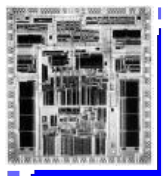
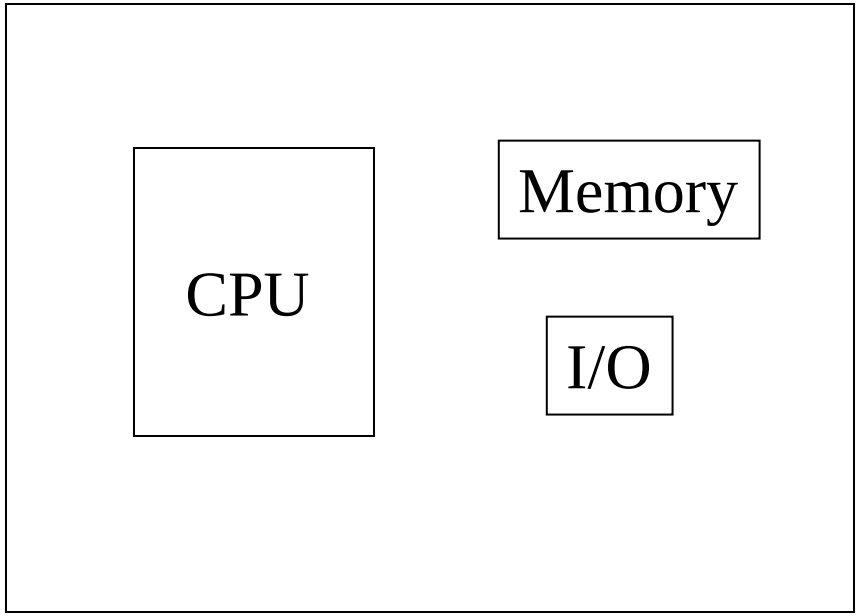


Embedded Systems

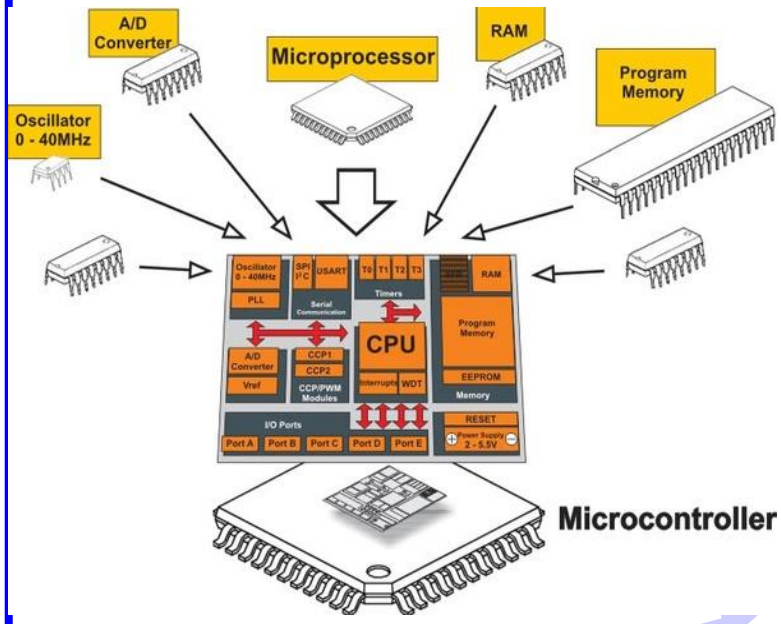
Basic Components of Computer

- CPU
- Memory
- I/O

Motherboard



Microcontrollers



A single chip (SoC)

CPU

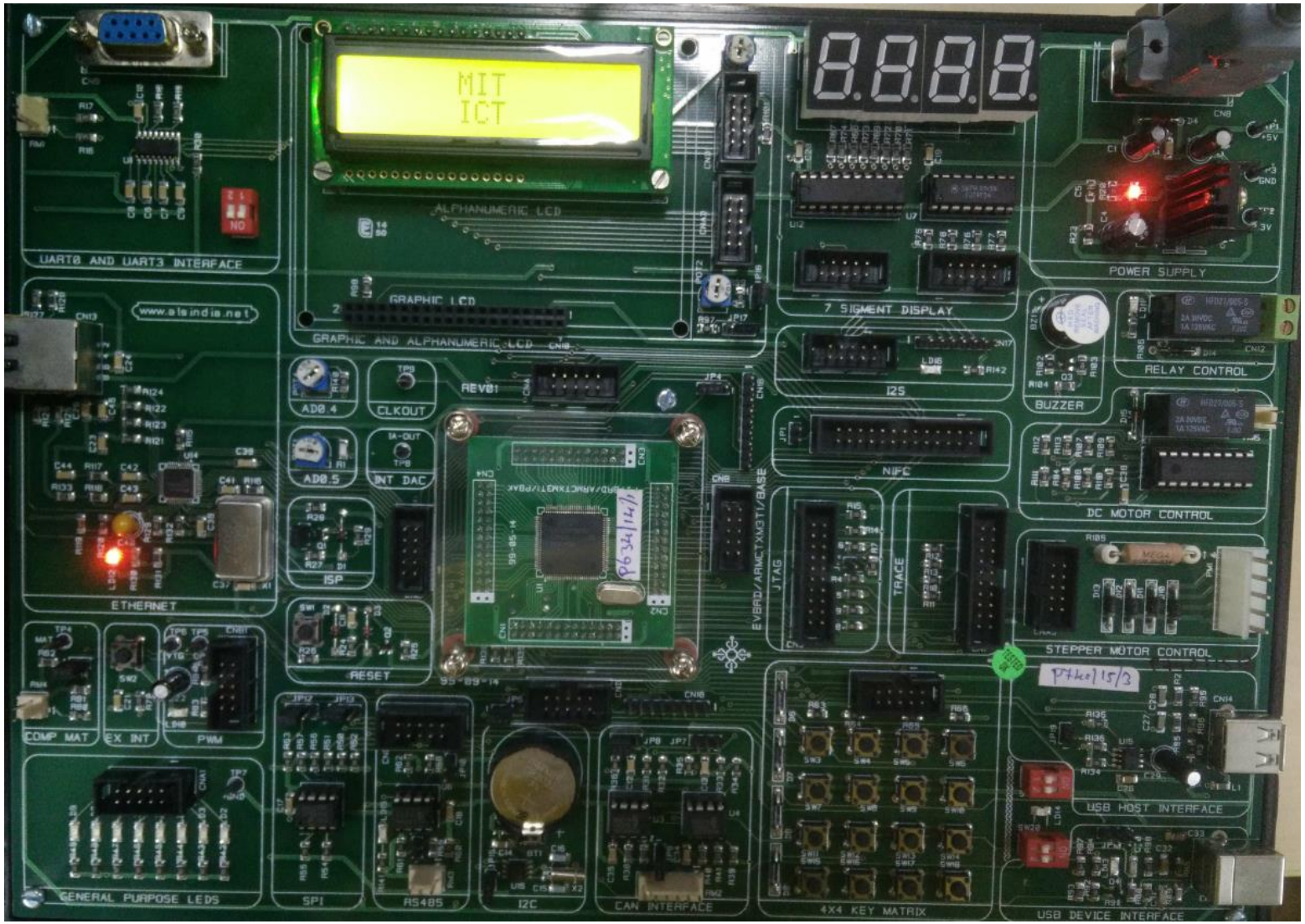
Memory

ROM	RAM

I/O

Subsystems:
Timers, Counters, Analog Interfaces, I/O interfaces

LPC 1768


MIT
ICT
ALPHANUMERIC LCD
GRAPHIC LCD
GRAPHIC AND ALPHANUMERIC LCD
UART0 AND UART3 INTERFACE
www.elsindia.net
ETHERNET
AD0.4
CLKOUT
AD0.5
INT DAC
ISP
RESET
REV01
EVBRD/ARMCTXM3T1/BASE
JTAG
TRACE
7 SEGMENT DISPLAY
I2S
NIFC
POWER SUPPLY
RELAY CONTROL
BUZZER
DC MOTOR CONTROL
STEPPER MOTOR CONTROL
USB HOST INTERFACE
USB DEVICE INTERFACE
4X4 KEY MATRIX
CAN INTERFACE
I2C
RS485
SPI
PWM
EX INT
COMP MAT
GENERAL PURPOSE LEDS

Microprocessor vs Microcontroller

Sl.no	Microprocessor	Microcontroller
1	General purpose processor	Specific application controller
2	Contains no RAM, no ROM, no I/O ports on chip itself.	Contains RAM, ROM, I/O ports on chip itself
3	Size of RAM/ROM can vary	Size of RAM/ROM is fixed
4	Makes the system bulkier	Make the system compact
5	More expensive	Less expensive
6	It has less bit handling instructions	It has more bit handling instructions
7	Less number of pins have multiplexed functions	more number of pins have multiplexed functions
8	More flexible in designer point of view	Less flexible in designer point of view
9	Limited power saving options compared to microcontrollers	Includes lot of power saving features
10	Eg: Desktop PC, 8086, i7	Eg: Digital Camera, 8051, msp430
11	Execution faster	Compared to µp slower
12	More general purpose registers	Less number of gen purpose registers
13	More addressing modes	Less addressing modes
14	Design time is more	Application design time less
15	Microprocessors are based on von Neumann model/architecture where program and data are stored in same memory module	Micro controllers are based on Harvard architecture where program memory and Data memory are separate
16	Cannot be used in compact systems and hence inefficient	Can be used in compact systems and hence it is an efficient technique

17	Example code:
	`ADD AX,BX`
	`ADD AX,CX`
	`ADD AX,DX`
		Example code:
		`MOV A,#2fh`
		`MOV B,#2fh`
		`ADD A,B`

18	It cannot be used as stand alone	Can be used as stand alone.
19	May or may not be real-time application oriented	Real-time application oriented
20	Fig : a	Fig :b

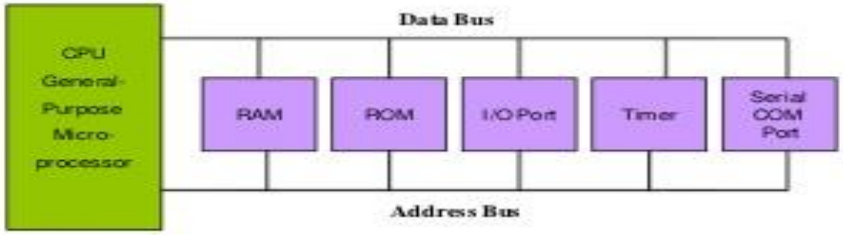


Fig : (a) Microprocessor

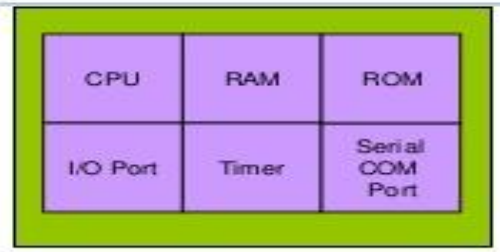
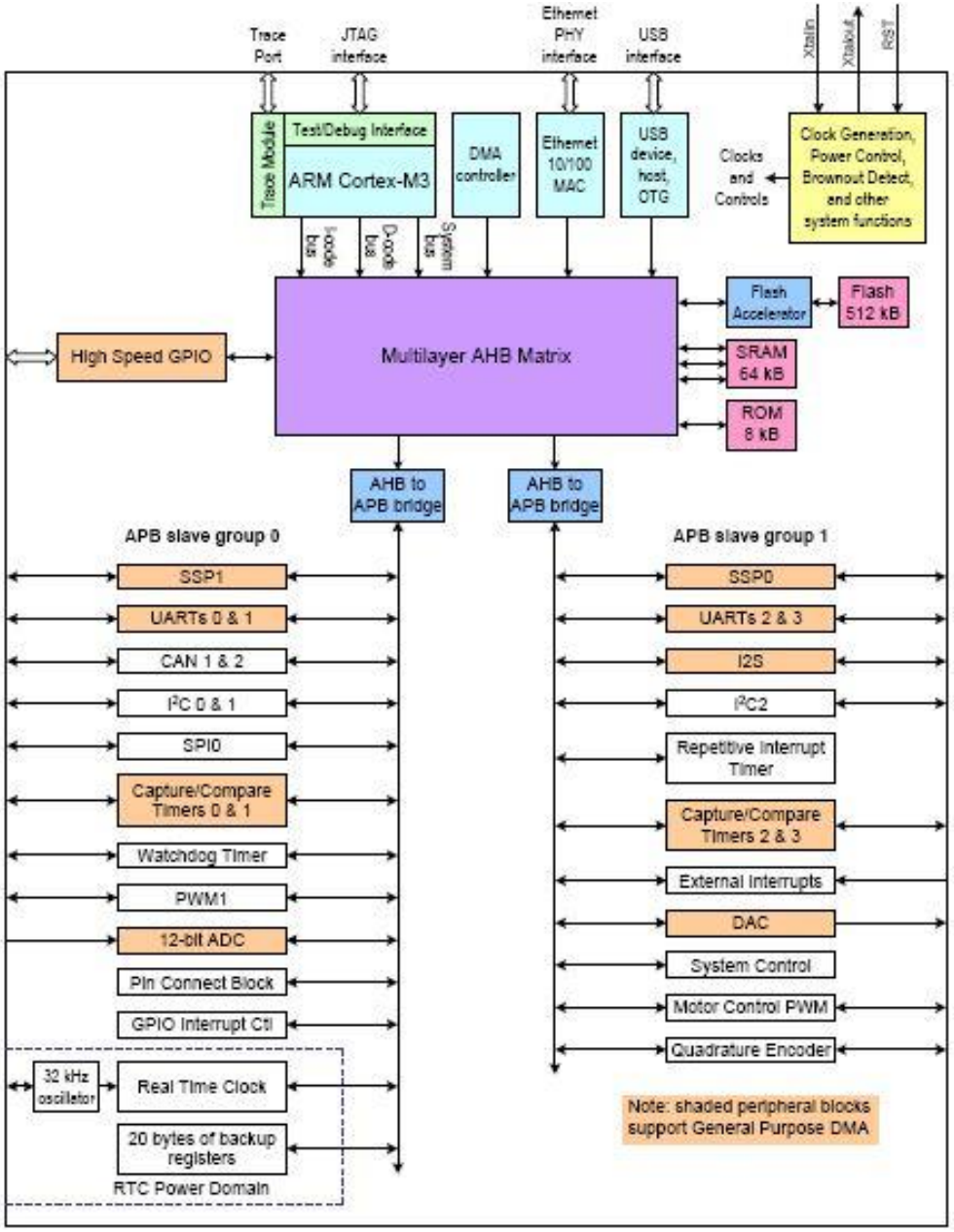
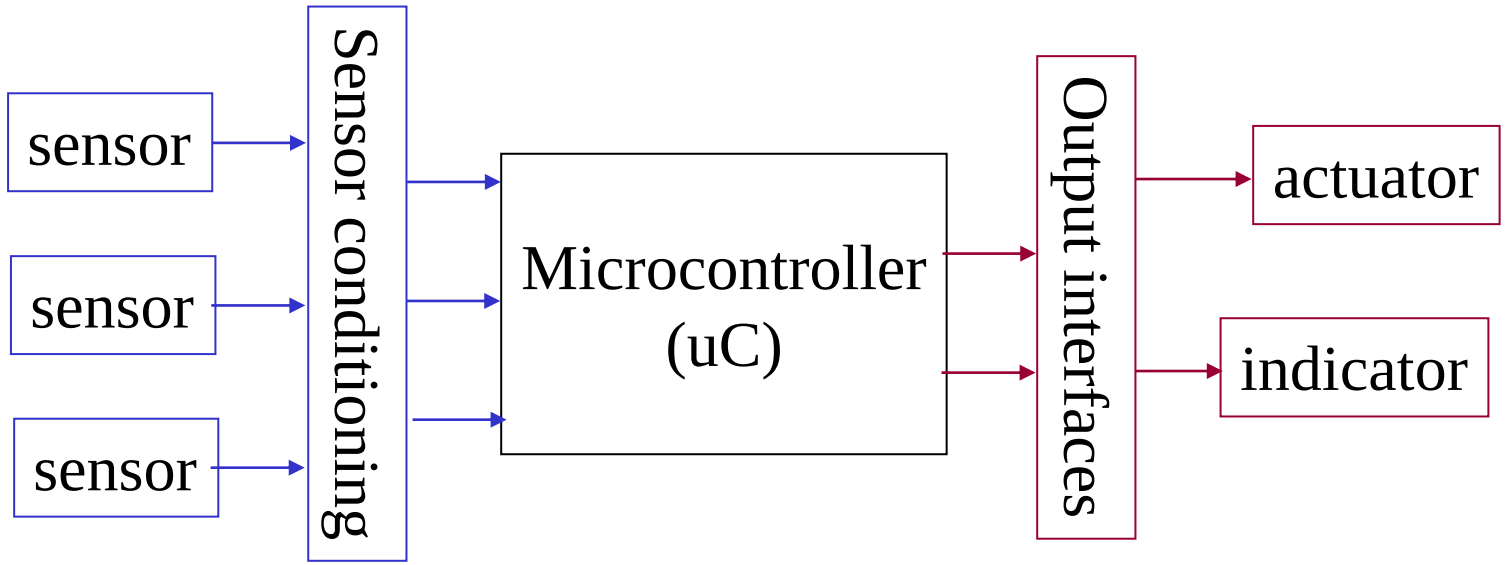


Fig : (b) Microcontroller

Microcontrollers

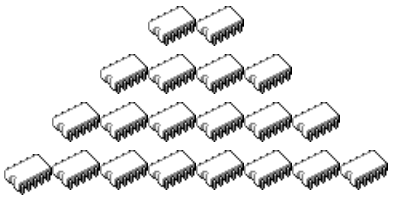


Embedded System General Block Diagram

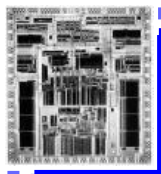


Embedded systems overview

- Embedded computing systems
  - Computing systems embedded within electronic devices
  - Nearly any computing system other than a desktop computer
  - Billions of units produced yearly, versus millions of desktop units



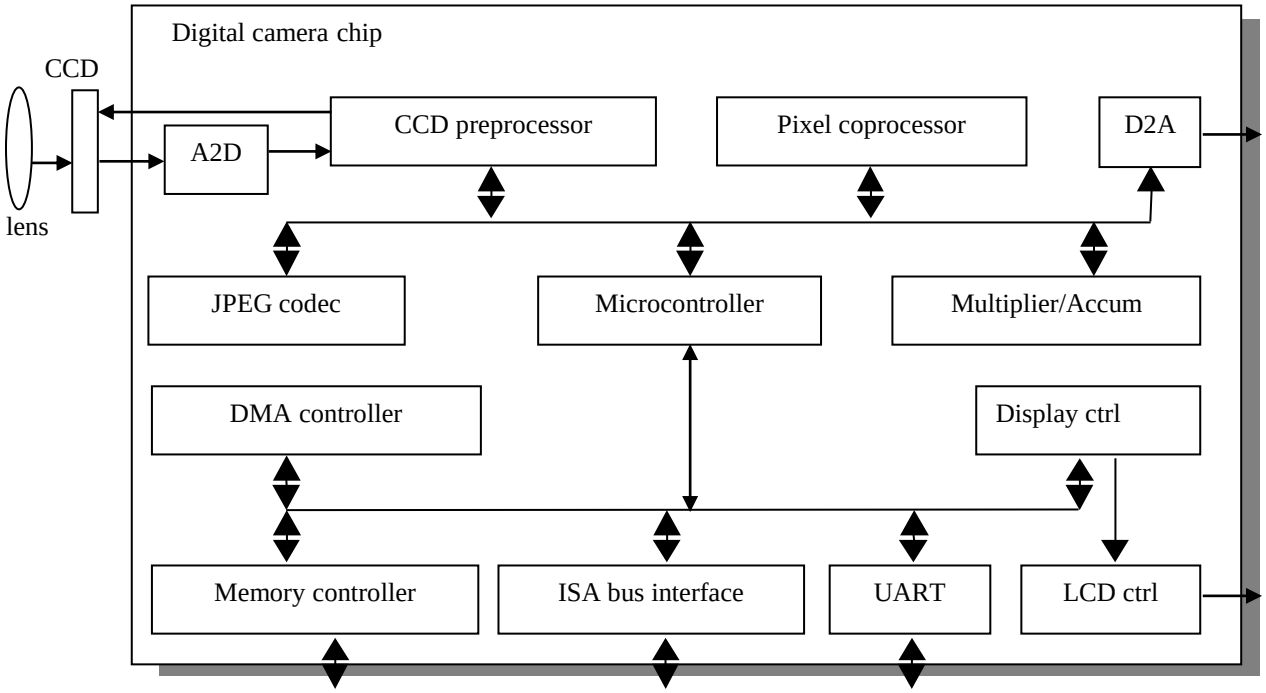
Lots more of these, though they cost a lot less each.



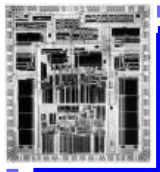
Some common characteristics of embedded systems

- Single-functioned
  - Executes a single program, repeatedly
- Tightly-constrained
  - Low cost, low power, small, fast, etc.
- Reactive and real-time
  - Continually reacts to changes in the system's environment
  - Must compute certain results in real-time without delay

An embedded system example -- a digital camera



- Single-functioned -- always a digital camera
- Tightly-constrained -- Low cost, low power, small, fast
- Reactive and real-time -- only to a small extent

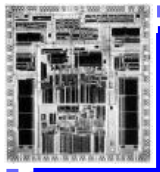


RISC vs CISC

RISC – Reduced Instruction Set Computer
CISC – Complex Instruction Set Computer

1. One of the major characteristics of RISC architecture is a large number of registers. All RISC architectures have at least 8 or 16 registers. Of these 16 registers, only a few are assigned to a dedicated function. One advantage of a large number of registers is that it avoids the need for a large stack to store parameters.

2. RISC processors have a fixed instruction size. In a CISC microprocessors such as the x86, instructions can be 1, 2, 3, or even 5 bytes. For example, look at the following instructions in the x86:

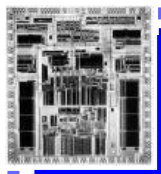


RISC vs CISC

3. RISC processors have a small instruction set. RISC processors have only basic instructions such as ADD, SUB, MUL, LOAD, STORE, AND, OR, EOR, CALL, JUMP, and so on.

The limited number of instructions is one of the criticisms leveled at the RISC processor because it makes the job of Assembly language programmers much more tedious and difficult compared to CISC Assembly language programming.

This is one reason that RISC is used more commonly in high-level language environments such as the C programming language rather than Assembly language environments.

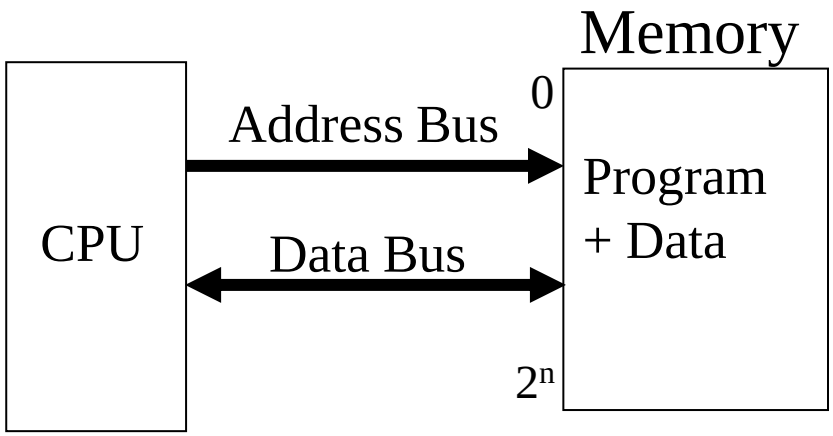


RISC vs CISC

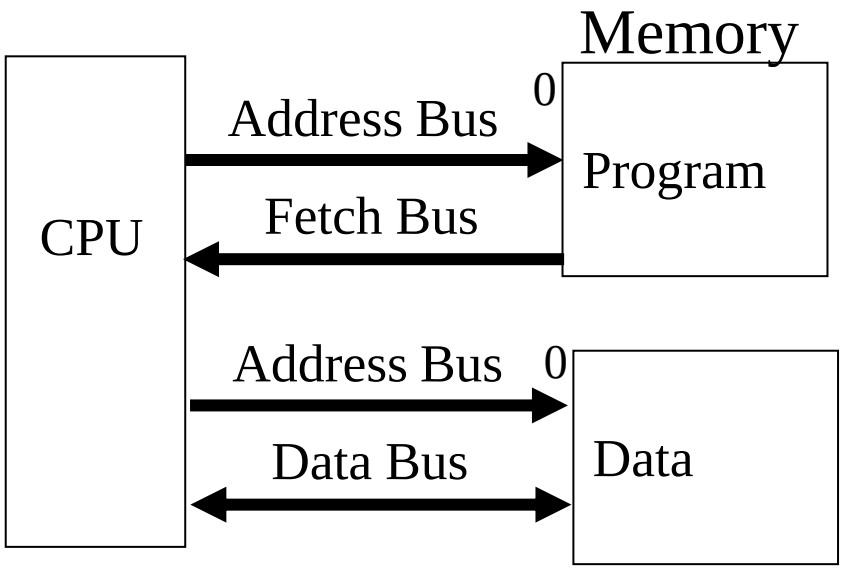
4. The most important characteristic of the RISC processor is that more than 99% of instructions are executed with only one clock cycle, in contrast to CISC instructions.

5. RISC processors have separate buses for data and code.

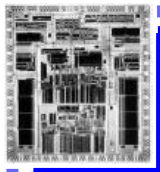
Microcontroller Architectures



Von Neumann Architecture



Harvard Architecture



RISC vs CISC

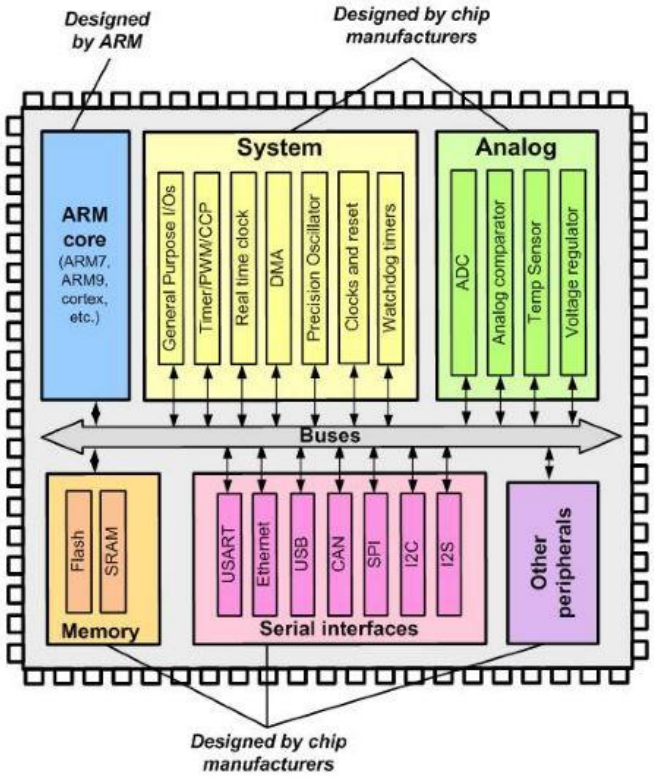
6. Because CISC has such a large number of instructions, each with so many different addressing modes, microinstructions (microcode) are used to implement them.

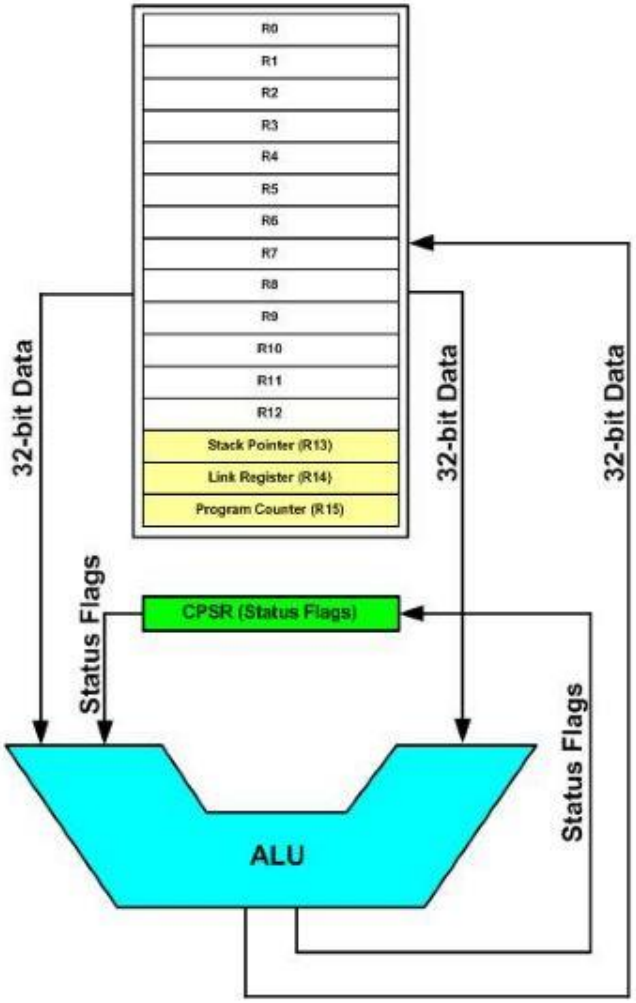
RISC instructions, however, due to the small set of instructions, are implemented using the hardwire method.

7. RISC uses load/ store architecture. In CISC microprocessors, data can be manipulated while it is still in memory.

8. Memory Mapped IO in RISC and IO mapped IO in CISC

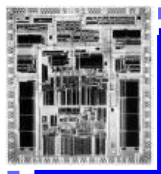
RISC vs CISC





N,Z,C,V flags

Bit No. 31, 30, 29. 28



Addressing Modes

The way in which an operand is specified in an instruction is called Addressing Mode

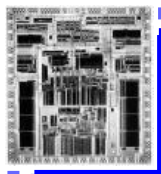
1. Register - MOV R0. R1
2. Immediate – MOV R0, #1
3. Register Indirect- LDR R0, [R1]
   STR R0, [R1]
4. Indexed

Addressing modes

Indexed Addressing Mode	Syntax	Pointing Location in Memory	Rm Value After Execution
Preindex	LDR Rd, [Rm,#k]	Rm+#k	Rm
Preindex with WB*	LDR Rd, [Rm,#k]!	Rm+#k	Rm + #k
Postindex	LDR Rd, [Rm],#k	Rm	Rm + #k

**WB means Writeback*

*** Rd and Rm are any of registers and #k is a signed 12-bit immediate value between -4095 and +4095*



Load/Store Byte/Half Word

LDRB – Load Byte

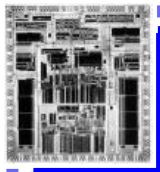
LDRH – Load Halfword

LDRSB – Load Signed byte

LDRSH – Load Signed Halfword

STRH – Store Halfword

Note : For all these instructions – Indirect/Indexed addressing modes are applicable)



MOV instruction

MOV Rd, Rn

MOV Rd, #0x12

MOVW Rd, #0x1234 (Move Word)

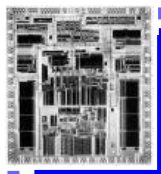
MOVT Rd, #0x1234 (Move Top)

MVN Rd, Rn (Move Negative)

MVN Rd, #0x12

MSR Special_Function_Reg, Rn (Move SFR from Register)

MRS Rn, Special_Function_Reg (Move Register from SFR)



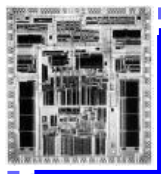
ADD instruction

ADD Rd, Rn, opr2 ; Rd ←Rn+opr2 (addition)

ADC Rd, Rn, opr2; Rd ← Rn + opr2 + C

Flags not affected.

Use S suffix to update flags: ADDS, ADCS



SUBTRACT

SUB Rd, Rn, opr2 ; Rd ←Rn-opr2 (Subtract)

SBC Rd, Rn, opr2; Rd ← Rn -opr2 – (1- C) (Subtract with Carry)

RSB Rd, Rn, opr2 ; Rd ←opr2-Rn (Reverse Subtract)

RSC Rd, Rn, opr2 ; Rd ←opr2-Rn – (1-C) (Reverse Subtract with Carry

Flags not affected.

Use S suffix to update flags : SUBS, SBCS, RSBS, RSCS

MULTIPLICATION

MUL Rd, Rn, Rm ; Rd = Rn × Rm (Multiply)
MLA Rd, Rs1, Rs2, Rs3 ; Rd = (Rs1 × Rs2) + Rs3 (Multiply and Accumulate)
MLS Rd, Rm, Rs, Rn ; Rd = Rn -(Rs × Rm) (Multiply and Subtract)
UMULL RdLo, RdHi, Rn, Rm ; RdHi:RdLo = Rm × Rn (Unsigned Multiply Long)
UMLAL RdLo, RdHi, Rn, Rm ;RdHi:RdLo =(Rm × Rn) + (RdHi:RdLo) (Unsigned Multiply and Accumulate Long)
SMULL Rdlo, Rdhi, Rn, Rm ;Rdhi:Rdlo = Rm × Rn (Signed Multiply Long)
SMLAL Rdlo, Rdhi, Rn, Rm ;Rdhi:Rdlo =(Rm × Rn) + (Rdhi:Rdlo) (Signed Multiply and Accumulate Long)

Flags not affected.

LOGICAL INSTRUCTIONS

```
AND  Rd, Rn, Op2          ;Rd = Rn ANDed Op2
ORR Rd, Rn, Op2           ;Rd = Rn ORed with Op2
ORN Rd, Rn, Op2           ;Rd = Rn ORed with 1' s comp of Op2
EOR   Rd, Rn, Op2         ;Rd = Rn XORed Op2
```

Flags not affected.
Use S suffix to update flags

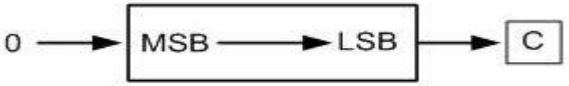
```
TEQ   Rn, Op2          ;performs Rn Ex-OR Op2
TST   Rn, Op2          ;performs Rn AND Op2
```

Flags N,Z affected

SHIFT AND ROTATE

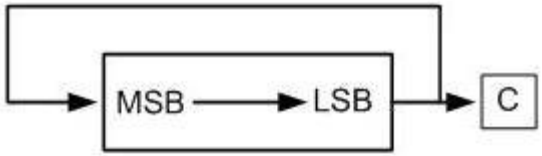
LSL Rd, Rn, Op2

LSR Rd, Rn, Op2

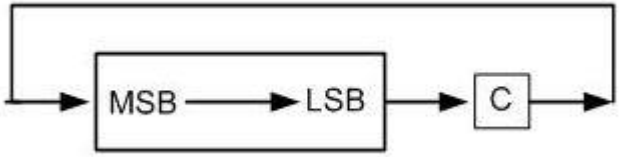


ASR Rd, Rn, Op2

ROR Rd, Rn, Op2



RRX Rd, Rm ;Rd = rotate Rm right 1 bit position



Flags not affected.

Use S suffix to update flags

Compare Instructions

CMP Compare

Flags: Affected: V, N, Z, C.

Format: CMP Rn,Op2 ;sets flags as if "Rn-Op2"

CMN Compare Negative

Flags: Affected: V, N, Z,C.

Format: CMN Rn,Op2 ;sets flags as if "Rn + Op2"

BRANCH INSTRUCTIONS

B **Branch (unconditional jump)**

Flags: Unchanged.

Format: B target ;jump to target address

Bxx **Branch Conditional**

Flags: Unaffected.

Format: Bxx target ;jump to target upon condition

BRANCH INSTRUCTIONS

Instruction		Condition
BCS	Branch if Carry Set	jump if C=1
BCC	Branch if Carry Clear	jump if C=0
BEQ	Branch if Equal	jump if Z=1
BNE	Branch if Not Equal	jump if Z=0
BMI	Branch if Minus/Negative	jump if N=1
BPL	Branch if Plus/Positive	jump if N=0
BVS	Branch if Overflow	jump if V=1
BVC	Branch if No overflow	jump if V=0

Based on single flag

BRANCH INSTRUCTIONS

"B condition" where the condition refers to the comparison of unsigned numbers. After a compare (CMP Rn,Op2) instruction is executed, C and Z indicate the result of the comparison, as follows:

	C	Z
Rn > Op2	1	0
Rn = Op2	1	1
Rn < Op2	0	0

Based on multiple flags

BRANCH INSTRUCTIONS

Instruction		Condition
BHI	Branch if Higher	jump if C=1 and Z=0
BEQ	Branch if Equal	[redacted] Z=1
BLS	Branch if Lower or same	jump if C=0 or Z=1
BLO	Brach if lower	C=0 and Z=0
BHS	Brach if Higher or same	C=1 or Z=1

For unsigned comparison

BRANCH INSTRUCTIONS

Rn > Op2	V=N or Z=0
Rn = Op2	Z=1
Rn < Op2	V inverse of N

For the signed comparison

Instruction		
BGE	Branch Greater or Equal	V=N or Z=1
BLT	Branch Less than	V is inverse N and Z=0
BGT	Branch Greater than	V=N and Z=0
BLE	Branch Less or Equal	Vis inverse N or Z=1
BEQ	Branch if Equal	jump if Z = 1

BCD to HEX

```
        AREA    MYCODE, CODE, READONLY

        ENTRY

        EXPORT Reset_Handler


Reset_Handler
        ldr r0,=bcd
        ldr r1,[r0]
        and r2,r1,#0x000000f0
        lsr r2,r2,#4
        and r3,r1,#0x0000000f
        mov r4,#10 ; or ox0A
        mla r5,r2,r4,r3
        ldr r0,=hex
        str r5,[r0]

stop    B stop

bcd     DCD 0x98
        AREA data1,DATA
hex     dcd 0
        end
```

Function Call & Return

BL Branch with Link (this is Call instruction)

Flags: Unchanged.

Format: BL Subroutine_Addr ;transfer

BX Branch Indirect (BX LR is used for Return)

Flags: Unchanged.

Format: BX Rm ;BX LR is used for Return from a subroutine

Function Call & Return

```
        BL HEX_BCD  ; call HEX_BCD
        MOV R0,R1
         ------
         -------
HERE B HERE

HEX_BCD MOV R4,R5
        ---------
        ---------
        BX LR ; Return
```

PUSH and POP

PUSH **PUSH register onto stack**

Flags: Unaffected.

Format: PUSH {reg_list} ;PUSH reg_list onto stack

POP **POP register from Stack**

Flags: Unaffected.

Format: POP {reg_list} ;reg_reg = words off top of stack

PUSH and POP

PUSH {R1,R3-R5}; same as PUSH {R1,R3,R4,R5}

POP {R1,R3-R5}

Example:

LDR R13,=0x10000010

LDR R1,=0x12345678

LDR R3,=0x89ABCDEF

LDR R4,=-2

LDR R5,=0x98765432

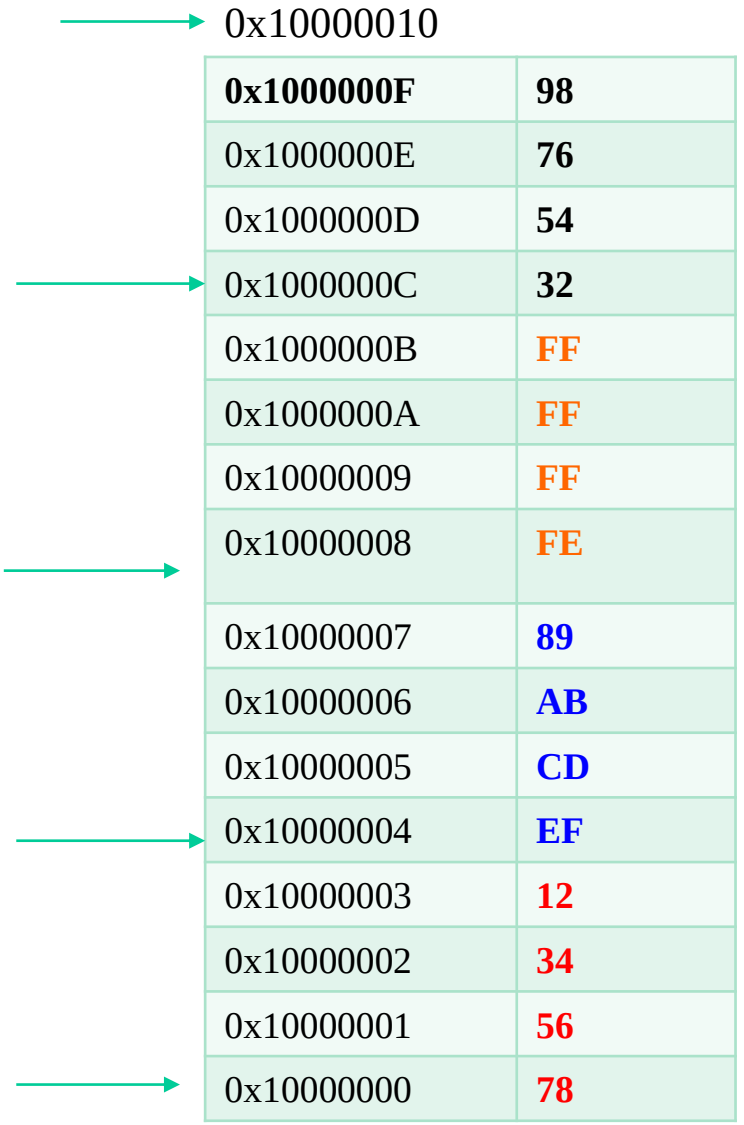
PUSH {R1,R3-R5}

POP {R2,R6-R8}

R13 = 0x10000000

R2= 0x12345678

R6=0x89ABCDEF…

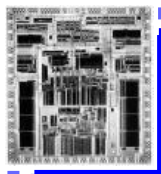


BCD addition

```
        AREA    MYCODE, CODE, READONLY

      ENTRY
      EXPORT Reset_Handler


Reset_Handler
;;;;;;;;;;User Code Starts from the next line;;;;

        ldr r0,=res
        ldr r3,=num1
        ldr r4,=num2
        ldr r1,[r3]
        ldr r2,[r4]
        mov r8,#2
        mov [redacted]
        mov r8,#8
loop    mov r3,r1
        mov r4,r2
        and r3,#mask
        and r4,#mask
        add r6,r3,r4
        add r6,r6,r5
        cmp r6,#0x0a
        blo rcarry1
```

BCD addition

```
        mov r5,#1
        sub r6,#0x0a
        b next
rcarry1 mov r5,#0
next    lsr r1, #4
        lsr r2, #4
        orr r7,r6
        ror r7,#4
        sub r8,#1
        teq r8,#0
        bne loop
next1
        str r7,[r0],#4
        str r5,[r0]
stop    b stop
num1    DCD 0x99999999
num2    DCD 0x99999999
mask equ 0x0f
        AREA data1,DATA
res     DCD 0,0
        end
```



HEX to BCD

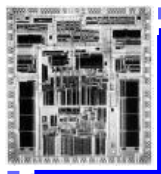
```
AREA  Example4, CODE, READONLY
                        ENTRY

Reset_Handler
                        ldr r0,=hex
                        ldr r2,=rem
                        mov r5,#0
                        mov r7,#32
                        ldr r1,[r0]
up2                     bl divide
                        cmp r1,#0
                        bne up2
                        ldr r0,=bcd
                        lsr r5, r7
                        str r5,[r0]
stop        B stop
```

HEX to BCD

```
divide      mov r3,#0
up1                 cmp r1,#0x0a
                    blo down
                    sub r1,#0x0a
                    add r3,#1
                    b up1
down
                    orr r5, r1
                    ror r5,#4
                    mov r1,r3
                    sub r7,#4
                    bx lr
hex                 DCD 0xfffe
                    AREA data1,DATA
bcd                 DCD 0
```

Convert NEG to POSITIVE in array

```
     LDR R0, =ARRAY
     MOV R1,#10
UP   LDR R2, [R0], #4
     CMP R2, #0
     RSBLT R2, #0
     STRLT R2, [R0, #-4]
     SUB R1, #0
     TEQ R1, #0
     BNE UP
```



Factorial using Recursion

```
AREA  Example4, CODE, READONLY
                ENTRY
Reset_Handler
                ldr r1,num
                ldr r13,=0x10001000
                bl fact1
                ldr r1,=fact
                str r2,[r1]
stop            b stop
fact1           cmp r1,#1
                beq exit
                push {r1}
                push {lr}
                sub r1, #1
                bl fact1
```

Factorial using Recursion

```
                pop {lr}
                pop {r1}
                mul r2,r1,r2
                bx lr
exit            mov r2,#1
                bx lr
num     DCD 0x07

                AREA data1,DATA
fact    DCD 0
                                end
```