Computer Organization and Software Systems

Session 5

BITS Pilani
Pilani Campus

Pruthvi Kumar K R

BITS Pilani
Pilani Campus

CISC Instruction Set (Intel x86 as an example)

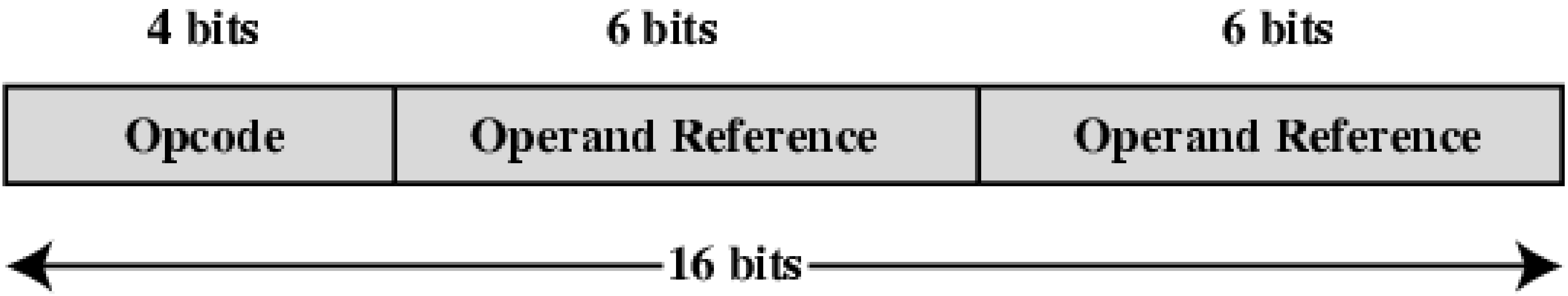
Today's Session

Contact Hour	List of Topic Title	Text/Ref Book/external resource
9-10	• **Instruction Set Architecture - CISC Vs RISC** • CISC Instruction Set (Intel x86 as an example) • Machine Instruction Characteristics • Types of Operands • Types of Operations • Addressing Modes • Instruction Formats	T1

Introduction

- What is an Instruction Set?
  - The complete collection of instructions that are understood by a CPU
- Elements of an Instruction
  - Operation code (Op code)
  - Source Operand reference
  - Result Operand reference
  - Next Instruction Reference
- Source and Destination Operands can be found in four areas
  - Main memory (or virtual memory or cache)
  - CPU register
  - I/O device
  - Immediate

Simple Instruction Format



4 bits	6 bits	6 bits
Opcode	Operand Reference	Operand Reference

16 bits

- During instruction execution, an instruction is read into an instruction register (IR) in the processor.
- The processor must be able to extract the data from the various instruction fields to perform the required operation.
- Opcodes are represented by abbreviations, called ***mnemonics***

  Example: ADD AX, BX ➔ Add instruction

Instruction Types

- Data processing : Arithmetic and logic instructions
- Data storage (main memory) : Movement of data into or out of register and or memory locations
- Data movement (I/O) : I/O instructions
- Program flow control : Test and branch instructions

Number of Addresses (1/2)

- 3 addresses
  - Result, Operand 1, Operand 2
  - c = a + b; add c, a, b
  - May be a forth - next instruction (usually implicit)
  - Needs very long words to hold everything
- 2 addresses
  - One address doubles as operand and result
  - a = a + b : add a, b
  - Reduces length of instruction
  - The original value of a is lost.

Number of Addresses (2/2)

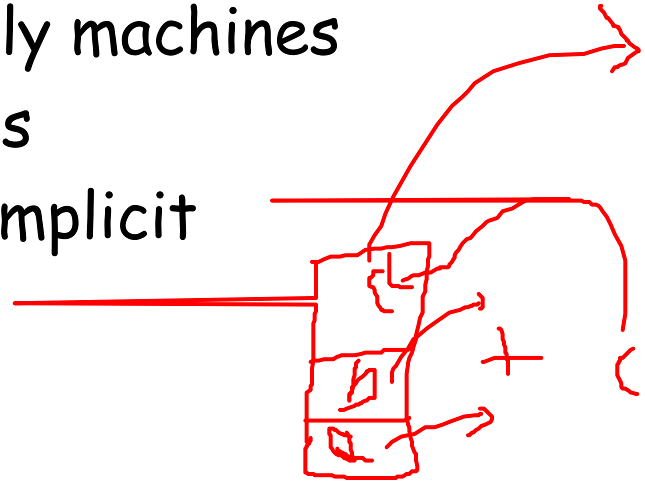
- 1 address
  - Implicit second address
  - Usually a register (accumulator)
  - Common on early machines
- 0 (zero) addresses
  - All addresses implicit
  - Uses a stack
  - e.g. c = a + b

    push a

    push b

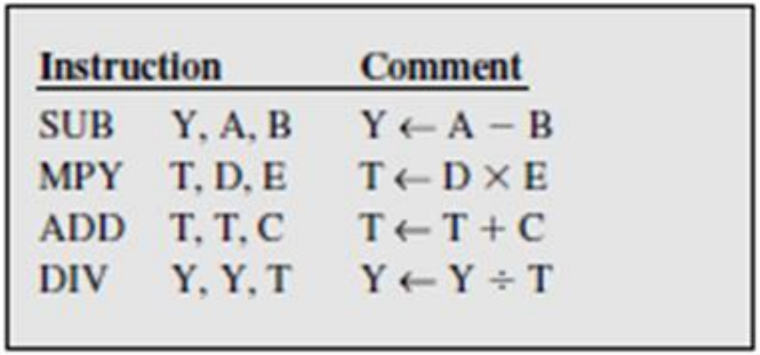
    add

    pop c



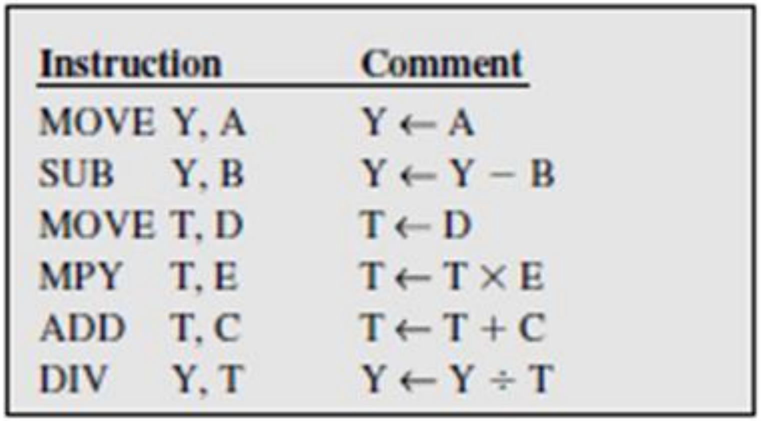
Example

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



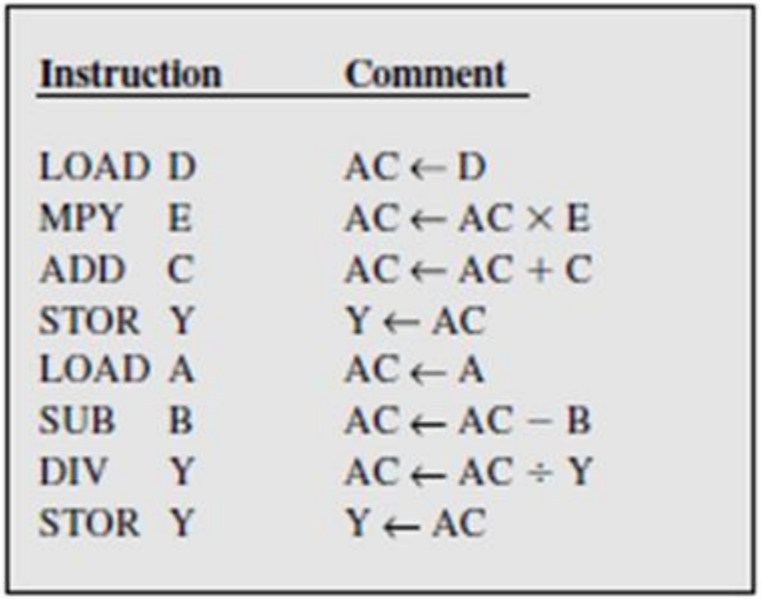
Instruction	Comment
SUB Y, A, B	$Y \leftarrow A - B$
MPY T, D, E	$T \leftarrow D \times E$
ADD T, T, C	$T \leftarrow T + C$
DIV Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions



Instruction	Comment
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



Instruction	Comment
LOAD D	$AC \leftarrow D$
MPY E	$AC \leftarrow AC \times E$
ADD C	$AC \leftarrow AC + C$
STOR Y	$Y \leftarrow AC$
LOAD A	$AC \leftarrow A$
SUB B	$AC \leftarrow AC - B$
DIV Y	$AC \leftarrow AC \div Y$
STOR Y	$Y \leftarrow AC$

(c) One-address instructions

How Many Addresses

- Fewer addresses
  - More Primitive instructions, shorter length instructions
  - Less complex instructions, hence requires less complex hardware
  - More instructions per program
    - Longer programs
    - More complex programs
    - Longer execution time
- Multiple address instructions
  - Lengthy instructions
  - More registers
    - Inter-register operations are quicker
  - Fewer instructions per program

Instruction set Design Decisions

- Operation repertoire
  - How many ops?
  - What can they do?
  - How complex are they?
- Data types
- Instruction formats
  - Length of op code field
  - Number of addresses
- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers?
- Addressing modes

Types of Operand

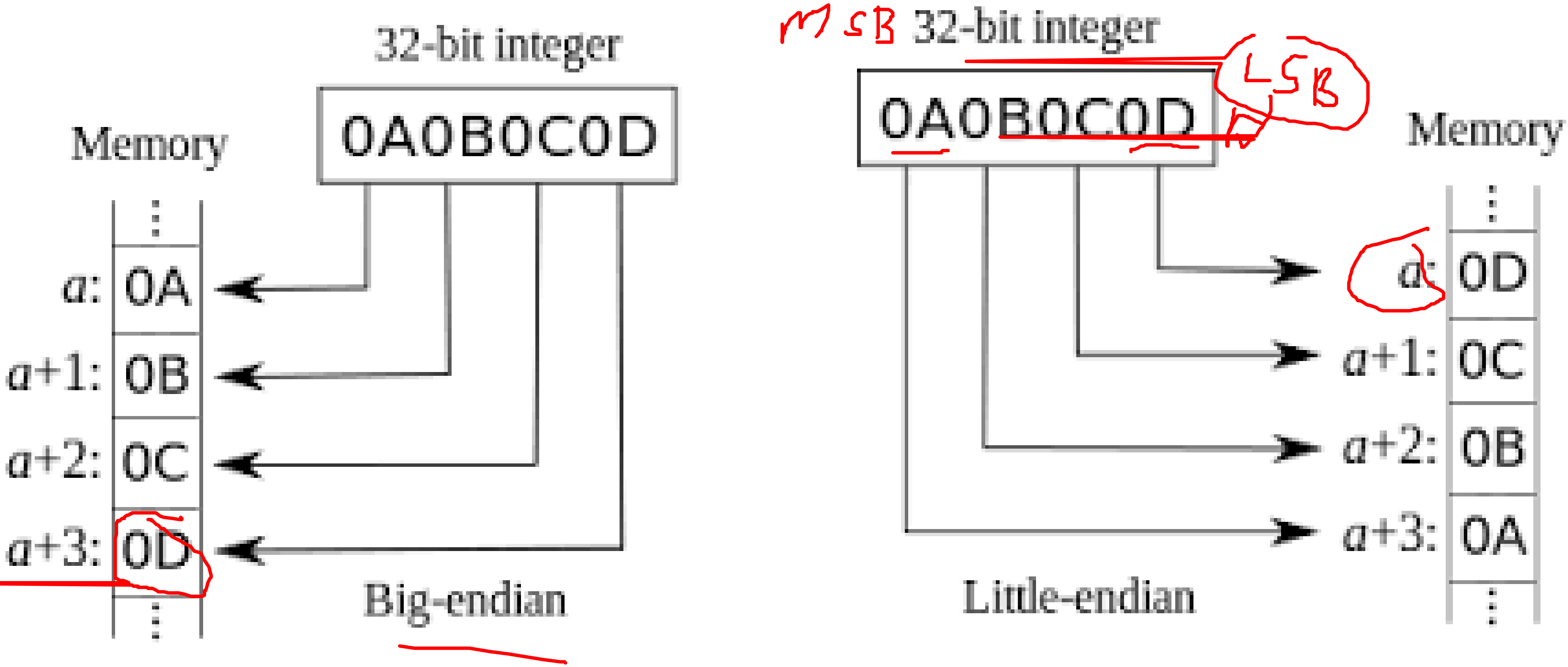
- Machine instructions operate on data
- General categories of data
  - Addresses
  - Numbers
    - Binary integer or binary fixed point, floating point, decimal
  - Characters
    - ASCII etc.
  - Logical Data
    - Bits or flags
- Packed Decimal
  - 36 : 0011 0110

Binary Representation

32 16 8 4 2 1

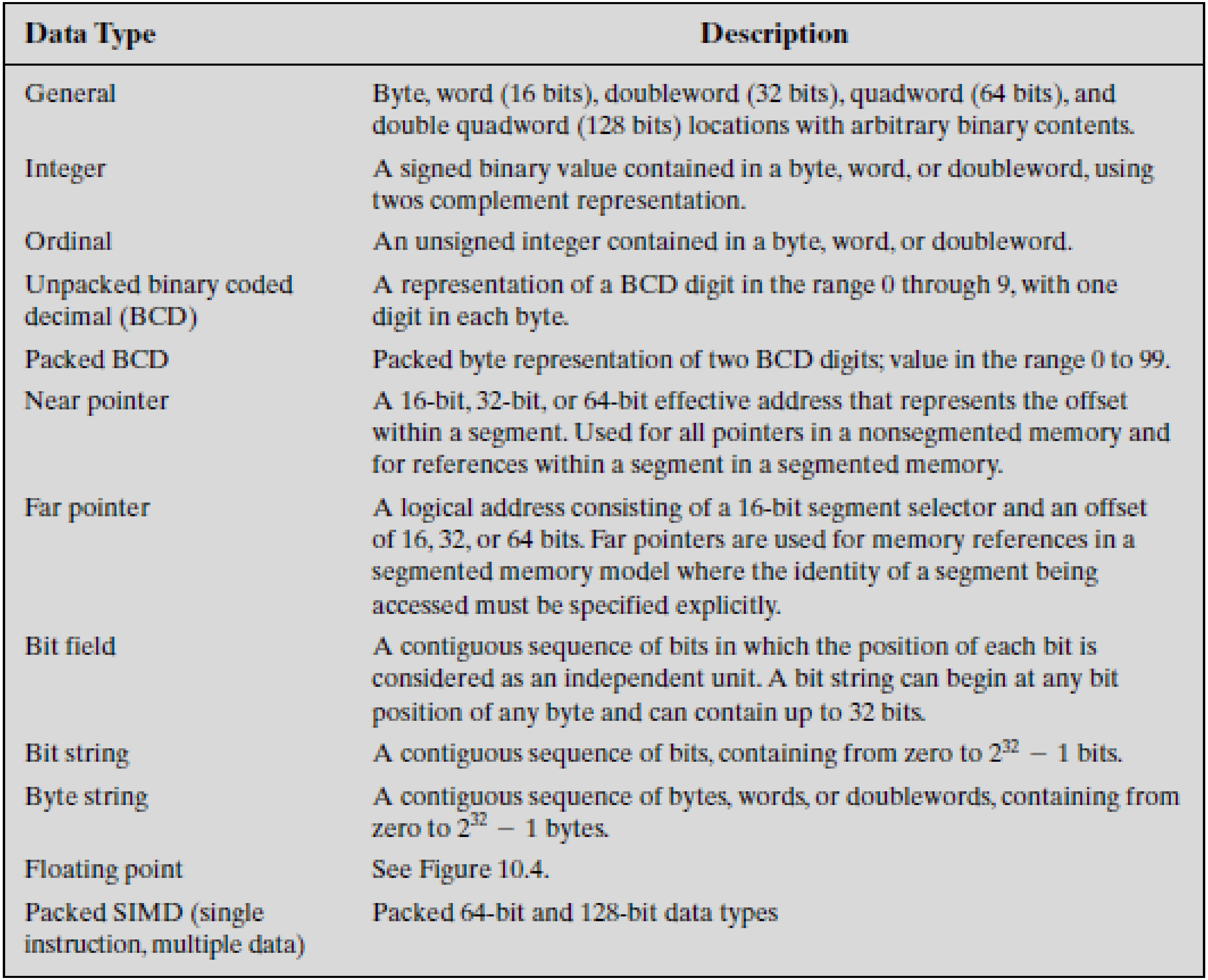
1 0 0 1 0 0

Byte Ordering



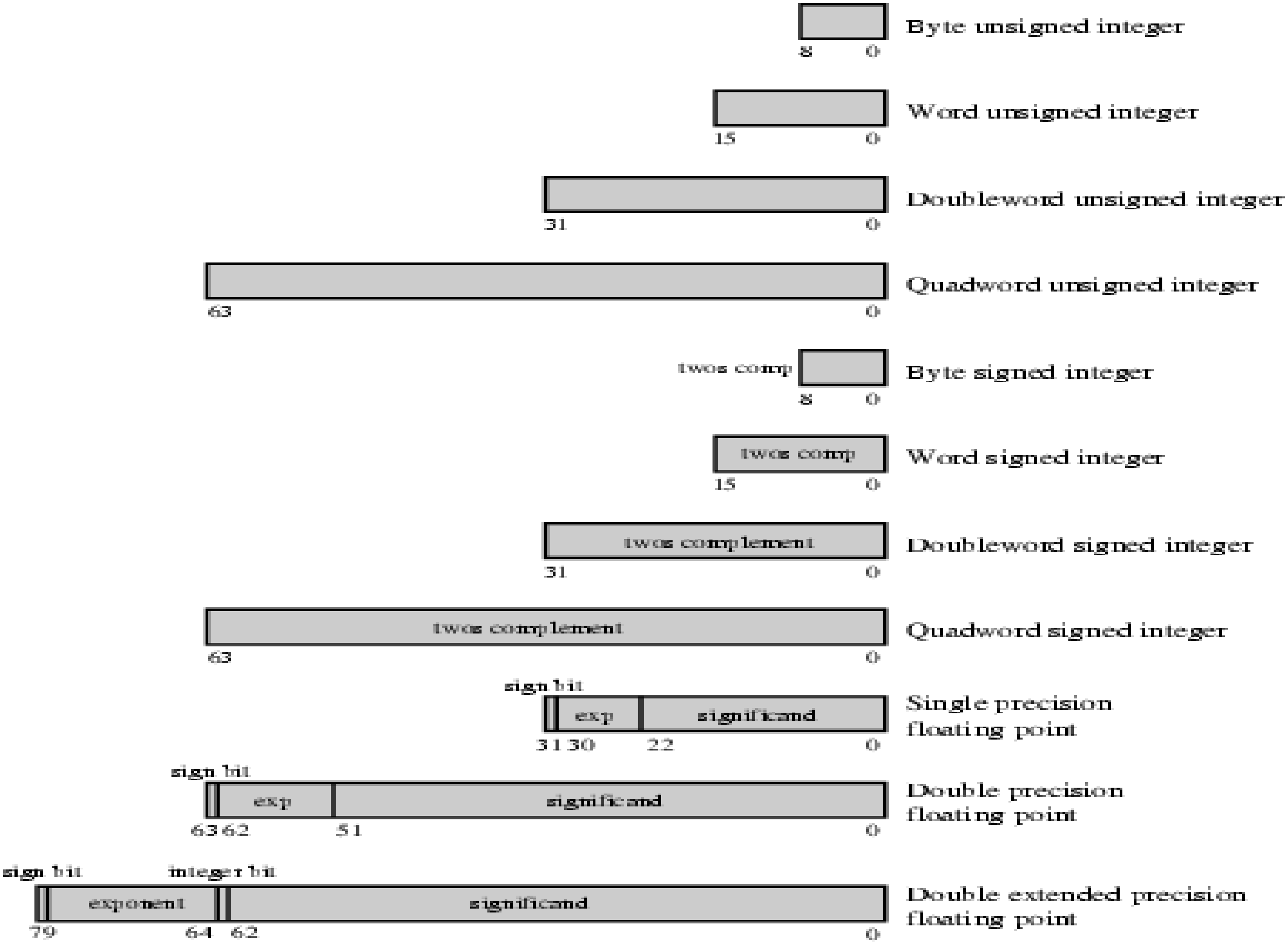
x86 Data Types

- General – Byte, Word, double word, quadword, double quad word - arbitrary binary contents
- Integer – signed binary using two's complement representation
- Ordinal - unsigned integer
- Unpacked BCD - One digit per byte
- Packed BCD - 2 digits per byte
- Near Pointer - 32 bit offset within segment
- Far pointer -
- Bit field : A contiguous sequence of bits in which the position of each bit is considered as an independent unit.
- Bit and Byte String
- Floating Point



Data Type	Description
General	Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.
Integer	A signed binary value contained in a byte, word, or doubleword, using twos complement representation.
Ordinal	An unsigned integer contained in a byte, word, or doubleword.
Unpacked binary coded decimal (BCD)	A representation of a BCD digit in the range 0 through 9, with one digit in each byte.
Packed BCD	Packed byte representation of two BCD digits; value in the range 0 to 99.
Near pointer	A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.
Far pointer	A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.
Bit field	A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.
Bit string	A contiguous sequence of bits, containing from zero to $2^{32} - 1$ bits.
Byte string	A contiguous sequence of bytes, words, or doublewords, containing from zero to $2^{32} - 1$ bytes.
Floating point	See Figure 10.4.
Packed SIMD (single instruction, multiple data)	Packed 64-bit and 128-bit data types

x86 Numeric Data Formats



Byte unsigned integer (8 … 0)

Word unsigned integer (15 … 0)

Doubleword unsigned integer (31 … 0)

Quadword unsigned integer (63 … 0)

Byte signed integer: twos comp (8 … 0)

Word signed integer: twos comp (15 … 0)

Doubleword signed integer: twos complement (31 … 0)

Quadword signed integer: twos complement (63 … 0)

Single precision floating point: sign bit, exp, significand (31 30 … 22 … 0)

Double precision floating point: sign bit, exp, significand (63 62 … 51 … 0)

Double extended precision floating point: sign bit, exponent, integer bit, significand (79 … 64 62 … 0)

Types of Operation

- Data Transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System Control
- Transfer of Control

Data Transfer

- Specify
  - Source
  - Destination
  - Amount of data
- Action:

1. Calculate the memory address, based on the address mode
2. If the address refers to virtual memory, translate from virtual to real memory address.
3. Determine whether the addressed item is in cache.
4. If not, issue a command to the memory module.

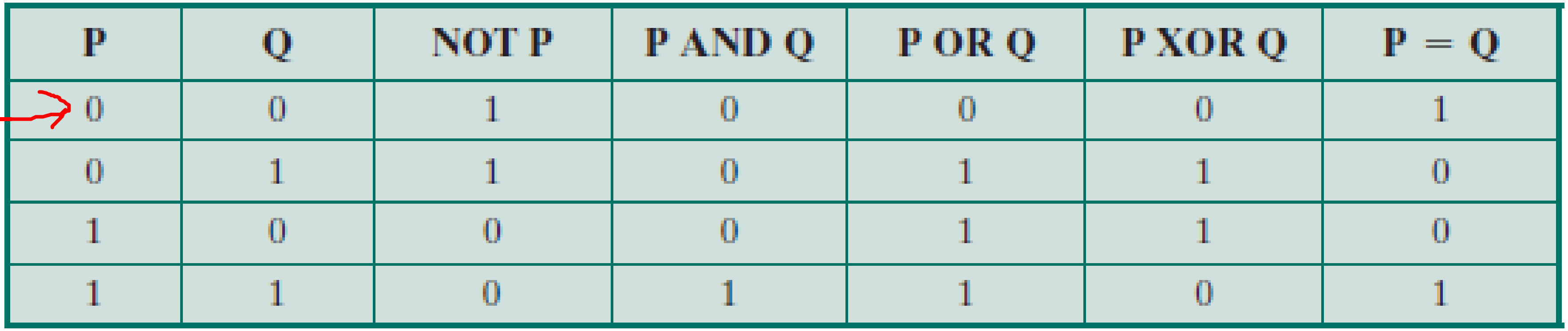
Arithmetic

- Add, Subtract, Multiply, Divide
- May include
  - Absolute value (|a|)
  - Increment (a++)
  - Decrement (a--)
  - Negate (-a)
- Signed Integer

Logical

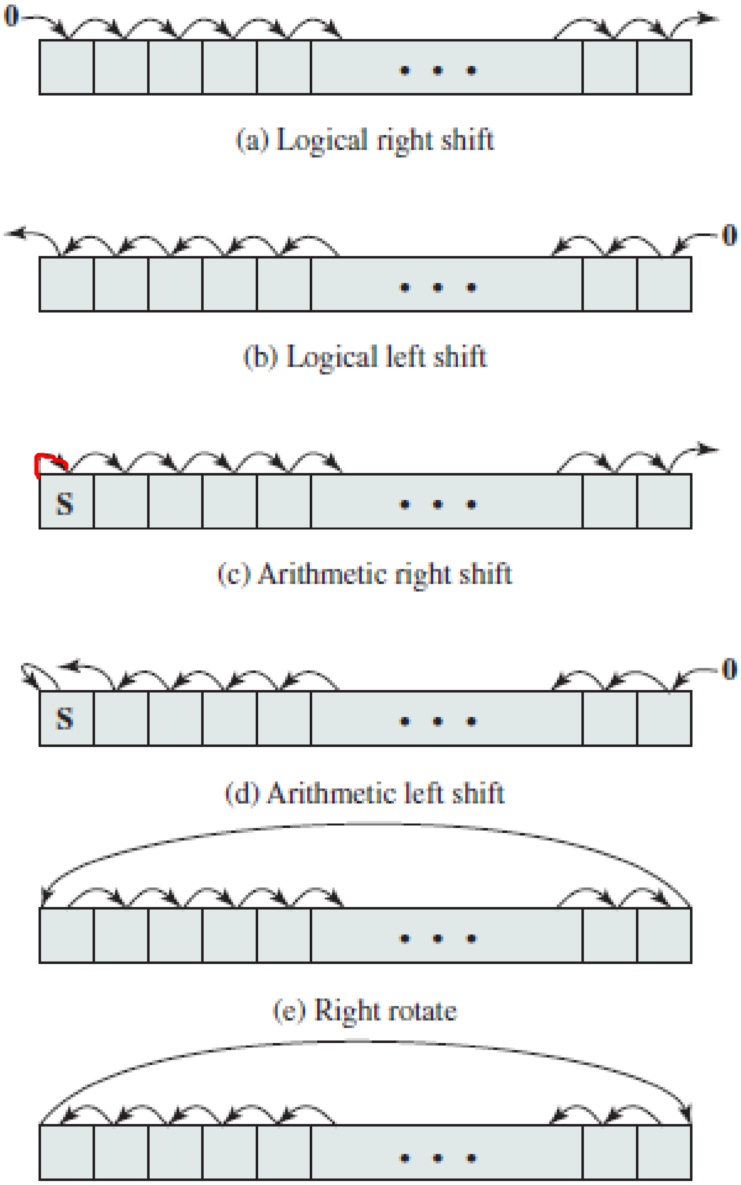
- Bitwise operations
- AND, OR, NOT

Basic Logical Operations



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

Shift and Rotate Operations



(a) Logical right shift

(b) Logical left shift

(c) Arithmetic right shift

(d) Arithmetic left shift

(e) Right rotate

(f) Left rotate

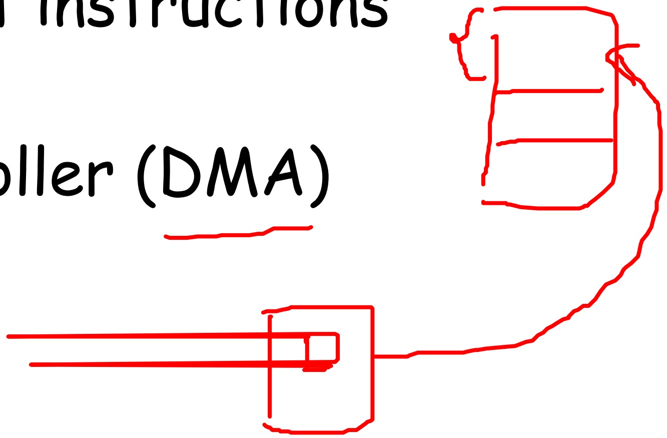
Input	Operation	Output
10101101	Logical right shift (3 bits)	10101101=> 00010101
10101101	Logical left shift (3 bits)	10101101=> **01101000**
10101101	Arithmetic right shift (3 bits)	10101101=> 11110101
10101101	Arithmetic left shift (3 bits)	10101101=> **11101000**
10101101	Right rotate (3 bits)	10101101=> **10110101**
10101101	Left rotate (3 bits)	10101101=> **01101101**

Conversion

E.g. Binary to Decimal

Input/Output

- May be specific instructions (I/O-Mapped I/O)
- May be done using data movement instructions (memory mapped)
- May be done by a separate controller (DMA)



Systems Control

- Privileged instructions
- CPU needs to be in specific state
  - User Mode
  - Kernel mode
- For operating systems use

Transfer of Control

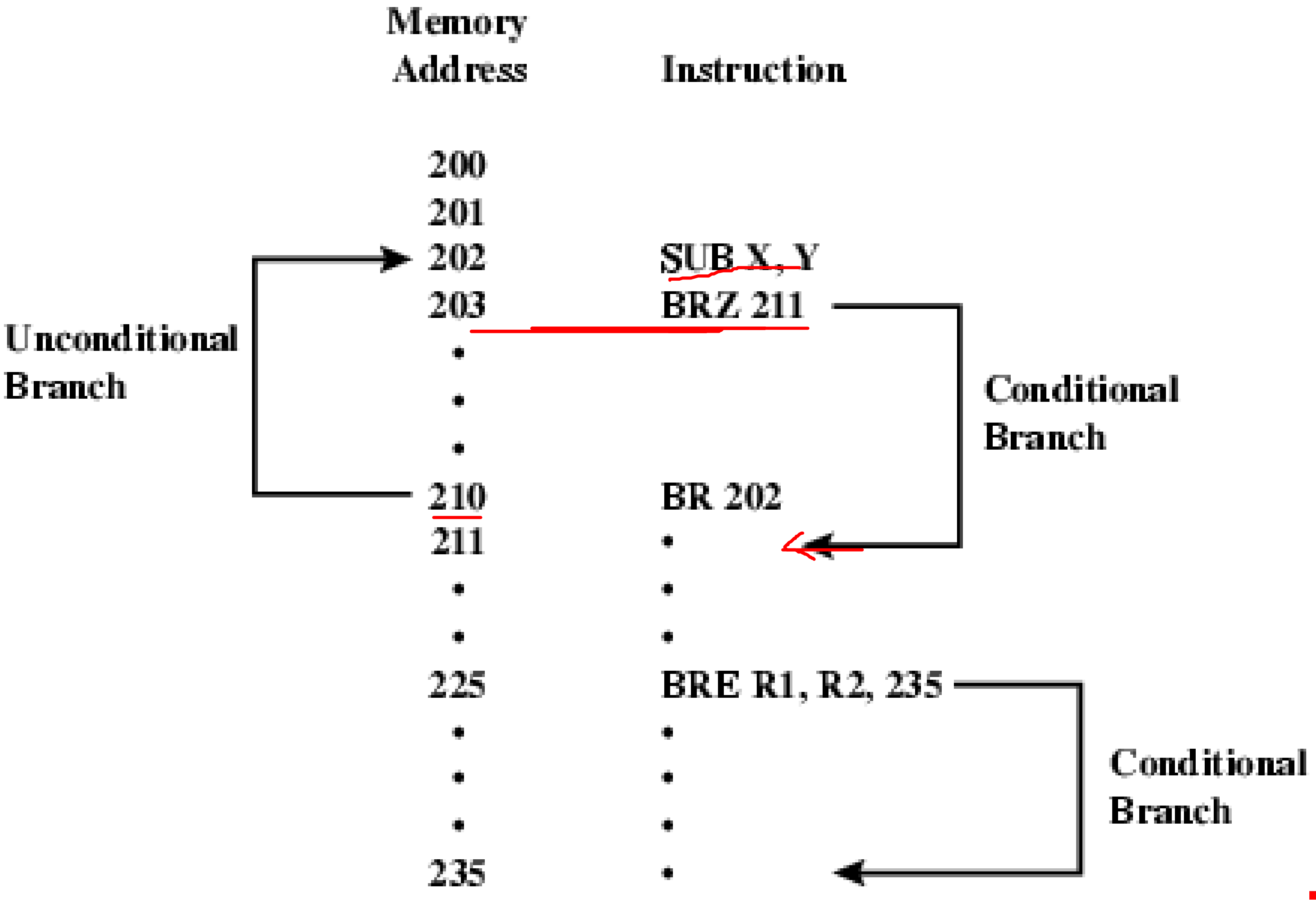
- Jump / Branch (Unconditional / Conditional)
  - e.g. jump to x if result is zero
- Skip (Unconditional / Conditional)
  - ○ skip (unconditional) : Increment to skip next instruction
  - e.g. increment and skip if zero

```
ISZ Register1
Branch xxxx
ADD A
```

$i = -1$

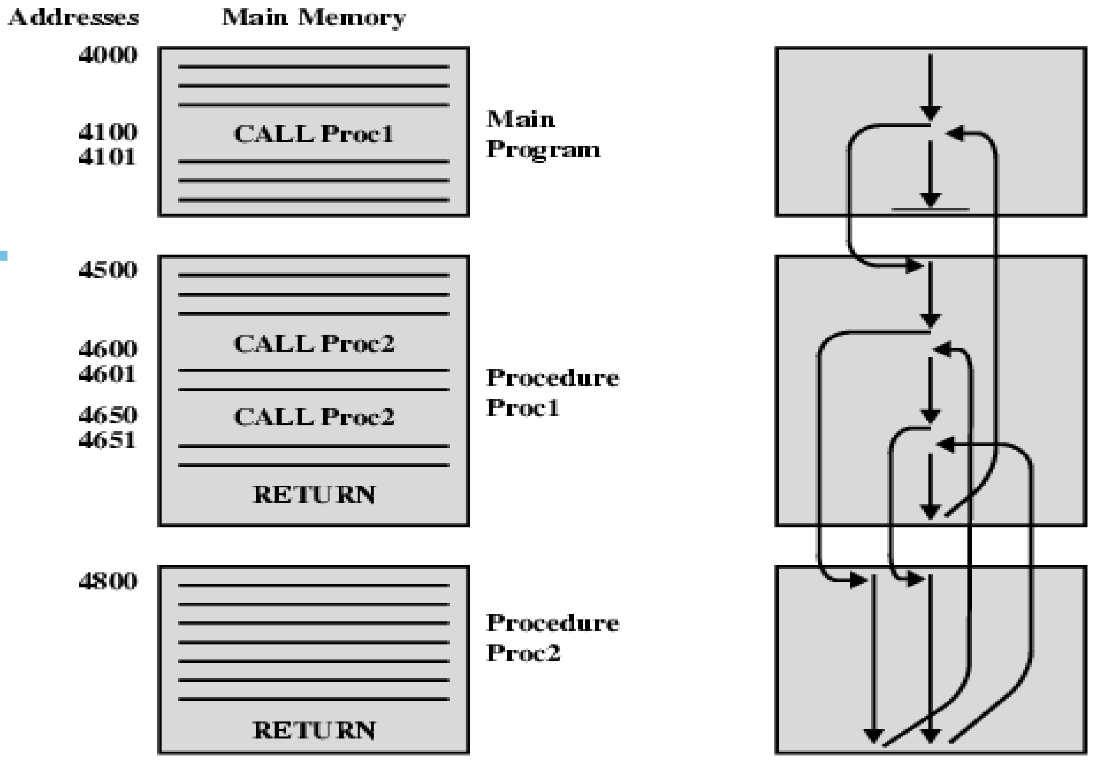
- Subroutine call
- interrupt call

Branch / Jump Instruction



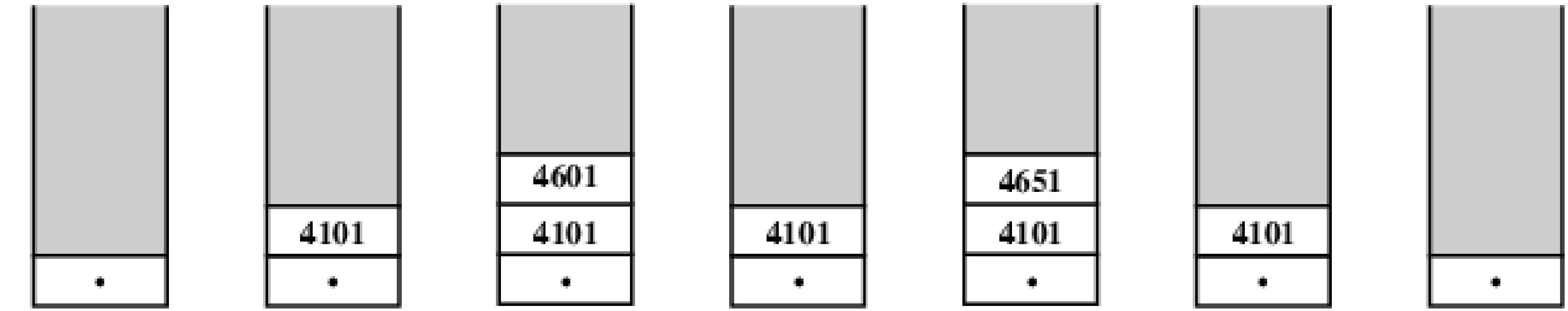
Memory Address	Instruction
200	
201	
202	SUB X, Y
203	BRZ 211
•	
•	
•	
210	BR 202
211	•
•	•
•	•
225	BRE R1, R2, 235
•	•
•	•
•	•
235	•

Use of Stack



(a) Calls and returns

(b) Execution sequence



(a) Initial stack contents

(b) After CALL Proc1

(c) Initial CALL Proc2

(d) After RETURN

(e) After CALL Proc2

(f) After RETURN

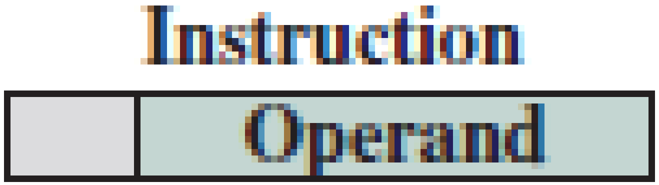
(g) After RETURN

Addressing Modes

- Addressing modes refers to the way in which the operand of an instruction is specified
- Types:
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register Indirect
  - Displacement (Indexed)
  - Stack

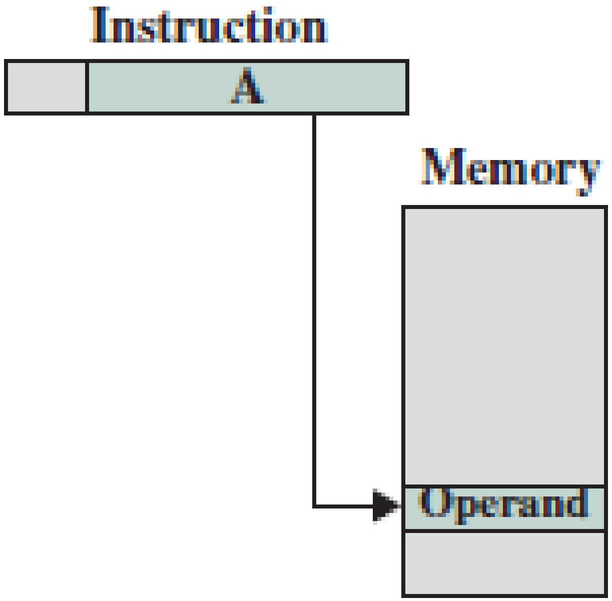
Immediate Addressing

- Operand is specified in the instruction itself
- e.g. ADD #5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range



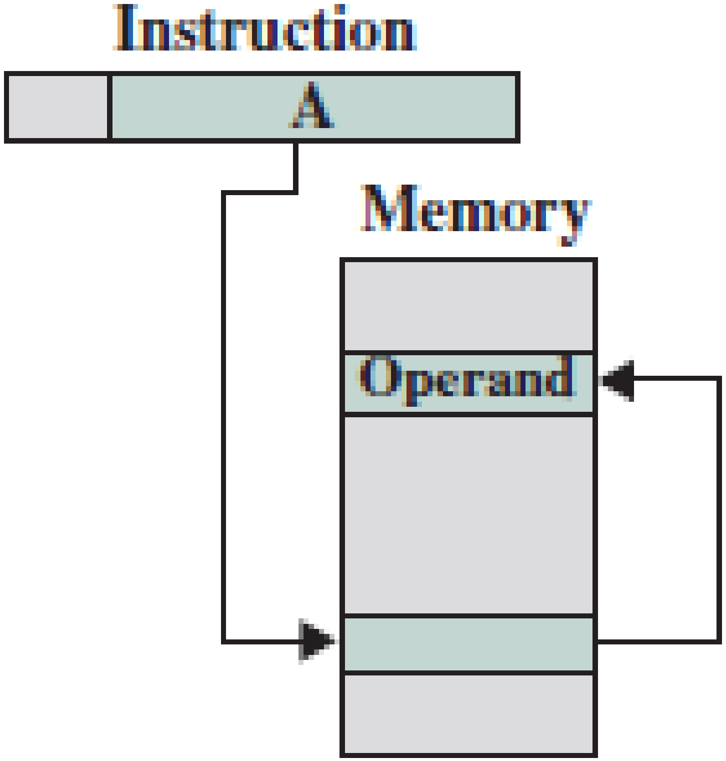
Direct Addressing

- Address of the operand is specified in the instruction
- Effective address (EA) = address field (A)
- e.g. ADD A
  - Add contents of memory cell whose address is A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space



Indirect Addressing

- Memory cell pointed to by address field of the instruction contains the address of (pointer to) the operand
- EA = (A)
  - Look in A, find address and look there for operand
- e.g. ADD (A)
  - Add contents of cell pointed to by contents of A to accumulator

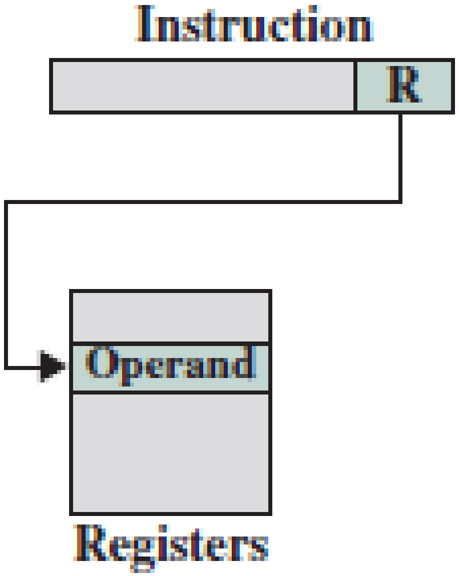


Indirect Addressing…

- Large address space
- 2^n where n = word length
- May be nested, multilevel, cascaded
  - e.g. EA = (((A)))
- Multiple memory accesses to find operand
- Slower

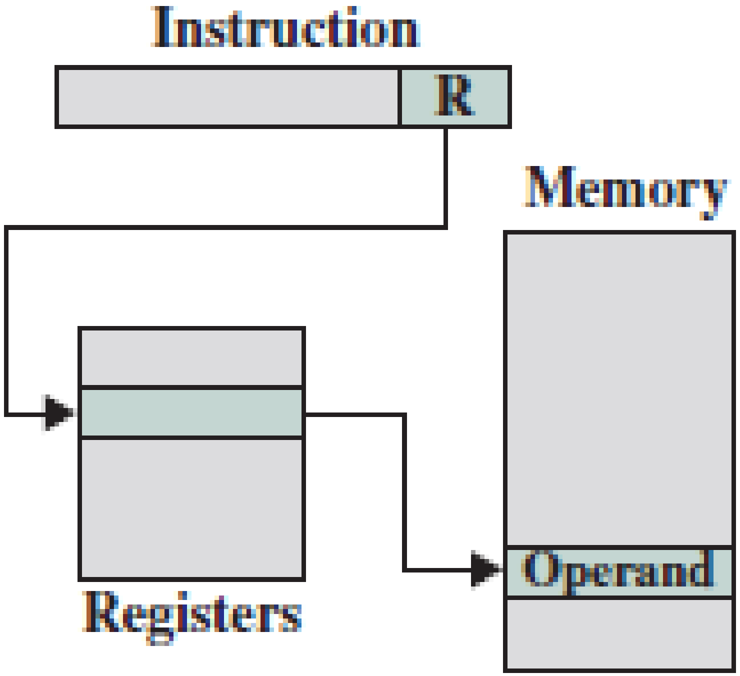
Register Addressing

- Operand is held in register named in address filed
- EA = R
- Limited number of registers
- Very small address field needed
  - Shorter instructions
  - Faster instruction fetch
- No memory access hence Very fast execution but very limited address space
- Multiple registers helps in improving performance
  - Requires good assembly programming or compiler writing
  - C programming : register int a;



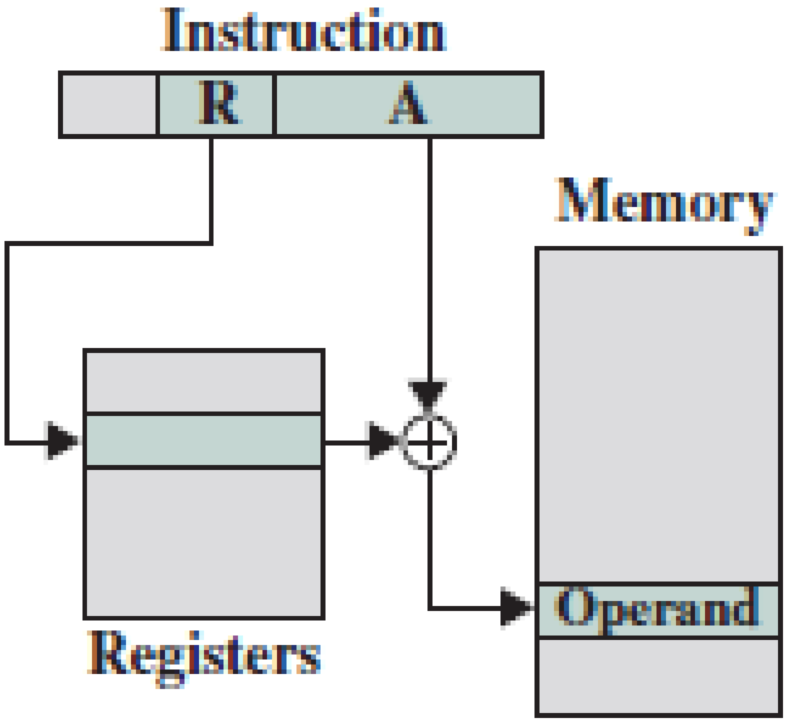
Register Indirect Addressing

- Similar to indirect addressing
- EA = (R)
- Operand is in memory cell pointed to by contents of register R
- Large address space (2^n)
- One memory access compared indirect addressing



Displacement Addressing

- EA = A + (R)
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa
- Three variants:
  - Relative addressing
  - Base register addressing
  - Indexing



Relative Addressing

- Also known as PC relative addressing
- A version of displacement addressing
- R = Program counter, PC
- EA = A + (PC)
- Relative addressing exploits the concept of locality
  - If most memory references are relatively near to the instruction being executed, then the use of relative addressing saves address bits in the instruction.

Base-Register Addressing

- The referenced register “R” contains a main memory address
- address field contains a displacement *A*
- R may be explicit or implicit
- e.g. segment registers in 80x86

Indexed Addressing

- The address field references a main memory address A
- The referenced register R contains a positive displacement from that address.
- EA = A + R
- Good for accessing arrays
  - EA = A + R
  - R++

Auto Indexing

- Auto indexing incase certain registers are devoted exclusively to indexing

$$EA = A + (R)$$

$$(R) \leftarrow (R) + 1$$

Example: LODSB : Load byte at DS:[SI] into AL. Update SI.

AL = DS:[SI]

SI is incremented or decremented based on direction flag.

D = 0 → increment SI

D = 1 → decrement SI

- Two types:
  - Postindex
  - Preindex

Post-indexing

- indexing is performed after the indirection

  EA = (A) + (R)

- Steps:
  1. The contents of the address field are used to access a memory location containing a direct address.
  2. Address is then indexed by the register value
- Use:
  - for accessing one of a number of blocks of data of a fixed format

Pre-indexing

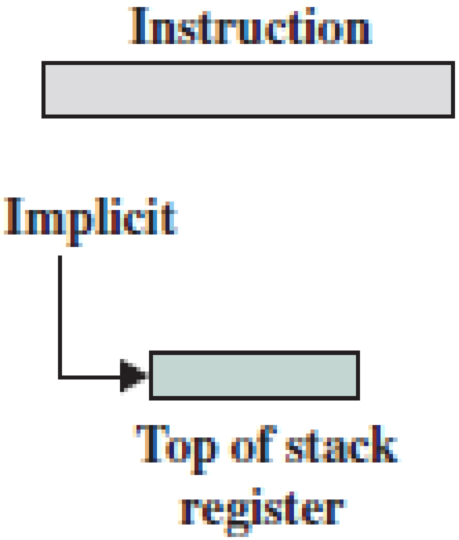
- An address is calculated as with simple indexing

  EA = (A + (R))

- Use:
  - to construct a multiway branch table

Stack Addressing

- Operand is (implicitly) on top of stack
- e.g.
  - ADD Pop top two items from stack and add, push the result on stack top



x86 Addressing Modes

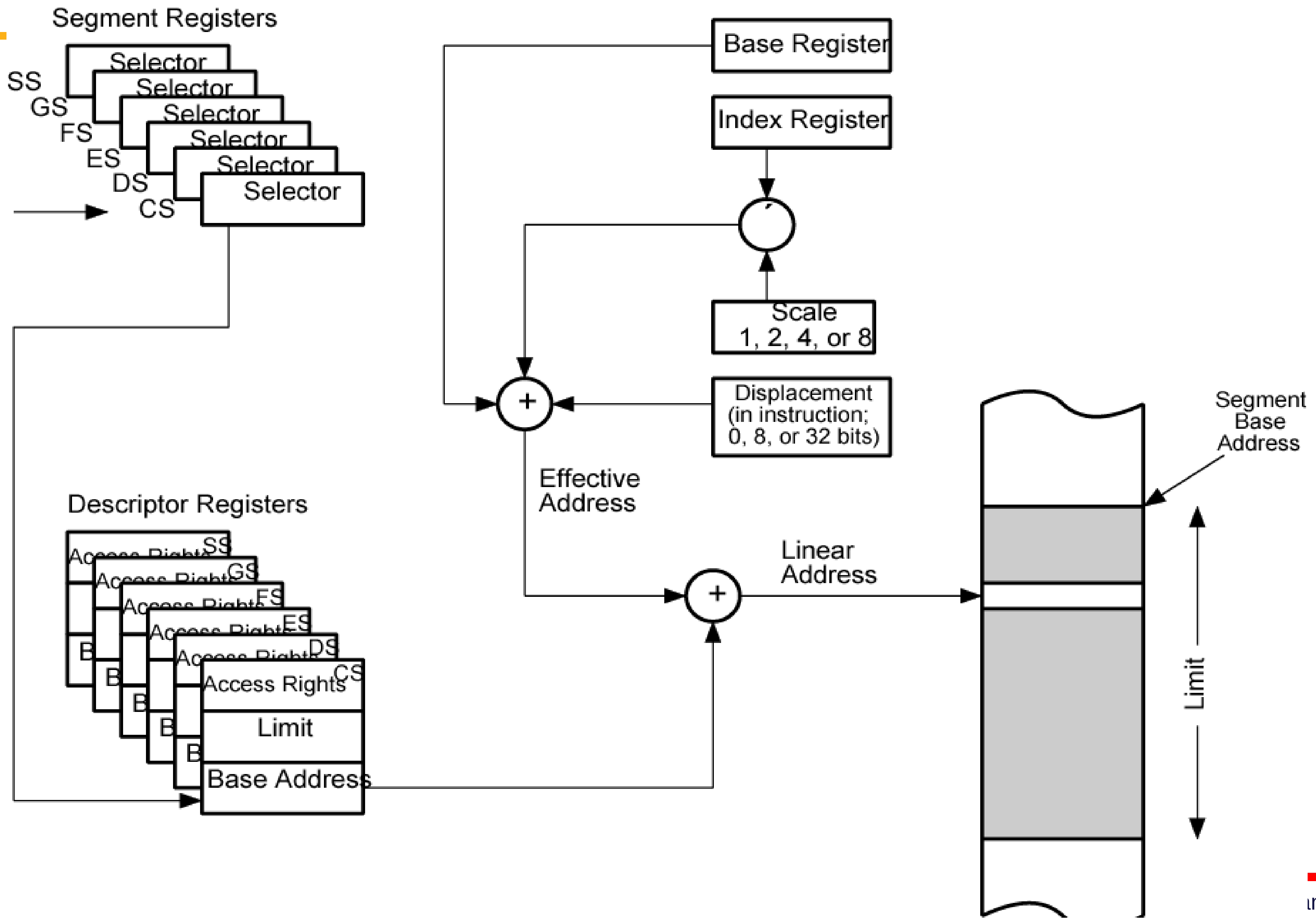
Virtual or effective address is offset into segment

- Starting address plus offset gives linear address
- This goes through page translation if paging enabled

12 addressing modes available

- Immediate
- Register operand
- Displacement
- Base
- Base with displacement
- Scaled index with displacement
- Base with index and displacement
- Base scaled index with displacement
- Relative

x86 Addressing Mode Calculation



Instruction Formats

- Layout of bits in an instruction
- Includes opcode
- Includes (implicit or explicit) operand(s)
- Usually more than one instruction format in an instruction set

Instruction Length

Affected by and affects:

- Memory size
- Memory organization
- Bus structure
- CPU complexity
- CPU speed

Trade off between powerful instruction repertoire and saving space

Allocation of Bits

- Number of addressing modes
- Number of operands
- Register versus memory
- Number of register sets
- Address range
- Address granularity