

## Section 11.3

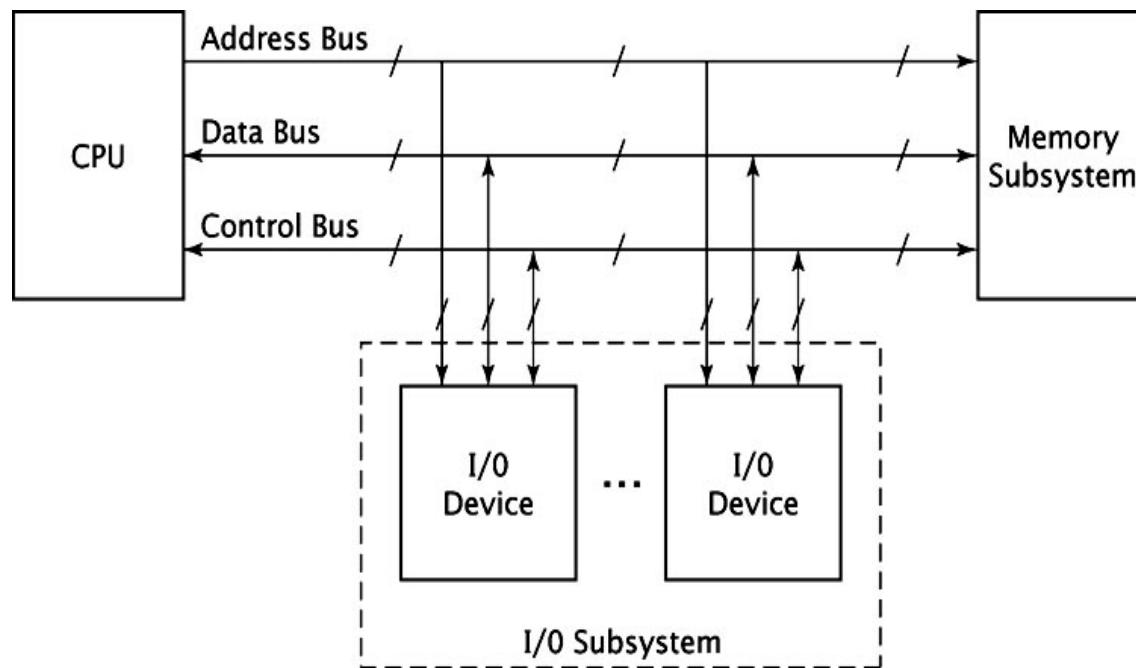
# Computer Sub Systems

# Computer Sub Systems

- Computer has three major components:
  - Central Processing Unit (CPU)
    - instructions are fetched and executed
  - Memory system
    - Programs and data are stored
  - Input/Output (I/O) system
    - Handles input and output data to and from the memory system

# Computer Sub Systems

- System structure shown in this diagram:



# Clocks

- Computer contains at least one clock
- A fixed number of clock cycles are required to:
  - carry out each data movement
  - carry out each computational operation
- Clock frequency
  - measured in megahertz or gigahertz
  - determines speed all operations are carried out
- Clock cycle time
  - the reciprocal of clock frequency
  - An 800 MHz clock has a cycle time of 1.25 ns

# Clocks

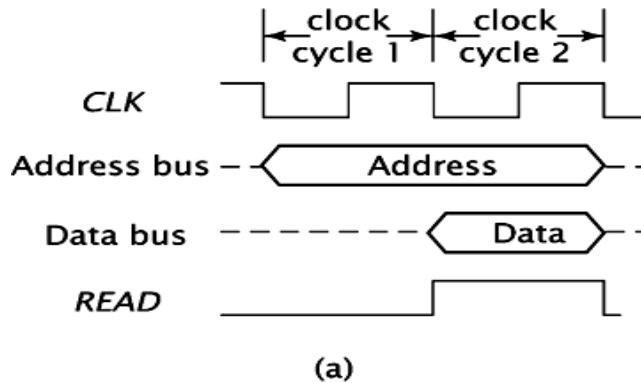
- Clock speed does not equate to CPU performance
- CPU time required to run a program is given by this equation:

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

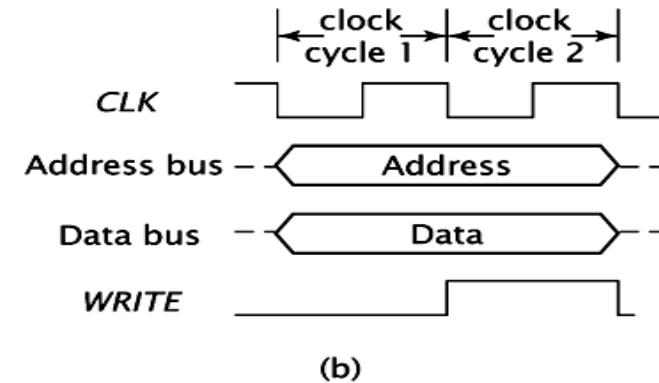
- Improve CPU throughput
  - Reduce number of instructions in program
  - Reduce number of cycles per instruction
  - Reduce number of nanoseconds per clock cycle

# Timing and Other Operations

- Components synchronized by the system clock
  - Clock "ticks" insure operations happen at regular intervals
- For memory read operations (Fig (a) below):
  - CPU places address on the address bus
  - Memory subsystem accesses the desired memory location
  - CPU asserts READ signal
  - Memory places data onto the data bus
- For memory write operations (Fig (b) below):
  - CPU places address and data on respective buses
  - Asserts WRITE signal
  - Memory writes data at the given address
- For operations (on computers with isolated I/O):
  - CPU must assert IO/M signal
  - Set to 1 means I/O operations, set to 0 means memory operation



(a)



(b)

# Memory System

- **Memory system - an array of memory locations**
  - Each stores multiple-bit binary data (typically 8 bits called a byte)
  - Memory locations are selected by address for either read or write operation
  - It normally has
    - address input for the address to select the location
    - data input for the data to be written to the location
    - data output for the data read from the location
    - chip enable control to enable the entire memory
    - read/write control to start read or write operation

# Memory Organization

- Computer memory

- Linear array of addressable storage cells similar to registers
- Byte-addressable, or word-addressable, word typically consists of two or more bytes
- Constructed of RAM chips, often referred to in terms of length  $\times$  width.

- If memory word size is 16 bits

- A  $4M \times 16$  RAM chip has 4 megabytes of 16-bit memory locations

# Memory Organization

## ● Memory Types

- **Read-Only Memory (ROM)**
  - Masked ROM is manufactured with data inside; does not change
- **Programmable ROM (PROM)**
  - program-once devices
- **Erasable PROM (EPROM)**
  - erased by holding the chip's "window" under ultraviolet light
- **Electrically erasable PROM (EEPROM)**
  - erased by the programming device; portions can be selectively erased
- **Flash EEPROM**
  - erasable in blocks (good for digital cameras).
- **All ROM chips have:**
  - n address pins ( $A_{n-1} \dots A_0$ ) selecting  $2^n$  locations,
  - m data pins ( $D_{m-1} \dots D_0$ ).
  - CE chip enable that enables/disables the entire chip
  - OE output enable that keeps the ROM from outputting data

# Memory Organization

- **Memory Types (cont)**

- **Random Access Memory (RAM)**

- ***Dynamic RAM (DRAM)***

- constructed from leaky capacitors and requires periodic recharge.

- ***Static RAM (SRAM)***

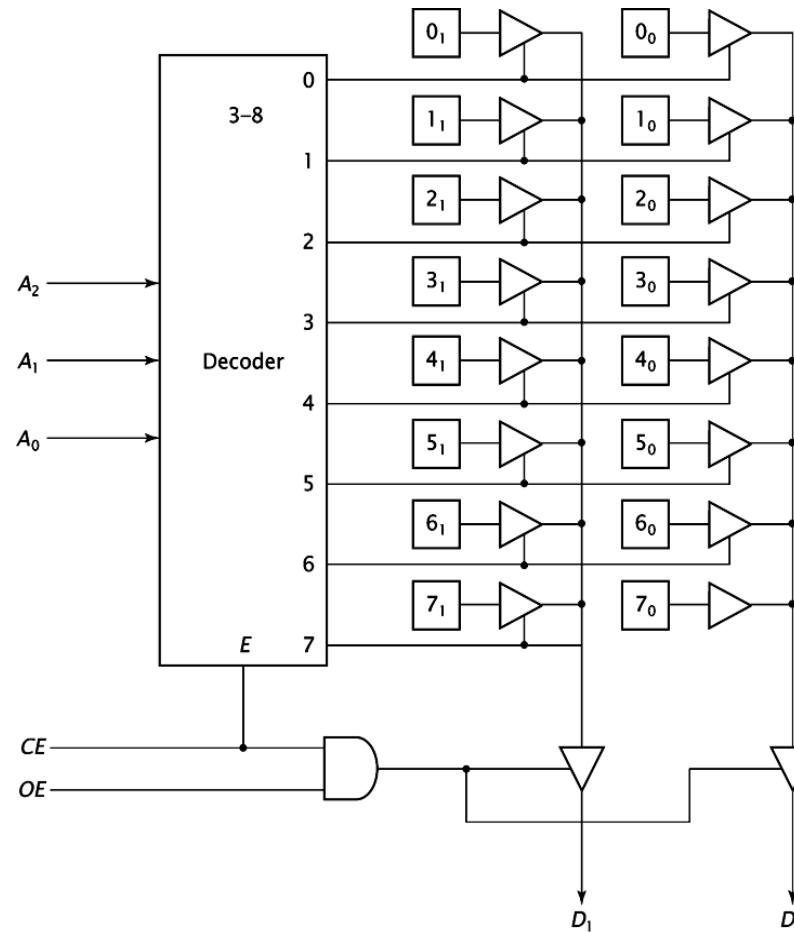
- does not need refreshing; more expensive

- **All RAM chips have:**

- n address pins ( $A_{n-1} \dots A_0$ ) selecting  $2^n$  locations
    - m data pins ( $D_{m-1} \dots D_0$ )
    - some kind of R/W input that indicates if RAM should read or write data

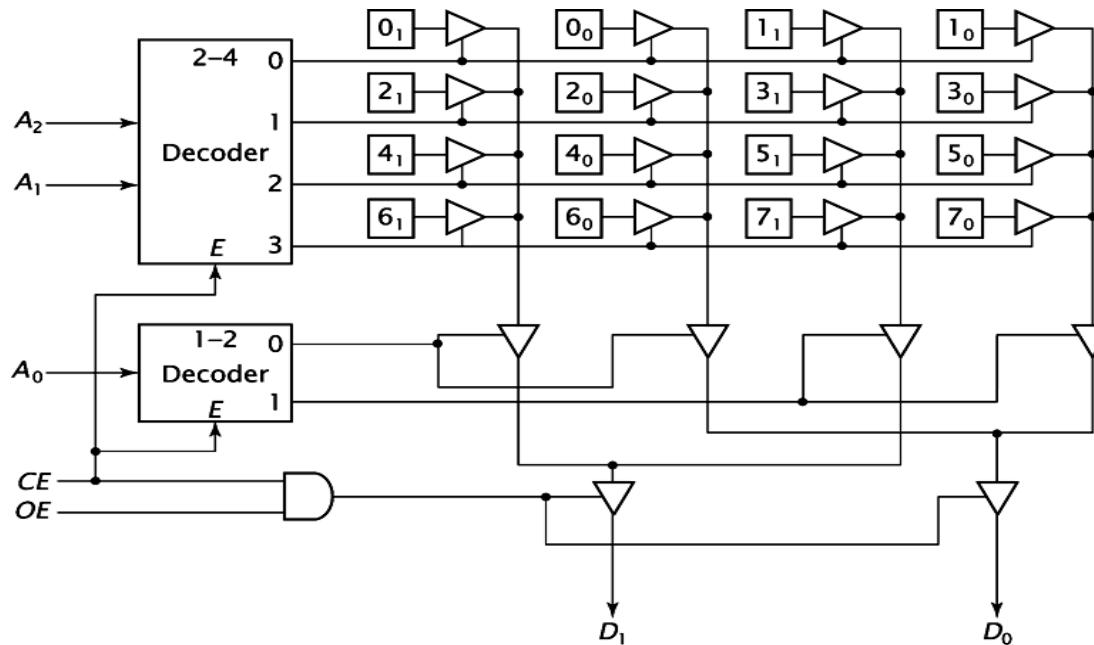
# Chip Organization

- Simple 8x2 ROM configuration using *linear organization, like fig 11.41*



# Chip Organization

- When number of memory locations becomes large
  - more than one decoder needed because the size of the decoder becomes too large
  - Below a  $8 \times 2$  ROM using a 2d organization

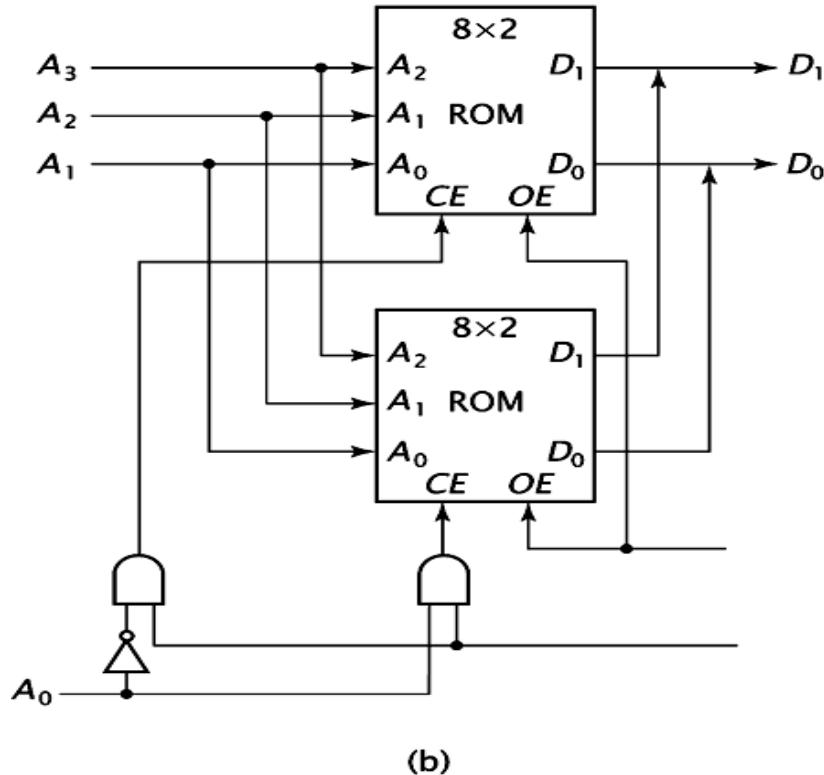


# Memory Subsystem

- The storage for each bit in the memory can be
  - a D-latch for static memory (fast and expensive)
  - or other devices for dynamic memory

# Memory Subsystem

- How to build a memory system out of more than one chip?
- Below a  $16 \times 2$  ROM out of two  $8 \times 2$  ROM's



Upper chip stores locations XXX0 and the lower chip stores locations XXX1.

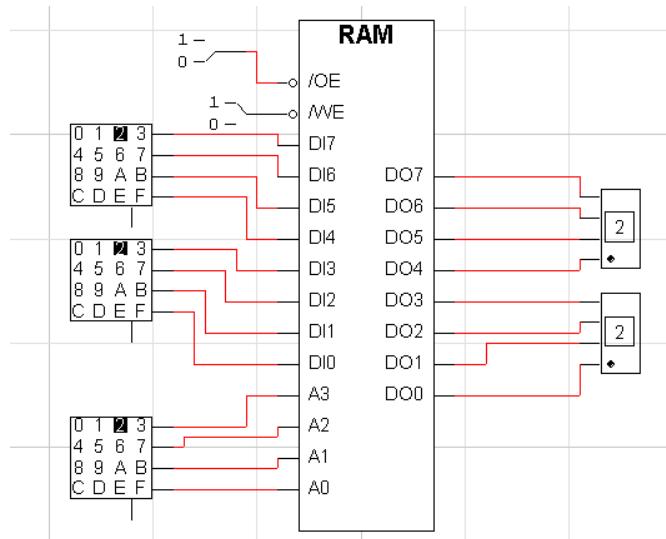
Low-order interleaving can provide multi-byte read capabilities.

# Memory Subsystem

- Physical memory usually consists of more than one RAM chip
- Access is more efficient when memory is organized into banks of chips with the addresses interleaved across the chips

# Memory Subsystem

- A 16x8 memory chip
  - /CE is active-low chip enable
  - /WE is read/write control
  - It is storing the value 22 at memory address 2



# Multi-byte Organization

- Byte ordering, or *endianness*
  - Another major architectural consideration
- A two-byte integer
  - may be stored so that the least significant byte is followed by the most significant byte
  - or vice versa.
- In *Little endian* machines
  - least significant byte is followed by the most significant byte
- *Big endian* machines
  - most significant byte first (at the lower address)

# Multi-byte Organization

- ***Big endian***: most significant byte in location X, next byte in location X+1.
- ***Little endian***: least significant byte in location X, next byte in location X+1.
- ***Alignment***: storing bytes so they can be read in a multi-byte read.
  - E.g., Motorola 68040 can read 4 bytes, but only if the 2 least significant address bits are 00. Can read addresses 100, 101, 102, and 103 in one instruction cycle. However, it cannot read 101, 102, 103, and 104.

# Multi-byte Organization

- Example: hexadecimal number 12345678
- big endian and small endian arrangements of the bytes are shown below

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

## Section 11.3

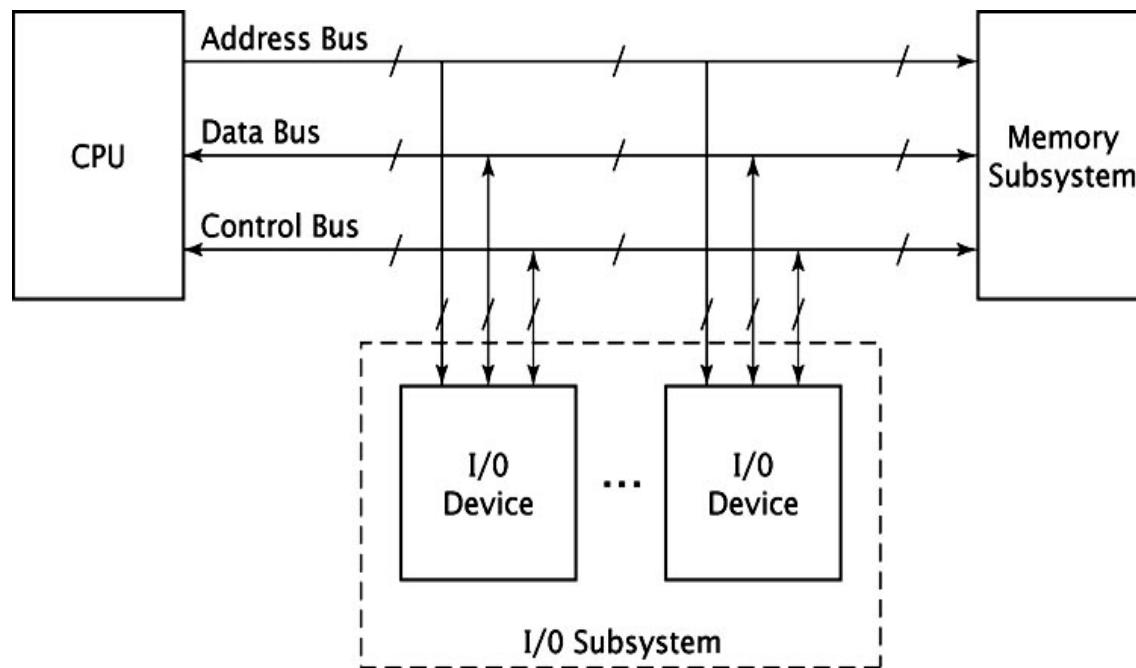
# Computer Sub Systems

# Computer Sub Systems

- Computer has three major components:
  - Central Processing Unit (CPU)
    - instructions are fetched and executed
  - Memory system
    - Programs and data are stored
  - Input/Output (I/O) system
    - Handles input and output data to and from the memory system

# Computer Sub Systems

- System structure shown in this diagram:



# Clocks

- Computer contains at least one clock
- A fixed number of clock cycles are required to:
  - carry out each data movement
  - carry out each computational operation
- Clock frequency
  - measured in megahertz or gigahertz
  - determines speed all operations are carried out
- Clock cycle time
  - the reciprocal of clock frequency
  - An 800 MHz clock has a cycle time of 1.25 ns

# Clocks

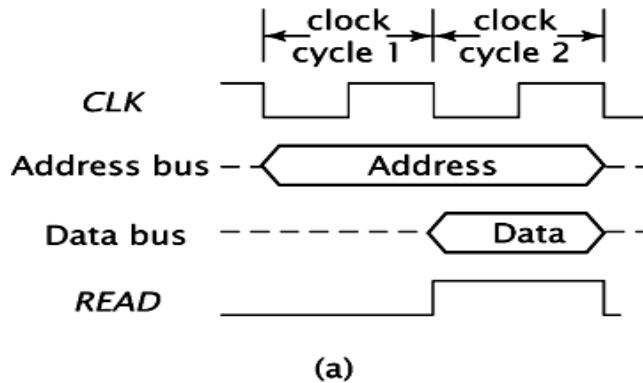
- Clock speed does not equate to CPU performance
- CPU time required to run a program is given by this equation:

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

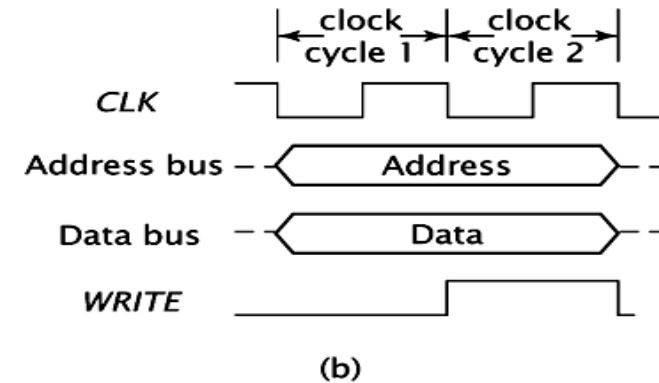
- Improve CPU throughput
  - Reduce number of instructions in program
  - Reduce number of cycles per instruction
  - Reduce number of nanoseconds per clock cycle

# Timing and Other Operations

- Components synchronized by the system clock
  - Clock "ticks" insure operations happen at regular intervals
- For memory read operations (Fig (a) below):
  - CPU places address on the address bus
  - Memory subsystem accesses the desired memory location
  - CPU asserts READ signal
  - Memory places data onto the data bus
- For memory write operations (Fig (b) below):
  - CPU places address and data on respective buses
  - Asserts WRITE signal
  - Memory writes data at the given address
- For operations (on computers with isolated I/O):
  - CPU must assert IO/M signal
  - Set to 1 means I/O operations, set to 0 means memory operation



(a)



(b)

# Memory System

- **Memory system - an array of memory locations**
  - Each stores multiple-bit binary data (typically 8 bits called a byte)
  - Memory locations are selected by address for either read or write operation
  - It normally has
    - address input for the address to select the location
    - data input for the data to be written to the location
    - data output for the data read from the location
    - chip enable control to enable the entire memory
    - read/write control to start read or write operation

# Memory Organization

- **Computer memory**

- Linear array of addressable storage cells similar to registers
- Byte-addressable, or word-addressable, word typically consists of two or more bytes
- Constructed of RAM chips, often referred to in terms of length  $\times$  width.

- **If memory word size is 16 bits**

- A  $4M \times 16$  RAM chip has 4 megabytes of 16-bit memory locations

# Memory Organization

## ● Memory Types

- **Read-Only Memory (ROM)**
  - Masked ROM is manufactured with data inside; does not change
- **Programmable ROM (PROM)**
  - program-once devices
- **Erasable PROM (EPROM)**
  - erased by holding the chip's "window" under ultraviolet light
- **Electrically erasable PROM (EEPROM)**
  - erased by the programming device; portions can be selectively erased
- **Flash EEPROM**
  - erasable in blocks (good for digital cameras).
- **All ROM chips have:**
  - n address pins ( $A_{n-1} \dots A_0$ ) selecting  $2^n$  locations,
  - m data pins ( $D_{m-1} \dots D_0$ ).
  - CE chip enable that enables/disables the entire chip
  - OE output enable that keeps the ROM from outputting data

# Memory Organization

- **Memory Types (cont)**

- **Random Access Memory (RAM)**

- ***Dynamic RAM (DRAM)***

- constructed from leaky capacitors and requires periodic recharge.

- ***Static RAM (SRAM)***

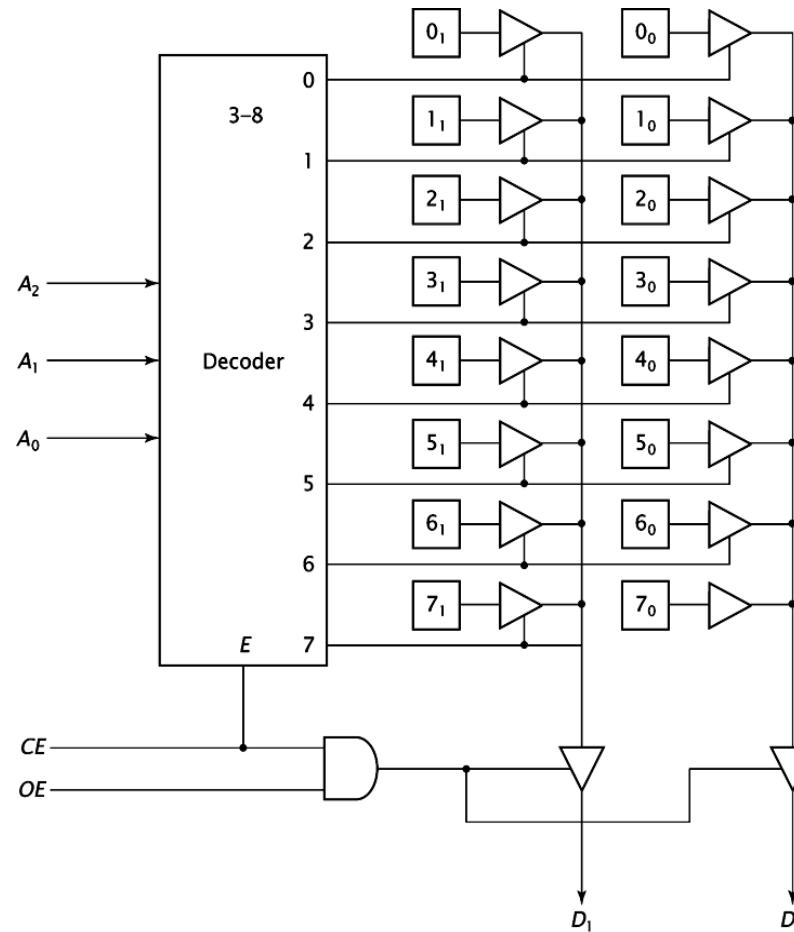
- does not need refreshing; more expensive

- **All RAM chips have:**

- n address pins ( $A_{n-1} \dots A_0$ ) selecting  $2^n$  locations
    - m data pins ( $D_{m-1} \dots D_0$ )
    - some kind of R/W input that indicates if RAM should read or write data

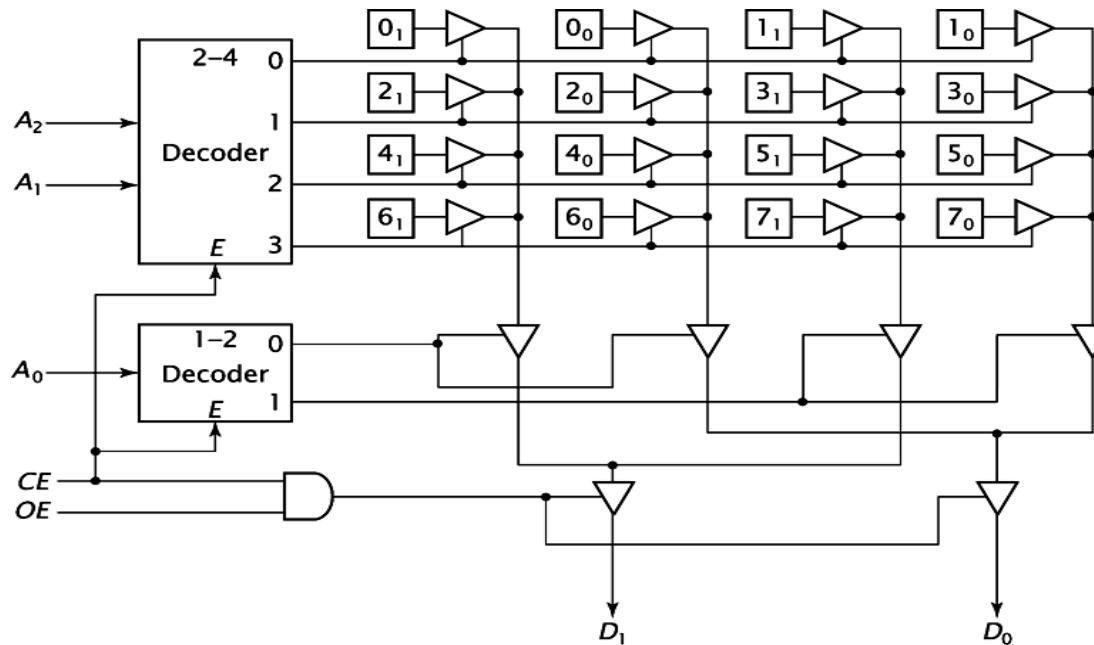
# Chip Organization

- Simple 8x2 ROM configuration using *linear organization, like fig 11.41*



# Chip Organization

- When number of memory locations becomes large
  - more than one decoder needed because the size of the decoder becomes too large
  - Below a  $8 \times 2$  ROM using a 2d organization

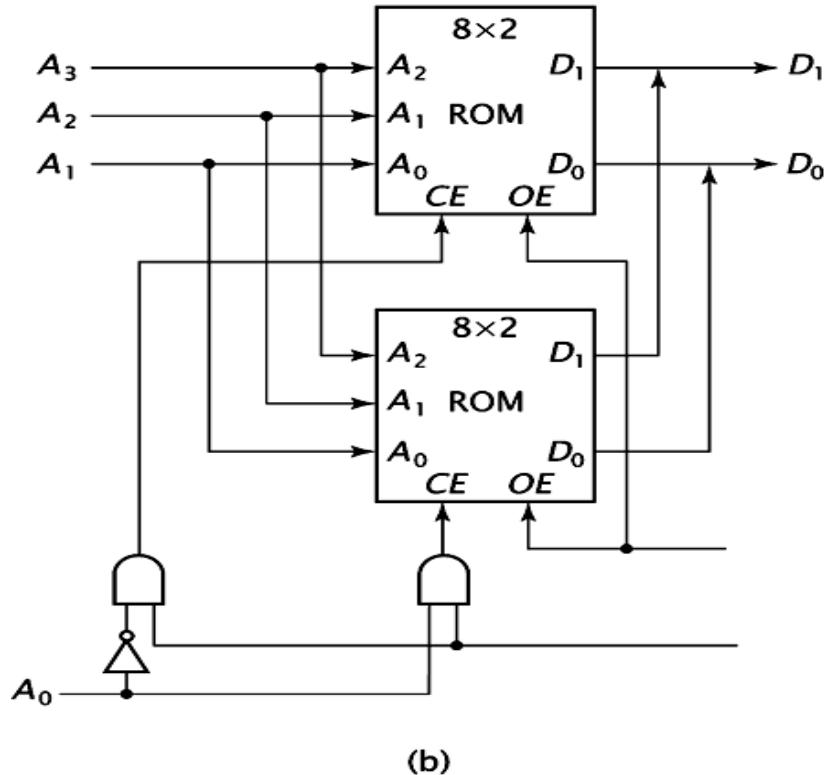


# Memory Subsystem

- The storage for each bit in the memory can be
  - a D-latch for static memory (fast and expensive)
  - or other devices for dynamic memory

# Memory Subsystem

- How to build a memory system out of more than one chip?
- Below a  $16 \times 2$  ROM out of two  $8 \times 2$  ROM's



Upper chip stores locations XXX0 and the lower chip stores locations XXX1.

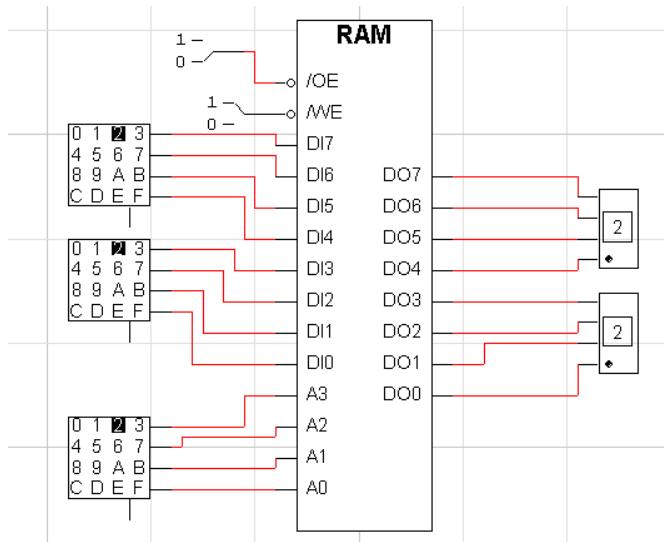
Low-order interleaving can provide multi-byte read capabilities.

# Memory Subsystem

- Physical memory usually consists of more than one RAM chip
- Access is more efficient when memory is organized into banks of chips with the addresses interleaved across the chips

# Memory Subsystem

- A 16x8 memory chip
  - /CE is active-low chip enable
  - /WE is read/write control
  - It is storing the value 22 at memory address 2



# Multi-byte Organization

- Byte ordering, or *endianness*
  - Another major architectural consideration
- A two-byte integer
  - may be stored so that the least significant byte is followed by the most significant byte
  - or vice versa.
- In *Little endian* machines
  - least significant byte is followed by the most significant byte
- *Big endian* machines
  - most significant byte first (at the lower address)

# Multi-byte Organization

- ***Big endian***: most significant byte in location X, next byte in location X+1.
- ***Little endian***: least significant byte in location X, next byte in location X+1.
- ***Alignment***: storing bytes so they can be read in a multi-byte read.
  - E.g., Motorola 68040 can read 4 bytes, but only if the 2 least significant address bits are 00. Can read addresses 100, 101, 102, and 103 in one instruction cycle. However, it cannot read 101, 102, 103, and 104.

# Multi-byte Organization

- Example: hexadecimal number 12345678
- big endian and small endian arrangements of the bytes are shown below

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

# **Chapter 11:**

# **Sequential Circuits and**

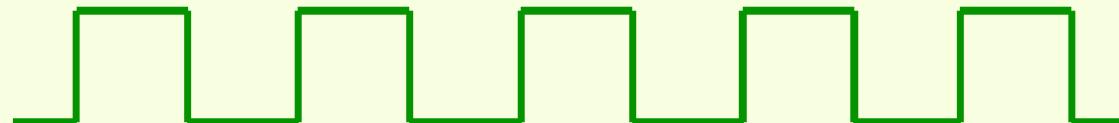
# **Finite State Machines**

# Sequential Circuits

- **Combinational logic**
  - Need immediate application of Boolean function to a set of inputs
- **Will not work when circuit needs to change value with consideration to current state and inputs**
  - Need to “remember” the current state
- ***Sequential logic circuits* provide this functionality**

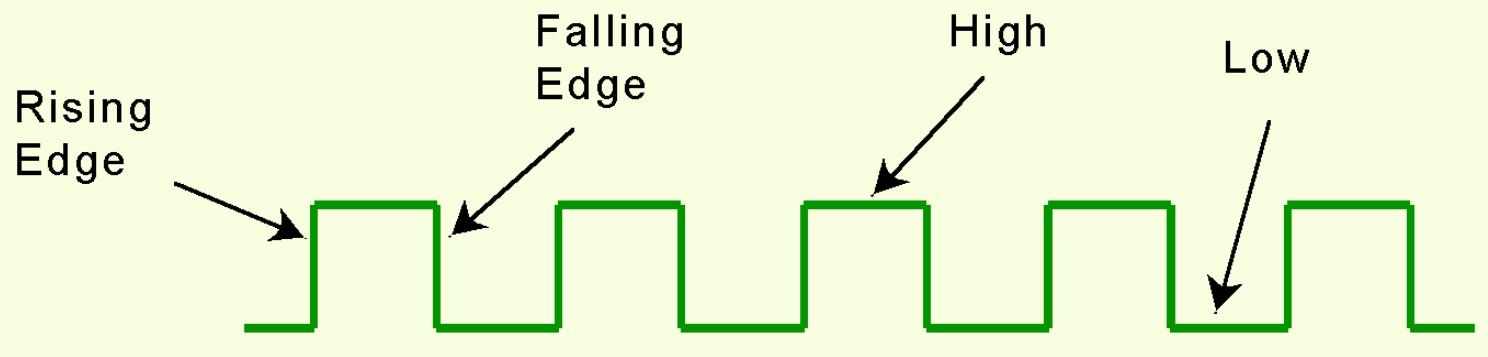
# Sequential Circuits

- Sequential logic circuits require events to be sequenced
- Clocks
  - Control state changes
  - Special circuit that sends electrical pulses through a circuit
  - Produce electrical waveforms as shown below



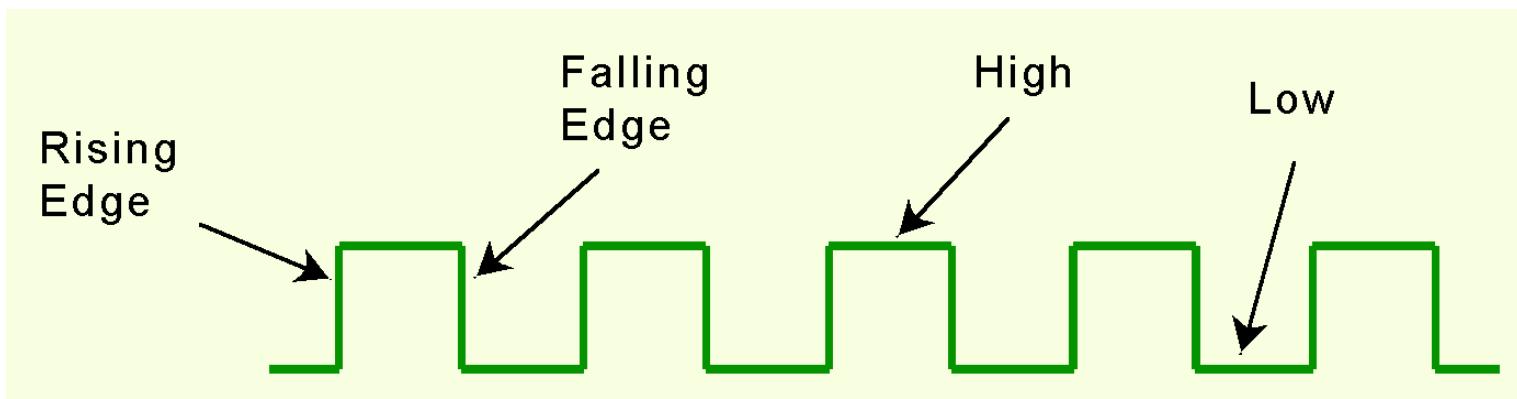
# Sequential Circuits

- State changes occur in sequential circuits at clock ticks
- Circuits can change state
  - on the rising edge -or-
  - falling edge -or-
  - when the clock pulse reaches its highest voltage



# Sequential Circuits

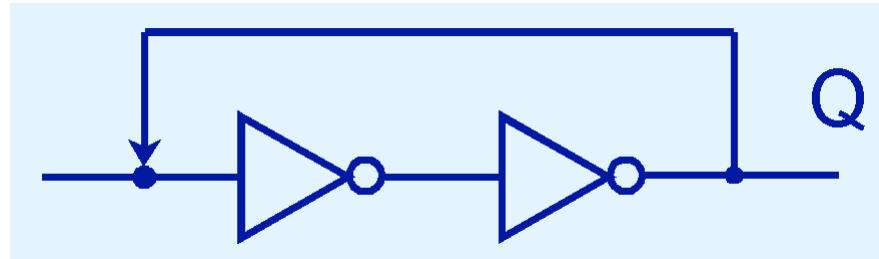
- ***Edge-triggered circuits change state on***
  - the rising edge of the clock pulse
  - or falling edge of the clock pulse
- ***Level-triggered circuits change state when***
  - clock voltage reaches highest or lowest level



# Sequential Circuits

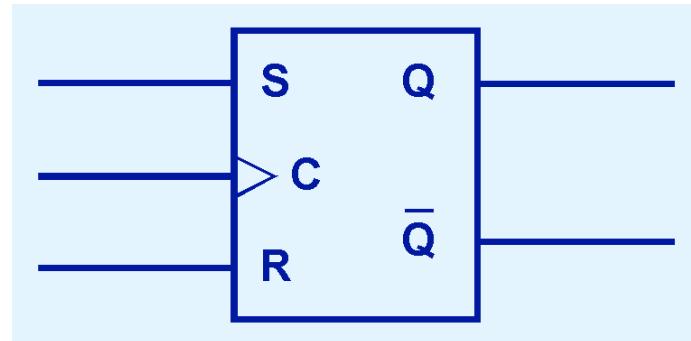
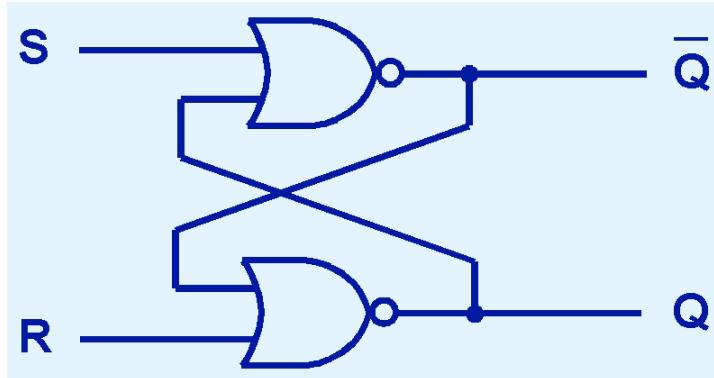
- Sequential circuits rely on *feedback*
- Feedback
  - output is looped back to the input
- Example of this concept shown below

If Q is 0 it will always be 0, if it is 1, it will always be 1. Why?



# Sequential Circuits

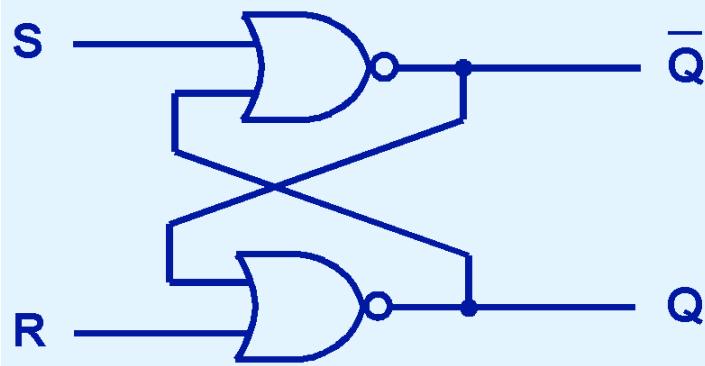
- SR Latch (flip-flop)
  - “SR” stands for set/reset
  - most basic sequential logic components
- SR latch circuit & block diagram shown below



# Sequential Circuits

- Behavior of SR latch
  - described by a characteristic table
- $Q(t)$ 
  - value of the output at time t
- $Q(t+1)$ 
  - value of Q after the next clock pulse

x	y	out = x NOR y
0	0	1
1	0	0
0	1	0
1	1	0



S	R	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

# Sequential Circuits

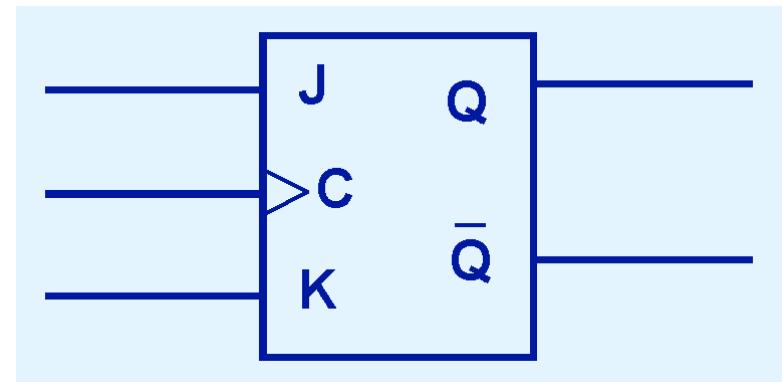
- SR latch- three inputs:
  - S, R, current output: Q
- Note: two undefined values
  - S and R are both 1 – SR flip-flop is unstable

Present State			Next State
S	R	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

# Sequential Circuits

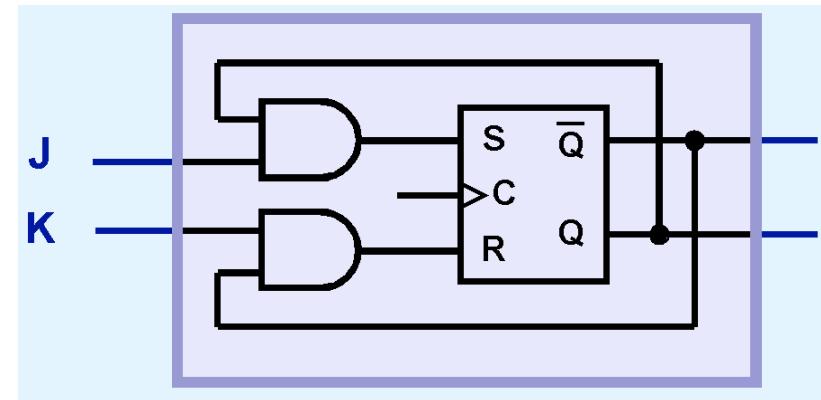
- If inputs to an SR latch will never both be 1
  - Then circuit will never be unstable
- Modified SR latch
  - Provides stable state when both inputs are 1

- Modified flip-flop is called JK flip-flop
- “JK” is in honor of Jack Kilby



# Sequential Circuits

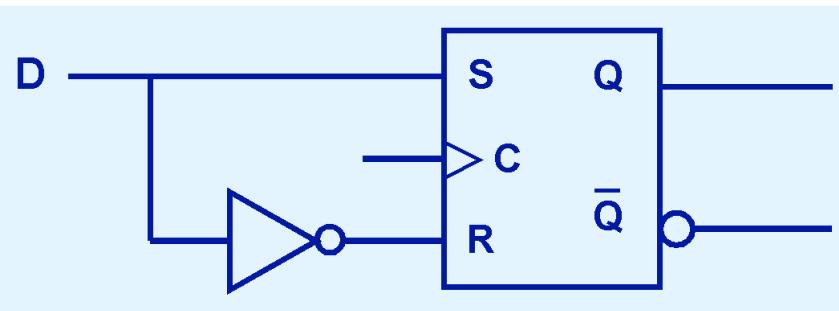
- JK flip-flop
  - Modified SR flip-flop
  - Adding 2 ANDs
- Characteristic table indicates
  - stable for all inputs



J	K	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}(t)$

# Sequential Circuits

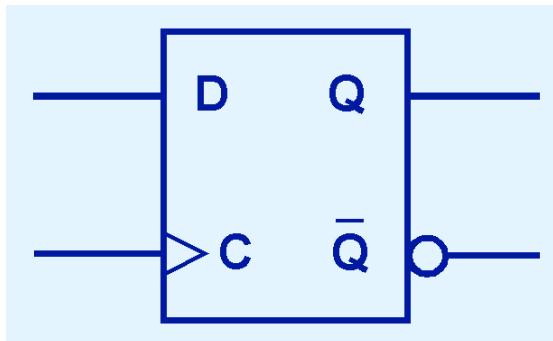
- D flip-flop – Data flip-flop
  - Another modification of SR flip-flop
  - Characteristic table is shown
- Output changes
  - when the value of D changes
  - Stores D (data) until next clock pulse



D	$Q(t+1)$
0	0
1	1

# Sequential Circuits

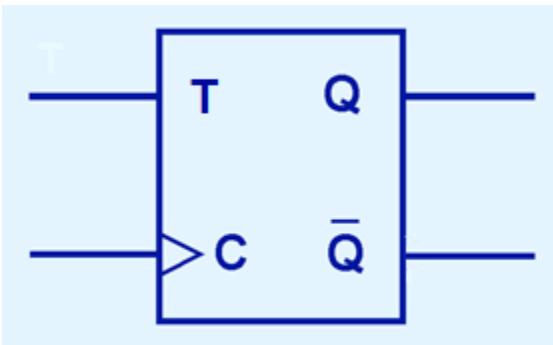
- D flip-flop
  - fundamental circuit of computer memory
- Block diagram & Characteristic table for D flip-flop are shown below



D	$Q(t+1)$
0	0
1	1

# Sequential Circuits

- T flip-flop – Toggle flip-flop
  - Only one input T
- Output changes
  - If  $T=0$  – state remains the same
  - If  $T=1$  – state toggles from 0 to 1 or from 1 to 0
- Block diagram & Characteristic table for T flip-flop are shown below



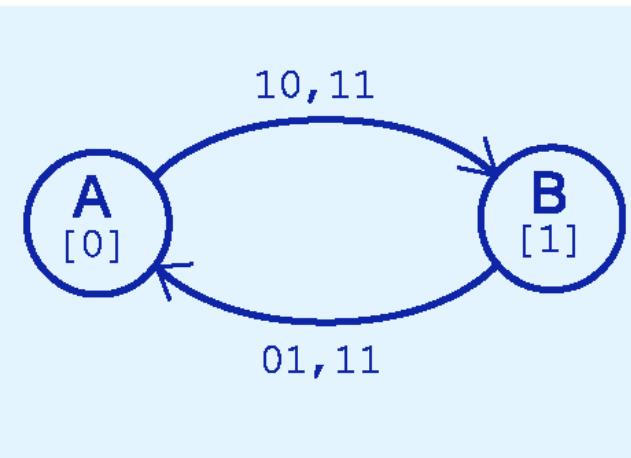
T(t)	Q(t)	Q(t+1)	Condition
0	0	0	no change
0	1	1	no change
1	0	1	Toggle
1	1	0	Toggle

# Sequential Circuits

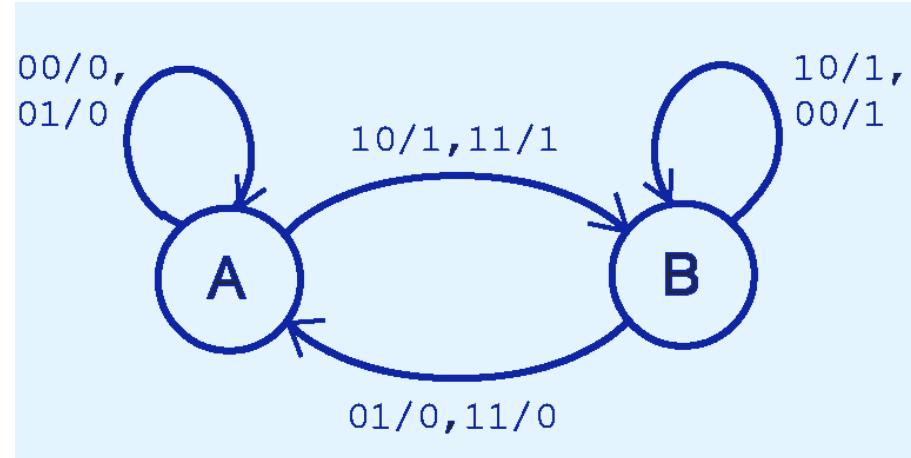
- Characteristic tables or finite state machines (FSMs)
  - Express behavior of sequential circuits
  - FSMs consist of
    - set of nodes that hold the states of the machine
    - set of arcs that connect the states
- Moore and Mealy machines are two types of FSMs
  - Moore machines place outputs on each node
  - Mealy machines present their outputs on the transitions
  - They are equivalent
- *Textbook uses Mealy Machine Convention*

# Sequential Circuits

- Behavior of JK flop-flop is depicted below



Moore machine

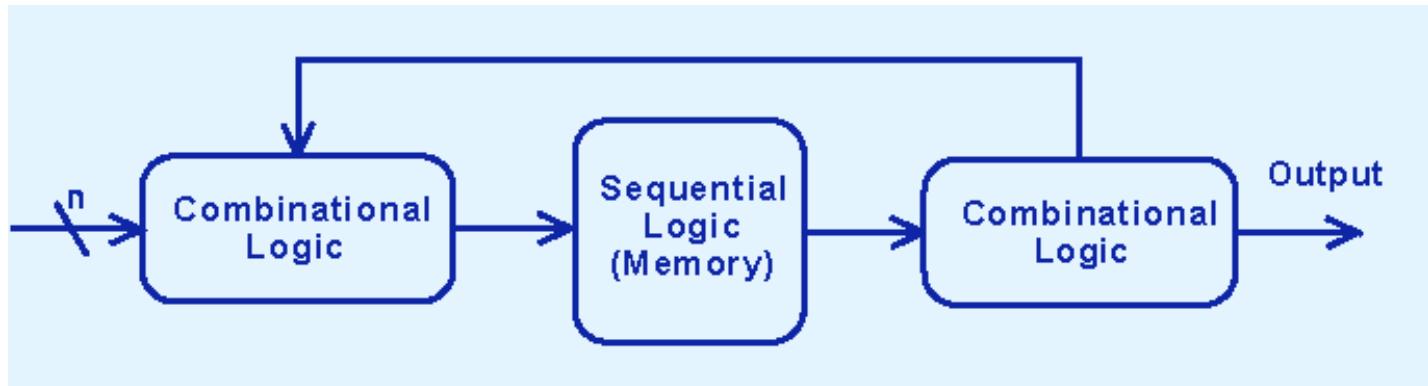
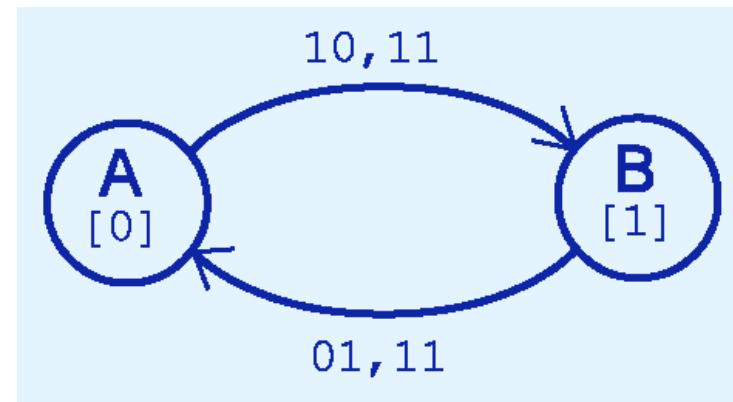


Mealy machine

# Sequential Circuits

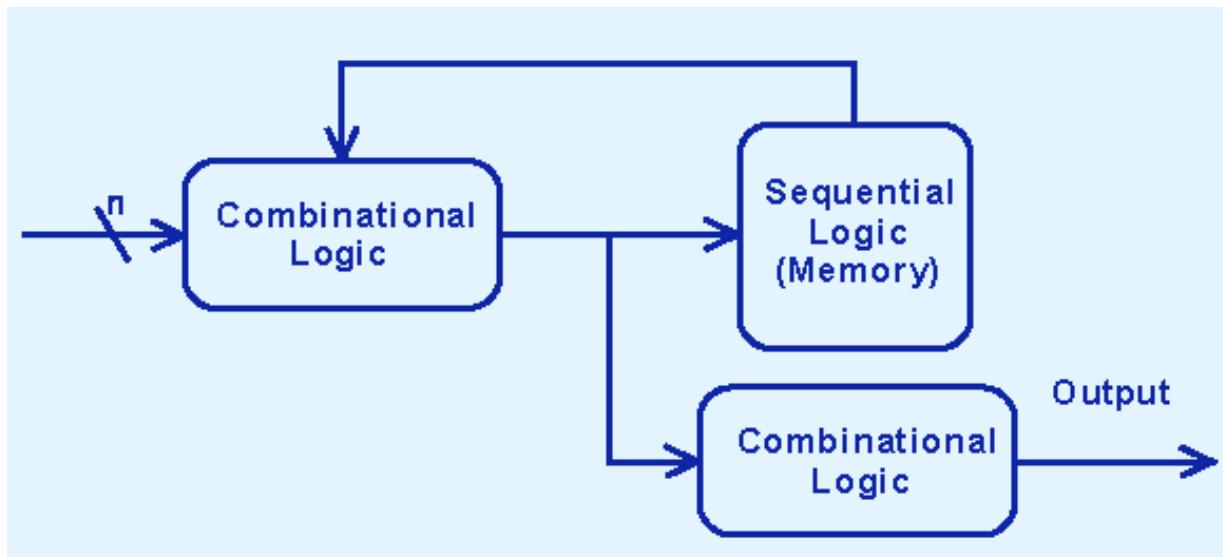
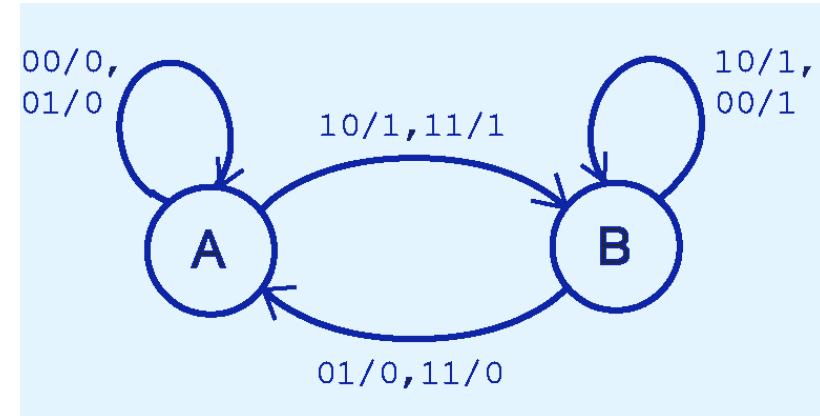
- Behavior of Moore and Mealy machines is identical
- Implementations differ

## Implementation of Moore machine



# Sequential Circuits

## Implementation of Mealy machine

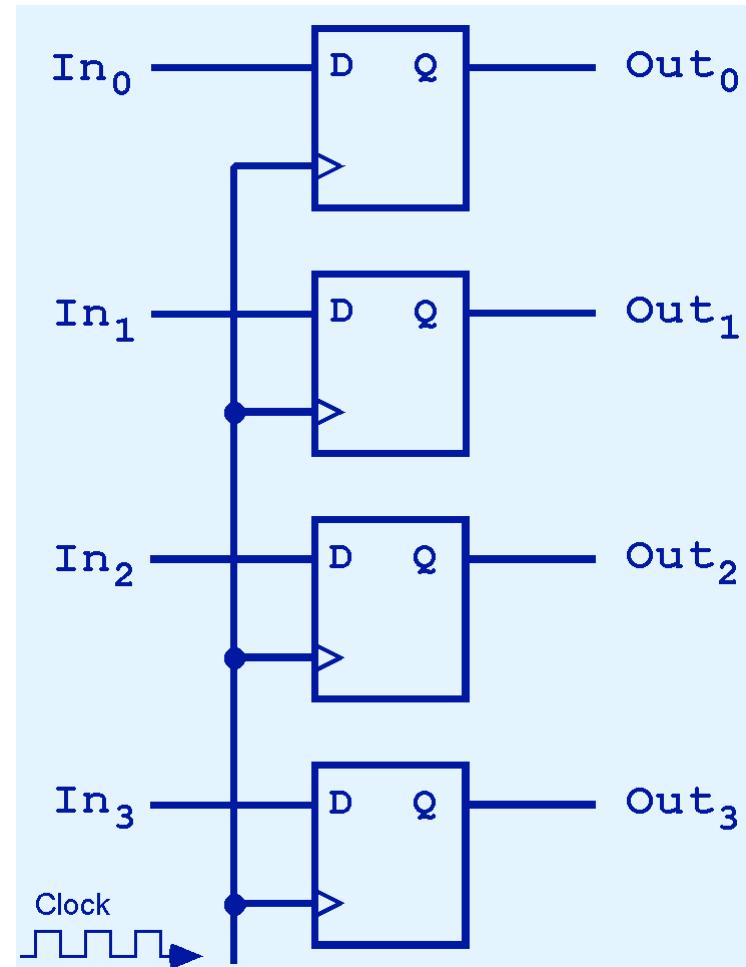


# Sequential Circuits

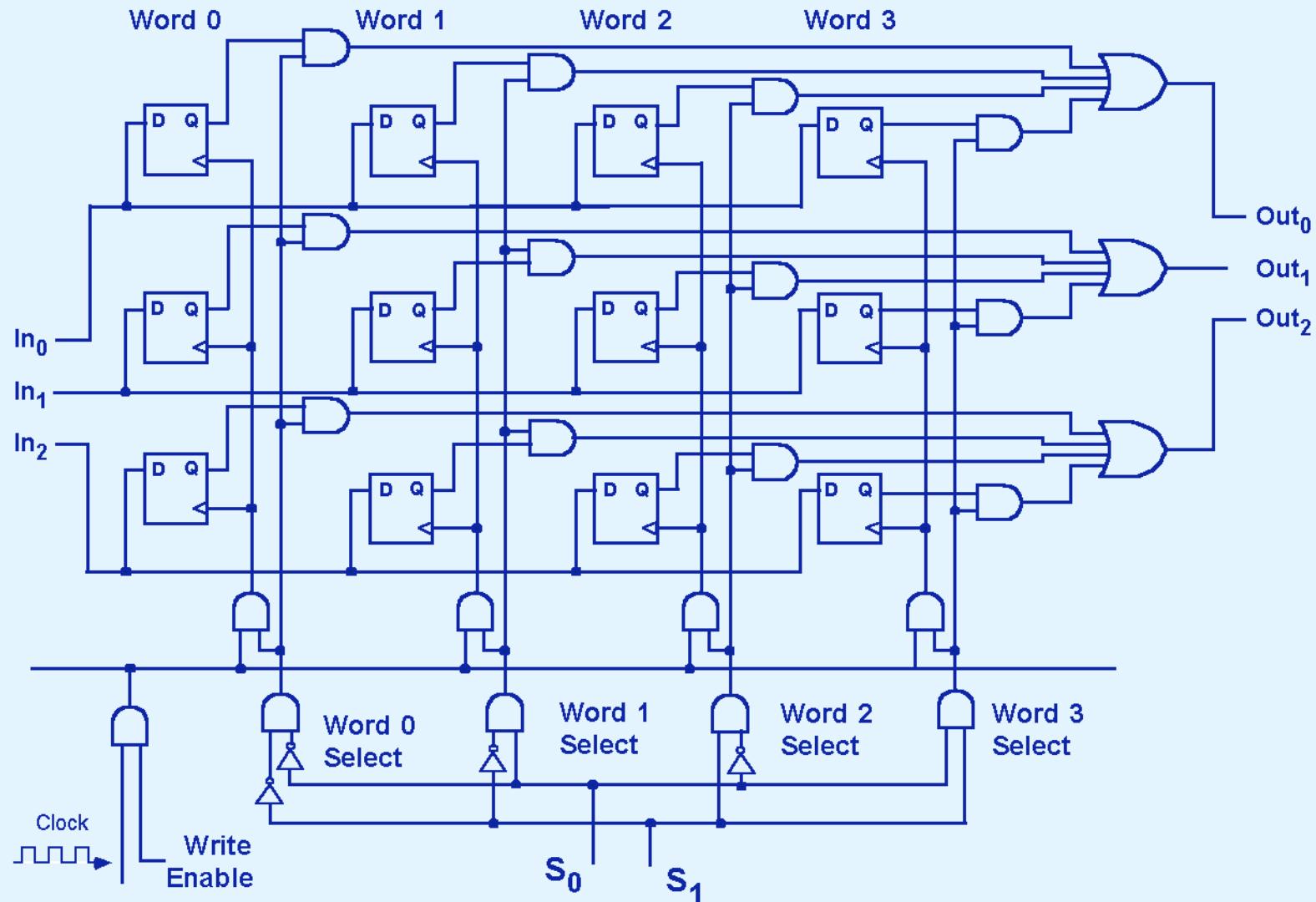
- Sequential circuits are used for “stateful” applications
  
- Stateful application - next state of the machine
  - Depends on the current state of the machine
  - Depends on the input.
  
- Stateful application requires
  - Combinational and
  - Sequential logic

# Sequential Circuits

- 4-bit register consisting of D flip-flops on the right
- Usual block diagram below

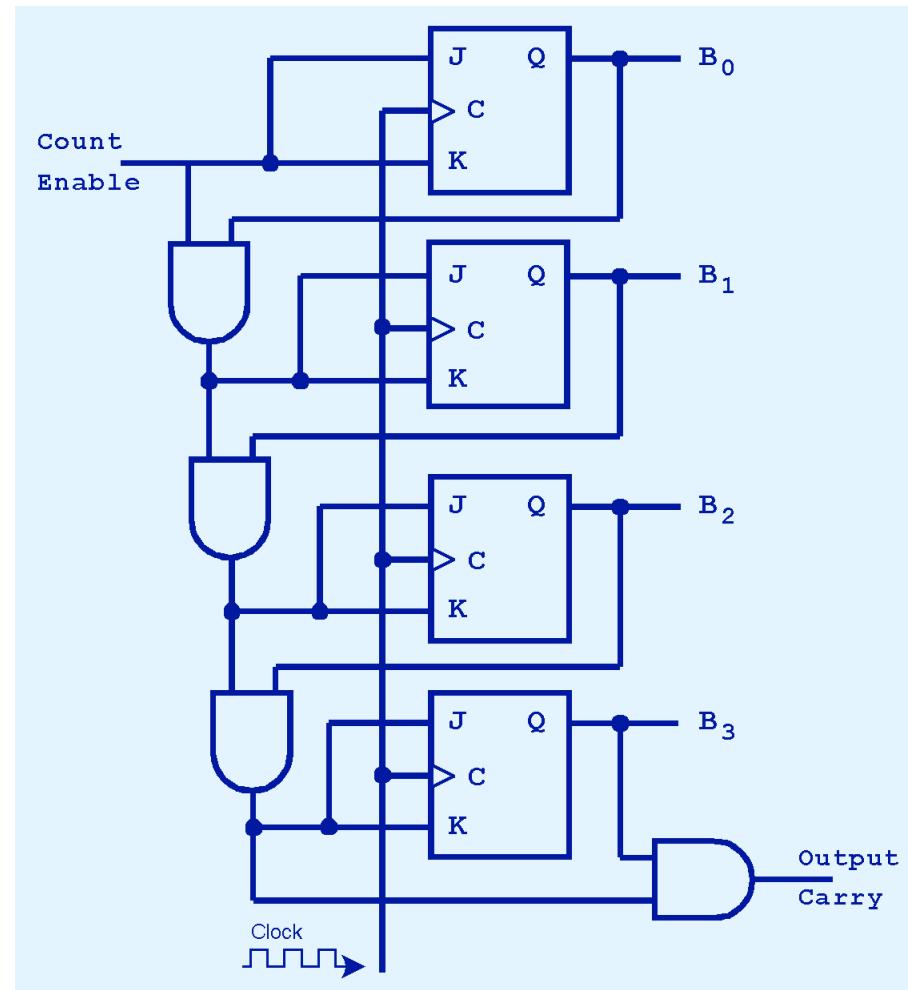


# Sequential Circuits



# Sequential Circuits

- Binary counter
- Low-order bit is complemented at each clock pulse
- Whenever it changes from 0 to 1, the next bit is complemented, and so on through the other flip-flops.



# Finite State Machines

- **Finite State Machine - tool to model sequential logic components**
  - Mealy/Moore machines describe sequential logic
  - Finite State Machines are used to design control units for CPUs
- **Finite state machine consists of**
  - a finite number of states represented by the different values stored in a register
  - a number of external inputs and external outputs
  - boolean functions for the outputs of the machine according to **the current state**
  - boolean functions to determine the next state according to **current state and the current values of the external inputs**

# Finite State Machines

- Design of a sequential circuit
  - More difficult because of "statefullness" of the circuit components.
    - First develop a *finite state machine* (FSM)
      - Consists of *states* and *inputs*
    - State and inputs determine *next state* and possible *outputs*
    - From FSM model, produce *state table*
    - From state table, produce sequential circuit

# Finite State Machines

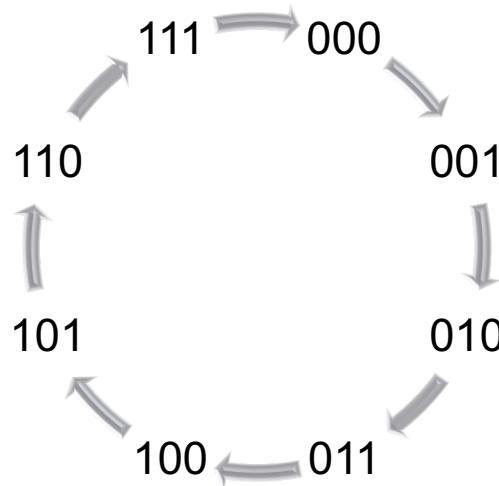
- Finite State Machine is represented by:
  - state transition diagram
    - each node represents a state
    - the edge from state  $s_1$  to  $s_2$  with label for the external input value represents that state transition from  $s_1$  to  $s_2$  under the condition of the inputs described by the label

# Finite State Machines

- **state of truth tables to describe the state transition functions where**
  - the inputs of the tables are the binary values of the current state and the current external inputs
  - the outputs of the functions are the binary values of the *next* state of the transition

# Finite State Machines

- Example: 3-bit up counter - finite state machine with
  - 8 states represented by a 3-bit register (3 D flip-flops) named as Q<sub>2</sub>, Q<sub>1</sub>, Q<sub>0</sub>
  - no external inputs
  - the following state transition diagram

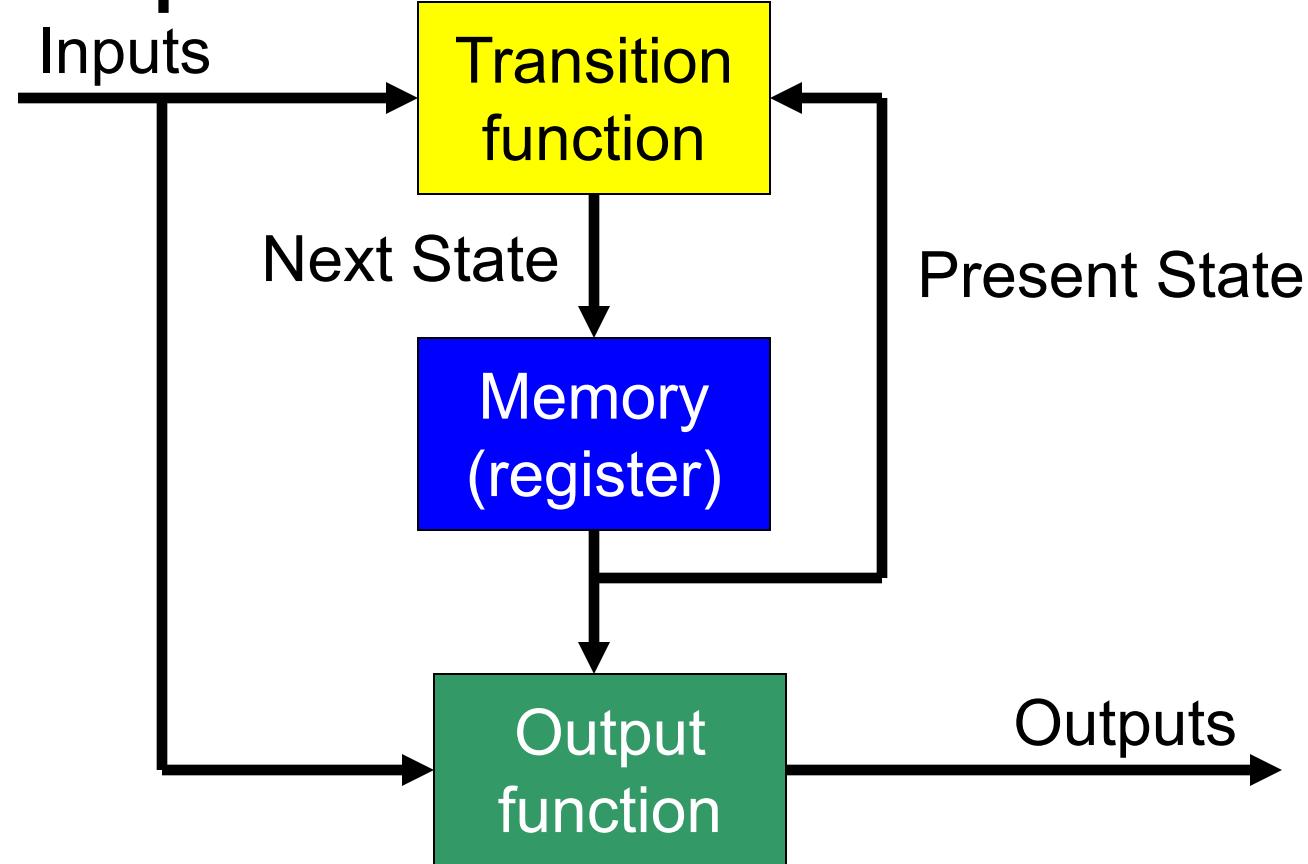


# Finite State Machines

- **Implementation of finite state machine**
  - state transitions - implemented by the combinational logic for the boolean functions which input:
    - the current state and
    - the current value of external inputs
  - use the outputs for the D inputs of the D flip-flops
  - So *next state is reached at beginning of next clock tick*

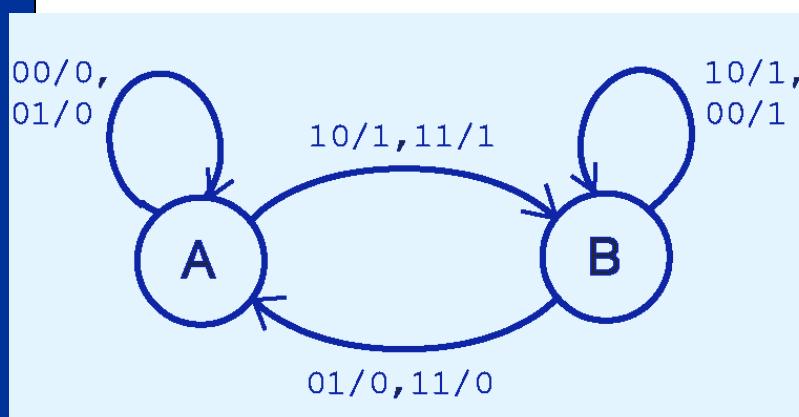
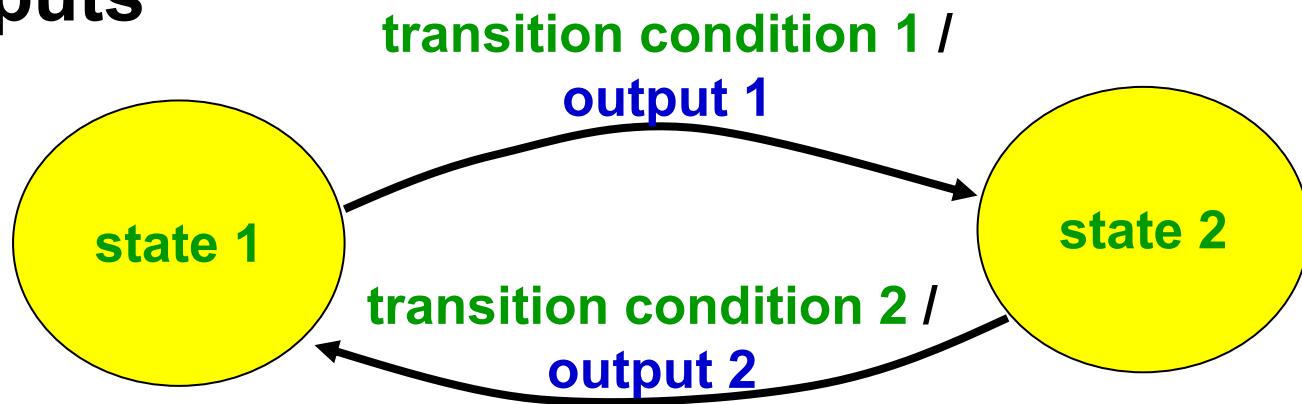
# Mealy FSM

- Output Is a Function of a Present State and Inputs



# Mealy Machine

- Describe Outputs as Concurrent Statements Depending on State and Inputs



J	K	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}(t)$

# JK Flip-flop Example

- Recall when  $J=1$ , sets flip-flop to 1  
when  $K=1$ , sets flip-flop to 0
- State table shown below

State tables for the JK flip-flop

All

Present State	J	K	Next State	Q
A	0	0	A	0
A	0	1	A	0
A	1	0	B	1
A	1	1	B	1
B	0	0	B	1
B	0	1	A	0
B	1	0	B	1
B	1	1	A	0

Simplified

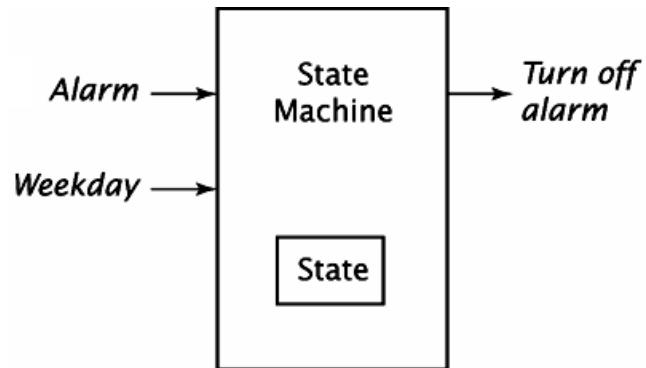
Present State	J	K	Next State	Q
A	0	X	A	0
A	1	X	B	1
B	X	0	B	1
B	X	1	A	0

# Finite State Machines

- Example 1: Alarm clock FSM example

- start with a state table
  - then create state "machine"
    - represent states as circles
    - each transition (action) is represented as an arc
  - FSM arcs must be mutually exclusive: cannot have two arcs with same input conditions (aka ***non-deterministic FSM***).

# FSM Alarm Clock Example



- **Events:**
  - Wake up at fixed time every day
  - Weekends: you don't need alarm, so you wake up, turn off the alarm and resume sleep
- **FSM modeling this chain of events, with:**
  - **Three states:**
    - Asleep
    - Awake but still in bed
    - Awake and up
  - **Inputs:**
    - Alarm
    - Weekday (determines how to react to alarm)
  - **Outputs:**
    - Turn off the alarm

# State tables

Present State	Inputs	Next State	Outputs

- Similar to the truth table
- Doesn't contain the system clock when specifying its transitions
- All the transitions occur on positive edge of the clock unless otherwise stated

# Alarm clock state table

Present State	Alarm	Weekday	Next State	Turn off alarm
Asleep	On	X	Awake in bed	Yes
Awake in bed	Off	Yes	Awake and up	No
Awake in bed	Off	No	Asleep	No

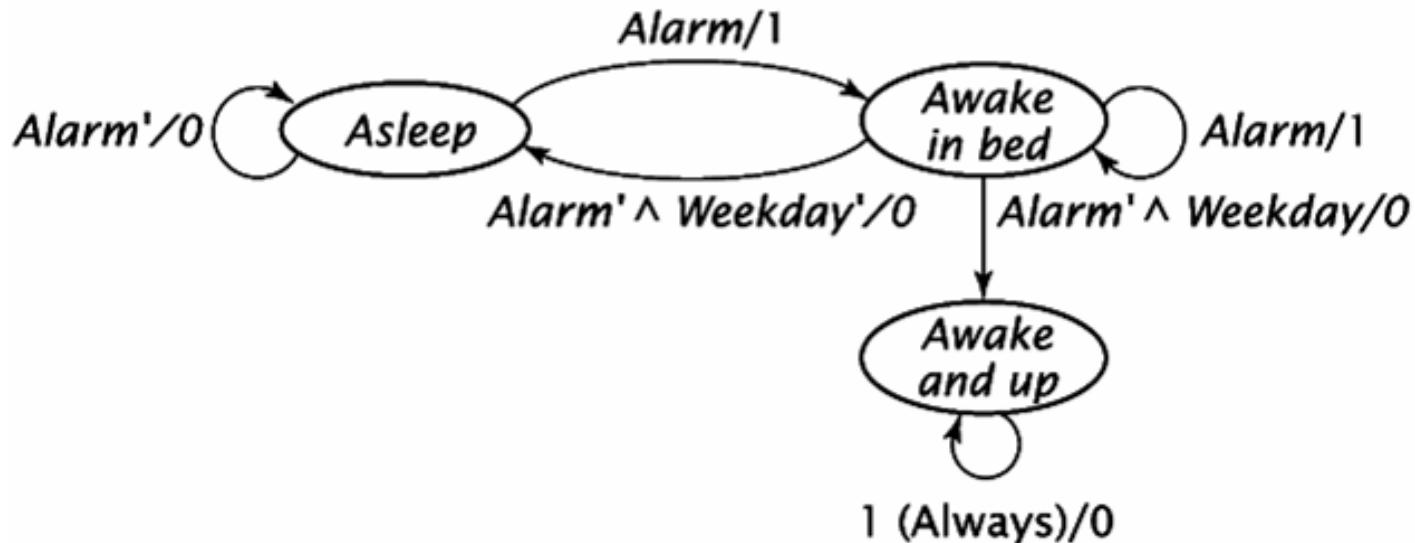
- **Asleep and alarm rings**
  - Transition from being asleep to being awake in bed
  - Turn off the alarm
- **Awake in bed and Weekday**
  - You get up
- **Awake in bed and not Weekday**
  - You go back to sleep
- **Table doesn't cover what you wouldn't do...**
  - (i.e. Asleep and the alarm is off, you remain asleep, etc..)

# Alarm clock state table

Present State	Alarm	Weekday	Next State	Turn off alarm
Asleep	Off	X	Asleep	No
Asleep	On	X	Awake in bed	Yes
Awake in bed	On	X	Awake in bed	Yes
Awake in bed	Off	Yes	Awake and up	No
Awake in bed	Off	No	Asleep	No
Awake and up	X	X	Awake and up	No

- Covers all the cases
  - First row: Asleep and the alarm is off so you remain asleep
  - Last row : Awake and up so you remain awake and up
  - Third row: Awake and in bed and the alarm rings so you remain Awake in bed and turn it off

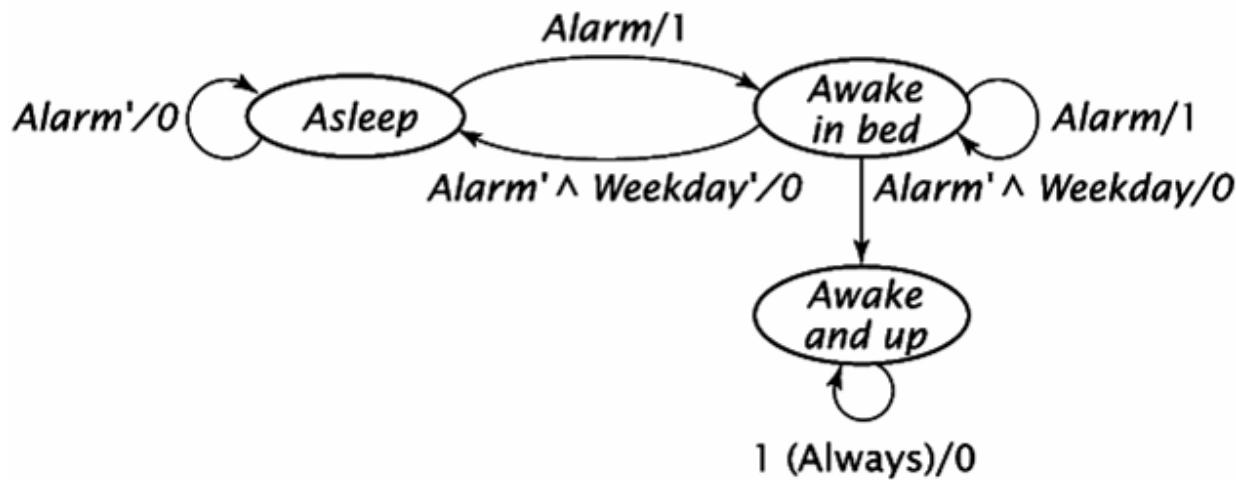
# State diagram



- Graphical representation of the state table
- Each state = circle vertex
- Each row of the state table = directed arc from:
  - present state vertex to the next state vertex
- Outputs are associated with the arcs
  - Output of 1 = “turn off the alarm” is Yes
  - Inputs which are don’t care and inactive outputs are not shown

# State machines

- Mealy machine (used in text):
  - Associates its outputs with the transitions
  - Format of the label of each arc is Inputs/Outputs
  - Self arcs must be shown (because the output values are shown on the arcs)



# Designing from System Specifications

- From a set of specifications, we want to develop
  - State Table
  - State Diagram
- Example 2. Modulo 6 Counter specifications:
  - Counts 000, 001, 010, 011, 100, 101
  - One input U:
    - 1 = increment,
    - 0 = don't increment
  - Three outputs (bits): V2, V1, V0
  - Output bit C is 1 when transitioning from S5 to S0 (remains 1 until first transition)
  - Machine has 6 states, representing six outputs: S0 ... S5

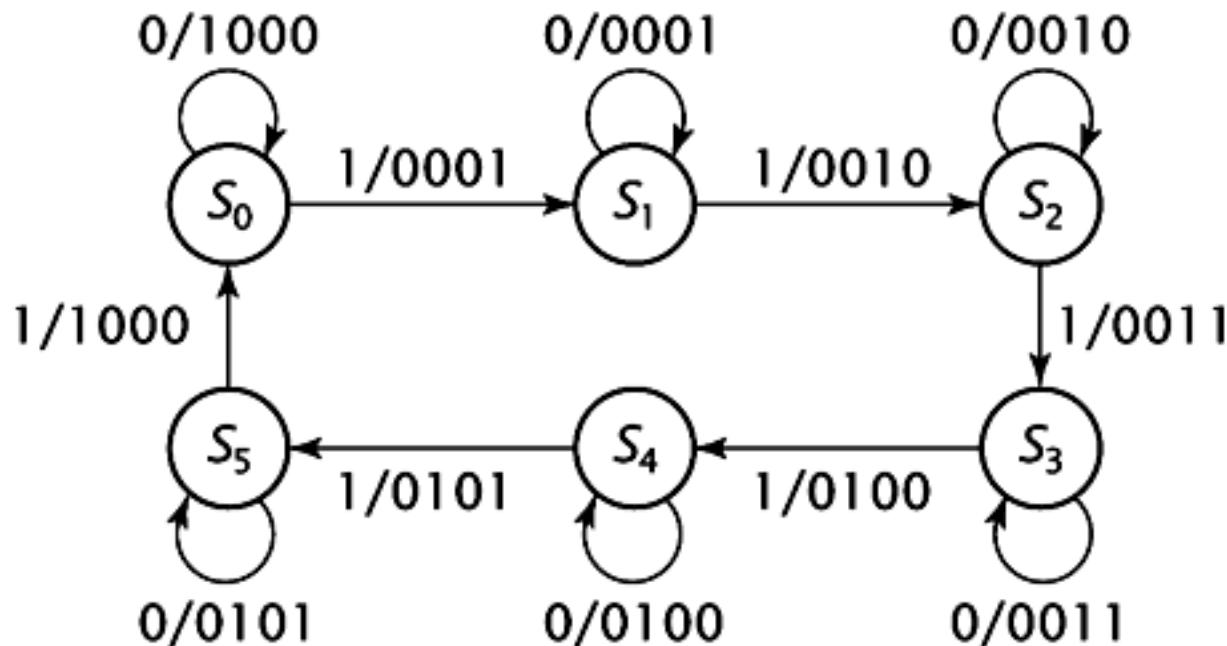
# State Table for modulo 6 counter

State table for the modulo 6 counter

Present State	$U$	Next State	$C$	$V_2 V_1 V_0$
$S_0$	0	$S_0$	1	000
$S_0$	1	$S_1$	0	001
$S_1$	0	$S_1$	0	001
$S_1$	1	$S_2$	0	010
$S_2$	0	$S_2$	0	010
$S_2$	1	$S_3$	0	011
$S_3$	0	$S_3$	0	011
$S_3$	1	$S_4$	0	100
$S_4$	0	$S_4$	0	100
$S_4$	1	$S_5$	0	101
$S_5$	0	$S_5$	0	101
$S_5$	1	$S_0$	1	000

# State Diagrams for Modulo 6 Counter

## ● Mealy Machine



(a)

# Finite State Machines

- Any Circuit with Memory Is a Finite State Machine
    - Even computers can be viewed as huge FSMs
  - Design of FSMs Involves
    - Defining states
    - Defining transitions between states
    - Optimization / minimization
- 
- *Above Approach Is Practical for Small FSMs Only*

# State Encoding

- State Encoding Can Have a Big Influence on Optimality of the FSM Implementation
  - No methods other than checking all possible encodings are known to produce optimal circuit
  - Feasible for small circuits only

# Types of State Encodings

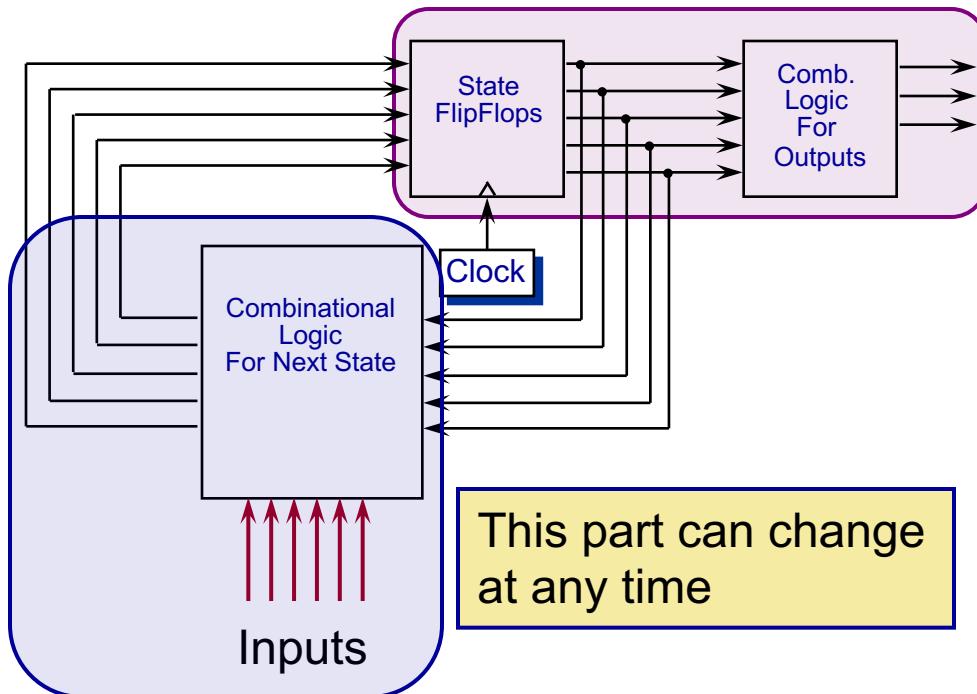
- **Binary (Sequential)** – States Encoded as Consecutive Binary Numbers **(used in the text)**
  - Small number of used flip-flops
  - Potentially complex transition functions leading to slow implementations
- **One-Hot** – Only One Bit Is Active **(used in many small real-world devices)**
  - Number of used flip-flops as big as number of states
  - Simple and fast transition functions

# Types of State Encodings (2)

State	Binary Code	One-Hot Code
S0	000	10000000
S1	001	01000000
S2	010	00100000
S3	011	00010000
S4	100	00001000
S5	101	00000100
S6	110	00000010
S7	111	00000001

# Inputs

- FSMs change state based on clock edges
  - I.e. Rising clock edge clocks all flip flops



This part can  
change only when  
clock “ticks”

**Synchronous Inputs:**  
Change in synch with  
the clock. Obey  
setup and hold time.

**Asynchronous  
Inputs:** Change at  
any time. May violate  
setup and hold times.

# Asynchronous vs. Synchronous Inputs

## ● Asynchronous

- Example: **Elevator pushbuttons**
- Arrive at **any time**
- Usually asserted for **many clock cycles**
- FSM logic must not make any assumptions about input timing

## ● Synchronous

- Example: **Data arriving on a serial line from a computer**
- Arrive **synchronized exactly to a clock**
- One bit of data per **clock cycle**
- FSM can assume that data changes once per **clock cycle**

# Chapter 10:

# Combinatorial Circuits

# Overview

- **Gates, latches, memories**
  - used to design computer
- **Understanding of digital logic**
  - Needed to learn computing systems organization and architecture
- **Two types of digital logic:**
  - Combinatorial logic: output is a function of inputs
  - Sequential logic: output is a complex function of inputs, previous inputs and previous outputs
- **Both are used in circuit design**

# Combinatorial logic

- **Output is dependent on its current inputs**
  - **When certain input values are set**
    - Generates output values corresponding to those input values
  - **When the input values are changed**
    - Outputs are also changed to reflect the changes in the new input values
  - **Previous values of the inputs do not matter**
    - current outputs depend solely on the current inputs

# Sequential logic

- Outputs depend on both the current inputs and on previous inputs and outputs of the circuit
  - Sequential elements have storage elements that record the state of the circuit.
    - State information combined with the inputs generates outputs
    - State and inputs also combine to generate a new state of the circuit
  - Same inputs may generate different outputs and different new states, depending on current state
- Both types of logic are used
  - Sequential logic includes combinatorial logic
  - the reverse is not true

# Boolean Algebra

- Boolean algebra is a mathematical system that can have one of two values
  - Formal logic: “true” and “false”
  - Digital systems:
    - “on” and “off”
    - 1 and 0
    - “high” and “low.”
- Boolean expressions created by performing operations on Boolean variables
  - Common Boolean operators
    - AND, OR, and NOT

# Boolean Algebra

- Boolean operator
  - described using truth table

- AND operator
  - Boolean product

- OR operator
  - Boolean sum

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

# Boolean Algebra

- NOT operation
  - designated by an overbar  
 $\bar{x}$
  - sometimes a prime mark  
 $x'$
  - sometimes an “elbow”  
 $\neg x$

NOT x	
x	$\bar{x}$
0	1
1	0

# Boolean Algebra

- A Boolean function has:
  - At least one Boolean variable
  - At least one Boolean operator
  - At least one input from the set {0,1}
- Output produced is also a member of set {0,1}

# Boolean Algebra

- Truth table for the Boolean function:

$$F(x, y, z) = x\bar{z} + y$$

- Ease evaluation
  - Add extra (shaded) columns to hold evaluations of subparts of function

$$F(x, y, z) = x\bar{z} + y$$

x	y	z	$\bar{z}$	$x\bar{z}$	$x\bar{z}+y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

# Boolean Algebra

- Boolean operations have rules of precedence
- NOT
  - highest priority
- AND
  - Next highest
- OR
  - last
- Extra (shaded) function subparts for highest priorities

$$F(x, y, z) = x\bar{z} + y$$

x	y	z	$\bar{z}$	$x\bar{z}$	$x\bar{z} + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

# Boolean Algebra

- Digital computers contain circuits that implement Boolean functions
- The simpler the Boolean function
  - The smaller the circuit that will result
  - Simpler circuits are:
    - cheaper to build
    - consume less power
    - run faster than complex circuits
- Always reduce Boolean functions to their simplest form
- Boolean identities help to do this

# Boolean Algebra

- Most Boolean identities have an AND (product) form as well as an OR (sum) form
- Here are identities using both forms
- This group is rather intuitive

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$x\bar{x} = 0$	$x + \bar{x} = 1$

# Boolean Algebra

- Second group of Boolean identities should be familiar from algebra:

Identity Name	AND Form	OR Form
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x + (y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$

# Boolean Algebra

- Last group of Boolean identities are the most useful
- If you have studied set theory or formal logic, these laws are also familiar to you

Identity Name	AND Form	OR Form
Absorption Law	$x(x+y) = x$	$x + xy = x$
DeMorgan's Law	$\overline{xy} = \overline{x} + \overline{y}$	$\overline{(x+y)} = \overline{x}\overline{y}$
Double Complement Law		$\overline{\overline{x}} = x$

# Boolean Algebra

- Use Boolean identities to simplify the function:  
as follows:

$$F(X, Y, Z) = (X + Y)(X + \bar{Y})(\bar{X}\bar{Z})$$

$(X + Y)(X + \bar{Y})(\bar{X}\bar{Z})$   
 $(X + Y)(X + \bar{Y})(\bar{X} + Z)$   
 $(XX + X\bar{Y} + XY + Y\bar{Y})(\bar{X} + Z)$   
 $((X + Y\bar{Y}) + X(Y + \bar{Y}))(\bar{X} + Z)$   
 $((X + 0) + X(1))(\bar{X} + Z)$   
 $X(\bar{X} + Z)$   
 $\bar{X}X + XZ$   
 $0 + XZ$   
 $XZ$

Idempotent Law (Rewriting)  
DeMorgan's Law  
Distributive Law  
Commutative & Distributive Laws  
Inverse Law  
Idempotent Law  
Distributive Law  
Inverse Law  
Idempotent Law

# Boolean Algebra

- Sometimes economical to build a circuit using the complement than to implement the function directly
- DeMorgan's law
  - provides way to find complement of Boolean function
- DeMorgan's law states:

$$\overline{(xy)} = \bar{x} + \bar{y} \quad \text{and} \quad \overline{(x+y)} = \bar{x}\bar{y}$$

# DeMorgan's Law

$$(ab)' = a' + b'$$

$$(a+b)' = a'b'$$

- Allows conversion of AND function to equivalent OR function and vice-versa
- May allow simplification of complex functions that will allow simpler design

# Generating the complement of a function using DeMorgan's law

$$(xy' + yz)' = (xy')'(yz)' = (x' + y)(y' + z') = x'y' + x'z' + yy' + yz'$$

(because  $yy' = 0$ ) =>  $(xy' + yz)' = x'y' + x'z' + yz'$

x	y	z	$x'y'$	$x'z'$	$yz'$	$x'y' + y'z' + yz'$
0	0	0	1	1	0	1
0	0	1	1	0	0	1
0	1	0	0	1	1	1
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	1	1
1	1	1	0	0	0	0

# Boolean Algebra

- DeMorgan's law can be extended to any number of variables.
- Replace each variable by its complement and change all ANDs to ORs and all ORs to ANDs.
- Thus, we find the the complement of:

$$F(X, Y, Z) = (XY) + (\bar{X}Z) + (YZ)$$

is:

$$\begin{aligned}\overline{F}(X, Y, Z) &= \overline{(XY) + (\bar{X}Z) + (YZ)} \\ &= \overline{(XY)} \overline{(\bar{X}Z)} \overline{(YZ)} \\ &= (\bar{X} + \bar{Y})(X + \bar{Z})(\bar{Y} + Z)\end{aligned}$$

# Boolean Algebra

- Numerous ways of stating the same Boolean expression
  - “synonymous” forms are *logically equivalent*
  - Logically equivalent expressions have identical truth tables
- To eliminate confusion:
  - Boolean functions are expressed in standardized or canonical form

# Boolean Algebra

- Two canonical forms for Boolean expressions:
  - sum-of-products
  - product-of-sums
- Recall:
  - Boolean product is the AND operation
  - Boolean sum is the OR operation
- Sum-of-products form:
  - ANDed variables are ORed together.
  - For example:
- $F(x, y, z) = xy + xz + yz$
- Product-of-sums form:
  - ORed variables are ANDed together:
  - For example:
- $F(x, y, z) = (x+y)(x+z)(y+z)$

# Boolean Algebra

- Convert a function to sum-of-products form using its truth table
- Note values of the variables that make the function true (=1)
- List the values of the variables that result in a true function value
- Each group of variables is then ORed together

$$F(x, y, z) = x\bar{z} + y$$

x	y	z	$x\bar{z} + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

# Boolean Algebra

- Sum-of-products form for our function is:

$$F(x, y, z) = \bar{x}\bar{y}\bar{z} + \bar{x}yz + x\bar{y}\bar{z} + xy\bar{z} + xyz$$

Note: this function is not in simplest terms. Our aim is only to rewrite our function in canonical sum-of-products form.

$$F(x, y, z) = x\bar{z} + y$$

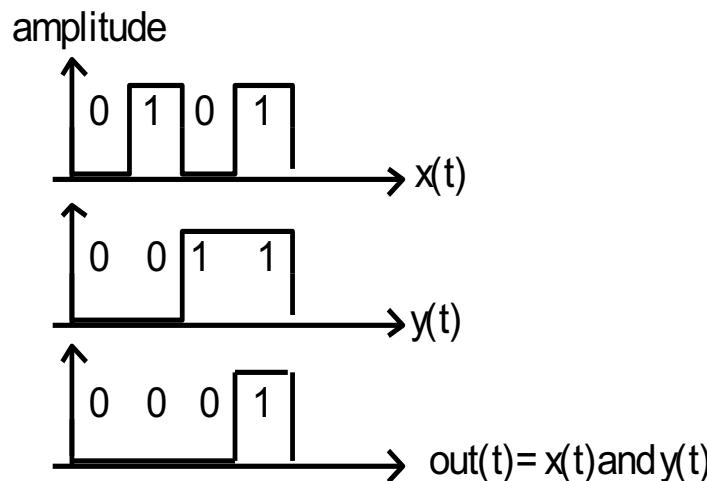
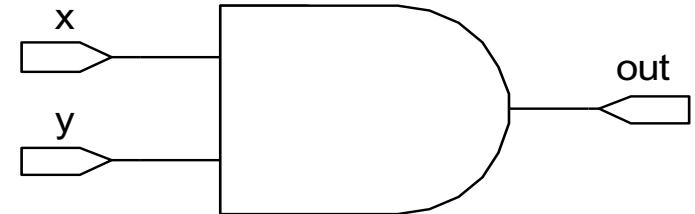
x	y	z	$x\bar{z} + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

# Logic Gates

- Boolean functions in abstract terms so far
- Boolean functions are implemented in digital computer circuits called gates
- Gate - electronic device that produces a result based on two or more input values
  - Gates consist of one to six transistors
  - digital designers think of them as a single unit
  - Integrated circuits contain collections of gates

# AND

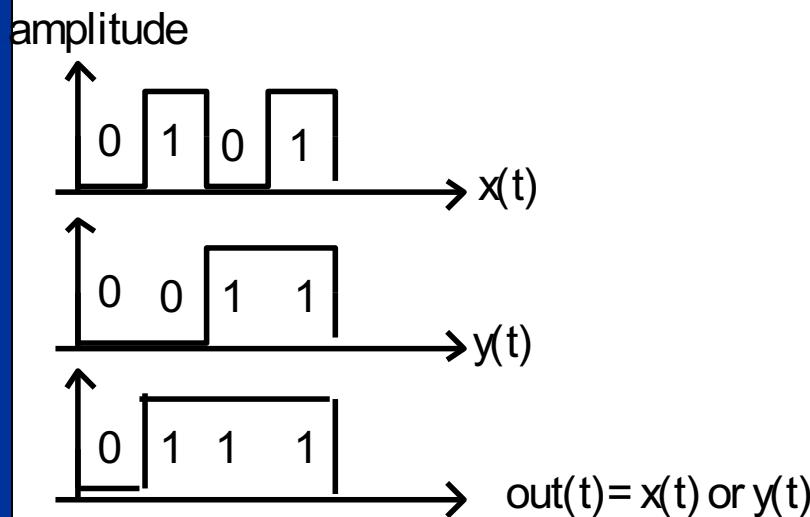
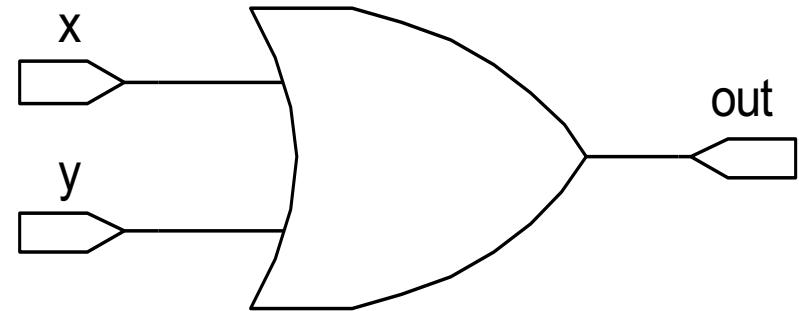
x	y	out = xy
0	0	0
1	0	0
0	1	0
1	1	1



- Output is one if every input has value of 1
- More than two values can be “and-ed” together
- For example  $xyz = 1$  only if  $x=1$ ,  $y=1$  and  $z=1$

# OR

x	y	out = x+y
0	0	0
1	0	1
0	1	1
1	1	1

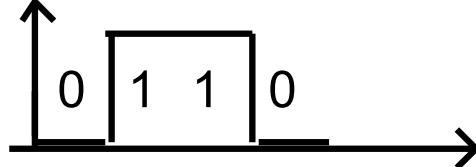
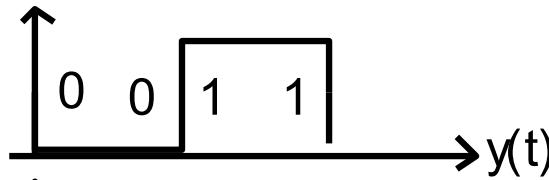
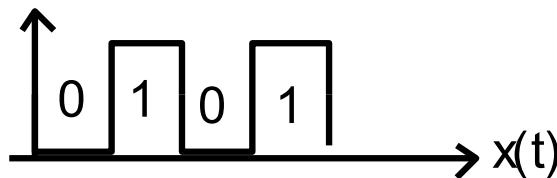


- Output is 1 if at least one input is 1.
- More than two values can be “or-ed” together.
- For example  $x+y+z = 1$  if at least one of the three values is 1.

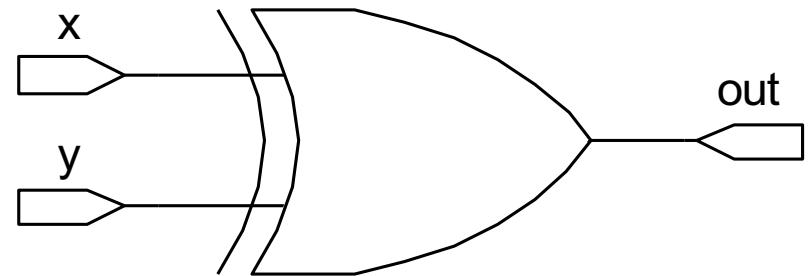
# XOR (Exclusive OR)

x	y	out = $x \oplus y$
0	0	0
1	0	1
0	1	1
1	1	0

amplitude



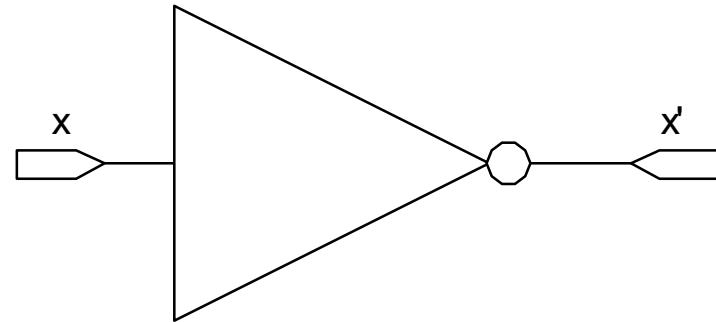
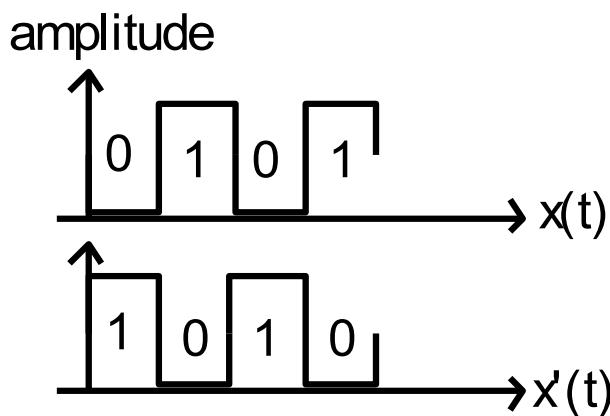
$$out(t) = x(t) \text{ xor } y(t)$$



- The number of inputs that are 1 matter.
- More than two values can be “xored” together.
- General rule: the output is equal to 1 if an odd number of input values are 1 and 0 if an even number of input values are 1.

# NOT

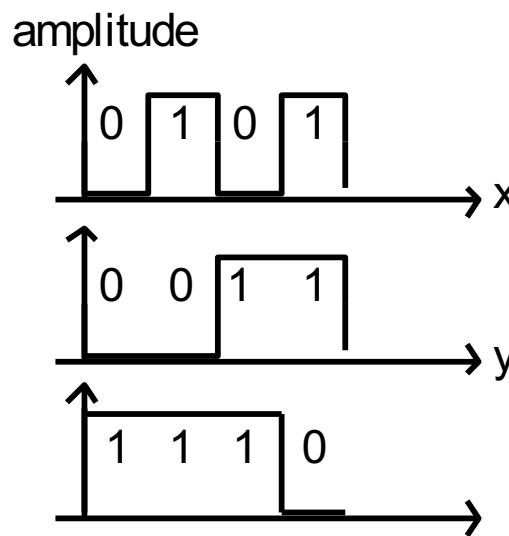
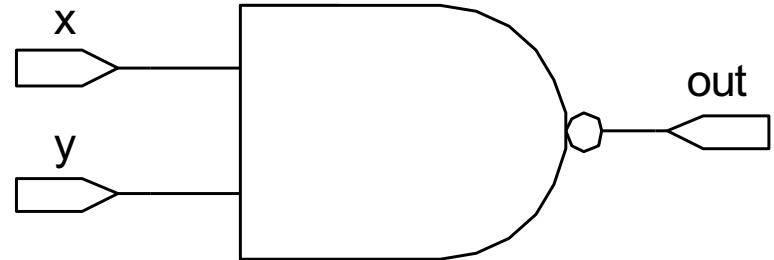
x	x'
0	1
1	0
0	1
1	0



- This function operates on a single Boolean value.
- Its output is the complement of its input.
- An input of 1 produces an output of 0 and an input of 0 produces an output of 1

# NAND

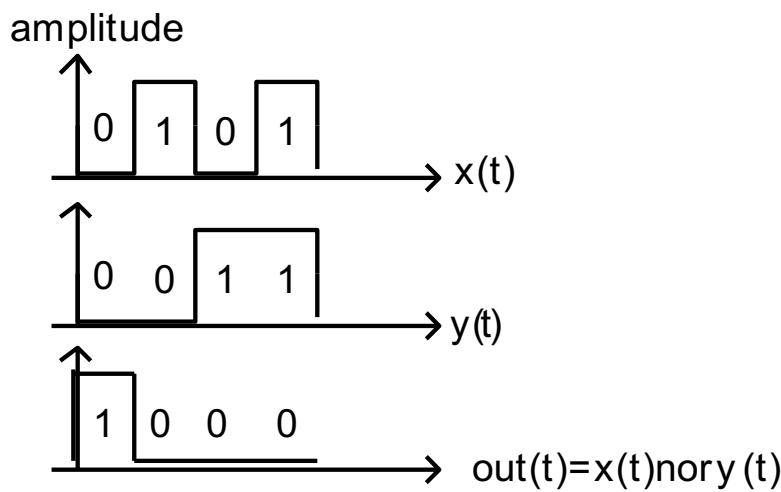
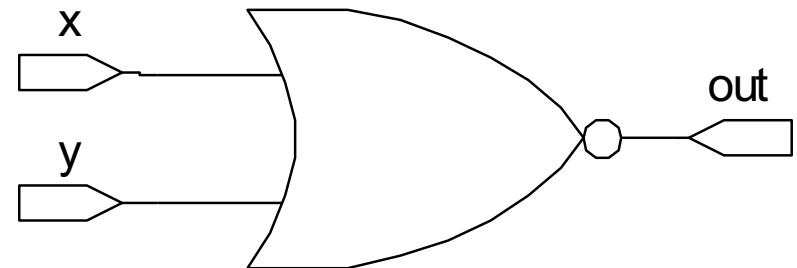
x	y	out = x NAND y
0	0	1
1	0	1
0	1	1
1	1	0



- Output value is the complemented output from an “AND” function.

# NOR

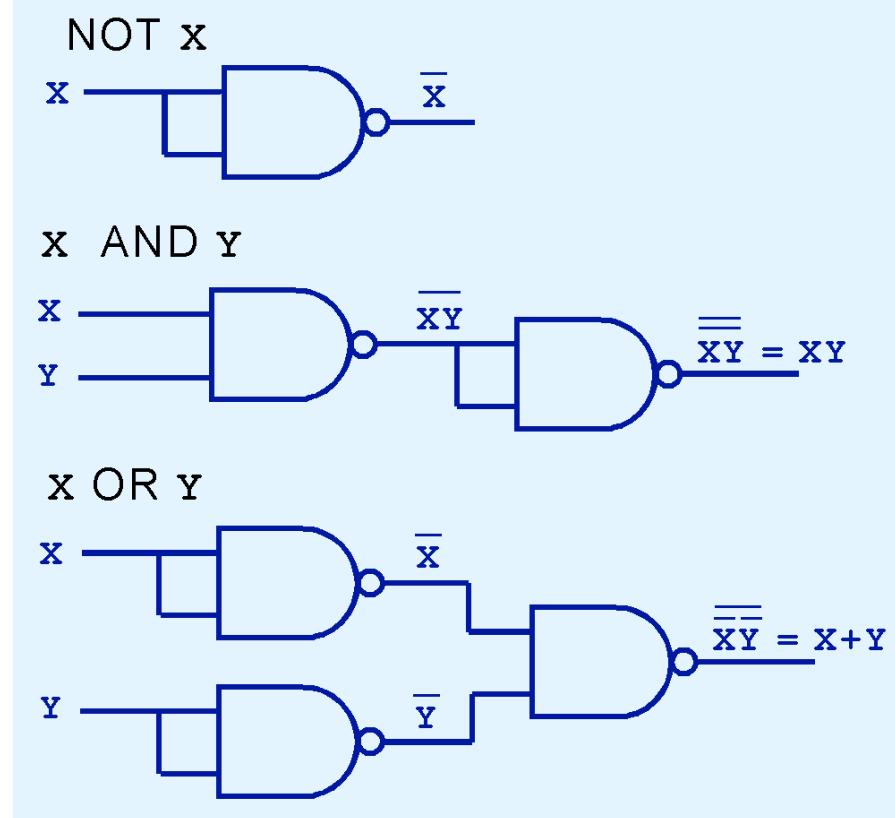
x	y	out = x NOR y
0	0	1
1	0	0
0	1	0
1	1	0



- Output value is the complemented output from an “OR” function.

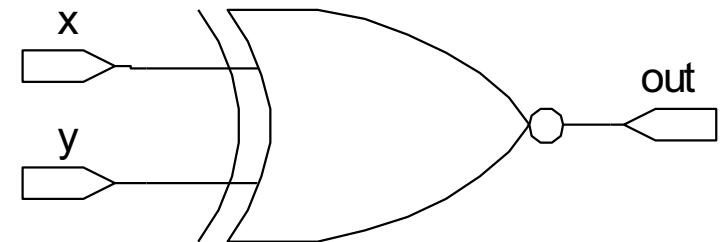
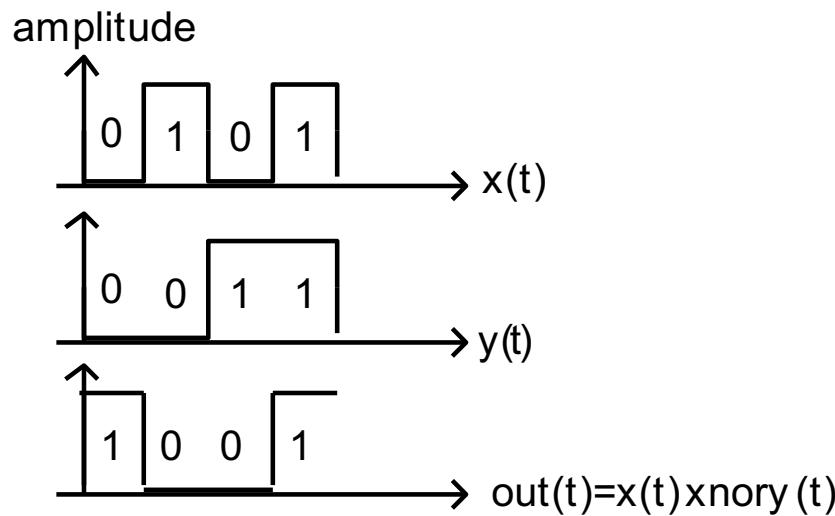
# Logic Gates

- **NAND and NOR**
  - *universal gates*
  - inexpensive to manufacture
  - Any Boolean function can be constructed using only NAND or only NOR gates.



# XNOR

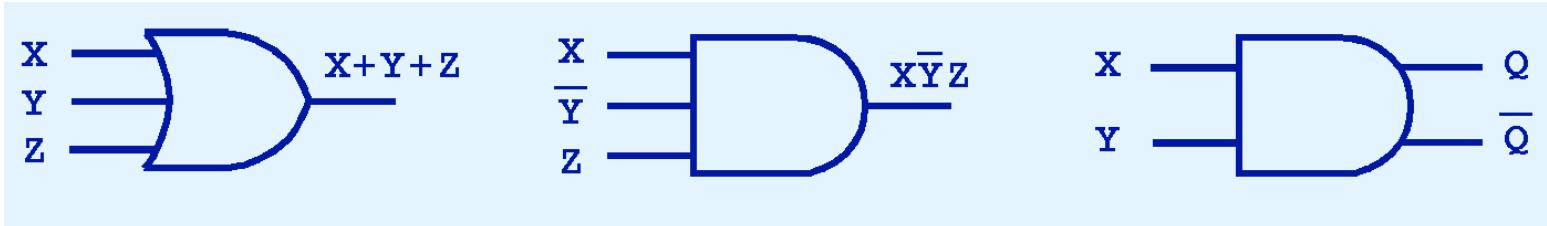
x	y	out = x xnor y
0	0	0
1	0	1
0	1	1
1	1	0



- Output value is the complemented output from an “XOR” function.

# Logic Gates

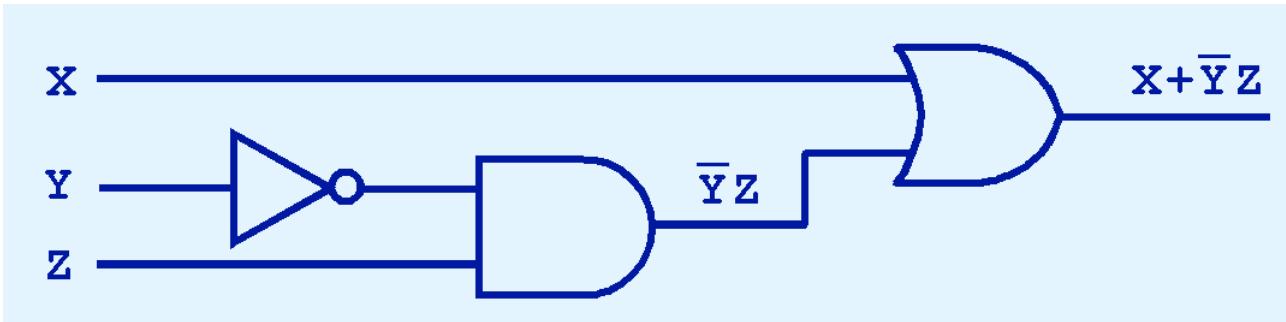
- **Gates:**
  - Can have multiple inputs
  - Can have multiple outputs
  - A second output can provide the complement of the operation



# Digital Components

- Combinations of gates implement Boolean functions
- The circuit below implements the Boolean function:

$$F(X, Y, Z) = X + \bar{Y}Z$$



We simplify our Boolean expressions so that we can create simpler circuits.

# Combinational Circuits

- Designed a circuit that implements the Boolean function:

$$F(X, Y, Z) = X + \bar{Y}Z$$

- Example of a *combinational logic* circuit
- Combinational logic circuits produce a specified output (almost) at the instant when input values are applied

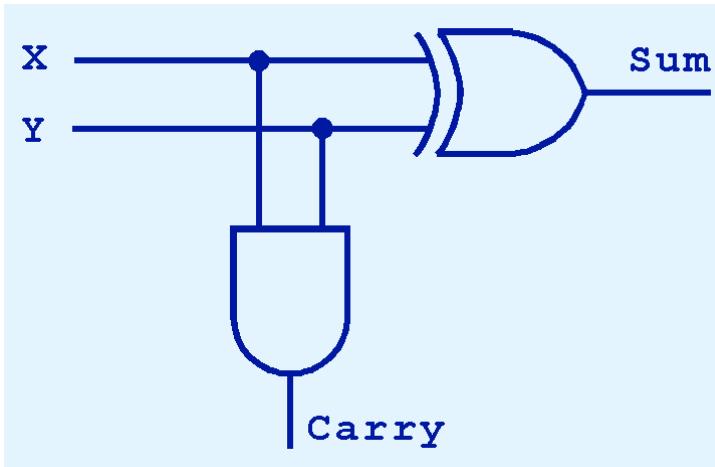
# Combinational Circuits

- Combinational logic circuits provide many useful devices
- HALF ADDER - which finds the sum of two bits
- HALF ADDER construction shown by its truth table

Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

# Combinational Circuits

- Sum found using the XOR operation
- Carry found using the AND operation



Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

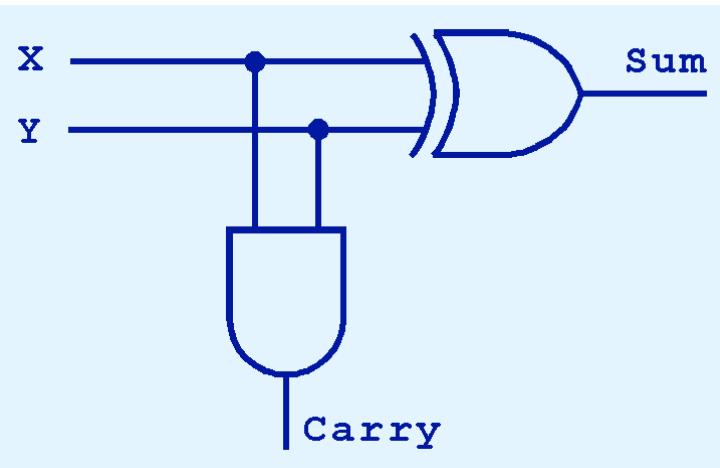
# Combinational Circuits

- Change HALF ADDER to FULL ADDER
  - include gates for processing carry in bit
- Truth table for a full adder is shown at the right

Inputs			Outputs	
Carry			Carry	
X	Y	In	Sum	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

# Combinational Circuits

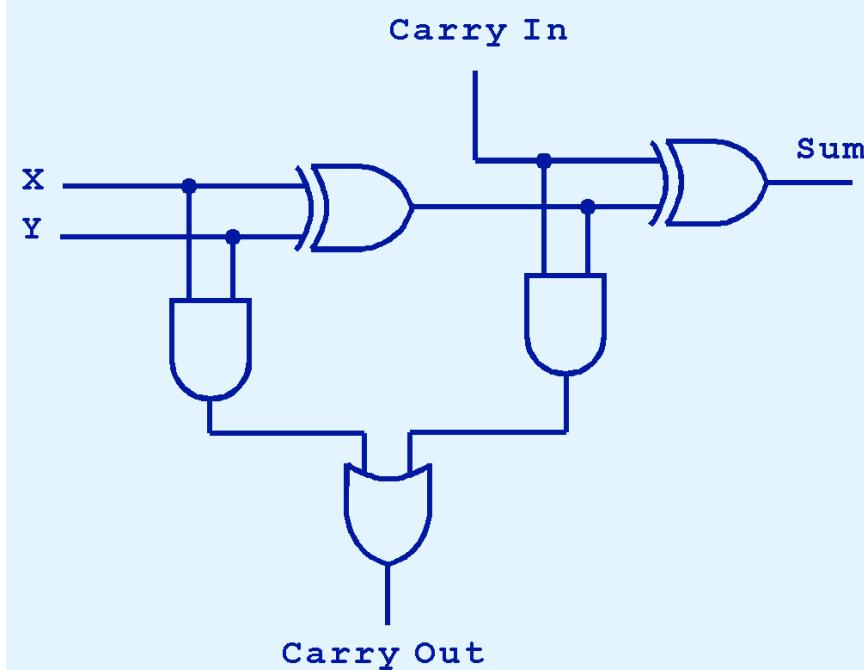
- How can we change the half adder shown below to make it a full adder?



Inputs			Outputs		
		Carry In		Carry Sum	Out
X	Y				
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

# Combinational Circuits

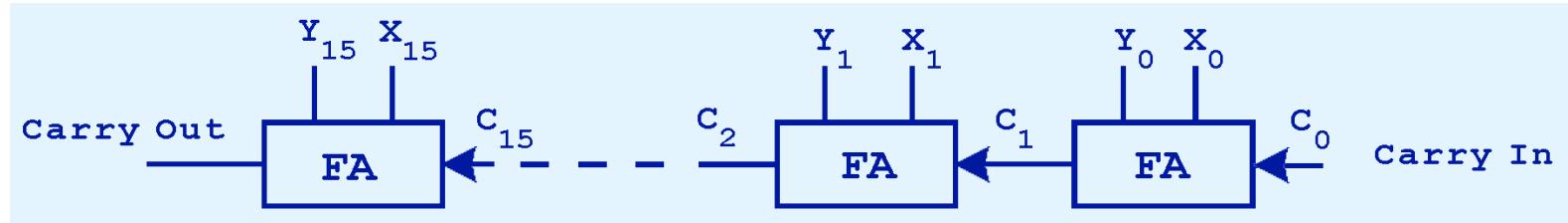
- Here is the completed FULL ADDER



Inputs			Outputs	
X	Y	Carry In	Sum	Carry 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

# Combinational Circuits

- FULL ADDERS can connected in series
- Carry bit “ripples” from one adder to the next
- Called a *ripple-carry adder*



Today's systems employ more efficient adders.

# Combinational Circuits

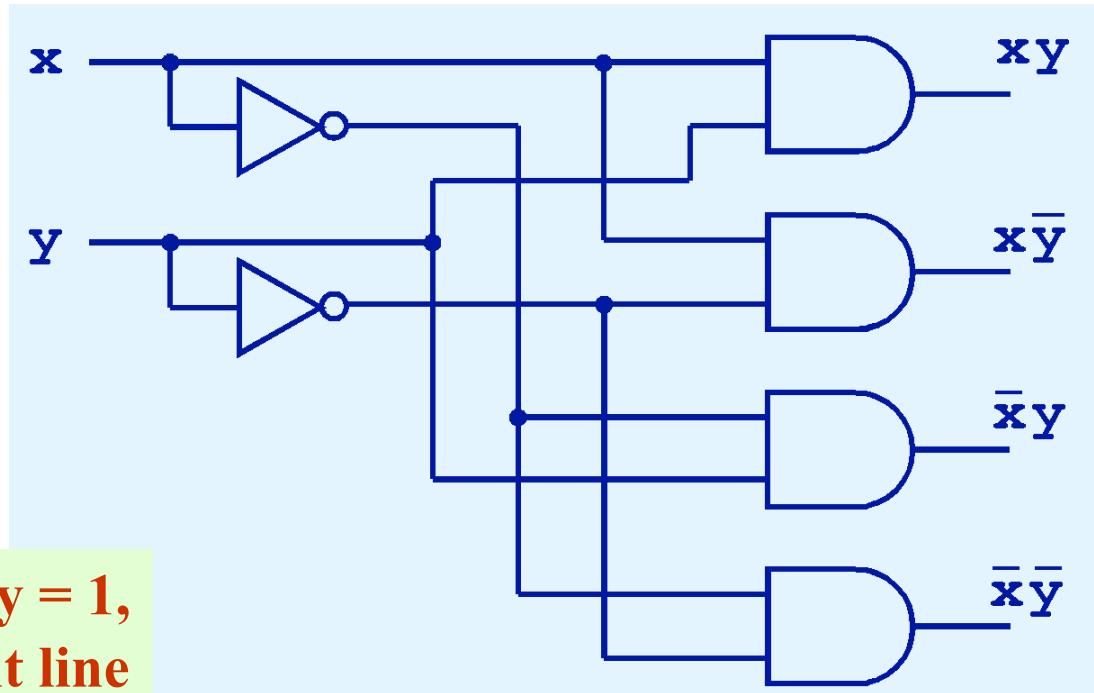
- Decoders –
  - another important type of combinational circuit.
- Useful in selecting a memory location
  - according a binary value placed on the address lines of a memory bus
- Address decoders with  $n$  inputs can select any of  $2^n$  locations

This is a block diagram for a decoder.



# Combinational Circuits

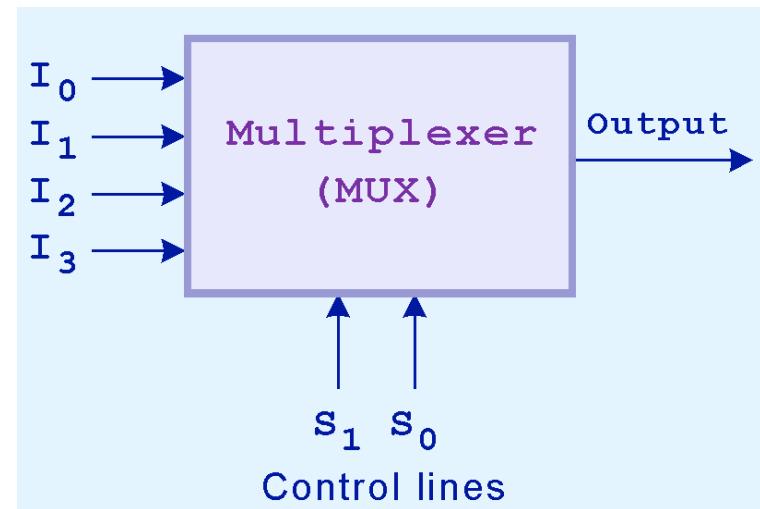
- This is what a 2-to-4 decoder looks like on the inside.



If  $x = 0$  and  $y = 1$ ,  
which output line  
is enabled?

# Combinational Circuits

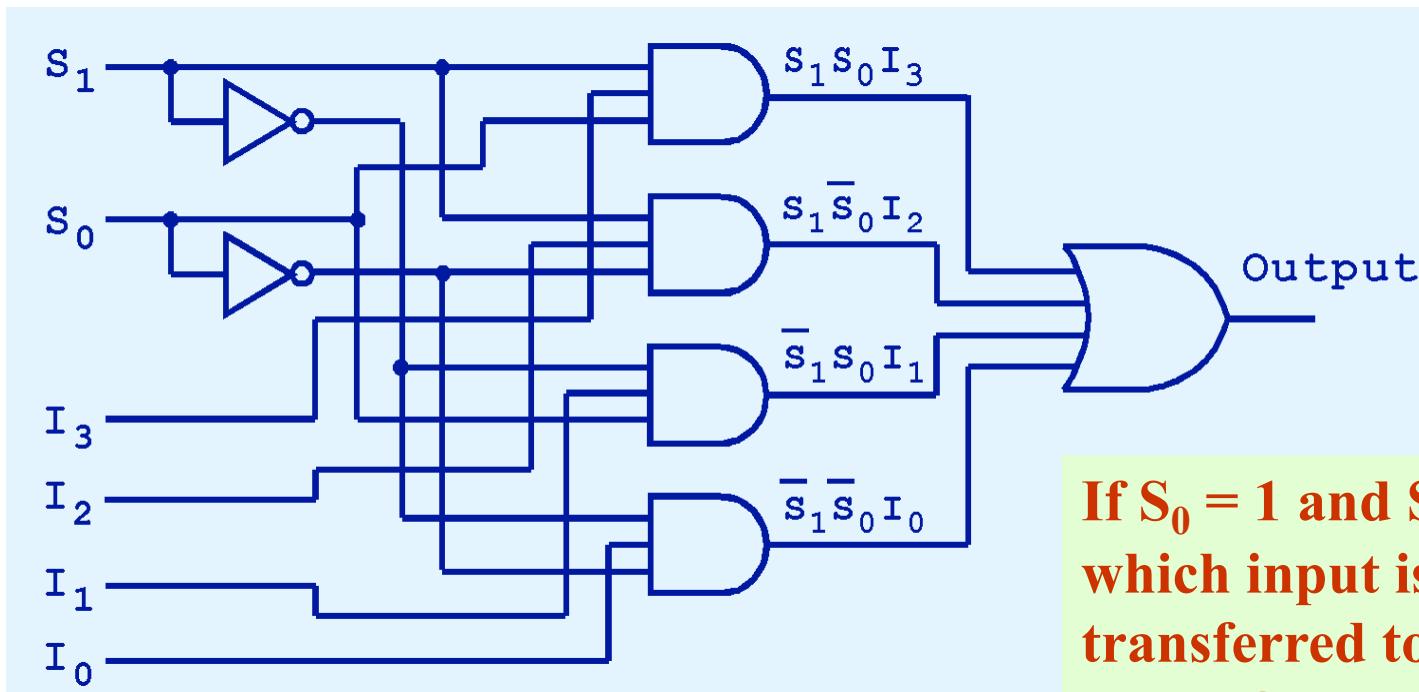
- **Multiplexer**
  - opposite of a decoder
- **Selects a single output from several inputs**
- **Particular input chosen for output**
  - determined by the value of the multiplexer's control lines
- **To select among  $n$  inputs**
  - $\log_2 n$  control lines are needed



This is a block diagram for a multiplexer.

# Combinational Circuits

- This is what a 4-to-1 multiplexer looks like on the inside.

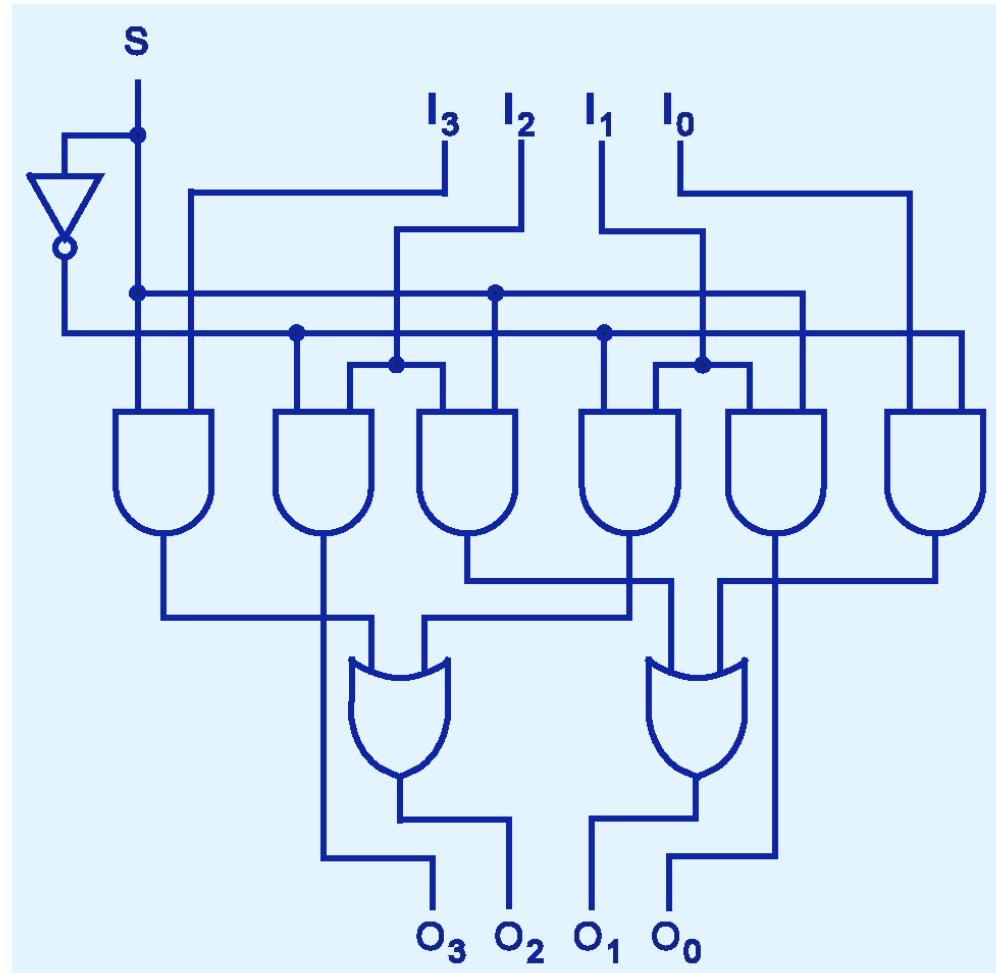


If  $S_0 = 1$  and  $S_1 = 0$ ,  
which input is  
transferred to the  
output?

# Combinational Circuits

- This shifter moves the bits of a nibble one position to the left or right.

If  $S = 0$ , in which direction do the input bits shift?



# **Chapter 06: Compiling to the Assembly Level Part 2 (Sections 6.4-6.5)**

# Objectives

## Learning Objectives

- **Describe how arrays & structures are stored in memory**
- **Compute locations of array cells in memory**
- **Define Jump Tables**
- **Show how Jump Tables and Indexed Addressing can implement a Switch/Case structure**

# Addressing Modes

Programs with Arrays will use the index modes of addressing

Addressing Mode	aaa	Letters	Operand
Immediate	000	i	OprndSpec
Direct	001	d	Mem [OprndSpec]
Indirect	010	n	Mem [Mem [OprndSpec]]
Stack-relative	011	s	Mem [SP + OprndSpec]
Stack-relative deferred	100	sf	Mem [Mem [SP + OprndSpec]]
Indexed	101	x	Mem [OprndSpec + X]
Stack-indexed	110	sx	Mem [SP + OprndSpec + X]
Stack-indexed deferred	111	sxf	Mem [ Mem [SP + OprndSpec] + X]

# The relationship between the operand and the operand specifier for all the addressing modes

- Immediate addressing:

$\text{Oprnd} = \text{OprndSpec}$

- Direct addressing:

$\text{Oprnd} = \text{Mem}[\text{OprndSpec}]$

- Stack-relative addressing:

$\text{Oprnd} = \text{Mem}[\text{SP} + \text{OprndSpec}]$

- Indexed addressing:

$\text{Oprnd} = \text{Mem}[\text{B} + \text{X}]$

# One dimensional arrays

## One-Dimensional Arrays

- Cells are stored in consecutive memory cells
  - Total size = (Size of one cell) \* (Number of cells)
  - Size of one cell depends on the data type
- To reach cell  $i$ ,
  - Start with the base address ( $\text{Cell}[0]$ )
  - Add  $i \cdot \text{DataTypeSize}$  to the base address

# Indexed Addressing and Arrays

- Used for accessing the elements of an array.
- Operand: is memory address that is the sum of the **base** and **index** register.
- Idea:
  - Base register contains the address of the first element of the array
  - Similar to the name of an array in a C program
  - The **index** register contains the **index** of the array

# Arrays

- Allocate the total number of bytes needed in a .BLOCK
- To access element  $v[i]$ :
  - Load  $i$  into the index register
  - Multiply by number of bytes/element
  - Use index addressing ( $x$ )
  - OprndSpec must be the address of the first element of the array
    - i.e., the name of the array if use labels

# Arrays

- Example. Assume v is an integer array

```
int v[4], j=3;
```

...

```
v[j] = 5;
```

- Becomes

```
v:
```

.BLOCK 8

; int v[4]

```
j:
```

.Word 3

...

```
LDA 5
```

```
LDX j, d
```

ASLX

STX v, x

; v is memory addr of

; the array, x is addr mode

# Indexed Addressing and Arrays

- Figure 6.34 (see next slide).
- DECI and DECO use indexed addressing
  - DECI `vector,x` and DECO `vector,x`
  - `vector` is the address of the first element of the vector
  - `x` indicates indexed addressing mode

# Global array example fig. 6.34

## High-Order Language

```
#include <iostream>
using namespace std;

int vector[4];
int j;

int main () {
    for (j = 0; j < 4; j++) {
        cin >> vector[j];
    }
    for (j = 3; j >= 0; j--) {
        cout << j << ' ' << vector[j] << endl;
    }
    return 0;
}
```

# Indexed Addressing and Arrays

```
0000 04000D      BR      main
0003 000000 vector: .BLOCK 8           ;global variable
          000000
          0000
000B 0000 i:     .BLOCK 2           ;global variable
```

# Indexed Addressing and Arrays

```
000D C80000 main:    LDX    0,i          ;for (i = 0
0010 E9000B          STX    i,d
0013 B80004 for1:    CPX    4,i
0016 0E0029          BRGE   endFor1
0019 1D              ASLX
001A 350003          DECT   vector
001D C9000B          LDX    i,d
0020 780001          ADDX   1,i
0023 E9000B          STX    i,d
0026 040013          BR     for1
0029 C80003 endFor1: LDX    3,i          ;for (i = 3
002C E9000B          STX    i,d
002F B80000 for2:    CPX    0,i          initialize the index register
0032 08004E          BRLT   endFor2
0035 39000B          DECO   i,d
0038 500020          CHARO  ',',i       check loop condition
003B 1D              ASLX
003C 3D0003          DECO   vector
003F 50000A          CHARO  '\n',i
0042 C9000B          LDX    i,d
0045 880001          SUBX   1,i
0048 E9000B          STX    i,d
004B 04002F          BR     for2
004E 00      endFor2: STOP
004F .END

```

initialize the index register

: an integer is two bytes

multiply index register by two

input with indexed addressing

increment the index

initialize the index register

check loop condition

, count <= 1

; << '

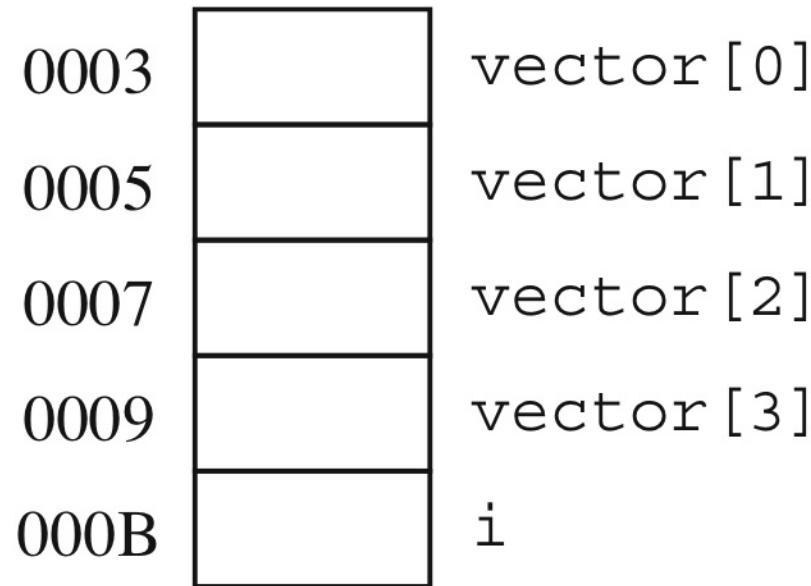
; an integer is two bytes

multiply index register by two

output with indexed addressing

decrement the index

# Accessing an array at Level HOL6 and Level Asmb5



The vector in memory

# Local Arrays

- Local arrays are allocated on the run-time stack
- Stack-indexed addressing
  - Oprnd = Mem[SP + OprndSpec + X]
  - OprndSpec must be the **address** of the **first element** of the array.
  - use letters **sx**

# Local Arrays

- Method:
  - The program allocates the appropriate number of bytes for the array on the stack using SUBSP
  - Deallocate the array using ADDSP
  - To access an element:
    - Load i into the index register
    - Multiply by number of bytes/element
    - Use stack-index addressing (**sx**)

# Local Arrays

- Example. Assume v is a local integer array

v[i] = 5;

- Becomes

LDA 5

LDX i, s ; assume that i is on  
stack

ASLX

STX v, sx ; v is memory addr of  
; the array, x is addr  
mode

# Local Arrays Example fig 6.36

## High-Order Language

```
#include <iostream>
using namespace std;

int main () {
    int vector[4];
    int j;
    for (j = 0; j < 4; j++) {
        cin >> vector[j];
    }
    for (j = 3; j >= 0; j--) {
        cout << j << ' ' << vector[j] << endl;
    }
    return 0;
}
```

# Local Arrays Example

```
0000 040003          BR      main
;
;***** main ()
vector: .EQUATE 2           ;local variable
i:     .EQUATE 0           ;local variable
0003 68000A main:         SUBSP  10,i
0006 C80000              LDX    0,i
0009 EB0000              STX    i,s
000C B80004 for1:         CPX    4,i      ;  i < 4
000F 0E0022              BRGE   endFor1
0012 1D                 ASLX
0013 360002              DECI   vector,sx ;  an integer is two bytes
0016 CB0000              LDX    i,s      ;  cin >> vector[i]
0019 780001              ADDX   1,i
001C EB0000              STX    i,s
001F 04000C              BR     for1
;
```

The assembly code shows the implementation of a local array 'vector' and its index 'i'. The stack frame is established at address 10. The local variables are stored on the stack.

A red oval highlights the stack pointer setup and the first assignment to the local variable 'vector'.

A yellow box contains the annotation: "Store local variables on stack".

A red oval highlights the assignment of the input value to the local variable 'vector'.

A yellow box contains the annotation: "Input using stack-indexed address".

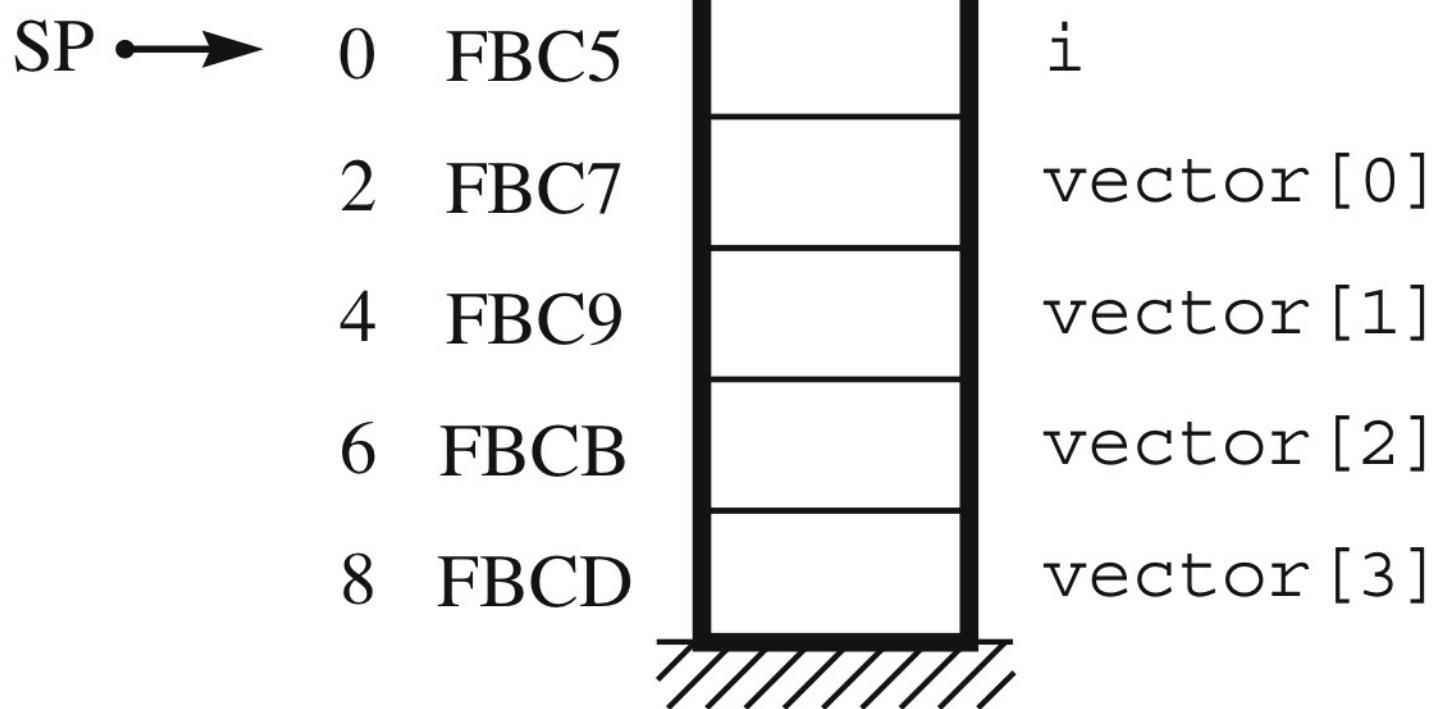
# Local Arrays Example

```
0022 C80003 endFor1: LDX    3,i          ;for (i = 3
0025 EB0000      STX    i,s
0028 B80000 for2:   CPX    0,i          ;  i >= 0
002B 080047      BRLT   endFor2
002E 3B0000      DECO   i,s          ;  cout << i
0031 500020      CHARO  ' ',i        ;      << ' '
0034 1D          ASLX
0035 3E0002      DECO   vector,sx    ;      << vector[i]
0038 50000A      CHARO  '\n',i
003B CB0000      LDX    i,s
003E 880001      SUBX   1,i
0041 EB0000      STX    i,s
0044 040028      BR     for2
0047 60000A endFor2: ADDSP  10,i
004A 00          STOP
004B .END
```

Output using stack-indexed address

Deallocate the array & index

# Local Arrays



The array on the stack

# Indexed Addressing and Arrays

- **Arrays as parameters**
  - In C arrays are passed as pointers
  - In C++ arrays are passed by reference
  - Pass array name by placing its address on the stack
- **Problem:**
  - To use indexed addressing need to use the memory address of the first element
  - How do we do this on the stack?

# Indexed Addressing and Arrays

- **Stack-indexed deferred addressing**
  - $\text{Oprnd} = \text{Mem}[\text{Mem}[\text{SP} + \text{OprndSpec}] + X]$
  - Letters: sxf
- **Use:**
  - **Store the address of the first element of the array on the stack in the appropriate place**
    - use MOVSPA, ADDA (with the offset and immediate addressing) and STA (with stack relative addressing).
    - In the subroutine, load the index variable from the stack into the index register
    - Multiply by the number of bytes per cell
    - Use sxf addressing

# Array Storage Global Vs. Local

Example: `int nums [4];`

Case 1: Global array  
starting at Address 100

Case 2: Local array  
at the top of the stack

Global array	Address
Nums[0]	100
Nums[1]	102
Nums[2]	104
Nums[3]	106

Local array	Address
Nums[0]	SP + 0
Nums[1]	SP + 2
Nums[2]	SP + 4
Nums[3]	SP + 6

# Pep/8 Array code Global Vs Local

## Sample Pep/8 Code for Arrays

	Allocation Example: <i>Declare Array Arr[N]</i>	Access Example: <i>LDA Cell Arr[i]</i>
Global array Static memory allocation	Compiler computes $B = N \times S$ , then <b>Arr: .BLOCK B</b>	<b>LDX i</b> <b>MULTX size</b> <b>LDA Arr,x</b> <b>; index mode</b>
Local array Stack memory allocation	Compiler computes $B = N \times S$ , then <b>Arr: .EQUATE ?</b> <b>SUBSP B,i</b>	<b>LDX i</b> <b>MULTX size</b> <b>LDA Arr,sx</b> <b>; stack index mode</b>

- $N$  = number of cells in the array
- $S$  = bytes per cell (depends on the data type)

# The Switch Statement

- Can treat main memory as an array of bytes.
- Indexes of the array correspond to byte addresses
- Can translate the switch statement as an array of addresses
- This array is called a **jump table**.

# .ADDRESS

- Used to generate code representing the address of a symbol

- Syntax:

**.ADDRESS symbol**

- Effect:

- place the address represented by the symbol into the machine language program at that point

# .ADDRESS

- Example:

- assume Case0 is a label for address 0029

0013 .ADDRESS Case0

...

0029 case0: STRO msg0,d

- In the machine code at 0013 will see

0013 0029

# The Switch Statement

- To implement a Switch statement:
  - Create a jump table using .ADDRSS

0013	001B	guessJT: .ADDRSS case0
0015	0021	.ADDRSS case1
0017	0027	.ADDRSS case2
0019	002D	.ADDRSS case3

- Use LDX to load the index register with the switch value
- Insert a ASLX
  - An address takes two bytes
- Execute a BR with indexed addressing

# Dynamic Memory Allocation

- Programs can request memory allocation dynamically
- Done in conjunction with the OS
- Dynamic memory is allocated from the *heap*

# Dynamic Memory Allocation fig 6.41

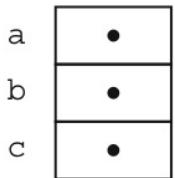
## High-Order Language

```
#include <iostream>
using namespace std;

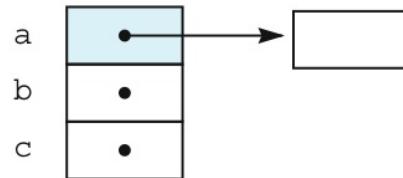
int *a, *b, *c;

int main () {
    a = new int;
    *a = 5;
    b = new int;
    *b = 3;
    c = a;
    a = b;
    *a = 2 + *c;
    cout << "*a = " << *a << endl;
    cout << "*b = " << *b << endl;
    cout << "*c = " << *c << endl;
    return 0;
}
```

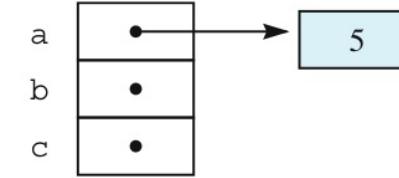
# Dynamic Memory Allocation



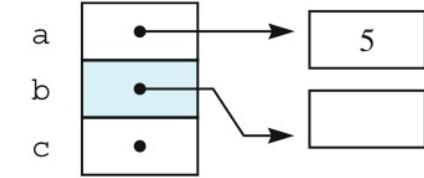
(a) Initial state.



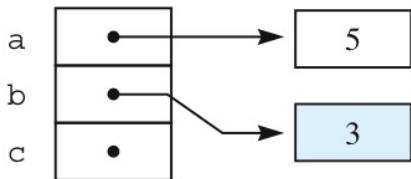
(b) `a = new int;`



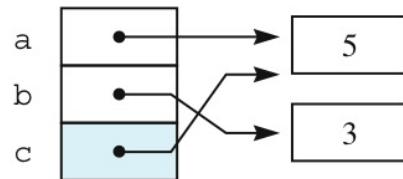
(c) `*a = 5;`



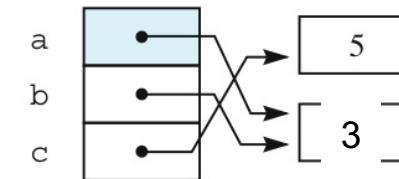
(d) `b = new int;`



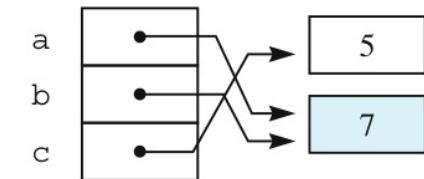
(e) `c = a`



(f) `c = a`



(g) `a = b;`



(h) `*a = 2 + *c;`

# Dynamic Memory Allocation

- Dynamic memory uses a *heap* which is allocated to the process by the OS
- Process can request memory and *free* memory.
- Memory management algorithms must be able to keep track of memory.
- This code is part of the OS.

# Dynamic Memory Allocation

- Pep/8 memory model simplifies the heap
- Pep/8 memory model:
  - Heap is allocated in main memory at the end of the application program
  - New (or malloc) operator allocates storage from the heap
  - Heap grows downward
  - Once memory is allocated can never be deallocated

# Dynamic Memory Allocation

- Pep/8 memory model
  - Compiler puts in code to allocate/manage heap
  - Compiler includes the *new* operator
  - This code/memory is placed at the bottom of the application program
  - See next slide

# Dynamic Memory Allocation

- Pep/8 memory model - How the heap works
  - Variable/Symbol hpPtr contains the address of the first free byte in the heap
  - hpPtr is initialized to the first byte in the heap:
    - hpPtr: .ADDRESS heap
    - Heap: .BLOCK 1
  - Note that since this code occurs at the end of the application, none of the following memory is used by the program
  - Unfortunately, the stack could grow too big and clash with the heap

# Dynamic Memory Allocation

- Pep/8 memory model - How the heap works
  - When a program allocates memory with the **new** command, it must
    - Load the **number** of needed bytes into the **accumulator**
    - Then calls **new**

```
LDA    2,I  
CALL   new
```

# Dynamic Memory Allocation

- Pep/8 memory model - How the heap works
  - New will load the address in the heap pointer (hpPtr) into the index register
    - This register will be used by the application for indirect addressing

**LDX hpPtr,d ; hpPtr contains the  
address ; of the first unused byte  
in the ; heap**

# Dynamic Memory Allocation

- Pep/8 memory model - How the heap works
  - Then new allocates the needed number of bytes
    - Adds the number of bytes needed (which are in the accumulator) to the current top-of-heap value (stored in hpPtr)
    - Then stores the result into the hpPtr location.
    - hpPtr now contains the address of the next unused byte in the heap

**ADDA**      **hpPtr,d**  
**STA**        **hpPtr,d**

# Dynamic Memory Allocation

## ● Pep/8 memory model

### – Compiler generated memory/func

When *new* is called first load the value of the first unused heap byte into the **index** register.

Then add the value in the accumulator (ie, the number of bytes to be allocated) to the current heap address.

Then store the new heap address (the first free address in the heap) back into the heap pointer.

		;	*****	operator		
		;		Precon		
		;		Postco		
006A	C90074	new:		LDX	hp	Initially, the value stored at hpPtr is
006D	710074			ADDA	hp	the address of heap. No memory is
0070	E10074			STA	hp	yet allocated.
0073	58			RETO		
0074	0076	hpPtr:	.ADDRESS	heap		; address of next free byte
0076	00	heap:	.BLOCK	1		; first byte in the heap
0077			.END			

# Dynamic Memory Allocation

- Using global pointer variables
  - Allocate storage for the variable with .BLOCK 2
  - Recall that a memory address is 16 bits or 2 bytes
  - Access the address stored at the pointer with direct addressing (d)
  - Access the value stored on the heap that the pointer is pointing to (Dereference the pointer) with *indirect* addressing (n)

# Dynamic Memory Allocation

- Calling the new function
  - Put the number of bytes to be allocated into the accumulator
  - CALL new
  - The index register will contain the address of the allocated bytes

**STX      b,d**

# Dynamic Memory Allocation Example

## fig 6.41

0000	040009		BR	main	
0003	0000	a:	.BLOCK		variable
0005	0000	b:	.BLOCK		variable
0007	0000	c:	.BLOCK		variable
		;			
		;	***** main ()		
0009	C00002	main:	LDA	2,i	; a = new int
000C	16006A		CALL	new	
000F	E90003		STX	a,d	
0012	C00005		LDA	3,i	
0015	E20003		STA	a,n	
0018	C00002		LDA	2,i	
0021	C00003		CALL	new	
0024	E20005		STX	b,d	
0027	C10003		LDA	3,i	
002A	E10007		STA	b,n	
002D	C10005		LDA	a,d	
0030	E10003		STA	c,d	
0033	C00002		LDA	b,d	; a = b
0036	720007		STA	a,d	
0039	E20003		LDA	2,i	; *a = 2 + *c
			ADDA	c,n	
			STA	a,n	

Use the pointer with  
indirect addressing.

Load Accumulator with  
number of bytes needed.  
Call new

; a = new int

On return from call, the  
address of the first byte of  
the allocated memory is  
in the index register.  
Save this address in the  
pointer.  
; c = a

; a = b

; \*a = 2 + \*c

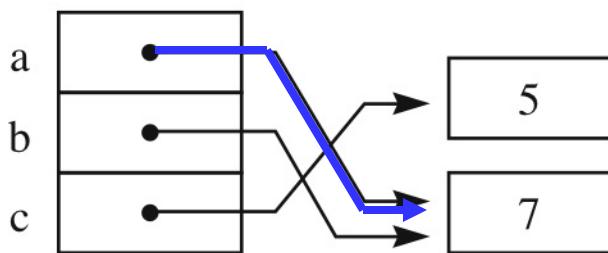
# Indirect Addressing

003C	410058	STRO	msg0,d	; cout << "*a = "
003F	3A0003	DECO	a,n	; << *a
0042	50000A	CHARO	'\n',i	; << endl
0045	41005E	STRO	msg1,d	; cout << "*b = "
0048	3A0005	DECO	b,n	; << *b
004B	50000A	CHARO	'\n',i	; << endl
004E	410064	STRO	msg2,d	; cout << "*c = "
0051	3A0007	DECO	c,n	; << *c
0054	50000A	CHARO	'\n',i	; << endl
0057	00	STOP		
0058	2A6120 msg0:	.ASCII	"*a = \x00"	
	3D2000			
005E	2A6220 msg1:	.ASCII	"*b = \x00"	
	3D2000			
0064	2A6320 msg2:	.ASCII	"*c = \x00"	
	3D2000			

# Indirect Addressing

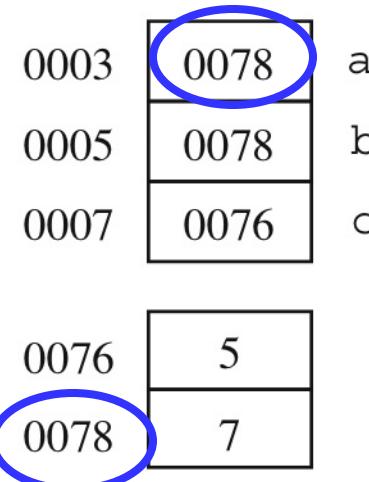
```
;***** operator new
;
;      Precondition: A contains number of bytes
;
;      Postcondition: X contains pointer to bytes
006A C90074 new:    LDX      hpPtr,d      ;returned pointer
006D 710074          ADDA     hpPtr,d      ;allocate from heap
0070 E10074          STA      hpPtr,d      ;update hpPtr
0073 58              RETO
0074 0076  hpPtr:    .ADDRESS heap        ;address of next free byte
0076 00  heap:       .BLOCK   1           ;first byte in the heap
0077               .END
```

# Indirect Addressing



(a) Global pointers at level HOL6.

The heap address  
is stored in the  
global pointer



(b) The global pointers at level Asmb5.

# Indirect Addressing

- $\text{Oprnd} = \text{Mem}[\text{Mem}[\text{OprndSpec}]]$

Addressing Mode	aaa	Letters	Operand
Immediate	000	i	OprndSpec
Direct	001	d	Mem [OprndSpec]
Indirect	010	n	Mem [Mem [OprndSpec]]
Stack-relative	011	s	Mem [SP + OprndSpec]
Stack-relative deferred	100	sf	Mem [Mem [SP + OprndSpec]]
Indexed	101	x	Mem [OprndSpec + X]
Stack-indexed	110	sx	Mem [SP + OprndSpec + X]
Stack-indexed deferred	111	sxf	Mem [ Mem [SP + OprndSpec] + X]

# Indirect Addressing

- Some machines also have a double indirect mode
  - $\text{Oprnd} = \text{Mem}[\text{Mem}[\text{Mem}[\text{OprndSpec}]]]$

# Dynamic Memory with local variables fig. 6.43

## High-Order Language

```
#include <iostream>
using namespace std;

int main () {
    int *a, *b, *c;
    a = new int;
    *a = 5;
    b = new int;
    *b = 3;
    c = a;
    a = b;
    *a = 2 + *c;
    cout << "*a = " << *a << endl;
    cout << "*b = " << *b << endl;
    cout << "*c = " << *c << endl;
    return 0;
}
```

# Dynamic Memory with local variables

- Same as pass-by-reference
- Allocate storage on the stack for the pointer using SUBSP
- Access the pointer with **stack-relative addressing (s)**
- Access the value stored on the heap that the pointer is pointing to (Dereference the pointer) with **stack-relative deferred addressing (sf)**

# Dynamic Memory with local variables fig. 6.43

0000	040003	BR	main	
		;		
		***** main ()		
		a: .EQUATE 4		;local variable
		b: .EQUATE 2		;local variable
		c: .EQUATE 0		;local variable
0003	680006	main:	SUBSP 6,i	;allocate locals
0006	C00002		LDA 2,i	;a = new int
0009	16006A		CALL new	
000C	EB0004		STX a,s	
000F	C00005		LDA 5,i	;*a = 5
0012	E40004		STA a, sf	
0015	C00002		LDA 2,i	;b = new int
0018	16006A		CALL new	
001B	EB0002		STX b,s	
001E	C00003		LDA 3,i	;*b = 3
0021	E40002		STA b, sf	
0024	C30004		LDA a,s	;c = a
0027	E30000		STA c,s	
002A	C30002		LDA b,s	;a = b
002D	E30004		STA a,s	
0030	C00002		LDA 2,i	;*a = 2 + *c
0033	740000		ADDA c, sf	
0036	E40004		STA a, sf	

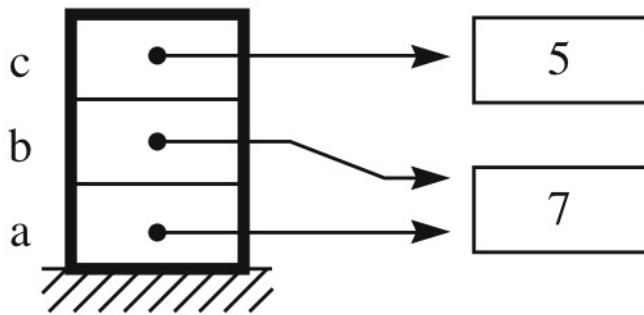
# Dynamic Memory with local variables

0039	410058	STRO	msg0.d	;cout << "*a = "
003C	3C0004	DECO	a.sf	; << *a
003F	50000A	CHARO	'\n',i	; << endl
0042	41005E	STRO	msg1.d	;cout << "*b = "
0045	3C0002	DECO	b.sf	; << *b
0048	50000A	CHARO	'\n',i	; << endl
004B	410064	STRO	msg2.d	;cout << "*c = "
004E	3C0000	DECO	c.sf	; << *c
0051	50000A	CHARO	'\n',i	; << endl
0054	600006	ADDSP	6,i	;deallocate locals
0057	00	STOP		
0058	2A6120 msg0:	.ASCII	"*a = \x00"	
	3D2000			
005E	2A6220 msg1:	.ASCII	"*b = \x00"	
	3D2000			
0064	2A6320 msg2:	.ASCII	"*c = \x00"	
	3D2000			

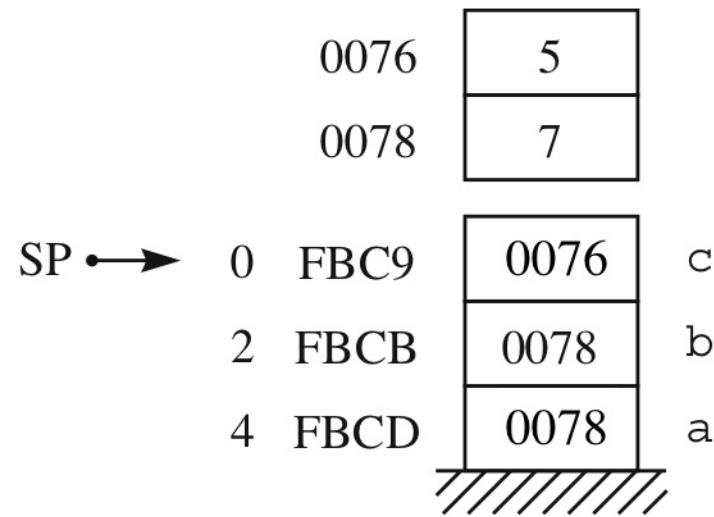
# Dynamic Memory with local variables

```
;***** operator new
;
;      Precondition: A contains number of bytes
;
;      Postcondition: X contains pointer to bytes
006A C90074 new:    LDX      hpPtr,d      ;returned pointer
006D 710074          ADDA     hpPtr,d      ;allocate from heap
0070 E10074          STA      hpPtr,d      ;update hpPtr
0073 58              RETO
0074 0076  hpPtr:    .ADDRSS heap        ;address of next free byte
0076 00  heap:       .BLOCK   1           ;first byte in the heap
0077 .END
```

# Indirect Addressing



(a) Local pointers at level HOL6.



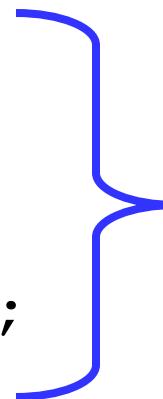
(b) The local pointers at level Asmb5.

# Structures fig. 6.45

- C struct:

```
struct person{  
    char first;  
    char last;  
    int age;  
    char gender;  
};
```

```
struct person bill;
```



fields

bill is a variable  
of type *struct person*

# Structures fig. 6.45

## High-Order Language

```
#include <iostream>
using namespace std;

struct person {
    char first;
    char last;
    int age;
    char gender;
};
person bill;

int main () {
    cin >> bill.first >> bill.last >> bill.age >> bill.gender;
    cout << "Initials: " << bill.first << bill.last << endl;
    cout << "Age: " << bill.age << endl;
    cout << "Gender: ";
    if (bill.gender == 'm') {
        cout << "male\n";
    }
    else {
        cout << "female\n";
    }
    return 0;
}
```

# Structures at level HOL6 and Level Asmb5

- Allocate storage for the total number of bytes needed for the structure.
- Create .EQUATE symbols for each field of the structure.
- To access a field:
  - Load the symbol of the field into the X register with immediate addressing
  - Use indexed addressing, using the name of the structure as the operand.

# Structures at level HOL6 and Level Asmb5

(Cont'd)

```
0000 040008          BR      main
                    first: .EQUATE 0
                    last:  .EQUATE 1
                    age:   .EQUATE 2
                    gender: .EQUATE 4
0003 000000          bill:  .BLOCK  5
                    0000
                    ;
                    ;***** main ()
0008 C80000 main:    LDX    first,i           ;cin >> bill.first
000B 4D0003            CHARI  bill,x
000E C80001            LDX    last,i           ;>>bill.last
0011 4D Load the offset for the field into the index register
0014 Use indexed addressing with the name of the struct variable
0017 350000             DECI  bill,_
001A C80004            LDX    gender,i         ;>>bill.gender
001D 4D0003            CHARI  bill,x
0020 41005A            STRO   msg0,d          ;cout << "Initials: "
0023 C80000            LDX    first,i          ;<< bill.first
0026 550003            CHARO  bill,x
0029 C80001            LDX    last,i           ;<< bill.last
002C 550003            CHARO  bill,x
002F 50000A            CHARO  '\n',i          ;<< endl
```

.EQUATE's give the field offsets

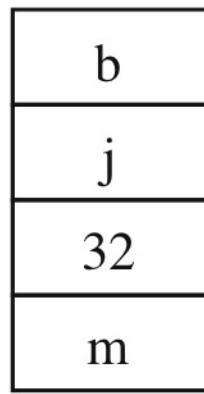
Allocate storage for the entire struct

# Structures at level HOL6 and Level

```
0032 410065      STRO    msg1,d          ;cout << "Age: "
0035 C80002      LDX     age,i           ;  << bill.age
0038 3D0003      DECO    bill,x
003B 50000A      CHARO   '\n',i          ;  << endl;
003E 41006B      STRO    msg2,d          ;cout << "Gender: "
0041 C80004      LDX     gender,i        ;if (bill.gender == 'm')
0044 C00000      LDA     0,i
0047 D50003      LDBYTEA bill,x
004A B0006D      CPA     'm',i
004D 0C0056      BRNE    else
0050 410074      STRO    msg3,d          ;  cout << "male\n"
0053 040059      BR     endIf
0056 41007A else: STRO    msg4,d          ;  cout << "female\n"
0059 00    endIf: STOP
005A 496E69 msg0: .ASCII   "Initials: \x00"
...
0065 416765 msg1: .ASCII   "Age: \x00"
...
006B 47656E msg2: .ASCII   "Gender: \x00"
...
0074 6D616C msg3: .ASCII   "male\n\x00"
...
007A 66656D msg4: .ASCII   "female\n\x00"
...
0082                      .END
```

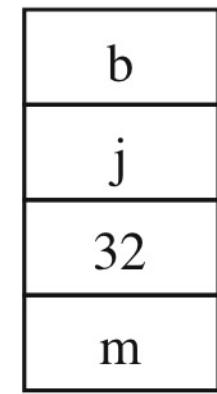
# Structures at level HOL6 and Level Asmb5 (*Cont'd*)

bill.first  
bill.last  
bill.age  
bill.gender



(a) A global structure at level HOL6.

0 0003  
1 0004  
2 0005  
4 0007



(b) The global structure at Asmb5.

# Linked Lists with a local pointer

- **Accesses a node:**
  - Equate the pointer field to the first byte of the node.
  - Load the offset into the index register
  - Access the field of the node using stack-indexed deferred addressing (sxf)

# Summary

## Addressing Mode Review

Mode		Operand	Uses
Immediate	i	OprndSpec	Constants, Offsets, Symbol address
Direct	d	M [OprndSpec]	Global variables
Indirect (deferred)	n	M [M [OprndSpec]]	Global pointer objects
Stack-relative	s	M [SP+OprndSpec]	Local variables
Stack-rel-deferred	sf	M [M [SP+OprndSpec]]	Local pointer objects
Indexed	x	M [OprndSpec+X]	Global array or struct
Stack-indexed	sx	M [SP+OperandSpec+X]	Local array or struct
Stack-ind-deferred	sfx	M [M [SP+OperandSpec+X]]	Local linked list

# Summary

## Review – Addressing Modes

- Stack Modes
  - Local Variables
- Indexed Modes
  - Arrays
  - Switch/Case
  - Structures
- Deferred/Indirect Modes
  - Pointer → Object
  - Call-by-reference actual parameter

# Summary

## Guidelines for Translating HOL → Assembly Language

- Expressions
  - Pay attention to *order of precedence*:  
Which part is computed first?
  
- Assignment statements
  - Load the right side of the statement into the accumulator
  - Store the result into the statement's left side variable

# Summary

## Review – Global / Local Variable

Variable	Global	Local
Memory allocation	Compile time	Run time stack
Assembly Directive	.BLOCK mem_size	.EQUATE stack_offset
Address Mode	Direct (d) Mem [var]	Stack Relative (s) Mem[SP + var]

# Summary

## Review – Data Directives

- **glob\_symb:**     .BLOCK                      *num\_bytes*
  - Global data
  - Load with Direct addressing mode
- **symb:**                 .EQUATE   *const*
  - Constant values
  - Offsets – stack, array, structure, case
  - Load with Immediate addressing mode
- **ptr\_symb:**         .ADDRSS                      *glob\_symb*
  - Address of a global symbol

# Chapter 10:

# Karnaugh Maps

# Karnaugh Maps

- **Simplification of Boolean functions yields:**
  - simpler digital circuits
  - usually faster digital circuits
- **Using Boolean identities**
  - time-consuming
  - error-prone
- **K-Maps**
- **Easy, systematic method to reduce Boolean expressions**

# Karnaugh Maps

- Maurice Karnaugh
  - telecommunications engineer
  - Bell Labs
  - 1953
- While exploring digital logic and its application to the design of telephone circuits
  - Invented a graphical way of visualizing
  - Applied it to simplifying Boolean expressions
- Graphical representation
  - named Karnaugh map or Kmap
  - in his honor

# Karnaugh Maps

- **Kmap**
  - matrix of rows and columns
  - represent output values of Boolean function
- **Output values placed in each cell**
  - derived from the minterms of a Boolean function
- ***Minterm* is a product term**
  - contains all of the function's variables exactly once
  - either complemented or not complemented

# Karnaugh Maps

- Minterms for function having the inputs  $x$  and  $y$ :

$\bar{x}\bar{y}$ ,  $\bar{x}y$ ,  $x\bar{y}$ , and  $xy$

- Consider the Boolean function:  $F(x, y) = xy + \bar{x}y$

- Its minterms are:

Minterm	x	y
$\bar{x}\bar{y}$	0	0
$\bar{x}y$	0	1
$x\bar{y}$	1	0
$xy$	1	1

# Karnaugh Maps

- Function with three inputs
- Minterms are shown in this diagram

Minterm	X	Y	Z
$\bar{x}\bar{y}\bar{z}$	0	0	0
$\bar{x}\bar{y}z$	0	0	1
$\bar{x}y\bar{z}$	0	1	0
$\bar{x}yz$	0	1	1
$x\bar{y}\bar{z}$	1	0	0
$x\bar{y}z$	1	0	1
$xy\bar{z}$	1	1	0
$xyz$	1	1	1

# Karnaugh Maps

- A Kmap has a cell for each minterm
- It has a cell for each line for the truth table of a function.
- Truth table for function

$$F(x,y) = xy$$

along with Kmap

		$F(x, y) = xy$
x	y	$xy$
0	0	0
0	1	0
1	0	0
1	1	1

x\y	0	1
0	0	0
1	0	1

# Karnaugh Maps

- Truth table and KMap for the function

$$F(x,y) = x + y$$

- Function is equivalent to OR of all of the minterms that have a value of 1

- Thus:

$$F(x, y) = X + Y = \bar{X}Y + X\bar{Y} + XY$$

$$F(X, Y) = X + Y$$

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

X	Y	0	1
0	0	0	1
1	1	1	1

# Karnaugh Maps

- Minterm function derived from Kmap
  - not in simplest terms
- Reduce the complicated expression to its simplest terms
  - find adjacent 1s in the Kmap
  - collected into groups that are powers of two

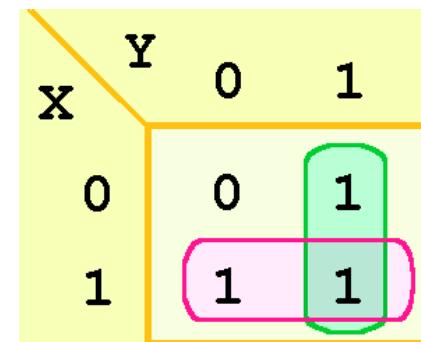
In this example, there are two such groups.

– Can you find them?

		Y 0	1
x 0	0	0	1
1	1	1	1

# Karnaugh Maps

- Best way of selecting two groups of 1s
  - shown below
- Both groups are powers of two and the groups overlap



# Karnaugh Maps

Rules of Kmap simplification are:

- Groupings can contain only 1s; no 0s
- Groups can be formed only at right angles; diagonal groups are not allowed
- Number of 1s in a group must be a power of 2 – even if it contains a single 1
- Groups must be made as large as possible
- Groups can overlap and wrap around the sides of the Kmap

# Karnaugh Maps

- Kmap for three variables
  - shown below
- Each minterm is placed in a cell
  - that cell will hold its value
  - Note: values for the  $yz$  combination
  - form a pattern that is not a normal binary sequence (It is the Grey-scale)

		yz	00	01	11	10
		x	00	01	11	10
x	0	$\bar{x}\bar{y}\bar{z}$	$\bar{x}\bar{y}z$	$\bar{x}yz$	$\bar{x}y\bar{z}$	
	1	$x\bar{y}\bar{z}$	$x\bar{y}z$	$xyz$	$x\bar{y}\bar{z}$	

# Karnaugh Maps

- First row of the Kmap
  - minterms where  $x$  is zero
- First column
  - minterms where  $y$  and  $z$  both are zero

		Y	Z	00	01	11	10
		x		00	01	11	10
x	0			$\bar{x}\bar{y}\bar{z}$	$\bar{x}\bar{y}z$	$\bar{x}yz$	$\bar{x}y\bar{z}$
	1			$\bar{x}\bar{y}\bar{z}$	$x\bar{y}z$	$xyz$	$xy\bar{z}$

# Karnaugh Maps

- Consider the function:

$$F(X, Y, Z) = \bar{X}\bar{Y}Z + \bar{X}YZ + X\bar{Y}Z + XYZ$$

- Its Kmap is given below.
  - What is the largest group of 1s that is a power of 2?

		YZ	00	01	11	10
		X	0	1	1	0
0	0	0	1	1	0	
	1	0	1	1	0	

# Karnaugh Maps

- Grouping shows changes in the variables  $x$  and  $y$  have no influence upon the value of the function
- The function:

$$F(X, Y, Z) = \bar{X}\bar{Y}Z + \bar{X}YZ + X\bar{Y}Z + XYZ$$

reduces to  $F(x) = z$ .

You could verify this reduction with identities or a truth table.

		YZ	00	01	11	10	
		X	0	0	1	1	0
x	0	0	1	1	0		
	1	0	1	1	0		

# Karnaugh Maps

- Next -Consider the function:

$$F(X, Y, Z) = \overline{XYZ} + \overline{XY}Z + \overline{X}YZ + \overline{XY}\overline{Z} + X\overline{Y}\overline{Z} + XY\overline{Z}$$

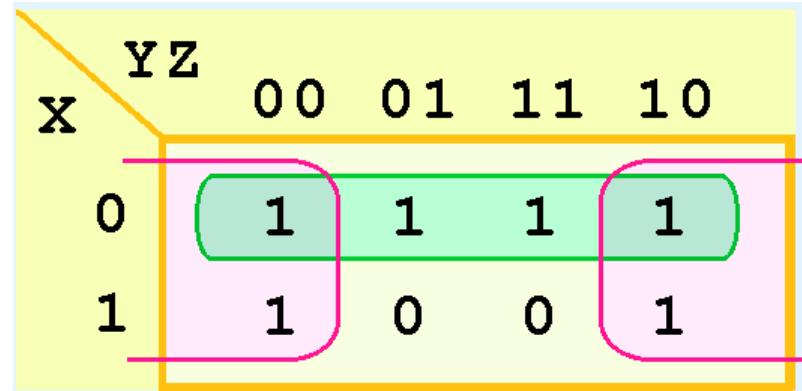
- Kmap is shown below
  - There are two groupings of 1s
  - Can you find them?

X \ YZ	00	01	11	10
0	1	1	1	1
1	1	0	0	1

# Karnaugh Maps

- One group wraps around the sides of the Kmap
- Values of  $x$  and  $y$  are not relevant to the term of the function in pink group
  - What does this tell us about the term of the function?  
 $Z = 0$  in the pink group so one term is  $\bar{z}$

What about the green group in the top row?

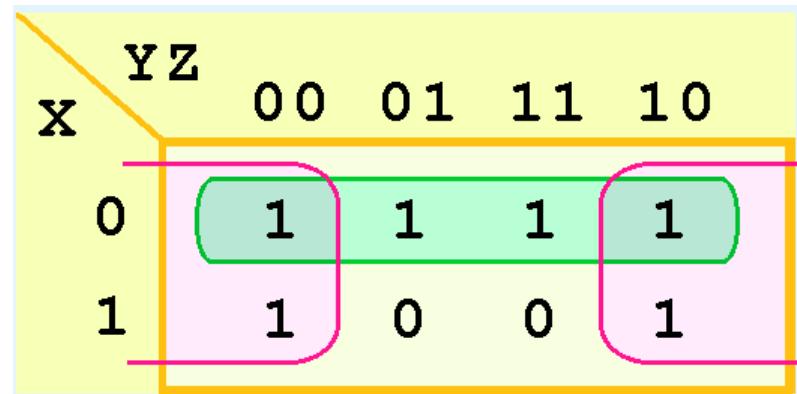


# Karnaugh Maps

- Green group the top row
  - only the value of  $x$  is significant  $X=0$
- It is complemented or 0 in that row
  - the other term of the reduced function is  $\bar{x}$
- Our reduced function is:

$$F(X, Y, Z) = \bar{x} + \bar{z}$$

Recall that we had four minterms in our original function!



# Karnaugh Maps

- Model can be extended to 16 minterms
  - produced by a four-input function
- Format below for 16-minterm Kmap

WX \ YZ	00	01	11	10
00	$\bar{W}\bar{X}Y\bar{Z}$	$\bar{W}X\bar{Y}Z$	$\bar{W}\bar{X}YZ$	$\bar{W}XY\bar{Z}$
01	$\bar{W}XY\bar{Z}$	$\bar{W}X\bar{Y}Z$	$\bar{W}XYZ$	$\bar{W}XYZ$
11	$W\bar{X}Y\bar{Z}$	$W\bar{X}YZ$	$WXY\bar{Z}$	$WXYZ$
10	$W\bar{X}Y\bar{Z}$	$W\bar{X}YZ$	$W\bar{X}YZ$	$W\bar{X}YZ$

# Karnaugh Maps

- Kmap shown below populated with the nonzero minterms from the function:

$$F(W, X, Y, Z) = \bar{W}\bar{X}Y\bar{Z} + \bar{W}\bar{X}Y\bar{Z} + \bar{W}\bar{X}Y\bar{Z}$$
$$+ \bar{W}XY\bar{Z} + W\bar{X}\bar{Y}\bar{Z} + W\bar{X}\bar{Y}Z + W\bar{X}YZ$$

- Can you identify (only) three groups in this Kmap?

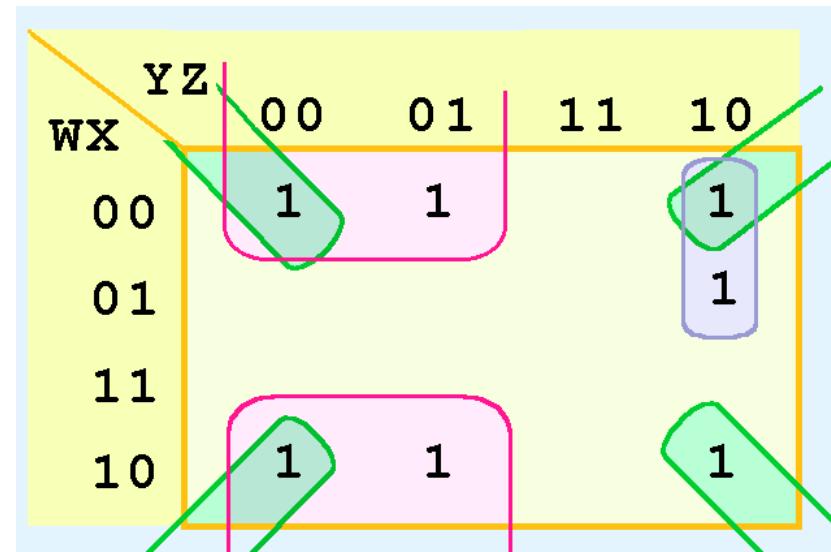
Recall that  
groups can  
overlap.

		YZ	00	01	11	10
		WX	00	01	11	10
W	0	1	1		1	
	1				1	
	2	1	1		1	

# Karnaugh Maps

- Three groups consist of:
  - A purple group entirely within the Kmap at the right.
  - A pink group that wraps the top and bottom.
  - A green group that spans the corners.
- Three terms in the final function:

$$F(W, X, Y, Z) = \bar{X}\bar{Y} + \bar{X}\bar{Z} + \bar{W}YZ$$



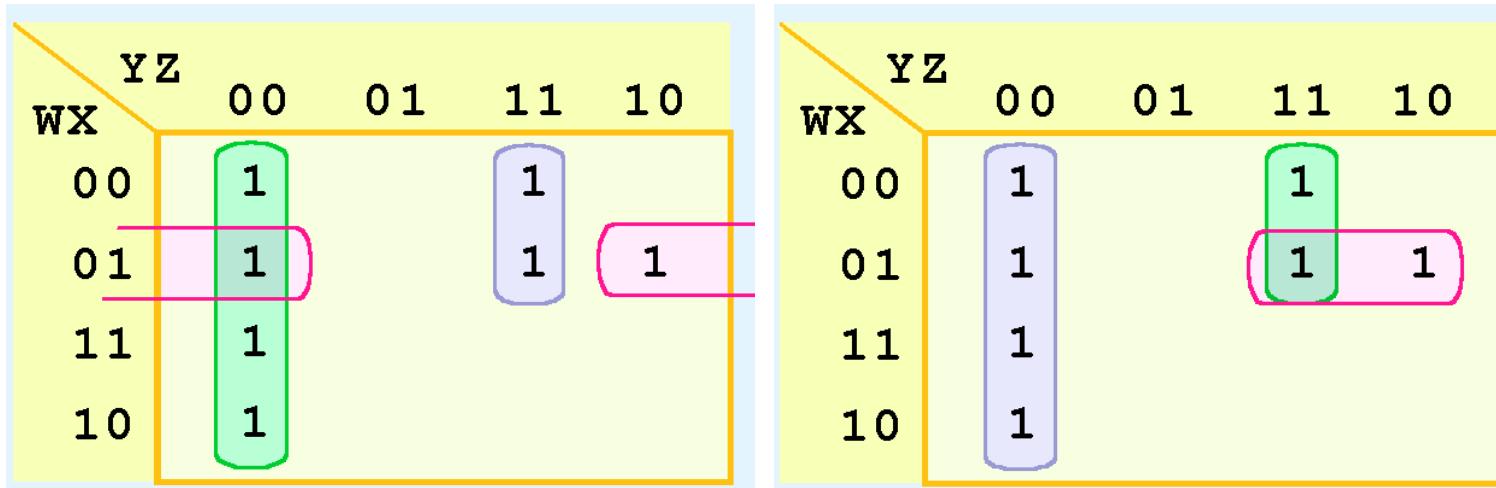
217

University of Illinois  
at Springfield



# Karnaugh Maps

- Different ways to pick groups within a Kmap
  - Must choose
  - Keep groups as large as possible
- Different functions that result
  - Logically equivalent.



# Don't Care: Example

- Real circuits don't always have an output defined for every input.
  - Calculator displays consist of 7-segment LEDs –  
These LEDs can display  $2^7 - 1$  patterns  
Only ten patterns are useful
- Circuit is design such that a particular set of inputs can never happen
  - don't care condition
- Don't Cares help simplify Kmap circuit

# Don't Care: Example

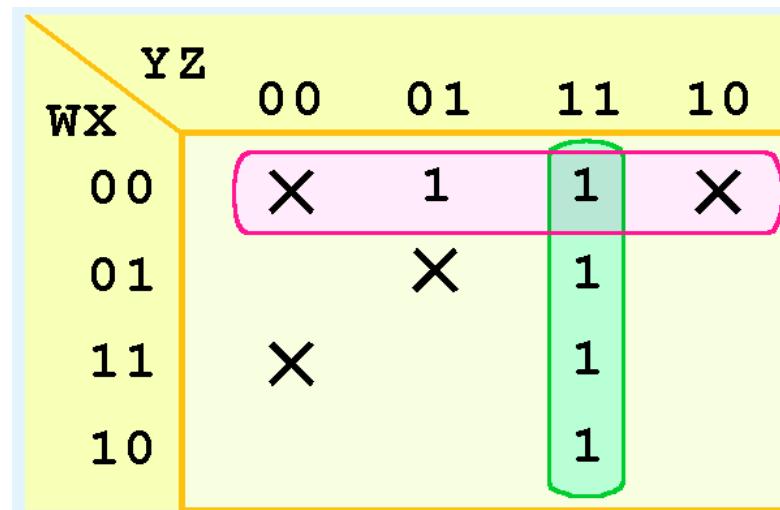
- Don't care condition is identified by an  $X$  in the cell of the minterm(s) for the don't care inputs
- $X$ 's can be included or ignored when creating groups for simplification

w\nx	y\nz	00	01	11	10
00		X	1	1	X
01			X	1	
11		X		1	
10				1	

# Don't Care: Example

- One grouping of the Kmap below
- This function is the result:

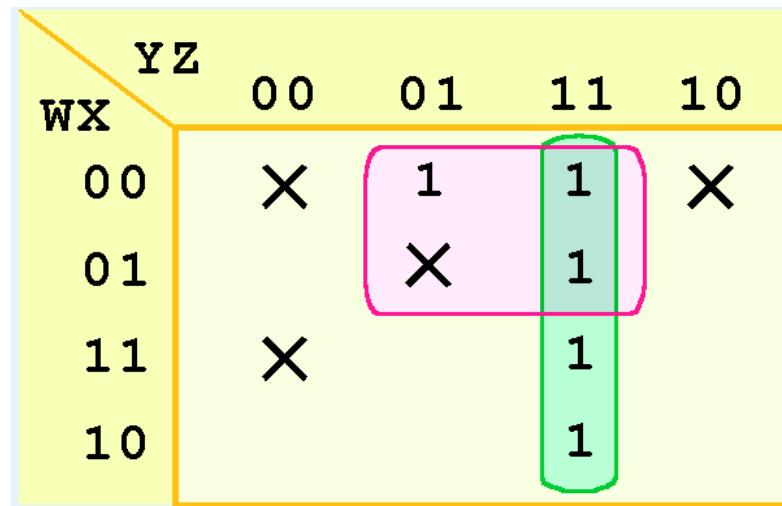
$$F(W, X, Y, Z) = \overline{W}\overline{X} + YZ$$



# Don't Care: Example

- A different grouping yields this function:

$$F(W, X, Y, Z) = \bar{W}Z + YZ$$



# Don't Care: Example

- The truth table of:

$$F(W, X, Y, Z) = \overline{W}\overline{X} + YZ$$

is different from the truth table of:

$$F(W, X, Y, Z) = \overline{W}Z + YZ$$

- Values for which they differ - the inputs for which have don't care conditions

WX	YZ	00	01	11	10
00	X	1	1	X	
01		X	1		
11	X		1		
10			1		

WX	YZ	00	01	11	10
00	X	X	1	1	X
01		X	X	1	
11	X		1	1	
10			1	1	

# Karnaugh Maps

- K-maps are a powerful tool for simplifying Boolean expressions to find the simplest functional equivalent
- Let's work an example step-by-step

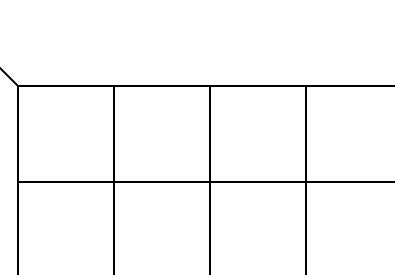
# Karnaugh Maps

Use K-map to simplify the following expression:

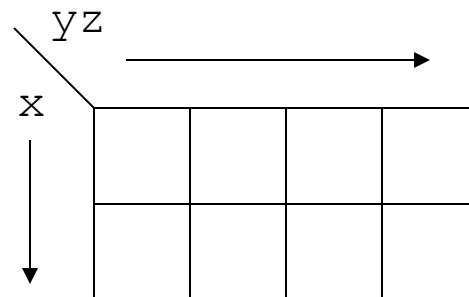
$$F(x, y, z) = xy'z' + x'y'z + xz' + y'z + x'$$

First note three variables in the expression  $F()$

therefore K-map will have  $2^3=8$  cells

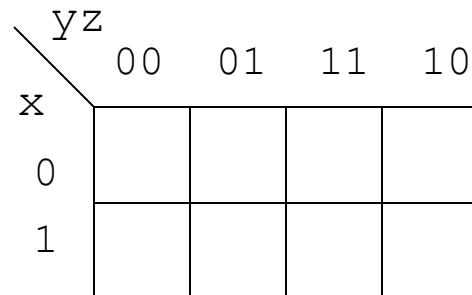


Next determine where each variable will be represented



Next arrange literal values for x,y and z so binary vectors next to each other differ by only one bit

Logically adjacent vectors are also physically adjacent. (grey code order)

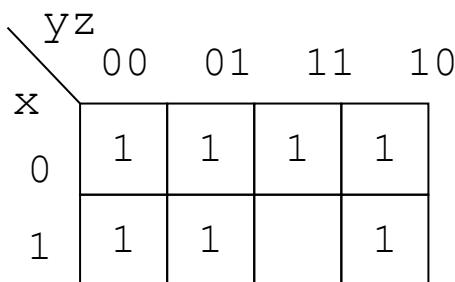


Now place ones (true values) in K-map to show which binary vectors satisfy the expression F()

$$F(x, y, z) = xy'z' + x'y'z + xz' + y'z + x'$$

↓                    ↓                    ↓                    ↓                    ↓

100                001                1-0                -01                0--



↓                    ↓                    ↓                    ↓                    ↓

100                001                110                101                000

↓                    ↓                    ↓                    ↓                    ↓

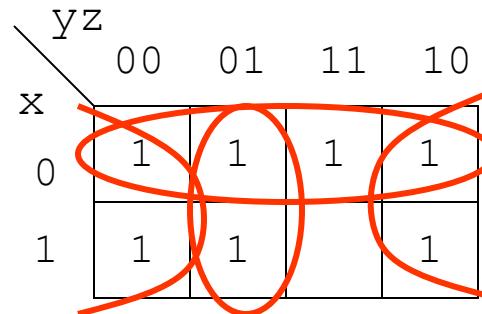
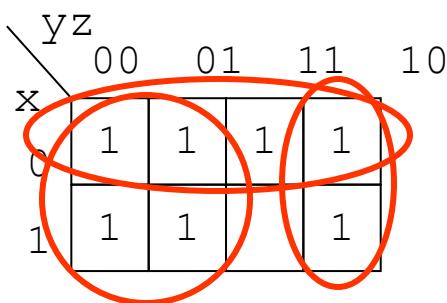
010                001                011                001                000

↓                    ↓                    ↓                    ↓                    ↓

010                001                011                001                000

Find smallest number of the largest rectangular patterns of 1's whose sizes are integer powers of two. The size of a selected rectangle of ones must contain  $1, 2, 4, \dots, 2^n$  cells. In this example 4 is the limit.

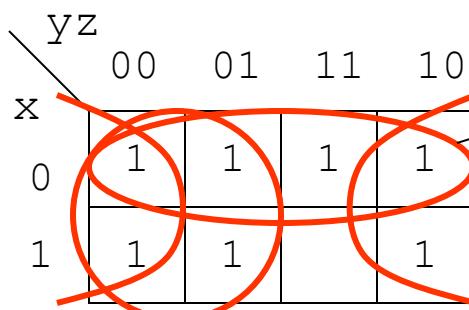
Choose a set of these rectangles such that every 1 is in at least one of the rectangles. Here are a few candidates:



In each case, all the 1's in three rectangles of the required dimensions.

Map below is best because it uses larger patterns (3 groups of 4).

Finally, convert these patterns back into terms in the simplified expression for  $F(x, y, z)$ .

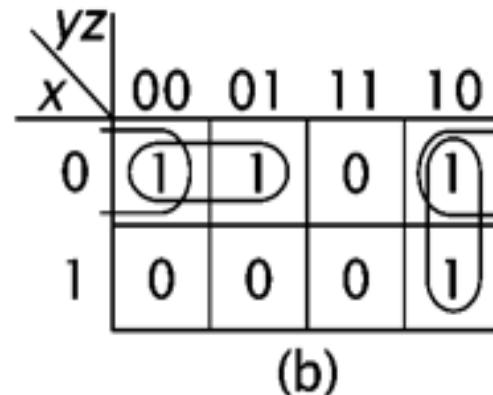
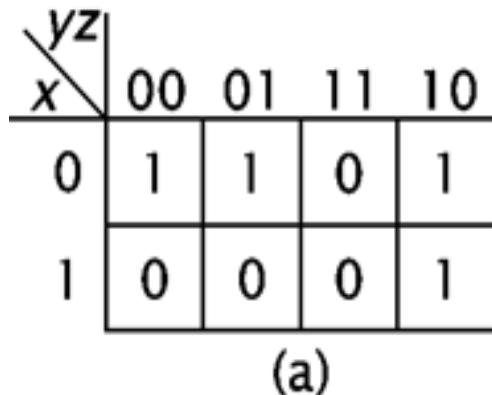


$$F(x, y, z) = x' + y' + z'$$

# K-map Example

- Let's consider  $(xy' + yz)' = x'y' + x'z' + yz'$
- Group together the 1s in the map:
  - $g_1: x'y'z' + x'y'z = x'y'(z' + z) = x'y'$
  - $g_2: x'yz' + xyz' = yz'(x' + x) = yz'$
  - $g_3: x'yz' + x'y'z' = x'z'(y + y') = x'z'$
- To derive a minimal expression we must select the fewest groups that cover all active minterms (1s).
- $(xy' + yz)' = x'y' + yz'$

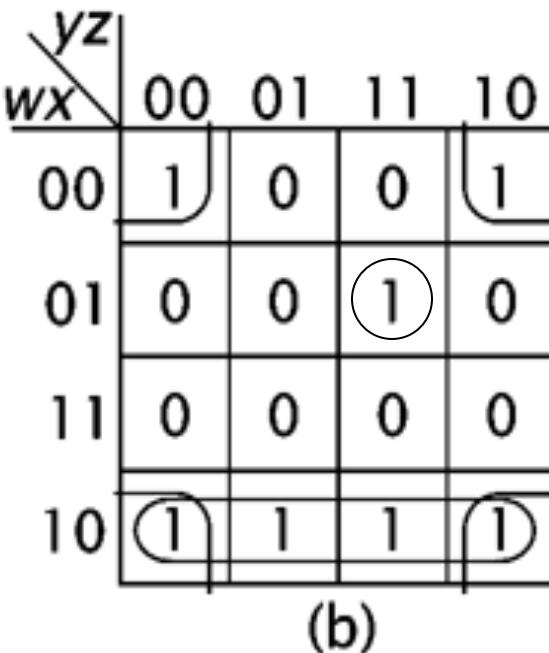
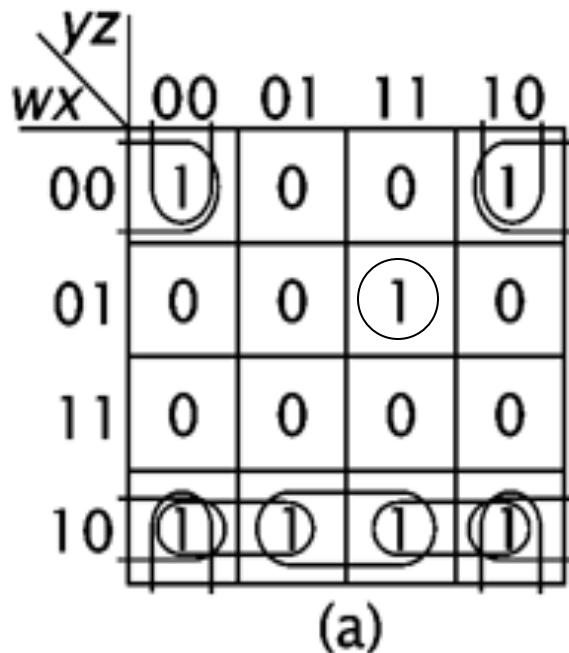
x	y	z	$x'y' + yz' + yz'$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



# K-map for more complex function

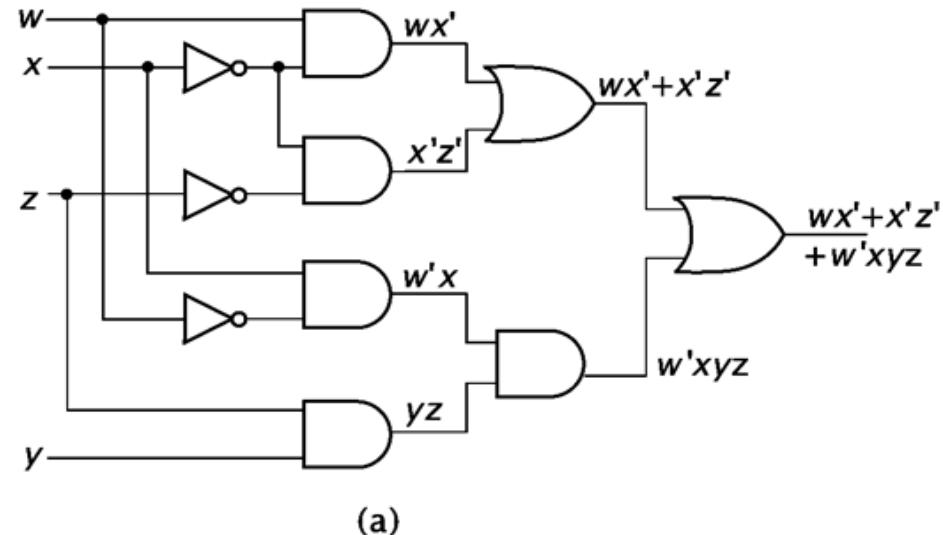
The final minimized function is:

$$x'z' + wx' + w'xyz$$



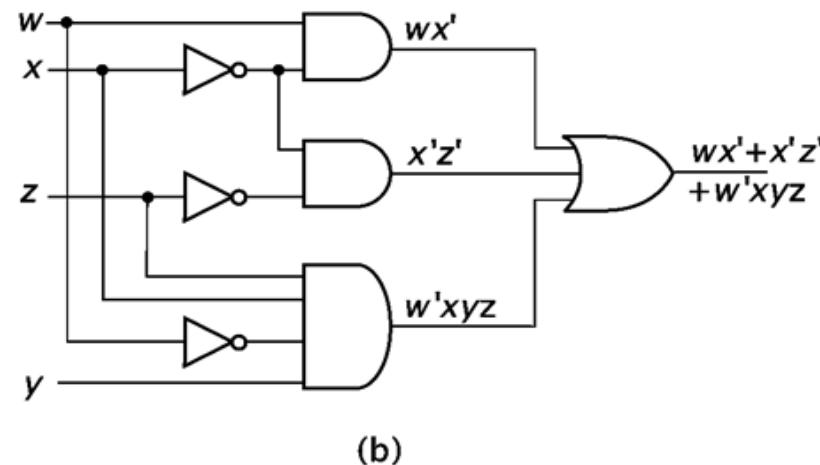
Circuit implementation  
of the function in the  
previous slide:

using two input gates  
(Figure a)



(a)

using two and four  
input gates (Figure b).



(b)

# Karnaugh Maps

- Kmaps - graphical method of simplifying Boolean expressions
- Kmap - matrix consisting of outputs of minterms of a Boolean function
- Kmaps - reviewed 2- 3- and 4-input
- Method can be extended to any number of inputs through use of multiple tables

# Karnaugh Maps

**Recapping the rules of Kmap simplification:**

- Groupings can contain only 1s; no 0s.
- Groups can be formed only at right angles; diagonal groups are not allowed
- Number of 1s in a group must be a power of 2
  - even if it contains a single 1
- Groups must be made as large as possible
- Groups can overlap and wrap around the sides of the Kmap
- Use don't care conditions when you can
- Select the fewest groups that cover all active minterms (1s)

# **Chapter 6:**

## **Fig 6.40**

# **Switch Statement**

# The Switch Statement fig. 6.40

```
#include <stdio.h>

int main( ) {
    int guess;
    printf("Pick a number 0..3");
    scanf("%d", &guess);
    switch(guess) {
        case 0: printf("Not Close");
                  break;
        case 1: printf("Close");
                  break;
        case 2: printf("Right on");
                  break;
        case 3: printf("Too High");
                  break;
    }
    printf("\n");
    return 0;
}
```

# The Switch Statement

```
0000 040003          BR      main
;
;***** main ()
guess:   .EQUATE 0           ;local variable
0003 680002 main:    SUBSP   2,i       ;allocate local
0006 410037           STRO    msgI
0007 000000             ADDSP   2,i       ;addresses occupy two bytes
0010 050013             BR      guessJT
0013 001B guessJT:    .ADDRESS case0
0015 0021               .ADDRESS case1
0017 0027               .ADDRESS case2
0019 002D               .ADDRESS case3
001B 41004C case0:    STRO    msg0,d   ;cout << "Not close"
001E 040030             BR      endCase  ;break
0021 410056 case1:    STRO    msg1,d   ;cout << "Close"
0024 040030             BR      endCase  ;break
0027 41005C case2:    STRO    msg2,d   ;cout << "Right on"
002A 040030             BR      endCase  ;break
002D 410065 case3:    STRO    msg3,d   ;cout << "Too high"
0030 50000A endCase:  CHARO   '\n',i   ;count << endl
0033 600002             ADDSP   2,i       ;deallocate local
0036 00                 STOP
```

The code inserted in the machine pr  
the address of each of the case labe.  
Use indexed addressing in  
the branch instruction

The Jump Table

# The Switch Statement

```
0037 506963 msgIn:    .ASCII  "Pick a number 0..3: \x00"
      ...
004C 4E6F74 msg0:    .ASCII  "Not close\x00"
      ...
0056 436C6F msg1:    .ASCII  "Close\x00"
      ...
005C 526967 msg2:    .ASCII  "Right on\x00"
      ...
0065 546F6F msg3:    .ASCII  "Too high\x00"
      ...
006E                      .END
```

# Jump Table

## Jump Tables

- Array of subprogram addresses
- Efficient way to combine:
  - 1-of-many Selection
  - Branching
- Requires translating the choice into a count variable
- Applications
  - Switch/Case instruction
  - Operating System calls
  - Interrupt traps

Sample Jump Table		
Address	Contents	Description of Contents
0140	1203	Address of Case 0
0142	13A5	Address of Case 1
0144	142B	Address of Case 2

# Jump Table

- How it works:

- Input value: **2**

DECI guess, s

instruction Register

0000

Index Register

FBCD

Stack Pointer

label	Address	Contents
		...
guessJT:	0013	00
	0014	1B
	0015	00
	0016	21
	0017	00
	0018	27
	0019	00
	0020	2D
		...
		FBCD
		FBCE
		FBCF

# Jump Table

- How it works:

- Input value: **2**

DECI guess, s

instruction Register

0000

Index Register

FBCD

Stack Pointer

label	Address	Contents
		...
guessJT:	0013	00
	0014	1B
	0015	00
	0016	21
	0017	00
	0018	27
	0019	00
	0020	2D
		...
	FBCD	00
	FBCE	02
	FBCF	

# Jump Table

- How it works:

- Input value: **2**

LDX guess, s

instruction Register

0002

Index Register

FBCD

Stack Pointer

label	Address	Contents
		...
guessJT:	0013	00
	0014	1B
	0015	00
	0016	21
	0017	00
	0018	27
	0019	00
	0020	2D
		...
	FBCD	00
	FBCE	02
	FBCF	

# Jump Table

- How it works:

- Input value: 2

ASLX

instruction Register

0004

Index Register

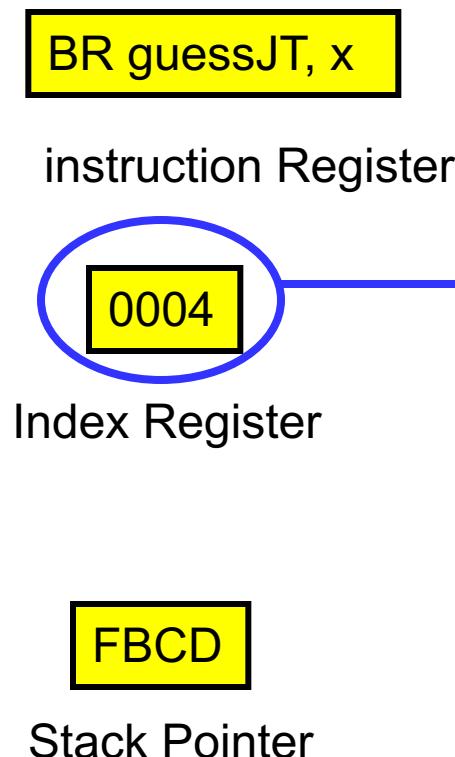
FBCD

Stack Pointer

label	Address	Contents
		...
guessJT:	0013	00
	0014	1B
	0015	00
	0016	21
	0017	00
	0018	27
	0019	00
	0020	2D
		...
	FBCD	00
	FBCE	02
	FBCF	

# Jump Table

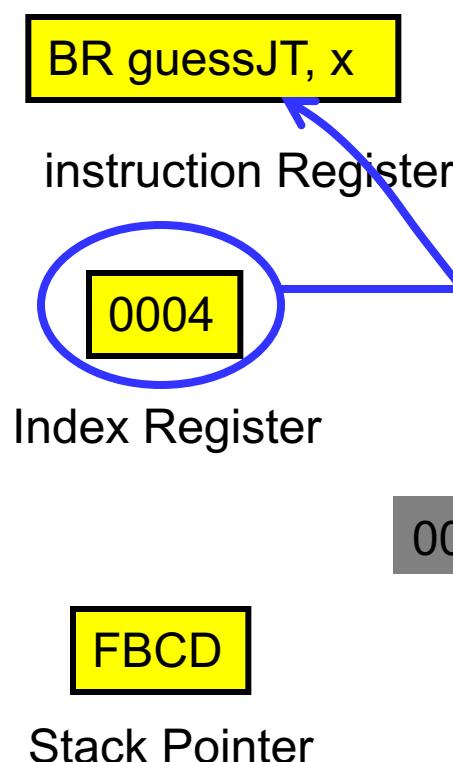
- How it works:
  - Input value: 2



label	Address	Contents
		...
<code>guessJT:</code>	0013	00
	0014	1B
	0015	00
	0016	21
	0017	00
	0018	27
	0019	00
	0020	2D
		...
	FBCD	00
	FBCE	02
	FBCF	

# Jump Table

- How it works:
  - Input value: 2



label	Address	Contents
		...
guessJT:	0013	00
	0014	1B
	0015	00
	0016	21
	0017	00
	0018	27
	0019	00
	0020	2D
0027 Becomes the operand for the branch		
		...
	FBCD	00
	FBCE	02
	FBCF	

# **Chapter 6:**

## **Fig 6.38**

# **Subroutine with Array parameter**

# A procedure call with an array parameter at Level HOL6 and Level Asmb5 fig. 6.38

## High-Order Language

```
#include <iostream>
using namespace std;

void getVect (int v[], int& n) {
    int j;
    cin >> n;
    for (j = 0; j < n; j++) {
        cin >> v[j];
    }
}

void putVect (int v[], int n) {
    int j;
    for (j = 0; j < n; j++) {
        cout << v[j] << ' ';
    }
    cout << endl;
}

int main () {
    int vector[8];
    int numItms;
    getVect (vector, numItms);
    putVect (vector, numItms);
    return 0;
}
```



# A procedure call with an array parameter at Level HOL6 and Level Asmb5

```
0000 040049      BR      main
;
;***** getVect (int
v:      .EQUATE 6
n:      .EQUATE 4
i:      .EQUATE 0
0003 680002 getVect: SUBSP 2,i
0006 340004      DECI   n,
0009 C80000      LDX    0,
000C EB0000      STX    i,
000F BC0004 forl:  CPX    n,sf
0012 0E0025      BRGE  endForl
0015 1D          ASLY
0016 370006      DECI   v,sxf
0019 CB0000      LDX    i,s
001C 780001      ADDX   1,i
001F EB0000      STX    i,s
0022 04000F      BR     forl
0025 5A          endForl: RET2
;
;pop local and retAddr
```

Input the value using **stack-relative** ( initialize i using stack-relative ) Test the loop using the relative

Double the index for int's Create Input using **stack-indexed** is deferred addressing. <sup>(r)</sup> i (which is stored in the index register).

# A procedure call with an array parameter at Level HOL6 and Level Asmb5

```
;***** putVect (int v[], int n)
v2:      .EQUATE 6                      ;formal parameter
n2:      .EQUATE 4                      ;formal parameter
i2:      .EQUATE 0                      ;local variable
0026 680002 putVect: SUBSP  2,i          ;allocate local
0029 C80000             LDX   0,i          ;for (i = 0
002C EB0000             STX   i2,s
002F BB0004 for2:    CPX   n2,s          ;    i < n
0032 0E0048             BRGE  endFor2
0035 1D                 ASLX
0036 3F0006             DECO  v2,sxf        ;    an integer is two bytes
0039 500020             CHARO ' ',i         ;    cout << v[i]
003C CB0000             LDX   i2,s          ;    << ' '
003F 780001             ADDX  1,i
0042 EB0000             STX   i2,s
0045 04002F             BR    for2
0048 5A     endFor2: RET2            ;pop local and retAddr
```

# A procedure call with parameters at Level HOL6

```
;***** main ()  
vector: .EQUATE 2  
numItms: .EQUATE 0  
  
0049 680012 main: SUBSP 18,i :allocate locals  
004C 02  
004D 700002 MOVSPA  
0050 E3FFE ADDA ve  
0053 02 STA -2,s  
0054 700000 MOVSPA  
0057 E3FFFC ADDA numItms,i  
005A 680004 STA 4,s  
005D 160003 SUBSP 4,i  
005E 600004 CALL getVect  
0060 600004 ADDSP 4,i ;pop params  
0063 02 MOVSPA ;push address of vector  
0064 700002 ADDA vector,i  
0067 E3FFE STA -2,s  
006A C30000 LDA numItms,s ;push value of numItms  
006D E3FFFC STA -4,s  
0070 680004 SUBSP 4,i ;push params  
0073 160026 CALL putVect ;putVect (vector, numItms)  
0076 600004 ADDSP 4,i ;pop params  
0079 600012 ADDSP 18,i ;deallocate locals  
007C 00 STOP  
007D .END
```

Load the address of the first element of the array onto the stack.

Move the address of the second parameter onto the stack.

Change the top of the run time stack

Pop off the parameters

Push parameters for next procedure call

;pop params  
;push address of vector

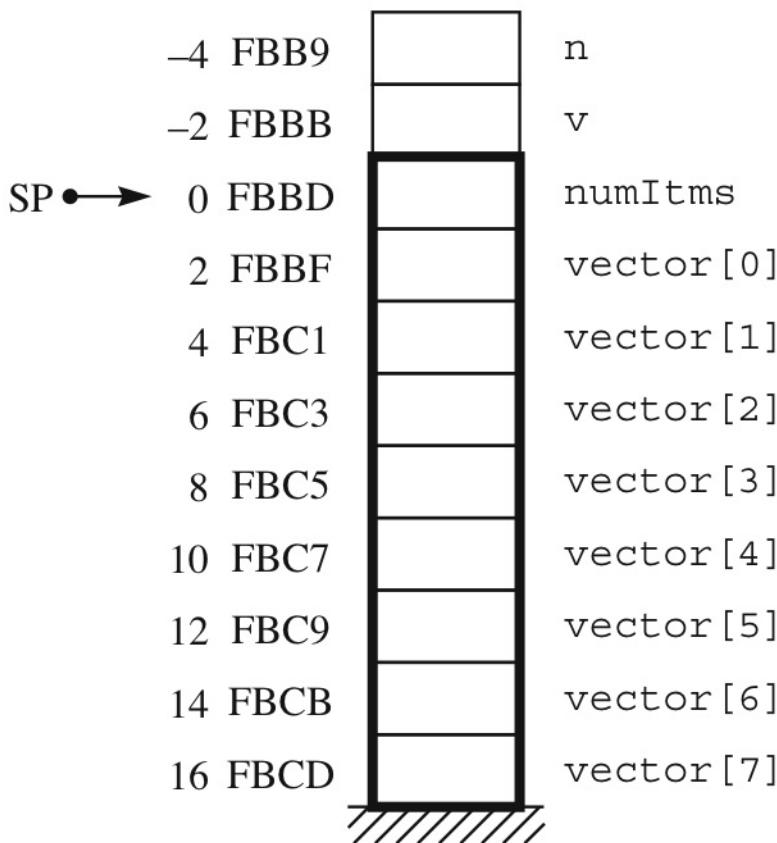
;push value of numItms

;push params  
;putVect (vector, numItms)

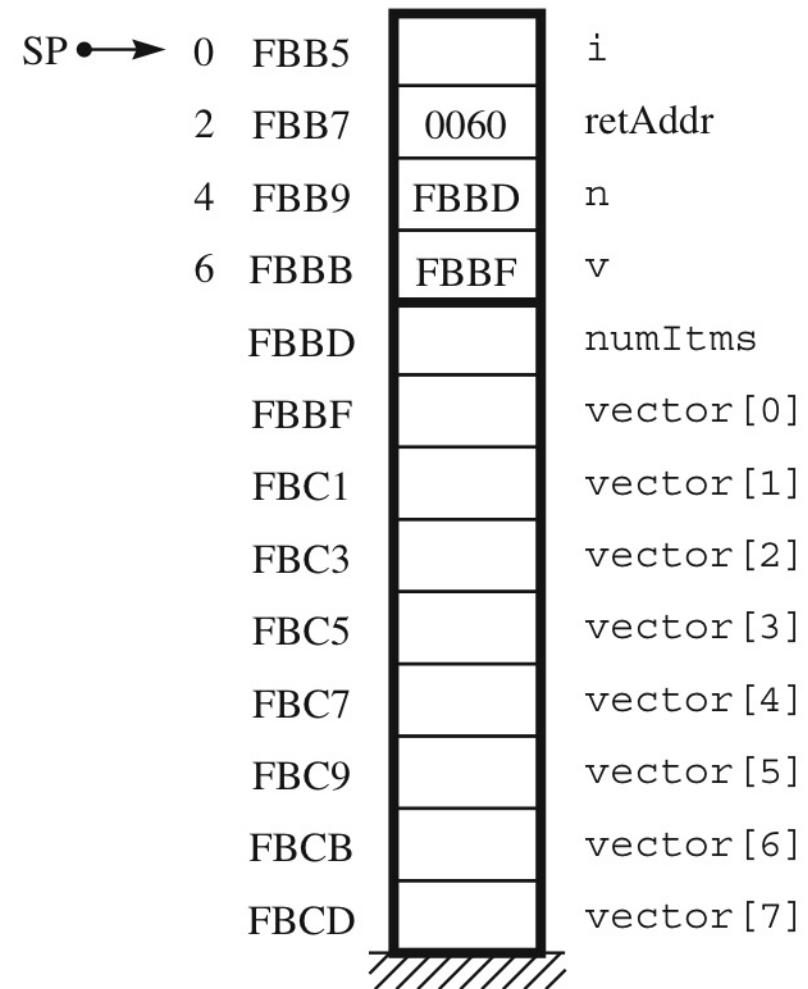
;pop params

;deallocate locals

# Arrays as parameters



(a) Before calling `getVect.`



(b) After calling `getVect.`

# Chapter 6: Fig 6.47 Linked Lists

# Linked Lists fig. 6.47

```
#include <iostream>
using namespace std;

struct node {
    int data;
    node* next;
};

int main () {
    node *first, *p;
    int value;
    first = 0;
    cin >> value;
    while (value != -9999) {
        p = first;
        first = new node;
        first->data = value;
        first->next = p;
        cin >> value;
    }
    for (p = first; p != 0; p = p->next) {
        cout << p->data << ' ';
    }
    return 0;
}
```

# Linked Lists

```
0000 040003          BR      main
                    data:   .EQUATE 0           ;struct field
                    next:   .EQUATE 2          ;struct field
                    ;
                    ;***** main ()
                    first:  .EQUATE 4           ;local variable
                    p:      .EQUATE 2           ;local variable
                    value:  .EQUATE 0           ;local variable
0003 680006 main:    SUBSP   6,i           ;allocate locals
0006 C00000          LDA     0,i           ;first = 0
0009 E30004          STA     first,s
000C 330000          DECI    value,s       ;cin >> value
000F C30000 while:   LDA     value,s       ;while (value != -9999)
0012 B0D8F1          CPA     -9999,i
0015 0A003F          BREQ    endWh
0018 C30004          LDA     first,s        ;  p = first
001B E30002          STA     p,s
001E C00004          LDA     4,i           ;  first = new node
0021 160067          CALL    new
0024 EB0004          STX     first,s
0027 C30000          LDA     value,s        ;  first->data = value
002A C80000          LDX     data,i
002D E70004          STA     first,sxf
```

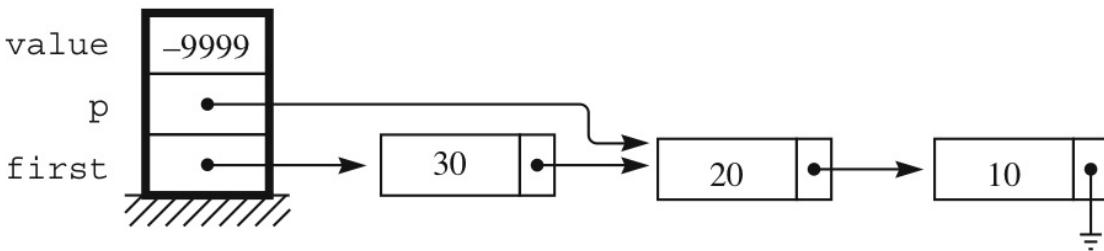
# Linked Lists

```
0030 C30002      LDA    p,s          ; first->next = p
0033 C80002      LDX    next,i
0036 E70004      STA    first,sxf
0039 330000      DECI   value,s      ; cin >> value
003C 04000F      BR     while
003F C30004 endWh: LDA    first,s      ; for (p = first
0042 E30002      STA    p,s
0045 C30002 for:  LDA    p,s          ; p != 0
0048 B00000      CPA    0,i
004B 0A0063      BREQ   endFor
004E C80000      LDX    data,i       ; cout << p->data
0051 3F0002      DECO   p,sxf
0054 500020      CHARO  ',i          ;     << '
0057 C80002      LDX    next,i       ; p = p->next)
005A C70002      LDA    p,sxf
005D E30002      STA    p,s
0060 040045      BR     for
0063 600006 endFor: ADDSP  6,i       ; deallocate locals
0066 00          STOP
```

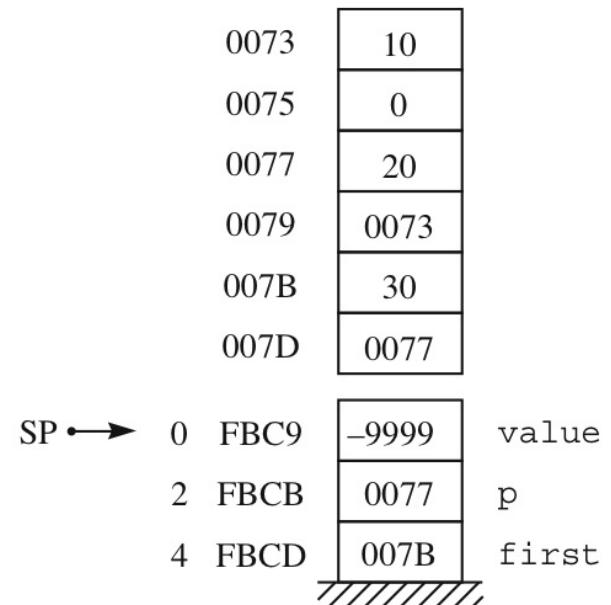
# Linked Lists

```
;***** operator new
;
;      Precondition: A contains number of bytes
;
;      Postcondition: X contains pointer to bytes
0067 C90071 new:    LDX      hpPtr,d      ;returned pointer
006A 710071          ADDA     hpPtr,d      ;allocate from heap
006D E10071          STA      hpPtr,d      ;update hpPtr
0070 58              RETO
0071 0073  hpPtr:    .ADDRESS heap        ;address of next free byte
0073 00  heap:       .BLOCK   1           ;first byte in the heap
0074                      .END
```

# Linked Lists



(a) The linked list at level HOL6.



(b) The linked list at level Asmb5.

# Chapter 6: Boolean Types

# Boolean Types

- Don't exist at assembly/machine level
- Can simulate with 1/0 values:

true:           .EQUATE           1

false:          .EQUATE          0

# Boolean Example

```
#include <iostream>
using namespace std;

const int LOWER = 21;
const int UPPER = 65;

bool inRange (int a) {
    if ((LOWER <= a) && (a <= UPPER)) {
        return true;
    }
    else {
        return false;
    }
}

int main () {
    int age;
    cin >> age;
    if (inRange (age)) {
        cout << "Qualified\n";
    }
    else {
        cout << "Unqualified\n";
    }
    return 0;
}
```

# Boolean Example

```
0000 040023      BR      main
                  true:   .EQUATE 1
                  false:  .EQUATE 0
                  ;
                  LOWER:  .EQUATE 21           ;const int
                  UPPER:  .EQUATE 65           ;const int
                  ;
                  ;***** bool inRange (int a)
                  retVal: .EQUATE 4           ;returned value
                  a:      .EQUATE 2           ;formal parameter
0003 C00015 inRange: LDA    LOWER,i          ;if ((LOWER <= a)
0006 B30002 if:      CPA    a,s
0009 10001C          BRGT   else
000C C30002          LDA    a,s           ;  && (a <= UPPER))
000F B00041          CPA    UPPER,i
0012 10001C          BRGT   else
0015 C00001 then:   LDA    true,i          ;  return true
0018 E30004          STA    retVal,s
001B 58              RETO
001C C00000 else:   LDA    false,i          ;  return false
001F E30004          STA    retVal,s
0022 58              RETO
```

# Boolean Example

```
;***** main ()
age:    .EQUATE 0           ;local variable
0023 680002 main:   SUBSP  2,i      ;allocate local
0026 330000          DECI   age,s   ;cin >> age
0029 C30000 if2:    LDA    age,s   ;if (
002C E3FFFC          STA    -4,s    ;store the value of age
002F 680004          SUBSP  4,i    ;push parameter and retVal
0032 160003          CALL   inRange ;  (inRange (age))
0035 600004          ADDSP  4,i    ;pop parameter and retVal
0038 C3FFE           LDA    -2,s    ;load retVal
003B 0A0044          BREQ   else2   ;branch if retVal == false (i.e. 0)
003E 41004B then2:  STRO   msg1,d  ;  cout << "Qualified\n"
0041 040047          BR     endif2
0044 410056 else2:  STRO   msg2,d  ;  cout << "Unqualified\n"
0047 600002 endif2: ADDSP  2,i    ;deallocate local
004A 00               STOP
004B 517561 msg1:   .ASCII  "Qualified\n\x00"
...
0056 556E71 msg2:   .ASCII  "Unqualified\n\x00"
...
0063                 .END
```

# Chapter 6: Call by Reference

# Call-by-reference

- Call by reference.
  - Formal parameter **points** to the actual parameter.
- Assembly level
  - Must push the **address** of the actual parameter onto the stack.
  - The called subroutine must use the value on the stack as an **address**.
- C++ (but not C) provides call-by-reference

# Call-by-reference

- Function gets a reference to the actual parameter
  - Example: passing a object to a function
- Reference = address
- Push the variable's address onto the stack
- In order for the function to use it, it needs *stack relative deferred mode*:
  - Mem [ Mem [ SP + Oprnd ] ]

# Call-by-reference

## Call-by-Reference Parameters

Parameter Type	Push <i>actual parameter</i> onto stack	Access <i>formal parameter</i> inside function
Global call-by-reference	LDA glob_var,i STA offset,s	Stack-relative deferred mode
Local call-by-reference	MOVSPA ADDA local_var,i STA offset,s	Stack-relative deferred mode

# Indexed Addressing and Arrays

- Assembly level: **calling**
  - Place **address** of the formal parameter onto stack.
    - Use the symbol with **immediate** addressing
- Assembly level: **called**
  - Access the formal parameter using **stack-relative deferred addressing**.

# Stack-relative deferred addressing

- Oprnd - Mem[Mem[SP + OprndSpec]]
  - Mem[SP + OprndSpec] becomes the address that is used to find the operand
- Addressing mode identifier: **sf**
  - ; this instr uses stack-relative deferred addressing to
  - ; load the acc
  - LDA r, sf**
  - ; this instr takes the result and stores it in a local var on
  - ; the stack
  - STA temp,s**

# A procedure call with a parameter called by reference at Level HOL6 and Level Asmb5

## fig. 6.27

```
#include <iostream>
using namespace std;

int a, b;

void swap (int& r, int& s) {
    int temp;
    temp = r;
    r = s;
    s = temp;
}

void order (int& x, int& y) {
    if (x > y) {
        swap (x, y);
    } // ra2
}
```

# A procedure call with a parameter called by reference at Level HOL6 and Level Asmb5

```
int main () {
    cout << "Enter an integer: ";
    cin >> a;
    cout << "Enter an integer: ";
    cin >> b;
    order (a, b);
    cout << "Ordered they are: " << a << ", " << b << endl; // ral
    return 0;
}
```



# A procedure call with a parameter called by reference at Level HOL6 and Level Asmb5 (*Cont'd*)

```
0000 04003C          BR      main
0003 0000  a:        .BLOCK  2           ;global variable
0005 0000  b:        .BLOCK  2           ;global variable
;
;***** void swap (int& r, int& s)
r:       .EQUATE 6           ;formal parameter
s:       .EQUATE 4           ;formal parameter
temp:   .EQUATE 0           ;local variable
0007 680002 swap:     SUBSP   2,i         ;allocate local
000A C40006             LDA     r,sf        ;temp = r
000D E30000             STA     temp,s
0010 C40004             LDA     s,sf        ;r = s
0013 E40006             STA     r,sf
0016 C30000             LDA     temp,s       ;s = temp
0019 E40004             STA     s,sf
001C 5A                 RET2              ;deallocate local, pop retAddr
```

# A procedure call with a parameter called by reference at Level HOL6 and Level Asmb5 (Cont'd)

```
;***** void order:  
x: .EQUATE 4  
y: .EQUATE 2  
001D C40004 order: LDA x,sf ;if (x > y)  
0020 B40002 STA y,sf  
0023 06003B BRLE endIf  
0026 C30004 LDA x,s ; push x  
0029 E3FFFFE STA -2,s  
002C C30002 LDA y,s ; push y  
002F E3FFFC STA -4,i  
0032 680004 SUBSP 4,i  
0035 160007 CALL swap  
0038 600004 ADDSP 4,i  
003B 58 endIf: RETO
```

load x using stack-deferred addressing. Compare with y using stack-deferred addressing

load x and y from stack (they contain addresses) to new position on stack (for next call)

# A procedure call with a parameter called by reference at Level HOL6 and Level Asmb5 (Cont'd)

```
003C 41006D main:    ;***** main ()  
003F 310003          STRO   msg1,  
0042 41006D          DECI   a,d  
0045 310005          STRO   msg1,  
0048 C00003          DECI   b,d      ;cin >> b  
004B E3FFE           LDA    a,i      ;push the address of a  
004E C00005          STA    -2,s  
0051 E3FFFC          LDA    b,i      ;push the address of b  
0054 680004          STA    -4,s  
0057 16001D          SUBSP  4,i      ;push params  
005A 600004 ral:    CALL   order     ;order (a, b)  
005D 410080          ADDSP  4,i      ;pop params  
0060 390003          STRO   msg2,d  ;cout << "Ordered they are: "  
0063 410093          DECO   a,d      ;      << a  
0066 390005          STRO   msg3,d  ;      << ", "  
0069 50000A          DECO   b,d      ;      << b  
006C 00              CHARO  '\n',i   ;      << endl  
006D 456E74 msg1:   STOP  
0080 4F7264 msg2:   .ASCII  "Enter an integer: \x00"  
0093 2C2000 msg3:   ...  
0096 00              .ASCII  "Ordered they are: \x00"  
0097 00              ...  
0098 00              .END
```

load *address* of **a** and **b**  
onto the stack; use  
immediate addressing

# Call-by-reference with local variables fig. 6-29

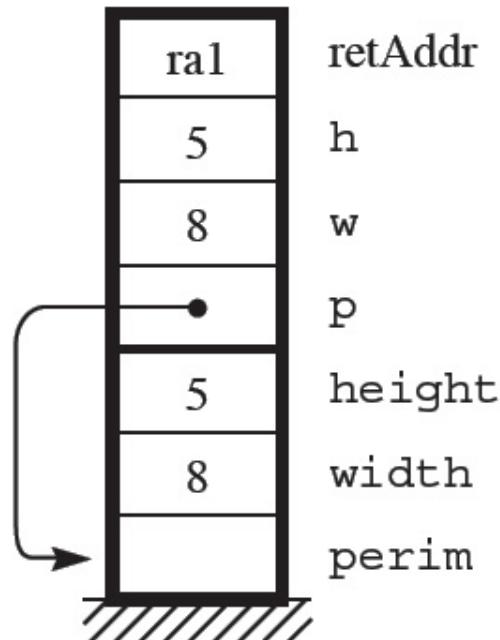
```
#include <iostream>
using namespace std;

void rect (int& p, int w, int h) {
    p = (w + h) * 2;
}

int main () {
    int perim, width, height;
    cout << "Enter width: ";
    cin >> width;
    cout << "Enter height: ";
    cin >> height;
    rect (perim, width, height);
    // rail
    cout << "perim = " << perim << endl;
    return 0;
}
```

# Call-by-reference with local variables

- Problem: the parameter p is a reference variable
  - But the variable it points to, perim is on the stack
  - How can we point to the stack?



# Call-by-reference with local variables

- Cannot load the name with immediate addressing

```
LDA perim, i  
STA -2, s
```

- **perim** is 4, the distance from the top of the stack
  - This is not the address of **perim**!
  - Need to load the stack address to the proper place on the stack for the parameter **perim**

# Call-by-reference with local variables

- New instruction: **MOVSPA**
  - Moves the contents of the SP to A:  $A \leftarrow SP$
  - There is **no** **MOVSPX**; cannot move SP to X
- To load a reference variable from the stack to the stack:

**MOVSPA**

; address must be offset from top of stack

**ADDA perim, i**

**STA -2, s**

# Call-by-reference with local variables

- To use a reference variable:

; stores the contents of the accumulator  
; into a reference variable

STA p, sf

# Call-by-reference with local variables

```
0000 04000E          BR      main
;
;***** void rect (int& p, int w, int h)
p:     .EQUATE 6           ;formal parameter
w:     .EQUATE 4           ;formal parameter
h:     .EQUATE 2           ;formal parameter
0003 C30004 rect:    LDA    w,s
0006 730002            ADDA   h,s
0009 1C                ASLA
000A E40006            STA    p,sp
000D 58                endif: RETO
                                ;pop retAddr
```

Store the accumulator in the referenced variable (the var p that is local to main)

# Call-by-reference with local variables

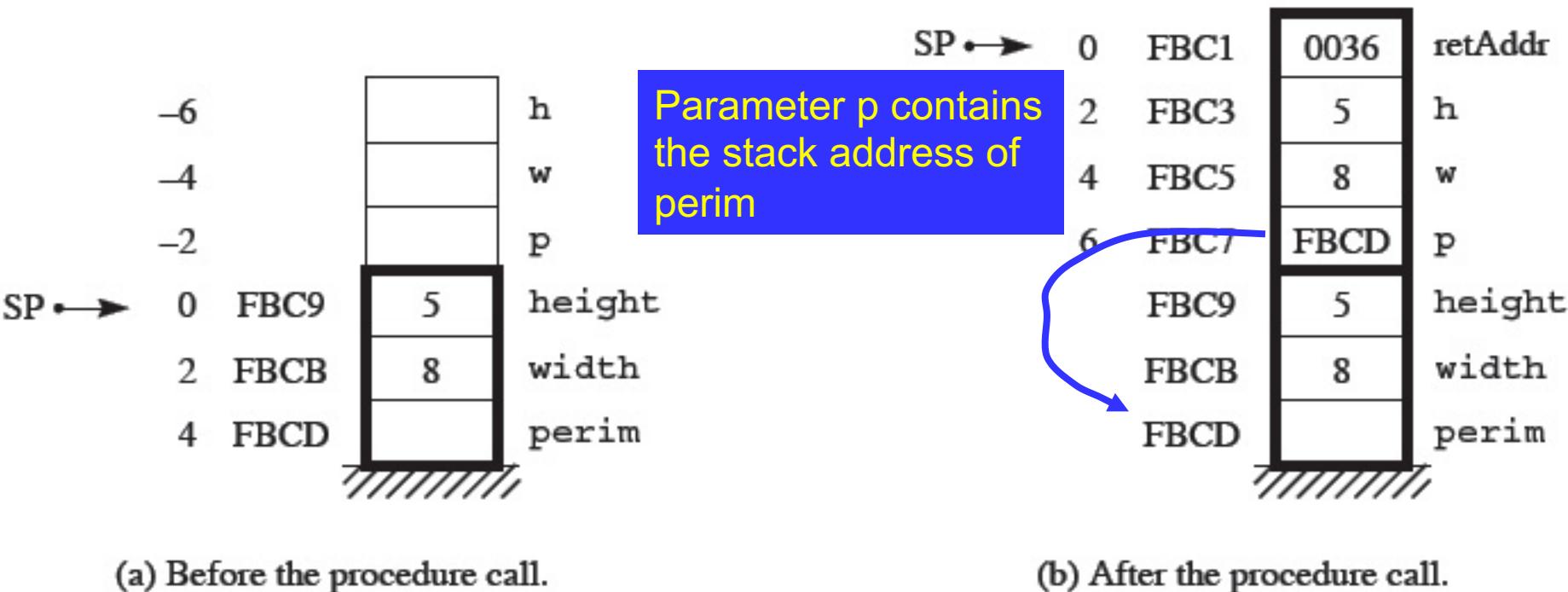
```
;***** main ()  
perim: .EQUATE 4 ;local variable  
width: .EQUATE 2 ;local variable  
height: .EQUATE 0 ;local variable  
000E 680006 main: SUBSP 6,i ;allocate locals  
0011 410046 STRO msg1,d ;cout << "Enter width: "  
0014 330002 DECI width,s ;cin >> width  
0017 410054 STRO msg2,d ;cout << "Enter height: "  
001A 330000 DECI height,s ;cin >> height  
001D 02 MOVSPA ;push the address of perim  
001E 700004 ADDI perim,i ;push the value of width  
0021 E3FFE STA -4,s ;push the value of height  
0024 C30002 Stc ;push params  
0027 E3FFFC int Add the offset so that the  
002A C30000 Store the completed address ;push the value of height  
002D E3FFFA ;push params  
0030 680006 ;rect (perim, width, height)  
0033 160003 ral: ADDSP 6,i ;pop params  
0036 600006 ;cout << "perim = "  
0039 410063 STRO msg3,d ;<< perim  
003C 3B0004 DECO perim,s ;<< endl  
003F 50000A CHARO '\n',i ;deallocate locals  
0042 600006 ADDSP 6,i ;deallocate locals  
0045 00 STOP
```

Stc :push params  
int Add the offset so that the  
Store the completed address onto the stack as the first parameter.  
;push params  
;rect (perim, width, height)  
;pop params  
;cout << "perim = "  
;<< perim  
;<< endl  
;deallocate locals  
;deallocate locals

# Call-by-reference with local variables

```
0046 456E74 msg1:    .ASCII  "Enter width: \x00"  
...  
0054 456E74 msg2:    .ASCII  "Enter height: \x00"  
...  
0063 706572 msg3:    .ASCII  "perim = \x00"  
...  
006C                      .END
```

# Call-by-reference with local variables



(a) Before the procedure call.

(b) After the procedure call.

# Chapter 6: Call by Value

# Pushing Parameters

- To push a global parameter:
  - Use LDA with direct addressing
  - Then use STA with stack-relative addressing

**LDA**      num,d

**STA**      -2, s

# Pushing Parameters onto the Stack

- Which addressing method to use depends upon whether:
  - Variables are Global or Local
  - Parameters are Call-by-value or Call-by-reference

<i>Addressing Mode</i>	<i>RTL Description</i>
Immediate	OprndSpec
Direct	Mem [OprndSpec]
Stack Relative	Mem [SP + OprndSpec]
Stack Relative Deferred	Mem [ Mem [SP + OprndSpec]

# Call-by-Value Parameters

Parameter Type	Push <u>actual parameter</u> onto stack	Access <u>formal parameter</u> inside function
Global call-by-value	LDA glob_var,d STA offset,s	Stack-relative mode
Local call-by-value	LDA local_var,s STA offset,s	Stack-relative mode

- Note: A local variable is already on the stack.  
If we use it as a function parameter,  
it gets copied from one stack frame to another.

# Call-by-Value Parameters

## Stack Frame Addressing Observations

- The compiler decides in advance the stack frame structure.
- It uses .EQUATE to reference each data item's stack offset value.
- The *Called* procedure can access the entire stack frame, but...
- The *Calling* procedure should have access to only the parameters and the return value.
- So, each will have its own .EQUATE values.  
The *Calling* procedure and *Called* procedure might use different offsets for the same data object (because SP will shift during CALL / RET)

# A procedure call with a parameter at Level HOL6 and Level Asmb5: Call by Value with Global Variable Figure 6.21

```
#include <iostream>
using namespace std;
    int numPts;
    int value;
    int j;

void printBar (int n) {
    int k;
    for (k = 1; k <= n; k++) {
        cout << "*";
    }
    cout << endl;
}
int main () {
    cin >> numPts;
    for (j = 1; j <= numPts; j++) {
        cin >> value;
        printBar (value);
    }
    return 0;
}
```

# A procedure call with a parameter at Level HOL6 and Level Asmb5 (*Cont'd*)

```
0000 04002B          BR      main
0003 0000  numPts:   .BLOCK  2           ;global variable
0005 0000  value:    .BLOCK  2           ;global variable
0007 0000  i:        .BLOCK  2           ;global variable
0008
;
;***** void printBar (int n)
n:       .EQUATE 4
j:       .EQUATE 0
0009 680002 printBar: SUBSP  2,i        Allocate local variable meter
000C C00001           LDA     1,i        ,local variable
000F E30000           STA     j,s
0012 B30004 for1:    CPA     n,s        ;j <= n
0015 100027           BRGT   endFor1
0018 50002A           CHARO  '*',i      ; cout << '*'
001B C30000           LDA     j,s        ;j++)
001E 700001           ADDA   1,i
0021 E30000           STA     j,s
0024 040012           BR     fo1
0027 50000A endFor1: CHARO  '\
002A 5A               RET2            dellocate local variable and return
                                         ;deallocate local, pop retAddr
```



# A procedure call with a parameter at Level HOL6 and Level Asmb5 (*Cont'd*)

		;	***** main ()	
002B	310003	main:	DECI	numPts,d ;cin >> numPts
002E	C00001		LDA	1,i ;for (i = 1
0031	E10007		STA	i,d
0034	B10003	for2:	CPA	numPts,d ;i <= numPts
0037	100058		BRGT	
003A	310005		DECI	Push actual parameter value,d ;cin >> value
003D	C10005		LDA	value,d ; call by value
0040	E3FFE		STA	-2,s
0043	680002		SUBSP	2.i : push parameter
0046	160009		CALL	Pop actual parameter push retAddr
0049	600002		ADDSP	2,i ; pop parameter
004C	C10007		LDA	i,d
004F	700001		ADDA	1,i
0052	E10007		STA	i,d
0055	040034		BR	for2
0058	00	endFor2:	STOP	
0059			.END	

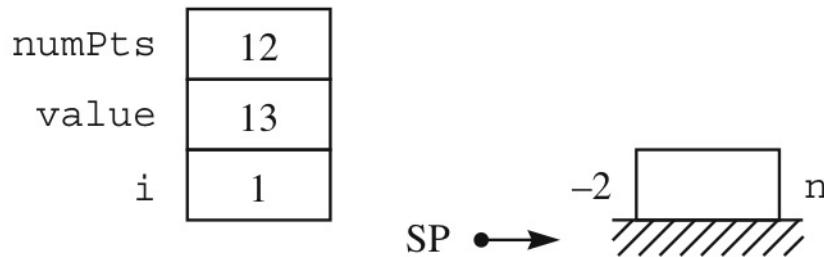
Push actual parameter

Pop actual parameter

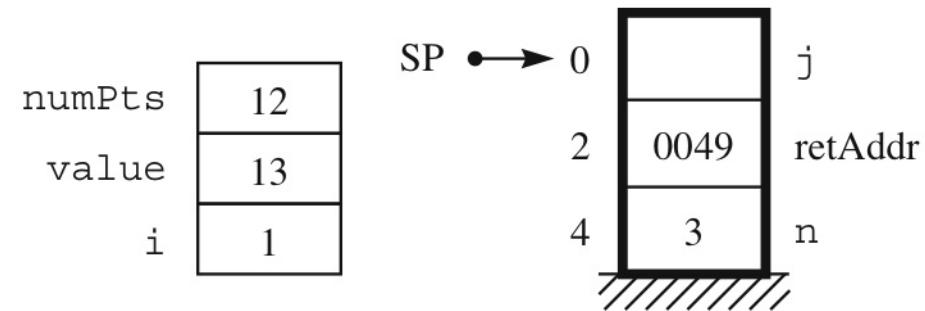
In the main program, we access global variables with Direct mode.

Within a function, we access parameters with some variation of Stack mode.

# The run-time stack for Program



(a) After `cin >> value`.



(b) After allocation with `SUBSP` in `printBar`.

# A function with Call-By-Value parameters and local Variables

- **Calling:**
  - Push the actual parameters
  - Push the return address
- **Called:**
  - Push storage for the local variables

# Call-by-value with local variables in main() fig. 6.23

```
#include <iostream>
using namespace std;

void printBar (int n) {
    int k;
    for (k = 1; k <= n; k++) {
        cout << '*';
    }
    cout << endl;
}

int main () {
    int numPts;
    int value;
    int j;
    cin >> numPts;
    for (j = 1; j <= numPts; j++) {
        cin >> value;
        printBar (value);
    }
    return 0;
}
```

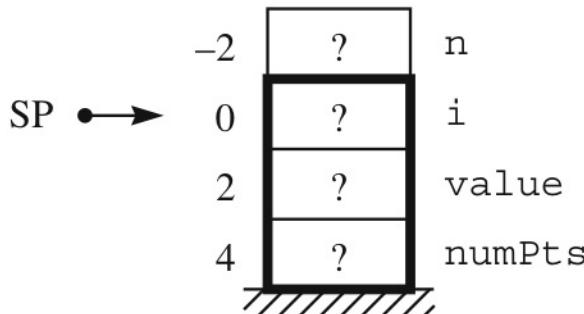
# Call-by-value with local variables

```
0000 040025          BR      main
;
;***** void printBar (int n)
n:     .EQUATE 4           ;formal parameter
j:     .EQUATE 0           ;local variable
0003 680002 printBar:SUBSP 2,i       ;allocate local
0006 C00001             LDA    1,i       ;for (j = 1
0009 E30000             STA    j,s
000C B30004 for1:       CPA    n,s       ;j <= n
000F 100021             BRGT   endFor1
0012 50002A             CHARO  '*',i    ;cout << '*'
0015 C30000             LDA    j,s       ;j++)
0018 700001             ADDA   1,i
001B E30000             STA    j,s
001E 04000C             BR     for1
0021 50000A endFor1:  CHARO  '\n',i    ;cout << endl
0024 5A                 RET2
;deallocate local, pop retAddr
```

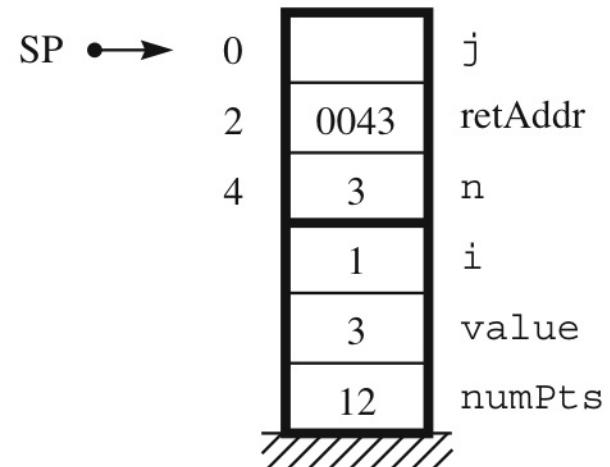
# Call-by-value with local variables

```
;***** main ()  
numPts: .EQUATE 4 ;local variable  
value: .EQUATE 2 ;local variable  
i: .EQUATE 0 ;local variable  
0025 680006 main: SUBSP 6,i ;allocate locals  
0028 330004 DECI numPts,s ;cin >> numPts  
002B C00001 LDA 1,i ;for (i = 1  
002E E30000 STA i,s  
0031 B30004 for2: CPA numPts,s ;i <= numPts  
0034 100055 BRGT endFor2  
0037 330002 DECI value,s ; cin >> value  
003A C30002 LDA value,s ; call by value  
003D E3FFE STA -2,s  
0040 680002 SUBSP 2,i ; push parameter  
0043 160003 CALL printBar ; push retAddr  
0046 600002 ADDSP 2,i ; pop parameter  
0049 C30000 LDA i,s ;i++)  
004C 700001 ADDA 1,i  
004F E30000 STA i,s  
0052 040031 BR for2  
0055 600006 endFor2: ADDSP 6,i ;deallocate locals  
0058 00 STOP  
0059 .END
```

# Call-by-value with local variables



(a) After `cin >> value.`



(b) After allocation with `SUBSP` in `printBar.`

# A recursive function at Level HOL6 and Level Asmb5

- **Calling:**

- Push storage for the returned value
  - Push the actual parameters
  - Push the return address

- **Called:**

- Push storage for the local variables

# A recursive function at Level HOL6 and Level Asmb5 fig 6.25

```
int binCoeff (int n, int k) {
    int y1, y2;
    if ((k == 0) || (n == k)) {
        return 1;
    }
    else {
        y1 = binCoeff (n - 1, k); // ra2
        y2 = binCoeff (n - 1, k - 1); // ra3
        return y1 + y2;
    }
}

int main () {
    cout << "binCoeff (3, 1) = " << binCoeff (3, 1); // ra1
    cout << endl;
    return 0;
}
```

# A recursive function at Level HOL6 and Level Asmb5

```
0000 040065          BR      main
;
;***** int binomCoeff (int n, int k)
 retVal: .EQUATE 10           ;returned value
 n:     .EQUATE 8            ;formal parameter
 k:     .EQUATE 6            ;formal parameter
 y1:    .EQUATE 2            ;local variable
 y2:    .EQUATE 0            ;local variable
0003 680004 binCoeff:SUBSP 4,i       ;allocate locals
0006 C30006 if:      LDA   k,s        ;if ((k == 0)
0009 0A0015           BREQ  then
000C C30008           LDA   n,s        ;|| (n == k))
000F B30006           CPA   k,s
0012 0C001C           BRNE  else
0015 C00001 then:     LDA   1,i        ;return 1
0018 E3000A           STA   retVal,s
001B 5C               RET4           ;deallocate locals, pop retAddr
```

# A recursive function at Level HOL6 and Level Asmb5

```
001C C30008 else:  
001F 800001  
0022 E3FFF C  
0025 C30006  
0028 E3FFF A  
002B 680006  
002E 160003  
0031 600006 ra2:  
0034 C3FFF E  
0037 E30002  
003A C30008  
003D 800001  
0040 E3FFF C  
0043 C30006  
0046 800001  
0049 E3FFF A  
004C 680006  
004F 160003  
0052 600006 ra3:  
0055 C3FFF E  
0058 E30000  
005B C30002  
005E 730000  
0061 E3000A  
0064 5C endIf:  
          LDA    n,s  
          SUBA   1,i  
          STA    -4,s  
          LDA    k,s  
          STA    -6,s  
          SUBSP  6,i  
          CALL   binCoeff  
          ADDSP  6,i  
          LDA    -2,s  
          STA    y1,s  
          LDA    n,s  
          SUBA   1,i  
          STA    -4,s  
          LDA    k,s  
          SUBA   1,i  
          STA    -6,s  
          SUBSP  6,i  
          CALL   binCoeff  
          ADDSP  6,i  
          LDA    -2,s  
          STA    y2,s  
          LDA    y1,s  
          ADDA   y2,s  
          STA    retVal,s  
          RET4
```

;push n - 1

First recursive call  
same structure as the  
subroutine call in main

; Pop the stack on return  
from subroutine call.  
; Get the returned value  
from previous stack frame  
; and save on this stack frame.  
; same structure as the  
subroutine call in main

;push params and retVal

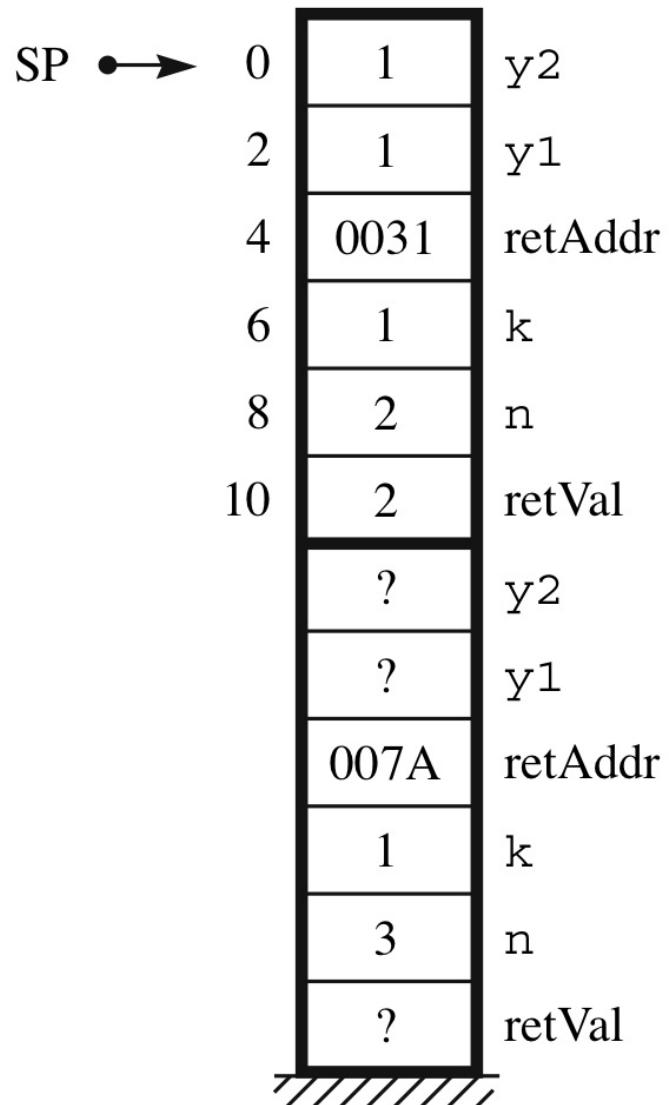
; Pop the stack on return  
from subroutine call.  
Pop off local variables  
and return (automatically  
pops off the return addr). ie.

# A recursive function at Level HOL6 and Level Asmb5

```
0065 410084 main:    STPO    msg,d
0068 C00003          LDA     3,i
006B E3FFFC          STA     -4,s
006E C00001          LDA     1,i
0071 E3FFFFA         STA     -6,s
0074 680006          SUBSP   6,i
0077 160003          CALL    binCoeff
007A 600006 ral:    ADDSP   6,i
007D 3BFFFFE          DECO    -2,s
0080 50000A          CHARO   '\n',i
0083 00              STOP
0084 62696E msg:    .ASCII  "binCoeff (3, 1
436F65
666620
28332C
203129
203D20
00
0097               .END
```

; cout << "binCoeff (3, 1) = "  
; Store value for first  
; Store value for second  
; ...  
; Jump to subroutine.  
; On return must pop off  
; Print the returned value.  
; Note that we can still  
access it even though the  
SP register has been  
modified to pop it off.

# The run-time stack of Program 6.10 immediately after the second return



# **Chapter 6: Compiling to the Assembly Level Part 1 (Sections 6.1-6.3)**

# Objectives

- Show relationship between high-order languages and assembly language
  - Show how a compiler might translate the C++ code
  - Learn new assembly language techniques
  - Assembly language programs shown will mimic the C++ programs
    - Thus the programs are **not** the most efficient implementations!

# Objectives

- Differences between high-level languages and assembly languages:
  - Assembly has no data types
  - Assembly has only branch statements, no decision or flow control statements like **if** and **while**

# 6.1 Local Variables & Stack Addressing

## Learning Objectives:

- Use Stack-relative Addressing Mode to push and pop local variables
- Compare and contrast the declaration and accessing of three kinds of variables:
  - Global variables
  - Local variables
  - Constants

# Local Variables & Stack Addressing

- Recall Ch. 2 variables & memory:
  - Global variables stored in program memory
  - Local variables stored on stack
- Run-Time Stack
  - Memory usage varies during execution
  - For local variables
  - For procedure/function calls

# Stack Relative Addressing

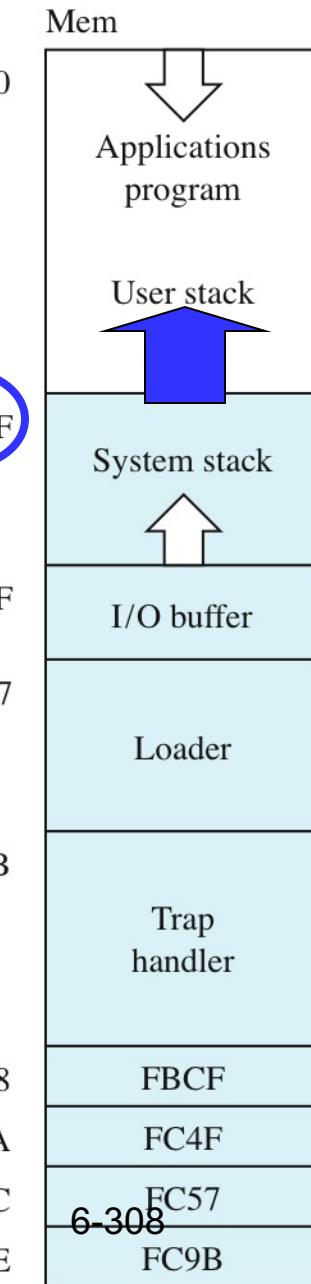
- Stack Relative Addressing allows programs to put data on the run-time stack
  - Program calling a subroutine must push parameters onto the stack
  - Subroutine must push local variables onto the stack

# Hardware: Stack Organization & Pointer

- Last in, First out
- Stack Pointer Register (SP) contains the address of the last item
- Loader initializes  $SP \leftarrow \text{Mem}[FFF8] = \text{FBCE}$
- As stack grows  $\rightarrow$  address decreases

# Run-time stack in Pep/8

FBCF is the initial top  
of the user stack



# Stack Relative Addressing

- **Stack-Relative Addressing:**

$$\text{Oprnd} = \text{Mem}[\text{SP} + \text{OprndSpec}]$$

- Stack pointer acts as a memory address to which the operand specifier is added
- Operand specifier is the **offset** from the top of the stack

# Stack Relative Addressing

- **Operand specifier** is the **offset** from the top of the stack
  - If the operand specifier is 0, the instruction accesses the top of the stack.
  - If the operand specifier is 2, it accesses  $\text{Mem}[\text{SP}+2]$ , the value two bytes **below** the top of the stack.
  - If the operand specifier is -2, it accesses  $\text{Mem}[\text{SP}-2]$ , the value two bytes **above** the top of the stack.

# Stack Relative Addressing

- ADDSP and SUBSP manipulate the stack pointer directly
  - Must use immediate addressing mode with these instructions
  - ADDSP adds a value to the stack pointer
  - SUBSP subtracts a value from the stack pointer
- Run-Time stack grows toward low addresses
  - To allocate storage on the stack, subtract a value with SUBSP
  - To deallocate storage, you add a value with ADDSP

# Stack Relative Addressing

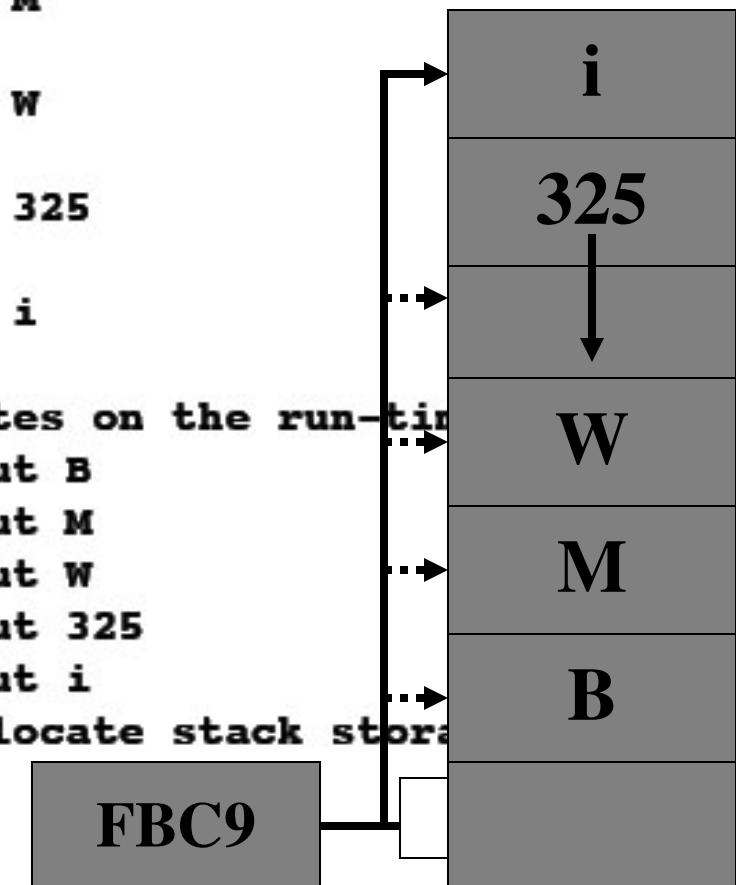
- CALL, RETn, and RETTR manipulate the stack pointer **indirectly**
- Note: There is no way to set the stack pointer by loading a value into it
  - There is **no LDSP instruction**

# Stack Relative Addressing

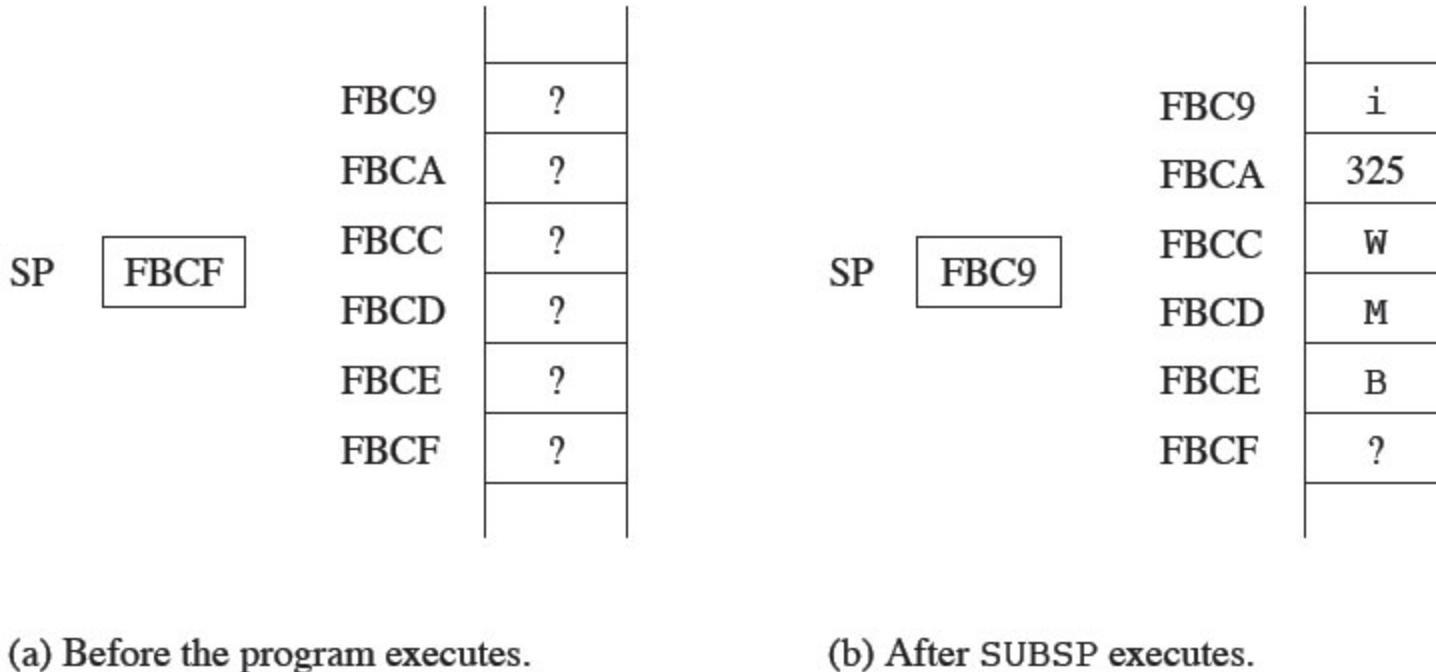
- The SP is initialized automatically when a program is executed
  - $\text{SP} \leftarrow \text{Mem[FFF8]}$
  - $\text{PC} \leftarrow 0000$
- Mem[FFF8] is located in Pep/8 ROM
  - contains the value FBCF, the bottom of the user stack.
  - Can only be changed by burning ROM

# Stack-relative addressing

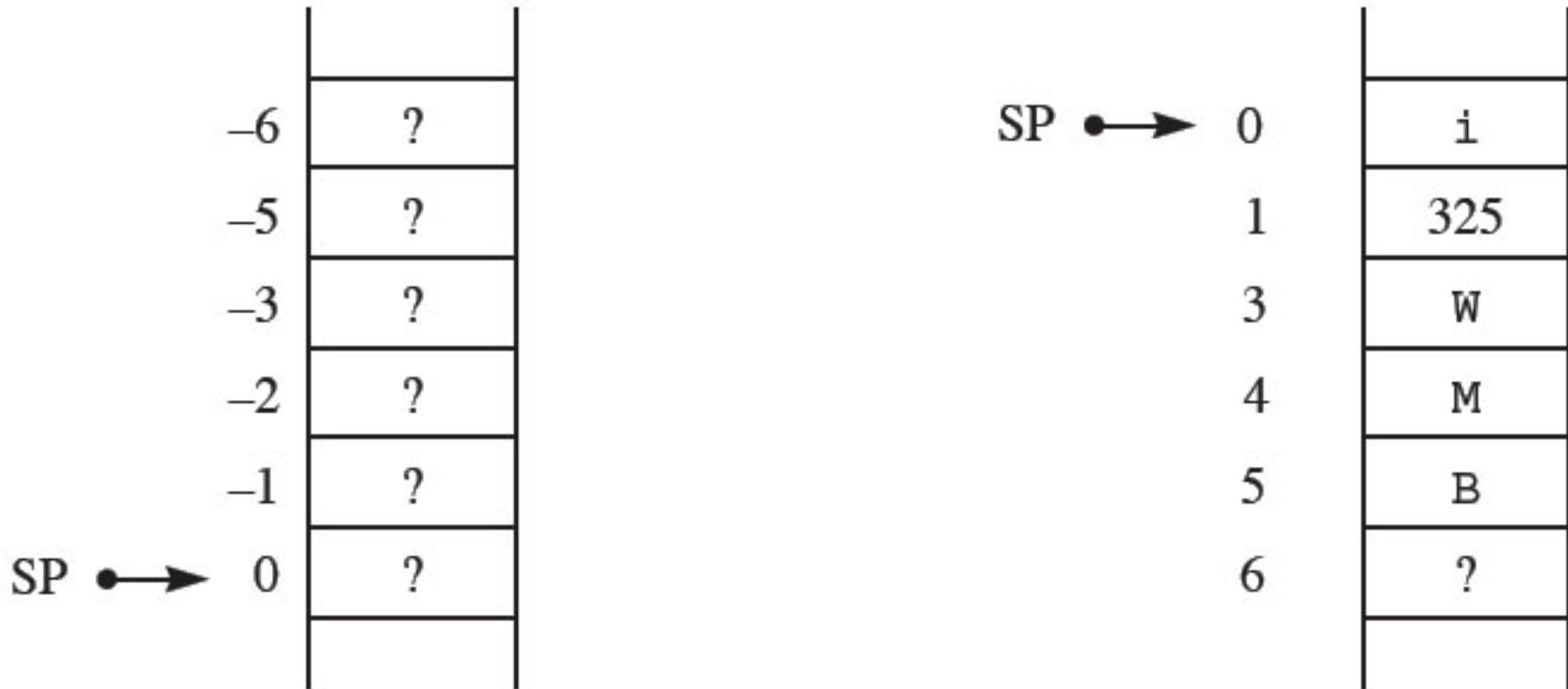
```
0000 C00042 LDA    'B',i      ;push B
0003 F3FFFF STBYTEA -1,s
0006 C0004D LDA    'M',i      ;push M
0009 F3FFFE STBYTEA -2,s
000C C00057 LDA    'W',i      ;push W
000F F3FFF D STBYTEA -3,s
0012 C00145 LDA    325,i      ;push 325
0015 E3FFFF STA    -5,s
0018 C00069 LDA    'i',i      ;push i
001B F3FFFA STBYTEA -6,s
001E 680006 SUBSP  6,i      ;6 bytes on the run-time stack
0021 530005 CHARO  5,s      ;output B
0024 530004 CHARO  4,s      ;output M
0027 530003 CHARO  3,s      ;output W
002A 3B0001 DECO   1,s      ;output 325
002D 530000 CHARO  0,s      ;output i
0030 600006 ADDSP  6,i      ;deallocate stack storage
0033 00      STOP
0034 .END
```



# Pushing “real” onto the stack in Program 6.8



# A simplified diagram of the stack in Program 6.8



# Good/Bad of Stack Relative Addressing

- Good

- Don't need/care about absolute address
- Only use what you need

- Bad

- Risk of unequal pushing/popping

# Translating HOL Local Variables

What does the compiler and/or machine need to know?

- Location of items on the stack
  - Sometimes we want items that aren't on top
  - Offset depends on data type and quantity
- When to move the stack pointer up/down

# Local Variables

- Local variables are stored on the run-time stack
- Method:
  - Allocate local variables with SUBSP
  - Access local variables with stack-relative addressing
  - Deallocate storage with ADDSP when function is finished.

# Local Variables

- Global variables

- .BLOCK command inserts bits in the assembly process
- Storage is reserved in RAM before program executes. Is part of the program
- Location is fixed for duration of program

- Local Variables

- SUBSP is an executable statement
- Space created at run time on run-time stack

# Local Variables

Same as program 5.26 except variables are declared local to *main( )*

```
#include <iostream>
using namespace std;

int main () {
    const int bonus = 5;
    int exam1;
    int exam2;
    int score;
    cin >> exam1 >> exam2;
    score = (exam1 + exam2) / 2 + bonus;
    cout << "score = " << score << endl;
    return 0;
}
```

Bonus is constant

Variables are local

# Local Variables

```
0000 040003      .EQUATE 5          ;constant  
                  .EQUATE 4          ;local variable  
                  .EQUATE 2          ;  
                  .EQUATE 0          ;  
  
0003 680006 main: SUBSP 6,i  
0006 330004      DECI  exam1,s  
0009 330002      DECI  exam2,s  
000C C30004      IDA   exam1,s  
000F 730002      ADDA  exam2,s  
0012 1E          ASRA  
0013 700005      ADDA  bonus,i  
0016 E30000      STA   score,s  
0019 410026      STRO  msg,d      ;cout << "score = "  
001C 3B0000      DECO  score,s  
001F 50000A      CHARO '\n',i  
0022 600006      ADDSP 6,i       ;deallcate locals  
0025 00          STOP  
0026 73636F msg: .ASCII  "score = \x00"  
                  726520  
                  3D2000  
002F             .END
```

Bonus is constant

Set constants for use in program

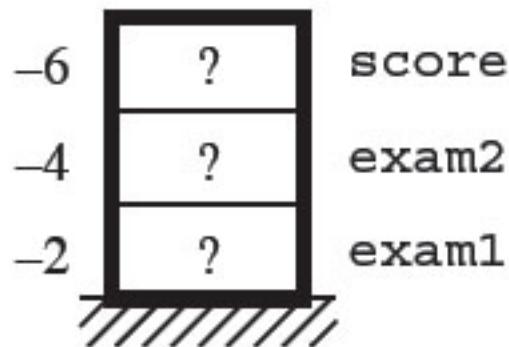
Create space on stack for locals

use constants with stack  
relative addressing

# Local Variables

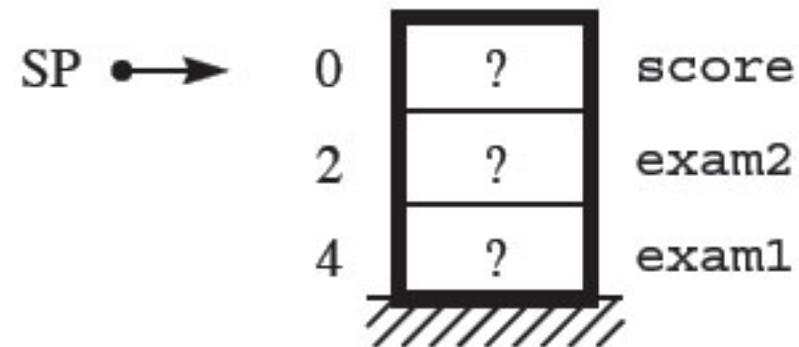
- **.EQUATE**
  - For bonus creates a constant
  - For other variables provides the *distance* from top of stack
  - Does **not** create space on the stack!
- **SUBSP**
  - Provides the space for the local variables on the stack
- **Local variables accessed with stack relative addressing**

# Local Variables



SP →

(a) Before SUBSP executes.



(a) After SUBSP executes.

- Note that there are no values on the stack yet.

# Translating Different Symbols

- Quick summary:

Symbol Genre	Directive	Symbol Value	Access Mode	Memory Allocation Time
Constant	.EQUATE <i>data_value</i>	Data_value	Immediate	Compile
Global variable	.BLOCK <i>#of_bytes</i>	Mem_addr	Direct	Compile
Local variable	.EQUATE Stack offset	Stack offset	Stack relative	Run-time

# 6.2 Branching and Flow of Control

## Learning Objectives:

- **Implement HOL control structures using Assembly Language Conditional Branching**
- **Define and use the CPr and BRx Instructions**
- **State the Structured Program Theorem and understand its significance for programming**

# Branching and Flow of Control

- **HOL Conditional Flow of Control**
  - Selection: IF and SWITCH/CASE
  - Repetition: FOR, WHILE, DO
- **Assembly Level Equivalent**
  - Conditional Branches
  - BR means jump to a new instruction address:  
PC ← Operand

# Conditional Branches

- Pep/8 has 8 varieties
  - Each depends on one or more Status Bits
    - N, Z, C, V
- CPr (comparison) instruction
  - Computes ( $r - \text{Operand}$ ), then
  - Sets the Status Bits, *but*
  - Doesn't store the result (i.e.  $r$  doesn't change)

# Branching

- Branches are based on status bits.
- Branch if the appropriate bits are set or unset
- If bits are set appropriately, place the branch operand into the PC.
- Otherwise do nothing.

# Branching Instructions

- **BRLE** Branch on less than or equal to
- **BRLT** Branch on less than
- **BREQ** Branch on equal to
- **BRNE** Branch on not equal to
- **BRGE** Branch on greater than or equal to
- **BRGT** Branch on greater than
- **BRV** Branch on V
- **BCRC** Branch on C

## Branching Instructions (*Cont'd*)

- BRLE  $N = 1 \vee Z = 1 \Rightarrow PC \leftarrow Oprnd$
- BRLT  $N = 1 \Rightarrow PC \leftarrow Oprnd$
- BREQ  $Z = 1 \Rightarrow PC \leftarrow Oprnd$
- BRNE  $Z = 0 \Rightarrow PC \leftarrow Oprnd$
- BRGE  $N = 0 \Rightarrow PC \leftarrow Oprnd$
- BRGT  $N = 0 \wedge Z = 0 \Rightarrow PC \leftarrow Oprnd$
- BRV  $V = 1 \Rightarrow PC \leftarrow Oprnd$
- BRC  $C = 1 \Rightarrow PC \leftarrow Oprnd$

# Branching

- Branches are based on the last operation that changed the status bits
- So BRLT checks to see if the last operation that changed the status bits was negative (ie,  $N = 1$ )
- BRLE checks to see if the last operation that changed the status bits was negative or equal to 0 (ie,  $N = 1$  OR  $Z = 1$ )
- BRGT checks to see if the last operation that changed the status bits was positive and not equal to 0 (ie,  $N = 0$  AND  $Z = 0$ )

# Branching

- If you want to branch, must first perform some operation that affects the status bits.
- Example:

LDA            num,d

BRLT            Place

- Note that a LDA will affect the status bits.

# IF Structure Flow and Translation

```
// HOL  
  
if (x > 0)  
{  
    total += cost;  
}  
  
cout total;
```

```
// Pep8 Assembly  
  
LDA x,d  
CPA 0,i  
BRLE fi  
  
LDA total,d  
ADDA cost,d  
STA total,d  
  
fi: deco total,d
```

- Note: In this HOL snippet, we test to see if we should enter the block.
- In the Assembly translation, we test to see if we should SKIP the block.

# The if statement at Level HOL6 and Level Asmb5

```
#include <iostream>
Using namespace std;
main(){
    int number;
    cin >> number;
    if (number < 0){
        number = -number;
    }
    cout << number;
}
```

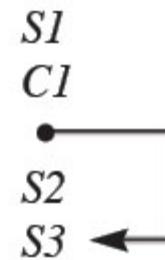
	main	
	number: .EQUATE 0	; local variable
	;	
0003	680002 main:	SUB
0006	330000	DEC
0009	C30000 if:	LDA
000C	0E0016	BRGE
000F	C30000	LDA
0012	1A	NEGA
0013	E30000	STA
0016	3B0000 endIf:	DECO
0019	600002	ADDSP 2,i
001C	00	STOP
001D		.END

**Note the inversion!**

# The if statement at Level HOL6 and Level Asmb5

```
S1  
if (CI) {  
    S2  
}  
S3
```

(a) The structure at Level HOL6.



(b) The structure at level Asmb5  
for Figure 6.6.

# If statements

- To make an if statement at the assembly level work **exactly** like an if statement at the C++ level must:
  - Make the condition relative to 0
    - ie,  $x > y$  becomes  $x - y > 0$
  - Invert the condition.
    - ie,  $x - y > 0$  becomes  $x - y \leq 0$

# Optimizing Compilers

0000	700005	BR
0003	0000	number:
	;	
0005	E90003	Main: DECI
0008	090003	If: LDA
000B	980018	BRGE
000E	090003	LDA
0011	38	NOTA
0012	180001	ADDA
0015	110003	STA
0018	F10003	EndIf: DECO
001B	00	STOP
001C		.END

```
Main
.BLOCK      2
number,d    ; cin >>number
number,d    ;if (number < 0)
EndIf
number,d    ;number = -number
```

## Redundant Instruction!

The value of *number* will still be in the accumulator from previous load at 0008.

# Optimizing Compilers

- Could eliminate the LOADA at address 000E in last program
- Optimizing compilers fix this, others don't
- Compiler must:
  - analyze the previous instructions generated
  - remember the content of the accumulator
  - don't generate a *load* if value hasn't changed

# Optimizing Compilers

- OC take longer to compile
  - If developing software and doing lots of compiles use a non-optimizing compiler
  - If have a large program that will be used by many users, use an optimizing compiler

# Compiling

- Traditionally, assembly language programs that were generated by a compiler were slower than assembly language programs created by hand.
- Not always true now; modern architectures are very complicated

# Compiling

- Why use higher-order languages?
  - Type checking
  - Structured programming is easier to understand.
  - Can be more verbose: can say `cout>>"my long string"`
- The biggest cost in software development is the programmers!

# CPR instruction

- CPr is identical to SUBr except
  - SUBr stores result in R register
  - CPr does **not** store result anywhere.
- Both instructions affect the NZVC bits.

# The if/else statement at Level HOL6 and Level Asmb5 - Program 6.2 (fig. 6.8)

```
#include <iostream>
Using namespace std;

int main( ) {
    const int limit = 100;
    int num;
    cin >> num)
    if (num >= limit) {
        cout << "high";
    }
    else {
        cout << "low";
    }
    return 0;
}
```

# The if/else statement at Level HOL6 and Level Asmb5 - Program 6.8

```
0000 040003      BR      main
                  limit: .EQUATE 100 ;constant
                  num:   .EQUATE 0  ;local variable
                  ;
0003 680002 main:  SUBSP  -4
                  DECI   0           allocate local
0006 330000          LDA    num,s      ;cin >> num
0009 C30000 if:    LDA    limit,i    ;if (num >= limit)
000C B00064          CPA    i
000F 080018          BRLT   -4           branch around else
0012 41001F          STOP
0015 04001B          BR    -4           cout << "high"
0018 410024 else:   STRO   -
001B 600002 endIf:  ADDSP  2,i        ;deallocate local
001E 00              STOP
001F 686967 msg1:  .ASCII  "high\x00"
                  6800
0024 6C6F77 msg2:  .ASCII  "low\x00"
                  00
0028              .END
```

simplify and invert

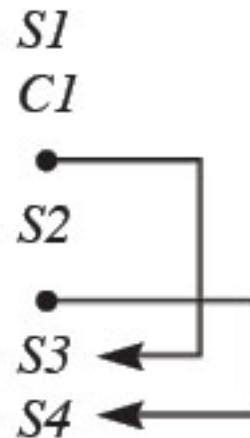
branch around else

branch on the inverted condition

# The if/else statement at Level HOL6 and Level Asmb5 (*Cont'd*)

```
SI  
if (CI) {  
    S2  
}  
else  
    S3  
}  
S4
```

(a) The structure at Level HOL6.



(b) The structure at level Asmb5  
for Figure 6.8.

# The **while** statement at Level HOL6 and Level Asmb5 (fig. 6.10)

```
#include <iostream>
Using namespace std;

char letter;
main( ) {
    cin >> letter;
    while (letter != '*')
    {
        cout << letter;
        cin >> letter;
    }
    return 0;
}
```

# The while statement at Level HOL6 and Level Asmb5

0000 040004		BR	main
0003 00	letter:	.BLOCK	1
	;		
0004 490003	main:	CHART	1
0007 C00000		LDA	0
000A D10003	while:	LDBYTEA	i
000D B0002A		CPA	'*' , i
0010 0A001C		BREQ	endWh
0013 510003		CHARO	
0016 490003		CHARI	
0019 04000A		BR	1 Test inverse condition
001C 00	endWh:	STOP	Unconditional jump to top
001D		.END	of loop to test condition again

Clear accumulator. Why?

Simplify condition:

letter - '\*' != 0

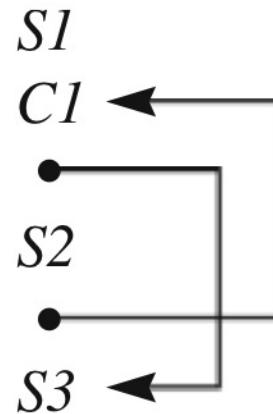
1 Test inverse condition

Unconditional jump to top  
of loop to test condition again

# The while statement at Level HOL6 and Level Asmb5

```
S1  
while (C1) {  
    S2  
}  
S3
```

(a) The structure at level HOL6.



(b) The structure at level Asmb5 for Figure 6.10.

# The do statement at Level HOL6 and Level Asmb5 (Fig. 6.12)

```
#include <iostream>
using namespace std;

int cop;
int driver;
main( ) {
    cop = 0;
    driver = 40;
    do {
        cop += 25;
        driver += 20;
    } while (cop < driver);
    cout << cop;
}
```

# The do statement at Level HOL6 and Level Asmb5 (Fig. 6.12)

0000	040007	BR	main	
0003	0000	cop:	.BLOCK 2	;global variable
0005	0000	driver:	.BLOCK 2	;global variable
		;		
0007	C00000	main:	LDA 0,i	;cop = 0
000A	E10003		STA cop,d	
000D	C00028		LDA 40,i	;driver = 40
0010	E10005		STA driver,d	
0013	C10003	do:	LDA cop,d	; cop += 25
0016	700019		ADDA 25,i	
0019	E10003		STA	
001C	C10005		LDA	
001F	700014		ADDA	
0022	E10005		STA	
0025	C10003	while:	LDA cop,d	;while (cop < driver)
0028	B10005		CPA	
002B	080013		BRLT	
002E	390003		DECO	
0031	00		STOP	
0032			.END	

Simplify:  
 $\text{cop} - \text{driver} < 0$        $\text{driver} += 20$

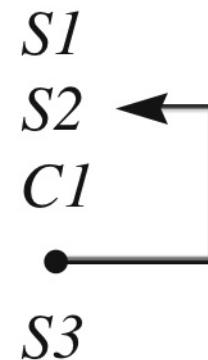
Test the same condition  
Do not transform      op  
Why?

A red circle highlights the LDA and BRLT instructions.

# The do structure

```
S1  
do {  
    S2  
}  
while (C1)  
S3
```

(a) The structure at level HOL6.



(b) The structure at level Asmb5 for Figure 6.12.

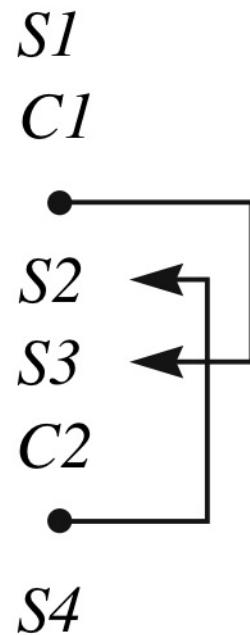
# The for statement at Level HOL6 and Level Asmb5 (Fig. 6.14)

```
#include <iostream>
using namespace std;
main ( ) {
    int j;
    for (j = 0; j < 3; i++)
    {
        cout << "j = " << j << endl;
    }
    cout << "j = " << j << endl;
return 0;
}
```

## The for statement at Level HOL6 and Level Asmb5

0000	040003		BR main	
	i:	.EQUATE 0		; local variable
	;			
0003	680002	main:	SUBSP	initialization
0006	C00000		LDA	
0009	E30000		STA	
000C	B00003	for:	CBA	
000F	0E0027		BRGE	
0012	410034		STRO	
0015	3B0000		DECO	
0018	50000A		CHARO	increment
001B	C30000		LDA	i,s
001E	700001		ADDA	
0021	E30000		STA	
0024	04000C		BR	branch to top of loop
0027	410034	endFor:	STRO	for
002A	3B0000		DECO	msg,d ; cout << "i = "
002D	50000A		CHARO	i,s ; << i
0030	600002		ADDSP	'\n',i ; << endl
0033	00			2,i ; deallocate local
0034	69203D	msg:	.ASCII	"i = \x00"
	2000			
0039			.END	

# The structure of the for



# Multiple Conditions

- Often necessary to have multiple conditions in a control statement

```
#include <iostream>
using namespace std;

int num1, num2;
main (){
    cin >> num1, num2;
    if ((num1 > 0) && (num2 > 0))
        cout << "Positive";
    else
        cout << "Negative";
}
```

# Multiple Conditions

0000	040007	BR	main`	
0003	0000		num1:.BLOCK 2	
0005	0000		num2:.BLOCK 2	
0007	310003	main:	DECI	num1,d ;cin >>num1,num2)
000A	310005		DECI	num2,d
000D	C10003		LDA	num1,d;num1 > 0
0010	06001F		BRLE	else
0013	C10005	test2:	LDA	num2,d; num2 > 0
0016	06001F		BRLE	else
0019	410023		STRO	pos,d ; the if body
001C	040022		BR	done
001F	41002C	else:	STRO	neg,d ; the else body
0022			done: STOP	
0023			.ASCII "Positive\00"	
002C			.ASCII "Negative\00"	
0023			.END	

# Multiple Conditions

- Often necessary to have multiple conditions in a control statement

```
#include <iostream>
using namespace std;

int num1, num2;
main (){
    cin >> num1,num2;
    if ((num1 > 0) || (num2 > 0))
        cout << "P";
    else
        cout << "N";
}
```

# Multiple Conditions

```
0000 040007 BR main
0003 0000 num1: .BLOCK 2
0005 0000 num2: .BLOCK 2
0007 310003 main: DECI num1,d ; cin >> num1,num2)
000A 310005      DFCI num2,d
000D C10003      LDA num1,d ;num1 > 0
0010 10001F      BRGT if
0013 C10005 test2: LDA num2,d ; num2 > 0
0016 10001F      BRGT if
0019 41002C      STRO neg,d ; the else body
001C 040022      BR done
001F 410023 if:   STRO pos,d ; the if body
0022           done: STOP
0023           .ASCII "Positive\00"
002C           .ASCII "Negative\00"
0023           .END
```

# Other control structures

- At assembly level can write control structures that do not correspond to any high level control structure.
- See Mystery Program

# A mystery program

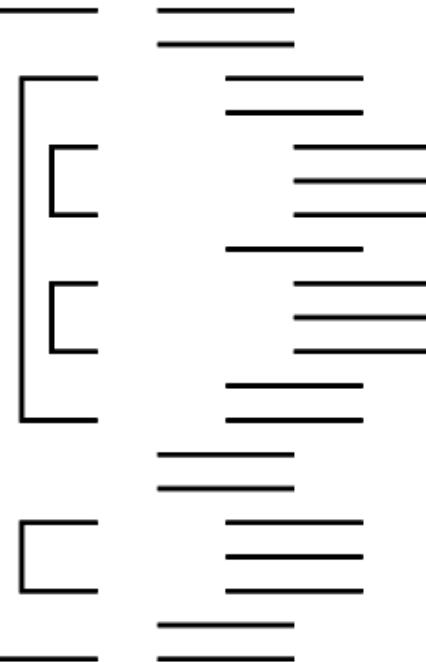
0000	040009		BR	main
0003	0000	n1:	.BLOCK	2
0005	0000	n2:	.BLOCK	2
0007	0000	n3:	.BLOCK	2
		;		
0009	310005	main:	DECI	n2,d
000C	310007		DECI	n3,d
000F	C10005		LDA	n2,d
0012	B10007		CPA	n3,d
0015	08002A		BRLT	L1
0018	310003		DECI	n1,d
001B	C10003		LDA	n1,d
001E	B10007		CPA	n3,d
0021	080074		BRLT	L7
0024	040065		BR	L6
0027	E10007		STA	n3,d
002A	310003	L1:	DECI	n1,d
002D	C10005		LDA	n2,d
0030	B10003		CPA	n1,d
0033	080053		BRLT	L5
0036	390003		DECO	n1,d
0039	390005		DECO	n2,d
003C	390007	L2:	DECO	n3,d
003F	00		STOP	



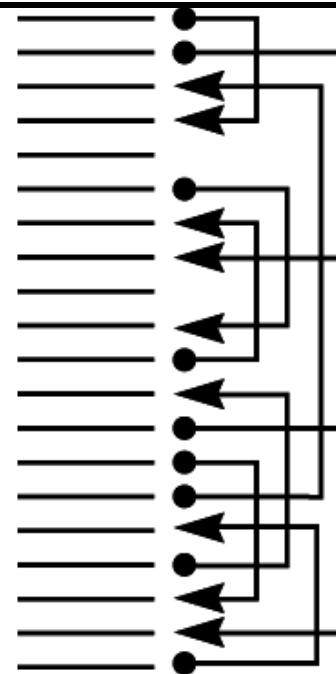
# A mystery program(cont)

0040	390005	L3:	DECO	n2,d
0043	390007		DECO	n3,d
0046	040081		BR	L9
0049	390003	L4:	DECO	n1,d
004C	390005		DECO	n2,d
004F	00		STOP	
0050	E10003		STA	n1,d
0053	C10007	L5:	LDA	n3,d
0056	B10003		CPA	n1,d
0059	080040		BRLT	L3
005C	390005		DECO	n2,d
005F	390003		DECO	n1,d
0062	04003C		BR	L2
0065	390007	L6:	DECO	n3,d
0068	C10003		LDA	n1,d
006B	B10005		CPA	n2,d
006E	080049		BRLT	L4
0071	04007E		BR	L8
0074	390003	L7:	DECO	n1,d
0077	390007		DECO	n3,d
007A	390005		DECO	n2,d
007D	00		STOP	
007E	390005	L8:	DECO	n2,d
0081	390003	L9:	DECO	n1,d
0084	00		STOP	
0085			.END	

# A mystery program (*Cont'd*)



(a) Structured flow.



(b) Spaghetti code.

# Structured vs. Unstructured Flow

- Assembly Language has unstructured flow
  - Possible to jump to any point in memory
  - Can jump into/out of the middle of loops
- Old HOL (pre-1970's) has GOTO command
- Modern HOL has structured flow of control
  - Every structure has a start and end
  - Can only jump to a start or end point

# Structured Programming Theorem

**Any** algorithm containing goto's can be written with **only** nested if statements and while loops

- Proven mathematically by Corrado Bohm and Giuseppe Jacopini in 1960

# The Goto controversy

- Two years after the Bohm/Jacopini paper, Edsger W. Dijkstra of the Technological University at Eindhoven, the Netherlands, wrote a letter to the editor of the same journal.
- He stated his personal observation that good programmers used fewer goto's than poor programmers.
- Proposed that goto's be abolished from all “higher level” programming languages.

# The Goto controversy

- Large controversy followed.
- It is now recognized that Dijkstra was correct.
- Reason
  - Cost. Less expensive to develop, debug, and maintain code without goto's.
- Note that an absence of gotos will not guaranteed well structured code.

# 6.3 Procedure and Function Calls

## Learning Objectives

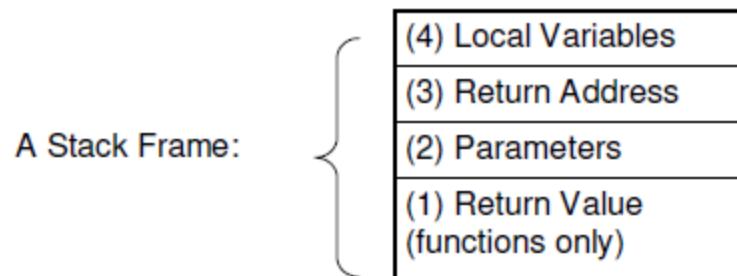
- **Describe assembly- & machine- level operations needed to CALL and RETURN from a function**
- **Show how to pass parameters to a function**
- **Show how to return a value from a function**

# Function Calls and Parameters

- C++ function call changes the flow of control
  - Must jump to the first instruction in the function
  - When function finishes, must return to the instruction that follows the function call

# Function Calls and Parameters

- Functions make a round-trip change of instruction flow:
  - Call
  - Return
- Functions use the stack for temporary data storage



# Subroutines

- Function calls in assembly language are called **subroutines**
- Two Assembly instructions to enable a subroutine:
  - **CALL**, jump to subroutine
  - **RETn**, return from subroutine
  - Both assume immediate addressing

# Subroutines

Call saves the return address, then jumps to the procedure.

Return takes us back to the return address.

## ■ CALL OperandSpec

1.  $SP \leftarrow SP - 2;$
2.  $Mem[SP] \leftarrow PC;$
3.  $PC \leftarrow Operand;$

/\*  
1. Push Ret Addr on the  
stack;  
2. Jump to Operand Addr.

Usually use Immediate  
Addressing Mode.

\*/

## ■ RETn

1.  $SP \leftarrow SP + n;$
2.  $PC \leftarrow Mem[SP];$
3.  $SP \leftarrow SP + 2;$

/\* Unary instruction  
(no operand);

1. Pop  $n$  bytes, where  
 $0 \leq n \leq 7$ . Intended for  
deallocating local variables;  
2. Pop return address

\*/

# Creating and Accessing a Stack Frame

- CALL & RETn do only part of the work needed for pushing and popping stack frame items

How do we do the rest?

*We'll try to answer this gradually.*

Stack Frame Contents	Allocate	Access	De-allocate
(4) Local variables	?	?	RETn
(3) Return Address	CALL	-	RETn
(2) Parameters	?	?	?
(1) Return value	?	?	?

# CALL

- Pushes the content of the PC onto the runtime stack
- Loads the operand into the PC

```
SP = SP - 2;  
// place return addr on RT stack  
Mem[SP] = PC;  
// jump  
PC = Oprnd;
```

# CALL

- CALL depends on the von Neumann execution cycle (fetch, increment, decode, execute, repeat).
- The three instructions on previous slide done in the execution part of the cycle.
  - Since increment PC before execute, the return address that is saved is the address of the instruction following the CALL.

# RETn

- Return from subroutine
- n is the number of local variables
- There are 8 versions of RETn:
  - RET0, RET1, RET2, ... RET7
  - Where n is the number of bytes occupied by the local variables in the subroutine
- Unary instruction

# RETn

- **Operation:**

- First, the instruction deallocates storage for the local variables by adding n to the SP
- Then, instruction moves the return address from the top of the stack into the PC
- Finally, instruction adds 2 to the SP to complete the pop operation.
- If there are more than 7 bytes of local variables, compiler will add ADDSP instruction to deallocate the rest of the locals.

# RETn

- Operation:

**SP = SP + n;**

**PC = Mem[SP]; // get return addr**

**SP = SP + 2; // pop stack**

# A procedure call at Level HOL6 and Level Asmb5 (Fig 6.18)

```
#include <iostream>
Using namespace std;

Void printTri( ) {
    cout << "*" << endl;
    cout << "**" << endl;
    cout << "***" << endl;
}
Int main ( ) {
    printTri( );
    printTri( );
    printTri( );
    return 0;
}
```

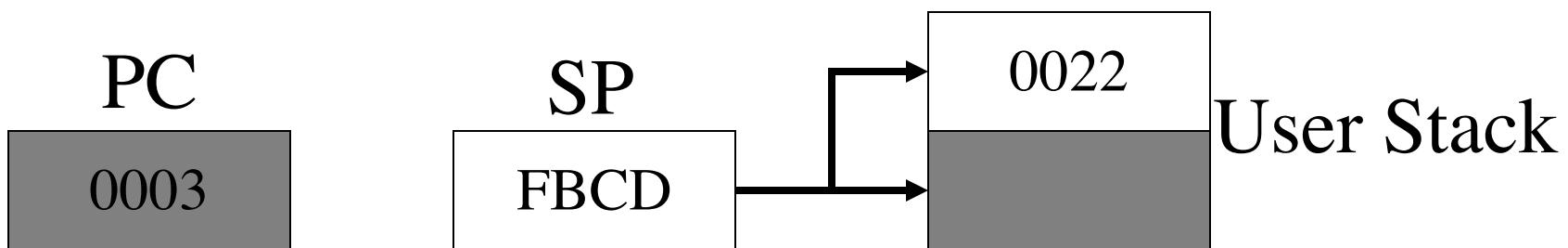
# A procedure call at Level HOL6 and

```
0000 04001F          BR      main
; ;***** address of the subroutine
0003 410016 printTri: STRO    msg1,d      ; cout << "*"
0006 50000A           CHARO   ' \n ',i     ;    << endl
0009 410018           STRO    msg2,d      ; cout << "**"
000C 50000A           CHARO
000F 41001B           STRO
0012 50000A           CHARO
0015 58              RETO
0016 2A00  msg1:     .ASCII   "*\x00"
0018 2A2A00 msg2:    .ASCII   "**\x00"
001B 2A2A2A msg3:    .ASCII   "***\x00"
00
;
;***** int main ()
001F 160003 main:    CALL     printTri    ;printTri ()
0022 160003           CALL     printTri    ;printTri ()
0025 160003           CALL     printTri    ;printTri ()
0028 00              STOP
. END
RET0 uses the address on the stack
0 because there are no local variables
```

# 6 and

```
0003 410016 printTri:STRO    msg1,d      ;cout << "*"
0006 50000A                 CHARO       '\n',i   ;  << endl
0009 410018                 STRO        msg2,d      ;cout << "**"
000C 50000A                 CHARO       '\n',i   ;  << endl
000F 41001B                 STRO        msg3,d      ;cout << "***"
0012 50000A                 CHARO       '\n',i   ;  << endl
0015 58                     RETO
0016 2A00      msg1:     .ASCII    "*\x00"
0018 2A2A00    msg2:     .ASCII    "**\x00"
001B 2A2A2A    msg3:     .ASCII    "***\x00"
001C 00
001D          ;
001E  ;***** int main ()
001F 160003  main:      CALL     printTri
0022 160003          CALL     printTri      ;printTri ()
0025 160003          CALL     printTri      ;printTri ()
0028 00
0029          .END
```

Pushes address 0022 onto stack  
puts address 0003 into PC



after execution of CALL

# A procedure call with a parameter at Level HOL6 and Level Asmb5

- **Convention for calling a subroutine:**

- Push storage for the return value(if exists)
- Push the actual parameters
- Push the return address
- Push storage for the local variables

# Subroutines

- An assembly program must contain explicit instructions to manipulate the run time stack.
- **Calling program**
  - pushes the actual parameters and
  - executes the CALL (pushes the return address)
- **The called program**
  - allocates stack space for local variables.

# Subroutines

- The **called program**
  - **deallocates** stack space of local variables and **pops** the return address by executing RETn.
- The **calling program**
  - **deallocates** storage of actual parameters.

# Summary

- **Calling** pushes actual parameters ( executes SUBSP)
- **Calling** pushes return address (executes CALL)
- **Called** allocates local variables (executes SUBSP)
- **Called** executes its body
- **Called** deallocates local variables and pops local variables (executes RETn, maybe ADDSP)
- **Calling** pops actual parameters (executes ADDSP)

# Summary

- Recall: The Return value is on the bottom of the stack frame
- Calling procedure allocates stack space
  - Shifts the stack point up (negative)
- Called procedure will store its result there
  - Both procedures will access it,
  - but may have different stack offsets

# **Chapter 12**

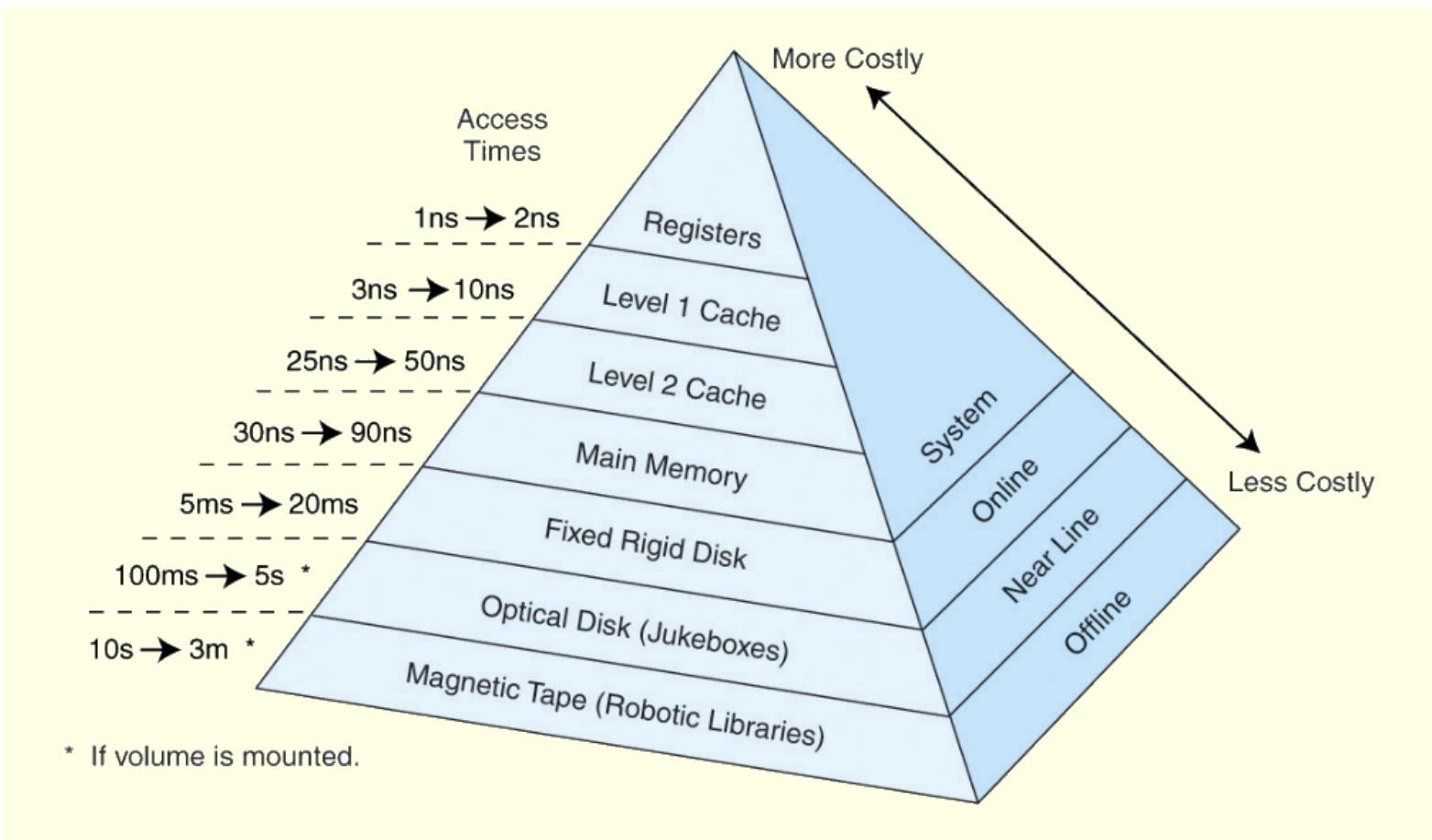
# **pages 634-642**

# **Cache Memory**

# Cache Memory

- Faster memory is more expensive than slower memory
- Memory is organized in a hierarchical fashion
  - Small, fast storage elements kept in the CPU
  - Larger, slower main memory accessed through the data bus
  - Larger, permanent storage - disk and tape drives still further from the CPU

- This storage organization can be thought of as a pyramid:

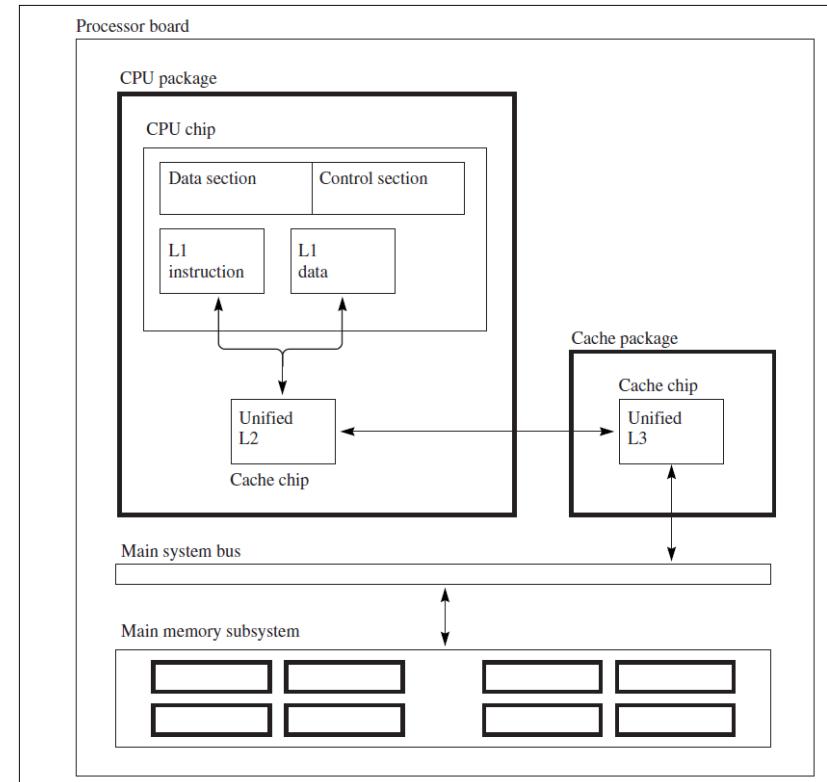


# Cache Memory

- To access a particular piece of data
  - CPU sends a request to its nearest memory, usually cache
  - If data not in cache-
    - then main memory is queried
  - If the data not in main memory-
    - then request goes to disk
- Once data is located
  - then data, and many nearby data elements are fetched into cache memory

# Cache Memory Levels

- Three levels of cache are common
  - Split L1 instruction and data cache on the CPU chip
  - Unified L2 cache in the CPU package
  - Unified L3 cache on the processor board



# Cache Memory Levels

- **Levels of cache form memory hierarchy**
- **Level 1 cache (8KB to 64KB)**
  - On the processor itself
  - Access time is about 4ns
- **Level 2 cache (64KB to 2MB)**
  - On the motherboard, or an expansion card
  - Access time is about 15 - 20ns

# Cache Memory Levels

- Systems that employ three levels of cache:
- Level 2 cache
  - on same die as CPU
  - reducing access time to about 10ns
- Level 3 cache (2MB to 256MB)
  - between processor and main memory

# Cache Memory

- **Purpose: to speed up accesses**
  - stores recently used data closer to the CPU
- **Cache**
  - Smaller than main memory
  - Access time is a fraction of that of main memory.
- **Accessed by content**
  - often called *content addressable memory*
- **Single large cache memory isn't always desirable-- it takes longer to search**

# Cache Memory

- ***Content addressed in content addressable cache memory***
  - subset of bits of a main memory address called a *field*
- **Fields**
  - provide a many-to-one mapping between larger main memory and the smaller cache memory
- **Many blocks of main memory map to a single block of cache**
- ***Tag field in cache block distinguishes one cached memory block from another***

# Cache Memory Designs

There are two types of cache designs covered in the chapter

- Direct-mapped cache
- Set-associative cache
  - Set-associative is derived from fully associative cache.

*There is also a third type of cache*

- Fully Associative
  - Found in exercises at the end of the chapter

# Direct-mapped Cache

**Physical address is divided into three fields**

- A high-order tag field
- A line field
- A low-order byte field

**Memory is divided into lines or blocks**

- If one byte is requested on a cache miss
  - Entire line containing the byte loaded into cache

# Direct-mapped Cache

- Direct mapped cache
  - **$N$  blocks of cache**
  - **Block  $X$  of main memory maps to cache block**  
$$Y = X \bmod N$$
- Example:
  - **10 blocks of cache**
  - **block 7 of cache may hold blocks**  
7 or 17, 27, 37, . . . of main memory
- ***valid bit is set for cache block***
  - lets the system know that the block contains valid data

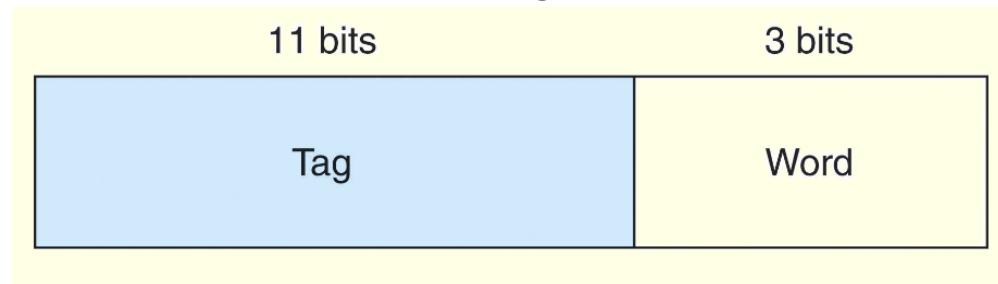
# Fully-associative cache

- Block can go anywhere in cache
- Cache will fill up before any blocks are evicted
- Memory address partitioned into two fields:
  - tag
  - word

# Fully-associative cache

- **Example:**

- 14-bit memory addresses
- cache with 16 blocks, each block of size 8
- field format of memory reference is:



- **When cache is searched**

- all tags searched in parallel to retrieve data quickly

- **Requires special, costly hardware**

- **Direct mapped cache**
  - evicts a block whenever another memory reference needs that block
- **Associative caches**
  - algorithm to determine which block to evict from cache
- ***victim block***
  - block that is evicted
- **Many ways to pick a victim**

# Set-associative cache

- **Set associative cache**
  - combines ideas of direct mapped cache and fully associative cache
- **N-way set associative cache mapping**
  - a memory reference maps to a particular location in cache (like direct mapped cache)
  - a memory reference maps to a set of several cache blocks (like fully associative cache works)
- **memory reference can map only to the subset of cache slots**

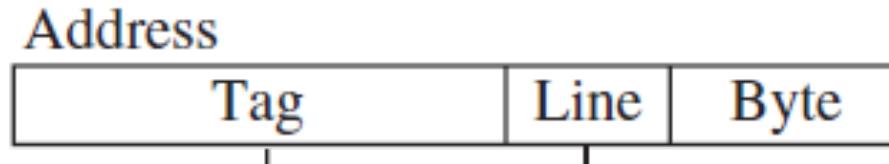
# Set-associative cache

- Number of cache blocks per set
  - varies according to system design
- Example: 2-way set associative cache
  - shown in schematic below
- Each set contains two different memory blocks

Set	Tag	Block 0 of set	Valid	Tag	Block 1 of set	Valid
0	00000000	Words A, B, C, ...	1	-----		0
1	11110101	Words L, M, N, ...	1	-----		0
2	-----		0	10111011	P, Q, R, ...	1
3	-----		0	11111100	T, U, V, ...	1

# Set-associative cache

- Set associative cache mapping:
  - memory reference divided into three fields: tag, set (line), and byte (or word),



- Byte field chooses byte within cache block
- Tag field uniquely identifies memory address
- Set (line) field determines set to which the memory block maps.

# Associative caches

- Fully associative and Set associative cache
  - *replacement policy* is invoked when needed to evict a block from cache
- *optimal replacement policy*
  - look into the future to see blocks not needed for the longest period of time
- Impossible to implement an optimal replacement algorithm
  - use as benchmark for assessing efficiency of any other scheme we come up with

# Associative caches

- Replacement policy chosen
  - depends upon locality to optimize
  - usually interested in temporal locality
- Least Recently Used (LRU) algorithm
  - Tracks last time block was accessed
  - Evicts the block that has been unused for the longest period of time
  - Disadvantage of this approach is complexity
    - LRU has to maintain access history for each block which ultimately slows down the cache

# Associative caches

- **First-in, first-out (FIFO)**
  - popular cache replacement policy
  - block in the cache the longest regardless of last used
- **Random replacement policy**
  - picks block at random and replaces with new block
  - Can certainly evict a block that will be needed often or needed soon

# Replacement Policies

- **Dirty blocks**
  - blocks updated while in cache
  - must be written back to memory
  - A *write policy* determines how
- **Two types of write policies**
  - write through
  - write back

# Replacement Policies

## ● Write through

- Updates cache and main memory simultaneously on every write
- Disadvantage:
  - memory must be updated with each cache write
  - slows down the access time on updates
  - usually negligible - majority of accesses usually reads not writes

## ● Write back or copyback

- Updates memory only when block is selected for replacement
- Advantage:
  - memory traffic is minimized
- Disadvantage:
  - memory does not always agree with the value in cache causes problems in systems with many concurrent users

# Alternative Caches

- **Unified or integrated cache**
  - Cache discussed previously
  - Instructions and data are cached together
- **Harvard cache**
  - Separate caches for data and instructions.
  - Provides better locality
  - Greater complexity
  - Larger cache
    - provides about the same performance improvement
    - without complexity

# Alternative Caches

- **Victim Cache**

- add small associative cache to hold blocks that have been evicted recently

- **Trace cache**

- variant of an instruction cache
  - holds decoded instructions for program branches
  - Gives illusion that noncontiguous instructions are really contiguous

# **Chapter 9 sections 9.1-9.3: Memory Management**

# Storage Management

- OS manages resources
  - CPU time
  - Main Memory (RAM)
  - Secondary storage (Disk, etc.)
- CPU time: Chapter 8
- Main Memory and Secondary Storage: Chapter 9

# Memory Management

- **Memory allocation**
  - Program resides on disk
  - Must be allocated RAM space by OS
  - Must be allocated CPU time by OS

# Memory Management

- Main Memory allocation **techniques:**
  - Uniprogramming
  - Fixed-partition multiprogramming
  - Variable-partition multiprogramming
  - Paging
  - Virtual memory
  - Segmentation
    - Covered in OS course

# Memory Management

## ● Uniprogramming

- System executes one job at a time
- Used in Pep/8
- OS resides at one end of memory,  
application at other
- Every job loaded at same place
  - Example: Pep/8 loads every app at **0000**
  - If program contains a burn directive,  
assembler assumes that the last byte will be  
loaded at the address at bottom of memory

# Memory Management

## ● **Uniprogramming**

### – Advantages:

- System is small and simple to design
- Easier to make system bug-free
- Little overhead
- App gets 100% of processor's time

### – Disadvantages

- Inefficient use of CPU time
- Inflexible job scheduling
  - Example: everyone must wait while a program does disk I/O
- Cannot multi-program two jobs

# Memory Management

- **Fixed-partition multiprogramming**
  - OS partitions MM into different regions
    - Storing different processes while they execute
  - Fixed-partition
    - OS subdivides memory into several regions
    - Regions sizes and locations do not change
    - OS occupies region at bottom of memory
    - Partitions are different sizes for different jobs

# Fixed-Partition

- **Problem:**

- **Memory references in object code must be adjusted for the region that they are loaded into.**
- **Example: program loaded into a Region**
  - OS loads it into a region starting at 8000(hex)
  - Object code was assembled starting at address 0000
  - All memory references are wrong

# Fixed Partition - Address Problem

- Solution:

- Logical Addresses
- Programmer/compiler generates code that starts at address 0000
- Called a Logical Address
- OS will translate the logical address to a physical address

# Fixed Partition

- **Problem: internal fragmentation**
  - Example: 10 KB program loaded into a 32KB Region
    - Need 10K, allocated 32K
    - 22K is Wasted

# Variable-Partition Multiprogramming

- **Variable-Partition Multiprogramming**
  - Solution to internal fragmentation
  - OS maintains partitions with variable boundaries
  - Establish partition when job is **loaded** into memory
  - Size of partition **exactly** matches size of job
  - **No** internal fragmentation

# Variable-Partition Multiprogramming

- **Holes**

- Job stops execution
  - That region of memory becomes available
- Called a **hole**
- OS allocates holes to subsequent jobs
- OS tries to maintain as many jobs in memory as possible
- Problem: **external fragmentation**

# Compaction

- **Compaction**

- **Solution to external fragmentation**
- **Shift processes up**
  - Strategy 1: Move all processes.
  - Strategy 2: Find process that can fill a hole.
- **Move only enough processes to create a hole big enough to satisfy the request**

# Paging

- **Goal:** avoid internal and external fragmentation
- **Method:**
  - Break up program to fit memory

# Paging

- Divide MM into **frames** of fixed size.
- Divide every program into **pages** of fixed size.
- Frame size **MUST** equal page size
- Allocate enough frames to program so that all of its pages will have a frame
- Program still assumes logical addressing
- OS converts logical addresses to physical addresses during execution

# Paging

- **Hardware**
  - partition system needs one base register
  - Paging needs a set of frame numbers, one for each page
  - Called a **page table**

# Paging Problems

- Internal fragmentation
  - Last page may not need entire frame
- Page size
  - **smaller** the page size = **less** fragmentation
  - **smaller** the page size =
    - greater the number of frames
    - **larger** the page table
  - **Page tables use fast memory = expensive**  
**Want small page tables**

# Virtual Memory

## ● Large Program Behavior

- Most programs have dozens of procedures
- Many never called
- Some initialization routines only called once
- Many loops
- Some declarations never used
  - e.g., an array is declared to be size 1000, but only 1st 10 elements used

# Virtual Memory

- Large Program Behavior

- **Result:** only part of a program needs to be in memory at a time
- **Working set:** set of active pages
- **As program progresses**
  - new pages enter the working set
  - old pages leave the working set

# Virtual Memory

- **Programmer illusion:**
  - program executes in contiguous memory
  - logical addresses begin at zero
- **Reality:**
  - System loads only a few pages at a time
  - Pages are dispersed throughout memory

# Virtual Memory

- **Demand Paging**
  - System loads a page into MM only when program demands it
- **Difference from paging**
  - Paging loads all pages into MM

# Page Replacement

- **Problem:** OS needs to read a page into memory - no frames available
- **Page Fault**
- **OS selects a page that is in memory to replace**
- **Page taken out of memory**
  - Has changed - write page to disk
  - No changes – do not write to disk

# Page Replacement

## ● Dirty Bit

- Frame table has a **dirty bit**
- When page is loaded into memory
  - dirty bit set to 0
- If store instruction is executed
  - dirty bit is changed to 1
- When page is selected for replacement
  - OS looks at dirty bit
- If Dirty bit = 1, write to disk
- If Dirty bit = 0, do not write

# Page Replacement

- **Page-Replacement Algorithms**
  - OS has two memory management tasks in a demand paged system:
    - Allocate frames to processes
    - Select a page for replacement when a page fault occurs and all frames are full

# Page Replacement

- **Frame Allocation Strategies:**
  - Assume that a large process needs more frames than a small process
    - Allocate frames proportionally
  - **Allocate frames dynamically**
    - Keep track of the working set
    - Larger working set = more frames allocated

# Page Replacement

- **Page Replacement Strategies**
  - Process has fixed number of frames
  - How does OS decide which page to replace?
    - FIFO (first in, first out)
    - LRU (least recently used)

# Page Replacement

- **FIFO: first-in, first-out**
  - Replace the oldest frame
  - New code & data added
  - Older code & data replace

# LRU

- Least Recently Used

- Replace page not **referenced** the longest
- Idea: page referenced recently in the past more likely to be referenced in the near future
- Order in page table changes as each page is referenced
- When a page is referenced, it is placed on top
- Page fault causes the bottom page to be removed
- LRU generally produces fewer page faults than FIFO

# Page replacement

- Each OS has own unique page-replacement algorithm
  - Depends on hardware features available
  - Most are approximations of the LRU
- Effective demand paging system
  - Page fault rate needs to be:

**less than one fault per 100,000 memory references**

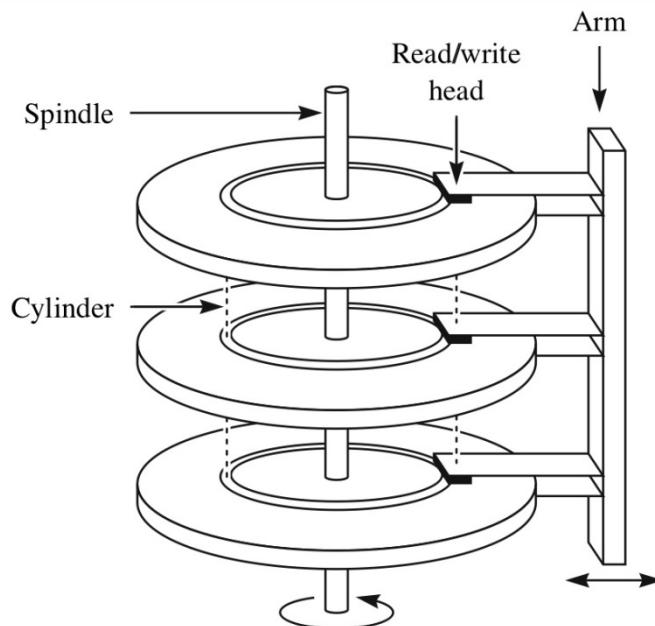
# Summary of the 4 main memory models

	>1 program in memory?	A program is in contiguous area?	All of a program must be in memory?
Uniprogramming	No	Yes	Yes
Partitioned main memory	Yes	Yes	Yes
Paging	Yes	No	Yes
Virtual memory	Yes	No	No

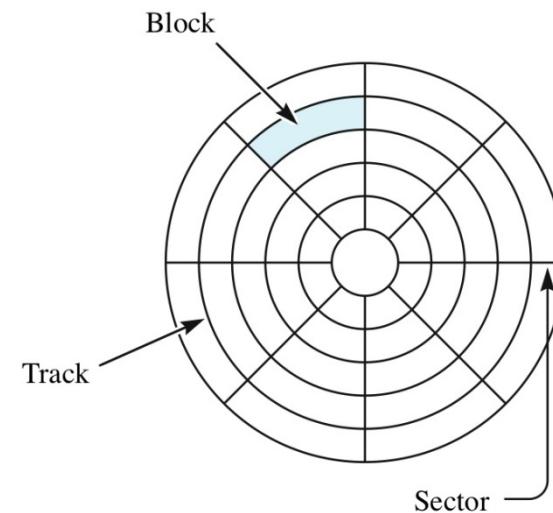
# Storage of data on discs

- Basic model of a disc

- Set of parallel rotating platters
- Read/write heads move as a unit into one of a finite number of positions



(a) A hard disk drive.



(b) A single disk.

# Disc addressing

- Projection of the read/write head on recording surface defines a track
- Track is divided into sectors
  - smallest addressable unit of the disc
- Block address is a three-part object:
  - Surface-number, track-number, sector-number

# Disc addressing

- Set of tracks (one per surface) having the same radius is called a *cylinder*
- Capacity of a disc is the product:
  - Recording\_surfaces \* tracks-per-surface \* sectors-per-track \* bytes-per-sector
- Some bytes
  - used for formatting information (e.g., track numbers)
  - not available for users

# Disc Access

- **Read/write a sector:**

- move the read/write heads to the appropriate track
  - wait for sector to rotate around
    - latency time is, on average, half a rotation
    - Faster spinning discs, e.g. 15,000 rpm vs 3600 rpm reduces latency
  - do read/write

- **Operating System**

- allocate space in clusters of sectors (e.g. an allocation unit might be 4 sectors)
  - reduce overheads
  - allocate data cylinder by cylinder rather than surface by surface

# Contributions to the disk access time

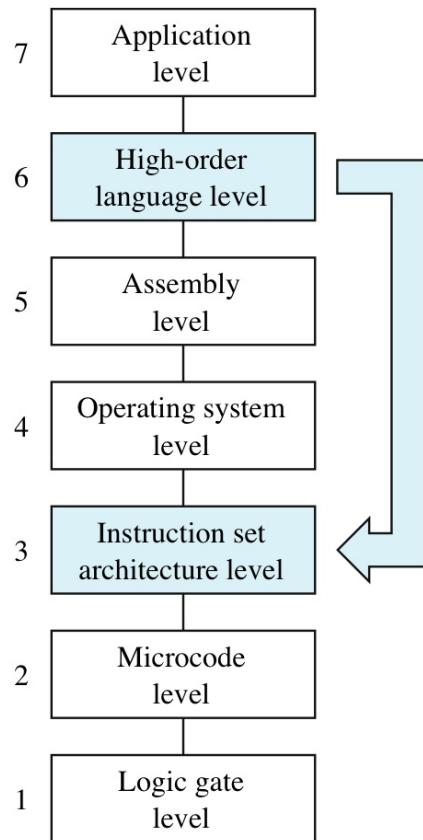
- **Seek time**
  - Time for the head to reach the cylinder
- **Latency**
  - Time for start of sector to rotate to head
- **Transmission time**
  - Time for sector to pass under head



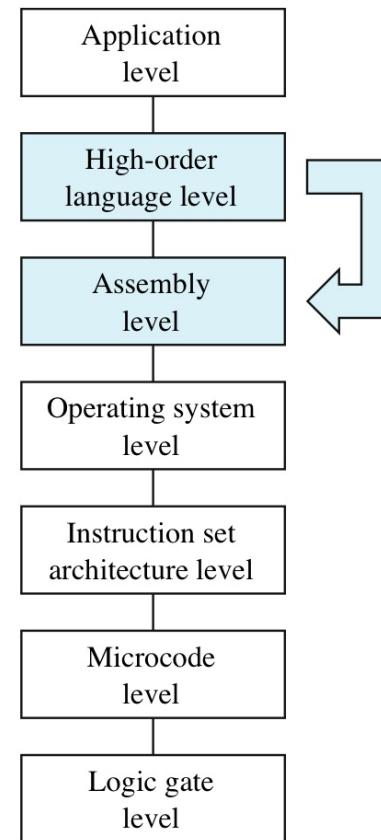
# Compilers

- Translates a high-level program into a lower level program.
  - Some into machine language
  - Others into assembly language or some intermediate code.
    - Then need an assembler.
  - Input: source program
  - Output: object program (regardless of what language)

# The function of a compiler



(a) Translation directly to machine language.



(b) Translation to assembly language.

# Compilers

Compiling is more complex than Assembling

- One HOL instruction → many lower level instructions
- Some translations are direct
- Some translations depend on context
- Some lines don't translate at all (directives)

# Compilers

- Example: next slide
  - 1 C++ statement to 4 assembly statements
  - Compilers do 1-to-many mappings
  - Include is not translated; pre-compiler would include the code or link to appropriate library code
  - Other translations of cout are possible.
  - Different compilers generate different machine programs.
    - All are correct
    - Some are more efficient.

# Compilers

- **Example: next slide**
  - **Return 0 is translated as "stop".**
    - Real compiler would translate as return to OS
    - Return value will be stored in a register and interpreted by the OS
    - Traditionally return value of '0' means no errors
  - **In functions other than main, would return from function instead.**

# The cout statement at Level HOL6 and Level Asmb5

## Example: 1-to-Many Translation

### High-Order Language

```
#include <iostream>
using namespace std;
int main () {
    cout << "Love" << endl;
    return 0;
}
```

### Assembly Language

0000	410007	STRO	msg,d
0003	50000A	CHARO	'\n',i
0006	00	STOP	
0007	4C6F76 msg:	.ASCII	"Love\x00"
	6500		
000C		.END	

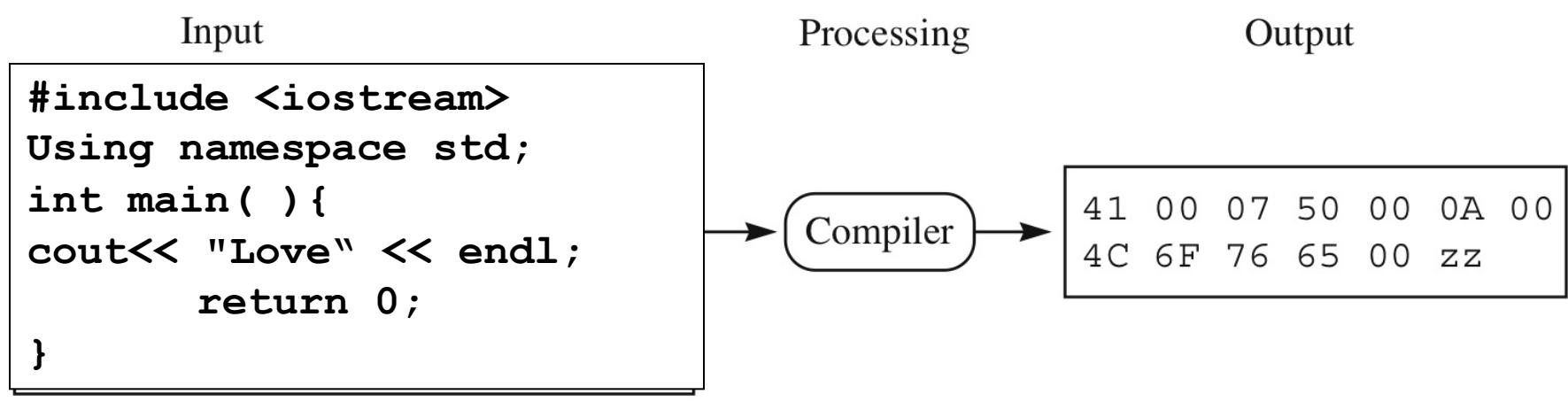
### Output

Love

Cout is an *abstraction* of 3 (or more) Asmb5 instructions

- String → STRO
- Char → CHARO
- Integer → DECO

# A compiler that translates directly into machine language



(a) A compiler that translates directly into machine language.

# A compiler that translates into assembly language

```
#include <iostream>
Using namespace std;
int main( ){
cout << "love" << endl;
    return 0;
}
```



```
STRO msg,d
CHARO '\n',i
STOP
msg: .ASCII "Love\x00"
.END
```

(b) A compiler that translates into assembly language.

# Variables and Types

- Every C++ variable has 3 attributes:
  - Name
  - Type
  - Value
- Variables correspond to memory locations
  - Level 6: refer to variables by names
  - Level 3: refer to variables by addresses

# Variables and Types

HOL6 Variable → ISA3 Variable

Name / Identifier → Memory Address\*

Value → Memory Contents

Type → No data type, only instruction type

- \* A global variable's memory address is fixed; a local variable's address isn't.

We need a data table to keep track of these mappings...

# Variables and Types

- Compiler translates variables to addresses
  - Uses symbol table
  - Similar to symbol table of assembler, but more complicated
  - Must also contain type
  - Temporary table to track which identifier = which memory address/value
  - Constructed and used by both Compilers and Assemblers
  - Not part of the completed object program itself
    - Traditionally, a symbol table would be discarded after translation
    - (*But, modern programs with dynamic linking and loading need to retain the symbol table until run time*)

# Symbol Table Contents

- 3 Fields:

<u>Symbol</u>	<u>Value</u>	<u>Data Type</u>
myChar	0003	sChar
myInt	0004	sInt

- Symbol values are usually memory *addresses*, not memory *contents*
  - Exception: constants
- Both instruction and data symbols

# Variables Vs. Constants

- Comparing High Level Language Declarations:
  - Variable:      `int myVar = 5;`
  - Constant:      `const int myConst = 5; // C++`  
                      `final int myConst = 5; // Java`
- Comparing Directives:
  - `.BYTE/.WORD` allocate program space with an initial value.
  - `.EQUATE` creates a symbol table entry. It is used for translating operands. It doesn't increase program size.
- If a value is constant, let the compiler know!

# Compiling Global Variable and Constants

## HOL6 code

```
// declare  
int score;  
const int bonus=5;
```

```
// assign a variable  
score += bonus;
```

## Asmb5 code

```
; Declarations translate to directives  
score: .BLOCK 2  
bonus: .EQUATE 5
```

;(Global) Variable uses *direct mode*

; Const uses *immediate mode*

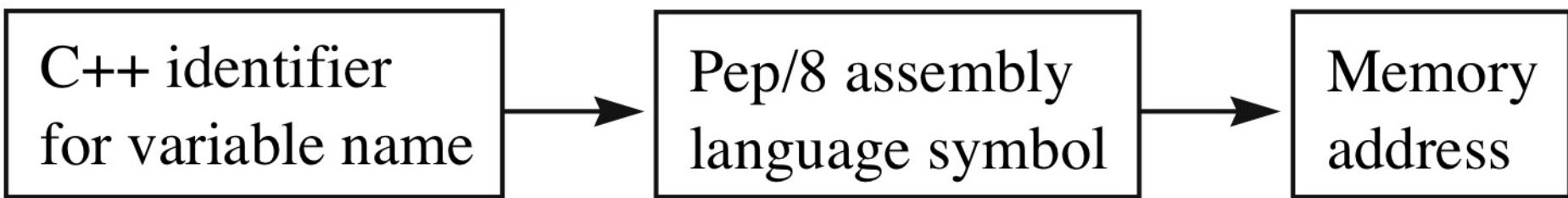
```
LDA score, d  
ADDA bonus, i  
STA score, d
```

# Compilers Vs Interpreters

- Interpreter - translates and executes as it goes
  - Can only see one or a few lines at a time
  - Runs slowly
- Compiler - translates a file to produce an executable file, for later execution
  - Can make multiple *passes* of the input file, to achieve a better result

# A hypothetical compiler for illustrative purposes

We will show translation from C/C++ to Assembly for illustrative purposes.



(b) A hypothetical compiler for illustrative purposes.

Variable names will correspond to Pep/8 Symbols. **Really** would be **addresses**.

# Assignment Statements

- **Global variables:**

- Allocated at a fixed location with a .block
  - Accessed with direct addressing ( $d$ )

- **Assignment statements**

- Load accumulator from the right hand side of the assignment with a LDA
  - Compute the RHS of the assignment if necessary.
  - Store the value to the variable named in the LHS with a STA

# Assignment Statements

- Example:

i +=5;

Becomes

LDA i,d

ADDA 5,i

STA i,d

# Assignment Statements

- Example:

ch++;

Becomes

LDBYTEA ch,d

ADDA 1,i

STBYTEA ch,d

# The assignment statement at Level HOL6 and Level Asmb5

```
#include <iostream>
using namespace std;

char ch;
int i;
main(){
    cin >> ch >>i;
    i+=5;
    ch++;
    cout << ch << endl << i << endl;
    return 0;
}
```



# The assignment statement at Level HOL6 and Level Asmb5

## Assembly Language

```
0000 040006          BR      main
0003 00      ch:     .BLOCK  1           ;global variable
0004 0000    i:     .BLOCK  2           ;global variable
               ;
0006 490003 main:   CHARI   ch,d       ;cin >> ch
0009 310004          DECI    i,d        ;    >> i
000C C10004          LDA     = 5
000F 700005          ADDA
               Why load a byte?
0012 E10004          STA     i,d
0015 D10003          LDBYTEA ch,d       ;ch++
0018 700001          ADDA   1,i
001B F10003          STBYTEA ch,d
001E 510003          CHARO   ch,d       ;cout << ch
0021 50000A          CHARO   '\n',i     ;    << endl
0024 390004          DECO    i,d        ;    << i
0027 50000A          CHARO   '\n',i     ;    << endl
002A 00              STOP
002B               .END
```

Why one byte?

Why load a byte?

# The assignment statement at Level HOL6 and Level Asmb5

Input

M 419

Output

N

424

# Compilers

- Compilers are just programs
  - Can be written in any language
  - Need symbol tables. See next slide.
  - An entry contains 3 parts: the symbol, its value (address) and the type.

# Type Compatibility

- Example: two variables in a C++ program
  - Note that Pep/8 cannot represent float variables
- int j;  
float y;
- Symbol table is on next slide.

# The symbol table for a Pep/99 compiler

	symbol	value	kind
[0]	j	0003	sInt
[1]	y	0005	sFloat
[2]	:	:	:

# Symbol tables and compilers

- During code generation phase, the compiler translates:  
`cin << ch << i;`
- Into  
`CHARI 0x003, d`  
`DECI 0x004, d`
- Based on the values in the symbol table
- Listings in these slides will show symbolic names to enhance understandability.

# Type Compatibility

- Consider a C++ program that declares two variables: `int j; float y;`
- Will get symbol table entries:

	symbol	value	kind
[0]	j	0003	sInt
[1]	y	0005	sFloat
[2]	:	:	:

# Type Compatibility

- Consider

- $j \% 8$

- In binary set all bits **except** rightmost 3 to 0

- Example:

- If  $j$  has value 59 (dec) = 0011 1011 (bin)
  - Then  $j \% 8$  is 3 (dec) = 0000 0011 (bin)
  - Same as value 59 with all **but** last 3 bits set to 0

# Type Compatibility

- Consider  
 $j \% 8$
- When compiler sees this expression it consults the symbol table and determines the type of  $j$  is **sInt**.
- Also recognizes 8 as integer constant
- So % operation **is legal**.

# Type Compatibility

- Compiler generates the code

**LOADA j,d**

**ANDA 0x0003,i**

**STOREA j,d**

# Type Compatibility

- Now consider:  $y = y \% 8$
- Compiler consults the symbol table and determines that the type of y is sFloat.
- Then determines that % is not legal on a sFloat type.
- Generates an error (real compiler would generate code to change y to integer).

# Type Compatibility

- Now consider:  $y = y \% 8$
- If did **not** check type, compiler would generate code:

LOADA y,d

ANDA 0x0003,i

STOREA y,d

- Which does **not** work because floats are stored in special format.

# Type Compatibility

- Note that the compiler catches type errors at compile time, **not run time**.
- Assembly/machine code does **not** check type.
- So there can be no run time check of type.

# Shift and rotate

- Pep/8 has
  - two arithmetic shift instructions
  - Two logical shift instructions

0001 110r	ASLr	Arithmetic shift left r	NZVC
0001 111r	ASRr	Arithmetic shift right r	NZC
0010 000r	ROLr	Rotate left r	C
0010 001r	RORr	Rotate right r	C

# Shift and Rotate

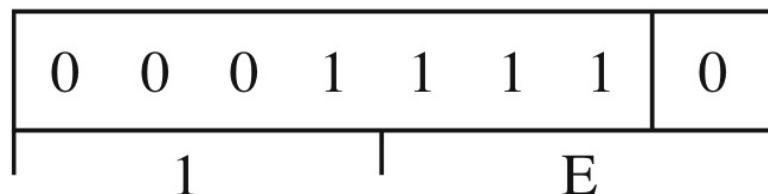
- No operand specifier in these instructions
  - Only work on specified register
- Effect:
  - Shift left multiplies by 2
  - Shift right divides (integer) by 2
  - Rotate left rotates each bit to the **left** one bit, sending the most significant bit into C and C into the least significant bit.
  - Rotate right rotates each bit to the **right** one bit, sending the most least bit into C and C into the most significant bit.

# Shift and Rotate

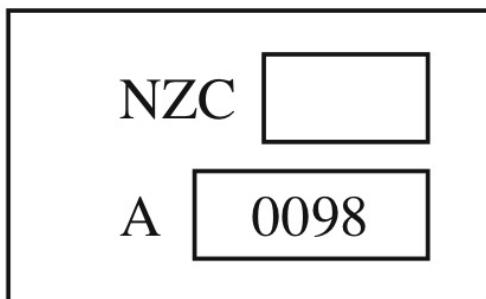
Instruction specifier

Opcode

r



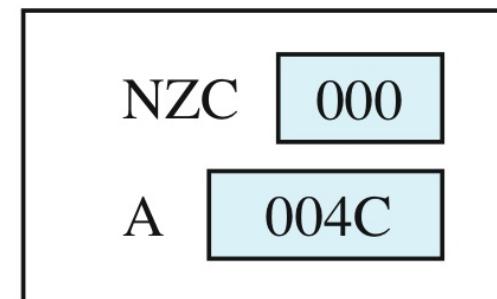
CPU



(a) Before

1E  
Arithmetic shift right  
accumulator

CPU



(b) After

# Shift and Rotate

## Arithmetic Shift Instructions

- ASLr/ASRr - shift a register's contents 1 bit left/right
  - Mathematically equivalent to multiply by 2/divide by 2
  - Example: 8-bit unsigned number: 01001101 (= 77)
  - Shift left, fill far right bit with a 0: 10011010 (= 154)
  - Or, shift right: 00100110 (= 38)
  - The bit on the left/right end gets shifted into the C status bit

Original: C r[0] r[1] r[2] r[3] r[4] r[5] r[6] r[7]

ASLr: r[0] r[1] r[2] r[3] r[4] r[5] r[6] r[7] 0

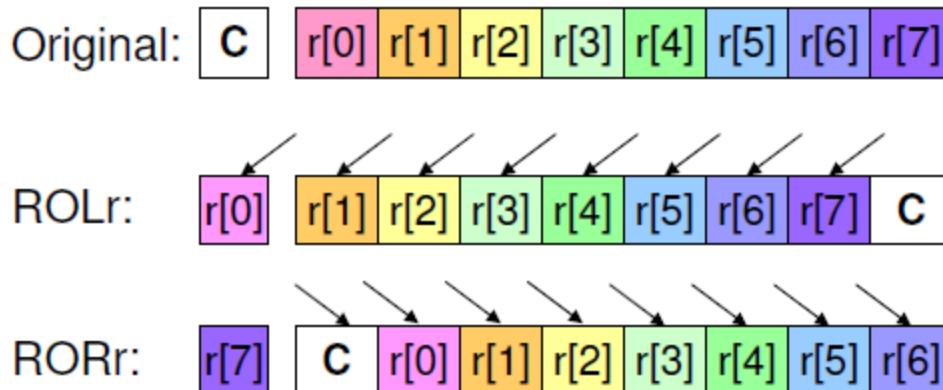
ASRr: r[7] r[0] r[0] r[1] r[2] r[3] r[4] r[5] r[6]

Due to the arithmetic interpretation, these operations affect the N,Z,C, and V status bits.

# Shift and Rotate

## Rotate Instructions

- ROL/RORr - shift a register's contents 1 bit left/right, but slightly differently!
  - Carry bit+register are treated like a ring.
  - Bits shift around the ring.



Due to the non-arithmetic interpretation, these operations do NOT affect N,Z and V status bits.

# Constants

- **Create constants with the .EQUATE command.**
  - Creates an entry in the symbol table
  - Does **not** create bits in the machine code
  - **Operation:**
    - It must be on a line that defines a symbol
    - It equates the value of the symbol to the value that follows the .EQUATE
    - It does not generate any object code

# Constants

- Example:

`linefeed:.EQUATE 0x000A`

- Everywhere in program that the symbol *linefeed* occurs, assembler replaces with 0x000A

- Similar to C++ code:

`const int lineFeed = 10;`

Or the preprocessor command:

`#define lineFeed 10`

# **Chapter 8: Process Management Sections 8.1 and 8.2**

# Operating Systems

## Level 4

- OS defines an abstract machine
- Provides environment for programming
- Allocates resources efficiently

# Operating Systems

- **Resources of a typical computer system:**
  - CPU time
  - Main Memory
  - Secondary Storage (disk, etc.)
- **Chapter 8: how OS allocates CPU time**
- **Chapter 9: how OS allocates Main Memory and Disk Memory**
- **Chapter 12.3: how the CPU cache memory works**

# Three types of operating systems

- **Single-user**

- Hand-held devices (Cell phones, Tablets, etc.)
  - PC
  - Pep/8

- **Multiuser**

- Servers
  - Databases

- **Real-time**

- Usually video or hardware control
  - Examples: microwaves, auto engines, nuclear power plants, etc.

# Pep/8

- Illustrates **some** of the techniques used in CPU allocation
  - Has a loader
  - Does **not** have a scheduler
- Does **not** illustrate memory management or disk management

# Loaders

- OS manages jobs submitted to be executed
  - Single user submitting several jobs
  - Several users submitting jobs
  - OS decides which job to run from a list of pending jobs
  - OS loads program into Main Memory and turns control of CPU to that program

# Loader

- **Pep/8 Loader**
  - 500 lines of assembly code
  - Usually OS is 90% C and 10% assembly
  - Assembly used for special effects (direct access to registers, etc.)

# Pep/8 Loader

- Real programs
  - Not in ASCII hex digits
  - Machine code in **binary**
  - Pep/8 uses ASCII characters for the object file
    - Makes machine language programming easier
- Commented Source Code for Pep/8 Loader
  - Shown in Fig 8.3

# Program Termination

- **Pep/8**

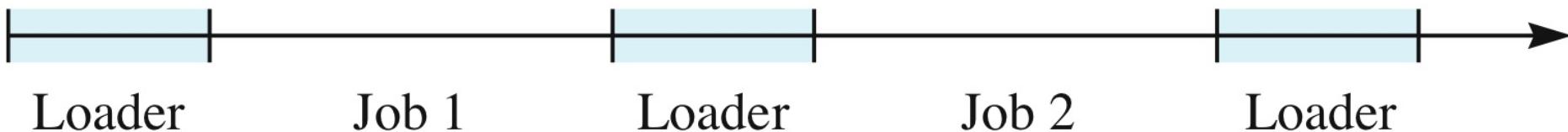
- Programs terminate with STOP instruction
  - Control returns to user of the Pep/8 simulator

- **Real OS**

- Computers don't have a STOP instruction
  - Have instruction that returns control to OS
  - On minicomputer:
    - OS would wait for another service request
  - On timesharing system:
    - OS would continue to process other jobs.

# Program Termination

- One processor: system **alternates** between user processes and OS processes.
- Time spent on OS processes is **overhead**
- OS manages the CPU using **Traps**
  - interrupt is a signal generated in hardware
  - when an interrupt occurs, control is given to the OS



# Traps

- Pep/8 has 5 instructions at level 5 (assembly) that are not at level 3 (machine code):
  - DECI
  - DECO
  - STRO
  - NOP (non-unary)
  - NOP0, NOP1, NOP2, NOP3 (all unary)
- No-op instructions do nothing;
  - They allow reprogramming by the OS programmer

# Traps

- When CPU fetches instructions with one of these opcodes, *hardware* generates a trap:
  - 00110            DECI            Tertiary decimal input
  - 00111            DECO            Tertiary decimal output
  - 01000            STRO            Tertiary string output
  - 001001nn        NOPn            Unary no-op
  - 00101            NOP            Tertiary no-op

# Traps

- Trap is similar to a subroutine jump (CALL)
- Trap handler: code that executes when a trap is called
- Trap returns control to the application by executing a return from trap (RETTR) call
- Traps also called:
  - Software interrupts
  - Synchronous interrupts

# Interrupt Mechanism

- When an interrupt is executed
  - state of running application must be stored including all registers
  - State includes all OS/hardware values that application needs to continue executing
- Where can we save the state?
- State must be saved on a stack
  - interrupt could be interrupted itself

# PCB

- **Process:** a program in execution
- **TRAP:** suspends a process so OS can perform a service
- **Process Control Block:**
  - Block of information in main memory that contains a copy of the interrupted process's registers
  - Example: the system stack is often the PCB

# RETTR

- After OS performs service, must return control of CPU to the suspended process
- Use the RETTR instruction
- RETTR instruction restores registers
  - Restoration done in hardware

# PEP/8 RETTR instruction

- RETTR pops off top 9 bytes from stack into appropriate registers
- IR is not popped

$\text{NZVC} \leftarrow \text{Mem}[\text{SP}] \langle 4..7 \rangle ;$   
 $\text{A} \leftarrow \text{Mem}[\text{SP} + 1] ;$   
 $\text{X} \leftarrow \text{Mem}[\text{SP} + 3] ;$   
 $\text{PC} \leftarrow \text{Mem}[\text{SP} + 5] ;$   
 $\text{SP} \leftarrow \text{Mem}[\text{SP} + 7]$

# RETTR

- Note: **IR** is not popped
- Next instruction executed indicated by restored PC
- Last register to change is the **SP**
  - Now points to the top of the user stack
- If Trap handler made changes to the **PCB**
  - Changes will be carried over to process when restored
- Process continues like before interrupt

# Loader & Trap

- Assembly code for Loader & all Traps
  - Shown in the text



# Problems with Assembler so far...

- Programmer must know where the data is in memory
- Data goes at the bottom of memory, must write program before know where data goes
- if you insert an instruction later all data addresses change
- Single digit decimal input/output is tedious
  - CHARI/CHARO handle a single *character, not numbers*

# Solution: New Instructions

## BR, DECI, DECO

- New instruction: unconditional branch
  - Instruction specifier: 0000 010a
  - Mnemonic: BR
- Jumps to a different memory location for the next instruction to be executed
- PC<- {Oprnd}
  - BR 0005 ; branch around data
  - Places the operand (0005) into PC.
  - Result: next instruction executed is at memory location 0005

# Unconditional Branch

- **New instruction:** unconditional branch
  - Instruction specifier: 0000 010a
  - Mnemonic: BR
- **Default:** immediate addressing.
  - If you do not specify addressing mode, assembler assumes immediate addressing.
  - Assembler places 000 in the addressing mode bits.

# Branch

- Correct operation of BR depends on details of von Neumann execution cycle  
**fetch increment execute repeat**
- Not  
**fetch execute increment repeat**

# Branching

## Why Branch?

- Flow of Control
  - Selection Flow (if-else) and Repetition Flow (loops) require jumping from one part of the program to another
- Modular Memory Use
  - Advantages to splitting up large sections of instructions and/or data into smaller units
    - Locality - keep data near the instructions that use them
    - Isolation - changes in one section won't affect other sections
  - Need to jump from one section to another

# **DECI and DECO**

- Do not exist at machine level
- Provided by the OS
- DECI: decimal input.
  - Converts sequence of ASCII digit characters
  - To a single word
  - In 2's complement representation
- DECO: decimal output.
  - Converts 2's complement representation
  - Of a word
  - To sequence of ASCII characters

# DECI and DECO

- **DECI: The decimal input instruction**
  - Instruction specifier: 0011 0aaa
  - Mnemonic: DECI
- **Convert a string of ASCII characters**
  - from the input device
  - into a 16-bit signed integer (corresponds to the 2's complement representation of the value)
  - stores it into memory
- Oprnd <- *{decimal input}*

# DECI and DECO

- Notes on DECI: decimal input

- Any number of leading spaces or line feeds
- First printable char: digit or + or –
- Following char must be digits
- Sets Z to 1 if input 0 and N to 1 if input negative
- Sets V to 1 if input a number out of range (-32768 to 32767)
- Does not affect C bit.

# DECI and DECO

- DECO: decimal output.
  - Instruction specifier: 0011 1aaa
  - Mnemonic: DECO
- Convert a 16-bit signed integer
  - from memory
  - into a string of ASCII characters
  - sends the string to the output device
- *{decimal output} <- Oprnd*

# DECI and DECO

- Notes on DECO:

- Prints a – for negative but does not print +
- Does not print leading 0's
- Outputs minimum number of digits possible to represent the value
- You cannot specify field width
- Does not affect NZVC bits



# **DECI and DECO**

- **Example**

**DECI 0003,d ; get a number**

**With input**

**-479**

**Converts the number to**

**1111 1110 0010 0001 (bin)**

**And stores it in**

**Mem[0003]**

# A program to add 1 to a decimal value. It illustrates Decimal I/O and Branch

```
0000 040005 BR      0x0005      ;Branch around data
0003 0000 .BLOCK    2           ;Storage for one integer
                                ;
0005 310003 DECI    0x0003,d   ;Get the number
0008 390003 DECO    0x0003,d   ;and output it
000B 500020 CHARO   ',' ,i     ;Output " + 1 = "
000E 50002B CHARO   '+',i
0011 500020 CHARO   ',' ,i
0014 500031 CHARO   '1',i
0017 500020 CHARO   ',' ,i
001A 50003D CHARO   '=' ,i
001D 500020 CHARO   ',' ,i
0020 C10003 LDA     0x0003,d   ;A := the number
0023 700001 ADDA    1,i        ;Add one to it
0026 E10003 STA     0x0003,d   ;Store the sum
0029 390003 DECO    0x0003,d   ;Output the sum
002C 00 STOP
002D .END
```

## Input

-479

## Output

-479 + 1 = -478



# Special characters and strings

- `\x` is the char escape prefix, meaning:
  - “this char is being represented by its numeric code.”
  - `\x5A` is the same as ‘Z’
  - `\n` is New Line. `\t` is Tab.
- In Pep/8 (as in C), a string is an array of chars.
  - To mark the array end, the last byte must be `\x00` (NULL)
  - In C,
    - If you use string commands, the compiler will automatically take care of the `\x00` sentinel byte.
    - If you use array commands, you must manually read and write the sentinel byte.
  - In Pep/8, you must always manually handle the sentinel byte.

# Special characters and strings

- Most strings don't fit in a Pep/8 16-bit operand specifier, so strings have to be referenced by memory location.
- Step 1: Use a symbol label and the .ASCII directive to declare a string:
  - myString1: .ASCII "Print this out.\x00"
  - This stores the string as bytes of char data in memory.  
Be sure to place this where the program will not interpret it as instructions.
- Step 2: Reference the string via its symbol label:
  - STRO myString1, d
  - The STRO directive repeats CHAR0 until the \x00 sentinel

# STRO

- **STRO (string output)**
  - Instruction specifier: 0100 0aaa
  - Mnemonic: STRO
  - Does not exist at machine level
  - Triggers a *trap* to the OS when executed
  - Allows you to output an entire string
    - Send a string of null-terminated ASCII characters to the output device
- ***{string output} <- Oprnd***

# STRO

## ● Notes on STRO

- **Operand is a contiguous sequence of bytes**
  - Each is interpreted as an ASCII character
  - Last byte must be a **byte** of all **0's**
- **Outputs all bytes up to (but not including) the sentinel 00**

# Fig. 5.12: A program that interprets a value as decimal and as hexadecimal

```
0000 04000D BR      0x000D          ;Branch around data
0003 0000 .BLOCK 2                  ;Storage for one integer
0005 202B20 .ASCII " + 1 = \x00"
                                    31203D
                                    2000
                                    ;
000D 310003 DECI    0x0003,d       ;Get the number
0010 390003 DECO    0x0003,d       ;and output it
0013 410005 STRO    0x0005,d       ;Output " + 1 =
0016 C10003 LDA     0x0003,d       ;A := the number
0019 700001 ADDA    1,i            ;Add one to it
001C E10003 STA     0x0003,d       ;Store the sum
001F 390003 DECO    0x0003,d       ;Output the sum
0022 00      STOP
0023      .END
```

## Input

-479

## Output

-479 + 1 = -478

# Interpreting Bit Patterns

- Remember –
  - CPU has no knowledge of what bits mean
- Next slide demonstrates this

# Interpreting Bit Patterns

```
0000 040009 BR    0x0009  
0003 FFFE .WORD 0xFFFFE  
0005 00 .BYTE 0x00  
0006 55 .BYTE 'U'  
0007 0470 .WORD 1136  
;  
0009 390003 DECO 0x0003,d  
000C 50000A CHARO \n,i  
000F 390005 DECO 0x0005,d  
0012 50000A CHARO \n,i  
0015 510006 CHARO 0x0006,d  
0018 510008 CHARO 0x0008,d  
001B 00 STOP  
001C .END
```

;Branch around data  
;First  
;Second  
;Third  
;Fourth

Generated as Hex  
Output as decimal

;Interpret First as decimal.  
;Interpret Second as decimal.  
;Interpret Third as character  
;Interpret Fourth as character

Interprets the number at address 0008 as a character. The number placed in address 0007 and 0008 is 1136 or 0470(hex). 70 is ASCII for 'p'.

## Output

-2

85

Up

# Disassemblers

- **Assembler**
  - translates each assembly language statement into exactly one machine language statement
  - One-to-one mapping
- **Given a single assembly language statement, can always determine the corresponding machine language statement**

# Disassemblers

- Inverse not true
- Example: 0101 0111
  - Is this a .ASCII with character W?
  - Or the CHARO mnemonic (stack-indexed deferred addressing)?
  - Same bits!
- CPU knows nothing about the assembly statements
- Next slide: two programs that produce same machine language and output.

# Disassemblers

## Assembly Language Program

```
0000  51000A CHARO    0x000A,d  
0003  51000B CHARO    0x000B,d  
0006  51000C CHARO    0x000C,d  
0009  00              STOP  
000A  50756E .ASCII    "Pun"  
000D  END
```

## Assembly Language Program

```
0000  51000A CHARO    0x000A,d  
0003  51000B CHARO    0x000B,d  
0006  51000C CHARO    0x000C,d  
0009  00              STOP  
000A  50756E CHARO    0x756E,i  
000D  .END
```

## Program Output

Pun

Two assembly language programs that produce the same object code and, therefore, the same output

# Disassemblers

- **Problem: pseudo-ops insert bit patterns**
  - If there were no pseudo-ops there would be a one-to-one mapping
- **Root problem:**
  - Data and instructions share memory
- **Programmers can sell machine code and buyers cannot change**
- **Open-source software movement: give away the source code with the program**
  - Companies get income by supporting the product
  - Customer can view code for security reasons and make bug fixes themselves

# Disassemblers

- **Disassembler:** a program that tries to recover the source program from the object program
  - Can never be 100% successful.
  - In large program data is scattered throughout the program.
  - Disassembler reads each byte and prints it out several times
    - Once as instruction
    - Once as ASCII
    - Once as 2's complement
  - People must reconstruct the program

# Symbols

- **Problem:** to know where data begins, must determine addresses
- **Problem 2:** if you remove data (eg a .WORD command), all addresses change
- **Solution:** symbols
- Instead of using addresses (at assembly level) can use symbols.
- Every time you assemble the program, assembler recalculates the addresses

# Symbols

- Rules:

- First char must be a letter
- Following char must be letters or numbers
- Max of 8 char long
- Case sensitive
- Define by placing the symbol at beginning of line
- Must terminate definition with a colon ( : )
- No spaces between last char of symbol and colon

# Symbols

## Example:

- Define the symbol num as address for a block of two bytes

num: .BLOCK d#2

- Use the symbol to reference the address

— do **not** use the colon

DECI num,d ; get the number

# Symbol Tables

- Assembler detects a symbol definition:  
stores the symbol and its value in a  
**symbol table**
  - The value of the symbol is the **address** in  
memory where the first byte of the object code  
generated from that line will be loaded.
- If you define symbols, the assembler  
listing will print out the symbol table
  - 2 pass assembler

# A program to add 1 to a decimal value

## Assembler Listing

Addr	Object code	Symbol	Mnemon	Operan
0000	04000D		BR	main
0003	0000	num:	.BLOCK	2
0005	202B20	msg:	.ASCII	" + 1 = \x00"
	31203D			
	2000			
000D	310003	main:	DECI	num, I
0010	390003		DECO	num, d
0013	410005		STRO	msg, d
0016	C10003		LDA	num, d
0019	700001		ADDA	1, i
001C	E10003		STA	num, d
001F	390003		DECO	num, d
0022	00		STOP	
0023			.END	

Replaced with BR 0x000D.  
Note that there is no colon here.

Replaced with DECI 0x0003,d  
Note that there is no colon here.

;Output + 1 =  
;A := the number  
;Add one to it  
;Store the sum  
;Output the sum

## Symbol table:

Symbol	Value	Symbol	Value
main	000D	msg	0005
num	0003		

# A nonsense program that illustrates the underlying von Neumann nature of the machine

## Assembler Input

What happens here?

```
this: DECO this,d  
STOP  
.END
```

## Assembler Listing

0000	390000	this:	DECO	this,d
0003	00		STOP	
0004			.END	

## Output

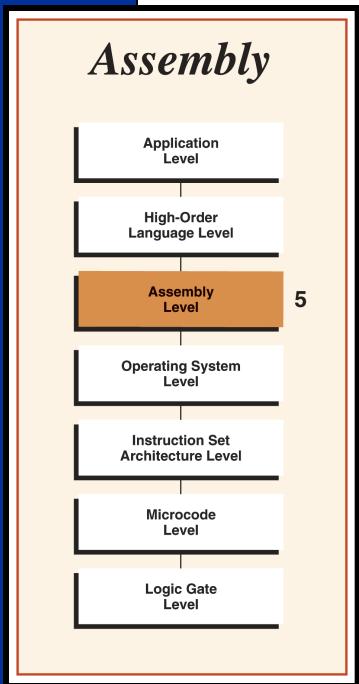
14592



# Chapter 5 Objectives

## ● Learning Objectives:

- **Describe the basic *assembly language format***
- **Translate simple instructions between assembly and machine languages**
- **Define the function of *directives***
- **Use directives to define data memory**
- **Use directives to instruct the assembler**
- **Describe the role of an assembler**



# Comparison of Languages

- Chapter 4 - ISA level (Level 3 )
  - Programs in Machine Language
- Chapter 5 - Assembly Language (level 5)

## Machine Language

```
31 00 14  
C1 00 14  
1C  
E1 00 16  
39 00 14  
41 00 18  
39 00 16  
00  
00 00  
00 00  
20 58 20 32 20 3D 20 00  
zz
```

## Assembly Language

```
DECI num, d  
LDA num, d  
ASLA  
STA answer, d  
DECO num, d  
STRO x2, d  
DECO answer, d  
STOP  
num: .block 2  
answer: .block 2  
x2: .ASCII " X 2 = \x00"  
.END
```

# Assembly Language

- ISA Level 3: 2 types of bit patterns
  - Instructions
  - Data
- Assembly Language: 2 types of corresponding statements
  - Instructions
    - Mnemonics for opcodes
    - Letters for addressing modes
  - Data
    - Pseudo-ops, also called dot commands

# Assembly Language

- **Assembly Language Instruction Format**
  - One instruction per line
  - Mnemonics replace bit codes:
    - Opcode
    - Register
    - Addressing mode
  - Extra spaces allowed
  - Comments may follow code

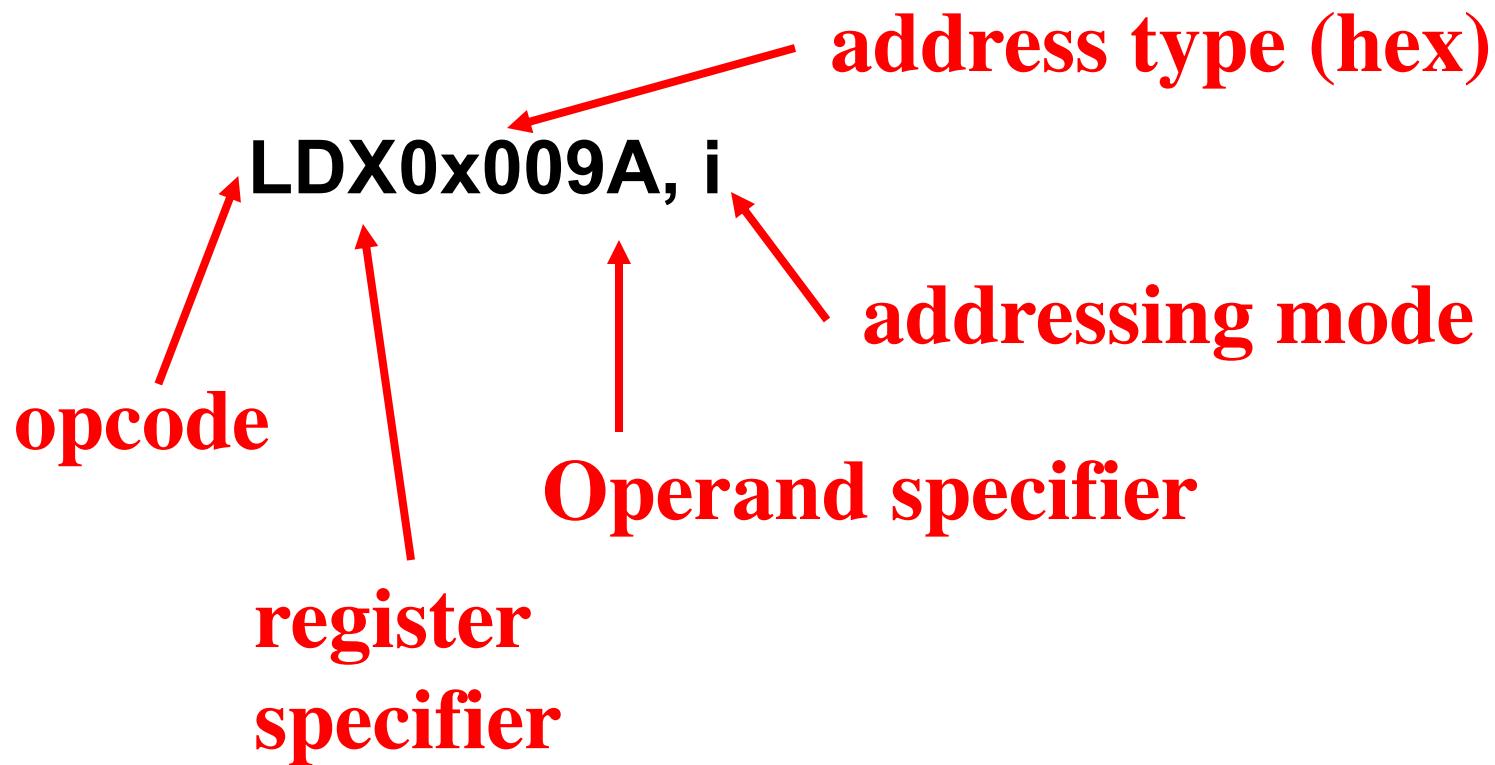
# Assembly Language

## 3 Types of Assembly Instructions

1. Direct translation of a machine instruction
2. Trap instruction
3. Pseudo-instruction

# Direct Translation: Instructions Mnemonics

- The instruction, register specifier, and addressing mode are all present:



# Addressing Modes

LDX 0x009A, i ← addressing mode

- Specify addressing mode by placing a comma followed by a letter at the end of the instruction

aaa	Addressing Mode	Letters
000	Immediate	i
001	Direct	d
010	Indirect	n
011	Stack-relative	s
100	Stack-relative deferred	sf
101	Indexed	x
110	Stack-indexed	sx
111	Stack-indexed deferred	sxf

# Direct Translation: Instructions Mnemonics

(Example 5.1 program comparison)

- LDX and LDA in assembler correspond to the same machine language statement (1100 raaa).
- The difference is in
  - the **r** (register bit).
    - A refers to the accumulator
    - X refers to the index register
  - And the Addressing mode bits **aaa**

1100 0011 0000 0000 1001 1010	LDA 0x009A, s
1100 0110 0000 0000 1001 1010	LDA 0x009A, sx
1100 1011 0000 0000 1001 1010	LDX 0x009A, s
1100 1110 0000 0000 1001 1010	LDX 0x009A, sx

# 1. Direct Translation of a Machine Instruction

Machine:            C        B        0        0        9        A  
                  1100 1011 0000 0000 1001 1010  
                            ↓  
Assembly:          LDI 0x009A, i ; my comment

	<i>Machine</i>	<i>Assembly</i>
Load	IR[0:3] = 1100	LDI
to index register	IR[4] = 1	
Mem address 009A	IR[8:23] = 00 9A	0x009A
Stack-relative addressing mode	IR[5:7] = 011	, i
Comments	Not allowed	Start with ;

## 2. Trap Instructions

- Exist at level 5 but not level 3
  - unimplemented opcode instructions from fig.4.6, pg 155
- Look like direct machine instructions, but in reality:
  - Transfer control to the operating system
  - Invoke functions that only the Operating System has the permission or capability to perform

NOPn	Unary “no operation”
NOP	Non-unary “no operation”
DECI	Decimal input
DECO	Decimal output
STRO	String output

# 3. Pseudo-Operations

- Assembly language statements
- Do not have opcodes and do not correspond to any instruction in Pep/8
- All pseudo-ops (except .BURN, .END, and .EQUATE) insert data bits into the machine language program
- Called assembly directives or dot commands

# Pseudo-Operations

- **ADDRESS**      The address of a symbol
- **ASCII**          A string of ASCII bytes
- **BLOCK**         A block of bytes
- **BURN**           Initiate ROM burn
- **BYTE**           A byte value
- **END**            The sentinel for the assembler
- **EQUATE**       Equate a symbol to a constant value
- **WORD**           A word value (two bytes)

# Pseudo-operations/Assembly directives

- Must place at least one space after the mnemonic or dot command
- No other restrictions on spacing
- No restrictions on capitalization.
  - Can use any combination of upper/lower case

# PEP/8 Example

## Assembler Input

```
;Stan Warford
;January 13, 2009
;A program to output "Hi"
;
CHARO    0x0007,d    ;Output 'H'
CHARO    0x0008,d    ;Output 'i'
STOP
.ASCII   "Hi"
.END
```

## Assembler Output

```
51 00 07 51 00 08 00 48 69 zz
```

## Program Output

```
Hi
```

- Program in Figure 5.3 is program in Fig 4.32 written in assembly language

- Pep/8 is line oriented.
  - One assembly language statement per line
  - Comments begin with semicolon and continue to end of line

# An assembly language program to output Hi

```
;Stan Wa Mnemonic for char output instruction
;January 20, 2008
;A program to output "Hi"
;
CHARO Assembler directive. Puts ascii code in
CHARO
STOP Halts assembler. Need both Stop
.END And .END
```

## Assembler Output

51 00 07 51 00 08 00 48 69 zz

## Program Output

Hi

# .ASCII Pseudo-op

- Generates contiguous bytes of ASCII characters
- Must enclose in double quotes
  - To include a double quote in the string, precede it with a backslash \"
  - To include a backslash, use two backslash characters \\
  - Newline is \n, tab is \t
  - To include arbitrary bytes in the string, use \x
    - Assembler expects the next two bytes to be hexadecimal digits

# .ASCII Pseudo-op

- Examples:

`.ascii "Hello\nWorld."`

`.ascii Hello\x0AWorld\x2E"`

Both produce the following sequence of bytes:

**48 65 6C 6C 6F 0A 57 6F 72 6C 64 2E**

# .END Pseudo-op

- .END is just a sentinel to the assembler to know when to stop translating
- .END actually produces zz

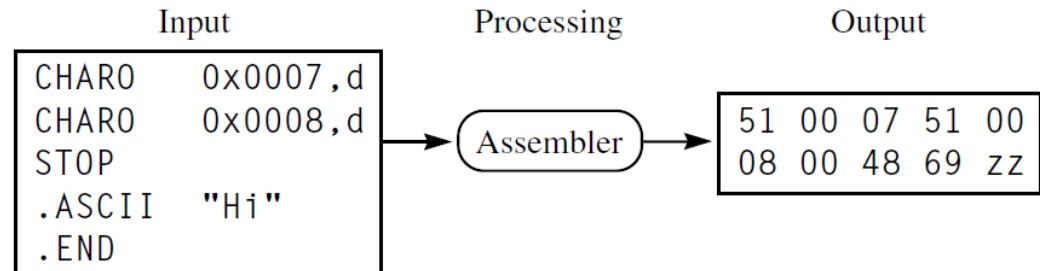
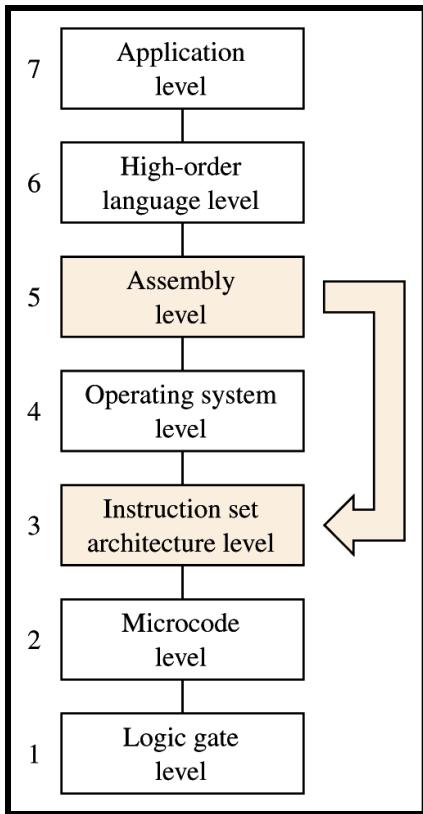
# Assemblers

- Assembly code must be translated into machine language
- Input: source program
- Output: object program
- Assembler does not load or execute a program
- Assembler - program written in machine language
- Assembler must be loaded before used

# Assemblers

- Output from the Pep/8 assembler:
  - Hex machine language file
    - In Object Code window pane
  - A *program listing*
    - In Assembly Listing window pane
    - Converts source code to consistent format of upper/lower case
    - Adds memory location address for each line
    - Adds machine code for each line
  - .END Pseudo-op does not generate any code

# The function of an assembler



The PEP/8 assembler takes your assembly code and translates it into machine code

- Most Asmb5 instructions -> ISA3 instructions
- Some Asmb5 instructions -> OS4 traps
- Some Asmb5 instructions -> ISA3 data or a Meta-instruction for the translator itself

# Program Figure 5.6

## Assembler Input

```
CHARI    0x000D,d      ;Input first character  
CHARI    0x000E,d      ;Input second character  
CHARO    0x000E,d      ;Output second character  
CHARO    0x000D,d      ;Output first character  
STOP  
.BLOCK   1             ;Storage for first char  
.BLOCK   1             ;Storage for second char  
.END
```

## Assembler Output

49 00 0D 49 00 0E 51 00 0E 51 00 0D 00 00 00 zz

## Program Input

up

## Program Output

pu

- **Assembly version of program Fig 4.34**

- **Inputs two char, prints them in reverse order**

# BLOCK...allocate bytes.

```
CHARI    0x000D,d  
CHARI    0x000E,d  
CHARO    0x000E,d  
CHARO    0x000D,d  
STOP  
.BLOCK 1  
.BLOCK 1  
.END
```

## Assembler Output

```
49 00 0D 49 00 0E 51 00 0E 51 00 0D 00 00 00 00 zz
```

## Program Input

up

## Program Output

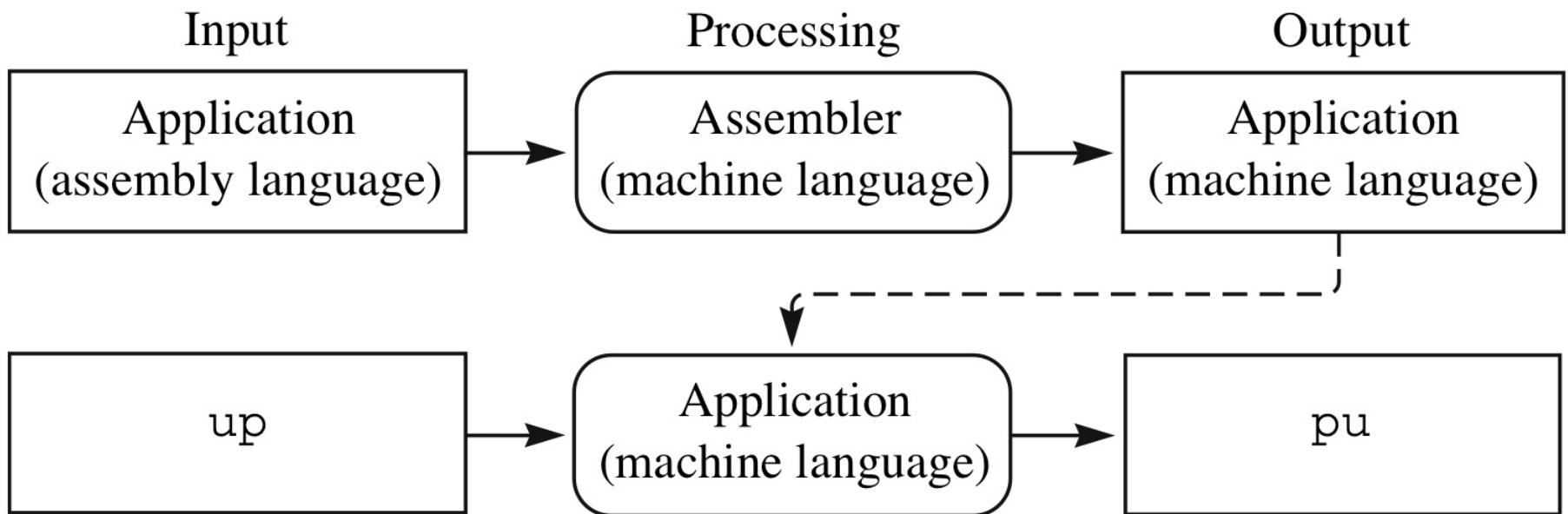
pu

Any number **not** prefaced with 0x is interpreted as decimal

;Output first character

Could replace these 2  
With a single  
**BLOCK 2**

# Two computer runs necessary for execution



# Assembler Listing for 5.8

Assembler Input (Allows some freedom in how you type it)

```
chari 0x000D,d    ;Input first character
CHARI  0x000E,d    ;Input second character
charo 0x000E,        d  ;Output second character
CHarO  0x000D,D    ;Output first character
      stop
.block 1 ;Storage for first char
.BLOCK 1      ;Storage for second char
      .END
```

Assembler Listing (An auxiliary output file, for the programmer's information and convenience)

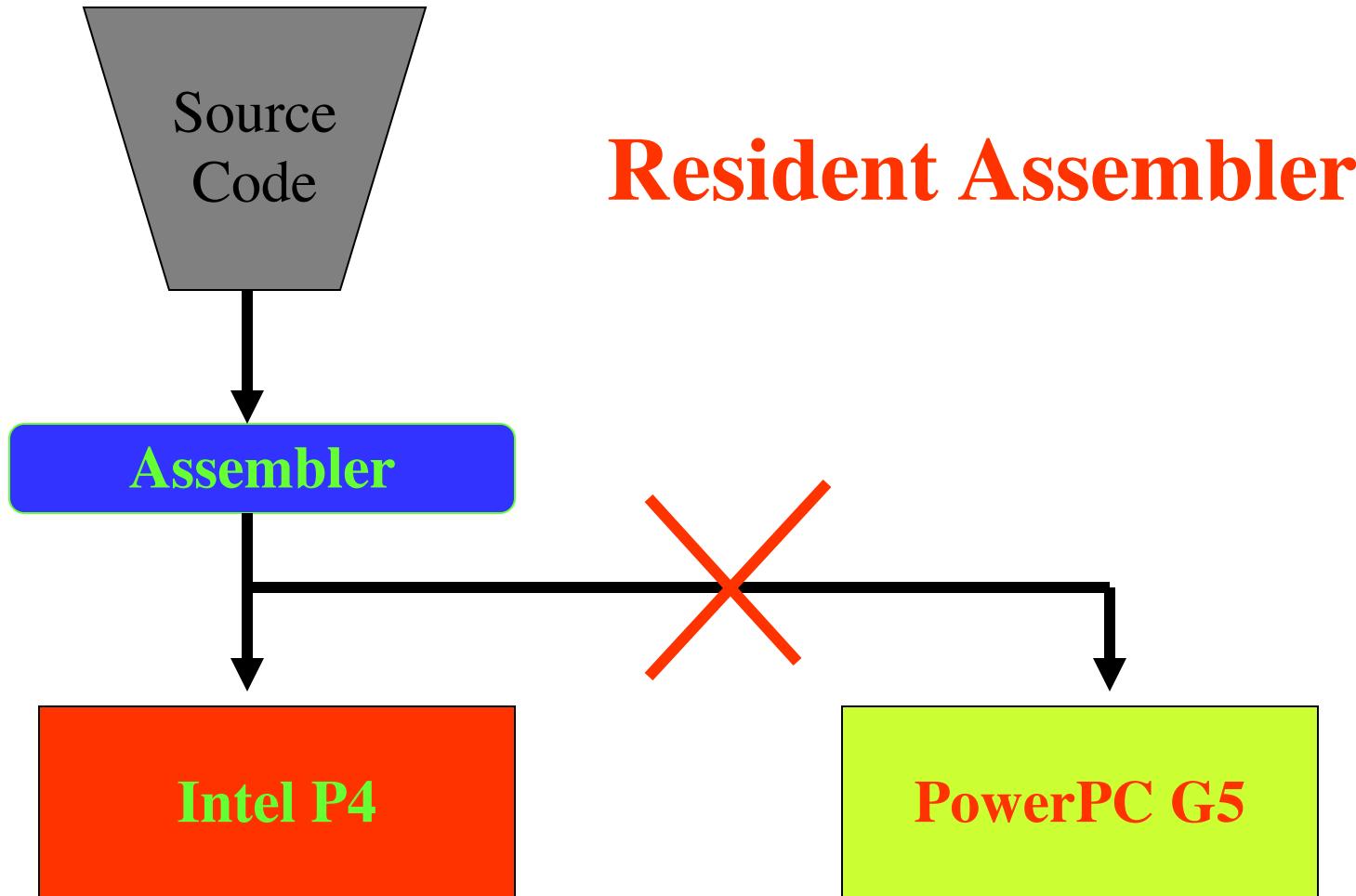
Object				
Addr	code	Mnemon	Operand	Comment
0000	49000D	CHARI	0x000D,d	;Input first character
0003	49000E	CHARI	0x000E,d	;Input second character
0006	51000E	CHARO	0x000E,d	;Output second character
0009	51000D	CHARO	0x000D,d	;Output first character
000C	00	STOP		
000D	00	.BLOCK	1	;Storage for first char
000E	00	.BLOCK	1	;Storage for second char
000F		.END		

- After Pep/8 assembles a program, can show an **Assembler listing**.
- Contains Address, hex code, assembly

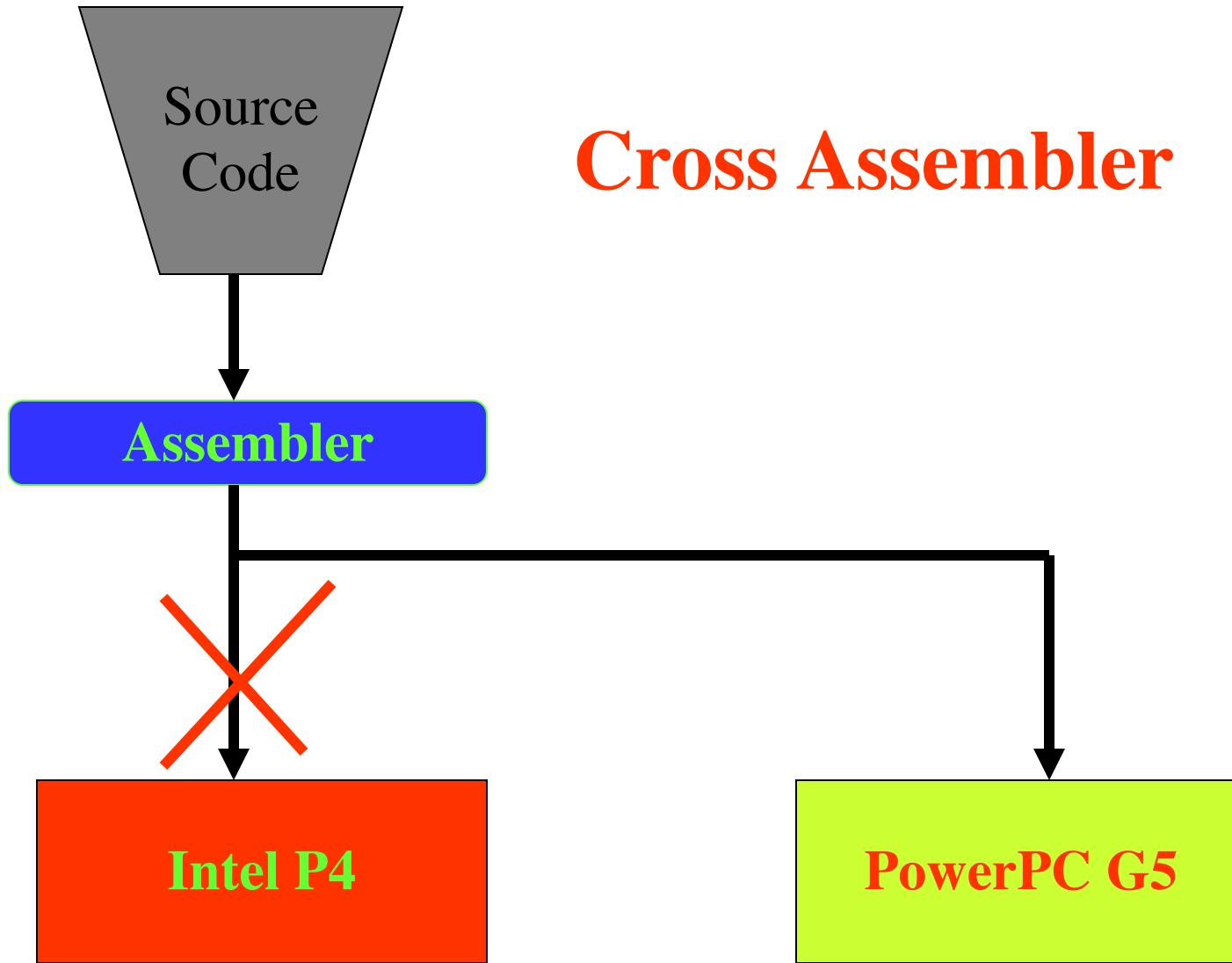
# Cross Assemblers

- Different chips have different instructions sets
- Assembler written in the same language as the language to which it translates
  - Called a resident assembler
- Can also write an assembler to translate to a language for a different CPU
  - Called a cross assembler

# Cross Assemblers



# Cross Assemblers



# Cross Assemblers

- Example: programming the controller in a microwave oven
  - Controller will have limited memory
  - Basic CPUs
  - So program and cross-assemble on a desktop computer
  - Transfer the code to the controller

# Immediate addressing

Learning Objectives:

- Define *immediate addressing* mode
- Describe *instructions and operand formats for data*: integers, characters, and strings
- Use immediate addressing for data output and computation
- Use *symbolic labels* for memory locations

# Direct addressing (from chapter 4)

- Oprnd = Mem[OprndSpec]
- Asmb5 letter: d
- Operand specifier is the *address* in memory of the operand

# Immediate addressing

- Oprnd = OprndSpec
- Asmb5 letter: i
- **The operand specifier *is* the operand.  
Do *not* go to memory.**

# Direct addressing vs. Immediate addressing

- Direct Address Mode (Ch 4): The Operand Specifier is the memory address of the Operand.
- Immediate Address Mode: the Operand Specifier IS the Operand
  - More efficient when you want to use a constant data value
- Comparison Example:
  - AddA 0x30,d = “Add the contents of mem cell 30 to the accumulator.”  
RTL:  $A \leftarrow A + \text{Mem}[30]$
  - AddA 0x30,i = “Add the value 30 (= 48 decimal) to the accumulator.”  
RTL:  $A \leftarrow A + 30$

# Direct addressing vs. Immediate addressing

- Suppose the machine's state is:
  - Reg[Accumulator] = F0F0
  - Mem[3E] = 0F0F

<i>After executing this instruction:</i>	AddA 0x3E, d (Direct mode)	AddA 0x3E,i (Immediate mode)
Accumulator	F0F0 + 0F0F = FFFF	F0F0 + 003E = F12E

# A program to output Hi using immediate addressing

```
0000 500048 CHARO
0003 500069 CHARO
0006 00 STOP
0007 .END
```

Output

Hi

'H', i  
'i' i

```
;Output 'H'  
;Output 'i'
```

character constant is enclosed in single quotes and generate 1 byte of code

Output

Hi

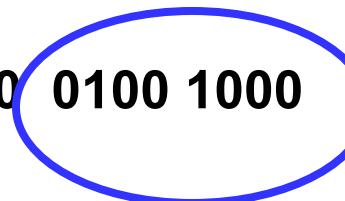
Machine bits: 0101 0000  
Addressing mode: 000

# Character constants

- Character constant - enclosed in single quotes
  - Always allocates one byte
    - In operand specifier, 9 leading 0's generated
      - 8 for first byte, 1 for first bit of second byte (ASCII is 7 bits)
      - Character constant is in the rightmost byte of the word
    - When `charo` executes, it puts the rightmost byte onto the bus for the output device

`charo 'H',i`

`0101 0000 0000 0000 0100 1000`



This byte is sent to  
the output device

# Immediate addressing

- **Advantages:**

- Program is shorter. 7 vs 9 bytes in this example
- Instruction executes faster; don't have to go to memory.

# Problems with Assembler so far...

- Programmer must know where the data is in memory
- Data goes at the bottom of memory, must write program before know where data goes
- if you insert an instruction later all data addresses change
- Single digit decimal input/output is tedious
  - CHARI/CHARO handle a single *character, not numbers*

C

## Chapter 4 Objectives

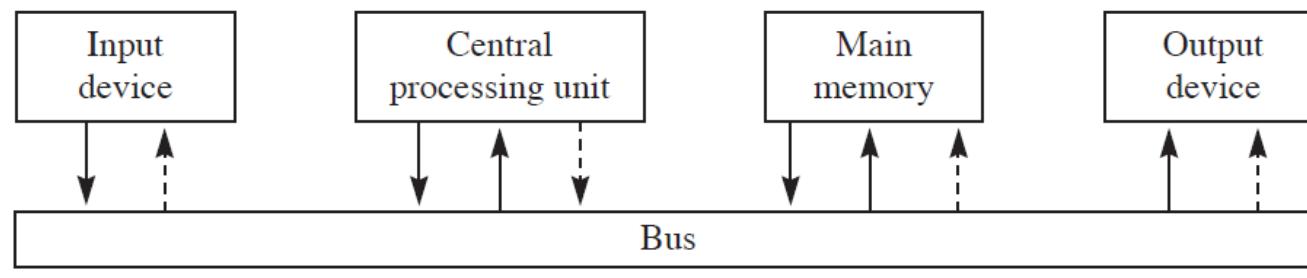
- Learn the components common to every modern computer system.
- Be able to explain how each component contributes to program execution.
- Understand a simple architecture invented to illuminate these basic concepts, and how it relates to some real architectures.
- Know how the program assembly process works.

# Hardware

- Computer - composed of 4 major components:
  - Central Processing Unit (CPU)
    - instructions are fetched and executed
  - Memory system
    - Programs and data are stored
  - Input Devices
  - Output Devices
    - (I/O) system is responsible for input and output data to and from the memory system

# Block Diagram of the Pep/8 Virtual Computer

- Overall system structure is shown in the following diagram:



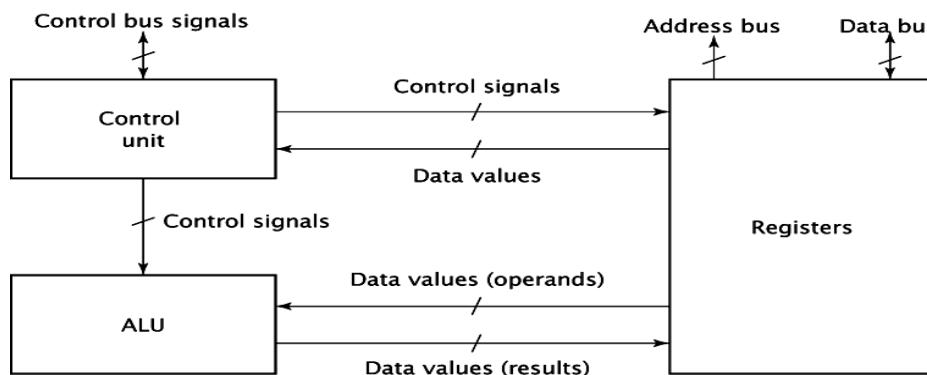
→ Data flow

---> Control

# CPU Organization

## Three basic components:

- 1) Registers
  - Store data in the CPU
  - Some are general purpose, some have specific purposes
    - program counter (PC) register*
    - instruction register (IR)*
- 2) ALU (Arithmetic Logic Unit)
  - Implements arithmetic and logic instructions
- 3) Control unit
  - Controls operation of the other CPU components
  - Asserts information on the control lines (READ, WRITE, IO/M)



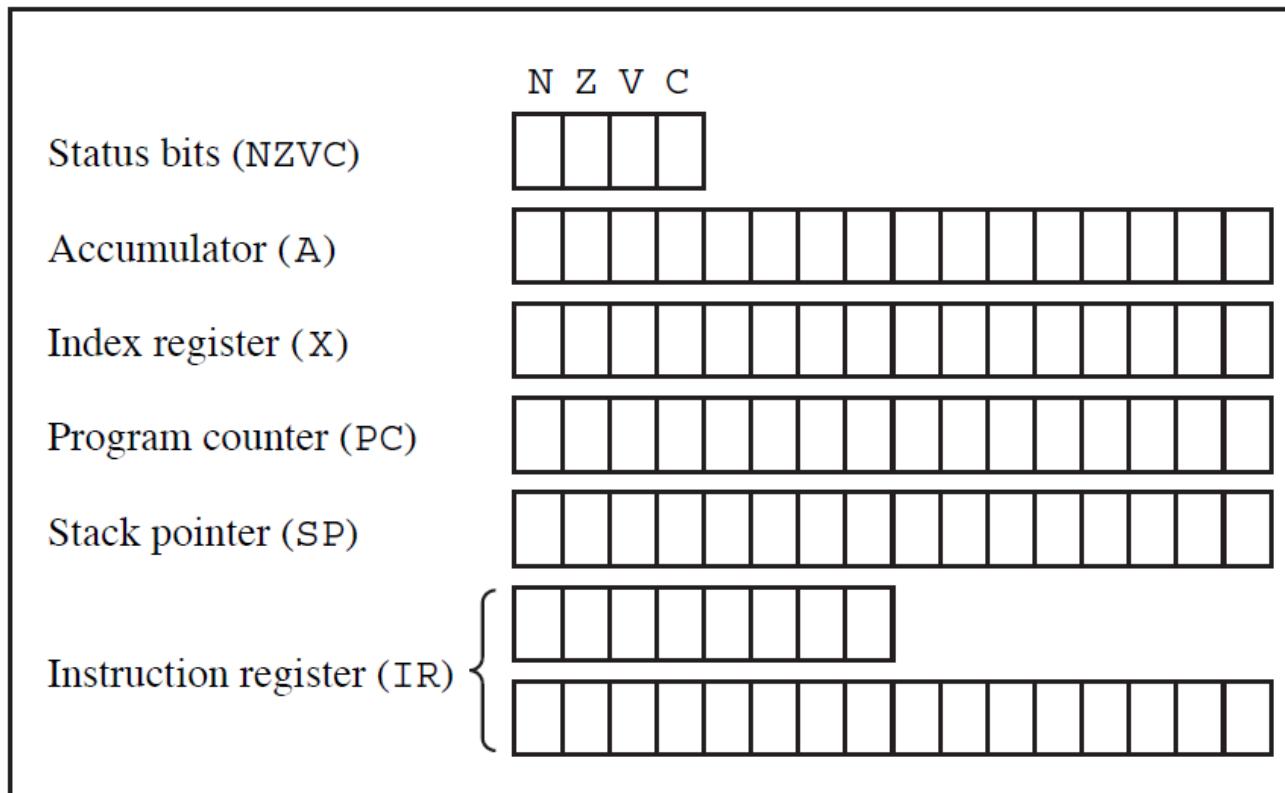
# PEP/8 CPU

- Our virtual computer contains six specialized registers:
  - 4-bit Status Register (NZVC)
  - 16-bit accumulator (A)
  - 16-bit index register (X)
  - 16-bit program counter (PC)
  - 16-bit stack pointer (SP)
  - 24-bit instruction register (IR)

# PEP/8 CPU

## PEP/8 CPU registers

Central processing unit (CPU)



# PEP/8 CPU

## PEP/8 CPU register definitions

Register (Abbrev)	Bits	Contents / Function
Status	4	Status bits {N, Z, V, C}
Accumulator (A)	16	Result of ALU operation
Index Register (X)	16	An array index
Program Counter (PC)	16	Addressing the program memory
Stack Pointer (SP)	16	Addressing the run-time stack
Instruction Register (IR)	24	The current program instruction

# Main Memory System

- **Memory system**
  - **Array of memory locations**
  - **Each store a multiple-bit binary data (typically 8 bits called a byte).**
  - **Each are selectable by address for either read or write operation**
  - **Normally has**
    - address input for the address to select the location
    - data input for the data to be written to the location
    - data output for the data read from the location
    - chip enable control to enable the entire memory
    - read/write control to start read or write operation

# Memory Organization

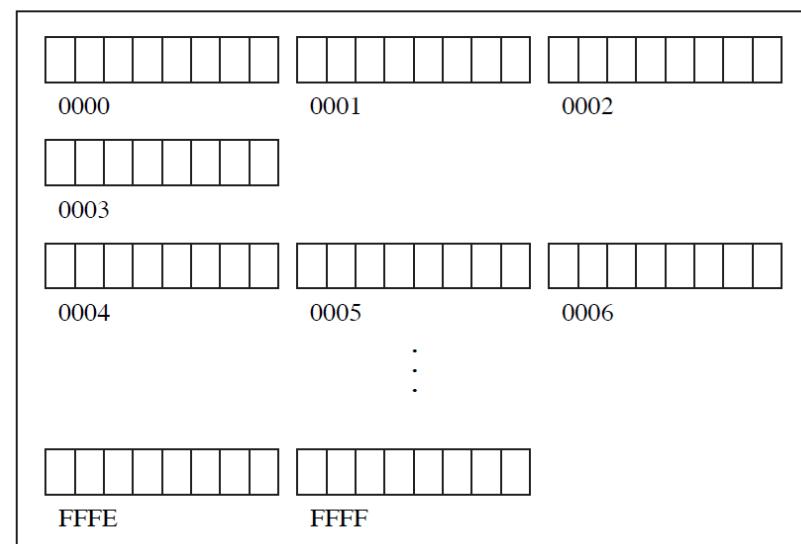
## Computer memory

- Linear array of addressable storage cells that are similar to registers
- Can be byte-addressable
- Can be word-addressable
  - typical word is two or more bytes

PEP/8 main memory:

- 65,536 Byte-size storage locations
- $65,536 = 2^{16}$ , meaning
  - 16-bit addresses
  - or 4 hex digits
  - 0000 to FFFF

Main memory



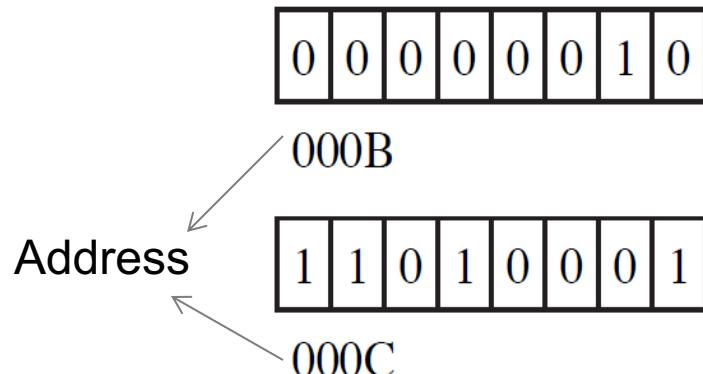
# Memory Organization

## ● PEP/8 main memory

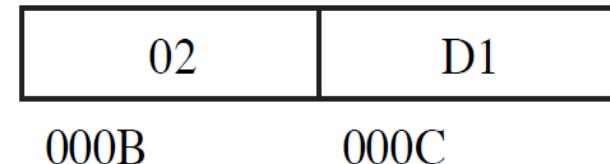
- Configuration: Array of 64K Bytes
- Use: Select a particular memory location by providing a binary address:
  - $64K = 2^{16}$  → Addresses are 16 bits long
  - Address range is 0000 : FFFF (hex)
- Cautionary notes:
  - Pay attention to memory address vs. memory contents
  - Memory cells do not know what type of data they hold (int, float, char, or instruction)

# Memory Organization

- PEP/8 main memory



(a) The content in binary.



(b) The content in hexadecimal.

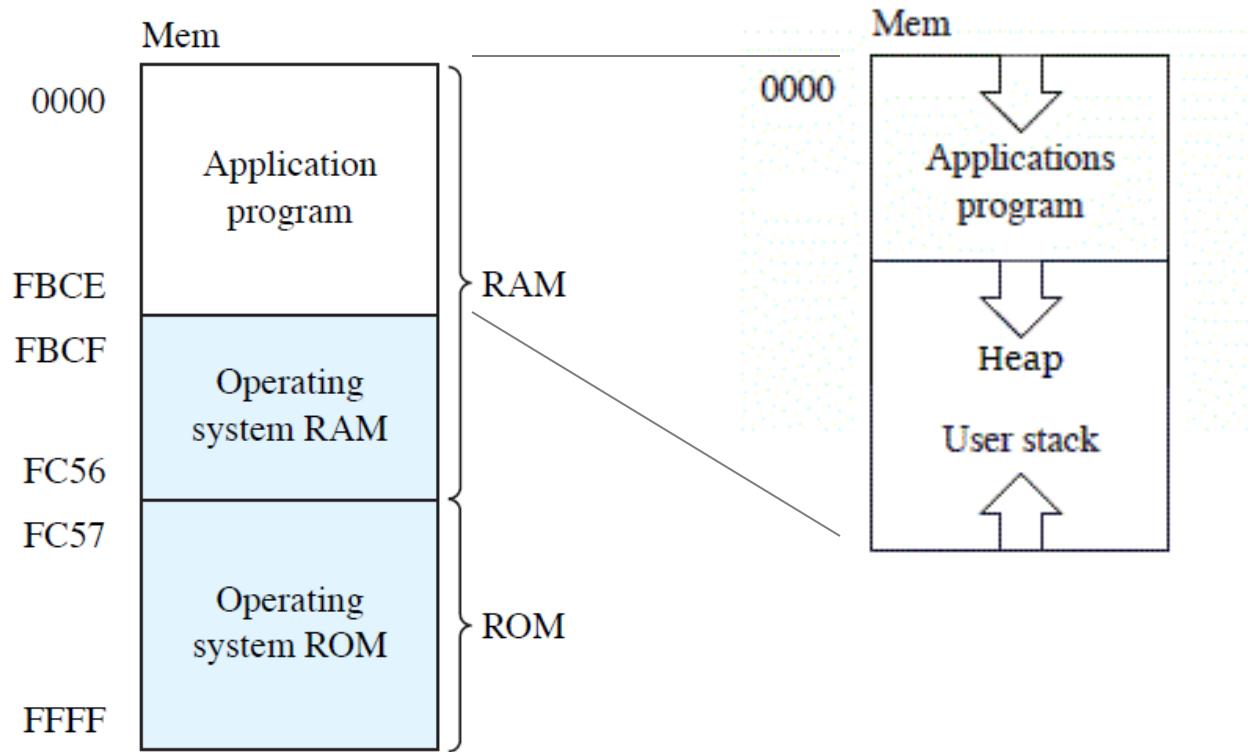
000B            02D1

(c) The content in a machine  
language listing.

© 2010 Jones and Bartlett Publishers, LLC ([www.jbpub.com](http://www.jbpub.com))

# Memory Organization

- Memory map for PEP/8:



# I/O Subsystem

- A computer communicates with the outside world through its input/output (I/O) subsystem.
- I/O devices connect to the CPU through various interfaces.
- PEP/8 Input devices
  - Text file (or GUI window)
  - Keyboard
- PEP/8 Output devices
  - Text file (or new GUI window)
  - Screen

# I/O Subsystem

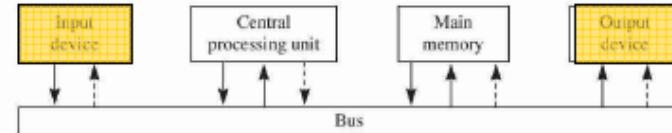
- Two modes for handling input and output:

- **Interactive Mode** (Pep/8 “Terminal I/O”)

- During program execution, the program (asks for and) receives input, from a keyboard, mouse, etc.
    - Output is displayed immediately, on a screen or printer.

- **Batch Mode**

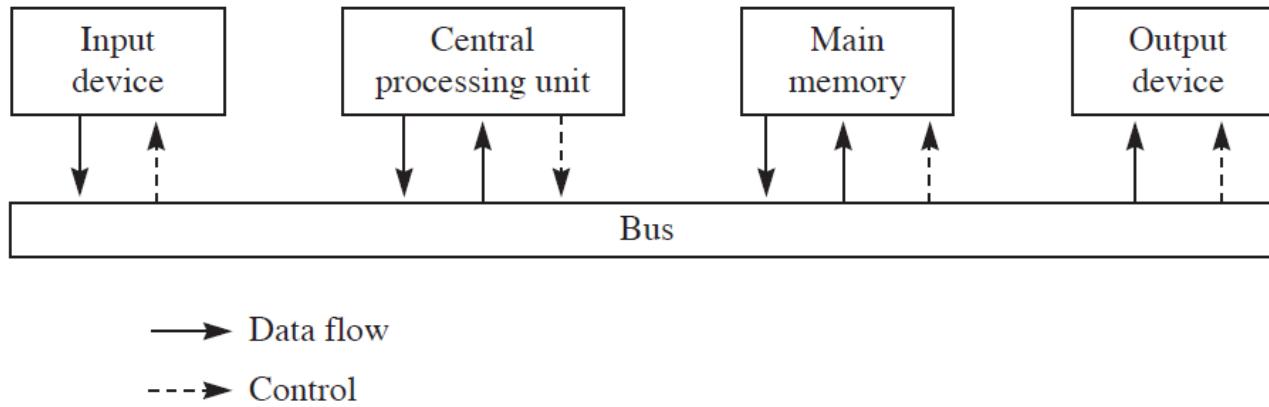
- All input must be prepared before the program runs (In the Pep/8 you type the input into a special window.)
    - Output might not be directly visible (e.g. stored in a file)



# Data and Control (system busses)

- Four major components are connected by:
  - Address bus - provides the address for the memory location to be addressed for read and write
  - Data bus - transfers data
    - from CPU or I/O to memory
    - from memory to CPU or I/O
  - Control bus - consists of
    - many control signals to control overall system operations
    - many status data from memory and I/O devices

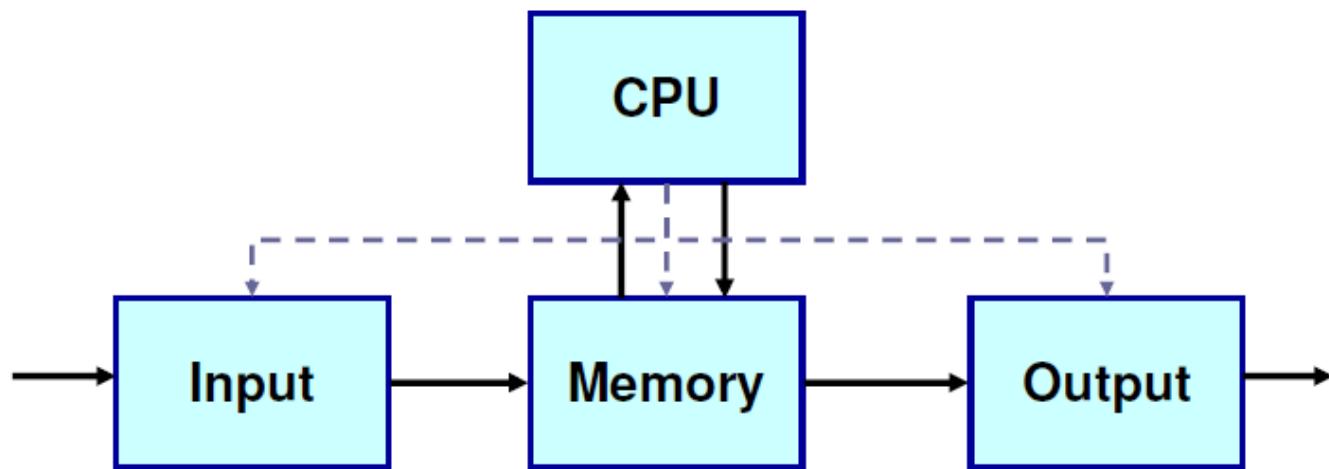
# Data and Control (PEP/8)



- **Control:** *always* flows from CPU → others
- **Data:** In Pep/8, Memory must *always* be either the source or the destination
  - Other architectures may allow other transfer modes

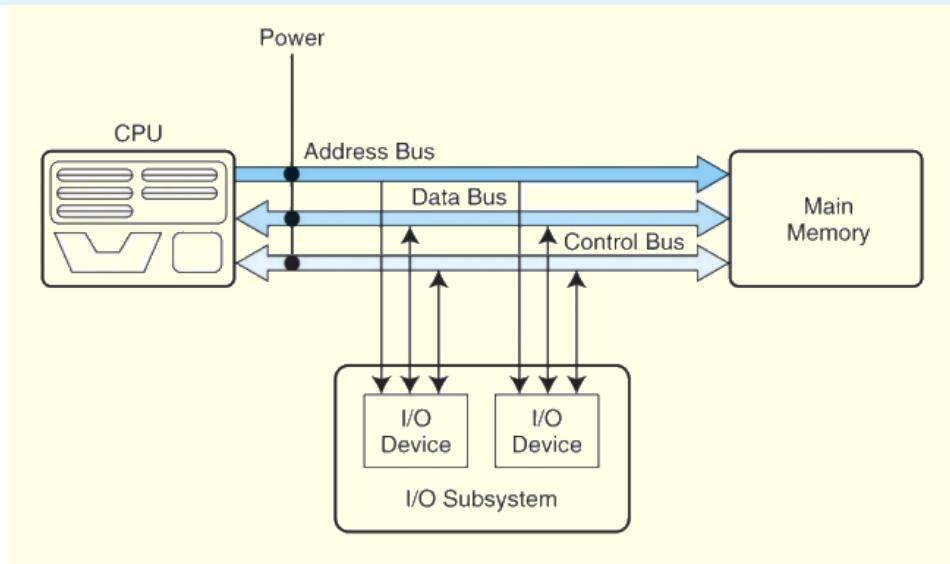
# Data and Control (PEP/8)

- No direct flow between Input/Output and the CPU.
- All the “computing” happens in the CPU.
- But a big part of computer operation is just moving data from place to place.



# System Busses

- **Buses consist of:**
  - **Data lines**
    - convey bits from one device to another
  - **Control lines**
    - determine the direction of data flow
  - **Address lines**
    - determine the location of the source or destination of the data



# The PEP/8 Virtual computer

- Bring together many of the ideas using a very simple model computer
- PEP/8 was designed for illustrating basic computer system concepts
- Too simple to do anything useful
- An understanding of PEP/8 functions will help comprehend more complex system architectures

# The PEP/8 Virtual computer

- What do we mean “Virtual Machine”?
  - Conceptual architecture is defined
  - Instruction set (language) is defined
  - No physical machine exists
  - Operation is simulated by software

# Machine Language Characteristics

- Machine Language is **machine-dependent**  
(tailored to a particular hardware architecture)
- A Machine Language program is simply a sequence of hardware instructions. Each instruction...
  - does only one basic operation
  - Is rigidly formatted
  - is subdivided into a set of fields
- **Operands** = parameters (input) and result (output)
  - Operands can be a register, a memory cell, or a constant
  - Pep/8 instructions have 0 or 1 operands

# The PEP/8 Virtual computer

- **Instruction Set Architecture (ISA)**
  - specifies the format of its instructions
  - Specifies the primitive operations that the machine can perform
- **ISA is an interface between a computer's hardware and its software**
- **Some ISAs include hundreds of different instructions for processing data and controlling program execution**
- **PEP/8 ISA consists of 39 instructions**

# The PEP/8 Virtual computer instruction list

Instruction Specifier	Instruction	Instruction Specifier	Instruction
0000 0000	Stop execution	0110 0aaa	Add to stack pointer (SP)
0000 0001	Return from trap	0110 1aaa	Subtract from stack pointer (SP)
0000 0010	Move stack pointer (SP) to accumulator (A)	0111 raaa	Add to register r
0000 0011	Move NZVC flags to accumulator (A)	1000 raaa	Subtract from register r
0000 010a	Branch unconditional	1001 raaa	Bitwise AND to register r
0000 011a	Branch if less than or equal to	1010 raaa	Bitwise OR to register r
0000 100a	Branch if less than	1011 raaa	Compare register r
0000 101a	Branch if equal to	1100 raaa	Load register r from memory
0000 110a	Branch if not equal to	1101 raaa	Load byte register r from memory
0000 111a	Branch if greater than or equal to	1110 raaa	Store register r to memory
0001 000a	Branch if greater than	1111 raaa	Store byte register r to memory
0001 001a	Branch if V		
0001 010a	Branch if C		
0001 011a	Call subroutine		
0001 100r	Bitwise invert register r		
0001 101r	Negate register r		
0001 110r	Arithmetic shift left register r		
0001 111r	Arithmetic shift right register r		
0010 000r	Rotate left register r		
0010 001r	Rotate right register r		
0010 01n	Unimplemented opcode, unary trap		
0010 1aaa	Unimplemented opcode, nonunary trap		
0011 0aaa	Unimplemented opcode, nonunary trap		
0011 1aaa	Unimplemented opcode, nonunary trap		
0100 0aaa	Unimplemented opcode, nonunary trap		
0100 1aaa	Character input		
0101 0aaa	Character output		
0101 1nn	Return from call with n local bytes		

- a or aaa is an addressing mode
- r is a register
- n is an integer



# The PEP/8 Virtual computer instructions

## ● This is the format of a PEP/8 instruction:

Instruction  
specifier



Operand  
specifier



(a) The two parts of a nonunary instruction

Instruction  
specifier



(b) A unary instruction

- The Instruction Specifier is mandatory
- Most instructions also need an Operand Specifier

# Pep/8 Instruction Specifier

Total of 8 bits, divided into 3 parts:

1. Opcode (4 to 8 bits)
  - Tells what type of operation this is. (Types include arithmetic, logical, memory transfer, branch)
2. Addressing mode (1 or 3 bits)
  - Tells how the Operand Specifier is to be translated to a memory address
3. Register selection (1 bit)
  - Tells which register should store the result.

# Addressing Modes

- How many ways can the processor access memory locations?

Eight for 3-bit *aaa* field for non-unary instructions:

- **Immediate**
  - Instruction specifies actual value to use, not address
- **Direct**
  - Instruction includes absolute memory address
- **Indirect**
  - Instruction contains an address that contains an address. (which means "read address 5, use its contents as an address to fetch")
- **Relative (Stack-relative, Stack-relative deferred)**
  - Instruction operand is an offset from some base address
- **Index (Stack-index, Stack-index deferred)**
  - Instruction operand supplies offset from an index register

Two for 1-bit *a* field for unary instructions

- **Immediate**
- **Indexed**

This chapter focuses only on the direct addressing mode.

# PEP/8 Machine Instruction

## A complete machine instruction

- Add the contents of memory address 5 to the Accumulator register:

0111 0001 0000 0000 0000 0101

- Opcode = ADD (first 4 bits) 0111raaa
- Register = Accumulator (next 1 bit) 0
- Addressing = Direct (next 3 bits) 001
- Memory address operand = 5 (next 16 bits)

0000 0000 0000 0101

- Note: we are not computing  $\text{Acc} = \text{Acc} + 5$ , we are computing  $\text{Acc} = \text{Acc} + \text{Mem}[5]$

# PEP/8 Machine Instruction

- What happens when the Add instruction is executed?

## The add instruction

- Instruction specifier: 0111 raaa
- Adds one word (two bytes) from memory to register r

$$r \leftarrow r + \text{Oprnd} ; N \leftarrow r < 0 , Z \leftarrow r = 0 , \\ V \leftarrow \{overflow\} , C \leftarrow \{carry\}$$



# Von Neumann

## John von Neumann

- Mathematician, physicist, and computer scientist
- Born 1903 in Hungary
- Emigrated to U.S. in early 1930's, teaching at Princeton
- Lead the Theoretical Division of the Manhattan Project, which developed the atomic bomb
- **Proposed an architecture for storing computer programs in memory, alongside the data**

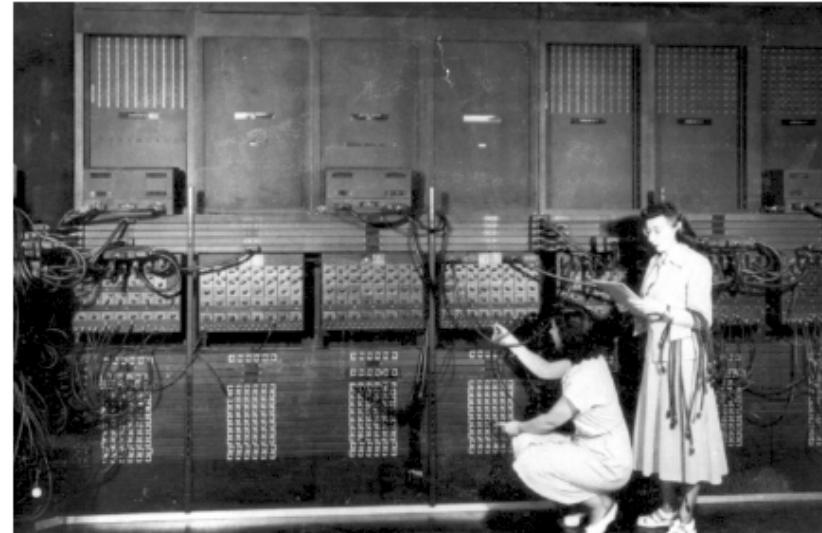


From "Researchers in Computer Architecture and Compilers", Carnegie Mellon University,  
<http://www.cs.cmu.edu/~mihai/whoswho/photos.html>

# Von Neumann

## Pre-von Neumann Computers

- Memory was used for data only.
- Programs were “hard-wired”.
  - Technicians routed hundreds of wires between circuits to implement a program.
  - Changes were slow and error-prone



"U.S. Army Photo" from the archives of the ARL Technical Library.  
Standing: Ester Gerston Crouching: Gloria Ruth Gorden. U.S. Army  
Research Laboratory. <http://ftp.arl.mil/ftp/historic-computers/>

# Von Neumann Execution Cycle

- CPU goes through four steps to process an instruction:
  - (1) *fetch* an instruction from memory
  - (2) *decode* the instruction and find out its opcode
  - (3) increment the program counter
  - (4) *execute* the instruction and store results

## von Neumann Execution Cycle

- load machine language program;
- initialize PC and SP registers;
- do { // von Neumann cycle
  - /\* 1 \*/ **Fetch** the next instruction;
  - /\* 2 \*/ **Decode** the instruction specifier;
  - /\* 3 \*/ **Increment** PC;
  - /\* 4 \*/ **Execute** the instruction fetched;
- }
- while (instruction != STOP);
- Virtually all computers today use the von Neumann architecture

# Von Neumann Execution Cycle

- In pseudo-code, particular to our PEP/8 computer:

*Load the machine language program into memory starting at address 0000*

$\text{PC} \leftarrow 0000$

$\text{SP} \leftarrow \text{Mem}[FFF8]$

**do** {

*Fetch the instruction specifier at address in PC*

$\text{PC} \leftarrow \text{PC} + 1$

*Decode the instruction specifier*

**if** (*the instruction is not unary*) {

*Fetch the operand specifier at address in PC*

$\text{PC} \leftarrow \text{PC} + 2$

}

*Execute the instruction fetched*

}

**while** ((*the stop instruction does not execute*) &&  
(*the instruction is legal*))

# Von Neumann Execution Cycle

## Pep/8 Execution Cycle

### 1. Fetch

- $IR[0:7] \leftarrow Mem[PC]$

### 2. Decode

- If  $IR[0:7]$  is non-unary =>      Fetch  $IR[8:23] \leftarrow Mem[PC+1];$

### 3. Increment

- If  $IR[0:7]$  is non-unary =>       $PC \leftarrow PC + 3;$   
Else =>                                     $PC \leftarrow PC + 1;$

### 4. Execute

Repeat

# Von Neumann Execution Cycle

## von Neumann Advantages

- Easy to load and revise programs
- General-purpose machine
  - Meets the definition of a Universal Turing Machine
- Multi-process, multi-level software →  
high level language
- Machine cannot distinguish instructions from  
data
  - Clever programmers can write self-modifying code

# Von Neumann Execution Cycle

## von Neumann Problems

- Machine cannot distinguish instructions from data
  - Common programming mistakes:
    - Interpreting data as instruction
    - Interpreting instruction as data
- Memory-Processor Bottleneck
  - To perform an instruction, both the instruction and data have to move from Memory to the CPU.
  - Bus traffic will be heavy.
  - Memory chips are many times slower than processor chips, so the processor wastes time waiting.



## Decimal to Binary Conversions

- **Binary numbering system**
- **Difficult to read long strings of binary numbers**
  - For example:  $11010100011011_2 = 13595_{10}$
- **Binary values expressed using hexadecimal**
- **Hexadecimal numbering system**
  - Uses numerals 0 through 9 and letters A through F
  - Decimal number 12 is C<sub>16</sub>.
  - Decimal number 26 is 1A<sub>16</sub>.
- **Easy to convert between base 16 and base 2**
  - Because  $16 = 2^4$ .
- **Group binary digits into groups of four to convert from binary to hexadecimal**

**A group of four binary digits is called a hextet**

## Decimal to Binary Conversions

- Group by hextets to convert binary number  $11010100011011_2$  ( $= 13595_{10}$ ) into hexadecimal:

0011	0101	0001	1011
3	5	1	B

- Octal (base 8) values are derived from binary by using groups of three bits ( $8 = 2^3$ ):

011	010	100	011	011
3	2	4	3	3

Octal was useful when computers used six-bit words

# Signed Integer Representation

- **Negative values - high-order bit indicates the sign**
  - High-order bit is the leftmost bit in a byte
  - High-order bit is called the most significant bit
- **Remaining bits contain the value of the number**
- **Three ways to express signed binary:**
  - Signed magnitude
  - One's complement
  - Two's complement

# Signed Integer Representation

- **Signed magnitude for 8-bit word**
  - absolute value of number in the 7 right
- **Example: 8-bit signed magnitude**
  - Positive 3 is: 00000011
  - Negative 3 is: 10000011
- **Computers perform arithmetic operations on signed magnitude**
  - ignore signs of the operands while performing a calculation
  - applying appropriate sign after calculation is complete

## Signed Integer Representation

- **Binary addition - easy - four rules:**

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

- **This system makes it possible for digital circuits to carry out arithmetic operations**
  - We will describe these circuits in later chapters.

**Let's see how the addition rules work with signed magnitude numbers . . .**

# Signed Integer Representation

- **Example:**
  - Using signed magnitude binary arithmetic
  - Find the sum of 75 and 46
- **Step 1**
  - convert 75 and 46 to binary
  - separate (positive) sign bits from magnitude bits

$$\begin{array}{r} 0 \quad 1001011 \\ 0 + 0101110 \\ \hline \end{array}$$

# Signed Integer Representation

- **Step 2**
  - Start with rightmost bit and work left
- **Step 3**
  - Note carries above next bit

$$\begin{array}{r} 0 \quad 1001011 \\ 0 + 0101110 \\ \hline 1 \end{array}$$

$$\begin{array}{r} 0 \quad 1001011 \\ 0 + 0101110 \\ \hline \quad \quad \quad 1 \\ \quad \quad 01 \end{array}$$

$$\begin{array}{r} \quad \quad \quad 1 \ 1 \ 1 \\ 0 \quad 1001011 \\ 0 + 0101110 \\ \hline 1001 \end{array}$$

# Signed Integer Representation

- **Example...**

- **Using signed magnitude binary arithmetic**
  - **Find the sum of 75 and 46**

- **Worked through all eight bits**

$$\begin{array}{r} & \overset{1}{1} \overset{1}{1} \\ 0 & 1 0 0 1 0 1 1 \\ 0 & + 0 1 0 1 1 1 0 \\ \hline 0 & 1 1 1 1 0 0 1 \end{array}$$

In this example, the two values sum fit into seven bits.  
If that is not the case, this needs additional methods.

# Signed Integer Representation

- **Example:**
  - Using signed magnitude binary arithmetic
  - Find the sum of 107 and 46
- **Carry from the seventh bit**
  - overflows and is discarded
  - erroneous result:  $107 + 46 = 25$

$$\begin{array}{r} & 1 & 1 & 1 & 1 \\ & | & | & | & | \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & + & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}$$

# Signed Integer Representation

- Signs in signed magnitude representation
- Example:
  - Using signed magnitude binary arithmetic
  - Find the sum of - 46 and – 25
- Signs are the same
  - Add the numbers
  - Supply negative sign when done

$$\begin{array}{r} & \overset{1}{1} \\ 1 & 0101110 \\ 1 & + 0011001 \\ \hline 1 & 1000111 \end{array}$$

# Signed Integer Representation

- Mixed sign addition (or subtraction) is done the same way.
- Example:
  - Using signed magnitude binary arithmetic
  - Find the sum of 46 and - 25
- Sign of result gets sign of number with larger absolute value
  - Note “borrows” from second and sixth bits

$$\begin{array}{r} & \text{0} & \text{2} & & \text{0} & \text{2} \\ & \text{0} & \cancel{\text{1}} & \text{0} & \text{1} & \cancel{\text{1}} & \text{0} \\ 1 & + & \text{0} & \text{0} & \text{1} & \text{1} & \text{0} & \text{0} \\ \hline & \text{0} & & & \text{0} & \text{1} & \text{0} & \text{1} \end{array}$$

# Signed Integer Representation

- **Signed magnitude representation**
  - Easy for people to understand
  - Requires complicated computer hardware
- **Another disadvantage of signed magnitude**
  - Allows two different representations for zero
    - positive zero
    - negative zero
- **Computers systems employ *complement systems* for numeric value representation**

# Signed Integer Representation

- **Complement systems**
  - Negative values are represented by some difference between a number and its base
- ***Diminished radix complement systems***
  - Negative value is given by the difference between the absolute value of a number and one less than its base
- **Binary system - one's complement**
  - Flip the bits of a binary number

# Signed Integer Representation

- **Example - in 8-bit one's complement**
  - Positive 3 is: 00000011
  - Negative 3 is: 11111100
- **One's complement**
  - Negative values are indicated by a 1 in the high order bit
- **Complement systems**
  - Eliminate the need for subtraction
  - Difference of two values is found by adding

# Signed Integer Representation

- **One's complement addition**
  - Carry bit is “carried around” and added to the sum
- **Example:**
  - Using one's complement binary arithmetic
  - Find sum of 48 and – 19
  - Note:
    - 19 in one's complement is 00010011
    - -19 in one's complement is 11101100
  - Result: 29

$$\begin{array}{r} & \overset{1}{\textcolor{yellow}{1}} \ 1 \\ & 00110000 \\ & 11101100 \\ \hline & 00011100 \\ & + \ 1 \\ \hline & 00011101 \end{array}$$

# Signed Integer Representation

- “End carry around” adds complexity
  - one’s complement is easier to implement than signed magnitude
- Disadvantage
  - two different representations for zero
  - positive zero and negative zero.
- Two’s complement solves this problem
- Two’s complement
  - *radix complement* of binary numbering system

# Signed Integer Representation

- **To derive two's complement:**
  - **Positive number**
    - convert it to binary
  - **Negative number**
    - Find the one's complement of number and add 1
- **Example:**
  - **One's complement - positive 3:** 00000011
  - **One's complement - negative 3:** 11111100
  - **Two's complement - negative 3:** 11111101

# Signed Integer Representation

- **Two's complement arithmetic**
  - Add two binary numbers
  - Discard any carries
- **Example:**
  - Using one's complement binary arithmetic
  - Find sum of 48 and -19
  - Note:
    - 19 in one's complement is:  
00010011
    - -19 in one's complement is:  
11101100
    - -19 in two's complement is:  
11101101
  - Result: 29

$$\begin{array}{r} & \textcircled{1} \\ & \swarrow \\ \begin{array}{r} 1 \ 1 \\ 00110000 \\ + 11101101 \\ \hline 00011101 \end{array} \end{array}$$

# Signed Integer Representation

- **Finite number of bits to represent a number**
  - Calculations may become too large for memory storage allotted
- **Can't prevent overflow**
  - Can detect overflow
- **Complement arithmetic**
  - Overflow condition is easy to detect

# Signed Integer Representation

- **Example:**
  - Using two's complement binary arithmetic
  - Find sum of 107 and 46
- **Nonzero carry from seventh bit**
  - overflows sign bit
  - erroneous result:  $107 + 46 = -103$

$$\begin{array}{r} \textcolor{pink}{1}1\ 1\ 1\ 1\\ 01101011 \\ + 00101110 \\ \hline 10011001 \end{array}$$

Rule for detecting signed two's complement overflow:  
When the “carry in” and the “carry out” of the sign bit differ  
overflow has occurred.

# Signed Integer Representation

- **Signed and unsigned numbers are both useful**
  - Memory addresses are always unsigned
- **Unsigned integers can express twice as many values as signed numbers**
  - Adding one column adds power of 2
- **Unsigned value “wraps around” causes error**
  - $1111 + 1 = 0000$
  
- **Good programmers stay alert for this kind of problem.**

## Signed Integer Representation

- **Overflow and carry are important in lower level programming**
- **Signed number overflow means nothing in the context of unsigned numbers, which set a carry flag instead of an overflow flag.**
- **If a carry out of the leftmost bit occurs with an unsigned number, overflow has occurred.**
- **Carry and overflow occur independently of each other.**

## Signed Integer Representation

Expression	Result	Carry?	Overflow?	Correct result
0100 (+4) + 0010 (+2) Signed magnitude	0110 (+6)	No	No	Yes
0100 (+4) + 0110 (+6) Signed magnitude	1010 (-6)	No	Yes	No
1100 (-4) + 1010 (-2) Signed magnitude	1110 (-6)	Yes	No	Yes
1100 (-4) + 1010 (-6) Two's Complement	0110 (+6)	Yes	Yes	No



## Character Codes

- Character codes evolved as computers evolved
- Richer character codes evolved for larger computer memories and storage devices
- Earliest computer coding systems used six bits
- Binary-coded decimal(BCD) - one of the early codes
  - Used by IBM mainframes in the 1950s and 1960s

## Character Codes

- **BCD was extended to an 8-bit code:**
  - Extended Binary-Coded Decimal Interchange Code (EBCDIC)
- **EBCDIC**
  - supported upper *and* lowercase alphabetic characters
  - supported special characters, such as punctuation and control characters
- **EBCDIC and BCD - still in use by IBM mainframes**

## Character Codes

- **ASCII (American Standard Code for Information Interchange)**
  - 7-bit code
  - Chosen by other computer manufacturers
- **BCD and EBCDIC - based on punched card codes**
- **ASCII - based on telecommunications(Telex) codes**
- **Until recently - ASCII was the dominant character code outside the IBM mainframe world.**

# Character Codes

- **Unicode**

- 16-bit system that can encode the characters of every language in the world
- Java programming language and some operating systems now use Unicode as their default character code

- **Unicode codespace - Divided into six parts**

- First part is for Western alphabet codes
  - Includes ASCII
  - Contains characters for English, Greek, and Russian

# Character Codes

- **Unicode codespace allocation is shown at the right.**
- **Lowest-numbered Unicode characters comprise the ASCII code**
- **Highest character types provide for user-defined characters**

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE



# Logical Operators

- Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values
  - Formal logic: “true” and “false”
  - Digital systems:
    - “on” and “off”
    - 1 and 0
    - “high” and “low”
- Boolean expressions are created by performing operations on Boolean variables
  - Common Boolean operators include:
    - AND, OR, and NOT

# Boolean Algebra

- Boolean operator described with truth table
- Truth table for AND and OR are shown at right
- AND operator is the Boolean product
- OR operator is the Boolean sum

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

# Boolean Algebra

- Truth table for the Boolean NOT operator is shown at the right
- NOT operation is often designated by an overbar
  - sometimes indicated by a prime mark ( ' )
  - sometimes an “elbow” ( $\neg$ ).

NOT x	
x	$\bar{x}$
0	1
1	0

# Boolean Algebra

- A Boolean function has:
  - At least one Boolean variable,
  - At least one Boolean operator, and
  - At least one input from the set {0,1}
- Output is also a member of the set {0,1}

Another reason why the binary numbering system is so handy in digital systems.

# Boolean Algebra

- Truth table for the Boolean function:

$$F(x, y, z) = x\bar{z} + y$$

is shown at the right.

- Extra (shaded) columns hold evaluations of subparts of function

$$F(x, y, z) = x\bar{z} + y$$

x	y	z	$\bar{z}$	$x\bar{z}$	$x\bar{z}+y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

# Boolean Algebra

- Boolean operations have rules of precedence
- NOT operator has highest priority
- AND
- OR
- This determined the (shaded) function subparts in table

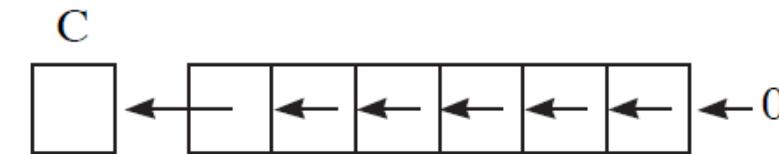
$$F(x, y, z) = x\bar{z} + y$$

x	y	z	$\bar{z}$	$x\bar{z}$	$x\bar{z} + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

# Arithmetic Operators

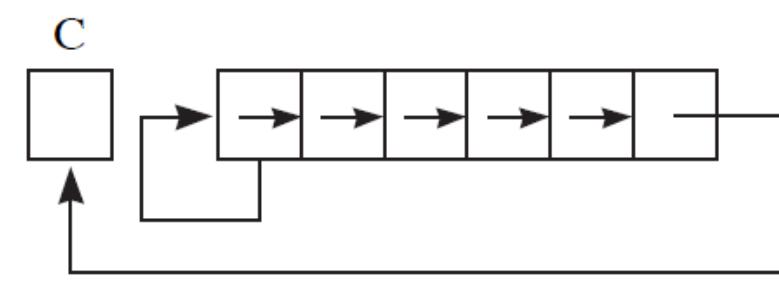
- **ASL – arithmetic shift left**

- Effectively multiplies by 2 ( if no carry)



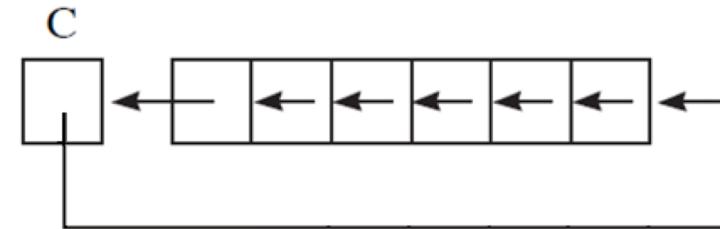
- **ASR – arithmetic shift right**

- Effectively divides by 2 (sign bit stays the same)

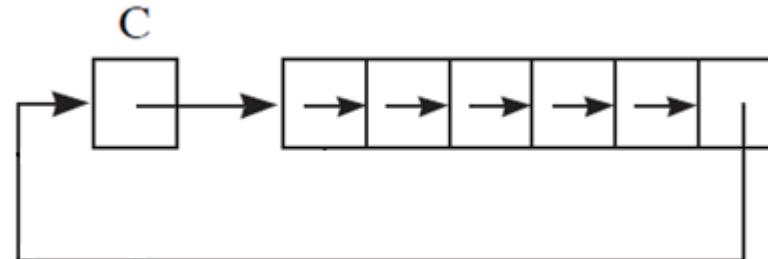


# Arithmetic Operators

- ROL – rotate left



- ROR – rotate right





# Floating-Point Representation

- **Signed magnitude, One's complement & Two's complement**
  - For Integer values only
- **Scientific & business applications**
  - Need real number values
- **Floating-point representation solves this problem**

# Floating-Point Representation

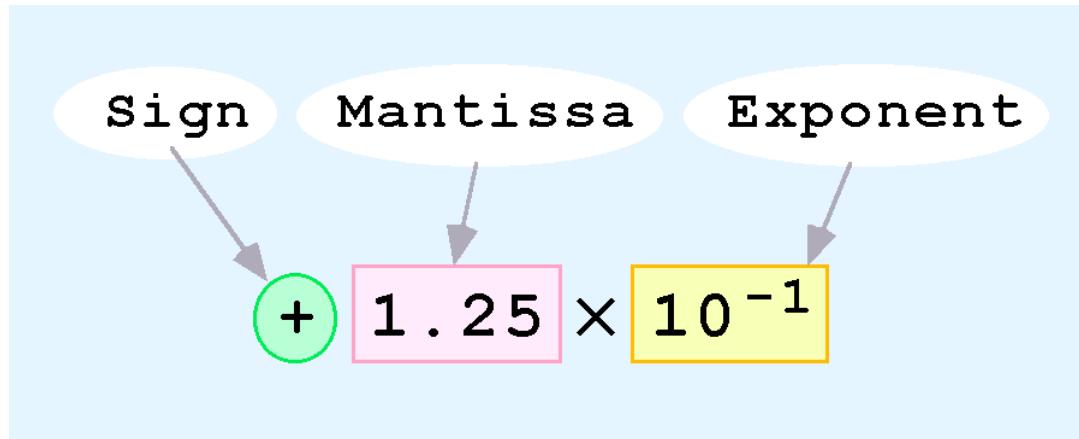
- ***Floating-point emulation***
  - Perform floating-point calculations using any integer format
  - Programs make it seem as if floating-point values are being used
- Today's computers have specialized hardware that performs floating-point arithmetic with no special programming required

## Floating-Point Representation

- **Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point**
  - For example:  $0.5 \times 0.25 = 0.125$
- **Scientific notation often used**
  - For example:  
 $0.125 = 1.25 \times 10^{-1}$   
 $5,000,000 = 5.0 \times 10^6$

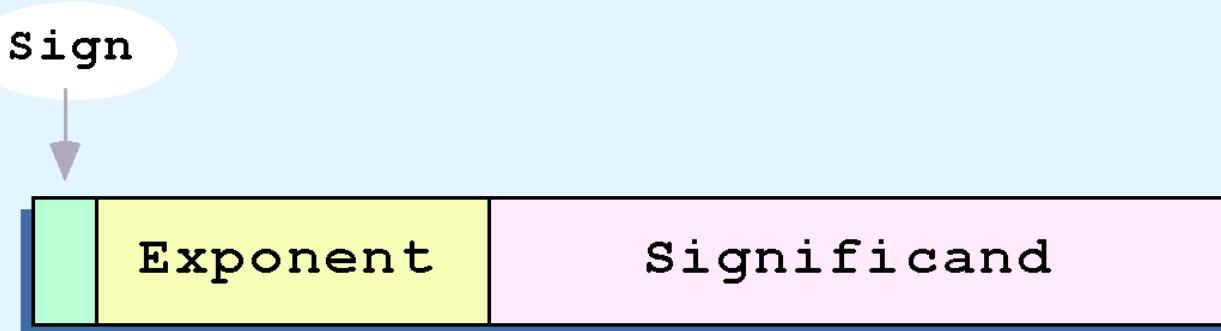
# Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



# Floating-Point Representation

- Components have three fixed-size fields:



- One-bit sign field - sign of the stored number
- Size of the exponent field - range of numbers
- Size of the significand - precision of the number

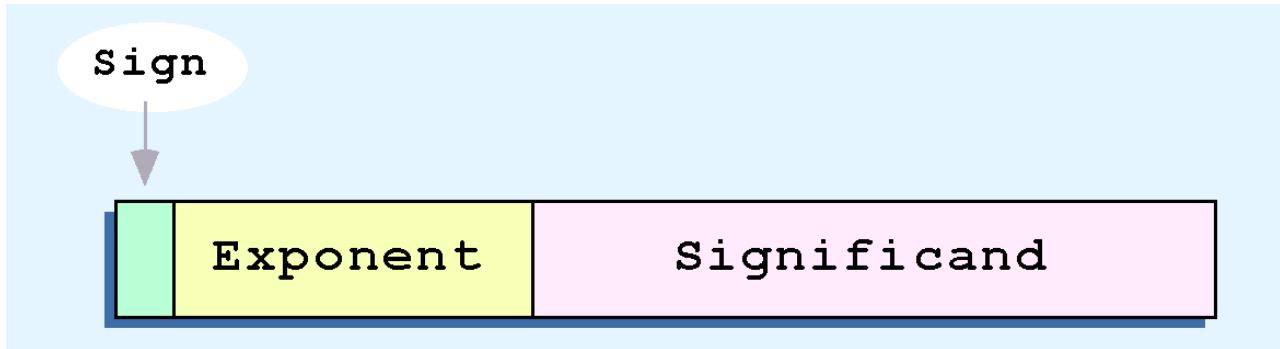
# Floating-Point Representation

- IEEE-754 *single precision* floating point standard
  - 8-bit exponent
  - 23-bit significand
- IEEE-754 *double precision* floating point standard
  - 11-bit exponent
  - 52-bit significand



For illustrative purposes, we will use a 14-bit model with a 5-bit exponent and an 8-bit significand.

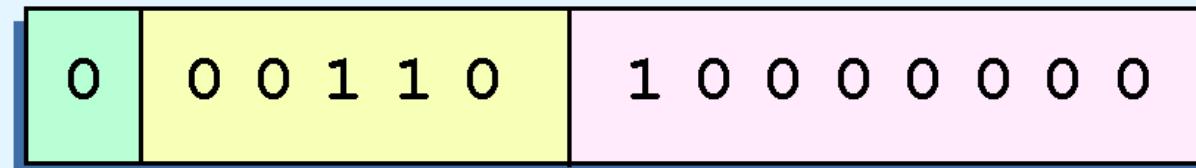
# Floating-Point Representation



- **Significand of a floating-point number**
  - preceded by an implied binary point
  - contains a fractional binary value
- **Exponent**
  - indicates power of 2 to which significand is raised

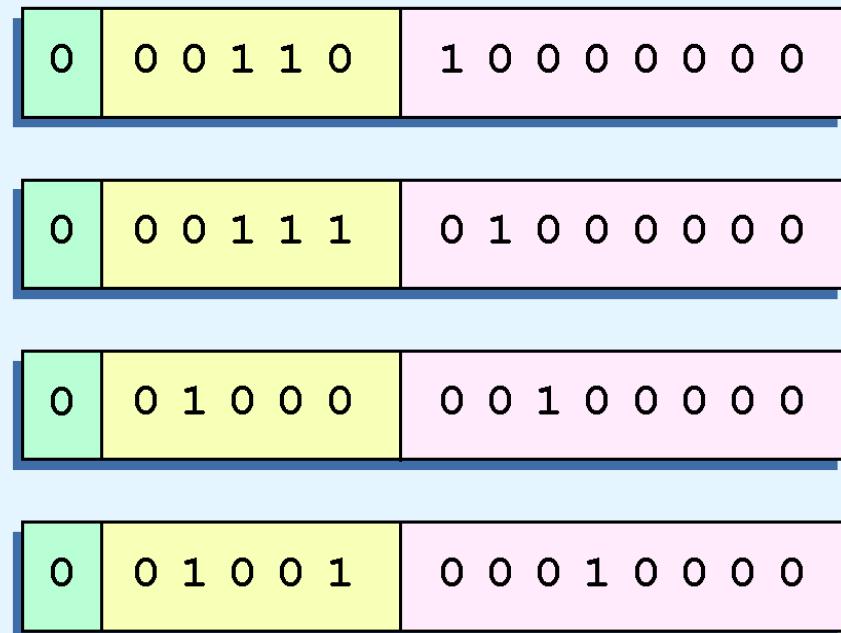
# Floating-Point Representation

- **Example:**
  - Express  $32_{10}$  in simplified 14-bit floating-point model
- Known: 32 is  $2^5$
- Scientific notation  $32 = 1.0 \times 2^5 = 0.1 \times 2^6$
- Sign is 0 (positive)
- Exponent becomes 110 (=  $6_{10}$ )
- Significand becomes 1



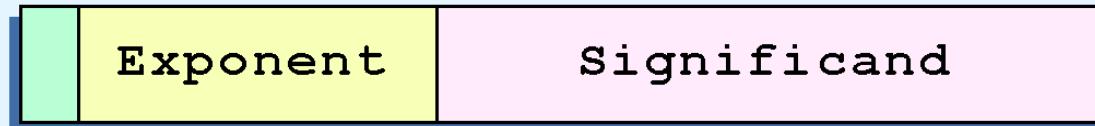
# Floating-Point Representation

- Illustrations at right
  - *all* equivalent representations for 32 using our simplified model
  - waste space
  - cause confusion



# Floating-Point Representation

Sign



- **No allowances for negative exponents**
  - no way to express  $0.5 (=2^{-1})$

These problems can be fixed by adding two rules to our basic model.

# Floating-Point Representation

- **New rule: significand must be normalized to have 1 to the left of the binary point**
  - 1 is assumed before the binary point
  - results in a unique pattern for each floating-point number
- **IEEE-754 standard**
  - 1 is assumed before the binary point
  - Increases precision of the representation by a power of two (Why?)

# Floating-Point Representation

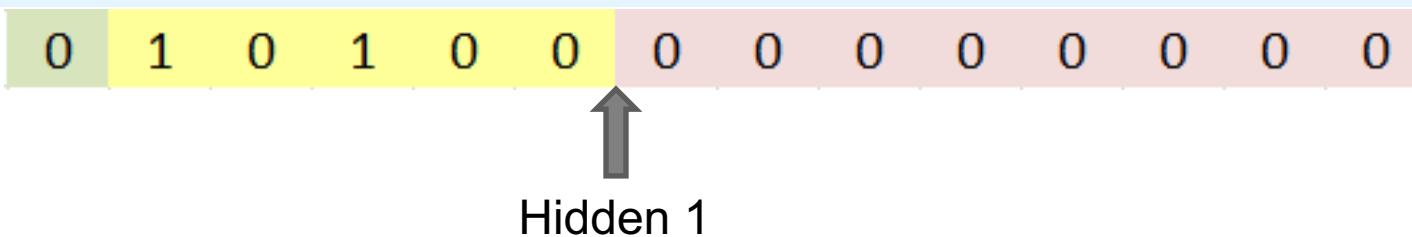
-15	00000
-14	00001
-13	00010
-12	00011
-11	00100
-10	00101
-9	00110
-8	00111
-7	01000
-6	01001
-5	01010
-4	01011
-3	01100
-2	01101
-1	01110
0	01111
1	10000
2	10001
3	10010
4	10011
5	10100
6	10101
7	10110
8	10111
9	11000
10	11001
11	11010
12	11011
13	11100
14	11101
15	11110

- ***Biased exponent***

- Provide negative exponents
- Bias is a number that is approximately midway in the range of values expressible by the exponent
- Subtract the bias from the value in the exponent to determine its true value
- New rule for our model
  - a 5-bit exponent
  - use 15 for our bias
  - excess-15 representation
  - exponent values less than 15 are negative, representing fractional numbers

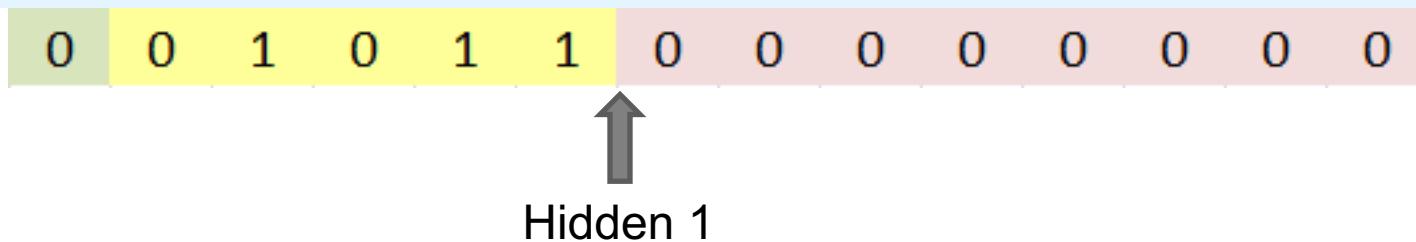
# Floating-Point Representation

- **Example:**
  - Express  $32_{10}$  in revised 14-bit floating-point model
- **Known:**  $32 = 1.0 \times 2^5$
- **To use excess 15 biased exponent:**
  - Add 15 to 5, giving  $20_{10}$  ( $=10100_2$ )
- **Graphically:**



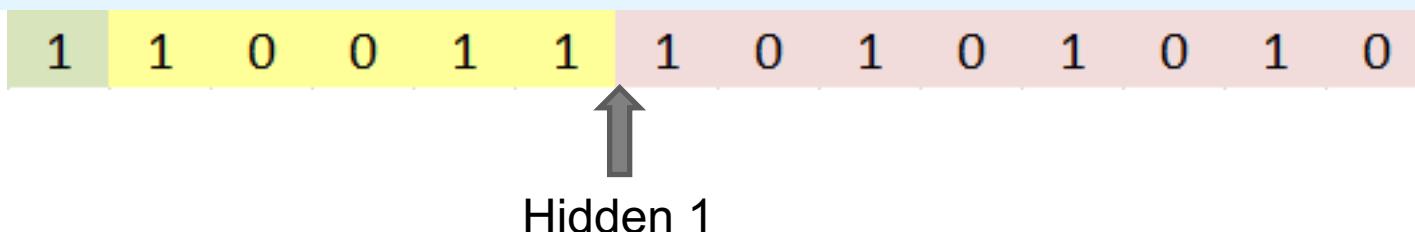
# Floating-Point Representation

- **Example:**
  - Express  $0.0625_{10}$  in revised 14-bit floating-point model
- **Known:**  $0.0625$  is  $2^{-4}$ 
  - $0.0625 = 1.0 \times 2^{-4}$
- **To use excess 15 biased exponent:**
  - Add 15 to  $-4$ , giving  $11_{10}$  ( $=01011_2$ )
- **Graphically:**



# Floating-Point Representation

- **Example:**
  - Express  $-26.625_{10}$  in revised 14-bit floating-point model
- **Calculate:**  $26.625_{10} = 11010.101_2$
- **Normalizing:**  $26.625_{10} = 1.1010101 \times 2^4$
- **To use our excess 15 biased exponent**
  - add 15 to 4, giving  $19_{10}$  ( $=10011_2$ ).
  - Add 1 in the sign bit
- **Graphically**



# Floating-Point Representation

- **IEEE-754 single precision floating point standard**
  - Bias of 127 over its 8-bit exponent
  - An exponent of 255 indicates a special value:
    - If significand is zero = value is  $\pm$  infinity
    - If significand is nonzero = value is NaN
      - “not a number”
- **IEEE-754 double precision standard**
  - Bias of 1023 over its 11-bit exponent
  - An exponent of 2047 indicates a special value
    - Like above...

# Floating-Point Representation

- **14-bit model and IEEE-754 floating point standard allow two representations for zero**
  - All zeros in exponent and significand
  - Sign bit can be either 0 or 1
- **Avoid testing a floating-point value for equality to zero**
  - Negative zero does not equal positive zero.

# Floating-Point Representation

- **Floating-point representation - model must be finite**
  - Limited to a number of bits
- **Real number system – infinite**
  - Models are an approximation of a real value
- **Every model breaks down - introducing errors**
- **A greater number of bits in the model reduces errors**
  - Never totally eliminate them
  - Limited by memory resources

# Floating-Point Representation

- Reduce error
  - or-
- Aware of the possible magnitude of error in calculations
- Errors can compound through repetitive arithmetic operations
- Example: our 14-bit model cannot exactly represent decimal value 128.5
- In binary it is 9 bits wide:  $10000000.1_2 = 128.5_{10}$

## Floating-Point Representation

- **128.5<sub>10</sub> in our 14-bit model - lose the low-order bit giving a relative error of:**

$$\frac{128.5 - 128}{128.5} \approx 0.39\%$$

- **A procedure that repetitively added 0.5 to 128.5**
  - Error of nearly 2% after only four iterations

# Floating-Point Representation

- **Floating-point errors reduced**
  - Use operands that are similar in magnitude
- **For problem adding 0.5 to 128.5**
  - Better to iteratively add 0.5 to itself
  - Then add 128.5 to this sum
- **Error was caused by loss of the low-order bit**
- **Loss of the high-order bit is more problematic**

# Floating-Point Representation

- **Floating-point overflow and underflow**
  - Cause programs to crash
- **Overflow**
  - There is no room to store high-order bits resulting from a calculation
- **Underflow**
  - When a value is too small to store, possibly resulting in division by zero

*Experienced programmers know that it's better for a program to crash than to have it produce incorrect, but plausible, results.*

# Floating-Point Representation

- **Floating-point numbers**
  - Understand the terms *range, precision, and accuracy*
- **Range**
  - The difference between the largest and smallest values that it can express
- **Accuracy**
  - how closely a numeric representation approximates a true value
- **Precision**
  - indicates how much information we have about a value
  - significant digits

# Floating-Point Representation

- Many problems with floating point numbers
- Usually- Greater precision leads to better accuracy
  - Not always true
  - Example, 3.1333 is a value of pi that is accurate to two digits, but has 5 digits of precision
- Truncated bits may give different results for commutative or distributive calculations
- This means that we cannot assume:  
 $(a + b) + c = a + (b + c)$  or  
 $a*(b + c) = ab + ac$

# Floating-Point Representation

- **Floating-point calculations at HOL 6 are easy**
  - Hidden problems
- **Floating-point calculations at Levels 3 &5 are more difficult**
  - Require careful planning and strict models
- **Floating-point numbers are an approximation**



## Chapter 3 Objectives

- **Fundamentals of numerical data representation in digital computers**
- **Learn to convert between various radix systems**
- **Overflow and truncation errors can occur**
- **Fundamentals of floating-point representation**
- **Review popular character codes**

# Introduction

- ***bit*** - most basic unit of information in a computer
  - “on” or “off”
  - Sometimes “high” or “low” voltage
- ***byte*** is a group of eight bits
  - A byte is the smallest *addressable* unit
  - “*addressable*” means it can be retrieved from its location in memory.

# Introduction

- ***word* is a contiguous group of bytes**
  - Words can be any number of bits or bytes
  - Word sizes of 16, 32, or 64 bits are most common.
  - Word-addressable system, a word is the smallest addressable unit of storage.
- ***nibble* (or *nybble*) is a group of four bits**
  - Bytes consist of two nibbles:
    - a high-order nibble
    - a low-order nibble

# Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2
  - The binary system is also called the base-2 system
  - Our decimal system is the base-10 system
  - Any integer quantity can be represented exactly using any base (or *radix*)

# Positional Numbering Systems

- The decimal number 947 in powers of 10 is:

$$9 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$$

- The decimal number 5836.47 in powers of 10 is:

$$5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 4 \times 10^{-1} + 7 \times 10^{-2}$$

# Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$\begin{aligned} & 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ = & 16 + 8 + 0 + 0 + 1 = 25 \end{aligned}$$

- The base is denoted by a subscript when not 10
  - Sometimes, the subscript 10 is added for emphasis:

$$11001_2 = 25_{10}$$

# Decimal to Radix Conversions

- Every integer value can be represented exactly using any radix system
  
- Two methods for radix conversion:
  - Subtraction method
  - Division remainder method
- Subtraction method
  - more intuitive
  - cumbersome

## Decimal to Radix Conversions – Subtraction Method

- Convert decimal number 190 to base 3
  - $3^5 = 243$  too large**
  - Start with  $3^4 = 81$**
  - $81 \times 2 = 162$**
  - Write down 2 (81s)**
  - Subtract 162 from 190**
  - Leaving 28**

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \end{array} = 3^4 \times 2$$

$$3^5 = 243$$

$$3^4 = 81$$

$$3^3 = 27$$

$$3^2 = 9$$

$$3^1 = 3$$

$$3^0 = 1$$

## Decimal to Binary Conversions – Subtraction Method

- Converting 190 to base 3...
  - Next  $3^3 = 27$
  - $27 \times 1 = 27$
  - Write down 1 (27s)
  - Subtract 27
  - Leaving 1
  - Next  $3^2 = 9$  too large
  - $9 \times 0 = 0$
  - Write down 0 (nines)
  - Subtract 0
  - Leaving 1

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \\ - 27 \\ \hline 1 \\ - 0 \\ \hline 1 \end{array} = 3^4 \times 2 + 3^3 \times 1 + 3^2 \times 0 + 3^1 \times 1 + 3^0 \times 0$$

$$3^5 = 243 \quad 3^4 = 81 \quad 3^3 = 27 \quad 3^2 = 9 \quad 3^1 = 3 \quad 3^0 = 1$$

## Decimal to Radix Conversions – Subtraction Method

- Converting 190 to base 3...

- Next  $3^1 = 3$  too large
  - $3 \times 0 = 0$
  - Write down 0 (threes)
  - Subtract 0
  - Leaving 1
  - Last  $3^0 = 1$
  - $1 \times 1 = 0$
  - Write down 1 (ones)
  - Result, reading top to bottom

$$190_{10} = 21001_3$$

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \\ - 27 \\ \hline 1 \\ - 0 \\ \hline 1 \\ - 0 \\ \hline 1 \\ - 1 \\ \hline 0 \end{array} = 3^4 \times 2 \\ = 3^3 \times 1 \\ = 3^2 \times 0 \\ = 3^1 \times 0 \\ = 3^0 \times 1$$

## Decimal to Radix Conversions - Division Method

- **Mechanical and easy**
- **Successive division by a base is equivalent to successive subtraction by powers of the base**
- **Apply division remainder method to convert 190 in decimal to base 3**

## Decimal to Radix Conversions – Division Method

- Converting 190 to base 3...
  - Divide the by the radix
  - Divide 190 by 3
  - Yields 63 with remainder of 1
  - Record quotient and remainder
  - Divide 63 by 3
  - Yields 21 and remainder is 0

$$\begin{array}{r} 3 \overline{)190} & 1 \\ 3 \overline{)63} & 0 \\ & 21 \end{array}$$

## Decimal to Radix Conversions – Division Method

- Converting 190 to base 3...
  - Continue until quotient is 0
  - Note final calculation:  
**3 divides into 2 zero times  
with a remainder of 2**
  - Result, reading bottom to top

$$190_{10} = 21001_3$$

A vertical division algorithm diagram. On the left, the divisor '3' is written above each of six quotient digits. To the right of the digits are the remainders: '1', '0', '0', '1', '2', and '0'. The first remainder '1' is above the first digit '1'. The second remainder '0' is above the second digit '0'. The third remainder '0' is above the third digit '0'. The fourth remainder '1' is above the fourth digit '1'. The fifth remainder '2' is above the fifth digit '2'. The sixth remainder '0' is below the last digit '0'. The quotient digits are aligned vertically under their respective remainders.

3	1	9	0	1
3	6	3	0	0
3	2	1	0	0
3	7		1	
3	2		2	
			0	

## Decimal to Binary Conversions - Fractions

- **Fractional values can be approximated in all base systems**
- **Fractions do not necessarily have exact representations under all radices**
- **$\frac{1}{2}$  is**
  - Exact representation in binary and decimal systems
  - Not exact in the ternary (base 3) numbering system

## Decimal to Radix Conversions - Fractions

- Fractional decimal values have nonzero digits to the right of the decimal point.
- Fractional values of other radix systems have nonzero digits to the right of the radix point.
- Numerals to the right of a radix point represent negative powers of the radix:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

$$\begin{aligned}0.11_2 &= 1 \times 2^{-1} + 1 \times 2^{-2} \\&= \frac{1}{2} + \frac{1}{4} \\&= 0.5 + 0.25 = 0.75\end{aligned}$$

# Decimal to Radix Conversions - Fractions

- **Two methods:**
  - subtraction method
  - multiplication method - easier
- **Subtraction method for fractions**
  - identical to the subtraction method for whole numbers
  - subtract negative powers of the radix
- **Start with the largest value first**
  - $n^{-1}$ , where  $n$  is our radix
  - work down using larger negative exponents

## Decimal to Radix Conversions – Fractions

### – Subtraction Method

- Example of subtraction method
  - Convert decimal **0.8125** to binary
  - Subtract negative powers of 2
  - Result:  
reading top to bottom  
 $0.8125_{10} = 0.1101_2$
  - Method works with any base - not just binary

$$\begin{array}{r} 0.8125 \\ - 0.5000 \\ \hline 0.3125 \\ - 0.2500 \\ \hline 0.0625 \\ - 0 \\ \hline 0.0625 \\ - 0.0625 \\ \hline 0 \end{array} = 2^{-1} \times 1 \quad 2^{-2} \times 1 \quad 2^{-3} \times 0 \quad 2^{-4} \times 1$$

# Decimal to Radix Conversions – Fractions

## – Multiplication Method

- **Example of multiplication method**

- Convert decimal **0.8125** to binary
- Multiply by the radix: **2**.
- Product carries into ones column
- Ignore value in ones column at when multiplying
- Continue multiplying fractional part by the radix until the fractional product is **0**
- Result:
  - reading top to bottom
$$0.8125_{10} = 0.1101_2$$
- Method also works with any base


$$\begin{array}{r} .8125 \\ \times 2 \\ \hline 1.6250 \end{array}$$
  
$$\begin{array}{r} .6250 \\ \times 2 \\ \hline 1.2500 \end{array}$$
  
$$\begin{array}{r} .2500 \\ \times 2 \\ \hline 0.5000 \end{array}$$
  
$$\begin{array}{r} .5000 \\ \times 2 \\ \hline 1.0000 \end{array}$$

# Data Representation in Computers

- Every integer value can be represented exactly using any radix system
- Fractional values can be approximated in all base systems
- All numbers and characters represented by sequences of 0 & 1
- Binary, Octal (base 8) and Hexadecimal
  - Common Bases

# **Chapter 2: High-Level Languages -Introduction**

**Fun with C++...really!**

# C++ as a High Order Language

- **Learning Objectives:**

- **Describe features and structure of HOL6**
- **Trace process from writing a program to execution**
- **Describe the attributes of variables & operators**
- **Read/Write a simple C++ program**

# Features of C++ as a HOL

- Machine Independent
- Syntax
  - Statements
  - Variables
  - Operators
  - Functions, Procedures, and Object Methods
- Flow of Control

# HOL Translation

- **Source code**
  - Universal text file format (e.g. ASCII)
  - Contains character-for-character whatever you type
  - Not understandable to the processor
  - Compiler translates source code to object code
- **Object code**
  - Binary language for a particular
  - Executable

# Variables

- **Attributes**

- Identifier (name)
- Type
- Value

- **Set-up**

- Declare = assign a name
- Define = assign a type
- Assign = set a value to a variable

- **Scope**

- Determined by location variable is declared
- Global variables: valid for the entire program
- Local variables: valid only for local subprogram or block

# Operators

- **Types of Operators**
  - Arithmetic, Relational, Boolean, and more
  - [Figure 2.9] [ Figure 2.11]
- 
- **Operator Interpretation**
  - Overloading, based on context / variable types
  - Ex: division operator [Figure 2.7]
- 
- **Expressions (formulas that compute values)**
  - Constructed from variables, operators and functions
  - Operator Precedence = priority of operators within an expression
  - [Http://www.cppreference.com/operator\\_precedence.html](http://www.cppreference.com/operator_precedence.html)

# C++ example

```
// Fig 2.4 - Stan Warford
// A nonsense program

#include <iostream>
using namespace std;

char ch;
int i;

int main () {
    cin >> ch >> i;
    i += 5;
    ch++;
    cout << ch << endl <<
        i << endl;
    return 0;
}
```

i = i + 5;      input:  
                  M 419  
  
ch = ch+1;      output:  
                  N  
                  424

- **Comments**

- **Compiler directives**

- **Variable declarations “nouns”**

- char ch, int i

•

- **Main program beginning**

- cin and cout

- refer to standard input (stdin) and output (stdout)

- **Operators = , <<, >> “verbs”**

- (C++ functions and Java methods are also like verbs)

- **Assignment statement**

- i += 5;

- ( ) { } ; “punctuation”

- (groupings & separations)

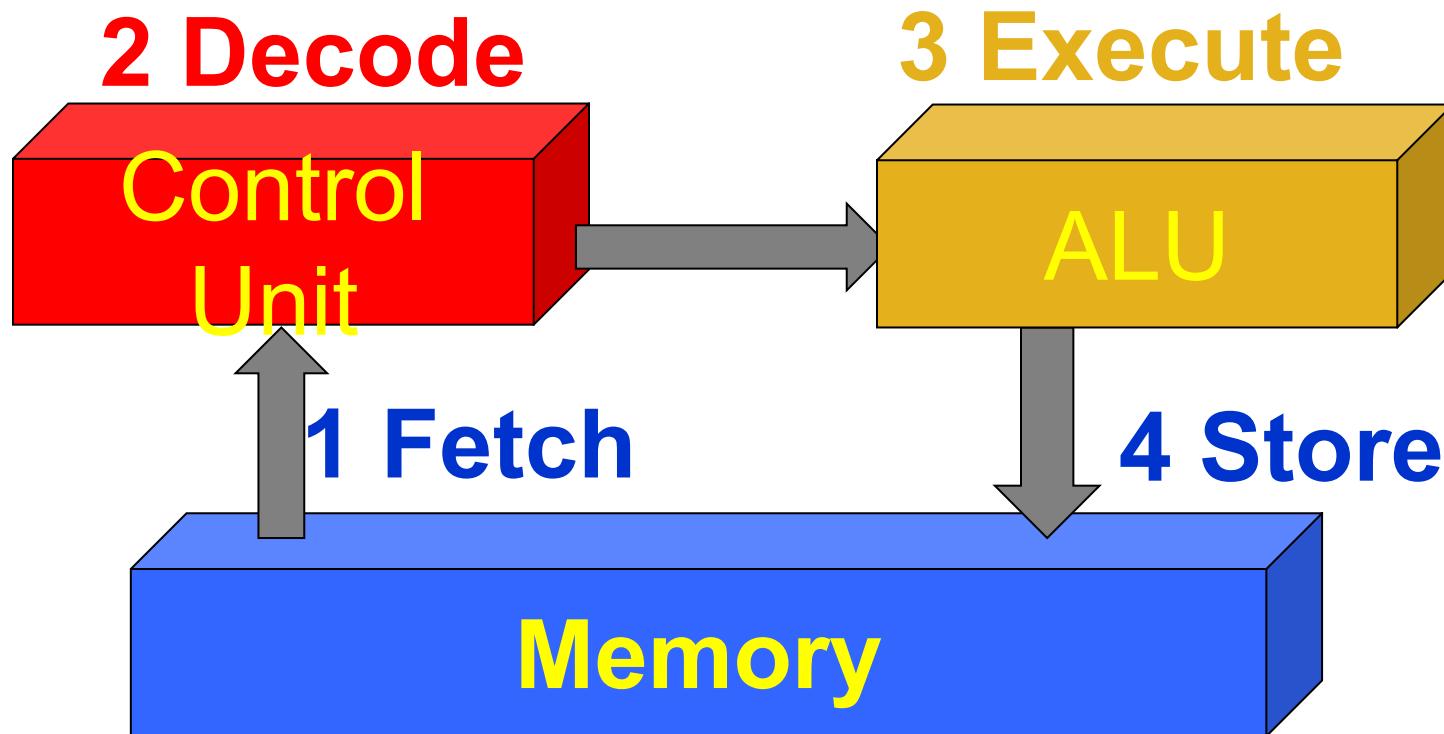
- **Program ending: return 0;**

# Running the program

- 3 things must happen before program is run:
  - **Compile** the high level instructions into machine level instructions
  - **Link** all of the files that are used in the program into one program
  - **Load** the result into memory
- Then the program in memory can run.

# Machine Level Execution

- What happens at the machine level when a program is executed?



# Machine Level Execution

- Two issues that are important:  
(more detail in chapters 3&4)
  - What do the instructions look like? How are they executed?
  - How are instructions/values stored in memory?

# Machine Level Execution

- What do the instructions look like? How are they executed?

(gdb) list 15,15

while (fahr <= upper) {

(gdb) disassem 0x10648 0x1066c

Dump of assembler code from 0x10648 to 0x1066c:

0x10648 <main+36>: ~~ld [ %fp + -24 ], %f3~~

0x1064c <main+40>: ~~fitos %f3, %f2~~

0x10650 <main+44>: ~~ld [ %fp + -32 ], %f3~~

0x10654 <main+48>: ~~fcmpes %f3, %f2~~

0x10658 <main+52>: ~~nop~~

0x1065c <main+56>: ~~fble 0x1066c <main+72>~~

0x10660 <main+60>: ~~nop~~

0x10664 <main+64>: ~~b 0x106f8 <main+212>~~

0x10668 <main+68>: ~~nop~~

End of assembler dump.

This is the C++ line of code

These are the corresponding assembly lines

This is the memory address of an instruction

# Machine Level Execution

- How is memory organized?
  - Instructions and variables can be stored anywhere.
  - Instructions are usually stored separately from variables, however.
  - Instructions **never** change.
  - Variables change.

# Machine Level Execution

- How is memory organized?
  - Two types of variables.
    - Variables that are declared before the main function are global to the program. These are **fixed** in memory.
    - Variables that are declared in a function. These are used, then discarded. The memory used is **not** fixed.

# Assignment in C++

- The “Course Materials” section on Blackboard has a C++ tutorial for you to work through to get you started prior to attempting your homework problems.
- The “Worked examples” section in Blackboard has several worked problems from the end of the chapter to use as guides in completing your assignment.
- Please do not hesitate to post any questions on the chapter discussion thread – you may also email me any source code you are having difficulty debugging so that I can help you.

# **Chapter 2: High-Level Languages**

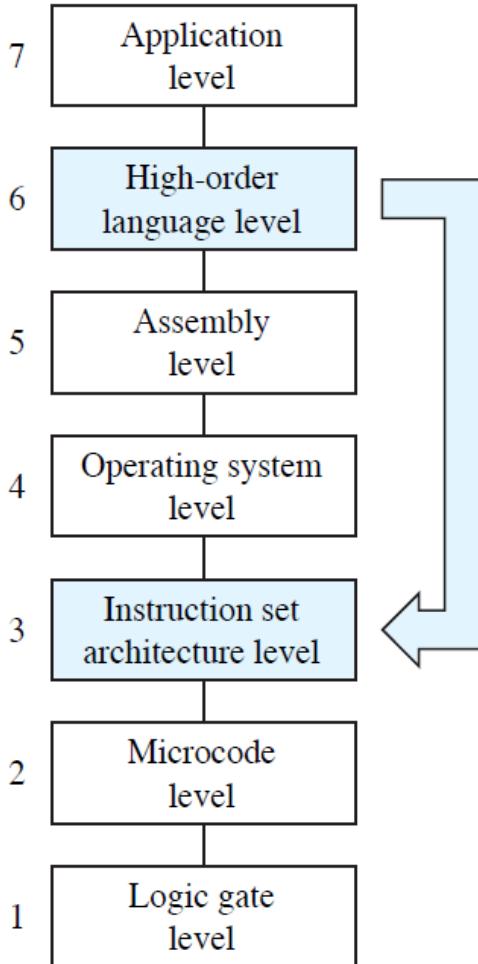
**Programming Tools**

**Compilers**

**Assemblers**

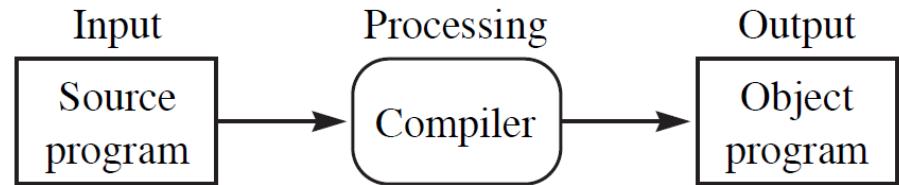
**Interpreters**

# Compilers



## Compilers

- **Translate Level 6 Languages into an equivalent program at a lower level of abstraction**



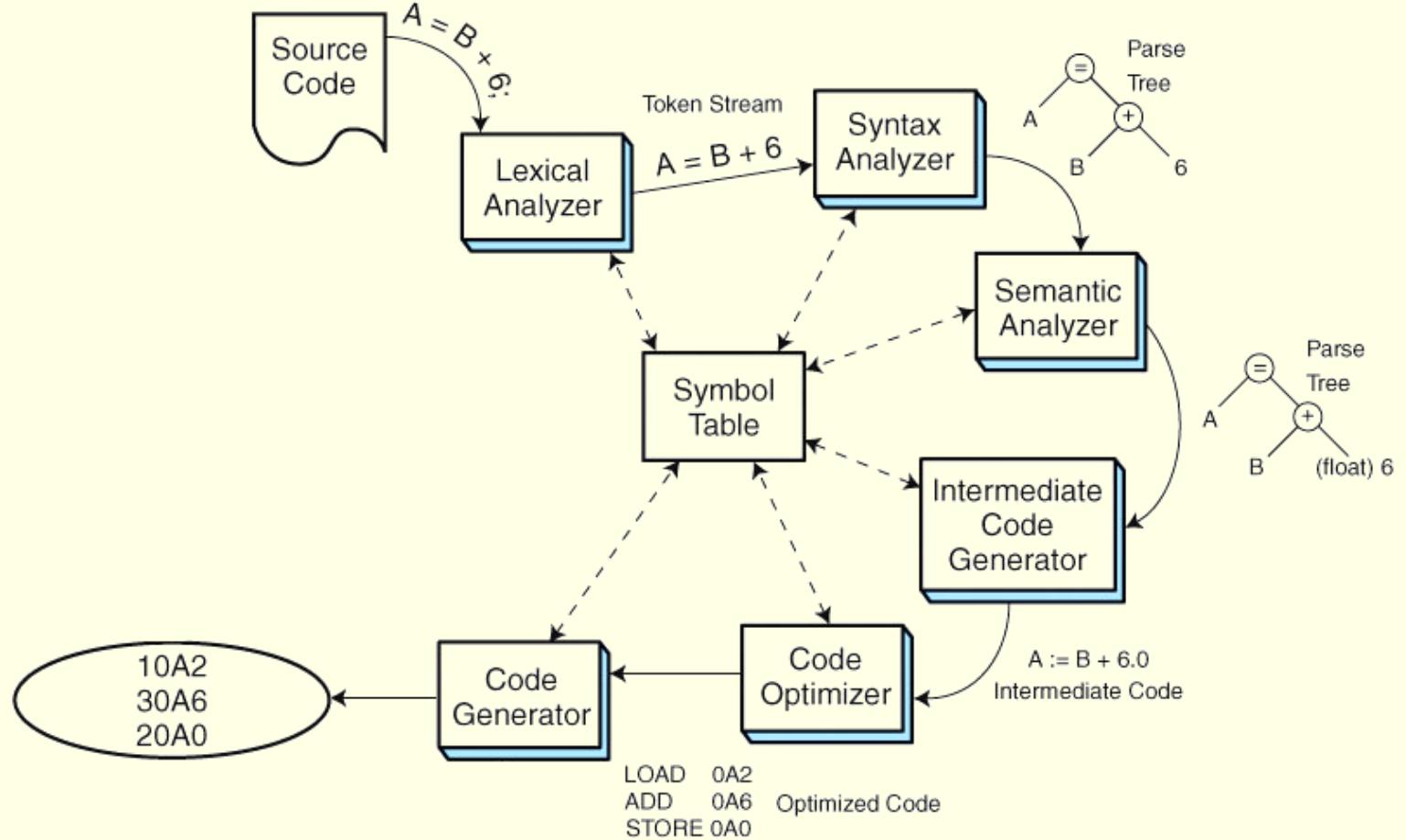
# Compilers

- **Most compilers do this translation in a six-phase process**
  - 1. Lexical analysis extracts tokens, e.g., reserved words and variables.**
  - 2. Syntax analysis (parsing) checks statement construction.**
  - 3. Semantic analysis checks data types and the validity of operators.**

# Compilers

- 4. Intermediate code generation creates *three address code* to facilitate optimization and translation
- 5. Optimization creates assembly code while taking into account architectural features that can make the code efficient.
- 6. Code generation creates binary code from the optimized assembly code.
- Compilers can be written for various platforms by rewriting only the last two phases

# Compilers



# Interpreted Languages

- Interpreters produce executable code from source code in real time
- Slower than compiled languages
- Less opportunity for error checking
- Very useful for teaching programming concepts
  - Feedback is nearly instantaneous
  - Performance is rarely a concern

# Assemblers

- **Assemblers**
  - Simplest of all programming tools
  - Translate mnemonic instructions to machine code
- **Most assemblers do this in two passes**
  - First pass partially assembles the code and builds the symbol table
  - Second pass completes the instructions by supplying values stored in the symbol table.

# Assemblers

- **Output of most assemblers is a stream of relocatable binary code.**
  - Operand addresses are relative to where the operating system chooses to load the program
  - **Absolute (nonrelocatable) code is most suitable for device and operating system control programming**
- **When loaded, special registers provide a base address**
- **Addresses are interpreted as offsets from the base address**

# Assemblers

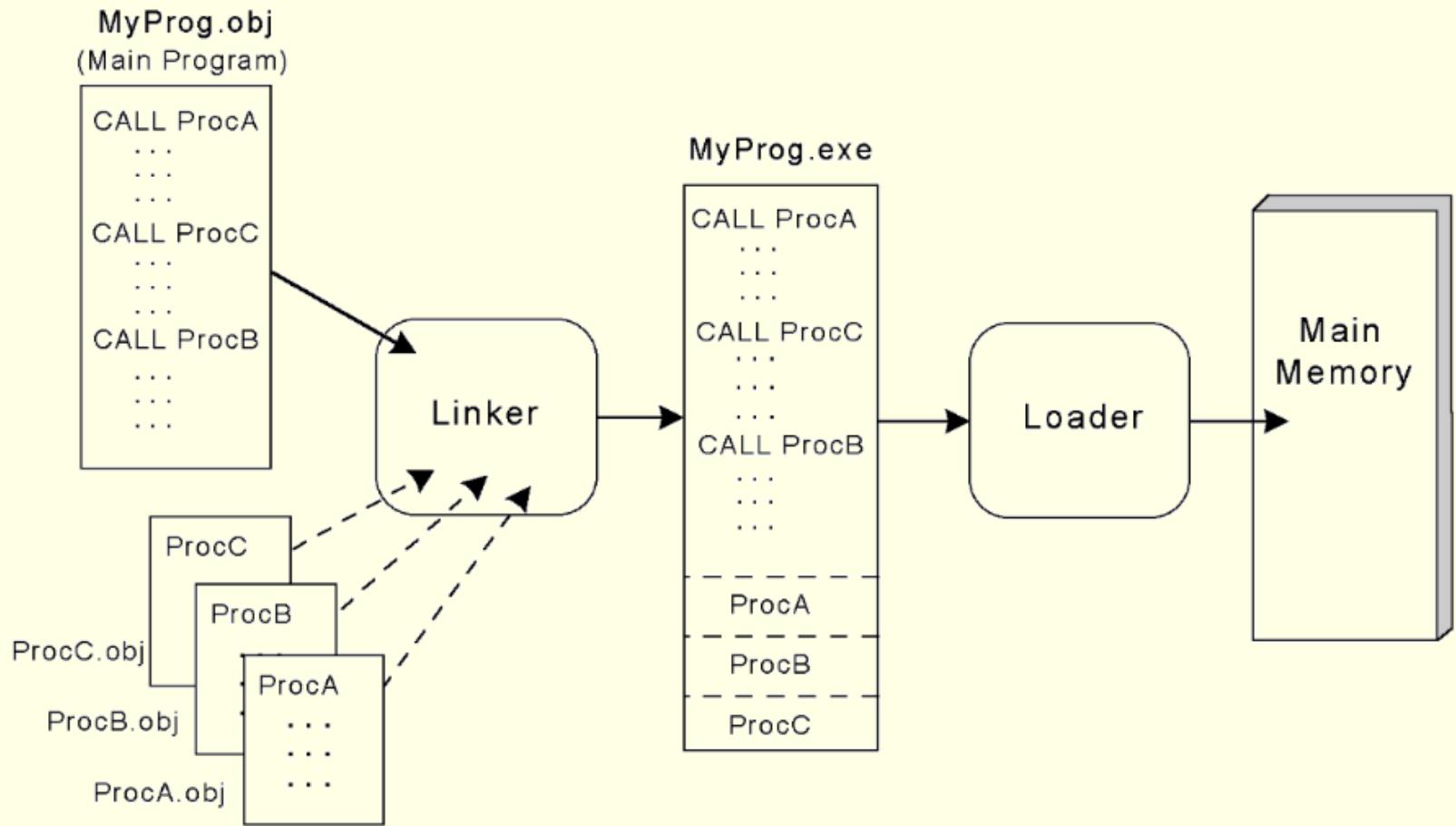
- **Binding is the process of assigning physical addresses to program**
- **Binding can occur at compile time, load time, or run time.**
  - **Compile time binding gives absolute code**
  - **Load time binding assigns physical addresses as the program is loaded into memory**
  - **Run time binding requires a base register to carry out the address mapping**

# Link Editors

- Link editors (Linker) take binary instructions and create an executable module
- Link editors concatenate various binary routines into a single executable
- Link editors perform two passes
  - First - create a symbol table
  - Second - resolve references to the values in the symbol table



# Linkers & Loaders



# Dynamic Linking

- Dynamic linking delays link editing until load time or run time
- External modules are loaded from *dynamic link libraries* (DLLs)
- Load time dynamic linking
  - Slower program loading
  - Calls DLLs faster
- Run time dynamic linking
  - External module is called first
  - Slower execution time
- Smaller program
- Programmer may not control DLL

# Programming Tools

- **High Level Languages**
  - Problem solving tools that are closer to how people think
  - Many layers from how the machine implements the solution

# **Chapter 2: C++ -Features**

- Call-by Value**
- Call-by Reference**
- Pointers**
- Structures**

# Call-By-Value

- Main stores actual parameter in a fixed location
- Function is passed parameter-
  - Parameter inside function is local
  - Stored on Run-time Stack
  - Parameter can be changed inside function
- Function ends-
  - Control is returned to main
  - Stack frame is deallocated
  - Changes to parameter inside function are gone
- Main – after function completion
  - Actual parameter remains original value

# Call-By-Reference

- Main stores actual parameter in a fixed location
- Function is passed parameter
  - Designated with & after the type in function definition
    - void swap (int& r, int& s) {}
  - Parameter inside function is pointer to actual parameter
  - Pointer is stored on Run-time Stack
  - Parameter can be changed inside function
- Function ends
  - Control is returned to main
  - Stack frame is deallocated
  - Changes to parameter exist in the original fixed location
- Main – after function completion
  - Actual parameter contains changes from the function



# Pointers

- **Global or Local**
- **Somewhat like a variable type**
  - Declared with \* before the variable name
    - `int *a, *b;`
  - Can be used in assignment statements
    - `a = b;` //this assigns the location b is pointing to - to the location a is pointing to
    - Point to actual values
      - `*a = *b;` //this assigns the value b is pointing to - to the value a is pointing to
- **Dynamic memory allocation on the heap**
- **Powerful – and dangerous**

# Structures

- **Consolidate variables into one data type**
  - Similar to arrays – groups of values
  - Variable can be different types
  - Variables are addressed using the dot notation  
very similar to object-oriented programming
    - student.name
- **Structure can be passed between functions and programs**
  - Eliminates need to pass individual variables or addresses

# Review in the Textbook

- **Review the examples for Call-by Value, Call-by-Reference, and Pointers**
- **Review the Figures depicting the Run-Time Stack for Call-by Value, Call-by-Reference, and Pointers**
- **HOL6 languages take care of memory allocation for you – Assembly language requires you to be intimate with the memory allocation**

# Chapter 2: High-Level Languages

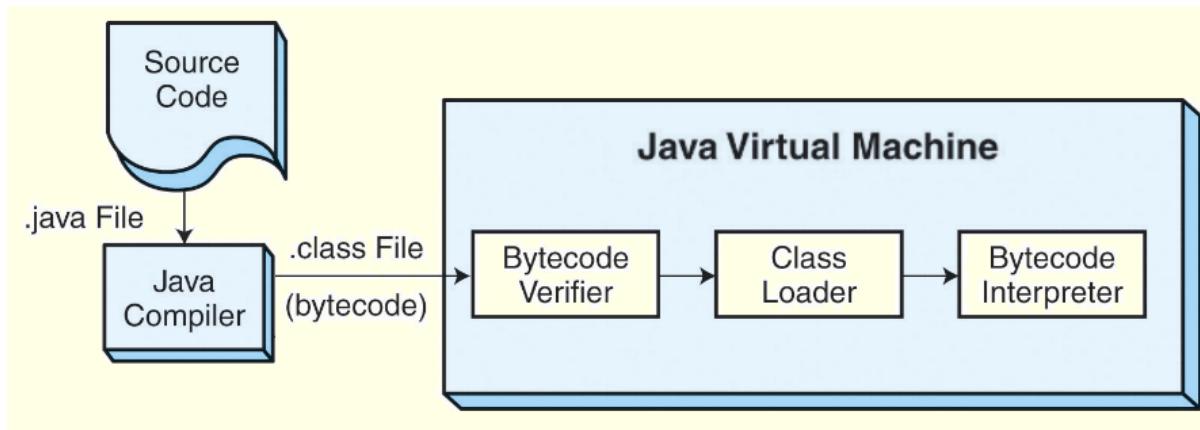
## Java

# Java: Compiled & Interpreted

- Java programs (*classes*) execute within the *Java Virtual Machine (JVM)*.
- Java can run on any platform with a virtual machine environment
- Java is both compiled and an interpreted
- Output of the compiler is an assembly-like intermediate code (*bytecode*)
- Bytecode is interpreted by the JVM

# Java: Compiled & Interpreted

- **JVM is a miniature operating system**
  - Loads programs
  - Links them
  - Manages program resources
  - Deallocates resources at program termination



- **JVM performance cannot match that of a traditional compiled language**

# Chapter 1: Computer Systems

- **Levels of Abstraction**
- **Hardware**
- **Software**
- **Database Systems**

# CSC376 COMPUTER ORGANIZATION AND ARCHITECTURE

## •Course Outline

- First - general introduction to computer architecture, data representation, and the von Neumann computer model.
- Second – C++ high level programming ,Pep/8 system, assembly programming and how high-level language features map onto assembly language.
- Then – Combinational and Sequential Circuits
- Finally - Introductions to systems topics covered in later courses.

# ARCHITECTURE & ORGANIZATION

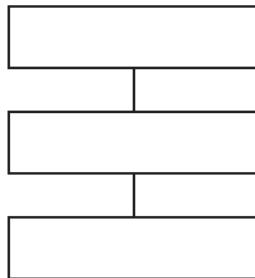
- **Architecture is those attributes visible to the programmer**
  - Instruction set,
  - Number of bits used for data representation
  - I/O mechanisms
  - Addressing techniques.
  - e.g. Is there a multiply instruction?
- **Organization is how features are implemented, typically hidden from the programmer**
  - Control signals
  - Interfaces
  - Memory technology.
  - e.g. Is there a hardware multiply unit or is it done by repeated addition?

# **ARCHITECTURE VS. ORGANIZATION**

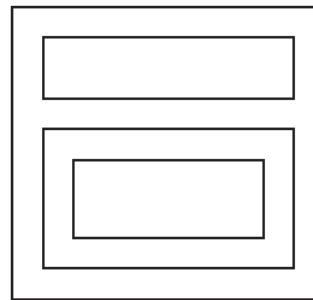
- All Intel x86 family share the same basic architecture
- The IBM System/370 family share the same basic architecture
- This gives code Backwards compatibility
  - i.e. older programs run on new processors
- Each generation increased in complexity
  - May be more efficient to start over with a new technology, e.g. RISC vs. CISC
- Organization is what differs between different versions

# ABSTRACTION

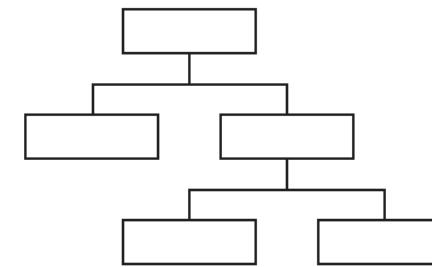
- Suppression of detail to show the essence of the matter
- An outline structure
- Division of responsibility through a chain of command
- Subdivision of a system into smaller subsystems



(a) A level diagram.



(b) A nesting diagram.



(c) A hierarchy, or tree, diagram.

# ABSTRACTION EXAMPLE

Henri Matisse



The Back I  
1909



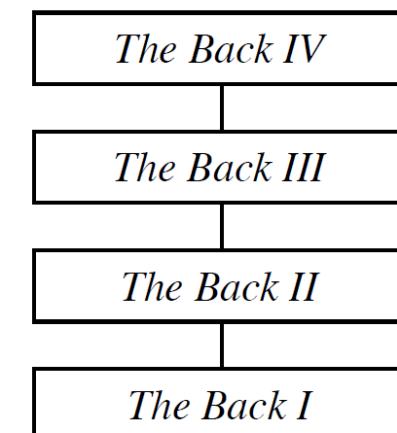
The Back II  
1913



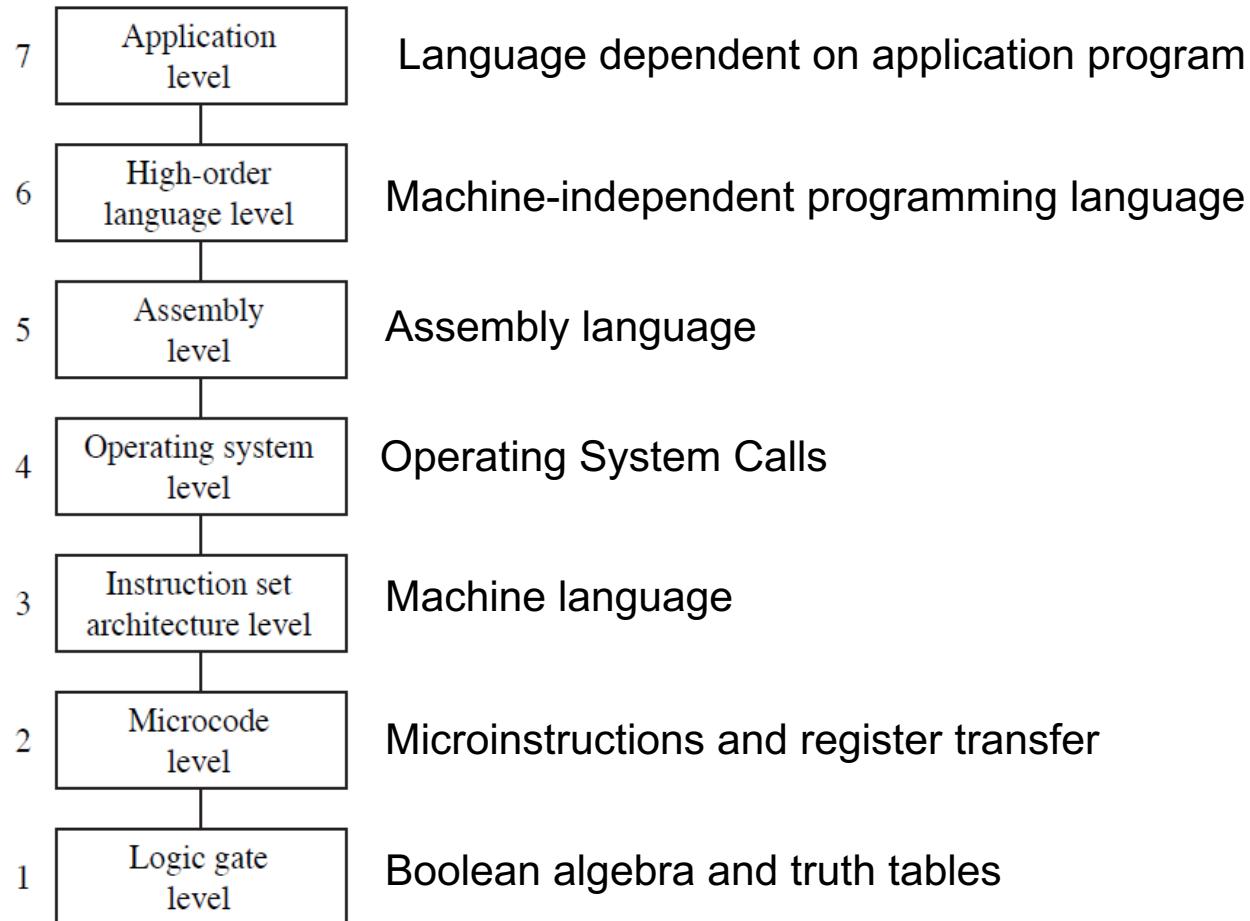
The Back III  
1917



The Back IV  
1930



# ABSTRACTION IN COMPUTER SYSTEMS

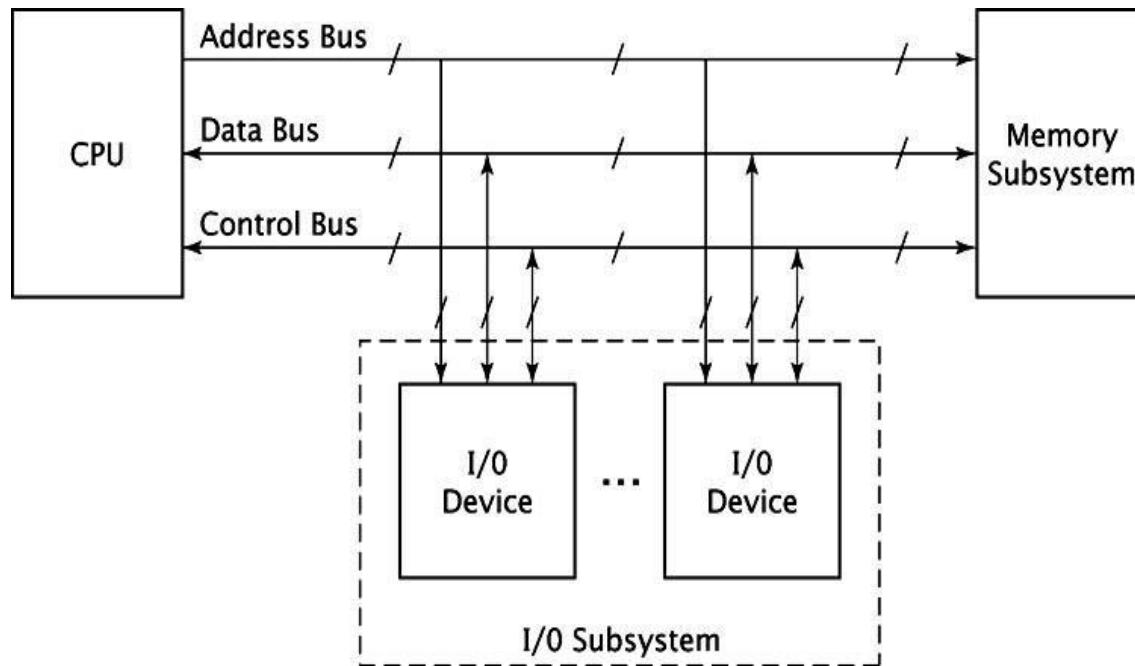


# THE MULTI-LEVEL MACHINE IDEA

- Level 7: Applications, e.g., Word, Excel, Video Games, Database Systems
- Level 6: High-level language, e.g., C, Java, which a compiler translates to assembly language
- Level 5: Assembly language (symbolic) translated by assembler into numeric form
- Level 4: Operating system. Some instructions in the output from the assembler are processed directly by level 3, others require action by the operating system. The Operating System itself is held in the form of machine code instructions.
- Level 3: “Machine Code” – binary instructions
- Level 2: Microinstructions - Very simple program that interprets the machine code program.
- Level 1: Circuits are activated according to the current microinstruction. This is where, finally, computation actually happens.

# Computer System Structure

- The overall system structure can be shown in the following diagram:



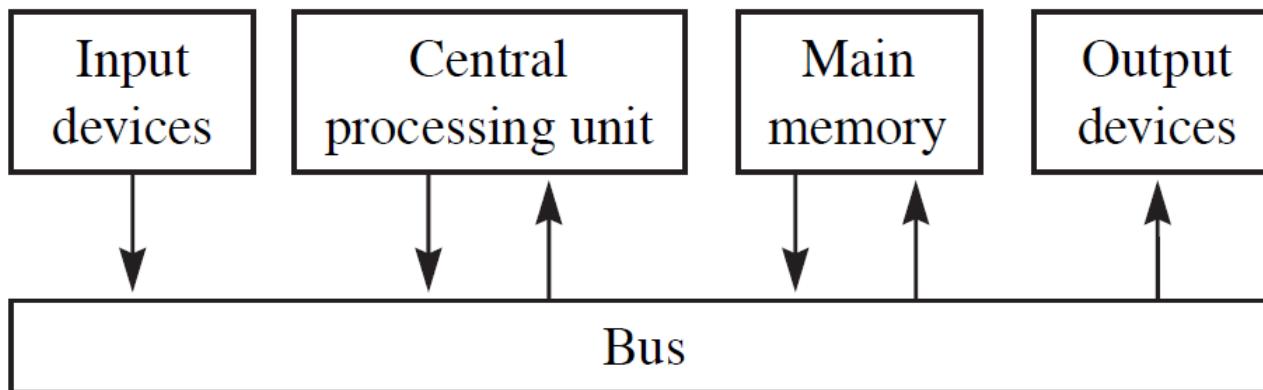
# HARDWARE

- The overall function of a computer system can be summed up with three activities:



# BASIC HARDWARE COMPONENTS

- Every computer system has four basic hardware components:



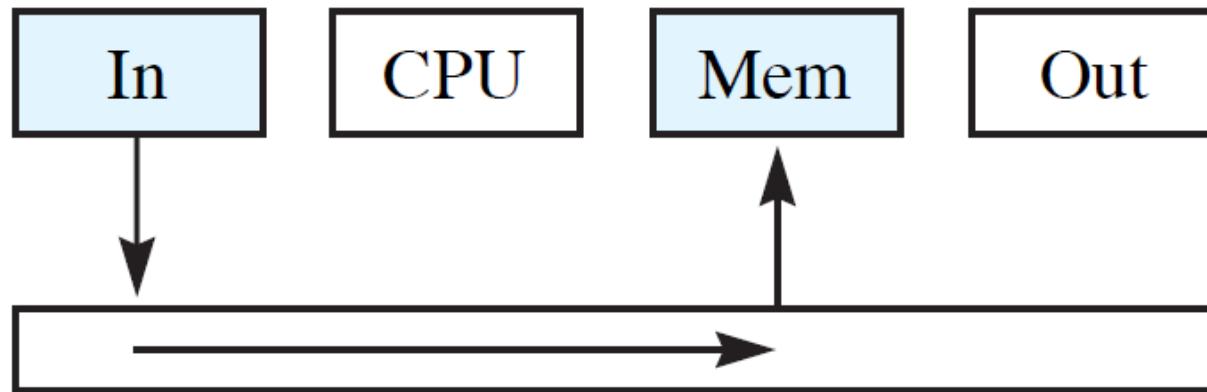
# BASIC HARDWARE

- **Central Processing Unit (CPU)**
  - Where instructions are fetched and executed
- **Memory system**
  - Where programs and data are stored on the main computer board
- **Input/Output (I/O) system**
  - Responsible for input and output data to and from the memory system.

# **INPUT DEVICES**

- Keyboards
- Disk drives
- USB Flash drives
- Solid State drives
- Magnetic tape drives
- Mouse devices
- Touchpads or Touchscreens
- Bar code readers

# INPUT DEVICE DATA PATH EXAMPLE



In      input devices  
CPU    central processing unit  
Mem    main memory  
Out    output devices

# HARDWARE LEVEL STORAGE

- At the lowest level, information is stored sequentially in Binary.
  - Each individual 0 or 1 is a binary digit: Bit
  - Eight bits is called a Byte



(a) Storage for an eight-bit byte.

0	1	1	0	1	0	1	1
---	---	---	---	---	---	---	---

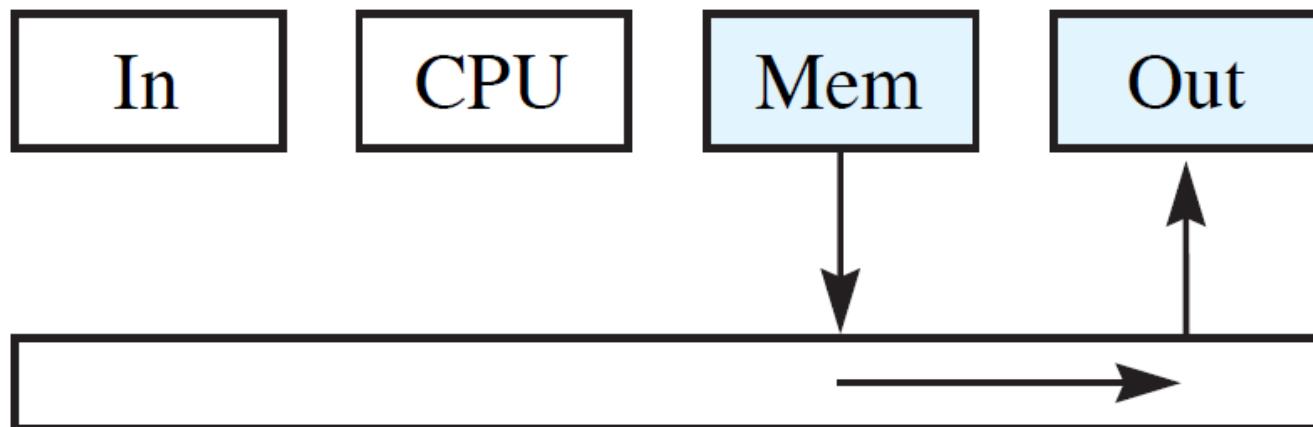
(b) The bit pattern for the character 'k.'

- What is the name of the common binary code that represents alphabetic characters?

# OUTPUT DEVICES

- Disk drives
- USB Flash drives
- Solid State drives
- Magnetic tape drives
- Screens
- Printers
- Sound Speakers

# OUTPUT DEVICE DATA PATH EXAMPLE



- Data flows from main memory onto the bus to the output device.

# I/O SUBSYSTEM

- A computer communicates with the outside world through its input/output (I/O) subsystem.
- I/O devices connect to the CPU through various interfaces.
- I/O can be memory-mapped-- where the I/O device behaves like main memory from the CPU's point of view.
- I/O can be instruction-based, where the CPU has a specialized I/O instruction set.

# MAIN MEMORY

- Main memory is volatile storage
  - Holds data and instructions for the CPU
  - RAM (random access memory)
- A memory system can be viewed as an array of memory locations each of which can store multiple-bit binary data
- Memory locations are selected by addresses for either read or write operations

# MEMORY ORGANIZATION

## Memory Types:

- **Read-Only Memory (ROM):**

- *Masked ROM* is manufactured with data inside; does not change.

- *Programmable ROM (PROM)* program-once devices.

- *Erasable PROM (EPROM)*: erased by holding the chip's "window" under ultraviolet light.

- *Electrically erasable PROM (EEPROM)*: erased by the programming device; portions can be selectively erased.

- *Flash EEPROM* is erasable in blocks (good for digital cameras).

- **Random Access Memory (RAM):**

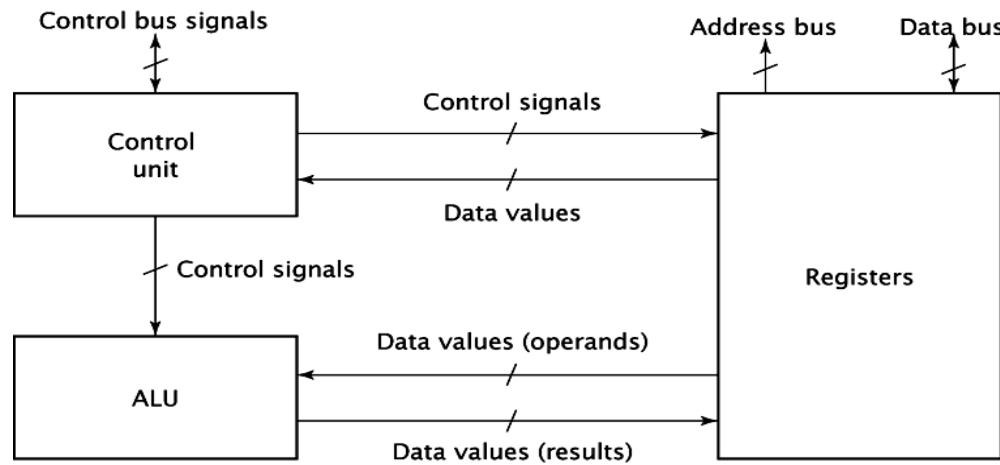
- *Dynamic RAM (DRAM)*: constructed from leaky capacitors and requires periodic recharge.

- *Static RAM (SRAM)*: does not need refreshing; more expensive.

# CPU ORGANIZATION

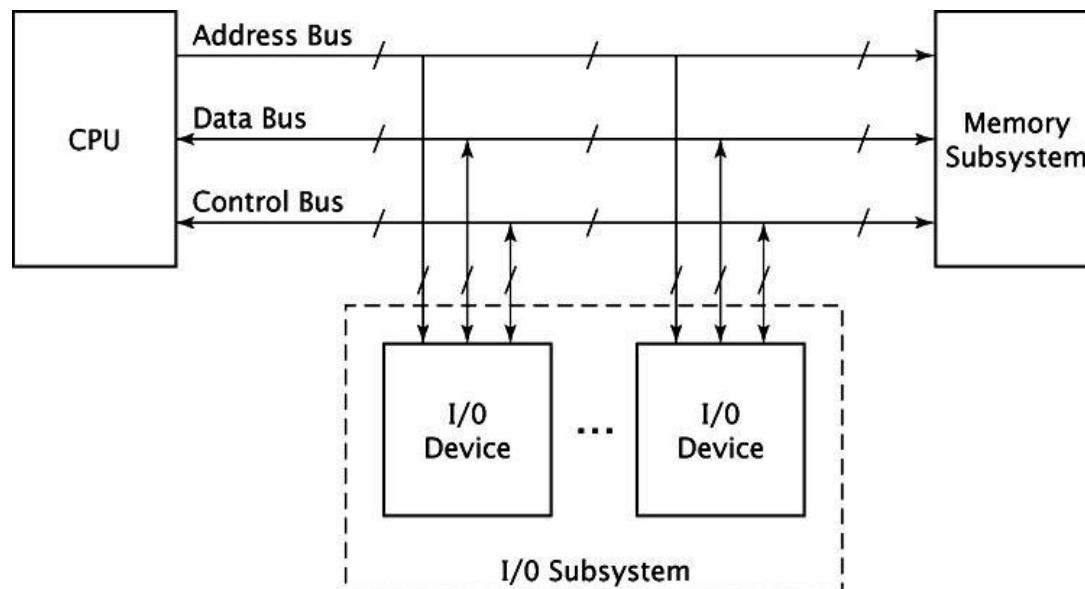
- Three basic components within the CPU:

- 1) Registers
- 2) Control unit
- 3) ALU



# SYSTEM BUSES

- The four major components are connected by buses
  - Address bus
  - Data bus
  - Control bus



# **SOFTWARE**

- **Algorithm:**

- Sequence of instructions that achieves a specified result in finite time
- Searching and sorting are common types of algorithms.

- **Program:**

- The implementation of an algorithm that can be executed by a computer



# **OPERATING SYSTEM: SYSTEM SOFTWARE**

- The most important program on a computer.
- It controls all essential functions, including:
  - File management
    - Documents
    - Programs
    - Data
  - Memory management
  - Processor management

# **TYPICAL OPERATING SYSTEM COMMANDS**

- **List the names of the files from the directory**
- **Delete a file from the disk**
- **Change the name of a file**
- **Print the contents of a file**
- **Execute an applications program**

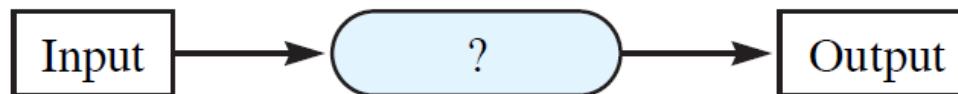
**For Windows 7, what is the limit of the number of characters that can be in a path + filename?**

# SOFTWARE ENGINEERING

- **Formalized process of analysis and Design of computer software systems**



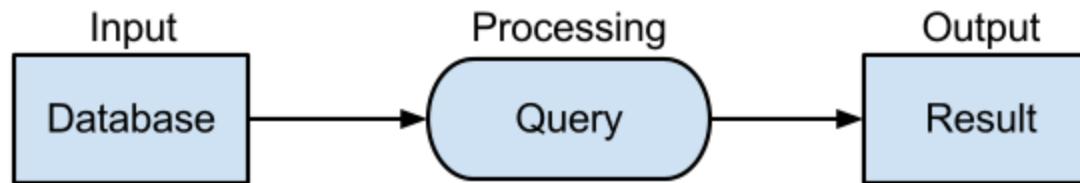
**(a) Analysis**—The input and processing are given.  
The output is to be determined.



**(b) Design**—The input and desired output are given. The processing is to be determined.

# DATABASE SYSTEMS

- DBMS - database management system
  - Most common application at Level App7
  - Stores data in records in a table format
  - Programs allow users to add, delete & modify records
  - SQL (Structured Query Language) Queries create output from data



# ABSTRACTION OF COURSE OUTLINE

