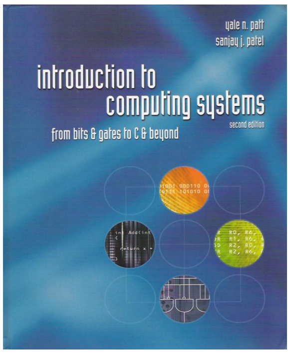
CompSci.210

Review



Introduction to computing

Computer = electronic genius?
- NO! Electronic idiot!
- Does exactly what we tell it to, nothing more.

Goal of CompSci.210
You will be able to write programs in C and understand what's going on underneath – no magic!

Approach
- Build understanding from the bottom up
- Bits ➨ Digital Logic ➨ Gates ➨ Processor ➨ Instructions ➨ C Programming

1-2

Two Recurring Themes

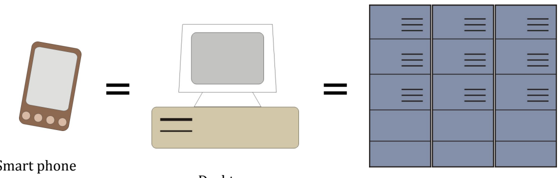
Abstraction
- Productivity enhancer – don't need to worry about details...
  Can drive a car without knowing how the internal combustion engine works.
- ...until something goes wrong!
  Where's the dipstick? What's a spark plug?
- Important to understand the components and how they work together

Hardware vs. Software
- It's not either/or – both are components of a computer system
- Even if you specialize in one, it is important to understand capabilities and limitations of both

1-3

Big Idea #1: Universal Computing Device

All computers, given enough time and memory, are capable of computing exactly the same things



Smart phone

Desktop

Supercomputer

1-4

The Turing Machine

Mathematical model of a device that can perform any computation – Alan Turing (1936)
- ability to read/write symbols on an infinite "tape"
- state transitions, based on current state and symbol

Every computation can be performed by some Turing machine. *(Turing's thesis)*

a,b → T_{add} → a+b

a,b → T_{mul} → ab

Turing machine that adds

Turing machine that multiplies

For more info about Turing machines, see http://www.wikipedia.org/wiki/Turing_machine/

For more about Alan Turing, see http://www.turing.org.uk/turing/

1-5

Universal Turing Machine

A machine that can implement all Turing machines -- this is also a Turing machine!
- inputs: data, plus a description of computation (other TMs)

T_{add}, T_{mul} → U

a,b,c → U → c(a+b)

Universal Turing Machine

U is programmable – so is a computer!
- instructions are part of the input data
- a computer can emulate a Universal Turing Machine

A computer is a universal computing device
Video http://vimeo.com/33559758

1-6

From Theory to Practice

In theory, computer can *compute* anything that's possible to compute

- (caveat) given enough *memory* and *time*

In practice, *solving problems* involves computing under constraints.

- time
  - weather forecast, next frame of animation, ...
- cost
  - cell phone, automotive engine controller, ...
- power
  - cell phone, handheld video game, ...

1-7

From Theory to Practice

In theory, computer can *compute* anything that's possible to compute

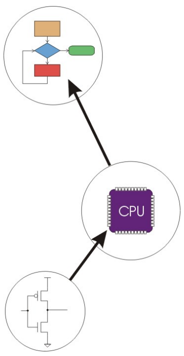
- (Caveat) given enough *memory* and *time*

In practice, *solving problems* involves computing under constraints.

- time
  - weather forecast, next frame of animation, ...
- cost
  - cell phone, automotive engine controller, ...
- power
  - cell phone, handheld video game, ...

1-8

Big Idea #2: Transformations Between Layers



Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

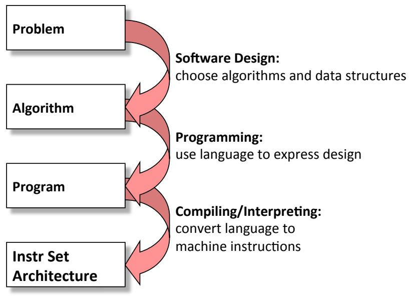
Circuits

Devices

1-9

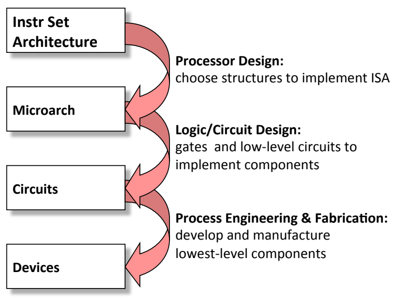
How do we solve a problem using a computer?

A systematic sequence of transformations
between layers of abstraction



1-10

Deeper and Deeper…



1-11

Descriptions of Each Level

Problem Statement
- stated using "natural language"
- may be ambiguous, imprecise

Algorithm
- step-by-step procedure, guaranteed to finish
- definiteness, effective computability, finiteness

Program
- express the algorithm using a computer language
- high-level language, low-level language

Instruction Set Architecture (ISA)
- specifies the set of instructions the computer can perform
- data types, addressing mode

1-12

Descriptions of Each Level (cont.)

Microarchitecture
- detailed organization of a processor implementation
- different implementations of a single ISA

Logic Circuits
- combine basic operations to realize microarchitecture
- many different ways to implement a single function (e.g., addition)

Devices
- properties of materials, manufacturability

1-13

How do we represent data in a computer?

- At the lowest level, a computer is an electronic machine
  - works by controlling the flow of electrons
- Easy to recognize two conditions:
  1. presence of a voltage – we'll call this state "1"
  2. absence of a voltage – we'll call this state "0"
- Could base state on *value* of voltage, but control and detection circuits much more complex.
  - compare turning on a light switch to measuring or regulating voltage

Unsigned Integers - binary

An *n*-bit unsigned integer represents any of 2^n (integer) values:
from 0 to 2^n-1.

2^2	2^1	2^0	Value
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

How to convert decimal to binary video http://youtu.be/qWxiXU02ZQM

Two's Complement Binary

- Problems with sign-magnitude and 1's complement
  - two representations of zero (+0 and –0)
  - arithmetic circuits are complex
    - How to add two sign-magnitude numbers?
      - e.g., try 2 + (-3)
    - How to add two one's complement numbers?
      - e.g., try 4 + (-3)
- *Two's complement* representation developed to make circuits easy for arithmetic.
  - for each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with "normal" addition, ignoring carry out

```
   00101  (5)          01001  (9)
+  11011  (-5)     +  _____   (-9)
   00000  (0)          00000  (0)
```

2-16

Sign Extension (sext)

- Sometimes we want to convert a small number of bits into a larger number of bits
- If we just pad with zeroes on the left:

4-bit		8-bit	
0100	(4)	**00000100**	(still 4)
1100	(-4)	**00001100**	(12, not -4)

- Instead, propagate the MS bit (the sign bit):

4-bit		8-bit	
0100	(4)	**00000100**	(still 4)
1100	(-4)	**11111100**	(still -4)

2-17

Overflow

- If operands are too big, their sum cannot be represented as an *n*-bit 2's comp number
- 5 bits can represent 2^5 or 32 unsigned integers
- Or 0 to 15 positive and -1 to -16 as signed integers

```
     unsigned               signed
   01110  (14)          01110  (14)
+  01000  (8)        +  01000  (8)
   10110  (22)          10110  (-10)
```

- We have overflow in signed binary if:
  - signs of both operands are the same, and
  - sign of sum is different.
- Another test -- easy for hardware:
  - carry into MS bit does not equal carry out

Addition/Subtraction with 2's Complement

- Two's complement representation allows addition and subtraction from a single simple adder.

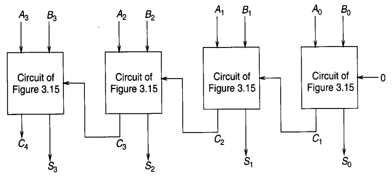


Figure 3.16 A circuit for adding two 4-bit binary numbers

- Circuit to add S = A + B
- To subtract S = A – B invert B and enable carry in

1-19

Logical Operations

- Operations on logical TRUE or FALSE
  - two states -- takes one bit to represent: TRUE=1, FALSE=0

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

- View *n*-bit number as a collection of *n* logical values
  - operation applied to each bit independently (bitwise)

Examples of Logical Operations

- AND
  - useful for clearing bits
    - AND with zero = 0
    - AND with one = no change

```
     11000101
AND  00001111
     00000101
```

- OR
  - useful for setting bits
    - OR with zero = no change
    - OR with one = 1

```
     11000101
OR   00001111
     11001111
```

- NOT
  - unary operation
    one argument flips every bit

```
NOT  11000101
     00111010
```

2-21

Hexadecimal Notation
(not a representation)

- It is often convenient to write binary (base-2) numbers using hexadecimal (base-16) notation instead.
  - fewer digits -- four bits per hex digit
  - less error prone -- easy to corrupt long string of 1's and 0's

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Binary	Hex	Decimal
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

2-22

Converting from Binary Hexadecimal

- Every four bits is a hex digit
  - start grouping from right-hand side

011	1010	1000	1111	0100	1101	0111
⇓	⇓	⇓	⇓	⇓	⇓	⇓
3	A	8	F	4	D	7

This is not a new machine representation, just a convenient way to write the number.

This video shows you how to convert binary to hex
http://www.youtube.com/watch?v=W_NpD248CdE
(with binary to octal thrown in)

2-23

Fractions: Fixed-Point

- How can we represent fractions?
  - Use a "binary point" to separate positive from negative powers of two -- just like "decimal point."
  - 2's comp addition and subtraction still work
    - only if binary points are aligned

$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$

```
  00101000.101 (40.625)
+ 11111110.110 (-1.25)
  00100111.011 (39.375)
```

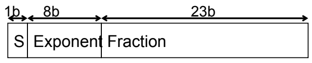
No new operations -- same as integer arithmetic

Video: how to convert decimal fractions to binary http://youtu.be/Y4Q9PnjKhac

2-24

Very Large and Very Small: Floating-Point

- Large values: 6.023×10^{23} -- requires 79 bits
- Small values: 6.626×10^{-34} -- requires >110 bits
- Use equivalent of "scientific notation": $F \times 2^E$
- Need to represent F (*fraction*), E (*exponent*), and sign.
- IEEE 754 Floating-Point Standard (32-bits):



- Exponent uses "biased" representation (no sign bit)
- Fraction has implicit 1

Video converting decimal to floating-point binary representation
http://youtu.be/iQFG7sAa7i4

2-25

Floating-Point Arithmetic

- Floating point operations may overflow but, more importantly, floating point operations are inherently *inexact*
- ➢ Some numbers (*e.g.* "repeating decimal") cannot be represented exactly.
- ➢ Introduces the "Rounding" problem
  - Every inexact result creates a difference between the mathematical value and the computed value.
  - Errors accumulate, often benignly by cancelling out.
  - Worst-case accumulation of error can be enormous.

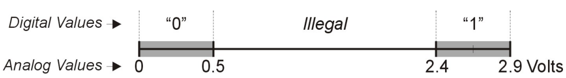
2-26

Logic Gates

Use switch behavior of transistors
to implement logical functions: AND, OR, NOT

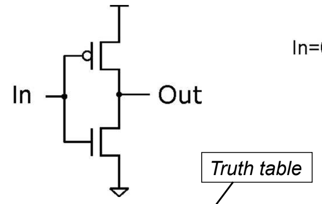
Digital symbols:

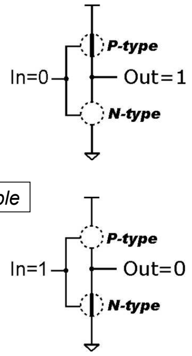
– recall that we assign a range of analog voltages to each digital (logic) symbol



– assignment of voltage ranges depends on electrical properties of transistors being used
  - typical values for "1": +5V, +3.3V, +2.9V
  - from now on we'll use +2.9V

Inverter (NOT Gate)

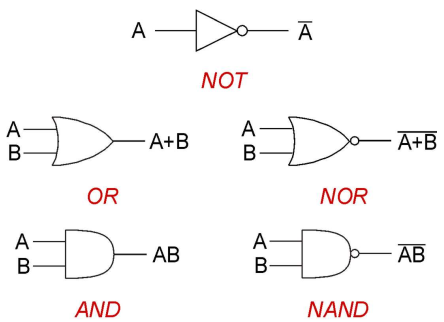




In	Out
0 V	2.9 V
2.9 V	0 V

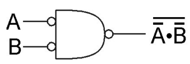
In	Out
0	1
1	0

Basic Logic Gates



DeMorgan's Law

Converting AND to OR (with some help from NOT)
Consider the following gate:



A	B	$\bar{A}$	$\bar{B}$	$\bar{A} \cdot \bar{B}$	$\overline{\bar{A} \cdot \bar{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

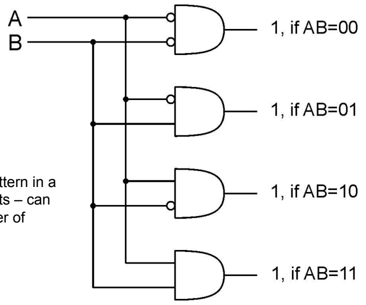
Shows that you can write an expression like "not (A or B)" as "(not A) and (not B)". Similarly, "not (A or B)" can be written as "(not A) and (not B)"

Watch this video
http://youtu.be/tKnS3s8fOu4

Therefore, you can implement any truth table using only NAND (or NOR) gates

Decoder

- n inputs, 2^n outputs
  - exactly one output is 1 (true) for each possible input pattern



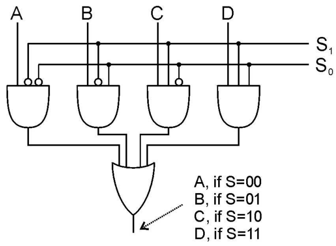
2-bit decoder

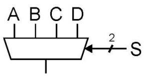
Can detect a pattern in a string of input bits – can have any number of inputs

Multiplexer (MUX)

n-bit selector and 2^n inputs, one output

- output equals one of the inputs, depending on selector S_1 & S_2

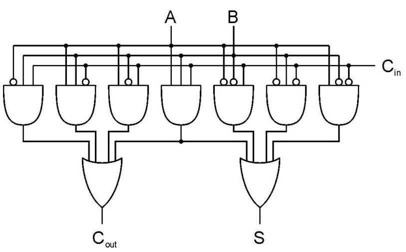




4-to-1 MUX

Full Adder

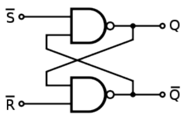
Add two bits and carry-in,
produce one-bit sum and carry-out



A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

SR-Latch: Simple Storage Element
Flip-Flop

S is used to "set" the element – set output Q to one
R is used to "reset" or "clear" the element – set output Q to zero



NAND SR-Latch

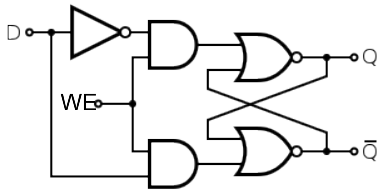
$\overline{SR}$ latch operation		
$\overline{S}$	$\overline{R}$	Action
0	0	Restricted combination
0	1	Q = 1
1	0	Q = 0
1	1	No Change

This gives us the ability to store a bit (either 0 or 1)

Watch the video http://youtu.be/ti5jD7Q7BSA

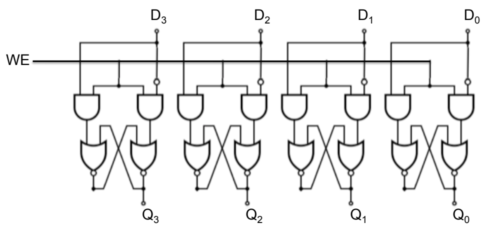
Gated D-Latch

Two inputs: D (data) and WE (write enable)
- when WE = 1, latch is set to value of D
  - S = D, R = NOT(D)
- when WE = 0, latch holds previous value
  - S = R = 0

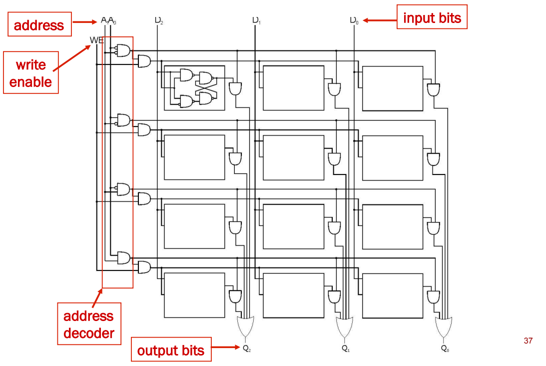


CS210 35

A 4 bit register



Memory



37

Finite State Machines

A description of a system with the following components:

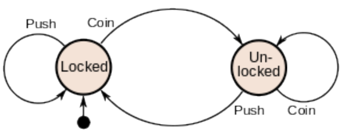
1. A finite number of states
2. A finite number of external inputs
3. A finite number of external outputs
4. An explicit specification of all state transitions
5. An explicit specification of what determines each external output value

Often described by a state diagram.
- Inputs trigger state transitions.
- Outputs are associated with each state (or with each transition).

Finite State Machines

The turnstile has 2 states
- **locked** and **unlocked**

The turnstile has 2 inputs
- putting in a coin (**coin**)
- pushing the bar (**push**)

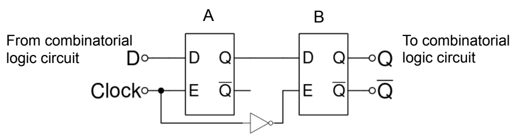


State diagram

CS210 39

Storage: Master-Slave Flip flop

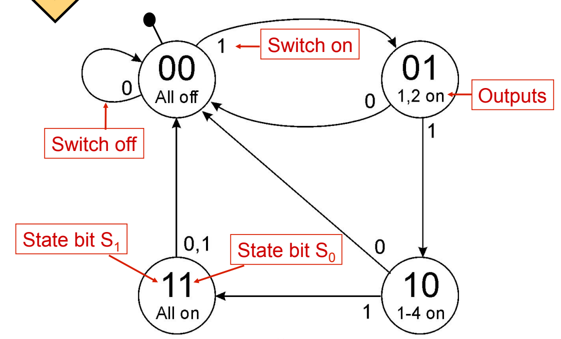
Master-slave edge triggered D flip-flop



During 1st phase (clock=1), previously-computed state in A becomes *current* state in Latch B and is sent to the logic circuit.

During 2nd phase (clock=0), *next* state, computed by logic circuit, is stored in Latch A.

Traffic Sign State Diagram



Transition on each clock cycle

Traffic Sign Truth Tables

Outputs
(depend only on state: S_1S_0)

Lights 1 and 2 (Z)
Lights 3 and 4 (Y)
Light 5 (X)

S_1	S_0	Z	Y	X
0	0	0	0	0
0	1	1	0	0
1	0	1	1	0
1	1	1	1	1

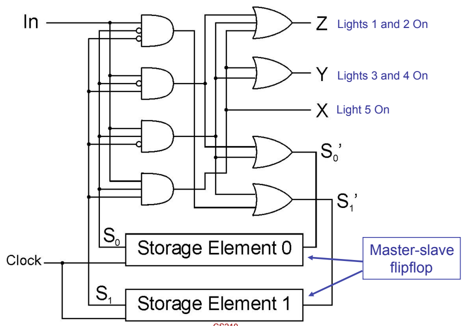
Next State: $S_1'S_0'$
(depend on state and input)

Switch (On)

On	S_1	S_0	S_1'	S_0'
0	-	-	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

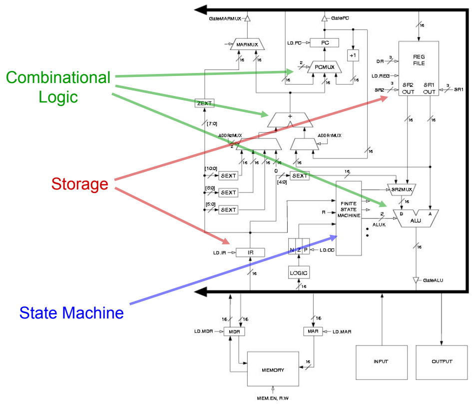
Whenever On = 0 (false), next state is 00 (off)
S_1 & S_0 are irrelevant

Traffic Sign Logic



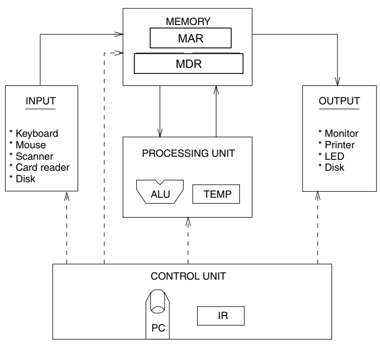
CS210 43

LC-3 Data Path



44

Von Neumann Model



CS210 45

Instruction Processing

- Fetch instruction from memory
- Decode instruction
- Evaluate addresses
- Fetch operands from memory
- Execute operation
- Store result

Instruction

- The instruction is the fundamental unit of work.
- Specifies two things:
  - ***opcode***: operation to be performed (e.g. ADD)
  - ***operands***: data/locations to be used for operation

- An instruction is encoded as a **sequence of bits** *(Just like data!)*
  - Often, but not always, instructions have a fixed length, such as 16 or 32 bits.
  - Control unit interprets instruction: generates sequence of control signals to carry out operation.
  - Operation is either executed completely, or not at all.

- A computer's instructions and their formats is known as its ***Instruction Set Architecture (ISA)***.

Example: LC-3 ADD Instruction

- LC-3 has 16-bit instructions.
  - Each instruction has a four-bit opcode, bits [15:12].
- LC-3 has eight *registers* (R0-R7) for temporary storage
  - Sources and destination of ADD are registers

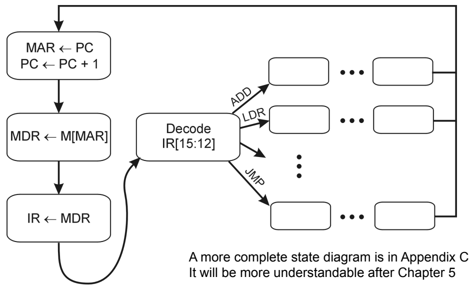
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD				Dst			Src1			0	0	0	Src2		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0

"Add the contents of R2 to the contents of R6, and store the result in R6."

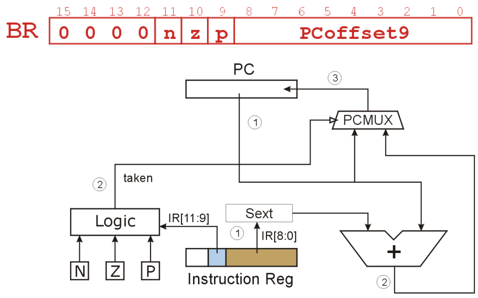
Control Unit State Diagram

The control unit is a state machine. Here is part of a simplified state diagram for the LC-3:



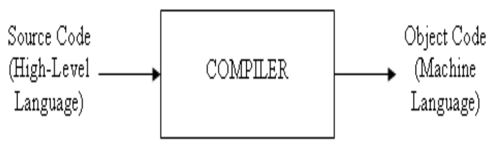
A more complete state diagram is in Appendix C
It will be more understandable after Chapter 5

BR (PC-Relative)

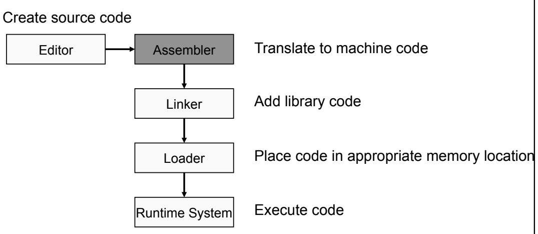


Compiler

- Translate high-level languages into machine code.
- The machine code version can be loaded into the machine and run without any further help as it is complete in itself.
- The high-level language version of the program is called the source code and the resulting machine code program is called the object code.



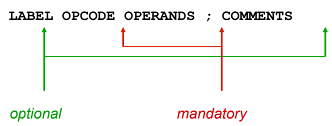
Assembler



/opt/lc3tools/lc3as
usage: ./lc3as <ASM filename>

LC-3 Assembly Language Syntax

- Each line of code is
  - An instruction
  - An assembler directive (or pseudo-op)
  - A comment
- Whitespace is ignored
- Instruction format:



C

- Developed at AT&T Bell Labs1969-73
- designed to provides constructs that map efficiently to machine instructions
- found lasting use in applications that had formerly been coded in assembly language
- Influenced C++, C#, Java, JavaScript, Limbo, LPC, Objective-C, Perl, PHP, Python…