Instruction Set

Ram Sarkar

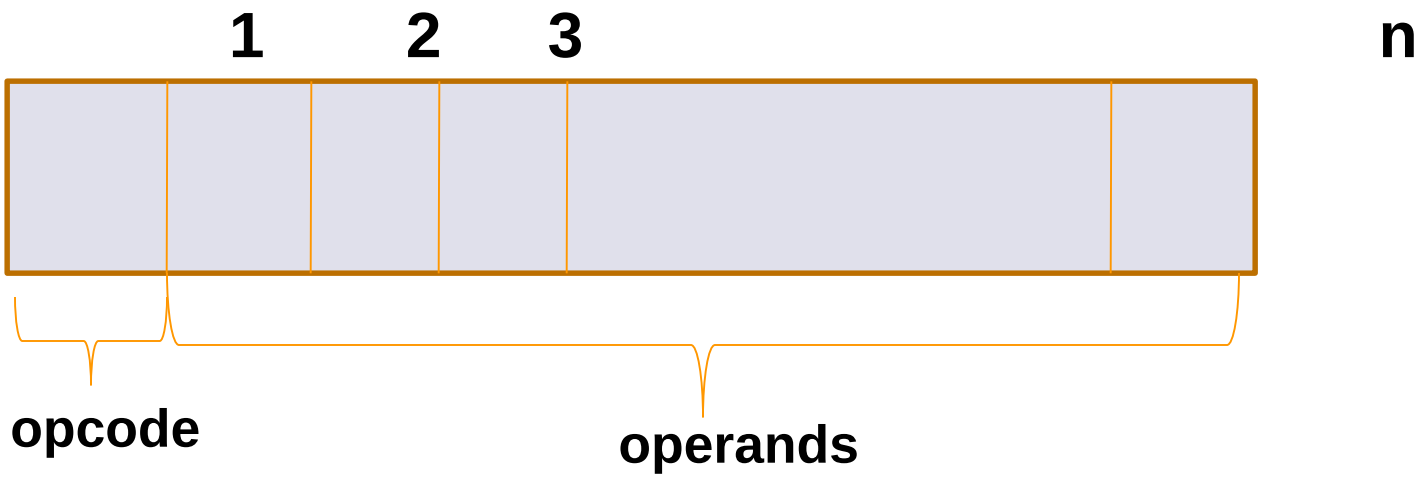
Instruction sets

Purpose: To **specify** an operation to be carried out & the set of operands/data to be used.

-The operation is specified by a field called the **opcode**.

- The operand fields contain the addresses of storage locations in main memory or in the processor.

Most instructions specify a register transfer operation of the form X_1 <- f(X_1,$X_{2,}$….,X_n),which involves n operands.



-To reduce the instruction size, specify m<n operands explicitly in the instruction, the remaining operands are implicit.

- Explicit address : Main memory
- Implicit address: Register

    -It is called *m-address machine*

- Implicit input operands must be placed in locations known to CPU.

Machine Instructions

- A computer must have instructions capable of performing four types of operations:
  - Data transfers between the memory and the processor registers
  - Arithmetic and logic operations on data
  - Program sequencing and control
  - Input/Output (I/O) transfers

Data Transfers: Possible Locations

- Memory locations
- Processor registers
- Registers in the I/O subsystem
- In the instruction itself (immediate data)

Most of the time we identify a location by a symbolic name standing for its hardware binary address:

- Memory Locations: LOC, A, VAR2
- Processor register names: R0, R5, R10
- I/O register names: DATAIN, OUTSTATUS

Data Transfers: Register Transfer Notation

[LOC] means the contents of the location LOC

R1 ← [LOC] means that the contents of memory location LOC are transferred into processor register R1

R3 ← [R1]+[R2] means that the sum of the contents of registers R1 and R2 is transferred into processor register R3

Data Transfers: Assembly Language Notation

Move LOC, R1

means that the contents of memory location LOC are transferred into processor register R1

This is equivalent to R1<- [LOC] in Register Transfer Notation

In the IBM PC the instruction MOV is equivalent to Move

Data Transfers: Assembly Language Notation

Move Source, Destination

means that the contents of memory location Source are transferred into memory location Destination.

As a result, the previous contents of memory location Destination will be replaced, but the contents of memory location Source will not be changed

The instruction Move copies the contents of one memory location to another one

Different Formats

Book:

- **Format**

  OPcode src, dst

IBM PC Assembly

- **Format**

  OPcode dst, src

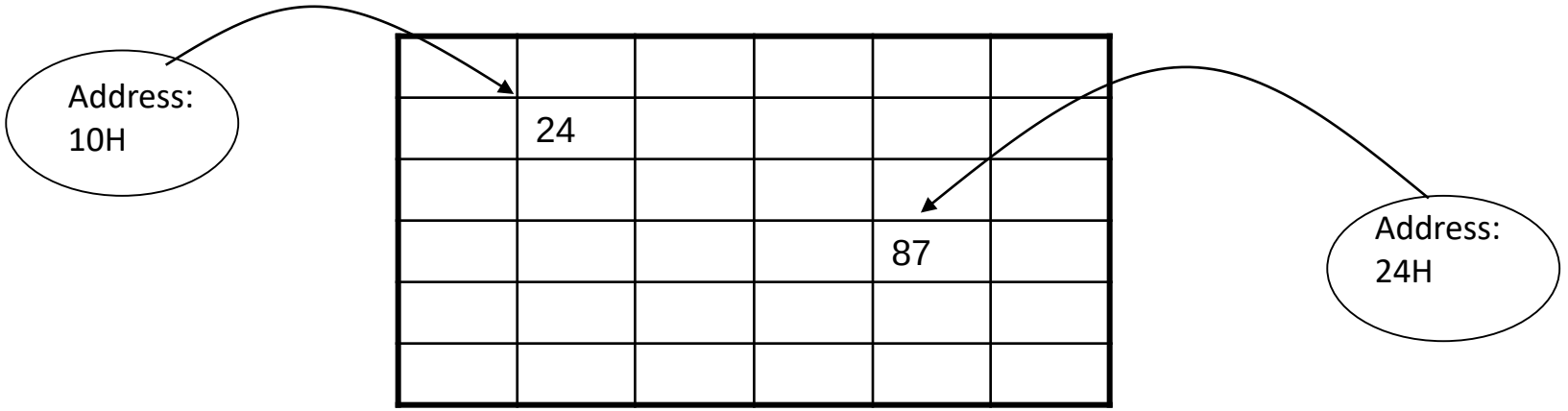
MOV Instruction

In IBM PC you are talking about the contents of a variable (Move), not its memory address (←), hence, if:

X is a variable with memory address 10H and its contents are 24H

1)The instruction: MOV AX,X moves 24H into AX

2) The instruction MOV AX,[X] moves 87H into AX



Basic Instruction Types

C=A+B

How is this high-level language command implemented in the computer?

To carry out the action

C ← [A]+[B]

the contents of memory locations A and B are fetched from memory and transferred into the processor, where their sum is computed and then transferred to memory location C

Three-Address Instruction

General form: **Operation Source1, Source2, Destination**

Add A, B, C

Disadvantage: This form has 3 operands. If memory addresses were to be used to specify operands, the memory space would be very limited.

Example: if *k*=10 bits (enough for a memory of 1 KB), then 30 bits will be needed for the 3 operands.

Two-Address Instruction

General form: **Operation Source, Destination**

Add A, B performs the operation B←[A]+[B].

The result is sent to memory and stored in location B, replacing the original contents of this location.

C ← [A]+[B] can be implemented as

Move B, C

Add A, C

Even a 2-address instruction is **too large** for a processor with a 32-bit address space !

One-Address Instruction

General form: **Operation Source**

Examples:

Add A : Add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator

Move A : Copy the contents of memory location A to the accumulator register

Store A means: Copy the contents of the accumulator register to memory location A

One-Address Instruction

Thus, **C ← [A]+[B]** can be implemented as

(Assembly Language)

Move A
Add B
Store C

Notice that now the 32-bits will be use only to access 1 memory location and to denote the operation

which means (Register Notation):

Accumulator ← [A]
Accumulator ← [Accumulator] + [B]
C ← [Accumulator]

A two-operand instruction, how is it possible?
One Register, One memory location

One-Address Instruction

In the IBM PC, **C ← [A]+[B]** can be implemented as

Assembly Language		Register Notation
MOV	**AX,A**	**; [AX] ← [A]**
ADD	**AX,B**	**; [AX] ← [AX] + [B]**
MOV	**C,AX**	**; [C] ←[AX]**

Notice that in the IBM PC Assembly Language the operation "addition" is represented by the instruction "ADD".

The symbol ";" is used to indicate the start of comments. Comments are useful to remind us what the program is doing, but they do NOT effect the behavior of it

Processor Registers

- Because the number of registers is relatively small, only a few bits are needed to specify, which register takes part in an operation
- For example, for 32 registers only 5 bits are needed to address them

Using Processor Registers for Arithmetic Operations

C=A+B that is, **C ← [A]+[B]** can be implemented as

Move A, R*i*
Move B, R*j*
Add R*i*, R*j*
Move R*j*, C

4 instructions

If we wanted to do it in this manner, for the IBM PC we would have:

MOV DX,A
MOV AX,B
ADD AX,DX
MOV C,AX

If One of the Arithmetic Operands is in Memory

C=A+B that is, **C ← [A]+[B]** can be implemented as

Move A, R*i*
Add B, R*i*
Move R*i*, C

3 instructions

For the IBM PC we would have:

MOV AX,A
ADD AX,B
MOV C,AX

Addressing Modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*] ; Increment R*i*
Autodecrement	–(R*i*)	Decrement R*i* ; EA = [R*i*]

EA = Effective address
Value = a signed number

Addressing Modes

- Operand associates with data X.
- To execute, CPU needs the current value of X.
- It can be specified in several ways: *addressing modes*
- If the X is constant, then its value can be placed in the operand field;

- X is called *immediate operand* and the mode of operand specification is called immediate addressing.

- Corresponding operand field contains the address X of the storage location containing the required value .
- Operand value can be varied without modifying any instruction address: **Direct Addressing**

- It is frequently useful to change the location of X without changing the address fields of any instructions that refer to X. - **Indirect Addressing**

- *Direct Addressing* requires **one** fetch operation to obtain an operand value, while *Indirect Addressing* requires **two**.

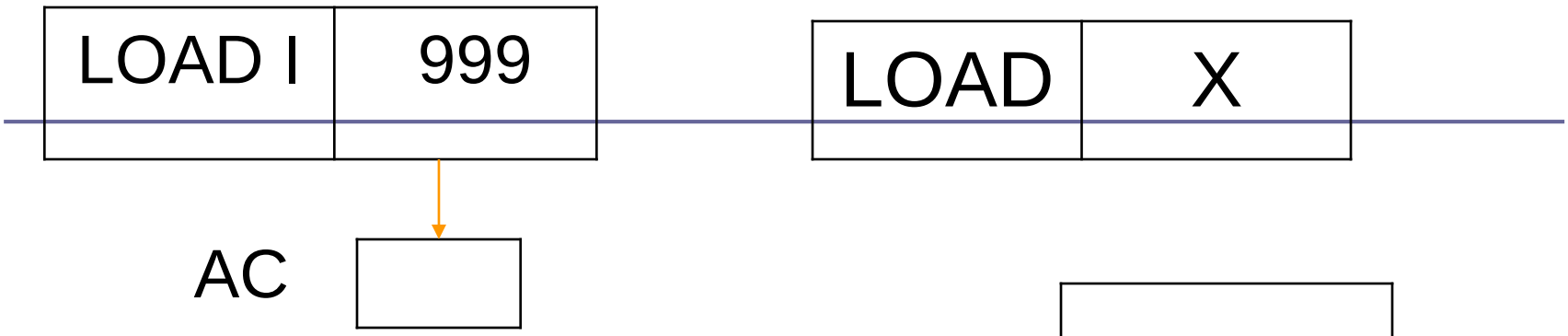
MOV A , B (A<-B)

Direct Addressing (register-to-register transfer)

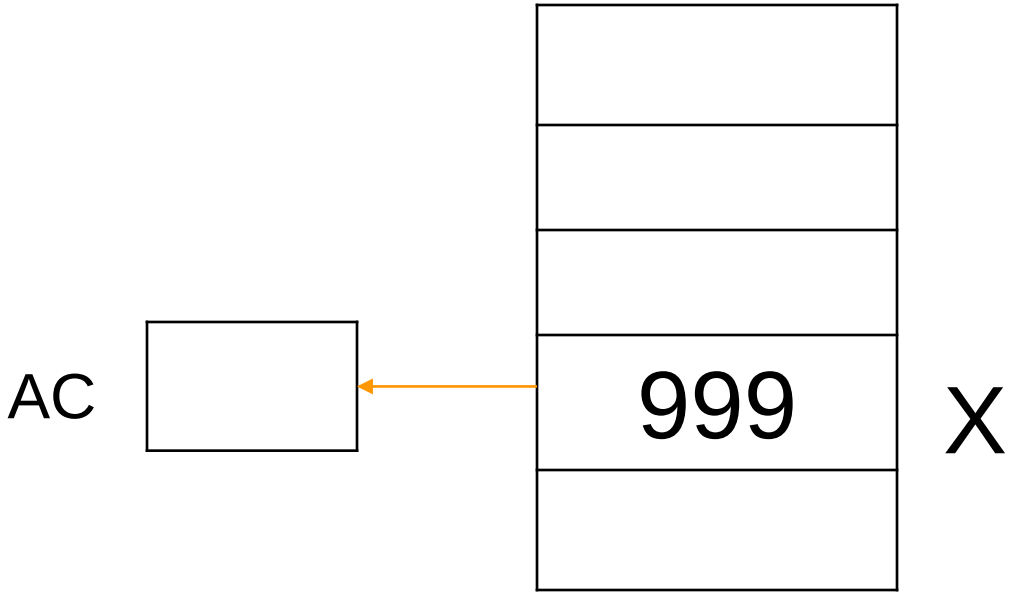
MVI A , 99 (A<- 99)

Immediate operand

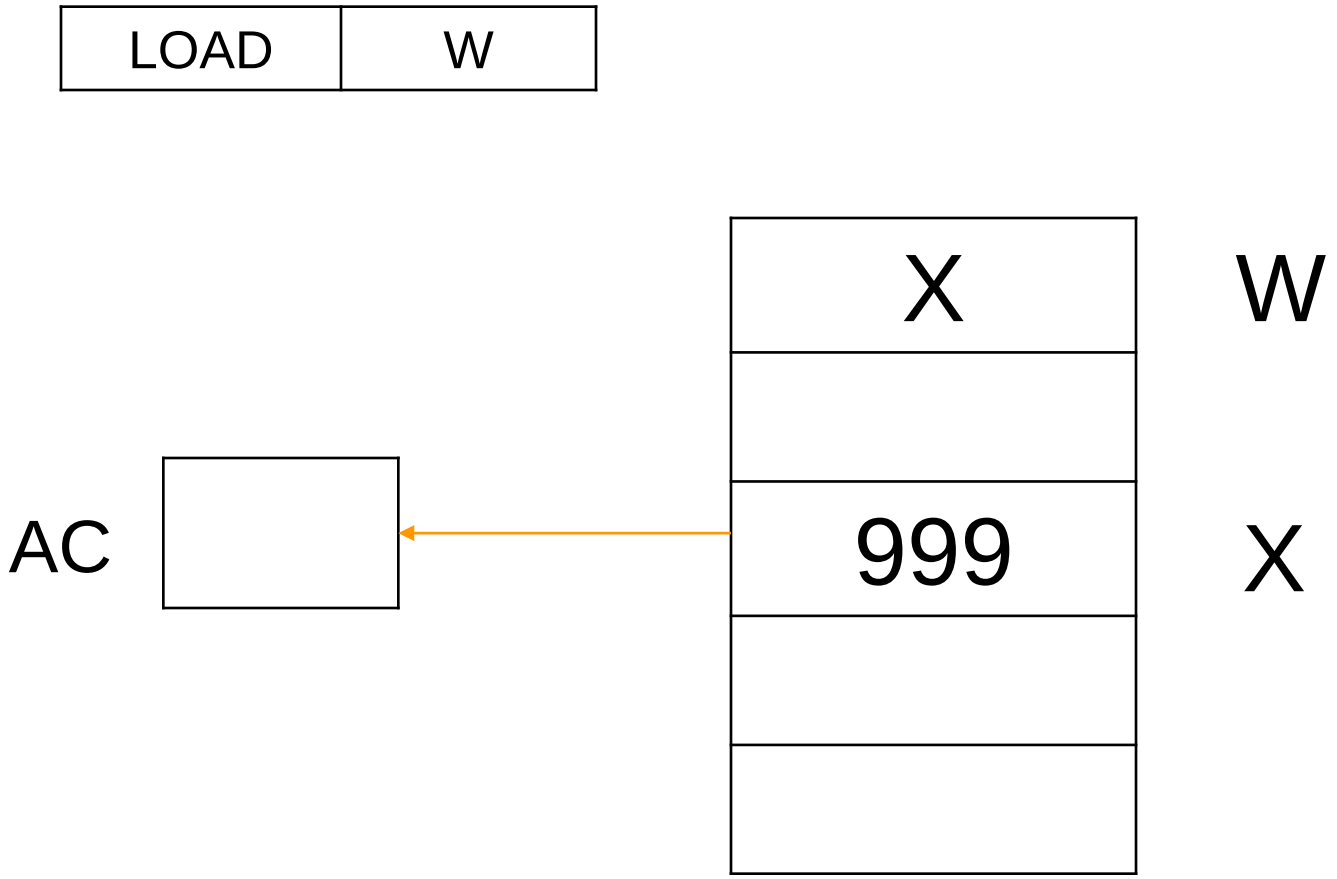
- uses both Direct & Immediate addressing modes.



(a) Immediate



(b) Direct Addressing



(c)Indirect Addressing

Direct Addressing Mode	Immediate Addressing Mode
Address fields contains the effective address of operand	There is no address field as the operand is a part of the instruction.
It requires one reference to memory.	It does not require any reference to memory.
It is slower compared to immediate mode.	It is a faster process.
It has more range than in immediate mode.	It has a limited range.
Example: Add (1001)	Example: ADD 5

Orthogonality:- The ability to use all relevant addressing modes in a **uniform and consistent** way with all opcodes of an instruction set.

- Simplifies programming both

- by reducing the number of opcodes needed &
- by simplifying the rules for operand address specification.

Absolute Addressing:- Simplest mode of (direct) address formation.

- Requires complex operand address to appear in the instruction operand field.

- Address is used without further modification to access the desired data item.

Relative Addressing:-

- Partial addressing information is included in the instruction.
- Complete address must be constructed by CPU.
- Operand field has a *relative address* or displacement D.

- Instruction can identifies storage locations R_1,R_2
- Effective address A is f(D,R_1,R_2,…,R_k)

In general, A = R + D , R may be PC.

Advantages:

1) Since all the address information need not be included, instruction length is reduced.

2) By changing the contents of R, CPU can change the absolute address referred to by a block of instruction B.

- Relocation of entire region becomes easier.
- R acts as a **base register**. (content is called base address)

3. Facilitates the process of indexed data

- R is called the **INDEX REGISTER**.
- Indexed terms X(0), X(1),…,X(n) are stored in the consecutive addresses of Main Memory.
- D contains the address of X(0)
- R contains the Index ‘i’.

- X(i)=D+R; by changing the contents of R, a single instruction can be made to refer to any item X(i) in the list.

Issues:

- Extra logic circuits and extra processing time required for address computation.

- Indexed items are frequently accessed sequentially, a process called ‘AUTOINDEXING’ is required which automatically increments or decrements an address.

(A)- and (A)+: First one indicates that contents of A should be *decremented* automatically before the instruction is executed, whereas the latter one *increments* automatically after the current instruction has been executed.

- *Autoindexing* facilitates another *addressing mode*:
    - **Stack addressing**: Part of Main Memory

-Two options:

Push - stores a new item at the top of the stack

Pop - removes the item stored at the top

-Controlled by the register called **STACK POINTER**.

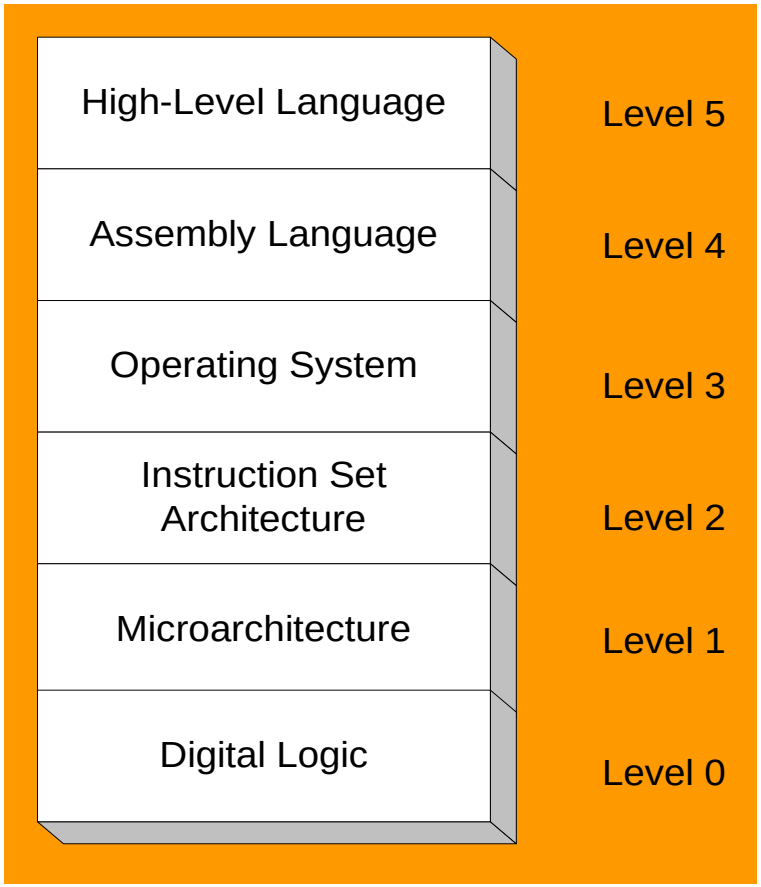
- Automatically adjusted by the push/pop operation.

- Stack grows towards the *low address*.

- **0-Address machine**: stores operands in a push-down stack

ADD	It causes the top two operands to be removed from the stack and added. The sum is placed at the TOS
PUSH A	Transfer A to TOS
PUSH B	
MULT	Remove A and B from stack and replace by A*B
PUSH C	
PUSH C	
MULT	Remove C and C, and replace the TOS by C
ADD	(C*C) + (A*B)
POP X	Transfer the results from TOS to X

Specific Machine Levels



Translating Languages

English: Display the sum of A times B plus C.

C++: cout << (A * B + C);

Assembly Language:

```
mov eax,A
mul B
add eax,C
call WriteInt
```

Intel Machine Language:

```
A1 00000000
F7 25 00000004
03 05 00000008
E8 00500000
```

Queues / FIFOs

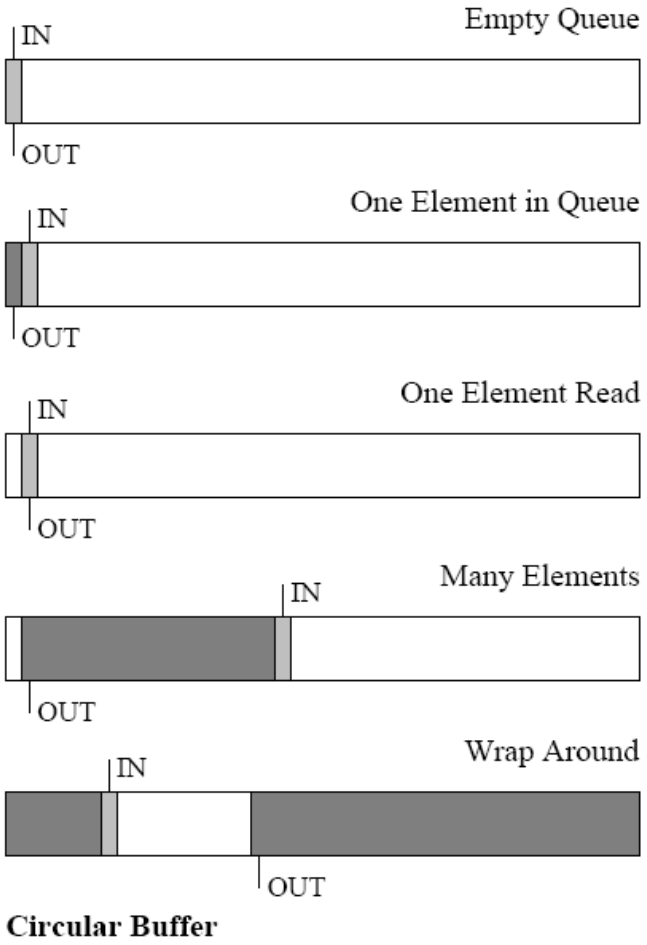
- First In — First Out
  - Buffer data between two entities:
    - Keyboard => Processor
    - Processor => Printer
    - Processor1 => Processor2
  - Scheduling as well:
    - Printer queue
    - Queue of processes for multi-tasking
  - Queues may use priority ranking

Queues / FIFOs

- First In — First Out
  - The two basic functions are:
    - APPEND an element on one end
    - REMOVE an element from the other
  - Frequently, one entity *appends* items to the queue and another *removes* it
  - Two moving pointers are needed:
    - IN: location for next APPEND
    - OUT: location for next REMOVE
  - Wrap-around is needed

Queues / FIFOs

- Pointers usually managed by hardware
- *Overflow* occurs when an element is appended to a full queue
- Underflow occurs when an element is removed from an empty queue
- May have Empty, Almost Full, Full flags



Circular Buffer

Stacks/LIFOs

- *Last In — First Out*
  - Stacks are used to *temporarily* store items
  - The two basic functions are:
    - PUSH an element on the top
    - POP the top element from the stack
  - Frequently, the same entity that *pushed* the item on stack also *pops* it
  - In contrast to the queue
    - Only *one* pointer needed – it points to the top element
    - PUSH and POP in different directions
    - Wrap-around is not needed

Stacks/LIFOs

- Processor Stack
  - Dedicated *Stack Pointer* (SP)
  - SP may be a general-purpose register
  - SP “grows” towards smaller addresses
  - Implementation of stack operations:

PUSH: Move NewItem, -(SP)

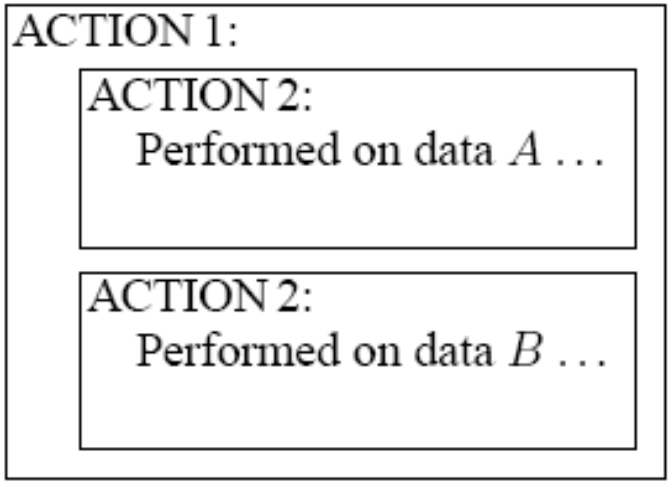
SP <- [SP] - 1, [SP] <- *NewItem*

POP: Move (SP)+,TopItem

TopItem <- [[SP]]; SP <- [SP] + 1

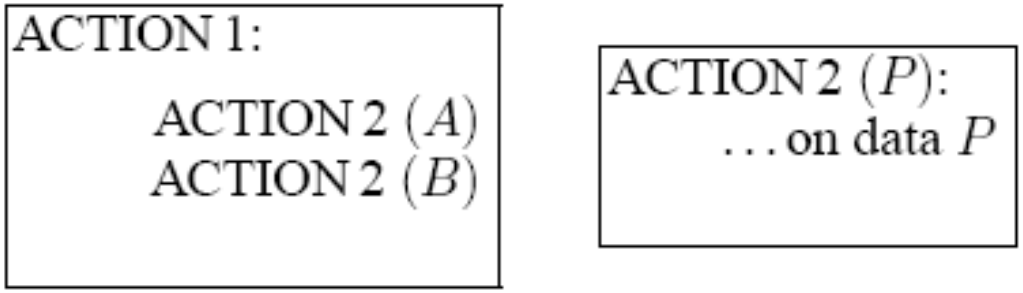
Subroutines

- Program segments may occur repeatedly:



Subroutines

- Solution: *Call* a subroutine multiple times



- Problems:
  - *Return address* needs to be known!
  - How to pass *parameters?*
  - Where to keep *local variables?*

Subroutine Calling

- Link register:
  - Call SUB LR <- [PC]
    PC <- *SUB*
  - Return PC <- [LR]
- Address on stack:
  - Call SUB SP <- [SP] – 1
    [SP] <- [PC]
    PC <- *SUB*
  - Return PC <- [[SP]]
    SP <- [SP] + 1

Parameter Passing

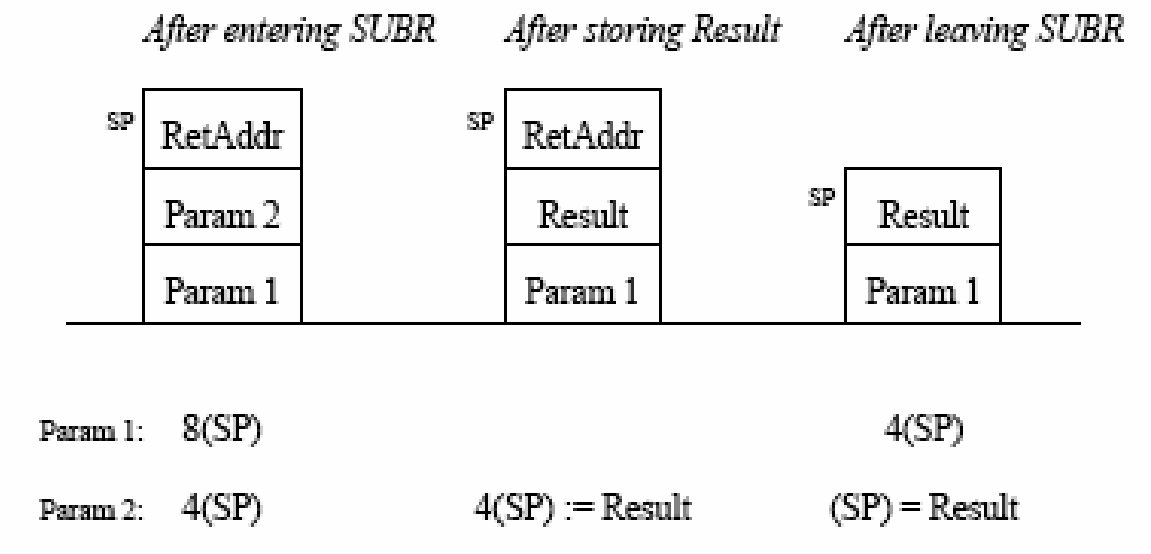
Stored in a designated memory area:					
MAIN	Move	parX, A	AddSub	Move	A, R0
	Move	parY, B		Add	B, R0
	Call	AddSub		Move	R0, C
	Move	C, Result		Return	
Stored in designated registers:					
MAIN	Move	parX, R0	AddSub	Move	R0, R2
	Move	parY, R1		Add	R1, R2
	Call	AddSub		Return	
	Move	R2, Result			
Using the stack:					
MAIN	Move	parX, -(SP)	AddSub	?	
	Move	parY, -(SP)		...	
	Call	AddSub		Return	
	Move	?, Result			

Parameter Passing through Stack

Simple Example Re-Using Parameter Space:			
MAIN	Move	PARAM1, -(SP)	; push Param1
	Move	PARAM2, -(SP)	; push Param2
	Call	SUBR	; call subroutine
	Move	(SP), RESULT	; get return value
	Add	#8, SP	; clean up stack
SUBR	Move	8(SP), R0	; load Param 1
	Add	4(SP), R0	; add Param 2
	Move	R0, 4(SP)	; store result
	Return		; Result = P1 + P2

Parameter Addressing

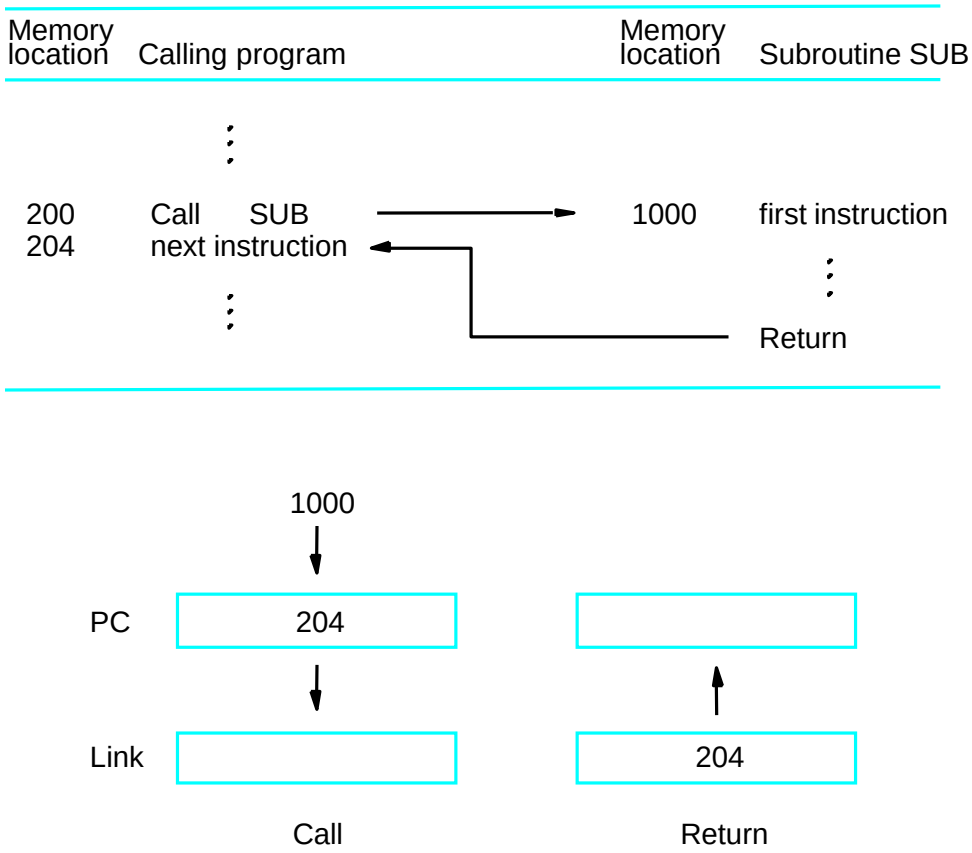
- Remember that *Return Address* is on the Stack too:



  - Assumptions:
    - two 32-bit parameters, no temporary items on stack, result size is 32-bit, result replaces parameter

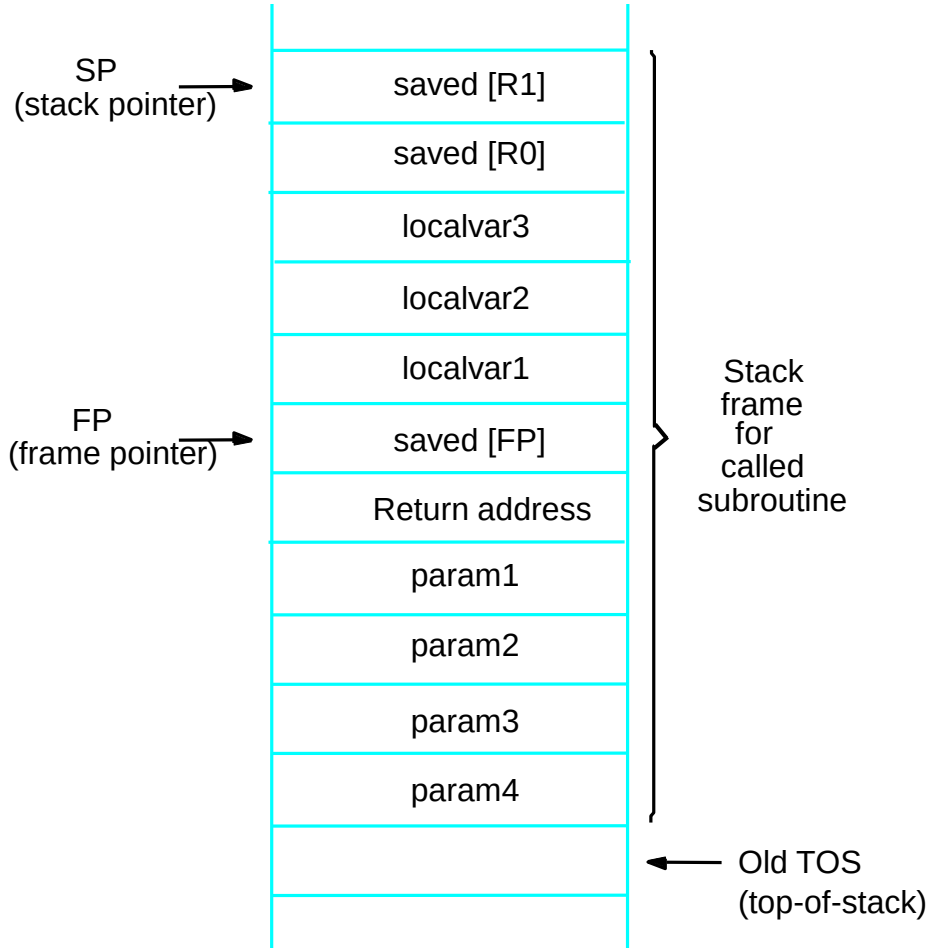
Subroutine Linkage

- *Call* instruction is used to call a subroutine
- Subroutine *returns* control to the calling program when it is done
- Stack is used to pass information



Subroutine Linkage

- Call pushes information onto stack
- Return pops information off of stack
- Stack pointer (SP) and Frame pointer (FP) used to access parameters and variables



Frame Pointer

- Each function has local memory associated with it to hold incoming parameters, local variables, and temporary variables.
- This region of memory is called a stack frame and is allocated on the process' stack.
- A frame pointer contains the base address of the function's frame.

Frame Pointer

- The code to access local variables within a function is generated in terms of offsets to the frame pointer.
- The stack pointer may change during the execution of a function as values are pushed or popped off the stack (such as pushing parameters in preparation to calling another function).
- The frame pointer doesn't change throughout the function.