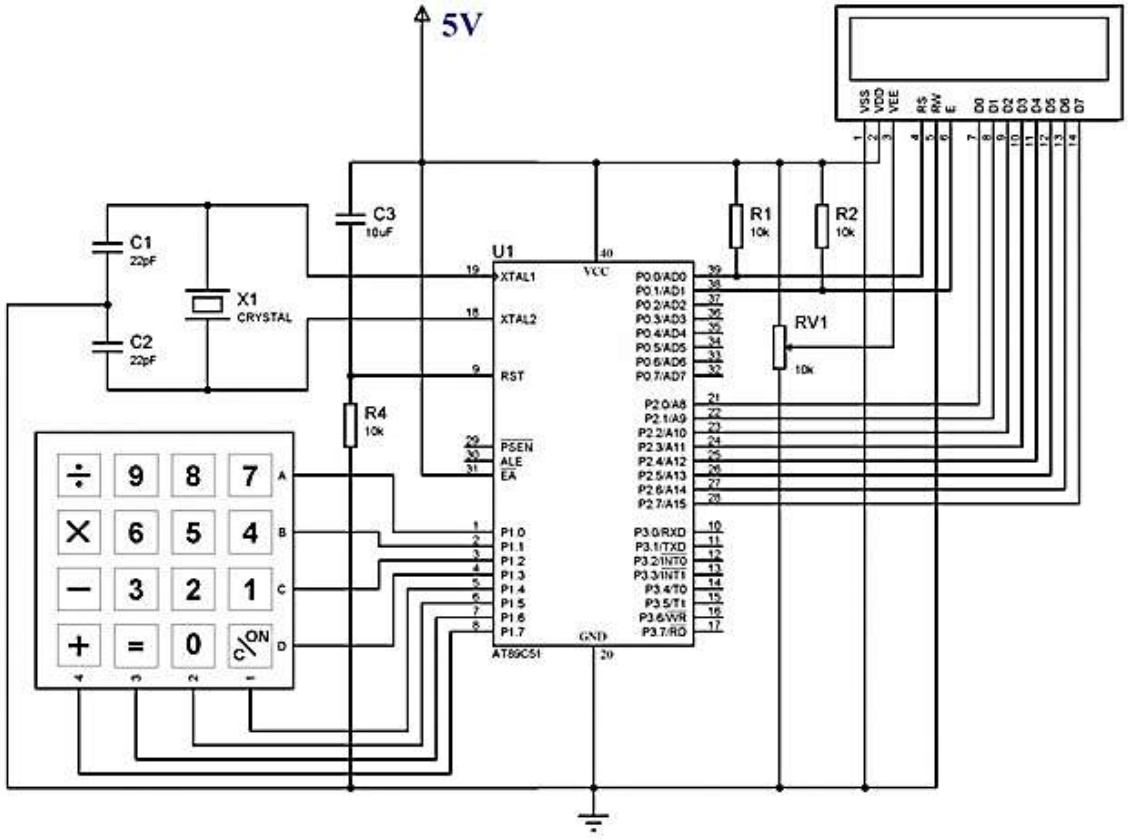


Program Coding:

```
#include<LPC214x.h>
// Define LPC2148 Header File #include <stdio.h>
 #define COIL_A 16 void motor_ccw(void);
 void delay(int); unsigned char STEP[] = {0x09, 0x08, 0x0C, 0x04, 0x06, 0x02, 0x03, 0x01
}
;
 void main(void)
 {
 unsigned char i = 0;
 PINSEL2 &= 0xFFFFFFF3;
 // P1.16 - P1.31 as GPIO IODIR1 = 0x000F0000;
 // P1.16 - P1.19 as Output while(1)
 {
IOCLR0 = 0xFF << COIL_A;
 motor_ccw();
// Stepper Motor counter clockwise
}
}
void delay(int n)
 {
int i,j;
for(i=0;i<n;i++)
 {
for(j=0;j<0x3FF0;j++)
 {;
}
```

```
 }
 }
void motor_ccw(void)
{
 unsigned int i=0;
 while (STEP[i] != '\0')
 {
 IOSET1 = STEP[i] <
<
COIL_A;
delay(1);
IOCLR1 = STEP[i] <
<
COIL_A;
delay(1);
i++;
 }
 }
```

18. Develop embedded C program to identify the key pressed and to display the pressed key in LCD display. *[Nov'17]*




```
#include<reg52.h> //including sfr registers for ports of the controller
#include<lcd.h>

//LCD Module Connections
```

```
sbit RS = P0^0;
sbit EN = P0^1;
sbit D0 = P2^0;
sbit D1 = P2^1;
sbit D2 = P2^2;
sbit D3 = P2^3;
sbit D4 = P2^4;
sbit D5 = P2^5;
sbit D6 = P2^6;
sbit D7 = P2^7;
//End LCD Module Connections

//Keypad Connections
sbit R1 = P1^0;
sbit R2 = P1^1;
sbit R3 = P1^2;
sbit R4 = P1^3;
sbit C1 = P1^4;
sbit C2 = P1^5;
sbit C3 = P1^6;
sbit C4 = P1^7;
//End Keypad Connections

void Delay(int a)
{
  int j;
  int i;
  for(i=0;i<a;i++)
  {
    for(j=0;j<100;j++)
    {
    }
  }
}

char Read_Keypad()
{
  C1=1;
  C2=1;
  C3=1;
  C4=1;
  R1=0;
  R2=1;
  R3=1;
  R4=1;
  if(C1==0){Delay(100);while(C1==0);return '7';}
  if(C2==0){Delay(100);while(C2==0);return '8';}
  if(C3==0){Delay(100);while(C3==0);return '9';}
  if(C4==0){Delay(100);while(C4==0);return '/';}
  R1=1;
```

```
  R2=0;
  R3=1;
  R4=1;
  if(C1==0){Delay(100);while(C1==0);return '4';}
  if(C2==0){Delay(100);while(C2==0);return '5';}
  if(C3==0){Delay(100);while(C3==0);return '6';}
  if(C4==0){Delay(100);while(C4==0);return 'X';}
  R1=1;
  R2=1;
  R3=0;
  R4=1;
  if(C1==0){Delay(100);while(C1==0);return '1';}
  if(C2==0){Delay(100);while(C2==0);return '2';}
  if(C3==0){Delay(100);while(C3==0);return '3';}
  if(C4==0){Delay(100);while(C4==0);return '-';}
  R1=1;
  R2=1;
  R3=1;
  R4=0;
  if(C1==0){Delay(100);while(C1==0);return 'C';}
  if(C2==0){Delay(100);while(C2==0);return '0';}
  if(C3==0){Delay(100);while(C3==0);return '=';}
  if(C4==0){Delay(100);while(C4==0);return '+';}
  return 0;
}

void main()
{
  int i=0;
  char c,p;
  Lcd8_Init();
  while(1)
  {
    Lcd8_Set_Cursor(1,1);
    Lcd8_Write_String("Keys Pressed:");
    Lcd8_Set_Cursor(2,1);
    Lcd8_Write_String("Times:");
    while(!(c = Read_Keypad()));
    p=c;
    while(p==c)
    {
      i++;
      Lcd8_Set_Cursor(1,14);
      Lcd8_Write_Char(c);
      Lcd8_Set_Cursor(2,7);
      Lcd8_Write_Char(i+48);
      Delay(100);
      while(!(c = Read_Keypad()));
    }
    i=0;
```

```
    Lcd8_Clear();
  }
}
```

PART A

1. What is the three stage pipelining in ARM processor? *[Nov'17]*
2. What is ARM datapath timing? *[Apr'18].*
3. What is five stage pipeline in ARM Processor? *(Nov'16)*
4. Write the operation carried out when CLZ instruction executed. *[Apr'18]*
5. What is the role of a Coprocessor? *(Apr'17)*
6. List few Embedded ARM Applications for ARM Processor. *(Nov'16)*
7. Give the details about the real time embedded ARM applications. *[Nov'17]*
8. Draw the structure of multicycle instructions of three stage pipeline operation? *(Apr'17)*

PART B

1. Explain the 5 – stage pipeline ARM cross – development tool kit. *[Apr'18]*
2. Explain the instruction set of ARM Processor. *(Nov'16)*
3. Discuss on coprocessor data transfer instruction of ARM processor. *[Apr'18]*
4. Explain the ARM coprocessor interface or Explain how does the coprocessor interface of the ARM work. *(Nov'16)*
5. Write short on coprocessor data and register transfer. *(Apr' 17)*
6. Explain the ARM floating point architecture. *[Apr'18]*
7. Elaborate the working principle of the VLSI ISDN Subscriber Processor in detail. *(Apr'17)*
8. Write a embedded C program to control the speed of the stepper motor and interface stepper motor with 8051. *[Nov'17]*
9. Develop embedded C program to identify the key pressed and to display the pressed key in LCD display. *[Nov'17]*

Reg. No. : | | | | | | | | | | | | |

Question Paper Code : 40984

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2018

Seventh Semester

Electrical and Electronics Engineering

EE6008 – MICROCONTROLLER BASED SYSTEM DESIGN

(Common to : Electronics and Instrumentation Engineering/Instrumentation and Control Engineering)

(Regulations 2013)

Time : Three Hours | Maximum : 100 Marks

Answer ALL questions

PART – A **(10×2=20 Marks)**

1. Draw the program memory organization of PIC16C6x microcontroller.
2. Write the operation carried out when these instructions executed by PIC.
   BTFSS f, b
   BCF f, b
3. What is the necessity of prescalar in the timer operation ?
4. How to display constant strings ?
5. Draw the start and stop conditions of I^2C.
6. Define baud rate.
7. Write the CPSR format of ARM Processor.
8. Differentiate little-endian and big-endian memory organizations.
9. What is ARM datapath timing ?
10. Write the operation carried out when CLZ instruction executed.

PART – B **(5×16=80 Marks)**

11. a) Explain the architecture of PIC16C6x microcontroller with neat block diagram. **(16)**

(OR)

b) i) Write PIC microcontroller assembly language program to arrange the given array having byte type data in ascending order. (8)

ii) With examples, explain the addressing modes of PIC16C6x microcontroller. (8)

12. a) Explain the various types of interrupts available in PIC microcontroller and also the step-by-step procedure to process an interrupt. (16)

(OR)

b) Explain the modes of Timer 1 of PIC16C6x microcontroller with block diagram. Also explain the function of associated registers. (16)

13. a) Write PIC microcontroller assembly language program to display the characters '2018' in the first row of 2 lines × 20 characters LCD. (16)

(OR)

b) Draw and explain the architecture of on chip ADC of PIC microcontroller and write a suitable assembly language program for configuring the ADC. (16)

14. a) i) Draw and explain the visible registers in an ARM processor. (8)

ii) Write ARM assembly language program to multiply two 32-bit binary numbers. (8)

(OR)

b) i) Explain the structure of the ARM cross-development tool kit. (8)

ii) Write a subprogram which copies a string of bytes from one memory location to another. The start of the source string will be passed in r_1, the length (in bytes) in r_2 and the start of the destination string in r_3. (8)

15. a) Explain the 5-stage pipeline ARM organization with neat diagram. (16)

(OR)

b) i) Discuss on coprocessor data transfer instructions of ARM processor. (8)

ii) Explain the ARM floating-point architecture. (8)

Reg. No. : 4 2 1 6 1 4 1 0 5 0 1 3

Question Paper Code : 50466

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2017

Seventh Semester

Electrical and Electronics Engineering

EE 6008 – MICROCONTROLLER BASED SYSTEM DESIGN

(Common to Electronics and Instrumentation Engineering/Instrumentation and Control Engineering)

(Regulations 2013)

Time : Three Hours | Maximum : 100 Marks

(Codes /Tables/Charts to be permitted, if any, may be indicated)

Answer ALL questions

PART – A (10×2=20 Marks)

1. Difference between microcontroller and PIC microcontroller.
2. List out the types of addressing mode.
3. What do you mean by state machine ?
4. Define subroutine.
5. List out some registers associated with UART.
6. Difference between bus operation and bus subroutine.
7. Define baud rate.
8. List out the four ARM development tools.
9. What is three stage pipelining in ARM processor ?
10. Give the details about the real time embedded ARM applications.

PART – B (5×16=80 Marks)

11. a) i) Detail description about the various types of addressing modes. (8)

ii) Explain about the instruction set of PIC microcontroller. (8)

(OR)

b) Draw and explain about the architecture of PIC Microcontroller. (16)

12. a) In detail give an account on Timer programming, RAM/ROM allocation in PC. (16)

(OR)

b) i) Define Interrupt. (4)

ii) Explain the interrupt structure of PIC microcontroller with neat diagram. (12)

13. a) Exhibit the operation of I2C bus and develop embedded C program to transmit data using I2C bus. (16)

(OR)

b) Explain the PIC interfacing with peripherals that includes ADC's with timer and sensors. (16)

14. a) i) Explain the various data operations involved in ARM. (8)

ii) Illustrate the concept of data operations in ARM processor. (8)

(OR)

b) With neat sketch explain the functional block diagram ARM architecture. (16)

15. a) Write a embedded C program to control the speed of the stepper motor and interface stepper motor with 8051. (16)

(OR)

b) Develop embedded C program to identify the key pressed and to display the pressed key in LCD display. (16)

Reg. No. :

Question Paper Code : 71757

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2017

Seventh Semester

Electrical and Electronics Engineering

EE 6008 — MICROCONTROLLER BASED SYSTEM DESIGN

(Common to Electronics and Instrumentation Engineering, Instrumentation and Control Engineering)

(Regulations 2013)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What are the benefits of having RISC architecture?
2. Define Brown out reset mode.
3. Mention the interrupts available in FTC Microcontroller.
4. Write an ALP to initialize the PORT A using PIC microcontroller.
5. Microcontroller based control is more advantageous than conventional control — Justify.
6. How is temperature sensor is interfaced with PIC Microcontroller?
7. Define Context Switching.
8. State the function of ARMulator and define its operations at various levels of accuracy.
9. Draw the structure of multicycle instruction of three stage pipeline operation.
10. What is the role of a co-pressure?

PART B — (5 × 16 = 80 marks)

11. (a) With neat functional block diagram explain the architecture of PIC16C7X Microcontroller in detail.

Or

(b) (i) Discuss in detail about memory organization of a PIC microcontroller. (8)

(ii) Explain the various addressing modes in PIC, for accessing data memory. (8)

12. (a) Explain the process and procedure to display constant strings and variable strings.

Or

(b) Explain the concept of interrupt logic and interrupt structure of PIC microcontroller with an example.

13. (a) Illustrate with suitable example how I^2C communication is carried out in PIC Microcontroller.

Or

(b) Explain the operation of ADC interfacing with PIC Microcontroller.

14. (a) (i) Explain the Arm Programmer's Model in detail, with supporting diagram. (10)

(ii) Write the subroutine program to output a text string following a CALL Instruction using ARM processor. (6)

Or

(b) Write short notes on ARM MMU architecture.

15. (a) Elaborate the working principle of VLSI ISDN subscriber processor in detail.

Or

(b) Write short notes on

(i) 5 stage pipeline ARM organization.

(ii) Coprocessor data and register transfer.

Reg. No. : | | | | | | | | | | | | |

Question Paper Code : 80363

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2016.

Seventh Semester

Electrical and Electronics Engineering

EE 6008 — MICROCONTROLLER BASED SYSTEM DESIGN

(Common to Electronics and Instrumentation Engineering and Instrumentation and Control Engineering)

(Regulations 2013)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. Write about the Status Register of PIC Microcontroller.
2. List out all the addressing Modes in PIC Microcontroller.
3. What is the minimum and maximum clock frequency for PIC 16CXX?
4. What is the role of TRISx register in I/O Port Management?
5. What is the value to be loaded into SPBRG register if we want 19200 baud rate with 10MHz clock source.
6. List the registers associated with UART.
7. What is the purpose of Program Counter?
8. List out some of ARM Development Tools.
9. What is five stage pipeline in ARM PROCESSOR?
10. List few embedded Application for ARM processor.

PART B — (5 × 16 = 80 marks)

11. (a) (i) Draw and explain the architecture of PIC 16 Microcontroller. (10)

(ii) Explain about the instruction set of PIC Microcontroller. (6)

Or

(b) Explain about the Various Memory organization of PIC Microcontroller. (16)

12. (a) Explain the functionality of TIMER for PIC Microcontroller with a suitable program. (16)

Or

(b) What is Interrupt? Explain the Interrupt structure of PIC Microcontroller with neat diagram. (16)

13. (a) What is meant by I^2C module? Explain how I^2C is interfaced with PIC Microcontroller. (16)

Or

(b) Using Suitable circuits, construct and explain how ADC is interfaced with PIC microcontroller. (16)

14. (a) With Neat sketch explain the functional block diagram ARM architecture. (16)

Or

(b) Explain the various Operating modes Programmers model in Arm Processor. (16)

15. (a) Using Suitable example, explain the various instruction set of ARM processor. (16)

Or

(b) Explain how does the coprocessor interface of the ARM work. (16)