



**CSE 4305**

**Computer Organization and Architecture**

# **Instruction Sets: Characteristics and Functions**

**Course Teacher: Md. Hamjajul Ashmafee**

**Lecturer, CSE, IUT**

**Email: [ashmafee@iut-dhaka.edu](mailto:ashmafee@iut-dhaka.edu)**

# Machine Instruction

- Normally, a **programmer** writes an application in high level language – very little of the architecture of the underlying machine is visible
- **Operation of a processor** is determined by its instructions – also called **computer instruction** or **machine instruction**– collection of those instruction named as **instruction set**
- **Machine instruction set** – a boundary where the **computer designer** (hardware oriented) and the **computer programmer** (software oriented) can view the machine in same way
- **Programming** in machine language (assembly language) – **concern** about registers and memory structure, data types supported by the machine, functions of ALU



# *Elements of Machine Instruction*

- Each instruction contains the information required by the processor for execution like:
  - **Operation Code:** Specifies the operation to be performed (ADD, I/O) specified by a code named as opcode
  - **Source Operand Reference:** Specifies one or more source operand as input
  - **Result Operand Reference:** Specifies operand where result or output will be stored
  - **Next Instruction Reference:** Tells the processor from where to fetch the next instruction after executing the current one – real or virtual address – **implicit** (next instruction) or **explicit** (supplied)



# *Areas of Operands*

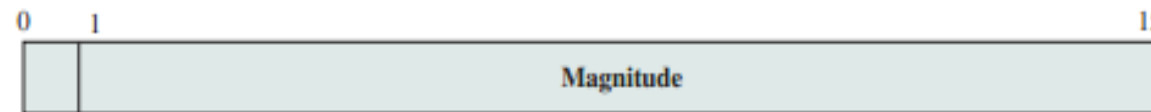
- Source or result **operands** can be in one of **four areas**:
  - **Main or virtual memory**: supplied by the instruction
  - **Processor registers**: Some processor contains register which can be referenced – if there is single register, reference is implicit otherwise unique identifier of those registers
  - **Immediate**: value of the operand itself contained in a field of the instruction
  - **I/O devices**: instruction specifies the I/O module and device for the operation (memory mapped or isolated I/O)

# Instruction Representation

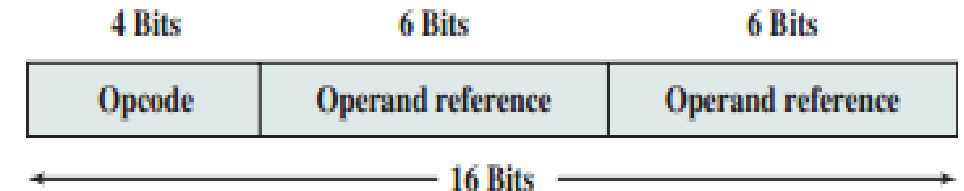
- In a computer, instructions are represented by a **sequence of bits** – divided into fields representing the elements of the instruction



(a) Instruction format



(b) Integer format



- For a particular instruction set, more than one instruction format
- During instruction execution, an instruction is read into an **IR register** – **processor** is able to extract data from different fields to perform operation

# Instruction Representation...

- But it is difficult to deal with binary representation of machine instructions – rather **symbolic representation** is used (e.g. **Load A, B**)
- **Opcodes** are represented by abbreviations – **mnemonics** – indicates the operation

ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
LOAD	Load data from memory
STOR	Store data to memory

- Operands are also represented symbolically – e.g. **Add R, Y** – R represents content of register R, Y refers to value contained in data location Y



# *Instruction Representation...*

- It is possible to write machine language program in symbolic form – **each symbolic opcode has its fixed binary representation** and **the location of the symbolic operands are specified by the programmer**
- A **program** should be there what can accept this symbolic input, converts the opcodes and operand reference to binary form and construct the binary machine instructions – So set of the instructions must be sufficient to execute any instruction from high level language (HLL)
- Now machine language programming is **obsolete** – usually programs are written **in high level language** – but is also required for describing computer operations



# *Relation between HLL and Machine Language*

- If any **high level language (HLL)** instruction is written as:

$$X = X + Y$$

How it will be **accomplished** in machine instructions?

**Answer:**

Let variables **X** and **Y** are corresponding to the location 513 and 514. Then, with simple set of machine instructions, it could be accomplished as:

1. Load a register with the contents of memory location 513.
2. Add the contents of memory location 514 to the register.
3. Store the contents of the register in memory location 513.

A **HLL** expresses operations in a concise algebraic form and **machine language** expresses operations in a basic form of data movement using registers





# Instruction Types

Categorized into **four** types:

1. **Data Processing:** Arithmetic and Logic instructions – **arithmetic instructions** provide computational capabilities for processing numeric data – **logic instructions** operate on bits rather than numbers – performed by **ALU** with **registers** in processor
2. **Data Storage:** Storing (moving) of data into or out of register or memory locations – using memory instructions like LOAD, STORE, etc.
3. **Data Movement:** I/O instructions – to move data and programs into memory and results to the user
4. **Control:** Test or branch instructions – **to test** the value or status of the data or computation and/ or **branch** to different set of instructions based on decision made



# *Number of Addresses in an Instruction*

- A way to **describe** the processor architecture – but become **obsolete** with the increasing complexity of processor design
- **How many addresses** will be required in an instruction?
  - In arithmetic and logic operations requires the **most operands** – **unary** or **binary** (source operands) – if including result, it may require **third address** (destination operand) – with next instruction address (may be implicit), it might require **total 4 addresses**
  - In most architectures, many instructions have **one, two** or **three** operand addresses – **or even more** (we'll explore later on)

# Three-Address Instructions

$$Y = \frac{A - B}{C + (D \times E)}$$

Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

**T is a temporary location to store intermediate results**

# Two-Address Instructions

$$Y = \frac{A - B}{C + (D \times E)}$$

<u>Instruction</u>	<u>Comment</u>
MOVE Y, A	$Y \leftarrow A$
SUB Y, B	$Y \leftarrow Y - B$
MOVE T, D	$T \leftarrow D$
MPY T, E	$T \leftarrow T \times E$
ADD T, C	$T \leftarrow T + C$
DIV Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions

**T is a temporary location to store intermediate results**

# One-Address Instructions

$$Y = \frac{A - B}{C + (D \times E)}$$

<u>Instruction</u>	<u>Comment</u>
LOAD D	AC ← D
MPY E	AC ← AC × E
ADD C	AC ← AC + C
STOR Y	Y ← AC
LOAD A	AC ← A
SUB B	AC ← AC - B
DIV Y	AC ← AC ÷ Y
STOR Y	Y ← AC

(c) One-address instructions


**Accumulator is an implicit operand here.**



# Comparison among 3 Instruction Formats

- Three-address instructions are **not common** as they are lengthy
- With two-address instructions, **one address** must do double duty as both an operand and a result (e.g. **SUB Y, B**)
- But two-address instructions have **some problems** – it changes the value of an operand as it is used also as result – some **temporary locations** are required there to avoid that situation
- For one-address instruction, **one operand must be implicit** – a processor register known as **accumulator (AC)** – both stores an operand and the result (double role)
- There are also some zero-address instructions which requires **no operand** – (**stack operations** - push, pop)

# Comparison Summarization

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	$A \leftarrow B \text{ OP } C$
2	OP A, B	$A \leftarrow A \text{ OP } B$
1	OP A	$AC \leftarrow AC \text{ OP } A$
 0	OP	$T \leftarrow (T - 1) \text{ OP } T$

AC = accumulator

T = top of stack

$(T - 1)$  = second element of stack

A, B, C = memory or register locations



# Comparison Summarization...

- Next instruction address is **implicit** in the instructions
- Fewer addresses per instruction – more primitive instruction, requires less complex processor, shorter in length - but longer and complex program, longer execution time
- **Use of registers** in instructions **speed up** the execution time as memory location in instructions requires memory access which slows down the execution – but use of more registers is **expensive** and more registers means **more bits for register reference**



# Instruction Set Design

- **Instruction Set** is a mean to **control the processor** – programmer's requirements should be considered
- **Fundamental issues** to design instruction sets:
  - **Operation Repertoire:** how many operations, which operations, how complex operations (?)
  - **Data types:** various types of data based on operations (?)
  - **Instruction format:** Instruction length in bits, number of addresses, size of various fields in instruction
  - **Registers:** number of registers can be referenced by the instructions and their respective uses
  - **Addressing:** the mode(s) by which the address of the operand is specified



# *Types of Operands*

General categories of operands (data) on which machine instructions are operated:

## **1. Addresses:**

- They are in fact a **form of data**
- Considered as **unsigned integer**
- Sometimes **some calculations** are needed on the operand references to determine the main or virtual memory address

# Types of Operands...

## 2. Numbers:

- **Numeric** data types
- Also required in processing non numeric data (string length, counters)
- They are **different from** ordinary mathematical data as their **limitations** of magnitude representation and precision of floating point numbers
- Different **issues** are there - **rounding, overflow, underflow** (try to store smaller value than its minimum range in negative numbers)
- **Three types of numerical data** common in computer – Binary integer, floating point and decimal (packed decimal – **BCD** – in multiple of 8 bits)
- Binary values requires **binary to decimal conversion** or vice versa
- BCD doesn't require conversion but they are less compact – Negative value represented as including a 4 bit sign digit at left or right end – 1100 (+) and 1101(-) – direct manipulation is possible.

# *Types of Operands...*

## 3. Characters:

- Common form of data
- Characters are **easily presented to human beings** to understand but **for machine a number of codes are devised to represent them by sequence of bits**
- Common **representation** – Morse Code, IRA, ASCII, EBCDIC
- **ASCII** represents each characters with a unique 7 bits code – an additional bit can be there as parity bit
- Characters can be **printable** or **control**
- Represents digits as **011XXXX** to convert them easily in BCD



# *Types of Operands...*

## 4. Logical Data:

- Each word or addressable unit can be treated as **single unit of data** – consider each bit as a data item – viewed as logical data (bit oriented)
- **Advantages:** data can be stored bit wise (Boolean values) more efficiently and manipulate the bits easily (shifting significant bits in floating point operations)

# *Types of Operations: Data Transfer*

- **Fundamental** operation
- **Instruction** specifies following information:
  - **Location(s)** of source and destination operands (reg. to reg., reg. to memory, memory to reg., memory to memory transfer)
  - **Length of data** to be transferred (8, 16, 32, 64 bits)
  - **Addressing modes** for all operands
- If both the source and destination are registers, data only transferred from one register to another – takes less time
- Otherwise, if both of them are in memory, the processor will do:
  - Calculate the memory address based on addressing modes
  - If the addresses refers to virtual memory, translate it from virtual to real memory addressing
  - Determine the addressed item is in cache or not
  - If not, issue a command to memory module

# *Types of Operations: Data Transfer...*

- Common instructions:

Type	Operation Name	Description
Data transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination

# *Types of Operations: Data Transfer...*

- Processor Actions:

Data transfer	Transfer data from one location to another
	<p>If memory is involved:</p> <ul style="list-style-type: none"><li>Determine memory address</li><li>Perform virtual-to-actual-memory address transformation</li><li>Check cache</li><li>Initiate memory read/write</li></ul>



# *Types of Operations: Data Transfer...*

- Example: IBM EAS/390 data transfer operations

Operation Mnemonic	Name	Number of Bits Transferred	Description
L	Load	32	Transfer from memory to register
LH	Load Halfword	16	Transfer from memory to register
LR	Load	32	Transfer from register to register
LER	Load (short)	32	Transfer from floating-point register to floating-point register
LE	Load (short)	32	Transfer from memory to floating-point register
LDR	Load (long)	64	Transfer from floating-point register to floating-point register
LD	Load (long)	64	Transfer from memory to floating-point register
ST	Store	32	Transfer from register to memory
STH	Store Halfword	16	Transfer from register to memory
STC	Store Character	8	Transfer from register to memory
STE	Store (short)	32	Transfer from floating-point register to memory
STD	Store (long)	64	Transfer from floating-point register to memory

# *Types of Operations: Arithmetic*

- Basic operations are add, subtract, multiply, divide
- Performed on signed integer (fixed point), floating point and packed decimal numbers
- Other single operand operations:
  - Absolute
  - Negate
  - Increment
  - Decrement
- **ALU** perform the job
- Involve data transfer operations to different register locations – AC, IR

# *Types of Operations: Arithmetic...*

- Common instructions:

Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand

# *Types of Operations: Arithmetic...*

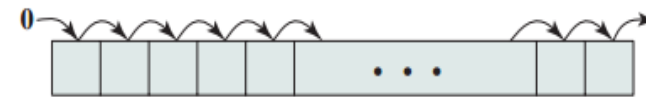
- Processor Actions:

Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags

# Types of Operations: Logical

- Operations to manipulate individual data bits – “**bit twiddling**”
- Also includes a variety of shifting and rotating functions – shift right/ left, rotate right/ left, arithmetic shift right/ left

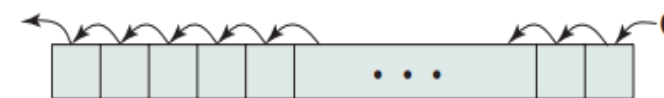
P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P = Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1



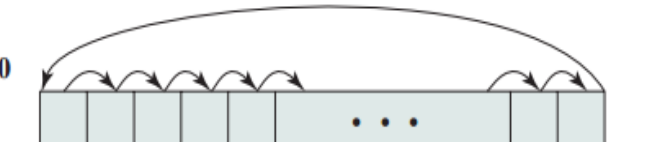
(a) Logical right shift



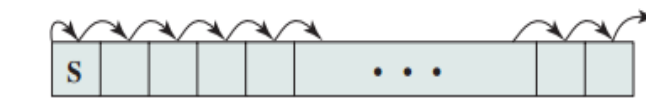
(d) Arithmetic left shift



(b) Logical left shift



(e) Right rotate



(c) Arithmetic right shift



(f) Left rotate

# Types of Operations: Logical...

- Basic logical operations:

Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT	(complement) Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end

- Processor Action:

Logical	Same as arithmetic
---------	--------------------



# *Types of Operations: Conversion*

- **Change the format of data** – e.g. decimal to binary
- **Translate (TR) instructions** – used to convert from one 8-bit code to another taking three operands

## ***TR R1 (L), R2***



- **R2** contains the starting address of a table of 8-bit codes
- **L** specifies how many bytes will be translated
- **R1** contains the first byte that is going to be translated
- **Each byte** being replaced by the **content of a table entry indexed** by that **byte** – e.g. EBCDIC to IRA
- **Example:** If location 1000 contains the hexadecimal translation code from *EBCDIC* to *IRA* and location 2100-2103 contain F1, F9, F8, F4; if R1 contains 2100 and R2 contains 1000 – ***TR R1 (4), R2*** results in \_\_\_\_\_ ? (2100-2103 contains 31 39 38 34)

# *Types of Operations: Conversion...*

- Basic logical operations:

Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary) <b>Using calculation</b>

- Processor Action:

Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
------------	--





# *Types of Operations: Input/ Output*

- Programmed I/O
  - Isolated I/O
  - Memory Mapped I/O
- Interrupt Driven I/O
- DMA
- I/O processor
- I/O instructions

**Already discussed in chapter 7**

# *Types of Operations: Input/ Output ...*

- Basic logical operations:

Input/output	Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination

- Processor Action:

I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

# *Types of Operations: System Control*

- These instructions **only be executed** while the processor is **in a certain privileged state**
- **Reserved** for operating system
- Some examples of operations:
  - **Read** control register
  - **Alter** control register
  - **Read/ modify** the storage protection key (to improve reliability of the program)
  - **Access** to process control blocks

# *Types of Operations: Transfer of Control*

- Generally the **next instruction** to be performed is the immediately following the current instruction – but sometimes this **sequence might be broken** after executing certain instructions by updating the PC
- It is **required when**:
  - To implement **LOOP** – to execute same instructions thousands times
  - To take **decision** – if else then condition
  - To **break the bigger task into smaller pieces** that can be worked on one at a time
- Most common **instructions**: Branch, Call (procedure), Skip



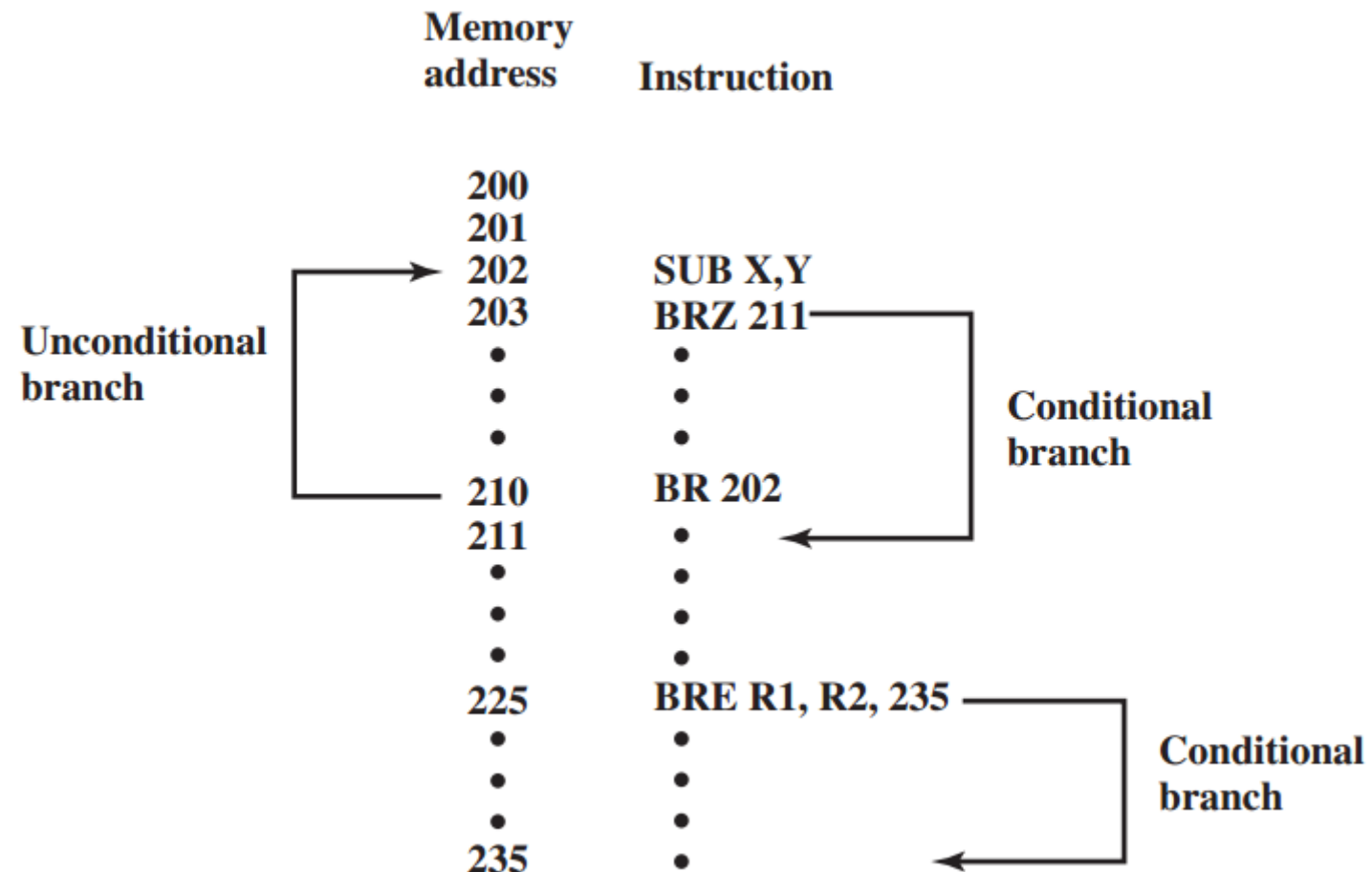
# *Types of Operations: Transfer of Control...*

## **Branch:**

- Also known as **Jump**
- It has one of its **operands** the address of the next address to be executed
- **Two types** of branch instructions :
  - **Conditional Branch:** based on a condition; if it is satisfied, the branch is made
  - **Unconditional Branch:** always branch is taken
- For **generating the condition** to be tested in conditional branch:
  - **Check the bits of condition code** (flags) updated as a result of most recent operations (e.g. add, sub can generate 0, positive, negative or overflow values can be reflected through the flags – BRP X, BRN X, BRZ X, BRO X)
  - Another approach to **compare in an instruction** and also mention the branch address to follow if the comparison satisfied (e.g. BRE R1, R2, X)
- Branch can be either **forward (higher address)** or **backward (lower address)**

# Types of Operations: Transfer of Control...

## Branch:



# *Types of Operations: Transfer of Control...*

## **Skip:**

- Includes an **implied address** – **skip one instruction**
- **Implicit address** = (next address + one instruction length) – no more explicit address
- **Example:** Increment-and-skip-if-zero (ISZ)

```
301  
:  
:  
309 ISZ R1  
310 BR 301  
311
```



# *Types of Operations: Transfer of Control...*

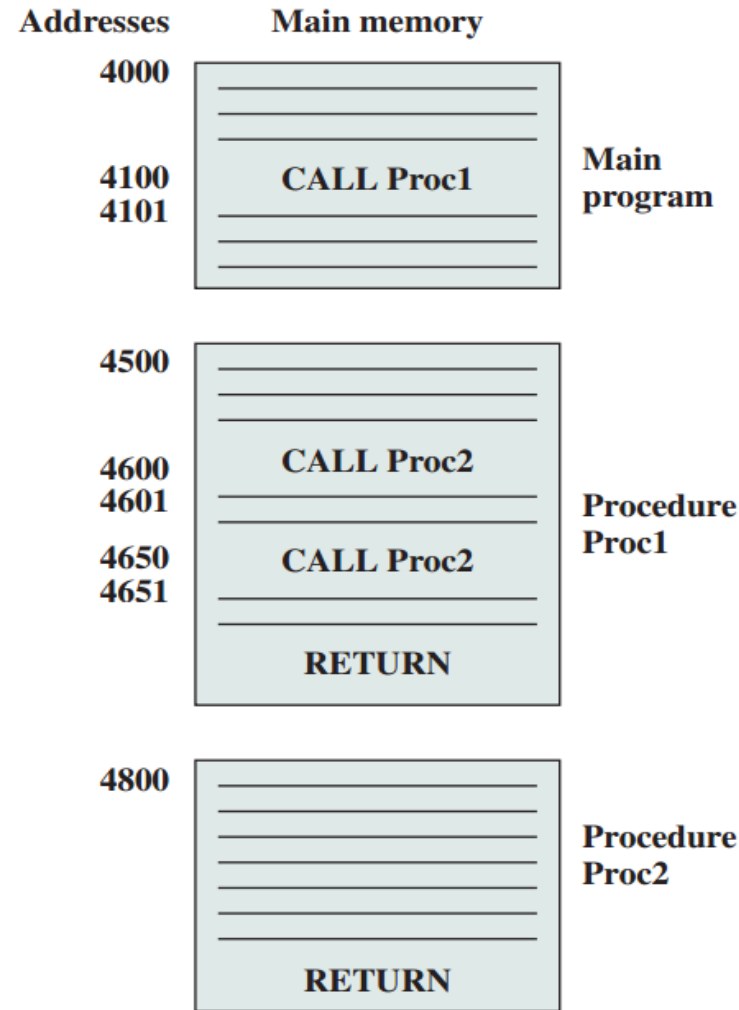
## **Procedure Call:**

- Most important **innovation in development** of programming language
- **Self contained computer program** –may be a **part of another larger program** which can **call** it any point
- **Reasons** to use procedure:
  - **Economy:** Use same code multiple times to use memory space efficiently
  - **Modularity:** Perform large task dividing into smaller units to ease the programming task
- Procedure involves **two basic instructions**:
  - **Call:** Branching from the present location to the procedure
  - **Return:** Returning from the procedure to the location from where it is called

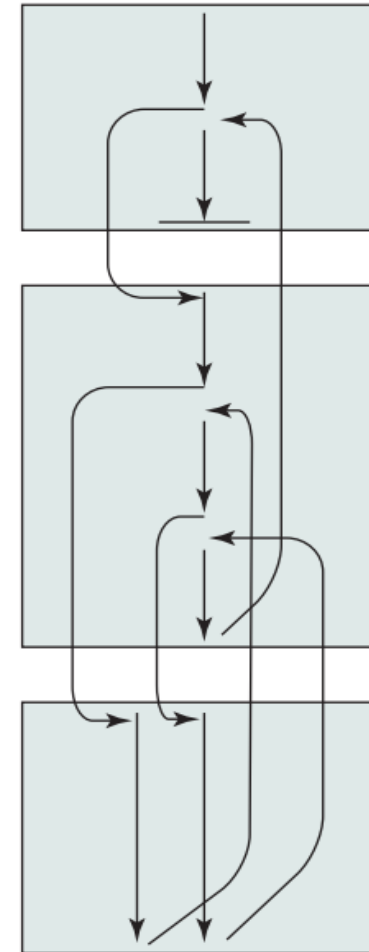


# Types of Operations: Transfer of Control...

## Nested Procedure



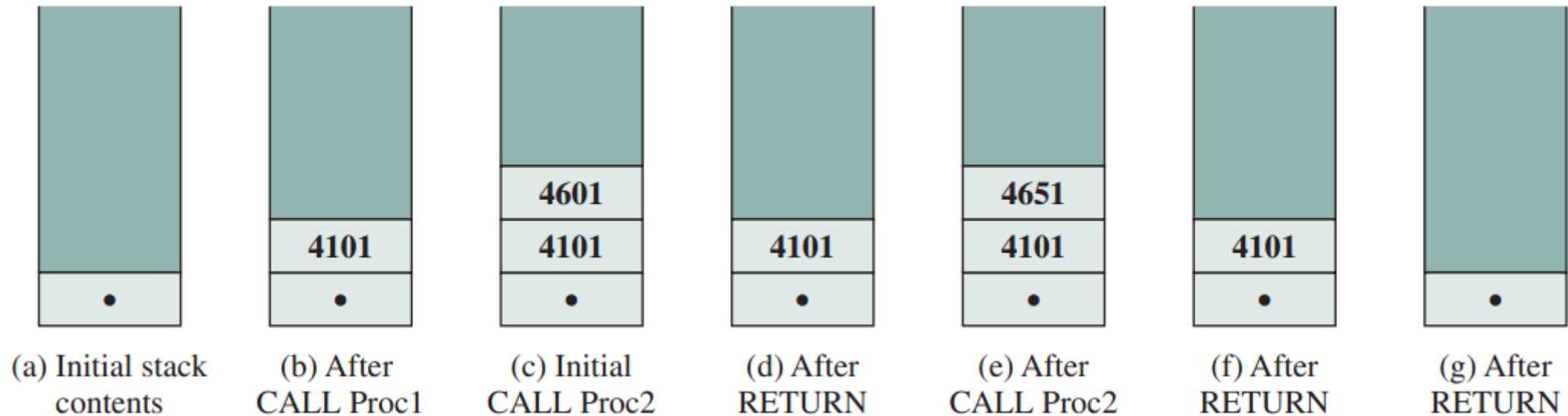
(a) Calls and returns



(b) Execution sequence

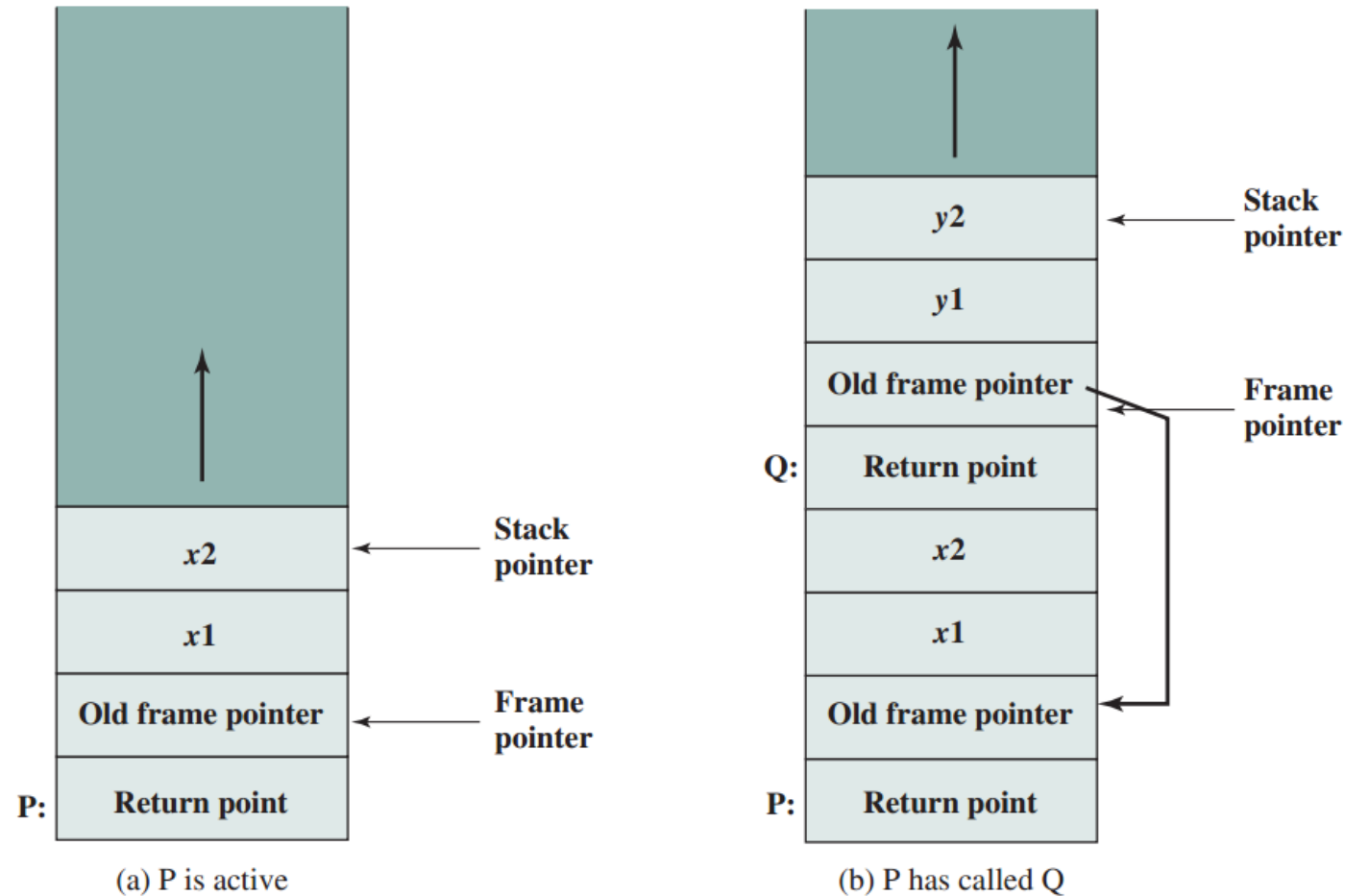
# *Types of Operations: Transfer of Control...*

- Use of stacks to implement nested subroutines:



# Types of Operations: Transfer of Control...

- Stack frame growth:



# Byte Ordering: Endianness

- How the bytes within a word and bits within a byte are represented?
  - e.g. 32 bit hexadecimal value 12345678 is stored in 32 bit word of byte addressable memory location 184 – two ways to store them

**Address Value**

184	12
185	34
186	56
187	78

**Address Value**

184	78
185	56
186	34
187	12

- Storing **MSB** in the lowest numerical byte address – **big endian** – left to right order like writing in western culture language
- Storing **LSB** in the lowest numerical byte address – **little endian** – right to left order of arithmetic operation



# Byte Ordering: Endianness...

- The concept of endianness arises when we treat multiple bytes as a single data unit with single address
- X86, VAX, alpha are **little** endian machines
- Motorola 680x0, SUN SPARC, RISC machines are **big** endian
- In both scheme data address should be same
- **Byte order** of big endian and little endian is **reverse of each other**
- Endianness **doesn't affect** the order of the data items within a structure
- There is **no general consensus** to declare which one is better



# *Byte Ordering: Endianness...*

- Good points for **Big** endian:
  - **Character string sorting**: to compare string
  - **Decimal/IRA(ASCII) dumps**: printing values from left to right - storing both integers and characters in same order (MSB comes first)
- Good points for **Little** endian:
  - Good for **arithmetic operations** like addition, subtraction
- **Bit ordering**:
  - Counting from zero or one?
  - Follow big or little endian to assign address?



# Endianness: Example

```
struct{  
    int    a;        //0x1112_1314        word  
    int    pad;      //  
    double b;        //0x2122_2324_2526_2728  doubleword  
    char*   c;        //0x3132_3334        word  
    char    d[7];     //'A','B','C','D','E','F','G'  byte array  
    short   e;        //0x5152        halfword  
    int     f;        //0x6162_6364        word  
} s;
```

# Endianness: Example...

		Big-endian address mapping							
Byte address		11	12	13	14				
00		00	01	02	03	04	05	06	07
		21	22	23	24	25	26	27	28
08		08	09	0A	0B	0C	0D	0E	0F
		31	32	33	34	'A'	'B'	'C'	'D'
10		10	11	12	13	14	15	16	17
		'E'	'F'	'G'		51	52		
18		18	19	1A	1B	1C	1D	1E	1F
		61	62	63	64				
20		20	21	22	23				

		Little-endian address mapping								Byte address
						11	12	13	14	
		07	06	05	04	03	02	01	00	00
		21	22	23	24	25	26	27	28	
		0F	0E	0D	0C	0B	0A	09	08	08
		'D'	'C'	'B'	'A'	31	32	33	34	
		17	16	15	14	13	12	11	10	10
				51	52		'G'	'F'	'E'	
		1F	1E	1D	1C	1B	1A	19	18	18
						61	62	63	64	
						23	22	21	20	20





# Endianness: Example...

00	11
	12
	13
	14
04	
08	21
	22
	23
	24
0C	25
	26
	27
	28
10	31
	32
	33
	34
14	'A'
	'B'
	'C'
	'D'
18	'E'
	'F'
	'G'
1C	51
	52
20	61
	62
	63
	64

(a) Big endian

00	14
	13
	12
	11
04	
08	28
	27
	26
	25
0C	24
	23
	22
	21
10	34
	33
	32
	31
14	'A'
	'B'
	'C'
	'D'
18	'E'
	'F'
	'G'
1C	52
	51
20	64
	63
	62
	61

(b) Little endian