IT 110

Computer Organization

From Week #1 to Week#7

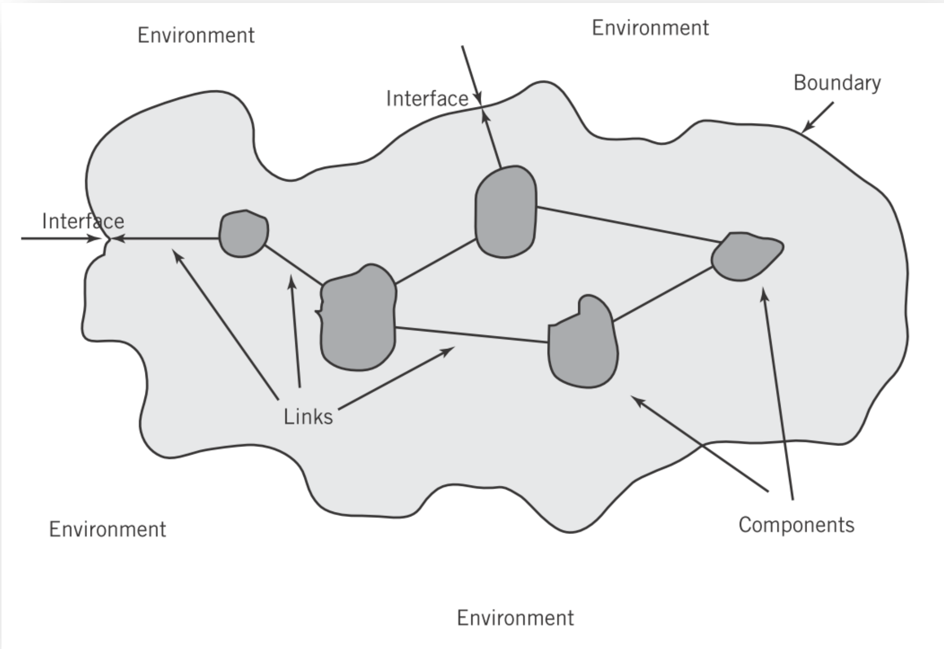
Introduction:

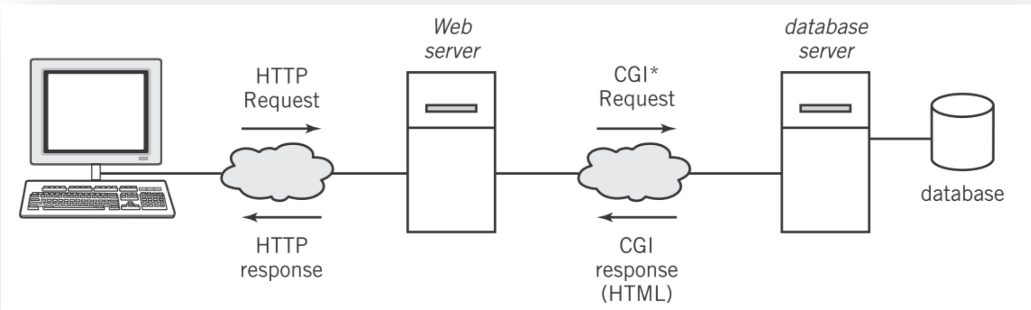
Why study computer organization?

To be a professional in any field of computing today, one should not regard the computer as just a black box that executes programs by magic. All students of computing should acquire some understanding and appreciation of a computer system's functional components, their characteristics, their performance, and their interactions… in order to structure a program so that it runs more efficiently on a real machine… [and] understand the tradeoff among various components such as CPU clock speed vs. memory size.

What is a system?

A system is a collection of components linked together and organized in such a way as to be recognizable as a single unit.





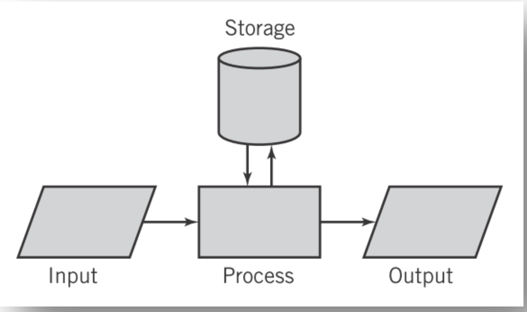
What is an architecture?

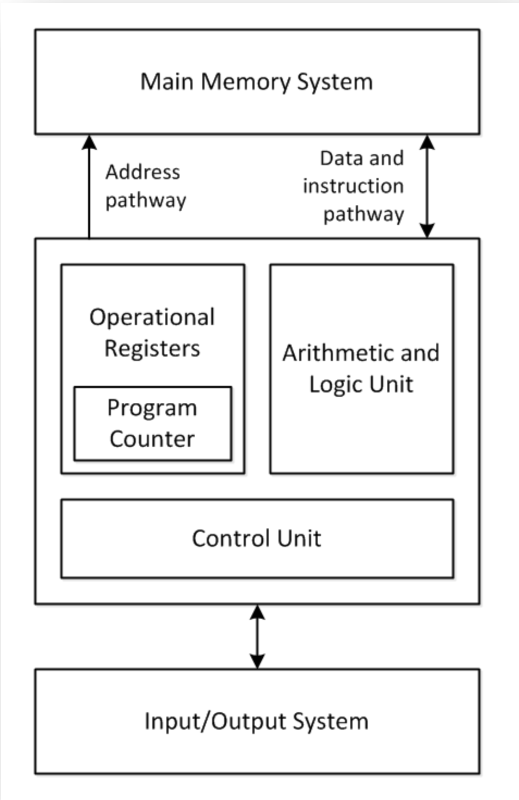
The fundamental properties, and the patterns of relationships, connections, constraints, and linkages among the components and between the system and its environment are known collectively as the architecture of the system.

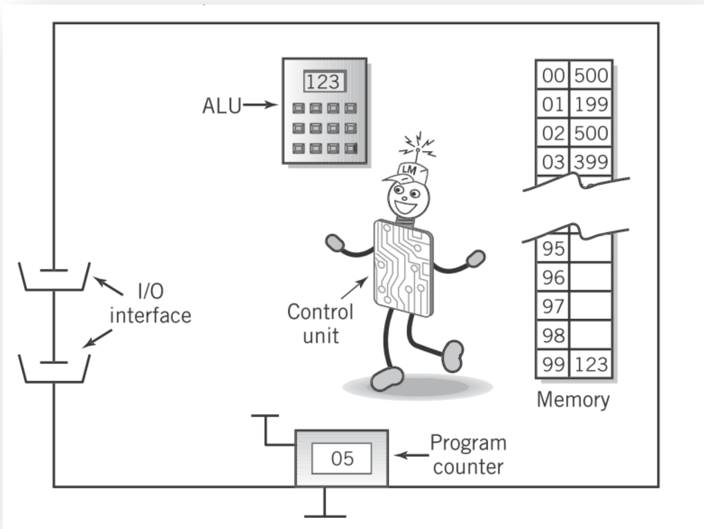
Elements of an information system architecture

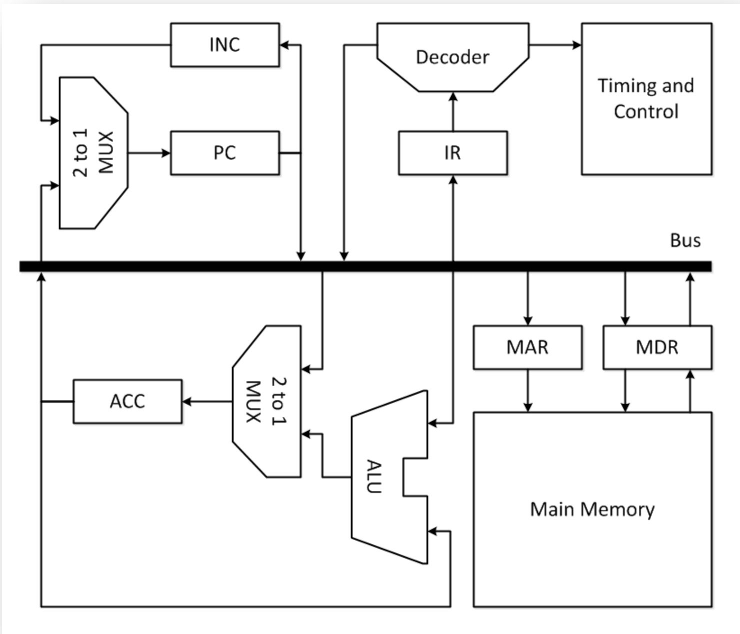
- Hardware
- Software
- Data
- People
- Networks

Models for computation









Models for computation

- Abstraction of hardware as a programming language
    - Input/output
    - Arithmetic, logic, and assignment
    - Selection, conditional branching (if-then-else, if-goto)
    - Looping, unconditional branching (while, for, repeat-until, goto)

<u>Summary</u>

- Studying computer organization is important for any technology professional.
- Information systems consist of components and links between them (hardware, software, data, people, networks).
- Information systems can be viewed at varying levels of detail and abstraction.

Counting Systems

Why do we use base 10?
Historically, it seems that the main reason that we use base 10 is that humans have ten fingers, which is as good a reason as any.

Base 10 Number System

- Ten one digit numbers (0–9)
- To expand beyond 1-digit, add a position on the left, representing the next power of ten.
- Each position represents a power of ten (a *positional number system*).
- Ex. 315,826.42
  $= 3 \times 10^5 + 1 \times 10^4 + 5 \times 10^3 + 8 \times 10^2 + 2 \times 10^1 + 6 \times 10^0 + 4 \times 10^{-1} + 2 \times 10^{-2}$
- Operations can take place at each position (e.g. adding two numbers by column with carry).

Base 2 Number System

- Two one digit numbers (0–1).
- To expand beyond 1-digit, add a position on the left, representing the next power of two.
- Leading zeros: Are insignificant, but often written to indicate the number of *bits* in a quantity.
  Ex. 0110 = 110.
  $0110 = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$

Converting to and from binary

- Base 10 to base 2 conversion: repeated division with remainders

  Ex.: Convert 92_{10} to binary.

  92_{10}

  $92 \div 2 = 46$ 0
  $46 \div 2 = 23$ 0
  $23 \div 2 = 11$ 1
  $11 \div 2 = 05$ 1
  $05 \div 2 = 02$ 1
  $02 \div 2 = 01$ 0
  $01 \div 2 = 00$ 1

  $92_{10} = 1011100_2$

Converting to and from binary

- Base 2 to base 10 conversion: repeated multiplication and addition

  Ex.: Convert 1011100_2 to decimal.

  $1011100_2 = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
  $= 64+0+16+8+4+0+0 = 92_{10}$

Base 8 and Base 16 Number Systems

Binary is cumbersome

- Long strings of 1's and 0's are hard to read. Group into sets or 3 (octal) or 4 (hexadecimal).

Base 10	Base 2	Base 8	Base 16
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Ex.: Rewrite 110111100101_2 as octal and hexadecimal.

Group by 3: 110 111 100 101 → 6745_8

Group by 4: 1101 1110 0101 → $DE5_{16}$

Summary

- Base 10 number systems are not universal.
- Computers employ a base 2 (binary) system.
- Number systems use positions representing powers of the base.
- Converting from base 10 to another base involves division by the base and examining the remainders.
- Converting from another base to base 10 involves multiplying by a power of the base and summing.
- Octal and Hexadecimal are convenience groupings of binary numbers.

Signed Integer Representations

Positional number arithmetic

- Addition, subtraction, multiplication, and division is done by column in base 10.
- The same general process can be followed in binary.

Ex:

$$\begin{array}{r} 101101001 \\ \underline{-001011000} \\ 100010001 \end{array}$$

$$\begin{array}{r} 1111 \\ 101101001 \\ \underline{+001011000} \\ 11000001 \end{array}$$

- Negative number representations—all use fixed width fields (i.e., a 16- or 32-bit integer)

  Simple binary—assumes all numbers are positive

  $66_{10} = 01000010_2$

  $194_{10} = 11000010_2$

- Negative number representations—all use fixed width fields (i.e., a 16- or 32-bit integer)
- We have three method to represent a negative number:
  1. Signed magnitude.
  2. 1's complement.
  3. 2's complement.
  - **Signed magnitude**—use most significant bit to represent the sign. 0 is positive, 1 is negative.

    $66_{10} = 01000010_2$

    $-66_{10} = 11000010_2$

    Disadvantage: Can't include the sign bit in addition
  - **1's complement** —All 0's become 1's and all 1's become 0's

    Ex. Find the representation of $(-17)_{10}$ using 1's complement.

    $17 \div 2 = 8 \quad 1$

    $08 \div 2 = 4 \quad 0$

    $04 \div 2 = 2 \quad 0$

    $02 \div 2 = 1 \quad 0$

    $01 \div 2 = 0 \quad 1$

    $(17)_{10} \Rightarrow (0001\ 0001)_2$

    $(-17)_{10} = (1110\ 1110)_2$

Ex. Subtract $(1)_{10}$ from $(5)_{10}$ using 1's complement.
$5-1 \Rightarrow 5+(-1)$
$(1)_{10} \Rightarrow (0001)_2, (-1)_{10} \Rightarrow (1110)_2, (5)_{10} = (0101)_2$

```
   1
   0101
 +1110
(1)0011
 +      1
   0100
```

over flow (1)→+
$(0100)_2 = (+4)_{10}$

- **2's complement** —Change the bit after the first 1 from the right hand side.
  0101⇒1011,00010100⇒11101100
  Ex. Subtract $(1)_{10}$ from $(5)_{10}$ using 2's complement.
  $5-1 \Rightarrow 5+(-1)$
  $(1)_{10} \Rightarrow (0001)_2, (-1)_{10} \Rightarrow (1111)_2, (5)_{10} = (0101)_2$

```
   0101
 +1111
(1)0100
```

  over flow (1)
  $(0100)_2 = (4)_{10}$

  Ex. $5-6 \Rightarrow 5+(-6)$
  $(6)_{10} \Rightarrow (0110)_2, (-6)_{10} \Rightarrow (1010)_2, (5)_{10} = (0101)_2$

```
   0101
 +1010
   1111
```

  (1) on the left = -
  $(0001)_2 = (-1)_{10}$

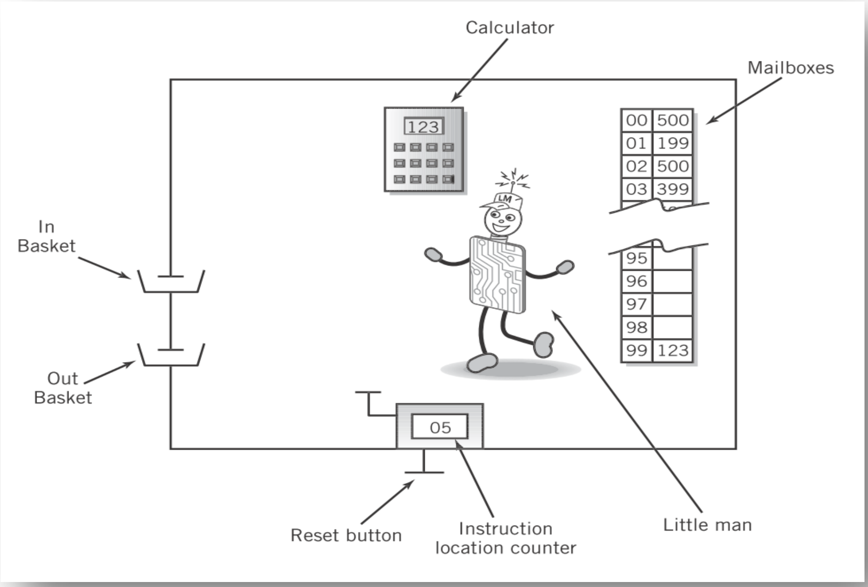
Summary

- Computers use positional arithmetic.
- Choices in representing negative numbers include signed magnitude, binary coded decimal, and two's complement.
- Two's complement solves several problems:
  - No "negative zero" representation
  - Subtraction becomes addition of a negative number, simplifying CPU hardware.

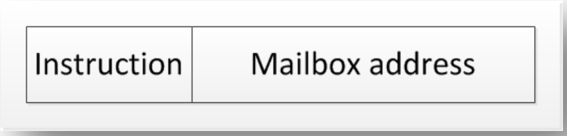
Little Man Computer and Instruction Cycle

Little Man Computer

- Developed by Dr. Stuart Madnick of MIT in 1965
- A model for how computers execute programs



- The Little Man executes *instructions* that are stored in memory. Like everything else, these are encoded.



Mnemonic	Code	Description
LDA	5XX	Load calculator with data from box XX
STO	3XX	Store calculator value in box XX
ADD	1XX	Add data in box XX to calculator
SUB	2XX	Subtract data in box XX from calculator
IN	901	Get input from inbox, put in calculator
OUT	902	Write calculator total to outbox
HLT	000	Stop executing
BRZ	7XX	Zero? Next instruction is in box XX
BRP	8XX	Positive? Next instruction is in box XX
BR	6XX	Next instruction is in box XX
DAT		Data storage reserved

Execute cycle

- Fetch
  - Little Man looks at the instruction counter.
  - Little Man retrieves the instruction from the mailbox indicated by the counter.
  - Little Man increments the instruction counter.
- Execute
  - Little Man performs the instruction retrieved from the previous step.

Example program: Read two numbers, add them, output the result

Box	Assembly	Code
01	IN	901
02	STO 07	307
03	IN	901
04	ADD 07	107
05	OUT	902
06	HLT	000
07	DAT	000

Summary

- LMC is a model for computation based on real principles.
- Instructions consist of an operation and, optionally, an operand on which to act.
- Fetch/execute cycle (simple):
  - Retrieve instruction indicated by PC.
  - Increment program counter.
  - Execute instruction.
- Operations of load, store, add, subtract, input, output, and branching are the simplest possible instruction set.

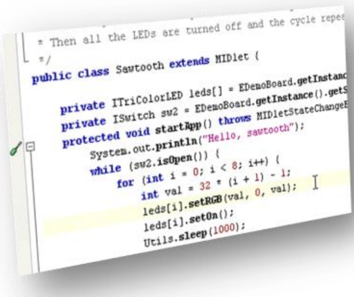
<u>**Assembly Language**</u>

Generations of programming languages

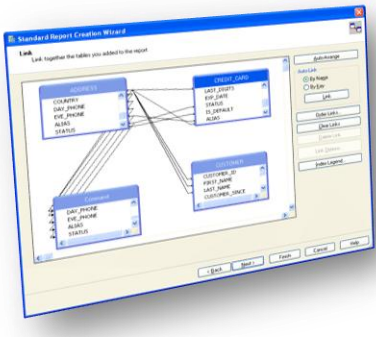
- First generation: programmed directly in binary using wires or switches.

- Second generation: assembly language. Human readable, converted directly to machine code.

- Third generation: high-level languages, while loops, if-then-else, structured. Most programming today, including object-oriented.



- Fourth generation: 1990s natural languages, non-procedural, report generation. Use programs to generate other programs. Limited use today.



- Key idea: Regardless of the language of writing, computers only process machine code.
- All non-machine code goes through a translation phase into machine code.
  - Code generators
  - Compilers
  - Assemblers

Language translation process

- High level languages use comparison constructs, loops, variables, etc.
- Machine code is binary, directly executed by CPU.

```
var i = 0;
var j = 1;
var k = 0;
while (k < 10) {
  var fib = i + j;
  i = j;
  j = fib;
  print(i);
  k = k + 1
}
```

- Convert high level language to if/goto.

```
      i = 0
      j = 1
      k = 0
loop: if (k - 10 == 0) goto done
      fib = i + j
      i = j
      j = fib
      print i
      k = k + 1
      goto loop
done: halt
```

- Convert if/goto to assembly (LMC here).

```
loop: LDA  k    ; if (k - 10 == 0) goto done
      SUB  ten  ;
      BRZ  done ;
      LDA  i    ; fib = i + j
      ADD  j    ;
      STO  fib  ;
      LDA  j    ; i = j
      STO  i    ;
      LDA  fib  ; j = fib
      STO  j    ;
      LDA  i    ; print i
      OUT       ;
      LDA  k    ; k = k + 1
      ADD  one  ;
      STO  k    ;
      BR   loop ; goto loop
done: HLT       ; halt
```

```
; data section
j:    DAT 0     ; i = 0
i:    DAT 1     ; j = 1
k:    DAT 0     ; k = 0
fib:  DAT 0     ;
ten:  DAT 10    ;
one:  DAT 1     ;
```

- Assemble the instructions to machine code.

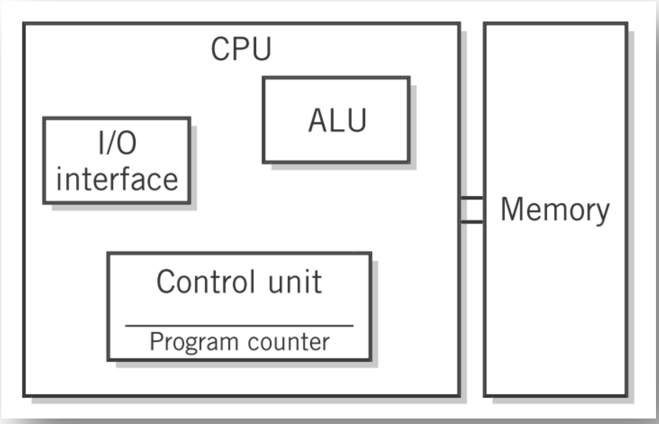
Box	Code	Assembler
01	520	LDA k
02	222	SUB ten
03	717	BRZ done
04	519	LDA i
05	119	ADD j
06	321	STO fib
07	519	LDA j
08	319	STO i
09	521	LDA fib
10	319	STO j
11	519	LDA i
12	902	OUT
13	520	LDA k
14	123	ADD one
15	320	STO k
16	601	BR loop
17	000	HLT

Summary

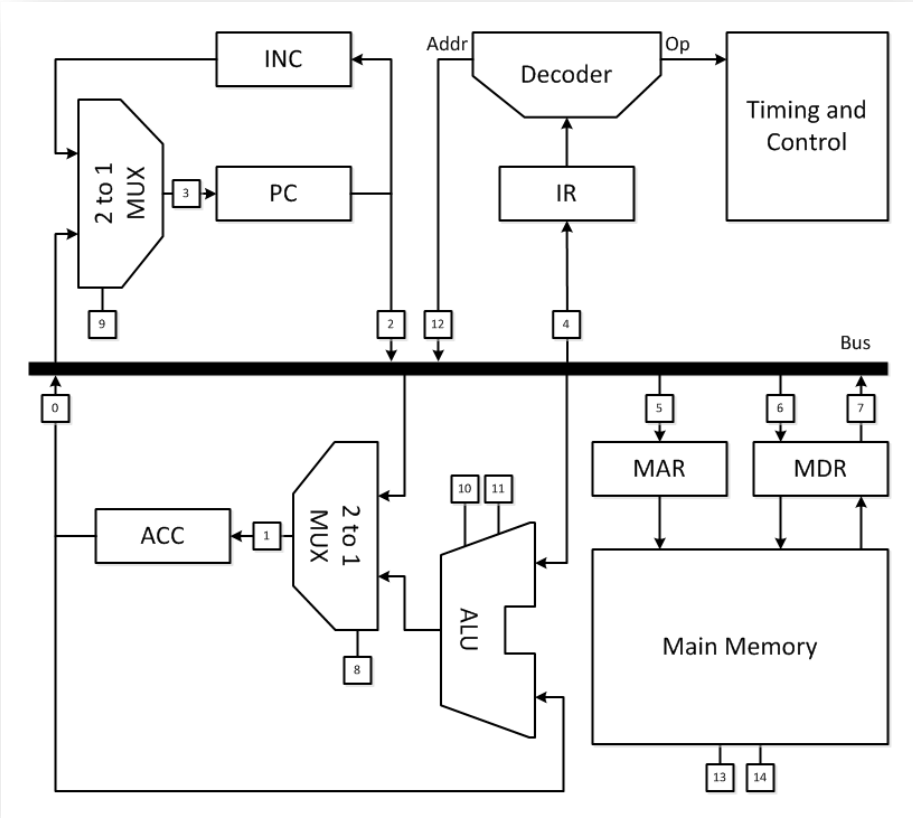
- High level languages are convenient to read and write for humans.
- Computers execute only binary machine code.
- Conversion between the two is required.
    - Compilers translate high level languages to machine code.
    - Assemblers translate assembly language into machine code.
- Use if/goto pseudo-code as an intermediate language between high level and assembler.

Fetch/Execute Cycle

Von Neumann Architecture

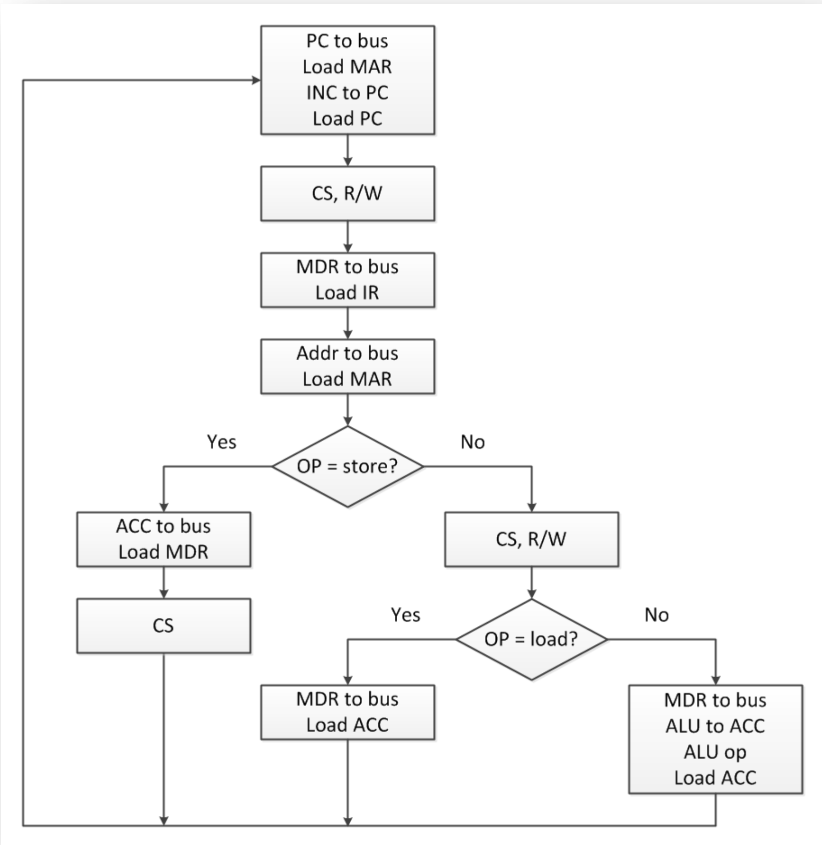


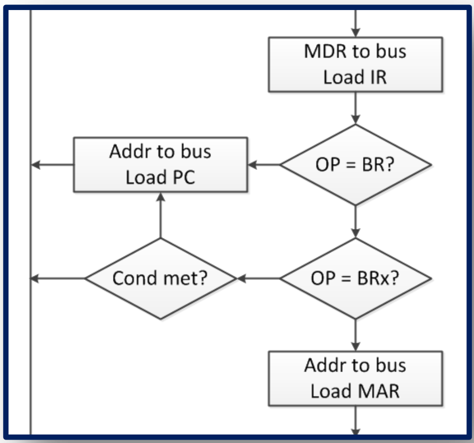
Detailed Architecture



Number	Operation	Number	Operation
0	ACC→bus	8	ALU→ACC
1	Load ACC	9	INC→PC
2	PC→bus	10	ALU operation
3	Load PC	11	ALU operation
4	Load IR	12	Addr→bus
5	Load MAR	13	CS
6	MDR→bus	14	R/W
7	Load MDR		

Detailed Fetch/Execute Cycle





Summary

- The fetch/execute cycle consists of many steps and is implemented in the control unit as microcode.
- Control signals select operations, control access to the bus, and allow data to flow from component to component.
- Adding new instructions means modifying the microprogram in the control unit.

Instruction Set Architectures

ISA determines instruction formats

- The LMC is a one-address architecture (an accumulator-based machine).

Instruction	Mailbox address

- There are other instruction set architectures, all based on the number of explicit operands.
  - 0-address (stack)
  - 1-address (accumulator)
  - 2-address
  - 3-address

0-Address Machines

- All operands for binary operations are implicit on the *stack*. Only push/pop reference memory.
  - e.g., calculating a = a * b +(c –(d * e))

Code	# Memory Refs
PUSH A	1
PUSH B	1
MUL	0
PUSH C	1
PUSH D	1
PUSH E	1
MUL	0
SUB	0
ADD	0
POP A	1

1-Address Machines

- Accumulator is a source and destination. Second source is explicit.

Opcode	Addr

a = a * b + c –(d * e)

Code	# Memory Refs
LOAD A	1
MUL B	1
ADD C	1
STORE T1	1
LOAD D	1
MUL E	1
STORE T2	1
LOAD T1	1
SUB T2	1
STORE A	1

2-Address Machines

- Two source addresses for operands. One source is also the destination.

Opcode	Addr1	Addr2

a = a * b + c –(d * e)

Code	# Memory Refs
MOVE T1, A	2
MUL T1, B	3
ADD T1, C	3
MOVE T2, D	2
MUL T2, E	3
SUB T1, T2	3
MOVE A, T1	2

3-Address Machines

- One destination operand, two source operands, all explicit

Opcode	Dst	Src1	Src2

a = a * b + c –(d * e)

Code	# Memory Refs
MPY T1, A, B	3
ADD T1, T1, C	3
MPY T2, D, E	3
SUB A, T1, T2	3

Comparison

- Assume 8 registers (3 bits), 32 op-codes (5 bits), 15-bit addresses, 16-bit integers. Which ISA accesses memory the least?

	Instructions	Data refs	Total
0-address	10 x 20 bits = 200 bits	6 x 16 bits = 96 bits	296 bits
1-address	10 x 20 bits = 200 bits	10 x 16 bits = 160 bits	360 bits
1½-address	7 x 23 bits = 161 bits	6 x 16 bits = 96 bits	257 bits
2 address	7 x 35 bits = 245 bits	18 x 16 bits = 288 bits	519 bits
3-address	4 x 50 bits = 200 bits	12 x 16 bits =192 bits	392 bits
3-address (regs)	4 x 38 bits = 152 bits	6 x 16 bits = 96 bits	248 bits

Summary

- The instruction set architecture determines the format of instructions (and therefore the assembly language).
- Four basic types with variations:
  - 0-address (stack)
  - 1-address (accumulator)
  - 2-address (register variant is 1½-address)
  - 3-address (with register variant)
- ISA dramatically affects the number of times memory is accessed.