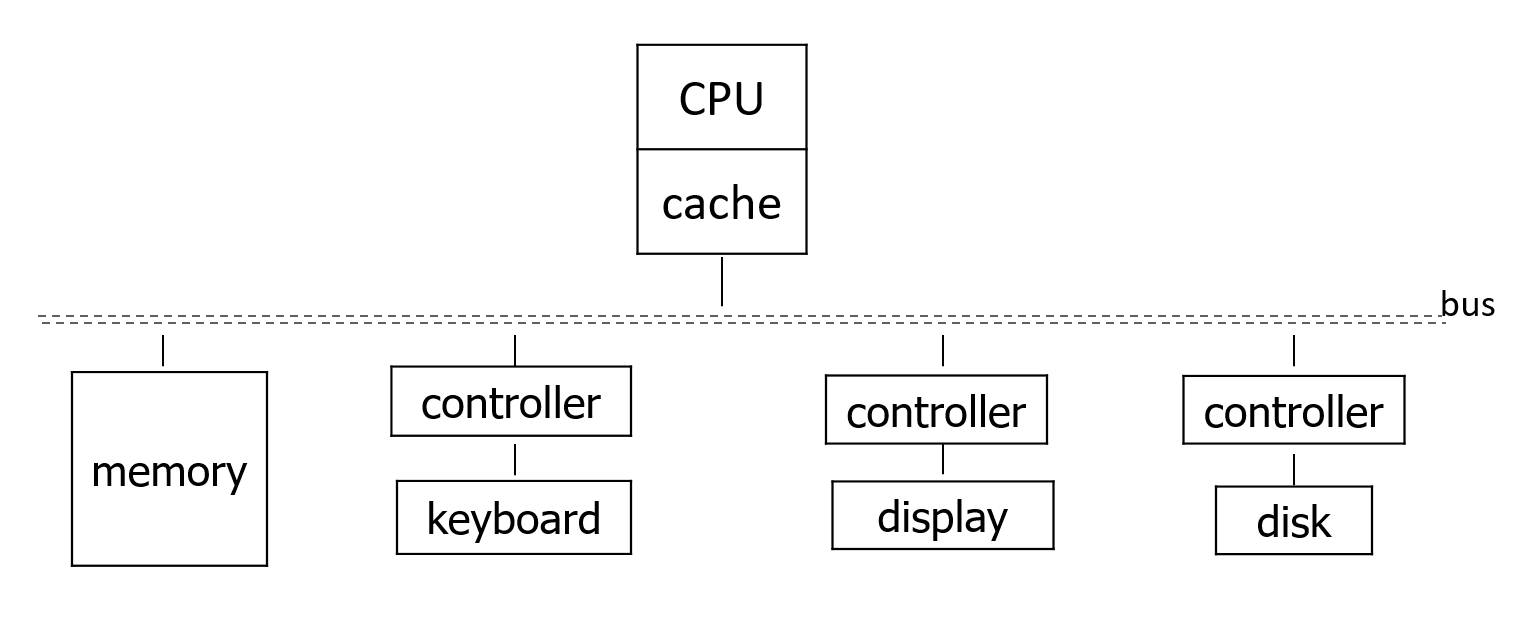
CPSC Test 1 Practice Session

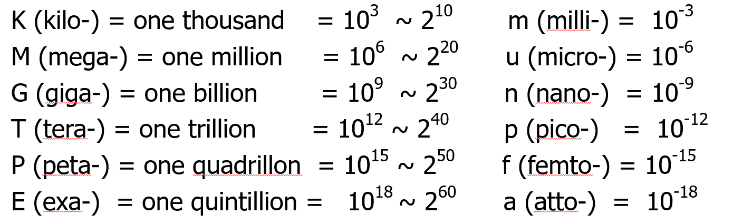
# Computer Systems



Components

* Input: Keyboard, mouse, scanner, etc
* Output: Display, printer, speakers, etc
* CPU, composed of two parts
  + Datapath (registers and function units)
  + Control Logic (sequencing of datapath actions)
  + Utilitize different instruction sets
  + Intel IA32, PowerPC, SPARC, ARM
* Memory: Multilevel hierarchy due to cost v speed tradeoffs
  + CPU registers
  + Cache (multi-level)
    - Static RAM, fast and expensive
  + Main memory
    - Dynamic RAM
  + Read-only memory
  + Long-term storage
    - Treated as I/O
    - Hard disks, CDs, DVDs

Prefixes for speed, time, and capacity

* main memory size is measured in powers of two, while speed is in powers of 10 (the capacity of most hard disks is measured in powers of ten)
* Powers of 2 sometimes referred to as “binary” instead of “byte” (Ki – Kibi)

Program Translation

* Symbolic languages
  + High-level (Python, C++)
  + Assembly (ARM)
    - Approx one-to-one correspondence with machine instructions
  + Machine instructions
    - Represented inside the computer in bit (0/1) patterns
* Compiler
  + Translates statements in a HLL into assembly code, performing optimizations and allocations
* Interpreter
  + Translates and executes at the same time
* Assembler
  + Takes assembly instructions and converts them into machine code for the computer to use, resulting in an object file

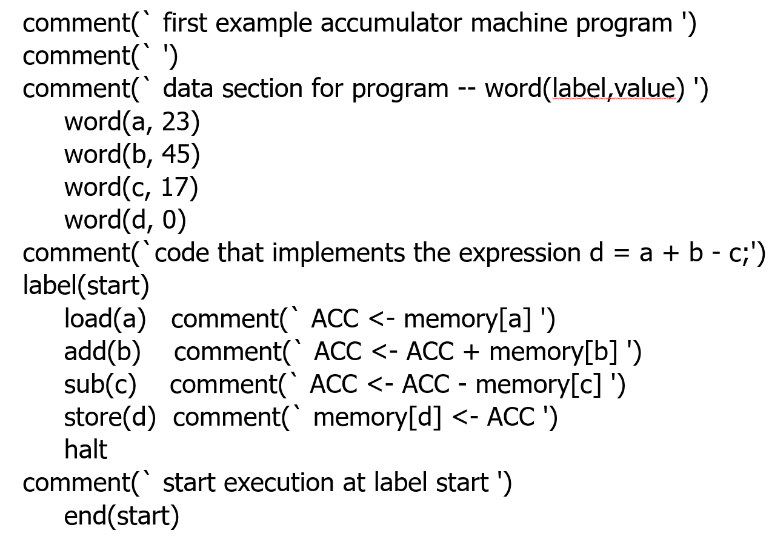
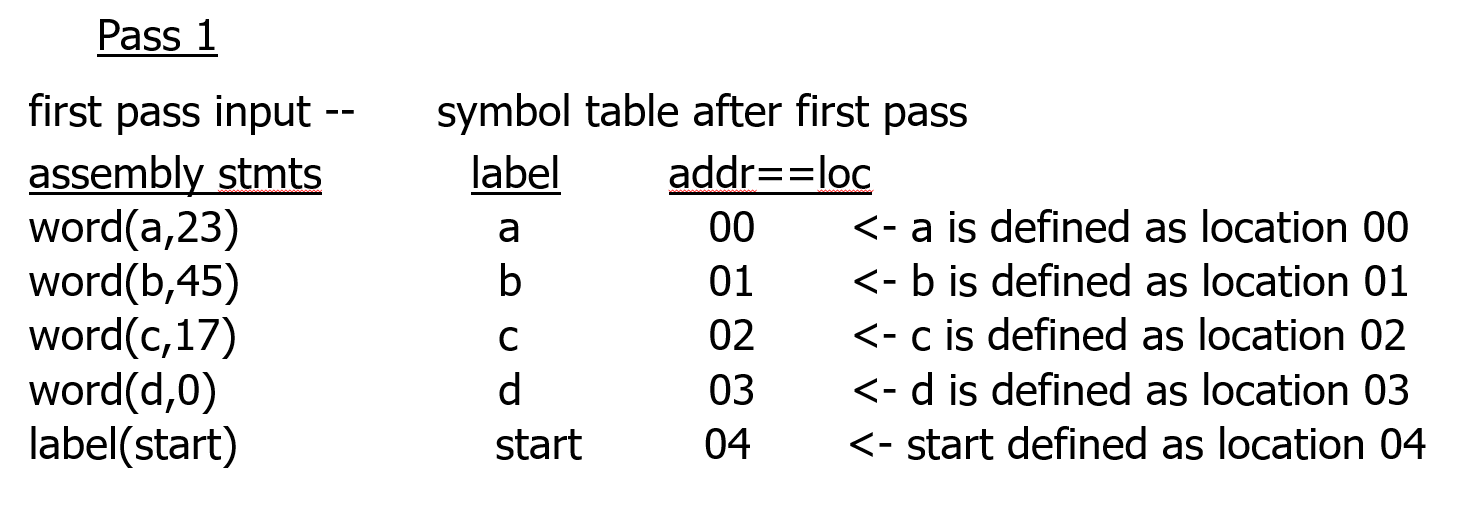
Assembler

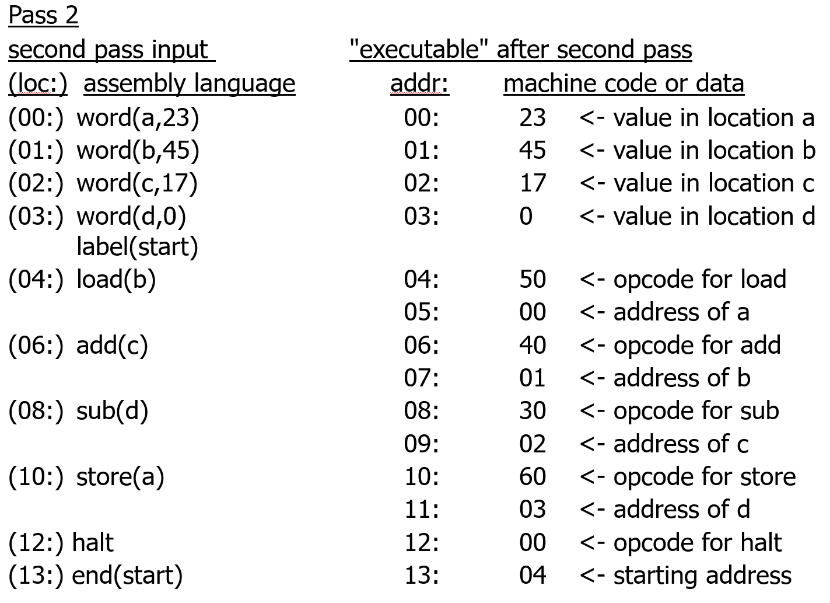
* Assembly Language
  + Statements called instructions
    - Operation code (opcode)
    - Operands (names of registers and/or info needed to generate address)
* Labels
  + Represent symbolic addresses of data and branch targets
  + Must be unique
* Have two-pass structure
  + Instructions can have forward or backward references to labels
  + Because of forward references, assemblers use two-pass assembly structure
  + Pass 1:
    - Increment location counter as it reads each statement
    - Builds symbol table by collecting label definitions with corresponding location counter values
  + Pass 2:
    - Using the symbol table, translate the assembly language statements into machine code

# Accumulator Machine

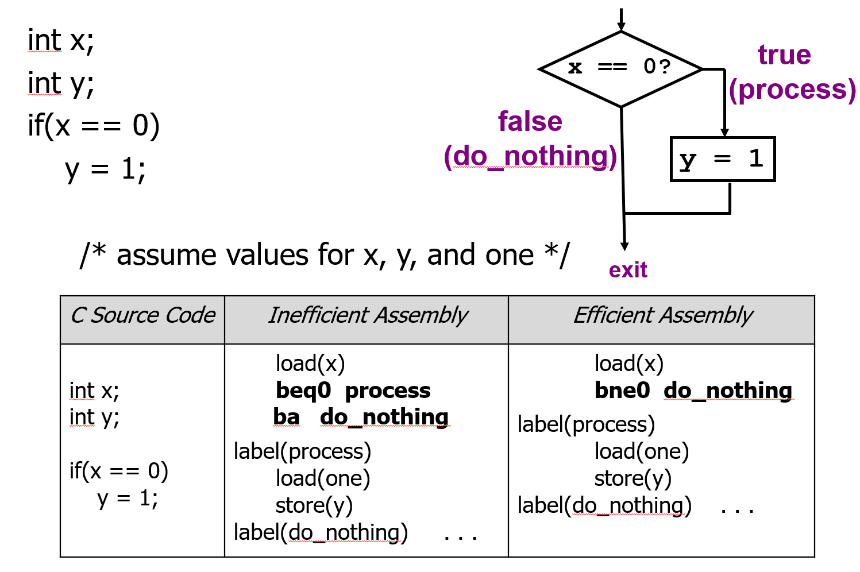
Using the accumulator machine architecture to demonstrate pass1 and pass2

* Has one register, the accumulator
* All other operands are in memory, and expressions require a sequence of load / operate / store instructions

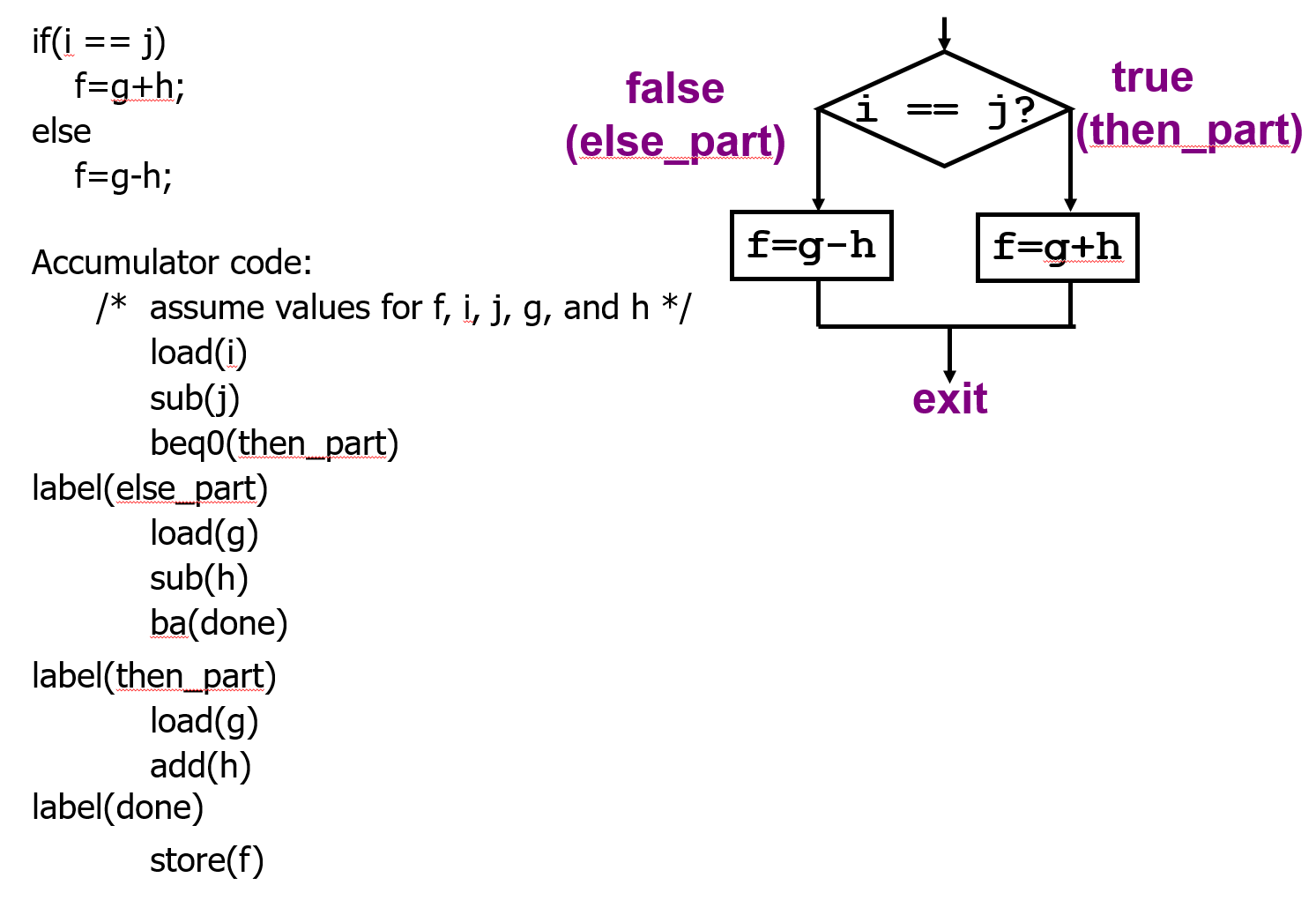
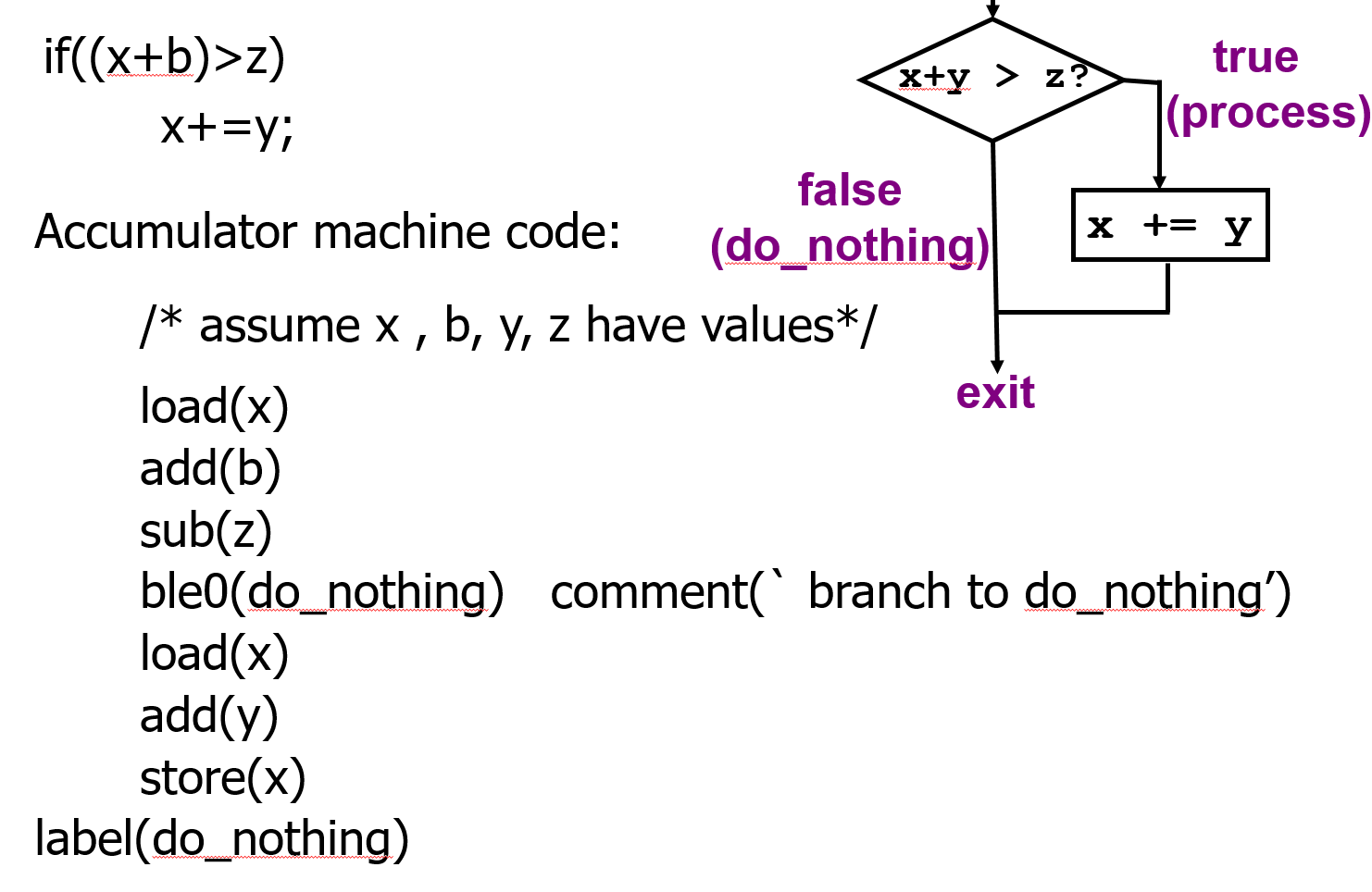


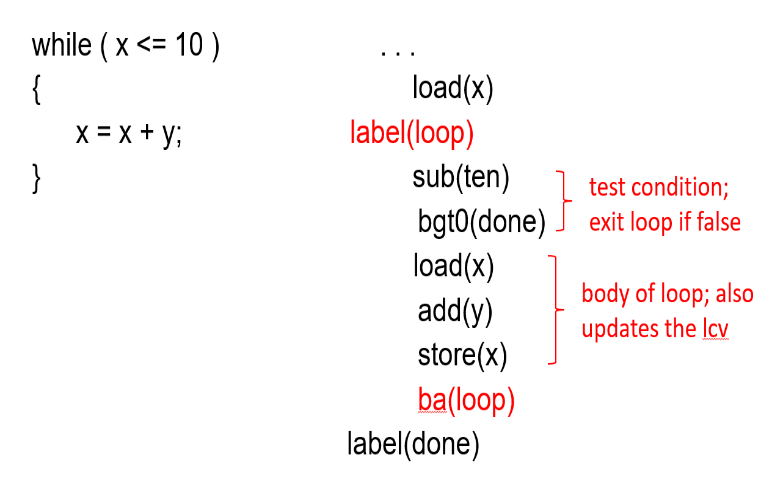


Control Structures



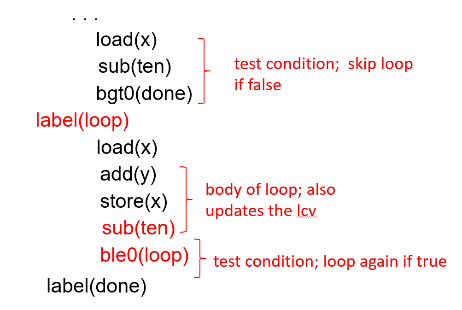
* Note that the inefficient code contains back-to-back branches
* One way to remove these is to change the condition





Loops

* All for loops, while loops, and do-while loops have an implicit branch from bottom to the top of the loop
* Branch instruction becomes explicit when translated into assembly
* Alternative implementation:



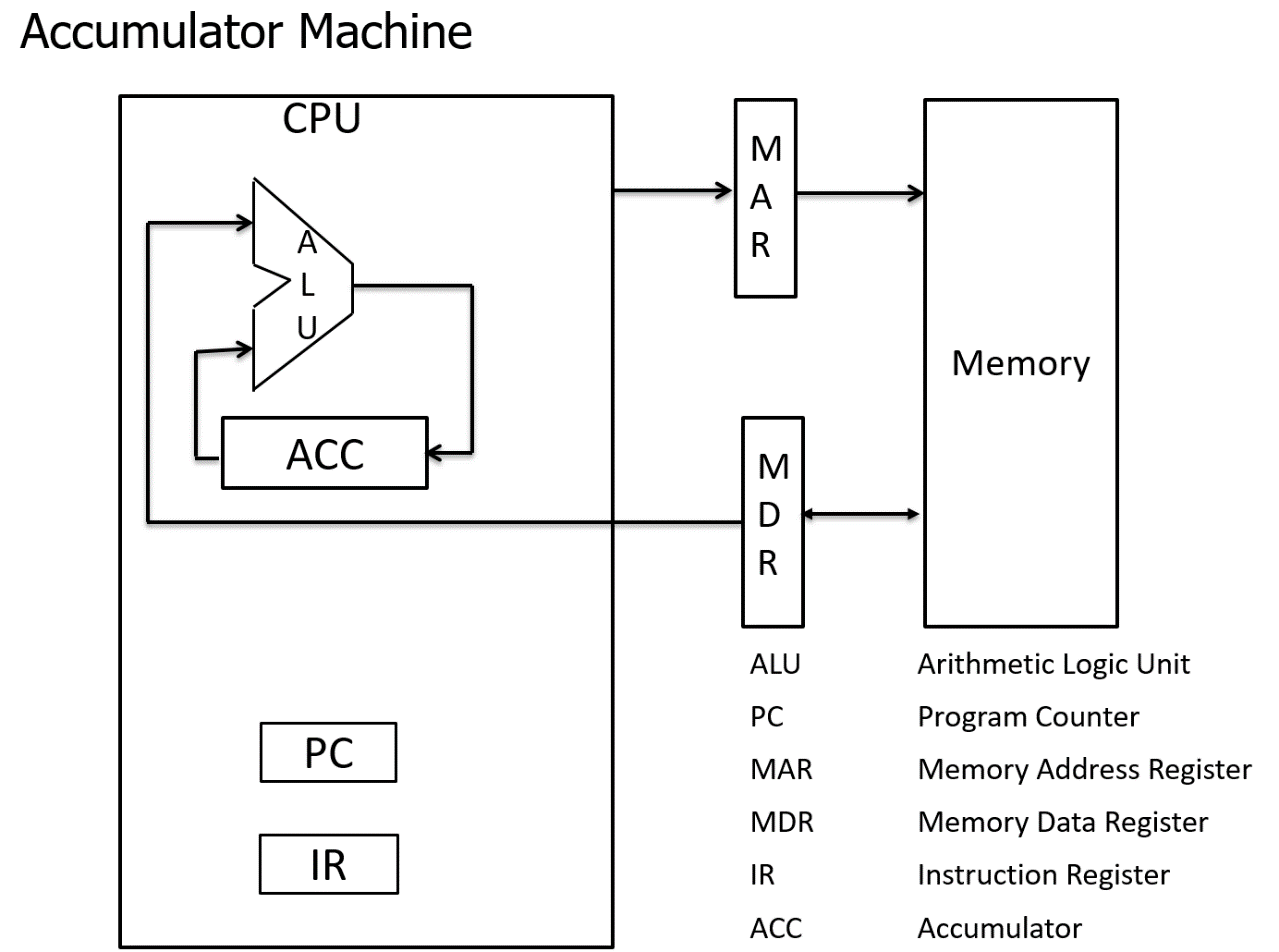
# Von Neumann Design

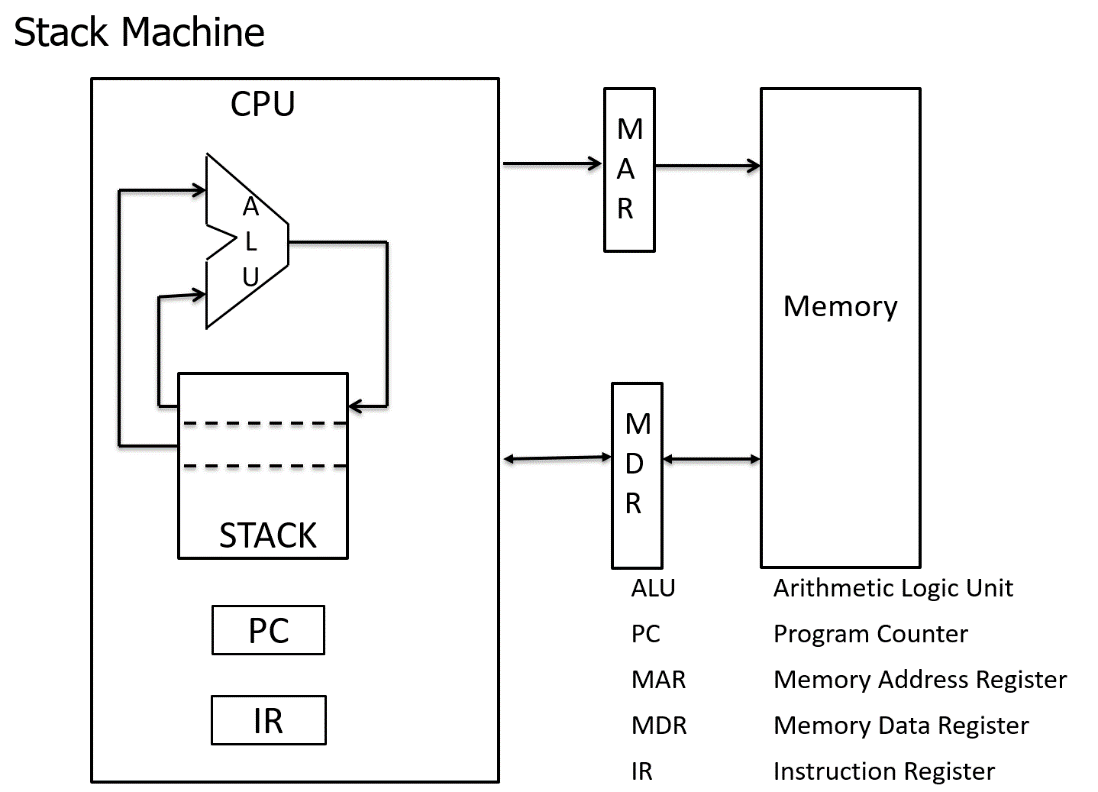
Prior to von Neumann, special memories were used to storage data via mercury or vacuum tubes, and programs were either stroed in separate memories or physicalled wired via plugboards. Von Neumann is credited with the idea of “stored program” computers, where the program is stored in the same memory as the data and indeed can be treated like data

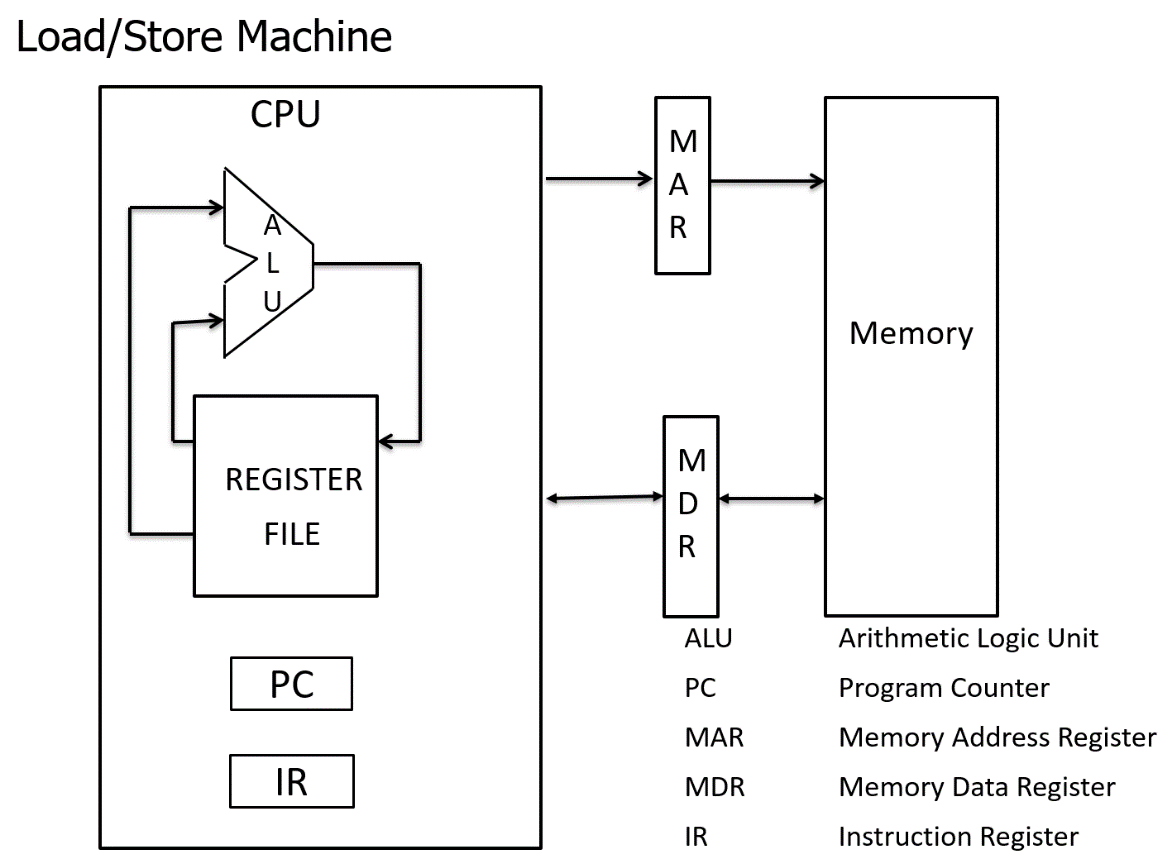
Von Neumann Machine Characteristics:

* Random-access, one-dimensional memory (vs sequential memories)
* Sequential processing (no pipelining)
* Stored program, no hardware disctinction between instructions and data, (vs Harvard architecture with separate instruction and data memories)
* Binary, parallel by word, two’s complement (vs decimal serial-by-digit, sign magnitude)
* Instruction fetch / execute cycle, branch by explicit change of PC (vs following a link address from one instruction to the next)
* Three register arithmetic – ACC, MQ, MBR
* Memory contains series of bits grouped into addressable units
* Data is accessed in memory by naming memory addresses (like a big array)
* Addresses are consecutive binary integers (insigned)
  + Convenient for addressing arrays and sequentially stepping through memories
* Bit strings have no inherent data type
  + Compiler typically type-checks all the variables so that all operations are legal

# Processors







Data Path

* Registers such as ACC or set of general registers
* Address registers such as SP and Xn
* ALU which executes operations
* Internal buses

Memory bus interface / bus interface unit

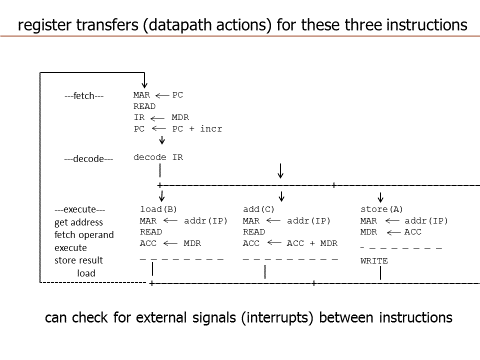
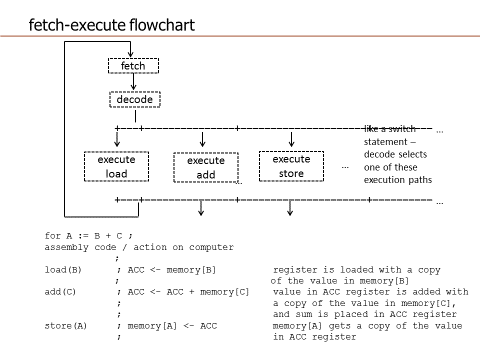
* MAR – CPU register that stores address from which data will be fetched to the CPU or the address to which data will be sent and stored
* MDR – a CPU register that contains the data to be stored in memory or the data after a fetch from memory

Control

* PC – points to next instruction
* IR – holds current instruction
* PSR – indicates results of previous operation (condition codes)

Fetch-execute cycle

* Each instruction generates a sequence of control signals
* Control signals determine the register transfers in the data path
* Timing – how long should control signals be active? Data signals must travel from a register, across the bus, through the ALU, to a register -> minimum clock cycle time



# Addressing Modes

Instruction Set Design

* Goal to minimize instruction length
* Designed with compilers in mind
* Determining how operands are addressed is a key component of instruction set design

Format

* Defines layout of bits in instruction
* Includes opcode and includes implicit or explicit operand(s)
* Usually there are several instruction forms in an instruction set
* Huge variety of instruction formats have been designed – varying widely between processors

Length

* Most basic issue
* Affected by and affects:
  + Memory size
  + Memory organization
  + Bus structure
  + CPU complexity
  + CPU speed
* Trade off between a powerful instruction repertoire and saving space with shorter instructions

Format Tradeoffs

* Large instruction set => small programs
* Small instruction set => large programs
* Large memory => longer instructions
* Fixed length instructions same size or multiple of bus width => fast fetch
* Variable length instructions may need extra bus cycles
* Processor may executre faster than fetch
  + Use cache memory or use shorter instructions
* Note complex relationship between word size, character size, instruction size, and bus transfer width
  + Modern computers these are all multiples of 8 and related to each other by powers of 8

Allocation of bits

* Determines…
  + Number of addressing modes
    - Implicit operands don’t need bits
  + Number of operands
    - 3 operand formats are rare
    - For two operand instructions we can use one or two operand mode indicators
  + Register versus memory
    - Tradeoff between amount of registers and program size
    - Studies suggest optimal number between 8 and 32
    - Most newer architectures have 32 or more
    - X86 architecture allows some computation in memory
  + Number of register sets
    - RISC architectures tend to have larger sets of uniform registers
    - Small register sets require fewer opcode bits
    - Specialized register sets can reduce opcode bits further by implicit reference (address v data registers)
  + Address range
    - Large address space requires large instructions for direct addressing
    - Many architectures have some restricted or short forms of displacement addressing
  + Address granularity
    - Size of object addressed
    - Typically 8, 16, 32, and 64 instruction variants

Addressing Modes

For a given instruction set architecture, addressing modes define how machine language instructions identify the operand of each instruction. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere. Different types of addresses involve tradeoffs between instruction length, addressing flexibility, and complexity of address calculation

Immediate

* Instruction contains value to be used in the operand
  + Add\_immediate(5)
* Value stored in memory immediately after opcode in memory
* Similar to using constant in a HLL
* Advantages:
  + Fast, no memory reference to fetch data
* Disadvantages:
  + Inflexible, limited range in machines with fixed length instructions

Direct

* Instruction states where value can be found, but the value is out in memory
* Address field contains address of operand
  + add(A)
  + Add contents of memory address A to the accumulator
* Frequently used for HLL global variables
* Advantages:
  + Single memory reference to access data
  + No additional calculations to determine effective address
  + More flexible than immediate
* Disadvantages
  + Limited address space

Memory-indirect addressing

* Memory cell pointed to by address field contains address of operand
  + Add\_indirect(A)
  + Add contents of memory cell pointed to by contents of A
* Advantages
  + Large address space
  + 2n where n = word length
  + May be nested, multilevel, cascaded
* Disadvantage
  + Multiple memory accesses, therefore slower

Register addressing

* Operand held in register named in address field
* Advantages
  + No memory accesses
  + Very fast execution
  + Very small address field needed
    - Shorter instructions
    - Faster instruction fetch
  + Multiple registers improve performance
* Disadvantages
  + Limited number of register

Register Indirect Addressing

* Operand is in memory cell pointed to by contents of register R
* Advantages:
  + Large address space
  + One fewer access than indirect addressing

Displacement addressing

* Analogy**:**
  + int list[5]
  + list -> location of array in memory (base address)
  + list[0] -> first value in list
  + list[1] -> displacement of 2\*sizeof(int) from base address
* Relative Addressing
  + Form of displacement addressing
  + R = Program counter, PC
  + EA = A + PC
  + Ie get operand from A cells from current location pointed to by PC