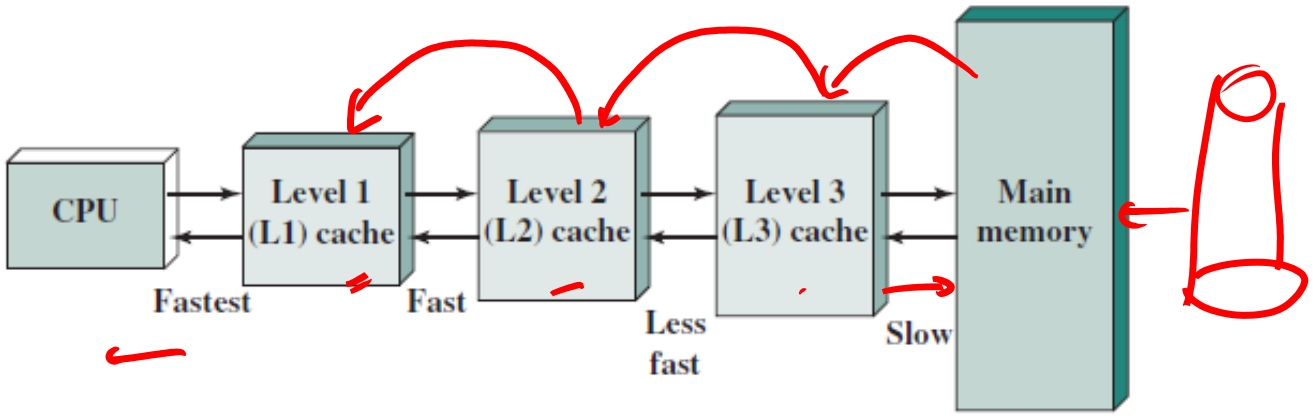
Computer Organization and Software Systems

Contact Session 6

Dr. Lucy J. Gudino

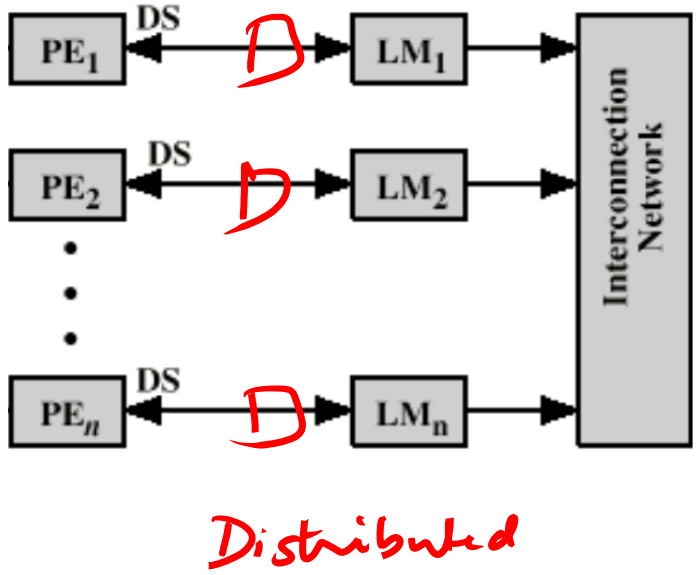
Issues with Writes

- Multiple copies of data exist:
  - L1, L2, L3, Main Memory, Disk
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)

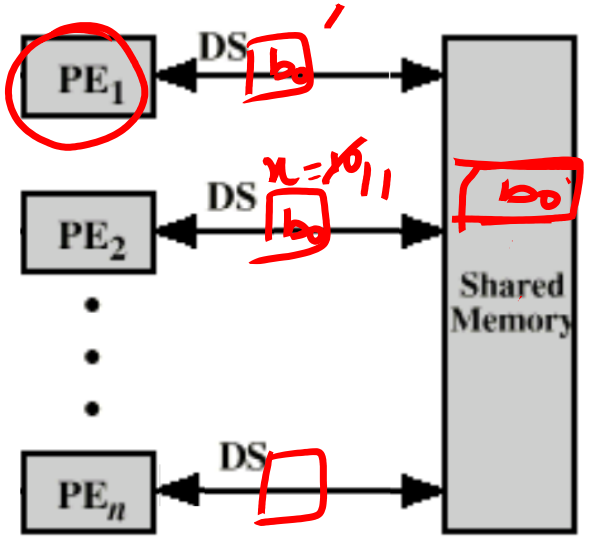


(b) Three-level cache organization

Cache and Main Memory



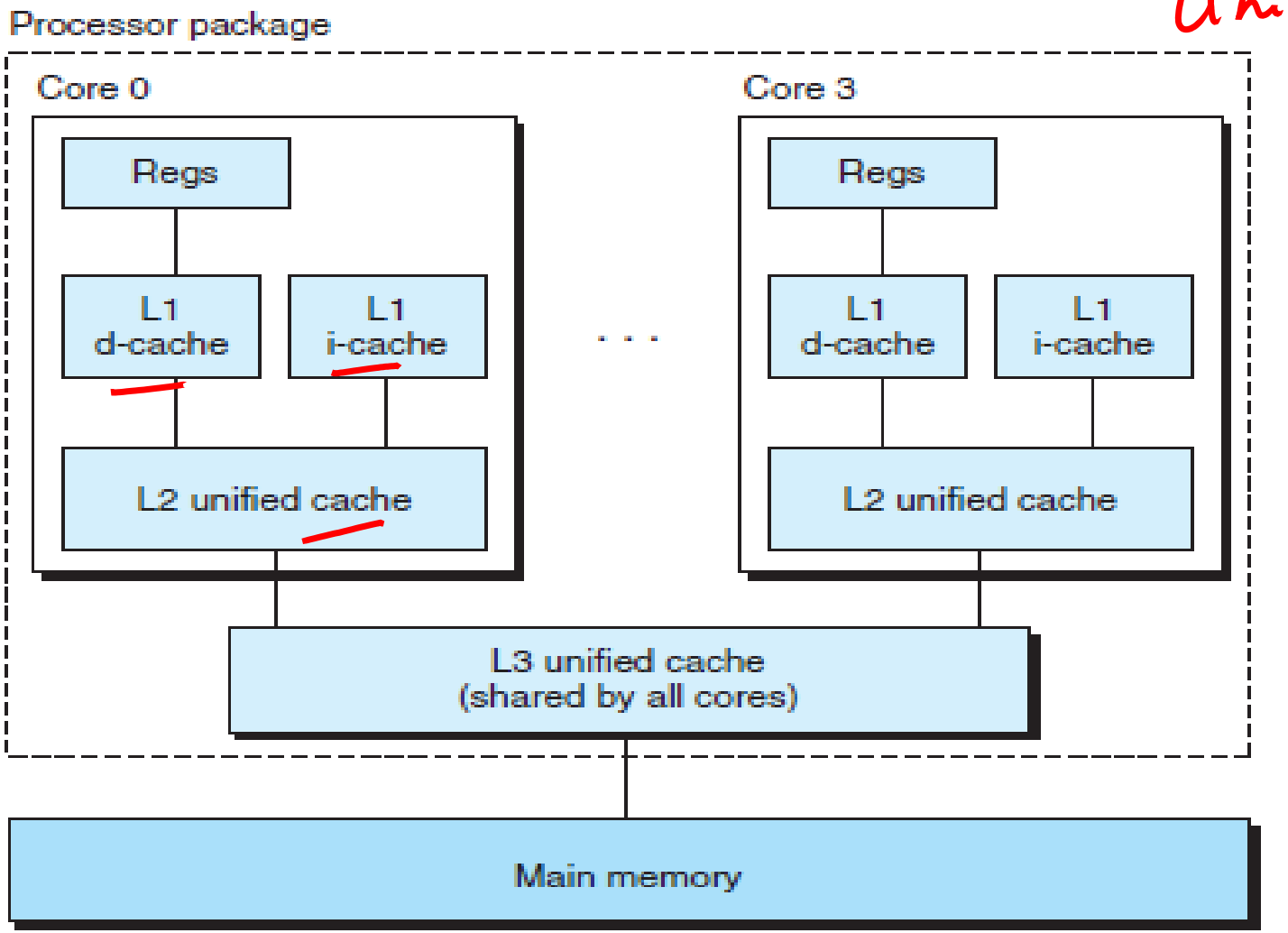
Distributed



Shared Memory

Intel Core i7 Cache Hierarchy

Instruction cache
Data cache
Unified cach



Intel Core i7 Cache Hierarchy

Cache type	Access time (cycles)	Cache size (C)	Assoc. (E)	Block size (B)	Sets (S)
L1 i-cache	4	32 KB	8	64 B	64
L1 d-cache	4	32 KB	8	64 B	64
L2 unified cache	11	256 KB	8	64 B	512
L3 unified cache	30–40	8 MB	16	64 B	8192

Characteristics of the Intel Core i7 cache hierarchy.

Performance Impact of Cache Parameters

- Associativity :
  - higher associativity ➔ more complex hardware
    - Higher Associativity ➔ Lower miss rate
    - Higher Associativity ➔ reduces average memory access time (AMAT)
- Cache Size
  - Larger the cache size ➔ Lower miss rate
  - Larger the cache size ➔ reduces average memory access time (AMAT)
- Block Size:
  - Smaller blocks do not take maximum advantage of spatial locality.

Revisiting Locality of reference

```
1    int sumvec(int v[N])
2    {
3        int i, sum = 0;
4
5        for (i = 0; i < N; i++)
6            sum += v[i];
7        return sum;
8    }
```

Does this function have good locality?

N=8								
Address	0	1	2	3	4	5	6	7
Contents	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]
Access Order	1	2	3	4	5	6	7	8

Stride k – reference pattern

Byte Addressable memory and word length is 1 byte

i	Address
0	0000
1	0001
2	0002
3	0003
4	0004
5	0005
6	0006
7	0007
8	0008
9	0009
10	000A
11	000B
12	000C

Stride 1

Address difference
Stride=----------
Word Length

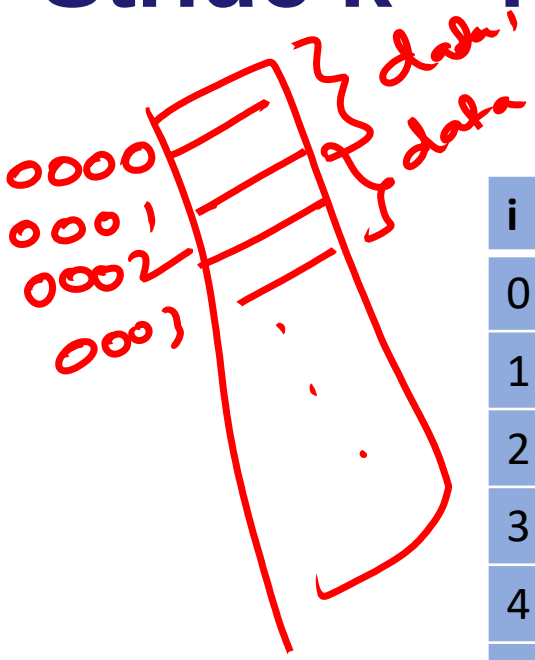
0001 - 0000 / 1 = 1

i	Address
0	0000
1	0001
2	0002
3	0003
4	0004
5	0005
6	0006
7	0007
8	0008
9	0009
10	000A
11	000B
12	000C

Stride 2

2 - 0 / 1 = 2

Stride k – reference pattern



Byte Addressable memory and word length is 2 bytes

i	Address
0	0000
1	0002
2	0004
3	0006
4	0008
5	000A
6	000C
7	000E
8	0010
9	0012
10	0014
11	0016
12	0018

Stride 1

Address difference
Stride=----------
Word Length

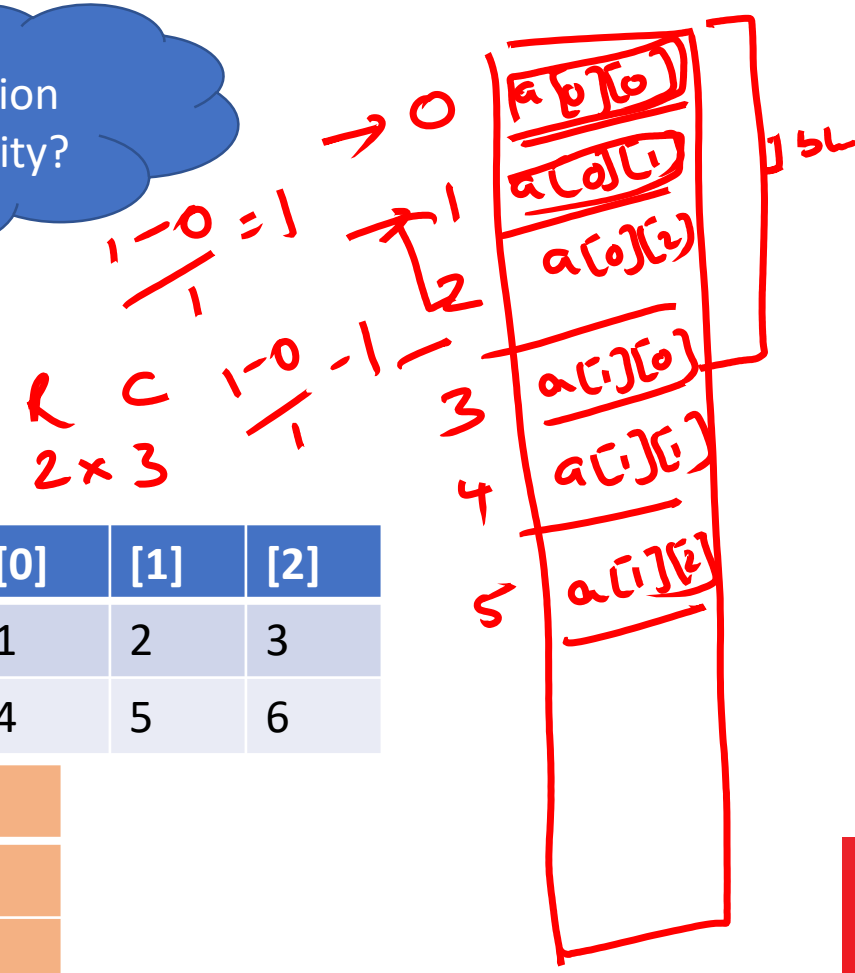
i	Address
0	0000
1	0002
2	0004
3	0006
4	0008
5	000A
6	000C
7	000E
8	0010
9	0012
10	0014
11	0016
12	0018

Stride 2

Revisiting Locality of reference

```
1   int sumarrayrows(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (i = 0; i < M; i++)
6           for (j = 0; j < N; j++)
7               sum += a[i][j];
8       return sum;
9   }
```

Does this function have good locality?



MxN	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6

M = 2, N=3						
Address(Index)	0	1	2	3	4	5
Contents	A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]
Access Order	1	2	3	4	5	6

Revisiting Locality of reference

```
1   int sumarraycols(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (j = 0; j < N; j++)
6           for (i = 0; i < M; i++)
7               sum += a[i][j];
8       return sum;
9   }
```

Does this function have good locality?

	M = 2, N=3					
Address(Index)	0	1	2	3	4	5
Contents	A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]
Access Order	1	3	5	2	4	6

NxM	[0]	[1]
[0]	1	2
[1]	3	4
[2]	5	6

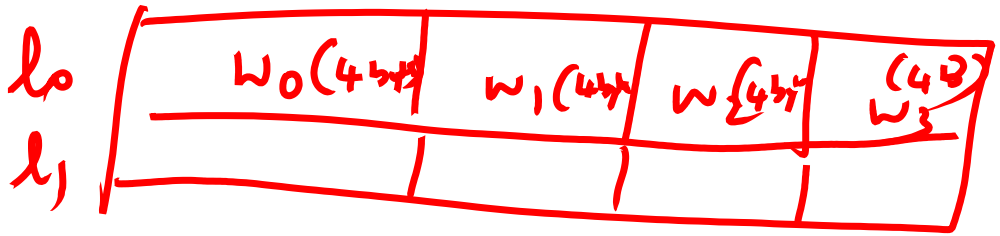
Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Example 1

```
int sumarrayrows(int a[4][4])
{
 int i, j, sum = 0;
 for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
     sum += a[i][j];
 return sum;
}
```



Capacity of each Cache line - 4 word
- 4 × 4 = 16 bytes

Assumption: byte addressable memory

- The cache has a block size of 4 words each, 2 cache lines
- Word size 4 bytes.
- C stores arrays in row-major order

Example 1(Contd..)

```
int sumarrayrows(int a[4][4])
{
 int i, j, sum = 0;
 for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
     sum += a[i][j];
 return sum;
}
```

a[i][j]	J = 0	J = 1	J = 2	J = 3
i = 0	W0 M	W1 H	W2 H	W3 H
i = 1	W4 M	W5 H	W6 H	W7 H
i = 2	W8 M	W9 H	W10 H	W11 H
I = 3	W12 M	W13 H	W14 H	W15 H

a[0][0]	W0
a[0][1]	W1
a[0][2]	W2
a[0][3]	W3
a[1][0]	W4
a[1][1]	W5
a[1][2]	W6
a[1][3]	W7
a[2][0]	W8
a[2][1]	W9
a[2][2]	W10
a[2][3]	W11
a[3][0]	W12
a[3][1]	W13
a[3][2]	W14
a[3][3]	W15

Example 2

```
int sumarraycols(int a[4][4])
{
 int i, j, sum = 0;
 for (j = 0; j < 4; j++)
   for (i = 0; i < 4; i++)
   sum += a[i][j];
 return sum;
}
```

Assumption:

- The cache has a block size of 4 words each, 2 cache lines
- Word size 4 bytes.
- C stores arrays in row-major order

Example 2(Contd..)

```
int sum_array(int a[4][4])
{
 int i, j, sum = 0;
 for (j = 0; j < 4; j++)
   for (i = 0; i < 4; i++)
   sum += a[i][j];
 return sum;
}
```

A[i][j]	J = 0	J = 1	J = 2	J = 3
i = 0				
i = 1				
i = 2				
I = 3				

L0	
L1	

W0	
W1	
W2	
W3	
W4	
W5	
W6	
W7	
W8	
W9	
W10	
W11	
W12	
W13	
W14	
W15	

Home Work – Which one is better ?

Program 1:

```
for (int i = 0; i < n; i++) {
    z[i] = x[i] – y[i];
    z[i] = z[i] * z[i];
}
```

Program 2:

```
for (int i = 0; i < n; i++) {
    z[i] = x[i] - y[i];
}
for (int i = 0; i < n; i++) {
    z[i] = z[i] * z[i];
}
```

Today's Session

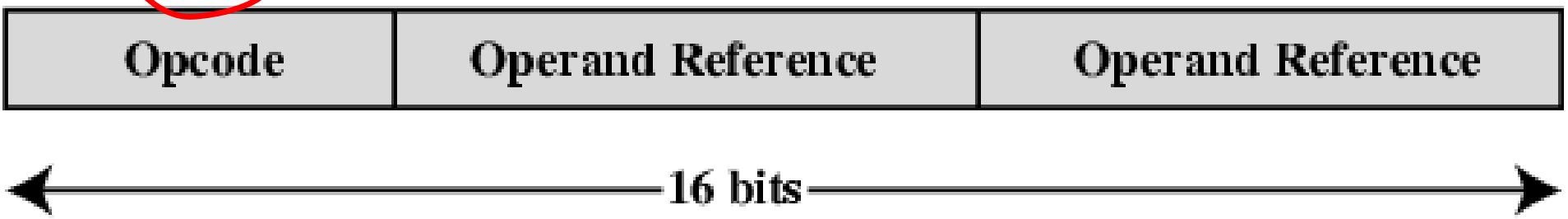
Contact Hour	List of Topic Title	Text/Ref Book/external resource

11-12	• **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



- 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 instr

Add
Sub
Mult
OR
AND
NOT
MOV
STA
IN
OUT
Branch
Subroutine
Test

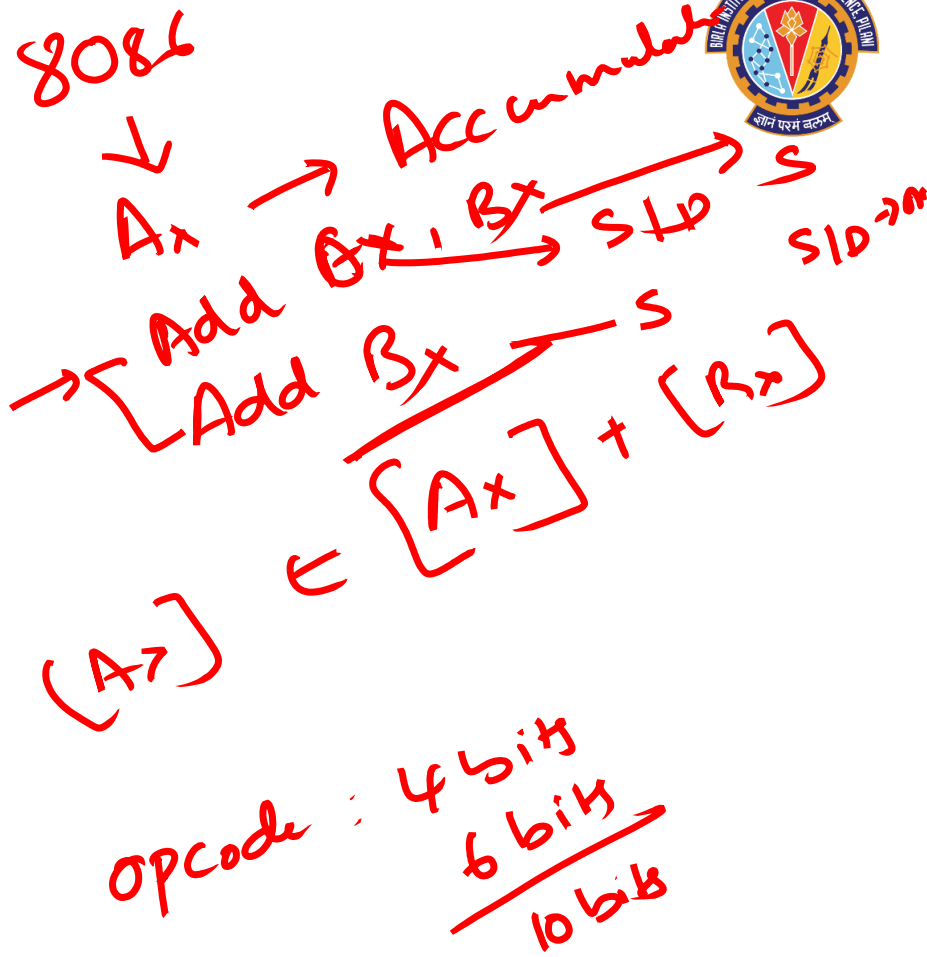
Number of Addresses (1/2)

- 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 ← A − B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y − B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction		Comment
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

How Many Addresses

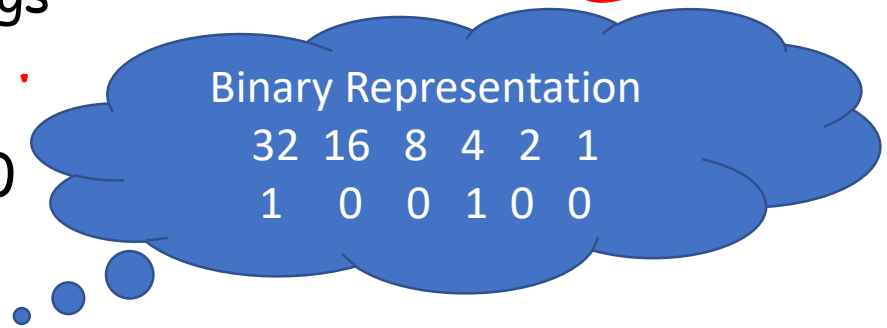
- Fewer addresses
  - More Primitive instructions, shorter length instructions
  - Less complex instructions, hence requires less complex hardware
  - More instructions per program
    - Longer programs
    - More complex programs
    - Longer execution time
- Multiple address instructions
  - Lengthy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

Instruction set Design Decisions

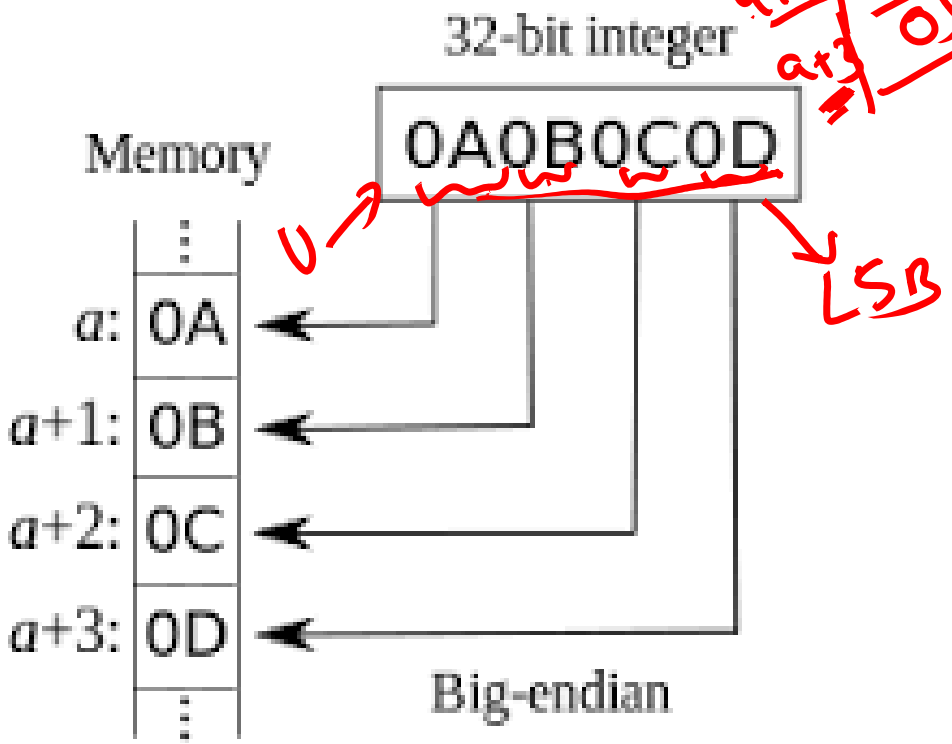
- Operation repertoire
  - How many ops?
  - What can they do?
  - How complex are they?
- Data types
- Instruction formats
  - Length of op code field
  - Number of addresses
- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers?
- Addressing modes

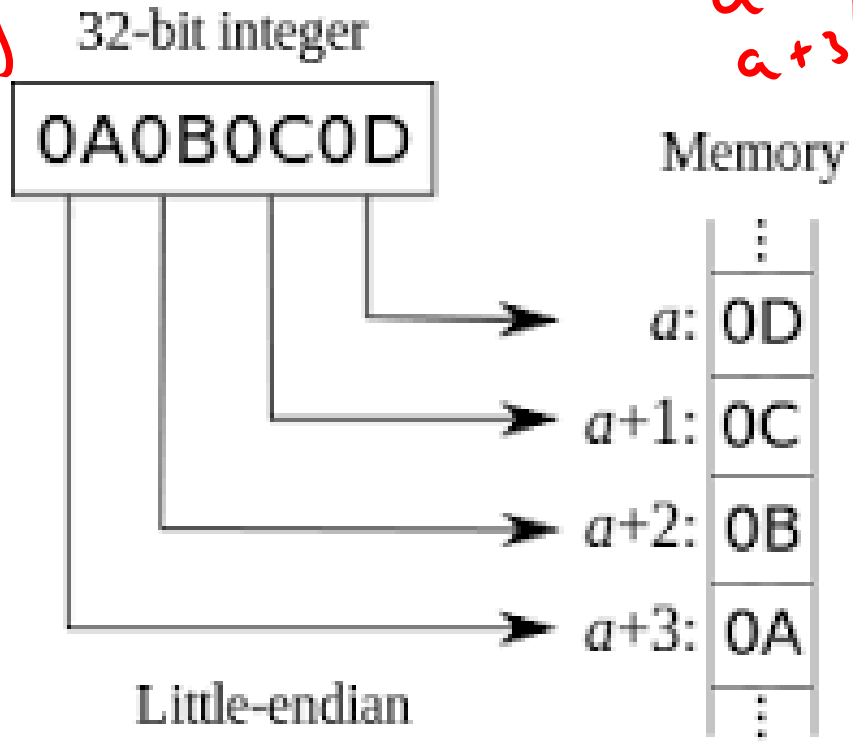
Types of Operand

- 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



Byte Ordering



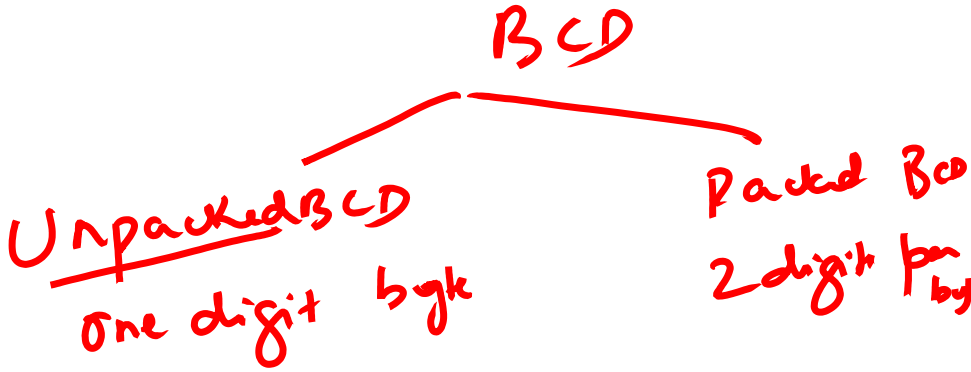


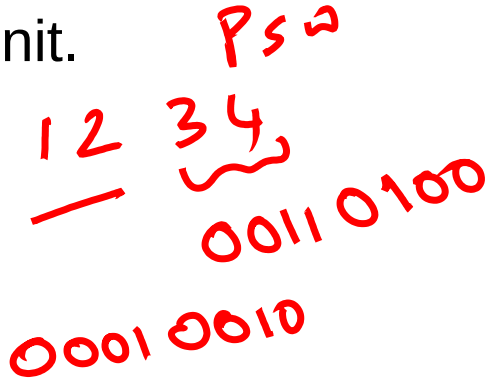
lower byte → higher address

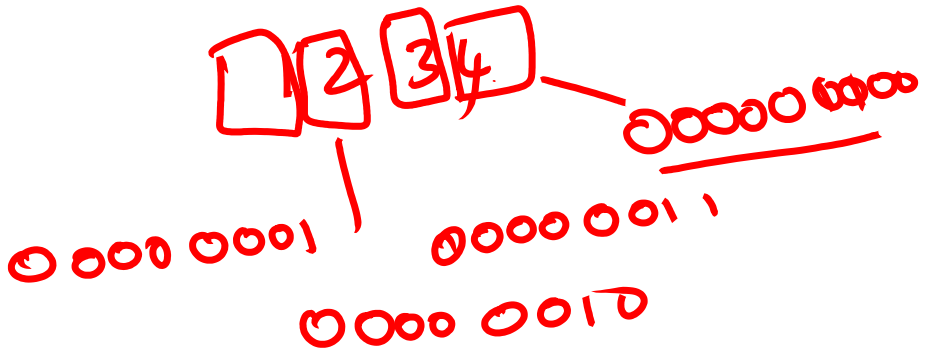
lower byte → Lower addr

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 – 16/32 bit offset within segment
- Far pointer - 16/32 bit offset outside segment
- 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

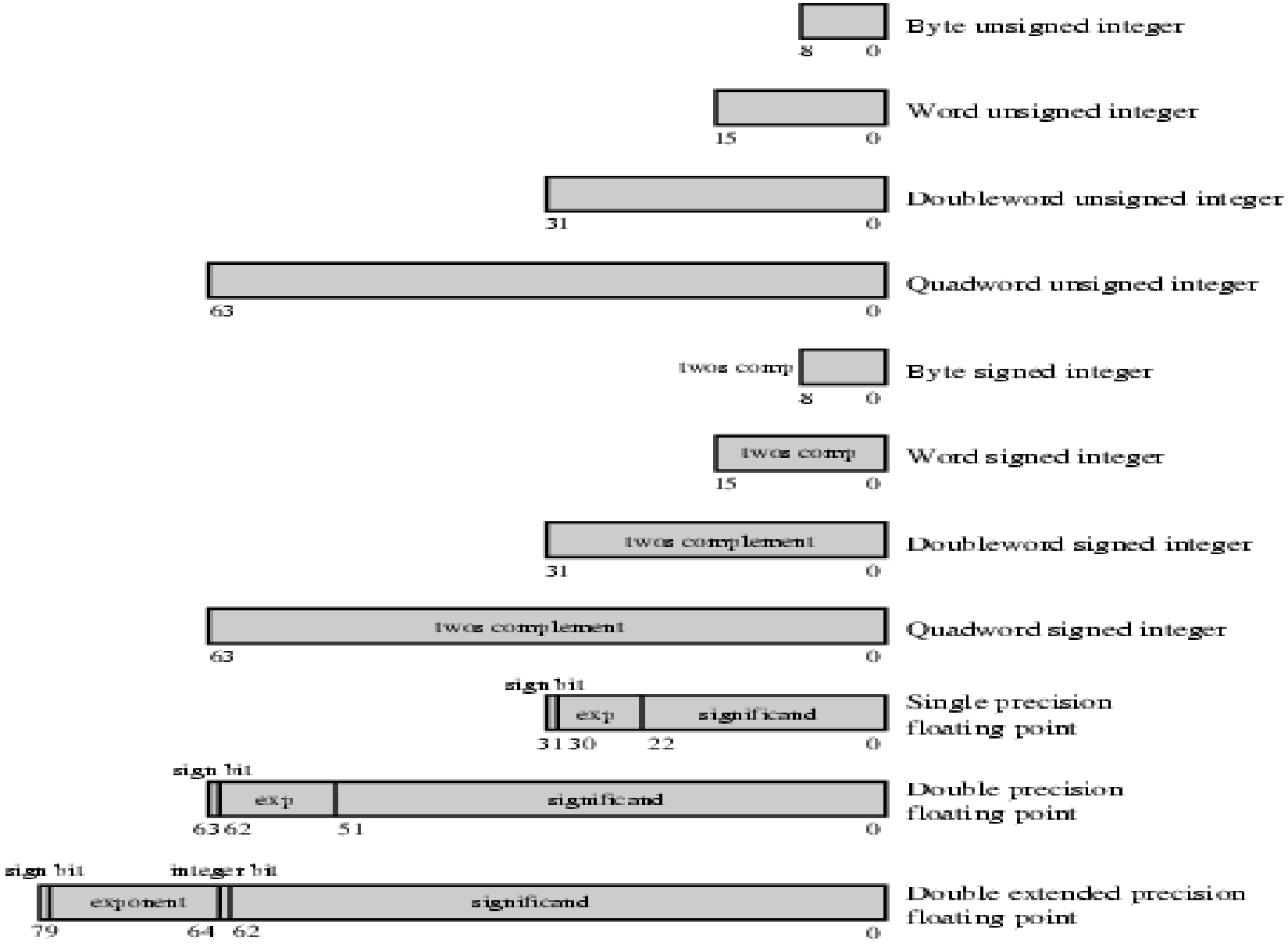






Data Type	Description
General	Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.
Integer	A signed binary value contained in a byte, word, or doubleword, using twos complement representation.
Ordinal	An unsigned integer 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



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

Arithmetic

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

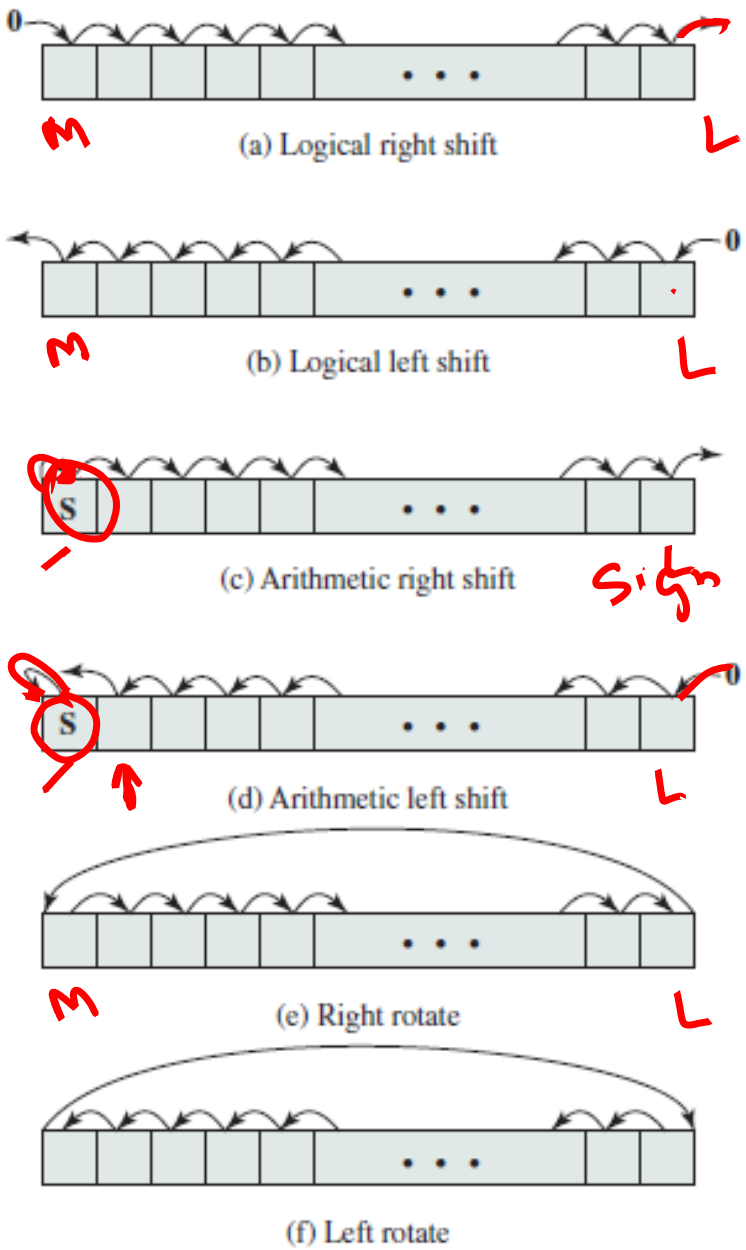
Logical

- Bitwise operations
- AND, OR, NOT

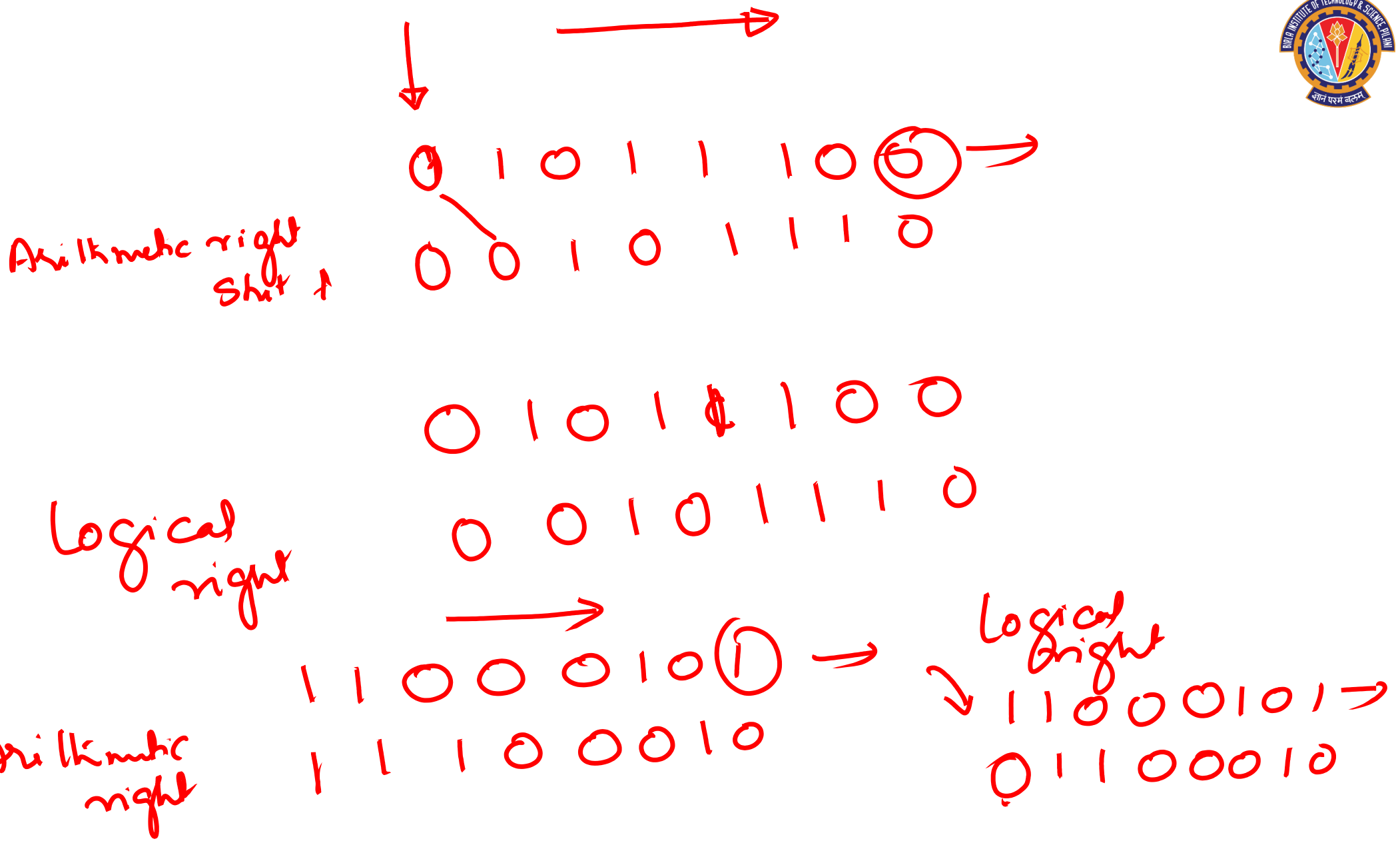
Basic Logical Operations

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

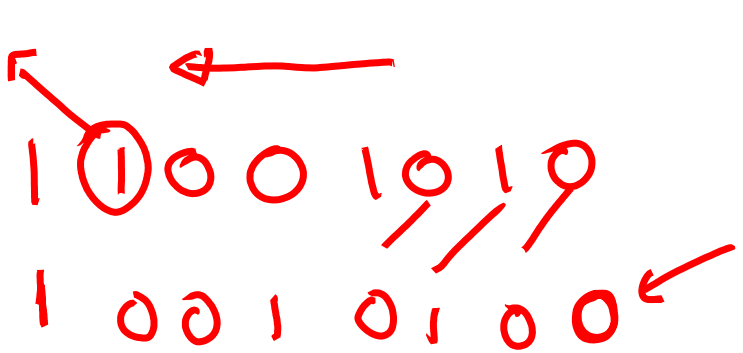
Shift and Rotate Operations



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**


Arithmetic right Shift
Logical right
Arithmetic right
Logical right

Arithmetic left shift



Logical left