Microprocessor based digital design (CST 503) (3L+1T)/week FM 100

Course objective

- Understanding the organization and architecture of real-life low cost control/computing devices.

- How a processor based design can cut cost, offer flexible solution and help rapid development of digital systems through modular design concepts.

- To learn assembly level language for optimum program performance and control operations.

Syllabus

- Architecture and organisation of typical CPUs.
- Programming Model and Assembly language
- Machine/Instruction cycle and Timing
- Interfacing memory & I/O devices
- Interrupts (hardware and software)
- Serial communication
- Peripheral devices and controllers (PIO, PIC, Counter/Timer, DMA)
- Microcontrollers
- Advanced microprocessors
- Small system design

References

- Microprocessor Based Design: A Comprehensive Guide To Effective Hardware Design : Michael Slater, PHI
- Advanced Microprocessors: D. Tabak, Penram International Publishing (India) Pvt. Ltd.
- Microprocessor Architecture, Programming and Applications with the 8085 : R. Gaonkar, Penram International Publishing (India) Pvt. Ltd.
- Manuals (microcontrollers) available in the net

Course requirements

Pre-requisites to this course are

i) knowledge of

- Digital logic; principles and design.
- High level language programming.

ii) core concepts of

- Computer Organisation/Architecture

iii) and rudimentary idea of

- Operating System

What is a Microprocessor?

- It is usually a single IC device that contains the CPU of a typical computer. So, a microprocessor is a CPU in a single chip.
- Major functional units of a microprocessor are:
  - Control Unit (CU)
  - Arithmetic and logic unit (ALU)
  - Instruction Decoder (ID)
  - Small high speed memory (Registers)
  - Bus interface units (BIU)
  - Buffers, Cache and different pipelines

These functional units are connected through internal buses and exclusive data paths.

Why Microprocessors?

Microprocessor based design has practically outdone all discrete IC based non-CPU oriented design due its many faceted advantages.

• Low cost high speed CPUs in a single chip ensures highly reliable and flexible system.

•Easy to develop a full system from scratch due to the availability of low cost compatible programmable peripherals.

• The use of suitable s/w to make the same h/w all pervasive with respect to the different system requirement.

Microprocessor: Advantages

- The development of h/w and s/w is very much modular and can be developed in parallel cutting the development time.

- Testing is easy and modular

- Advancement in VLSI has led to the development of all pervasive microcontroller (Microprocessor + RAM + RAM + I/O + Timer + other special facilities) in a single chip that is now used in all embedded design.

Microprocessor: Architecture

Three common architectures:

- Stack machine
  - All operations are done on stack-top (0-address machine). This is an obsolete architecture.
- Accumulator based
  - Old architecture with limited hardware resources.
  - One register (Accumulator) is used to hold one of the operands and the result (1-address machine).
- General Register
  - All registers are equally powerful.
  - All modern processors are of this type (3-address machine).

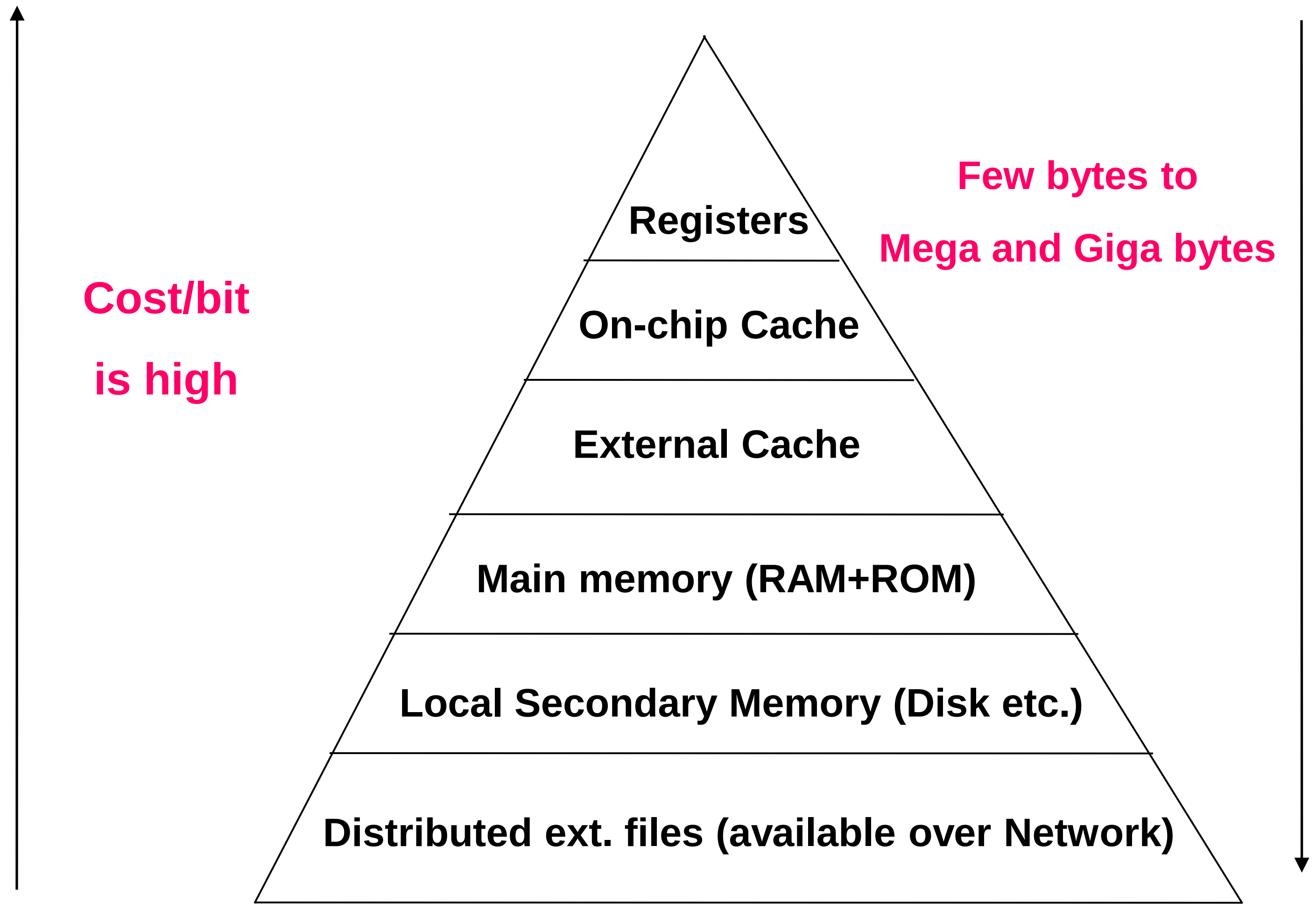
Processor: Resource and action

- Irrespective of the architecture, the main resource of any processor is the memory in which instruction and data are stored. A processor is connected to this resource through Buses; Address and data buses.
- The processor does very simple external (i.e., bus) operations; namely, bus read and bus write in order to fetch instruction and data from the memory and to write the computed results back into the memory.

Memory: The main resource

- Memory holds everything; data, instruction and any other information.
- By the term memory we mean the external semiconductor memory (RAM + ROM).
- CPU usually has a small amount of very high speed memory, known as register.
- Registers hold data and addresses to reduce CPU memory (external) interactions for faster processing.
- Modern CPUs also have internal cache memory.

The Memory Hierarchy



As we move from the base to top access becomes faster and faster.

CPU Memory interaction

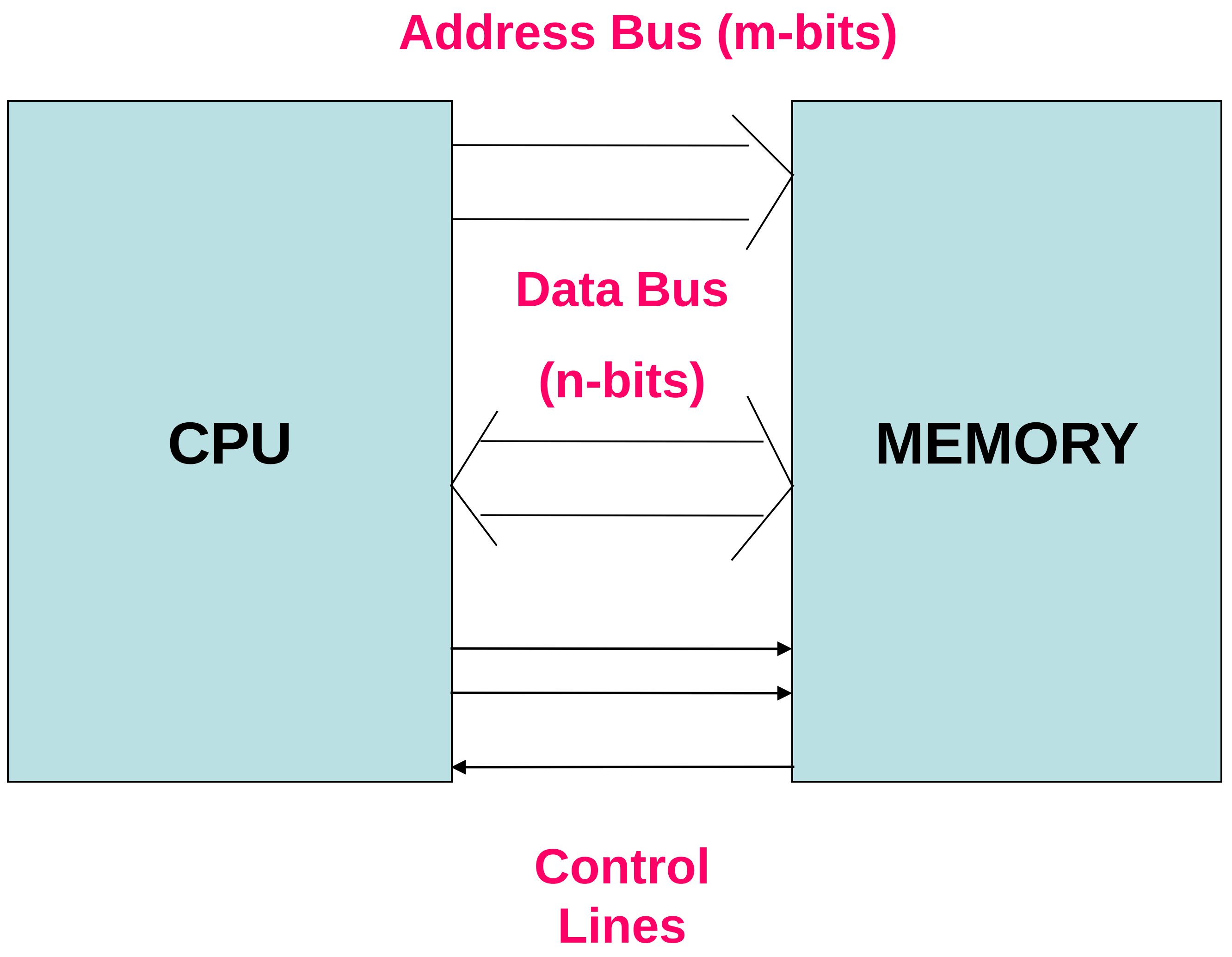
- The processor, as the bus master, generates the address of the memory location on which either a read or write operation will be performed. Thus, the address bus is unidirectional; carrying address information from CPU to memory.
- What is to be read comes from the memory to the CPU (via data bus) while what is to be written is supplied by the CPU and that goes to the memory via the data bus; thus the data bus is bi-directional.

Control Lines (Bus)

Other than the address and data buses we need some more lines to synchronise and dictate bus operations. These are known as control lines. As you can guess most common lines are read, write, and various status lines.

Besides these control lines some special lines are also available to carry out interaction with DMA, PIC and other controllers. We will see their applications later on.

Processor & Memory



Microprocessor: Word size

- Arithmetic and logic operations are carried out in the Arithmetic and Logic Unit (ALU).
- The size of the operands in bits is known as the word size.
- If a processor can process N bit operand at a time (for majority of its instruction); the word size is N and consequently the CPU is known as an N-bit processor.
- These days, N happens to be an integral multiple of 8. So, 8, 16, 32 or 64 bit CPUs are commonly available.

Word size vs. Bus size

- Address bus size dictates the memory address space (2^m for m bits; e.g., 64K for 16 bits).
- Data bus size determines the number of bits that can be read/written in a single bus cycle. It is usually equal to the word size of the processor; i.e., N bits for N-bit CPU. However, exceptions are there and the CPU reads/writes word data through ½ Word size (or less) through multiple bus cycles. Performance penalty is compromised for the ease of backward integration with the available peripherals as well as for cost benefits.

Should we start with 8-bit

- 8-bit processors, though not very powerful, are ideal to learn the basic principles and mastering the hardware design due to their simplicity, low cost and high availability of the support systems.
- As such we cannot say that a particular processor is the best one; there are so many with different attributes and varying computing power.
- A particular processor may be suitable for a particular application.
- However, the underlying principles are same for all processors.

Hypothetical or real life CPU

- We will use INTEL 8085A 8-bit microprocessor for learning the basic principles. It is a low cost simple CPU with high availability of all sorts of hardware and software support for learning and development systems.
- Sometimes, a hypothetical, ideal, all powerful CPU is considered for training. However, this approach suffers from the non-availability of all types of real life development and testing facilities; simulation can only be done.

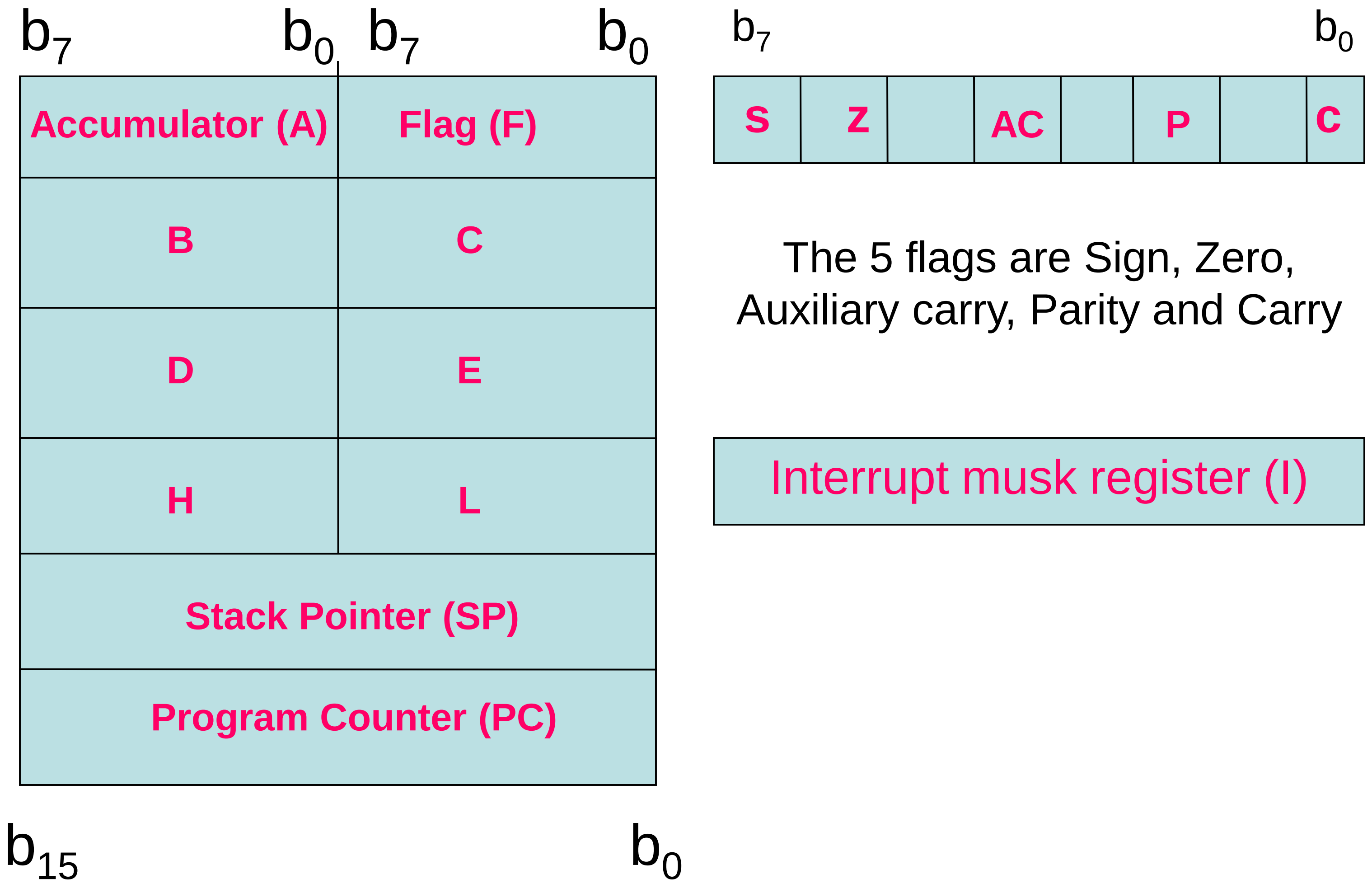
Hardware and Software

- For small system design (most usage fall in this category) hardware development is easy as off the self and compatible components and peripherals are available from the vendors and the design is more of an assembly of functional blocks.
- Software development calls for more effort (70% or more) in the development cycle of a product.
- S/W for small system is developed (usually) in assembly language; a pre-requisite to this is to know the processor programming model.

Processor Programming Model

- Processor programming model is the graphical representation of different registers whose contents can be manipulated through machine instructions.
- For assembly language programming we also need to know the instruction set and the addressing modes available for a particular CPU.
- Instruction set is the set of machine instructions for computing and other operations while the addressing modes dictate different ways of getting the operands from memory to CPU.

8085A: Programming Model



The 5 flags are Sign, Zero, Auxiliary carry, Parity and Carry

8 bit register pairs (drawn side by side) may be used to hold 16 bit operands

Instruction Set

Instructions may be classified into 4 major groups

- Data Movement
  - Any movement between registers, register to memory or memory to register.
- Arithmetic and Logic
  - Any processing involving the ALU (or Register).
- Branch
  - All jumps, calls, returns & s/w interrupts; i.e., whenever we break the normal sequential flow of execution.
- Special or Machine Control
  - Any instruction other than the first three types.

Instruction Set: Examples

Here is a non-exhaustive list of instructions.

- Data Movement: MOV, MVI, LXI, PUSH, POP, XCHG, XTHL, IN, OUT
- Arithmetic and Logic: ANA, ORA, XRA, CMP, ADD, ADI, ADC, SUB, SUI, SBB, INC, DCR, INX, DCX, DAD, DAA, CMC, STC
- Branch: JMP, JZ, JNZ, JC, JNC, JPO, JPE, JP, JS, CALL, CZ, CNZ, CC, CNC, RET, RZ, RST
- Special or Machine control: HLT, NOP, EI, DI, SIM, RIM

Data Movement

Some examples:

```
MOV    A, B; Register to Register
XCHG

MOV    M, C    ; Register to Memory
PUSH  B

MOV    D, M    ; Memory to Register
POP        H
```

Arithmetic and Logic

Addition, subtraction and all standard logic operations are available. Moreover, instructions for increment, decrement of register value or memory locations are also present.

Multiplication and Division instructions, however, are not available. Some examples:

```
ADD    B  ; add without carry(A<- A + B )
ADC    B  ; add with carry (A<- A+B+Cy flag)
ADI    A, 5  ; A<- A + 5
ADD    M     ; A <- A + [HL]
```

Logic instructions

AND, OR, XOR and complementation (of Acc. register) can be done. Carry flag can also be changed through explicit instructions.

```
ANA    B
XRA    L
CMA         ; complement accumulator
STC    ; set carry
```

Branch Instructions

- All conditional and unconditional Jumps, subroutine calls and returns as well as the software interrupts are used to deviate from the sequential execution flow where the next instruction in memory is not executed and the execution control is diverted to the instruction stored elsewhere in the memory. e.g.,

JNZ Loop1; CALL Delay

RET ; PCHL; RST 5

Special and Machine Control

These instructions usually have no memory operands and are used for special purpose. Notable examples are;

NOP	; No operation -- good as place holder
HLT	; Halt operation – CPU is logically dead
EI (DI)	; enable (disable) interrupt mechanism
SIM (RIM)	; set (read) interrupt mask

Addressing Modes

The 8085A has the following addressing modes

- Direct
- Immediate
- Register
- Register Indirect

Unfortunately one of the most important modes, namely, Relative mode addressing is not provided.

Other than the 4 basic modes stated above special and machine control instructions which do not work on some operands use Implied addressing mode.

Direct Addressing Mode

Here the operand address is directly (fully) specified in the instruction. For example;

STA 2050H; encoding →32 50 20

Note that this is a 3 byte instruction; where the 1st byte specifies the op-code and the next 2 bytes specify full 16-bit address of the operand in little-endian form.

While it is simple, this mode has disadvantages; e.g.,

•Address is encoded directly in the instruction; so manipulation is not possible.

•All m-bits (address) must be present in the instruction; thus the instruction length will be m + op-code bits.

Immediate Addressing Mode

Registers or memory locations are to be initialised frequently in a program according to the need and this is done by this mode. Here the operand immediately follows the op-code in the instruction. For example;

```
MVI  A, 5          ;    3E 05
LXI   SP, 0FBAAH   ;    31 AA FB
```

Note that the 2 (3) byte instruction is formed by the 1 byte op-code which is followed by the 1 byte (2 byte) immediate operand i.e., 05 (0FBAAH).

Register Addressing Mode

Operand (data or address) is available in register(s). For example;

ADD B; $A \leftarrow A + B$

INX H; $HL \leftarrow HL + 1$

PUSH H; $(--SP) \leftarrow H, (--SP) \leftarrow L$

DAD B; $HL \leftarrow HL + BC$

Register Indirect Mode

Here the address of the operand is available in the register(s). This is advantageous for address manipulation. No. of bits (In 8085A only 3-bit is necessary to specify any 8 bit register) required to represent register(s) is also much lower than full m-bit address. e.g.,

```
MOV    A, M ; Move the content of location M
          ; ( held in HL registers) to A
PUSH  H  ; Push H & L into stack top
whose           ; address comes from SP register.
```

Branch Instruction

Execution of instructions is usually sequential. However after executing n_{th} instruction we may not execute the very next, i.e., the $(n+1)_{th}$ instruction. This is done by branch instruction; branch is either conditional or unconditional. Examples:

Unconditional branch

```
JMP     508AH; go to location 508AH
PCHL    ; Load PC with HL : basically jump to
   ; location stored in HL unconditionally
```

Conditional branch

```
JZ      to_addr; jump if Z flag = 1
```

Special or machine control

Some instructions do not really need any memory or register operand and are used for special purpose. Examples:

NOP	; No operation
HLT	; Halt

Addressing mode for all such instructions is known to be implied mode.

Summary of Instruction set and addressing modes

Instruction set:

OK for 8-bit low cost CPU; however multiplication and division instructions are not available. Swap instruction is not available as well as true shift operations. Overflow flag should be present. Flag register contents can only be moved to other register through stack only.

Addressing modes:

Register indirect mode is available only in its crude form; offsets cannot be used. A negative point is the absence of relative addressing mode.

Assembly Language Programming

- Assembly language is CPU specific
- It has a very simple form; a single line contains a single instruction
- An instruction contains 3 parts
  - ***label*: a symbolic reference to a location**
  - ***op-code*: part of the machine instruction specifying the type of operation (movement, branch etc.) to be performed.**
  - ***operand*: one or more operands on which the operation will be done.**
- Comments (starts with a ; i.e., a semi-colon) can also be added anywhere to improve readability and are ignored by the assembler.

Instructions and addressing modes

- Note that an instruction class (say, data movement) is available for many addressing modes; e.g.;

```
MOV  B, L        ; register mode
MOV  C, M        ; register indirect mode
LDA  0B035H      ; direct mode
LXI      SP, 20FFH      ; immediate mode
```

- Ideally each type of instruction should be available for all addressing modes and all registers should hold operands for all instructions. However, this orthogonal property may not be observed in all real life CPUs.

Assembly instructions: examples

```
      LXI   SP, 20FFH ; op-code and operand only

      NOP             ; op-code only

L1:   ADD   B         ; label, op-code and
                                ; operand

      END             ; a pseudo op-code (or
                           ; assembler directive)
                      ; indicating the end of the
                           ; source file
```

A Program

```
      LXI     SP, 2FFFH; initialize SP
      XRA     A      ;  A←0 and carry←0
      MVI     C, 10        ; loop
count                MVI         B, 1  ; start value of
the term
L1:   ADC     B      ; add with carry; if any
      INR     B      ; generate next terms
      DCR     C      ; decrease loop
count         JNZ          L1     ; If (Count <> 0)
redo
      RST     1      ; return to monitor
```

Op-code

- op-code may be a m/c instruction or pseudo-opcode. Pseudo opcodes (also known as assembler directives) do not generate m/c code but informs the assembler to take some special action. Pseudo-opcodes are used to control the PLC or define constant, data, macros etc.
- op-code (or pseudo) part of an instrcution is mandatory while label and operand are not.

Operands

- Operands are part of the machine instruction specifying the register(s), address or data.
- Some instructions have only the op-code part.
- Sometimes special symbols are used in operand part to indicate the addressing mode;

  e.g., # indicates immediate addressing and

  @ represents register indirect addressing.
- Number of operands depend on the type of instruction and the machine architecture (1-address or more)

Label

- Symbolic reference for a location.
- Assembler collects all such symbols to create symbol table for code generation.
- Frequently used constants are also represented by symbols for visibility and better comprehension. These are literals and also find place in symbol table.
- In Intel assemblers the label name string is followed by a colon (:). This helps identify the labels easily.

Pseudo op-codes

There are so many pseudo operations (or directives) for any assembler. However they may be classified to:

- PLC control
- Constant definitions
- Data definitions

Other than the above there are usually many macro related directives used in any macro-assembler.

Assembler directives

- PLC control
  - ORG (origin) is used to initialize the PLC; e.g.,

    ORG 2000H; Next byte of code/data will be

    ; assembled from 2000H
  - $ is to get the current value of the PLC; e.g.,

    ORG $+16; PLC $\leftarrow$ PLC + 16

($ is appearing in operand field; a special use)

Constants

EQU (equate) is used to define a constant, e.g;

```
PORTA       EQU   30H
DELAY_VAL   EQU   1234H
```

Once defined through EQU a constant cannot be changed in the same program. Directive SET may be used where change in values take place in the same program; i.e., we can write:

```
  SYMBOLX   SET   15H ; and then
            ; elsewhere
  SYMBOLX   SET   10H
```

Data directives

Two directives, DB (define byte) and DW (define word), are used in 8085A. DB is also used in representing strings (byte array). Example:

```
 DB 5, 7Q, 0FAH, 11100001B ; 4 consecutive
                           ; bytes in decimal,
                           ; octal, hex  and
                           ; binary form
DB   'ABC'; generates 3 bytes; 41 42 43 in hex
DW   12FAH     ; encoded in little-endian form.
```

A sample program

```
VAL       EQU     10
          ORG     2000H
          MVI     C, VAL ; Set counter
          SUB     A         ; A← 0; CY flag ← 0
          LXI     H, myData     ; get a pointer (HL) to
                                ; the byte data aray
L1:       ADD     M         ; Add first term
          INX     H         ; Adjust data pointer
          DCR     C         ; Decrease count
          JNZ     L1        ; More to ADD?
          RST     1         ; Return to OS

          ORG     $ + 10H  ; Current PLC + 16
myData:   DB      1, 2, 3, 4, 5, 6, 7, 8, 9, 10
          END
```

Subroutines (Procedures)

The key to build a modular program is to develop the program using a number of small but useful subroutines which may be used for other tasks or can be shared with others. An example

```
delay : this is a routine to produce delay
; in : delay value in DE pair
; out : None
; destroys: A

delay:      DCX     D    ; DCX has no effect on  Z flag
        MOV     A, E
        ORA     D    ;  A <>0 if DE has a single 1 bit
                ; in any bit position in D or E
        JNZ     delay    ; Z = 1 if A = 0
        RET
; the dealy routine can be called  from the main program as
        LXI  SP, 08000H   ; assuming RAM ends here
        LXI  D, 20F1H ; load delay value in DE pair
        call delay
```

Subroutine contd.

From the subroutine return to the calling program is ensured by

- Putting a RET instruction at the end of the subroutine.
- RET instruction pops off the return address from the stack (portion of r/w memory).
- CALL instruction to subroutine has automatically initiated pushing of return point in the caller (i.e., address of the instruction next to call).
- Initialise SP to a free area in R/W memory ensuring storage for the return address

Stack and Stack pointer

- Stack is a LIFO data structure and grows downward.
- SP is a register that points to the stack top.
- SP should be initialised to the top of the RAM for maximum available stack area.
- Stack is used to i) store return address of the caller; ii) store temporary values and iii) pass parameters and return values to and from the procedures.
- PUSH instruction pushes a word in the stack top and POP gets back the word from the top.

Macros

- Macros, like subroutines, are useful for modular development of programs.
- Macro is basically a text expansion facility where the reference to a macro (macro call) is replaced by the body (i.e., macro expansion) of the macro.
- Macro is quite powerful if we consider parameter passing and conditional macro expansion.
- Other than the normal use macros are needed in simulating a new language and for customized system generation for different hardware.

Macro: text expansion

Consider the following macro which does NAND operation with the contents of A and B.

```
NANDB      MACRO
      ANA    B
      CMA
      ENDM
```

- Name of the macro is NANDB.
- MACRO and ENDM are directives indicating the beginning and the end of the Macro.

Calling a Macro

Calling a macro is done by referring to its name as a pseudo op-code in an instruction. e.g.,

```
NANDB
NOP
NANDB
```

Would be expanded by the macro-processor as:

```
ANA    B
CMA
NOP
ANA    B
CMA
```

So each call is replaced by the macro body. Note that call and return overheads of the subroutine are absent.

Macro with parameter

```
NAND   MACRO   &R
       ANA     &R
       CMA
       ENDM
```

- This is a more powerful macro and can perform NAND operation with Accumulator and any other 8 bit register. e.g.,

```
NAND   L      ; NAND of A and L
NAND   B      ; NAND of A and B
```

- Multiple parameters can also be used if necessary.

Conditional assembly

Other than copy-code and parameter passing the third facility offered by macro assembler is conditional assembly.

- It is similar to an IF statement in high level language to test a condition and either assemble a block of instruction or bypass it.
- This is useful to configure part(s) of a system software to accommodate different h/w facilities for the same platform. So, conditional assembly is important and useful for the so called system generation.

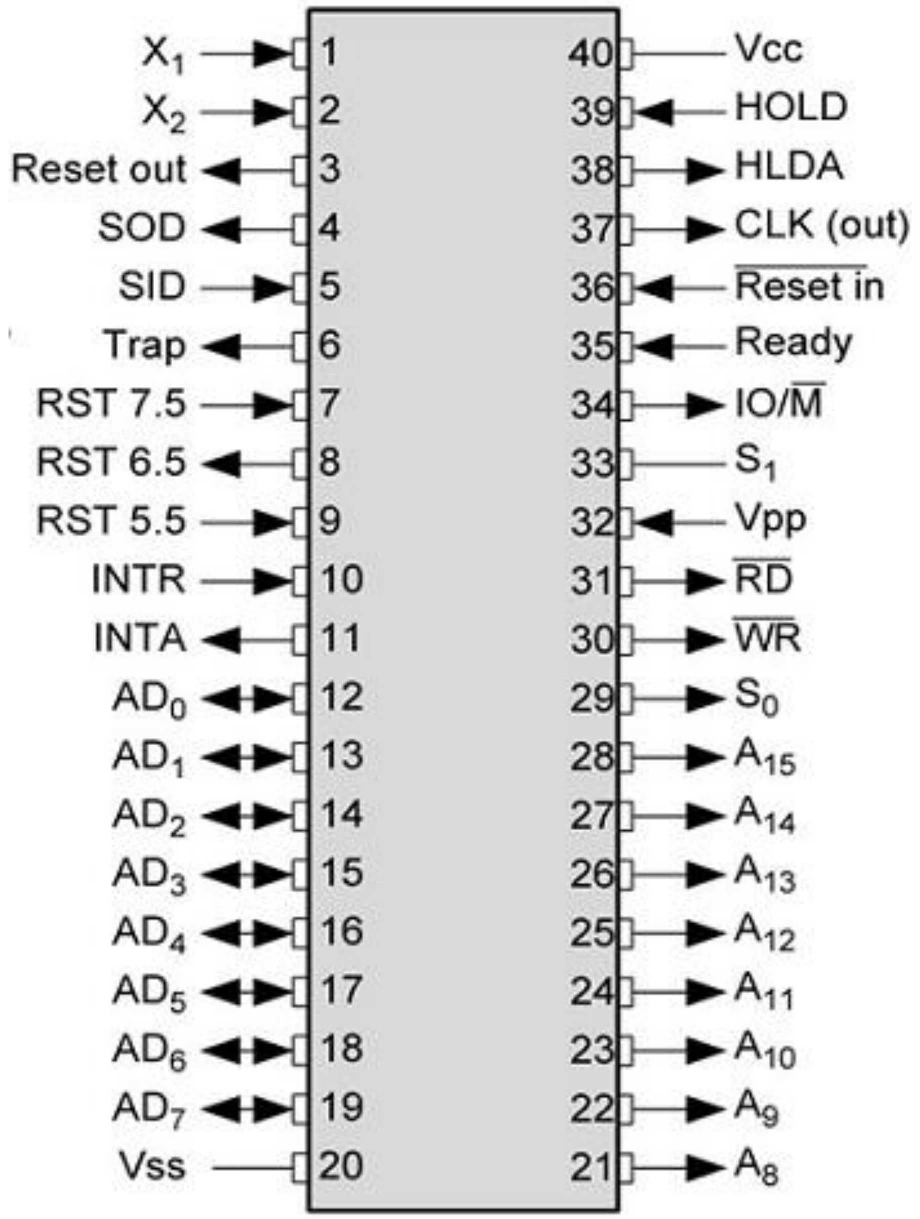
8085A Hardware

- 40 pin DIP package
- 8-bit data and 16-bit address lines
- Separate address space for Memory and I/O
- 5 interrupting inputs
- Multiplexed data and address (lower order) bus
- Special built-in serial ports
- Single +5 V power supply
- On-chip clock generator circuit
- DMA facility
- Ready input for interfacing with slow devices

CPU: Pins (Function-wise)

- Address lines
- Data lines
- Control lines
- Status lines
- Interrupt input and acknowledge lines
- DMA/Bus arbitration lines
- Interfacing with slow peripherals line
- Master reset line
- Clock out, power supply and ground lines etc.
- Special purpose lines

8085A pin diagram



CPU: Pins (Function-wise)

- Address lines (A8-A15)
- Data lines (AD0 – AD7)
- Control lines (RD’, WR’, INTA’)
- Status lines (IO/M’, S0, S1)
- Interrupt input and acknowledge lines (TRAP, RST7.5, RST6.5, RST5.5, INTR, INTA’)
- DMA/Bus arbitration lines (HOLD, HLDA)
- Interfacing with slow peripherals line (READY)
- Master reset lines (RESTETIN’, RESETOUT)
- Clock REF, clcokout, power supply and ground lines etc. (X1, X2, CLKOUT, VCC, VSS)
- Special purpose lines (SID, SOD)

Execution of Instructions

- Each instruction is fetched from the memory.
- Instruction is decoded in ID and decision is taken about the next steps for execution; this includes getting necessary operands from the memory and carrying out the desired operation (e.g., ADD, MOV etc.).
- An instruction contains all information like the operation to be carried out (op-code) and the details of the operands and addressing modes.
- Length of the instruction has no connection with the complexity of the instruction.

BUS operation

- All external activities of the CPU can be explained in terms of Bus operations.
- If we disregard CPU internal operations, execution of instruction too can be seen in terms of BUS operations (BUS read/write etc.).
- BUS operations are represented primarily as BUS READ or BUS WRITE (these are also known as machine cycles)
- Through BUS operation the CPU interact with the memory; hence the basic machine cycles are MEMORY READ and MEMORY WRITE

Machine cycles

- Instruction cycle consists of a number of machine cycles
- Basic machine cycles are read and write
- Intel 8085A defines 7 machine cycles
  - **Op-code Fetch (OF)**
  - **Memory read (MR)**
  - **Memory write (MW)**
  - **I/O read (IOR)**
  - **I/O write (IOW)**
  - **Interrupt acknowledge (INA)**
  - **Bus idle (BI)**
- These 7 cycles may be mapped to 2 cycles only.

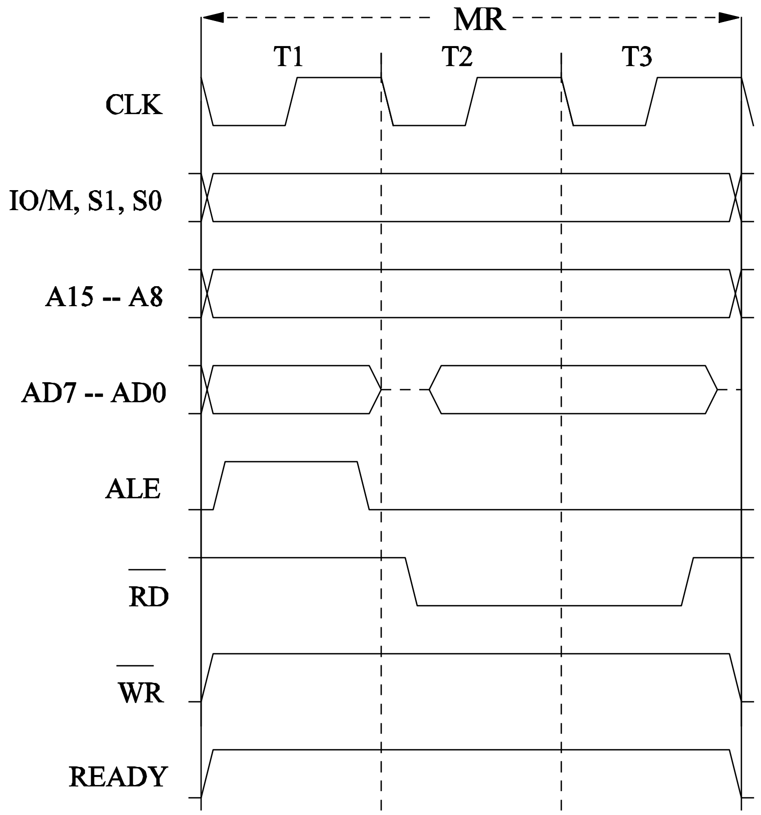
Machine and BUS cycles

Considering the bus activity the 7 machine cycles can be mapped either as memory read or memory write due to the following facts.

- I/O devices are logically equivalent to memory device.
- Bus activity is stopped during BI; so it may be ignored as an external operation.
- OF machine cycle is same as MR with extra time needed to decode the instruction during which bus activity is stopped.
- In INA machine cycle memory read operation is done to get the device identification number.

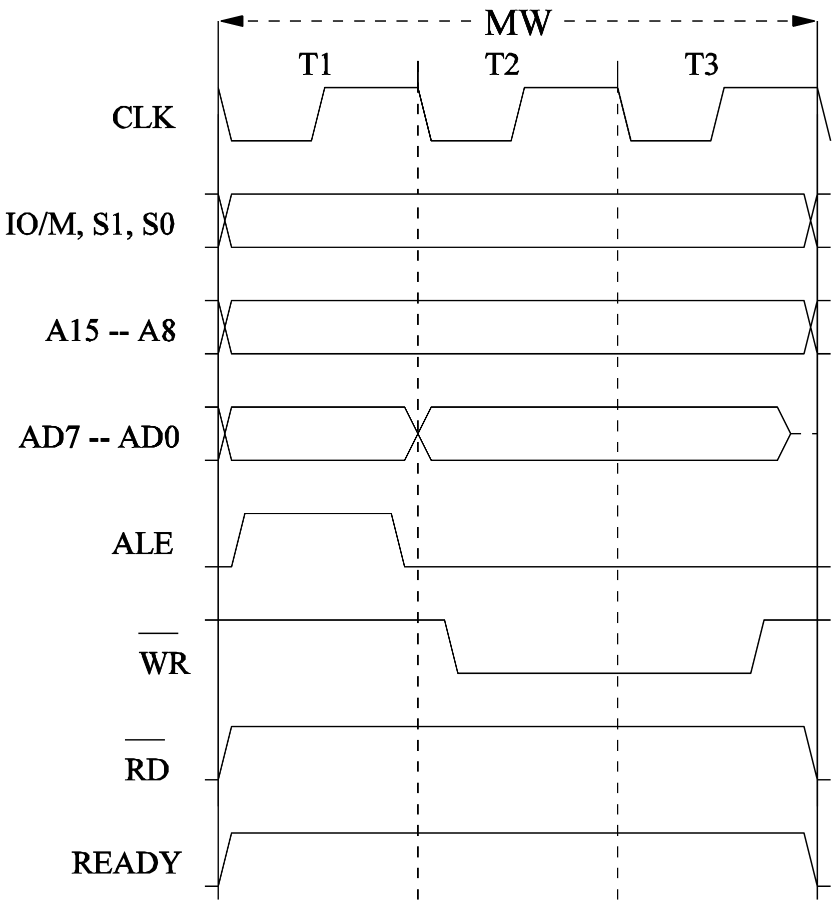
Memory Read

- It takes 3 T states
- Low order address is latched during T1
- Memory needs time to drive data bus with the required data during which the data bus is in High Z state
- Throughout the cycle Write and Ready lines are high.



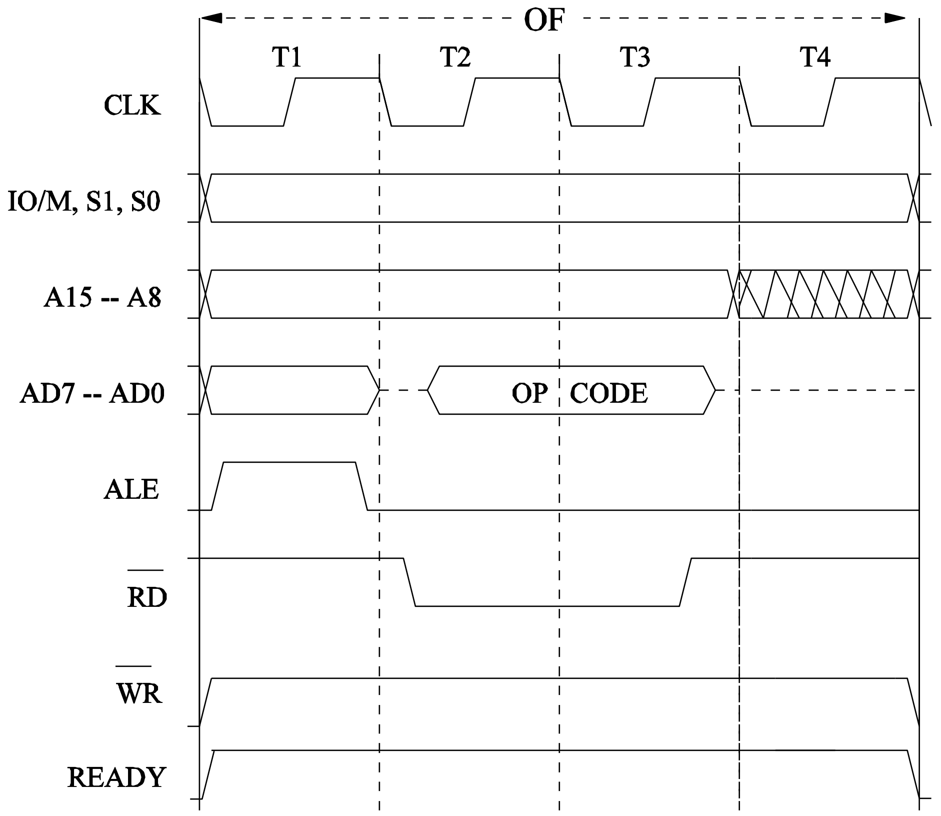
Memory Write

- It is similar to MR cycle.
- Address and Data, both are supplied by the CPU.
- Data is available in the BUS right from the beginning of T2; so there is no high Z state between T1 and T2 (a notable difference with MR where Memory supplies data and it takes time to drive the data bus and hence there is a high Z state between T1 and T2)



Op-code Fetch machine cycle

- It takes 4T states
- Operation in the first 3T states is similar to MR
- One more T state is required to decode the instruction.
- Many instructions do not take extra time as they call for internal operation and that can be carried out just after the decoding. OF for a few instructions take 6T for complex internal operations.



Machine cycles/Instruction

- Each instruction consists of a number of machine cycles (Min. 1 to Max. 6).
- Each machine cycle consists of a number of clock cycles (3, 4 or 6); e.g.,

  STA 2050H ; encoding 32 50 20
- This instruction consists of the 4 machine cycles (OF, MR, MR and MW) and 13T states.
- OF fetches the op-code (32H); CPU knows that it requires 3 more machine cycles; two MRs to read the address (50H and then 20H) and finally an MW is needed to write the contents of A register in the direct address 2050H.

Machine cycles/Instruction

For instruction MOV A, B; (code 78H)

- We need only the OF (4T) machine cycle.
- The opcode is fetched within the first 3T states.
- In next T state the code is deciphered and contents of B is copied to A.
- As the registers are internal to the CPU it is possible to decode and execute within 1 T.

However, MOV A, M; (code 84H)

- Takes OF and MR (i.e., 4T + 3T or 7T states)
- OF fetches the instruction that calls for a memory read (external operation) to transfer the contents from memory to A.

Instruction complexity

- Most OF cycles are 4T long; however some takes 6T. To be precise fetching op-code is similar to memory read and takes 3T. However, complex instruction takes more time to decode and even to complete internal operations it takes extra time; e.g.,

  INX D ; it takes only one machine cycle and that is a 6T long OF. Because, increasing the content of a register pair 1T is not enough.

Instruction length vs. complexity

- Complexity and length of the instruction are usually independent. Complexity depends on the number of operands, type of addressing mode and type of operation to be carried out.
- DAD instruction is only 1 byte long. However, it takes 10T to complete it. This extra time is required as it does a 16 bit add operation utilizing 8 bit ALU.
- RET instruction is again 1 byte. It takes 10T states as it involves two MR (reading the stack top) operations to get the return address from the stack.

Interfacing with Memory & I/O devices

- For any workable system memory and I/O devices must be interfaced with the CPU.
- Requirement for a small system design is CPU, RAM, ROM, I/O, Timer and, if required, other special purpose units.
- Due to the availability of a variety of compatible functional units (usually from the same vendor who is offering the CPU) interfacing is now an easy task.
- Basic requirement is the use of a suitable decoding scheme to utilize the memory (I/O) address space according to the system requirement.

Decoders

- Linear decoding of address lines may be utilised for a very simple system.
- Standard 2 -- 4 or 3 -- 8 decoders are very popular for all small system design.
- PROM/PLA based design may also be done
- Fold-back is often allowed to reduce decoding cost.
- Multistage decoding may be employed to prevent bus contention. So, decoder outputs (where input is only the address lines) are often qualified with control signals (read, write etc.).

Address space

- Byte addressability is the basic requirement.
- So any address space is a collection of N no. of bytes as a linear array.
- The address bus lines provide the address of a particular location.
- In some old CPUs (e.g., 8085) address space is divided into memory space and I/O space where the CPU control line IO/M’ segregates the space (0 indicates memory access and 1 indicates I/O access).
- Modern CPUs have a single space (Memory).

Putting devices in the address space

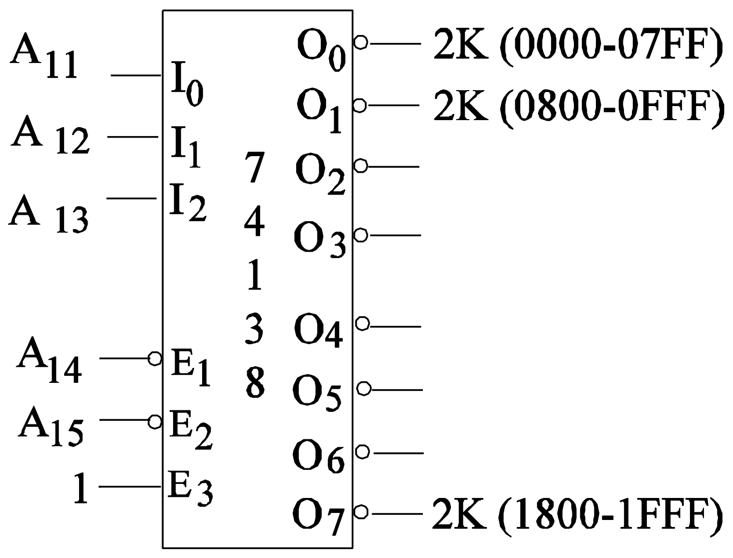
- A device requires a fraction of the total address space provided by the CPU. Moreover, a single memory or I/O device may not be suffice for any real life system.
- Each device thus has a master control line (namely the chip select line) which has to be asserted to use the device.
- At any point of time the CPU talks to a single device.
- Decoding circuitry generates separate chip select signals for non-overlapping chunks of the address space.

Linear decoding

- CPU address lines may be directly used to select the devices (ICs).
- Advantages are simplicity and low cost
- Disadvantages are the division of address space and its improper utilisation.
- Each address line divides the available address space by half. For example if we use A15 as the chip select we immediately reserve full 32 K for that particular device.
- Rest of the space i.e., 32 K will then be available for addressing all other devices.

Using Decoder

- Using decoders we can use smaller chunks of address spaces.
- Figure shows selection of 16K space (lower 16K out of 64K) using a 3 to 8 decoder, where each output selects 2K space.
- 8 devices may be selected each occupying 2K non-overlapping space.



Decoders

- Standard TTL decoders are dual 2-4 (74LS155/156), single 3-8 (74LS138) and single 4-16 (74LS152).
- Decoders usually have more than one enable lines for cascading (converting low order multiple decoders to single high order decoder).
- Output lines are active low as they normally drive active low chip select or similar control lines.
- Decoder enable lines are also used for qualifying the chip select signals with control signals like read or write.
- Decoder output may also be available as OCC for the ease of wired-and operation facilitating chip select generation for devices with unequal address spaces.

Fold back

If the line used to assert the chip select of a device covers MKB and the device space is NKB where, M > N, we have fold-backs. Fold-backs waste memory (or I/O) space as Multiple logical address sets are mapped into a single physical address set. For example, if the decoder output selects 4KB space (say, 2000H--2FFFH) and the device it selects is only 2KB then the first byte of the device will have two valid addresses; 2000H and 2800H. This will be true for all other addresses as well.

- Fold-back occurs when we do not consider all the higher order address bits for decoding.
- For small system with less chance of expansion fold back may be a deliberate choice to reduce decoding cost.

More on assembly language programming

Use of stack is one of the important aspects of assembly language programming. Note:

- Stack is the portion of read-write memory.
- CPU has a register (Stack Pointer) which always points to the current stack top.
- Stack grows downward; when you push a word in the stack SP is decreased, the word is written. When you pop a word it is the reverse operation; i.e., you read the word and SP is then increased.
- So it is a LIFO; i.e., word pushed last is popped out first.

Use of stack

Stack is used for the following purposes:

- Storing the return address
  - When you call a subroutine the execution control goes from the calling routine to the called routine. Prior to this branch, return address is automatically stored in the stack.
- As temporary storage space
  - Registers are pushed to stack to make room for other operands and are restored when the job is done.
- Passing parameters to the function
  - Stack is used as buffer storage for parameter passing between the called and calling routines.

Saving return address

- This is usually done by the hardware automatically as a response to execute a call instruction.
- The PLC value of the instruction next to the call instruction (i.e., the return point) is pushed to the stack.
- The stack pointer must point to the read/write memory for proper saving of the return address. Thus it is always safe to initialise the SP so as to get a pointer in some free R/W space (stack segment) for all subsequent stack operations.

Getting back to the caller

- While the return address is pushed to the stack automatically the return requires the use of an explicit RET instruction in the called program.
- Many processors store extra information (Processor status word) along with the return address for proper context switch to serve an interrupt. For this reason two different return instructions may be available;

i) RET ; used in subroutines and gets back the return address from the stack

ii) IRET ; used in Interrupt Service Routines to get back the return address as well as the extra words that have been saved by the processor.

Table based conversion

- In many situations we need to convert information from one form to the other and we may take help of a table for such a conversion.
- As an example the students attendance register may be considered as a table which may be consulted to get (determine) the name (or marks obtained in viva-voce of a particular subject) of a student using his/her roll-no as an index to the table.
- Here is a similar example for Binary to Gray code conversion using a table.

Code conversion through a table

; Main prog. calling bin2Gray twice to convert 4-bit binary number

; to corresponding 4-bit gray code

```
LXI     SP, 9000H
MVI     A, 5
CALL    bin2Gray
.
.
.
MVI     A, 15
CALL    bin2Gray
.
.
.
```

```
bin2Gray:       push    B
          push    H
          mvi     B, 0
          mov     C, A
          lxi     H, gTab
          dad     B
          mov     A, M
          pop     H
          pop     B
          ret
gTab:  db   0, 1, 3, 2, 6, 7, 5, 4
         db  12, 13, 15, 14, 10, 11, 8,
9
```

Jump table pseudo subroutine

Implementation of multi-way branch as per the value in A register. If A = 0 jump to RTN0; if A = 1 jump to RTN1 and so on.

```
LXI     SP, 9000H
LXI     H, JMPTAB
MVI     B, 0
ADD     A       ; A ← A * 2
MOV     C, A
DAD     B
MOV     E, M
INX     H
MOV     D, M
XCHG
PCHL
:
```

```
RTN0:   <instr>
:
RTN1:   <instr>
:
:
RTNn:   <instr>


JMPTAB:  DW     RTN0
   DW     RTN1
    :
   DW     RTNn
```

Peripherals

- For any real life computing system we need to connect peripheral devices with the basic CPU-Memory computing backbone. These peripherals are primarily I/O devices of different types. In order to reduce the burden on the CPU off-the-shelf peripheral controllers are also used to interface them. These peripheral controllers under instruction from the CPU increases the throughput of the system by making the CPU free from routine work of a structured and synchronised approach to respond to the need of these devices and to ensure a fair and maximum utilisation of them.

Peripheral controllers

- Practically for any I/O device we thus need peripheral controllers.
- These controllers are programmable; i.e., the CPU initializes them and send basic operating instructions.
- Peripheral controllers are logically the extended arm of the CPU relieving it from the daily chores of managing the need of the I/O devices connected to the system.
- Off the shelf peripheral controllers are PIO, PIC, DMA, SIO, CRT and Keyboard controllers.

Interrupts

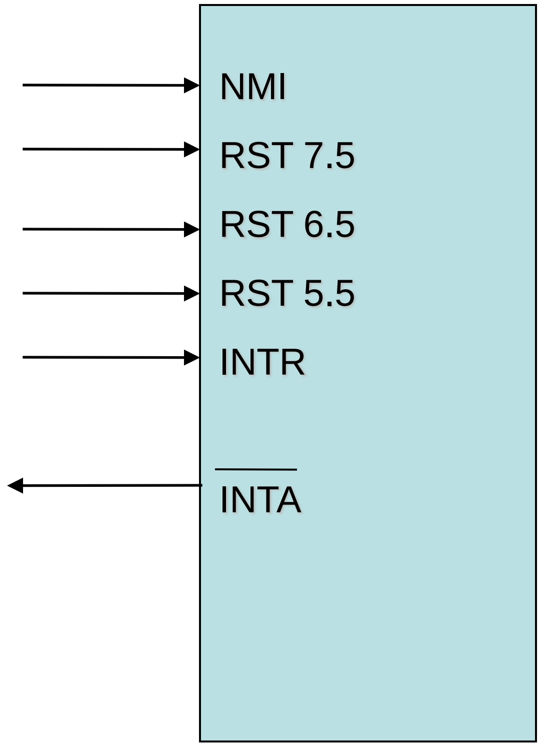
- Interrupts are requests made by the external devices to get some service from the CPU.
- CPU after getting interrupt request suspends the current task, identifies the interrupting device and executes a selected Interrupt Service Routine (ISR) to serve the device.
- The suspended task is resumed once the ISR is done.
- Other than the external h/w interrupts s/w interrupts and internal h/w interrupts are also posssible.

External H/W Interrupts

- By the term interrupt we normally mean External hardware interrupts from the I/O devices.
- These are asynchronous external event and are used to increase the I/O throughput of the system.
- In case of multiple simultaneous interrupts the CPU applies some priority logic to decide whom to serve first.
- CPU may not accept or ignore any interrupt requests if it is busy doing something important.
- A non-maskable interrupt input is usually available in the CPU.

Interrupt lines 8085A

- There are 5 interrupting input lines.
- NMI being the highest priority line and Non-maskable while the others are maskable.
- INTR is the lowest priority line and non-vectored; others are auto-vectored.
- $\overline{\text{INTA}}$ is the acknowledgement line and used in Interrupt Acknowledge Machine cycle. INTA cycle is basically a pseudo OF cycle where $\overline{\text{INTA}}$ signal replaces $\overline{\text{RD}}$.



Interrupt related instructions

There are 5 interrupt related instructions in 8085.

They are

1. DI or disable interrupt : This instruction disables the interrupt inputs except the NMI.
2. EI or Enable interrupt: Enables the interrupt system.
3. SIM (Set interrupt mask) : Hardware non NMI lines can selectively enabled or disabled through appropriate mask bits.
4. RIM (Read interrupt mask): Read the interrupt mask bits and pending interrupts.
5. RST <n> is the software interrupt insruction.

Software Interrupts

- An RST n is a software interrupt.
- The byte n may take any value; 0 to 7.
- The instruction next to the Restart (RST) is considered as the return point and this is pushed to stack once RST n instruction is encountered.
- Control is transferred to location n x 8. ISR should start from that location.
- Two instructions EI and DI are used for enabling and disabling the interrupt mechanism.
- Interrupt mask bits are stored by SIM and are read by RIM instruction.

H/W Interrupts (contd.)

- It is difficult for a dumb I/O device to provide the code for the long call (0CDH) instruction followed by the address of the ISR in three successive INTA machine cycles.
- A programmable Interrupt Controller (PIC) is a better choice which however is not cost effective in small systems.
- In most of the current CPUs a vector table is maintained and the device is supposed to supply an offset (a byte) to this table which holds the addresses of the ISRs.

Interrupts (contd.)

- In 8085 the four (RST 5.5 to TRAP) higher priority lines are auto-vectored. So the starting point of the ISRs are automatically defined.
- Interrupt through INTR line needs elaborate arrangement as the interrupting device is supposed to supply a byte value which is considered to be an op-code.
- So we have only two options; i) Use a long CALL or, ii) a short CALL instruction to branch to ISR and force the CPU to store return address in the address.

Interrupts: Priority and Masking

- Priorities are sometimes predefined in case of multiple interrupt input lines; e.g., TRAP is the highest priority line, followed by RST 7.5 and so on; and INTR is the lowest priority line.
- A physical proximity based h/w priority interrupt mechanism known as Daisy Chain is very popular.
- Interrupt inputs can be masked out. However, a Non Maskable Interrupt (NMI) is common which has the highest priority and cannot be masked out. NMI is used to track very important event like power failure or other exigencies.

Microcontroller

With the improvement in VLSI technology which offers an ever increasing no. of transistors to be integrated in the same silicon the designers had two choices in 1990s:

1) Design more powerful (16, 32 or 64 bit) CPUs for the computational need and

2) Integrating all small system requirements i.e., CPU + MEMORY + I/O etc. in a single IC.

Both were actively pursued by the designers and now we have extremely powerful CPU in a single chip as well as specially designed CPU along with all other functional units together for all sorts of embedded applications.

Microcontroller contd.

The idea of getting everything together had given a new opportunity for the designer in the form of Microcontroller. A microcontroller is a small systems designers best choice to optimize the design requirements. Advantages over the conventional CPU based designs are outlined.

Single chip solution leads to have

- less component, less power requirement
- more reliability
- lower cost
- less design time etc.

Microcontroller

MCS-48 series from INTEL was introduced in early eighties. This family has a number of devices with varying hardware capabilities.

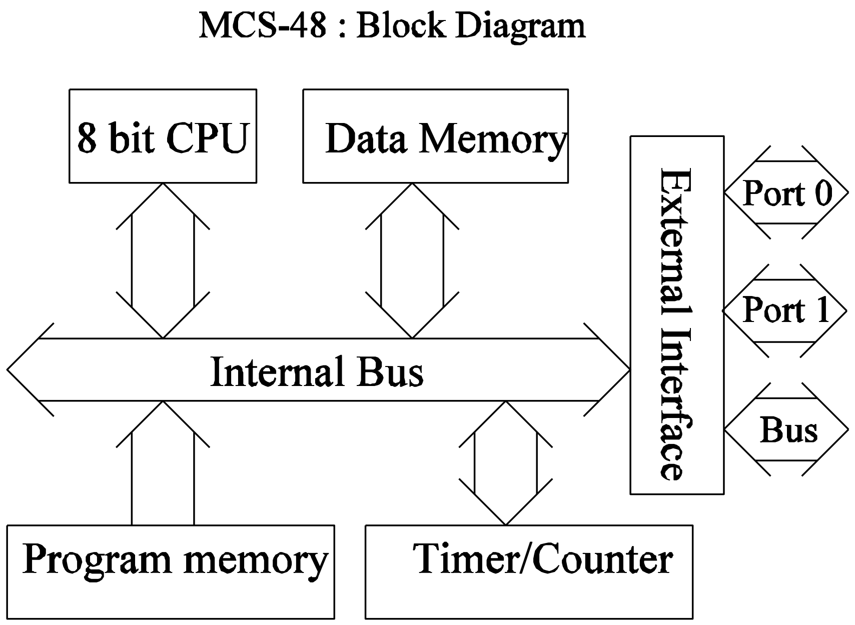
Device	Program Memory	Data Memory
8050AH	4K x 8	256 x 8
8049AH	2K x 8	128 x 8
8048AH	1K x 8	64 x 8
8749H	2K x 8 (EPROM)	128 x 8
8040AHL	none	256 x 8
8039AHL	none	128 x 8

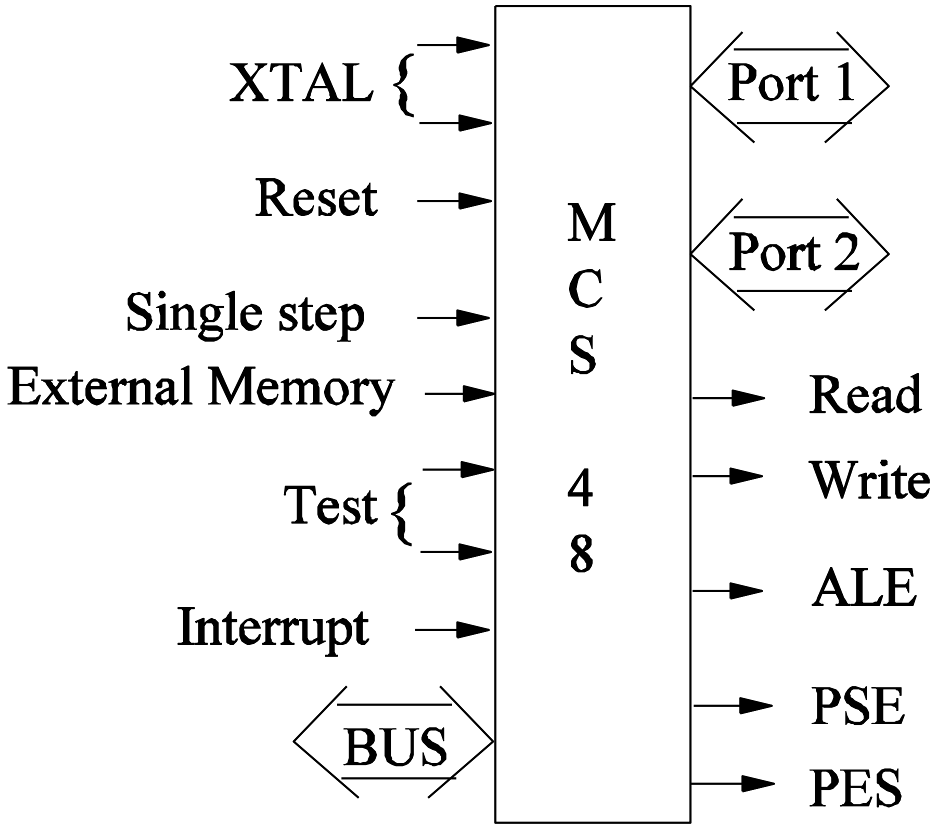
MCS-48

General features of MCS-48 series devices

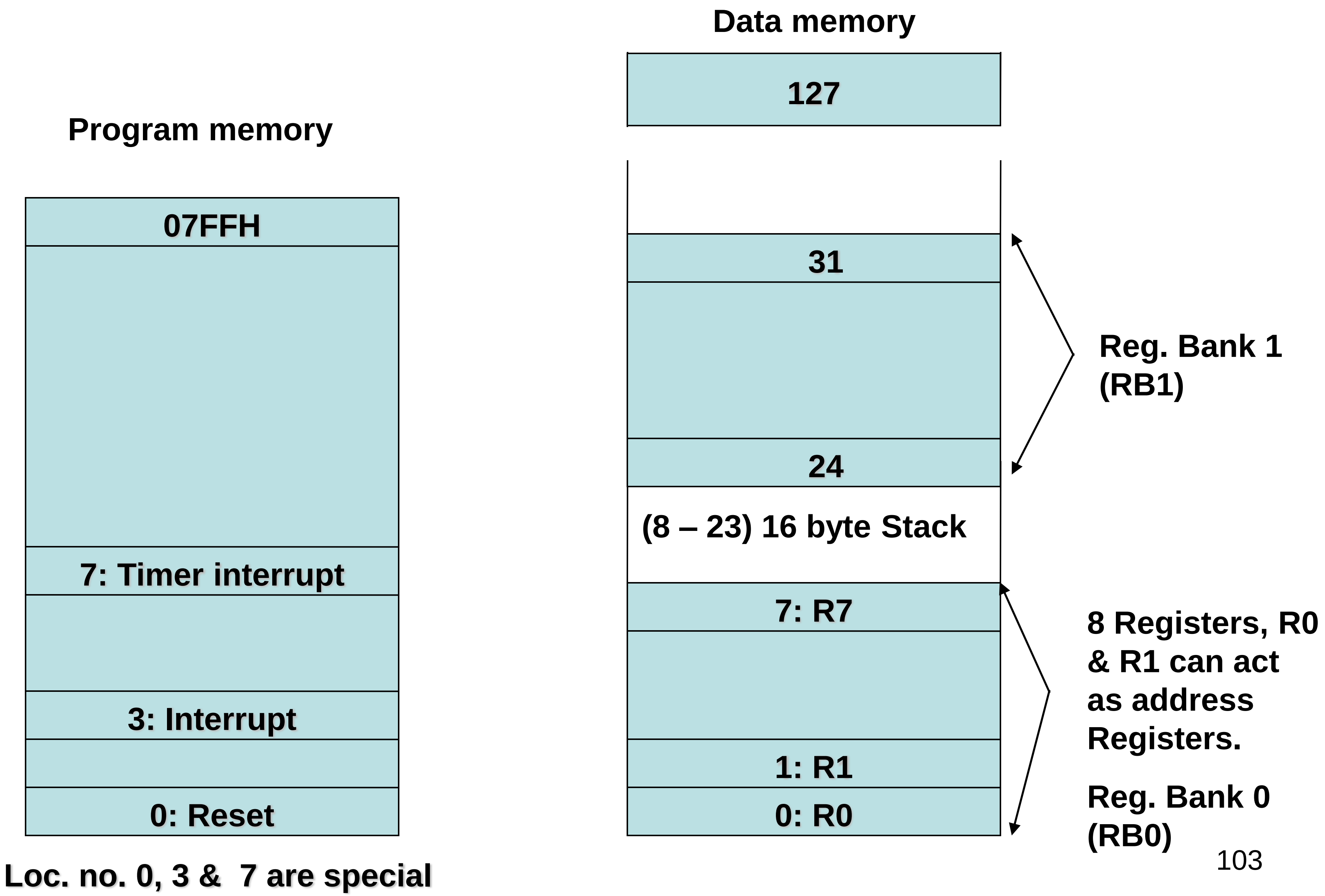
- 8-bit powerful CPU needs a single +5v supply
- 4KB (Max) Prog. memory 256 (Max) bytes Data Mem.
- 1 or 2 cycles instruction
- Over 90% 1 byte instruction
- Compatible with 8085 peripherals
- Reduced power consumption
- Easily expandable memory and I/O
- Two single level interrupt

MCS-48: Block and Functional diagram



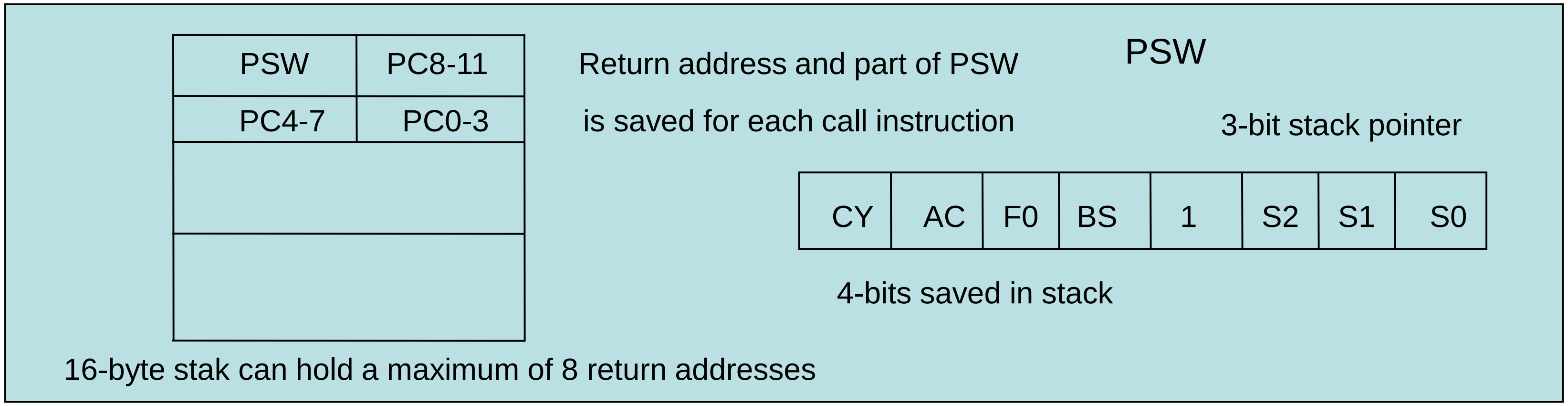


Programming Model



Stack

MCS-48 uC's have limited data memory stack (16 bytes only) allowing the user up to 8 level of nesting. For all practical purpose this is enough. However, implementing recursive routine will be risky due to small stack space. PC bits and some flag (PSW7-4) bits are stored in the stack automatically.


PSW
PC8-11
PC4-7
PC0-3
Return address and part of PSW
is saved for each call instruction
PSW
3-bit stack pointer
CY
AC
F0
BS
1
S2
S1
S0
4-bits saved in stack
16-byte stak can hold a maximum of 8 return addresses

Addressing Modes

MCS-48 is equipped with all the standard addressing modes as well as special modes like paged mode. Here are examples through instructions.

- Direct ; JMP address (12 bits)
- Immediate ; MOV A, #data
- Register ; ADD A, Ri (i=0,1,…,7)
- Register Indirect ; ADD A, @Rx (x=0,1)
- Paged ; MOVP A, @A
- Relative ; J(cond) address (8 bits)
  - several possible conditions are possible; e.g., JC/JNC/JZ/JNZ/JBb etc.

Instructions

- In comparison to 8085, MCS-48 instructions are short but powerful.
- Most of the instructions (over 90%) are 1 byte and executed in a single cycle.
- Bit testing facility is available (e.g., JBb <addr>) allows user to test any bits (b=0 to 7) of the accumulator to branch to an address.
- Logic operations can be done directly at the I/O ports and facilitates control programs (e.g., instruction ORL   P0, #data; does a logical OR operation with the current data at port P0 with the mask specified by #data

Sample programs

pakdig: ; packs bits 0 – 3 of locations 50-51 into location 50.

```
    mov     R0, #50
    mov     R1, #51
    XCHD    A, @R0    ; exch. bits 0-3 of Acc
                      ; with location 50
    SWAP    A         ; exch bits 0-3 & 4-7
                      ; of Acc
    XCHD    A, @R1
    MOV     @R0, A
```

More examples

```
LOC3:   JNI   INIT  ; Jump to routine INIT if
                    ; interrupt input is 0
INIT:   MOV         R7, A
        SEL         RB1
        MOV         R7, #0FA
        :
        SEL         RB0
        MOV         A,R7
        RETR                  ; RET FROM INTR
                              ; RESTORE A & PC
```

More examples

This routine disables interrupt; but jumps to interrupt routine after 8 overflows and stop timer.

```
START:  DIS TCNTI
CLR     A
MOV     T, A
MOV     R7, A
STRT    T
MAIN:   JTF COUNT

JMP     MAIN
COUNT:  INC R7
MOV     A, R7
JB3     INT
JMP     MAIN
INT: STOP    TCNT
JMP     7H
```

More examples

```
mov128:    mov         A, #128
           movp        A, @A
```

This two instructions would move the contents of memory location 129 (in the current page) to the accumulator.

```
page3:     mov         a, #0bFH
           ani         a, #7FH
           movp3       a, @a
```

This will transfer the contents of location no. 38H of page 3 to Accumulator. This two instructions are useful to access data stored permanently from program memory.

Logic operations at I/O ports

- Logic operations can be done directly at the ports. Let us assume that one 8-bit 8255 port; say bit 3 is controlling a process. Now you would like to set bit 3 without disturbing the other bits.

```
; 8085 example

CONWORD  DS  1  ; allocate a byte

LXI    H, CONWORD
MOV    A, M
ORI    A, 00001000B
MOV    M, A
OUT    PORTA

; The above program assumes that the control
; word will be stored in the memory variable
; CONWORD and the output port is PORTA
```

```
; MCS-48 example

ORL   P1, #00001000B

; Note that the port retains the original byte and also
; allows logic operations right at the port (port P1
; in this case). See the code reduction.
```

Powerful CPUs

- 8-bit processors are good for simple applications. Natural extension to these 8-bit CPUs are the 16-bit processors which may be used to design general purpose low cost computers.
- Now 32 or even 64 bit CPUs are available in the market and are extensively used in computing.
- It may be noted that the general principles are same for any processor; however the higher order processors are powerful and have more throughput due to various factors.

Powerful CPUs

- No practical limitation on address space and data width.
- Superscalar performance exploiting pipeline and other mechanism for parallel operation.
- Use of multiple level cache to reduce the CPU memory speed gap.
- Special Hardware/Software features to implement multiprocessing/multitasking OS.

We will briefly discuss INTEL X-86 as a typical example of a 1st Generation 16-bit processor.

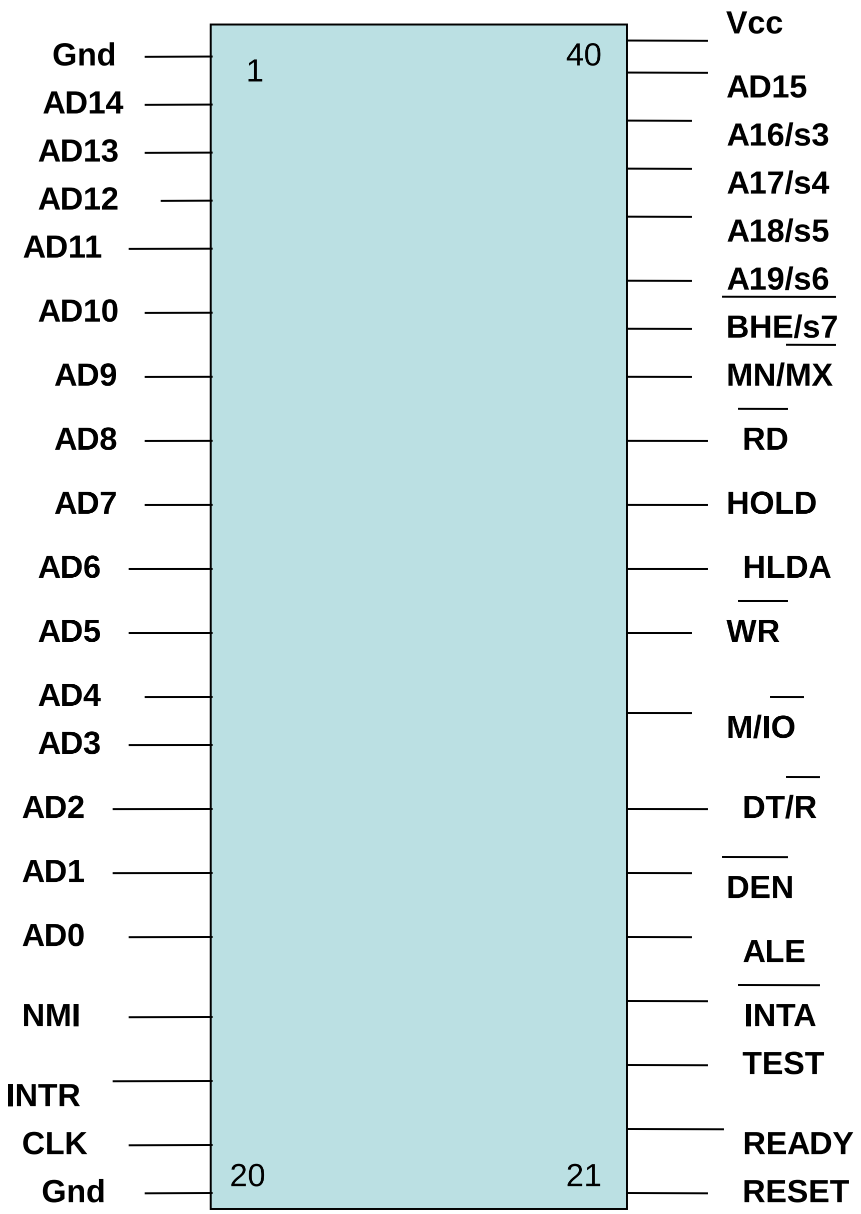
Intel X-86

- It is an extension of 8080/85 architecture into the 16-bit domain.
- Address space is increased to 1 MB in a segmented architecture where code, data and stack segment base values for a running process are available through segment registers.
- Architecture is not orthogonal in order to maintain backward compatibility.
- Current high end product like Pentium is having powerful features (full 32 bit CPU) as well as flat addressing modes and different mode of operations making it suitable for implementation of a variety of Operating Systems.

8086: 1st member of X86

- 16 bit CPU packaged in 40 Pin IC
- 16 bit data BUS (multiplexed with lower 16 bit of Address Bus)
- 20 bit Address Bus
- 2 – level interrupt
- Segmented architecture
- More Instructions and addressing modes
- Instruction pipeline
- Backward compatibility with earlier CPUs

8086 Pin diagram



Maximum mode pin functions are given below

31. HOLD ($\overline{RQ}/\overline{GT0}$)

30. HLDA ($\overline{RQ}/\overline{GT1}$)

29. $\overline{WR}$ ($\overline{LOCK}$)

28. M/$\overline{IO}$ ($\overline{S2}$)

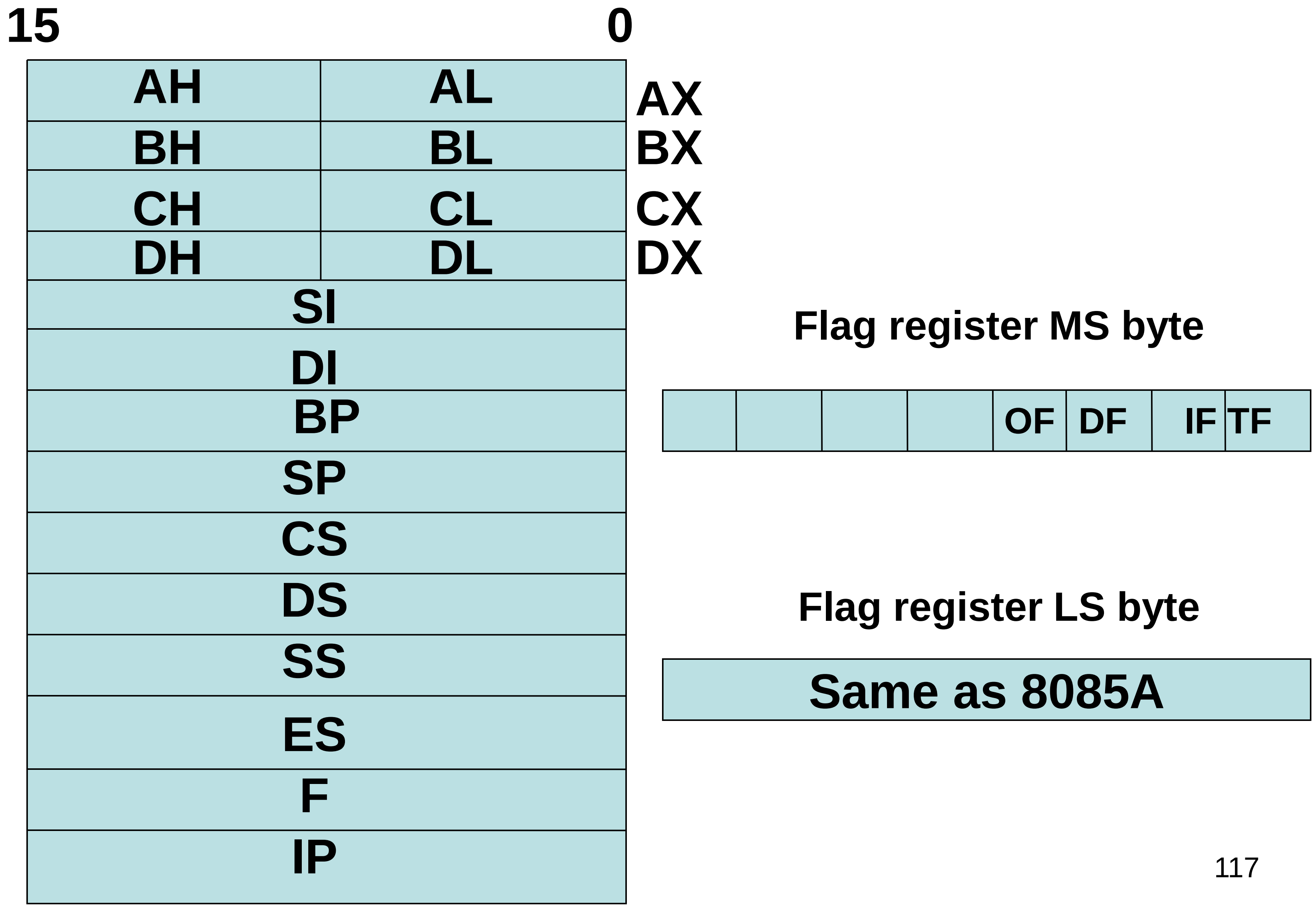
27. DT/$\overline{R}$ ($\overline{S1}$)

26. $\overline{DEN}$ ($\overline{S0}$)

25. ALE (QS0)

24. $\overline{INTA}$ (QS1)

8086 Programming model



8086 Operand addressing modes

Mode	Forms and alternatives	Examples
Direct Offset	<dir_address>	MOV AX, VAR_1
Register	<reg>	MOV AX, BX
Register Indirect (Single register)	[BX], [BP*], [SI], [DI] * Default is SS	MOV AX, [BX]
Register Indirect (Single reg. + offset)	[BX+c], [BP*+c], [SI+c], [DI+c]	MOV AX, [BX+5] MOV AX, VAR_1[BX]
Register Indirect (Two registers)	[BX+SI], [BX+DI] [BP*+SI], [BP*+DI]	MOV AX, [BX][SI] MOV AX, [BP+DI]
Register Indirect (Two registers + offset)	[BX+SI+c], [BX+DI+c] [BP*+SI+c], [BP*+DI+c]	MOV AX, VAR_1[BX][SI] MOV AX, [BP+DI+10]

Assembly Language Examples

- X86 assembly language is complex but powerful.
- A program consists a number of logical segments each pointed through a segment register.
- The segments define code, data and stack segment of the program that are mapped to the memory for execution.
- The assembler directives are very powerful and support complex data structure as well as macro operations.

We present next a hypothetical program showing different aspects of X-86 assembly language.

```
; example program
main_data   segment
   packed_number db 4 dup (0)
main_data   ends

other_data  segment
   unpacked_number  db 8, 7, 6, 5, 4, 3, 2, 1
other_data  segment

prog_data   segment
   assume cs:prog_data, ds:main_data, es:other_data
   prog_start:      mov   ax, main_data
            mov   ds, ax
            mov   ax, other_data
            mov   es, ax
```

```
                mov     bx, offset packed_number
                mov     si, 0
                mov     di, si
                mov     cx, 4
pack:           mov     ax, word ptr es: unpacked_number[si]
                mov     dx, cx
                mov     cl, 4
                shl     ah, cl
                add     al, ah
                mov     [bx][di], al
                add     si, 2
                inc     di
                loop    pack
                hlt             ; a better option is a syscall to OS
prog_code       ends
                end     prog_start
```

More examples

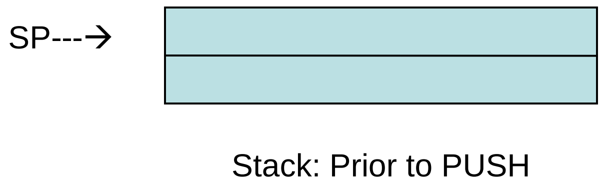
; One of the common use of the stack is parameter passing between functions. The following program fragments show an example.

Assume that an array processing function needs two parameters; start address and the number of items to be processed. The calling program pushes these two parameters in the stack and the called function gets the parameter from the stack. Called function also uses stack for keeping its own local variables.

Parameter passing

```
stack_seg   segment
    dw          20 dup (?)      ; 20 word stack
    stk_top     label   word    ; stack pointer – no stoarge
stack_seg   ends
data_seg    segment
    array1      db      10 dup (?)      ; 10 element array
    array2      db      20 dup (?)      ; 20 element array
data_seg    ends
prog_seg    segment
  assume    cs:prog_seg, ds:data_seg,
                    ss:stack_seg, es:nothing
example     proc    far     ; intersegment call nedded
```

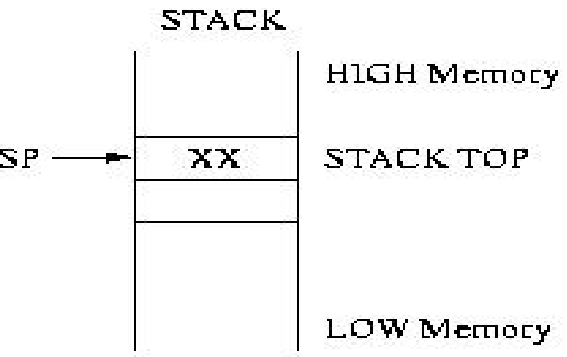
```
; procedure prologue
  push  bp      ; save bp
  mov   bp, sp     ; establish base pointer
  push  bx
  push  cx      ;save caller’s
  pushf         ; reg & flags
  sub     sp, 6     ; allocate local storage
; end of prolgue
```



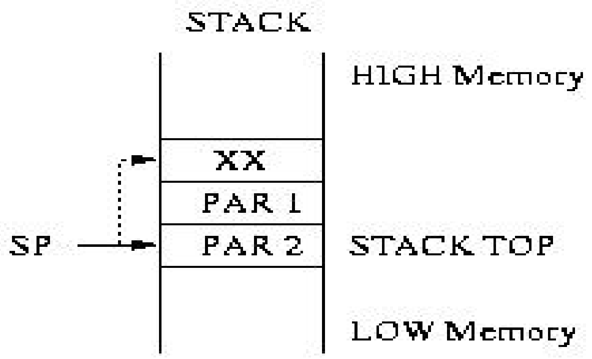
; The above program segment is a standard for any function in
; Assembly language accepting parameters through stack. BP
; is first saved to set up a 2nd stack pointer which is done by
; copying SP to BP. Register(s), flags are stored and depending
; on the no. of local variables storage is allocated. The
; parameters and local variables are accessed through BP
; keeping SP free for normal stack operations (push, pop).

```
; procedure body
  mov     cx, [bp+8] ; get element count
  mov     bx, [bp+6] ; get offset of the 1st element
    ; procedure code for the processing
    ; 1st parameter can be accessed as [bx] and local
    ; storage refs’ are [bp-8], [bp-10] ad [bp-12]
; end of procedure body
;  procedure epilogue
  add    sp, 6  ; de-allocate local storage
  popf
  pop    bx
  pop    cx
  pop    bp
; end of epilogue
; procedure return
ret4   ; discard 2 parameters
example    endp ; end of procedure example
prog_seg   ends
; now how the calls are made
caller_seg  segment
  assume cs:caller_seg, ds:data_seg, ss:stack_seg
  mov  ax, data_seg
  mov  ds, ax
  mov ax, stack_seg
  mov ss,stack_seg
  mov sp, offset stk_top ; point sp to top of stack
```

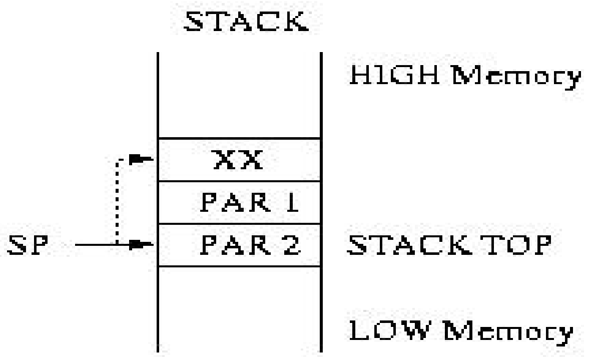
```
; assume array 1 is initialised
mov   ax, size array_1  ;
lea     push parameter 1
```



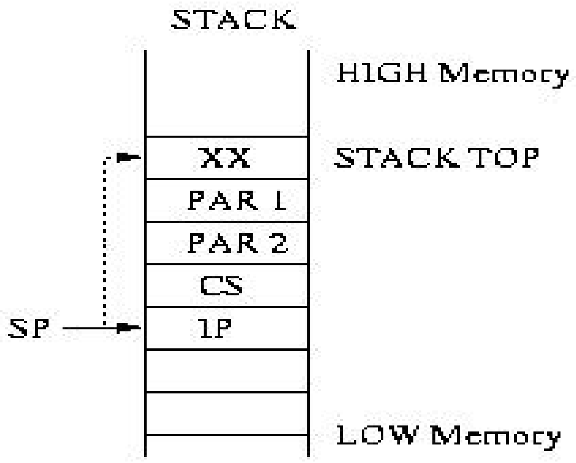
A: Initial Position



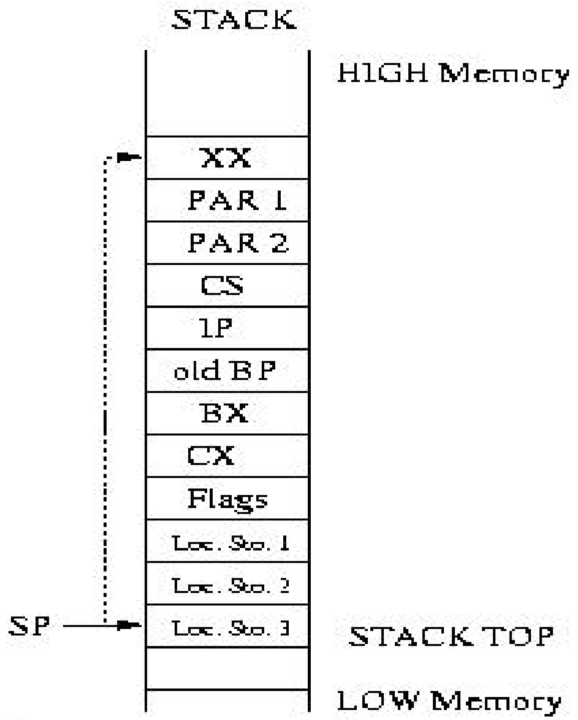
B : After PUSHing two parameters



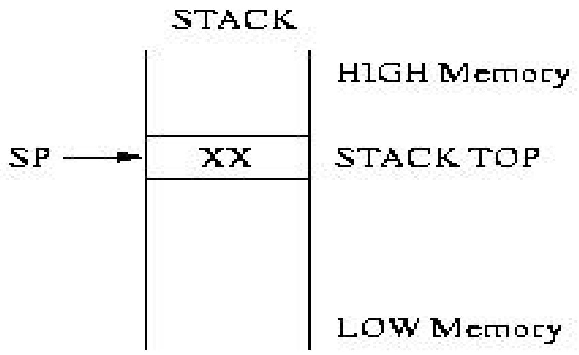
E: After all pop operations



C: After calling the procedure



D: After copy SP to BP and allocating 3 word of local storage



F: Final position after RET 4
Note that SP has gone to its original position. The portion of the stack used for parameter passing is now logically empty and may be reused.

RISC Processor

The processors we have discussed so far are known as CISC (Complex instruction set computer) characterized by:

- Large no. of instructions of varied types.
- Many addressing modes.
- Complex instruction format with varied length of instruction

The motive of the designers were to reduce the semantic gap between higher level language and the machine level (or assembly level) facilities.

RISC Processor contd.

In late 70s' through some studies it was found that complex instructions and addressing modes are not frequently used by the compliers leading to the questions --why not go for a RISC (reduced instruction set computer) processor with

- Reduced number of instructions (with fixed length format for all instructions) and
- Limited addressing modes so as to get a much simpler decoding and control circuitry.
- Load/Store machine for less memory access.
- More compile time effort to produce optimal code

.

RISC Processor contd.

In real life also bricks are the building blocks of

- walls and
- subsequent bigger and complex structures; the reverse is not true.
- Moreover, bigger and complex structures can be built with the help of lower level structures.

These observations gave rise to the quest for having less complex CPU with more computing power.

RISC Processor contd.

The benefits of Reduced no. of instructions and addressing modes are manifold.

- Fixed length simple instruction with less addressing complexity
  - Less complex decoder
    - saves space in the IC.
      - More on-chip register.
      - Bigger on chip cache
      - allows h/w control instead of microprogramming
  - Single cycle instruction execution
    - simplified pipeline

Berkeley RISC I : Features

One of the early projects endorsing the advantages of RISC architecture was Berkeley RISC I. Features of RISC I are:

- 32 bit CPU
- 32 bit address
- 8, 16 or 32 bit data
- 32 bit instruction; three instruction formats only.
- 31 Instructions
- 3 addressing modes (Register, Immediate and PC relative)
- 138 CPU register (32 is active at a time – called a window)
- load store machine
- Overlapped register window for faster subroutine call and return

RISC Processor contd.

One out of 8 instruction in HLL is a call/ret and that has a very high execution overhead due to stack (memory) access. Moreover, the standard technique of stack based parameter passing also calls for more memory access. This overhead is highest considering the number of instructions in the low level code and corresponding memory references compared to all other instructions.

This bottleneck is alleviated nicely in Berkeley RISC I by a overlapping register window technique where the caller and called share common registers effectively avoiding memory access.

Berkeley RISC I : Instruction Format

Instruction Formats:

op-code	SCC	Rd	Rs	IMF	Not Used	S2
7	1	5	5	1	8	5

(a) Register Mode (S2 specifies a Register; if IMF=0)

op-code	SCC	Rd	Rs	IMF	S2 (13 bit)

(b) Register- Immediate Mode (S2 : an operand; if IMF=1)

op-code	SCC	COND	Y (19 bits)

(c) PC relative mode (Y specifies the branch offset) (cond is 5 bit)

scc is set condition code: controls if the flags will be affected or not

Berkeley RISC I : Instruction Set

Op-code	operand	Register Transfer	Description
DATA MANIPULATION (ARITHMETIC) INSTRUCTIONS			
ADD	Rs, S2, Rd	Rd ← Rs + S2	Integer Add
ADDC	Rs, S2, Rd	Rd ← Rs + S2 + C	Add with carry
SUB	Rs, S2, Rd	Rd ← Rs – S2	Integer Subtract
SUBC	Rs, S2, Rd	Rd ← Rs – S2 - C	Subtract with borrow
SUBR	Rs, S2, Rd	Rd ← S2 – Rd	Subtract reverse
SUBRC	Rs, S2, Rd	Rd ← S2 – Rd - C	Subtract reverse with borrow

Berkeley RISC I : Instruction Set

Op-code	operand	Register Transfer	Description
DATA MANIPULATION (LOGIC) INSTRUCTIONS : CONTD			
AND	Rs, S2, Rd	Rd ← Rs ^ S2	AND
OR	Rs, S2, Rd	Rd ← Rs v S2	OR
XOR	Rs, S2, Rd	Rd ← Rs S2	XOR
SLL	Rs, S2, Rd	Rd ← Rs	LEFT SHIFT (LOGICAL) SHIFTED BY S2
SRL	Rs, S2, Rd	Rd ← Rs	RIGHT SHIFT (LOGICAL) SHIFTED BY S2
SRA	Rs, S2, Rd	Rd ← Rs	SHIFT RIGHT (ARITHMETIC) SHIFTED BY S2 and copying sign bit in the vacant positions

Berkeley RISC I : Instruction Set

Op-code	operand	Register Transfer	Description
DATA TRANSFER (LOAD/STORE) INSTRUCTIONS : CONTD			
LDL	(Rx) S2, Rd	Rd ← M[Rs + S2]	LOAD LONG
LDSU	(Rx)S2, Rd	Rd ← M [Rs+S2]	LOAD SHORT UNSIGNED
LDSS	(Rx)S2, Rd	Rd ← M [Rs+S2]	LOAD SHORT SIGNED
LDBU	(Rx)S2, Rd	Rd ← M [Rs+S2]	LOAD BYTE UNSIGNED
LDBS	(Rx)S2, Rd	Rd ← M [Rs+S2]	LOAD BYTE SIGNED
STL	(Rx)S2, Rm	M[Rx + S2] ← Rm	STORE LONG

Berkeley RISC I : Instruction Set

Op-code	operand	Register Transfer	Description
DATA TRANSFER (LOAD/STORE) INSTRUCTIONS : CONTD			
STS	Rm, (Rx)S2	M[Rx + S2] ← Rm	STORE SHORT
STB	Rm, (Rx)S2	M[Rx + S2] ← Rm	STORE BYTE

Berkeley RISC I : Instruction Set

Op-code	operand	Register Transfer	Description
PROGRAM CONTROL INSTRUCTIONS :			
LDHI	Y, Rd	Rd(31—13)← Y, Rd(12-0) ← 0	LOAD HIGH IMMEDIATE
GETPSW	Rd	Rd ← PSW	LOAD STATUS WORD
PUTPSW	Rm	PSW ← Rm	SET STATUS WORD
JMP	CON, S2(RX)	PC ← Rx + S2	CONDITIONAL JUMP
JMPR	CON, Y	PC ← PC + Y	CONDITIONAL REL. JUMP
CALL	S2(Rx), Rd	CWP--; Rd ← PC; Next PC ← Rx + S2	CALL SUBROUTINE AND CHANGE WINDOW

Berkeley RISC I : Instruction Set

Op-code	operand	Register Transfer	Description
PROGRAM CONTROL INSTRUCTIONS :			

CALLR	Rd, Y	CWP--;
		Rd $\leftarrow$ PC;
		Next PC $\leftarrow$ PC + Y;
			CALL SUBROUTINE AND CHANGE WINDOW (REL)

RET	Rx(S2)	PC $\leftarrow$ Rx + S2; Next CWP++	RETURN & AND CHANGE WINDOW
CALLINT	Rd	CW--; Rd $\leftarrow$ last PC;	INTERRUPT CALL; DISABLE INTERRUPT
RETINT	Rx(S2)	PC $\leftarrow$ Rx + S2; Next CWP++	RETURN & AND CHANGE WINDOW (ENABLE INTERRUPT)
GTLPC	Rd	Rd $\leftarrow$ PC	GET LAST PC

Berkeley RISC I : Instruction Set

Most notable absence in the instruction set is the MOV instructions (Register to Register). However, an innovative idea of storing a 0 (zero) always in R0 solves the problem.

```
i)    ADD      R0, R2, R3     ; (MOVE)  R3 ←R2
ii)   ADD      R0, #100, R3 ; (MOVE IMM.) R3 ← 100
iii)  ADD      R0, #1, R3;          ; (INC) R3++
```

Berkeley RISC I : Instruction Set

The load/store instruction moves data (byte, word or long word) to and from register to memory location using different addressing modes. Here are some examples with loading long words.

```
i)LDL (R20)#0, R21      ; R21 ← [R20] – REGISTER INDIRECT
ii)LDL(R20)#100, R21 ; R21 ← [R20+100] – REGISTER
                               ; INDIRECT WITH DISPLACEMENT
iii)LDL    (R0)#100, R21   ; R21 ←[100] -- DIRECT
```

Berkeley RISC I : Register Window Scheme

- No of calls (& corresponding return) instruction are frequent; one in 7/8 instructions in HLL; and the highest in the no. of instructions (#33) and the memory references (#45).
- Reducing this overhead improves the overall performance.

HLL instruction	HLL #occurrence	Weighted #instruction	Weighted #memory-reference
call/ret	12 +/- 5	33 +/- 14	45+/- 19
loops	3 +/- 1	32 +/- 6	26 +/- 5
assign	38 +/- 15	13 +/- 5	15+/- 6
If	43 + /- 7	21 +/- 8	13+/- 5
case	1 +/- 1	1 +/- 1	1 +/- 1
go-to	3 +/- 1	0	0

Berkeley RISC I : Register Window Scheme

- Pushing/Popping return address into/from stack during call/ret and STACK based parameter passing can be avoided by doing the same through CPU registers provided a lot of registers can be made available right inside the CPU.
- Simpler instruction decoder and control lead to space saving and in RISC I architecture a lot of CPU registers could be provided.
- Nesting depth of the call/ret rarely exceeds 8 and a 8-window based circular overlapped window is made available in RISC I to reduce memory references and faster parameter passing.

Berkeley RISC I : Register Window Scheme

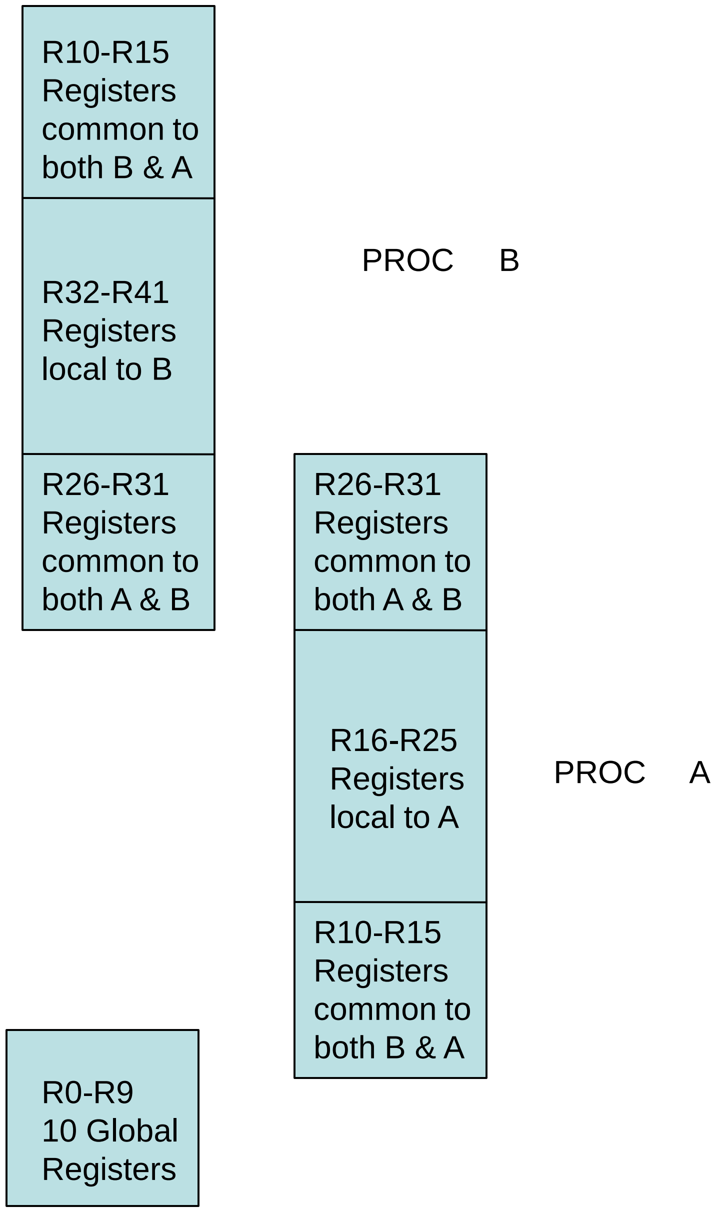
Salient points:

- RISC I has provided 138 registers in the CPU and at any point of time 32 of them are active.
  - It has been decided to use for the current function in execution.
    - » 10 Global registers are in use (always)
    - » 10 registers local to the function
    - » 6 registers (common to the 'calle' and the 'called' function at any level)

 This way 8 separate window may be maintained (with a 3-bit Current window pointer CPW) to handle a nesting depth of 8 where parameter passing overhead is zero as all will be available within the CPU (see the figure in the next slide)

Berkeley RISC I : Register Window Scheme

- The scheme is shown for two windows only
- At a time there are 32 active registers
- Total Requirement is 10 + 10 + 2 x 6 = 42 Registers
- For 8 windows the requirement is 10 + 10 x 8 + 8 x 6 = 138 registers


R10-R15 Registers common to both B & A
R32-R41 Registers local to B
R26-R31 Registers common to both A & B
PROC B
R26-R31 Registers common to both A & B
R16-R25 Registers local to A
PROC A
R10-R15 Registers common to both B & A
R0-R9 10 Global Registers

RISC

- RISC architecture – in order to reduce the code size has increased the number of instructions; going against the initial objective.
- CISC has also adopted a lot of the nice features of RISC.
- There is a deviation from so called pure RISC characteristics in commercial RISC CPUs to achieve higher throughput and to compete with established CISC machines.
- So, the distinction between CISC & RISC is blurred

The ARM (Advanced Risc Machine) Processor

THE ARM PROCESSOR IS SPECIALLY DESIGNED TO BE SMALL TO REDUCE POWER REQUIREMENT (TO BE SOURCED FROM THE BATTERY) AND ARE IDEAL FOR EMBEDDED AND MOBILE APPLICATIONS.

- High code density
- Focus is not on raw processing speed but higher system performance
- Embed specialized peripherals right onto the chip
- Allow the use of low cost memory to reduce system cost

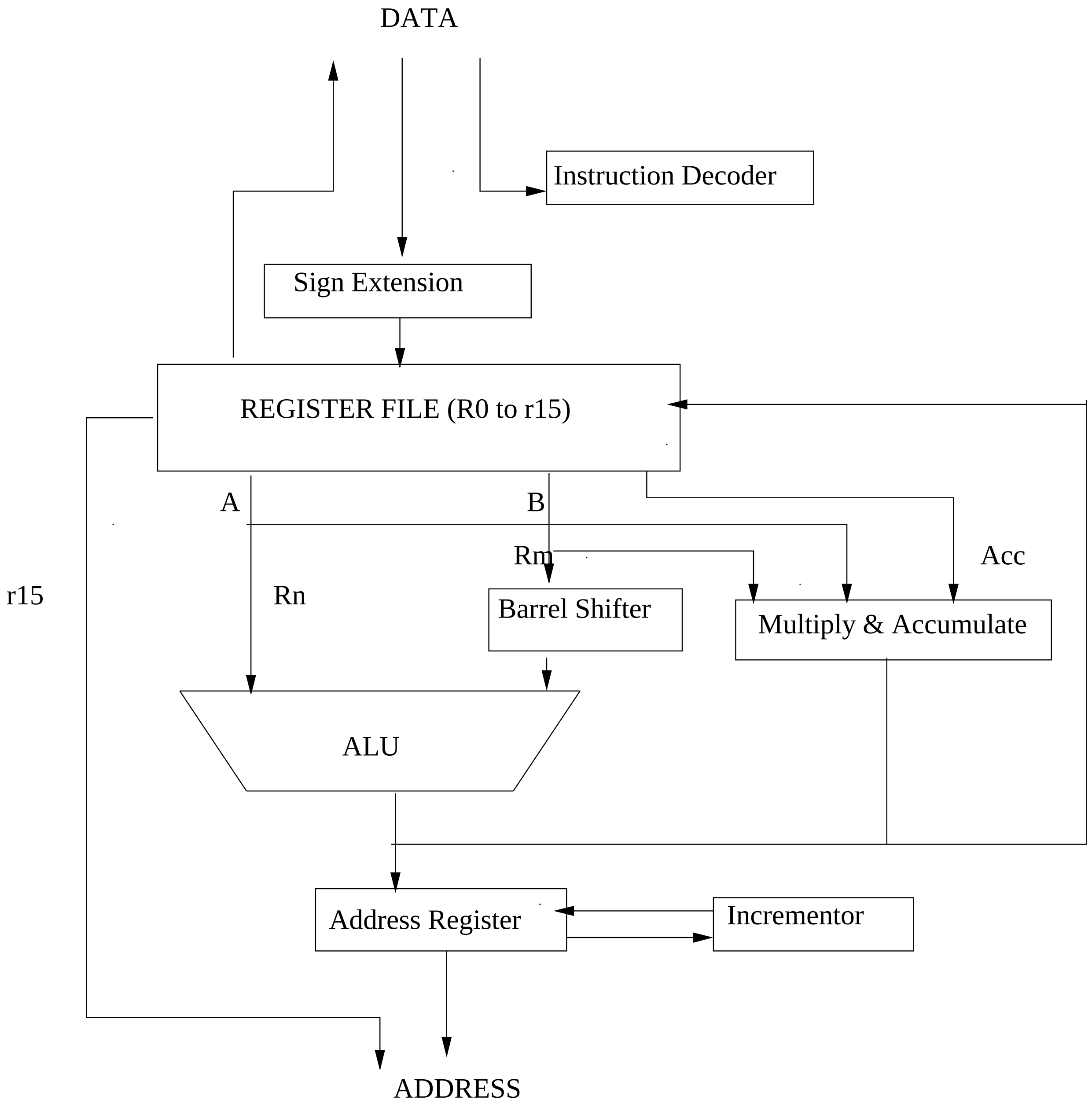
The ARM Instruction Set

The ARM instruction differs the pure RISC instructions in certain ways as depicted below.

- Variable cycle instruction: For efficiency every instruction is not single cycle; e.g., LOAD/STORE instruction can transfer multiple words in consecutive memory location.
- In-line barrel shifter: A component which preprocesses one of the input registers before use. This expands the capabilities of many instructions to improve core processing and code density.
- Thumb (16 bit) Instructions: This second set of instruction can be employed , whenever possible, to improve code density by about 30% over 32-bit fixed length instruction.
- Conditional execution: Instruction executes on condition; this improves code density by avoiding branching.
- Enhanced Instruction: DSP instructions were added to support 16 x 16 bit multiplier operations.

The ARM Dataflow

DATAFLOW MODEL (ARM :VON NEUMAN)



The ARM Dataflow

- The instruction and data comes from the memory. Instructions are put to ID and the data is passed through the sign-extender. The processed data also sent back through the same bus.
- ARM being a RISC machine follows the LOAD/STORE principle and the operations are carried on the ALU (Typical 3 address instructions are used; <op> Rd, Rn, Rm; Rd ← Rn <op> Rm). The result goes back to one of the Registers (Rd). One of the operands (Rm) is passed through a barrel shifter; optionally it shifts the operand before submitting the sane to the ALU.
- The ALU or MAC takes the operand and the result from the data processing instruction is written back to Rd. .
- The address bus is driven by R15 (the PC) for instruction fetch. LOAD/STORE instructions use ALU to generate an address to be held by the address register and placed on the address bus.

The ARM Registers and Modes

- The ARM uses 16 registers r0 to r15. r0 to r13 are fully orthogonal and available for all instructions. They hold both the data and address. r13 is the sp, r14 is the link register holding the return address and r15 is the PC.
- Two more registers, namely cpsr (current program status register) and spsr (saved program status register) are also used by ARM.
- The psr is 32 bit holding the NZVC flags (b31,b30,b29 & b28). Bit 0 to 4 specifies MODE and b7 and b6 are the interrupt masks for normal interrupt and fast interrupt operations. Bit b5 indicates the thumb state.
- ARM processor acts in one of the 7 modes; i) user; ii) fast interrupt request ; iii) interrupt request; iv) supervisor; v) undefined; vi) abort and vii) system mode. Other than the user mode rest are privileged and can change the psr.

The ARM Registers and Modes

A processor mode determines which register is active and the access rights to cpsr. A privilege mode allows a full r/w access to cpsr. While a non privilege mode allows read access of the cpsr and r/w to the flags only.

- User mode (Non privilege) : used for programs and applications
- Supervisor: goes to this mode after power-up. OS kernel runs in this mode
- System: Similar to user mode but has full r/w to cpsr
- Interrupt: moves to this mode on interrupt request
- Fast Interrupt: moves to this mode on first interrupt request
- Abort: enters this mode on failed attempt to access memory
- Undefined: enters this mode when unknown instruction is encountered.

The Banked Registers

- There are 37 registers of which 20 are hidden from a program at different times. These hidden registers are banked registers active with the change of mode. Banked registers maps one one and onto a CPU register. So, programming model does not change. For an example, when theCPU goes from user to interrupt mode due to an interrupt r13 and r14 are banked (replaced) with r13_irq and r14_irq. The cpsr register is copied to spsr_irq as well. It means the reference to r13 in the program is actually dealing with r13_irq during the irq (interrupt request) mode; and on return to user mode after serving the interrupt r13_irq and r14_irq are silently replaced by r13 and r14. On return, the original cpsr is restored from spsr_irq as well.
- The user and system modes are similar in the sense that banking is not done.
- Three registers are banked in irq, abt, svc, und modes. While in frq (fast interrupt request) mode r8 to r14 are all replaced by the banked registers, namely r8_frq to r14_frq, respectively.

ARM INSTRUCTIONS

- There are 58 core instructions in ARM set and 30 in Thumb set.
- Basically a three address instruction type (e.g., SUB Rd, Rn, Rm) which can be modified using the conditions.
- A typical arithmetic instruction takes the following form:
- <instruction>{<cond>}{s} Rd, Rn, N; Rd <- Rn - N
  - {<cond>} stands for conditional execution
  - {s} forces change in condition codes
  - N is a register or immediate operand.
- As an example; sub r2, r3, r5; r2 ← r3 – r5 unconditional where as
- *subgt r2, r3, r5*; does the same if the <gt> condition is satisfied and
- subgts *r2, r3, r5* ; does the same with the change in conditional bits in cpsr;
- Guess what *subs r1, r1, #1* does if r1=1 prior to the execution. And
- sub r2, r3, r5, LSL #2; does r2 ← r3 – (4 times r5)

Use of conditional execution : Reducing code size

For two integers the algorithm to find out the GCD is presented below in HLL & ARM assembly language

----

```
While ( a != b)
   if (a >b) a -= b;
  else b -= a;
```

```
gcd   cmp   r1, r2                  ; ARM 1st version
beq         done
blt         lessthan
sub         r1, r1, r2
b           gcd
Lessthan
sub         r2, r2, r1
b           gcd
done
```

```
gcd         cmp   r1, r2          ; ARM 2nd version with conditional execution
      subgt r1, r1, r2    ; subtract if r1 > r2 (r1 ← r1 – r2)
      sublt r2, r2, r1    ; subtract if r2 > r1 (r2 ← r2 –r1)
      bne   gcd
done
```