Architecture Review
Instruction Set Architecture

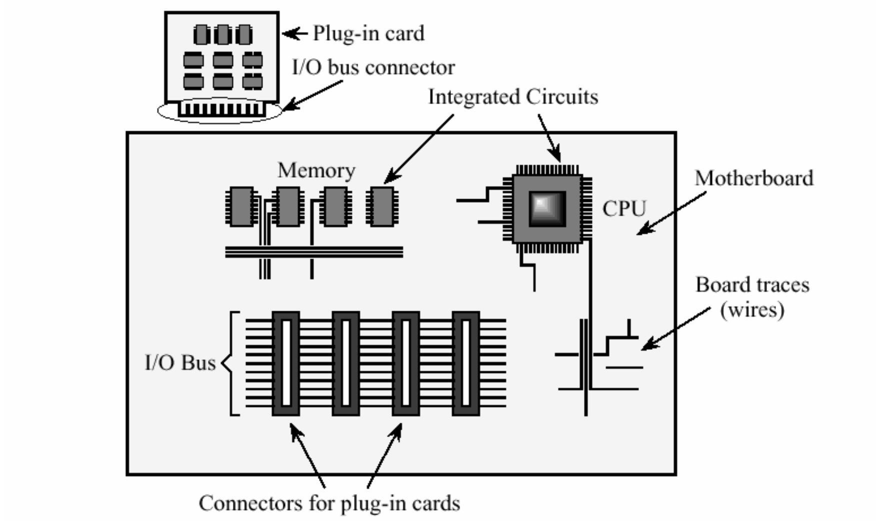
CS 221

Instruction Set Architecture

- Chapter 4 of text – Instruction Set Architecture (ISA)
  - This chapter is pretty heavy in material
  - This material is intended to lead you up to the concepts presented in chapter 4

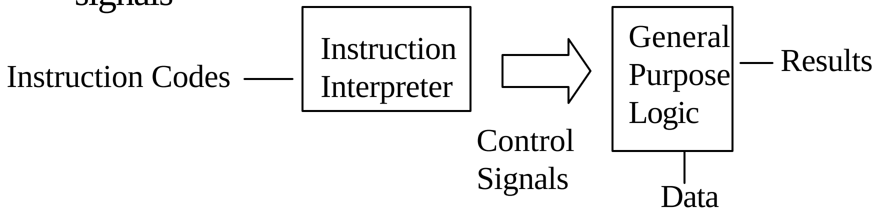
Simple Bus Architecture

- A simplified motherboard of a personal computer (top view):



Architecture Review - Program Concept

- Hardwired systems are inflexible
  - Lots of work to re-wire, or re-toggle
- General purpose hardware can do different tasks, given correct control signals
- Instead of re-wiring, supply a new set of control signals



What is a program?

- Software
  - A sequence of steps
  - For each step, an arithmetic or logical operation is done
  - For each operation, a different set of control signals is needed – i.e. an instruction

Function of Control Unit

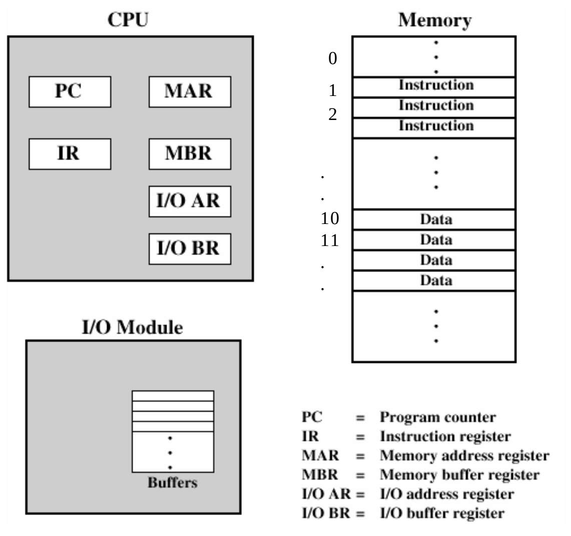
- For each operation a unique code is provided
  - e.g. ADD, MOVE (i.e. COPY)
- A hardware segment accepts the code and issues the control signals

- We have a computer!

Components

- Central Processing Unit
  - Control Unit
  - Arithmetic and Logic Unit
- Data and instructions need to get into the CPU and results out
  - Input/Output
- Temporary storage of code and results is needed
  - Main memory

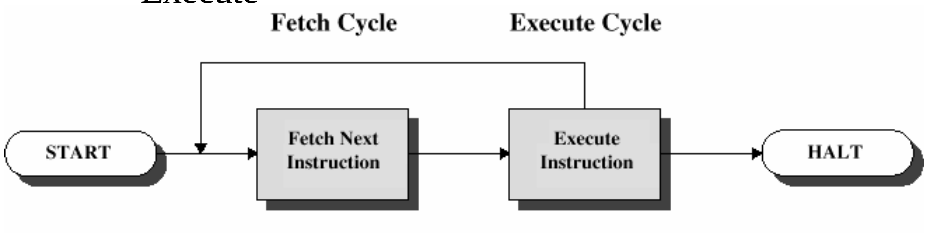
Computer Components: Top Level View



PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	I/O address register
I/O BR	=	I/O buffer register

Simplified Instruction Cycle

- Two steps:
  - Fetch
  - Execute



Fetch Cycle

- Program Counter (PC) holds address of next instruction to fetch - also called Instruction Pointer (IP)
- Processor fetches instruction from memory location pointed to by PC
- Increment PC to point to next sequential instruction
- Instruction loaded into Instruction Register (IR)
- Processor interprets instruction and performs required actions (e.g. more fetches may be necessary)

Execute Cycle

- Processor-memory
  - data transfer between CPU and main memory
- Processor I/O
  - Data transfer between CPU and I/O module
- Data processing
  - Some arithmetic or logical operation on data
- Control
  - Alteration of sequence of operations
  - e.g. jump, where we change the value in the PC
- Combination of above

What is an instruction set?

- The complete collection of instructions that are understood by a CPU
  - The physical hardware that is controlled by the instructions is referred to as the Instruction Set Architecture (ISA)
- The instruction set is ultimately represented in binary **machine code** also referred to as **object code**
  - Usually represented by assembly codes to human programmer

Elements of an Instruction

- Operation code (Op code)
  - Do this
- Source Operand reference(s)
  - To this
- Result Operand reference(s)
  - Put the answer here
- Next Instruction Reference
  - When you are done, do this instruction next

Where are the operands?

- Main memory
- CPU register
- I/O device

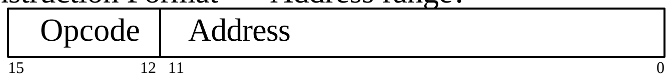
- To specify which register, which memory location, or which I/O device, we’ll need some addressing scheme for each

Instruction Representation

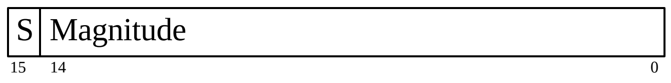
- In machine code each instruction has a unique bit pattern
  - e.g., 1011010101010011110
- For human consumption (well, programmers anyway) a symbolic representation is used called a **mnemonic**
  - e.g. ADD, SUB, LOAD
- Operands can also be represented with the mnemonic
  - ADD R0,R1

Hypothetical Machine

- Instruction Format - Address range?

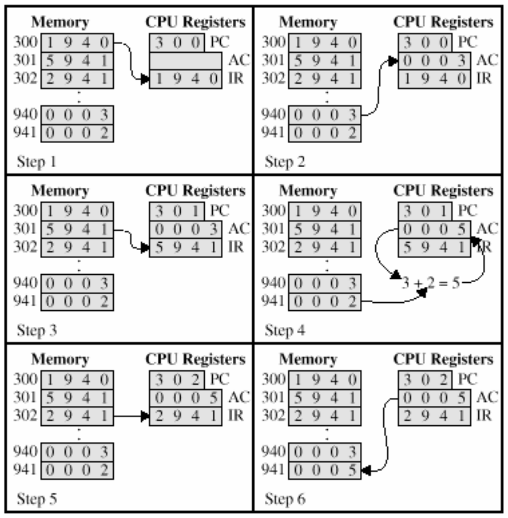


- Integer Format - Data range?



- Registers
  - PC = Program Counter, IR = Instruction Register, AC = Accumulator
- Partial List of Opcodes
  - 0001 = Load AC from Memory
  - 0010 = Store AC to Memory
  - 0101 = Add to AC from Memory

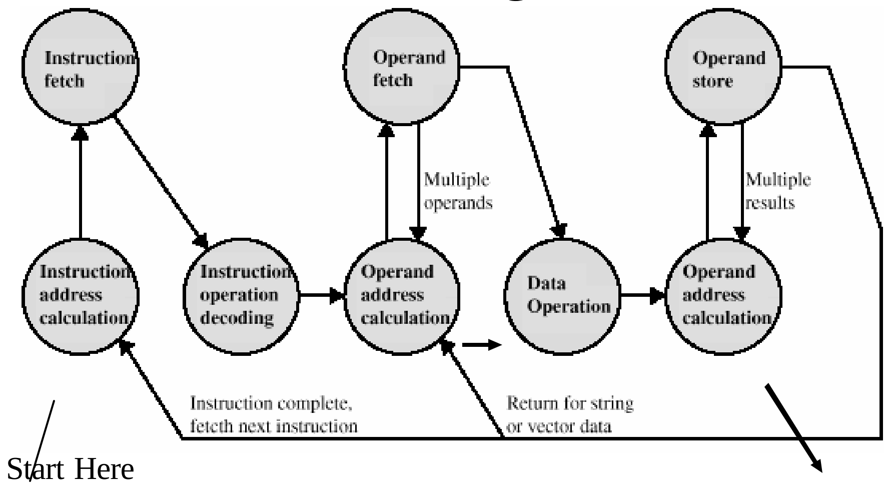
Example of Program Execution



Modifications to Instruction Cycle

- Simple Example
  - Always added one to PC
  - Entire operand fetched with instruction
- More complex examples
  - Might need more complex instruction address calculation
    - Consider a 64 bit processor, variable length instructions
  - Instruction set design might require repeat trip to memory to fetch operand
    - In particular, if memory address range exceeds word size
  - Operand store might require many trips to memory
    - Vector calculation

Instruction Cycle - State Diagram


Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Instruction complete, fetcth next instruction
Return for string or vector data
Start Here

Note: this diagram is still incomplete!

Instruction Set Design

- One important design factor is the number of operands contained in each instruction
  - Has a significant impact on the word size and complexity of the CPU
  - E.g. lots of operands generally implies longer word size needed for an instruction
- Consider how many operands we need for an ADD instruction
  - If we want to add the contents of two memory locations together, then we need to be able to handle at least **two** memory addresses
  - Where does the result of the add go? We need a **third** operand to specify the destination
  - What instruction should be executed next?
    - Usually the next instruction, but sometimes we might want to jump or branch somewhere else
    - To specify the next instruction to execute we need a **fourth** operand
- If all of these operands are memory addresses, we need a really long instruction!

Number of Operands

- In practice, we won’t really see a four-address instruction.
  - Too much additional complexity in the CPU
  - Long instruction word
  - All operands won’t be used very frequently
- Most instructions have one, two, or three operand addresses
  - The next instruction is obtained by incrementing the program counter, with the exception of branch instructions
- Let’s describe a hypothetical set of instructions to carry out the computation for:

  Y = (A-B) / (C + (D * E))

Three operand instruction

- If we had a three operand instruction, we could specify two source operands and a destination to store the result.
- Here is a possible sequence of instructions for our equation:

  Y = (A-B) / (C + (D * E))

```
– SUB R1, A, B          ; Register R1 ← A-B
– MUL R2, D, E          ; Register R2 ← D * E
– ADD R2, R2, C         ; Register R2 ← R2 + C
– DIV  R1, R1, R2       ; Register R1 ← R1 / R2
```

- The three address format is fairly convenient because we have the flexibility to dictate where the result of computations should go. Note that after this calculation is done, we haven’t changed the contents of any of our original locations A,B,C,D, or E.

Two operand instruction

- How can we cut down the number of operands?
  - Might want to make the instruction shorter
- Typical method is to assign a **default** destination operand to hold the results of the computation
  - Result always goes into this operand
  - Overwrites and old data in that location
- Let's say that the default destination is the first operand in the instruction
  - First operand might be a register, memory, etc.

Two Operand Instructions

- Here is a possible sequence of instructions for our equation (say the operands are registers):

  $$Y = (A-B) / (C + (D * E))$$

  - SUB A, B ; Register A ← A-B
  - MUL D, E ; Register D ← D*E
  - ADD D, C ; Register D ← D+C
  - DIV A, D ; Register A ← A / D
- Get same end result as before, but we changed the contents of registers A and D
- If we had some later processing that wanted to use the original contents of those registers, we must make a copy of them before performing the computation
  - MOV R1, A ; Copy A to register R1
  - MOV R2, D ; Copy D to register R2
  - SUB R1, B ; Register R1 ← R1-B
  - MUL R2, E ; Register R2 ← R2*E
  - ADD R2, C ; Register R2 ← R2+C
  - DIV R1, R2 ; Register R1 ← R1 / R2
- Now the original registers for A-E remain the same as before, but at the cost of some extra instructions to save the results.

One Operand Instructions

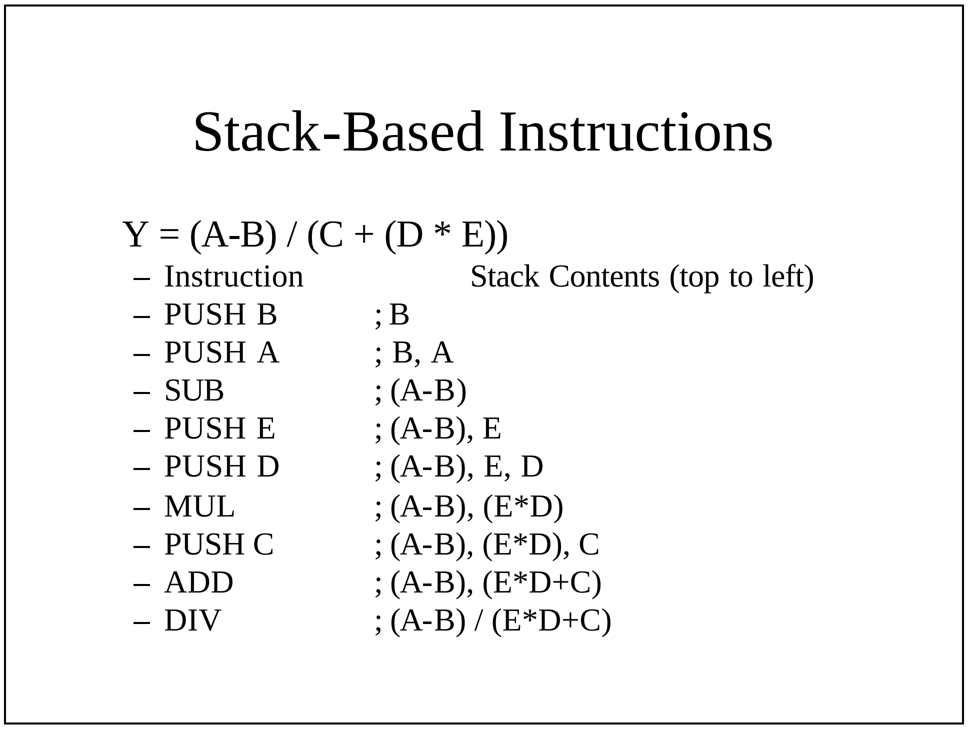
- Can use the same idea to get rid of the second operand, leaving only one operand
- The second operand is left implicit; e.g. could assume that the second operand will always be in a register such as the Accumulator:

  Y = (A-B) / (C + (D * E))

  - LDA D ; Load ACC with D
  - MUL E ; Acc ← Acc * E
  - ADD C ; Acc ← Acc + C
  - STO R1 ; Store Acc to R1
  - LDA A ; Acc ← A
  - SUB B ; Acc ← A-B
  - DIV R1 ; Acc ← Acc / R1
- Many early computers relied heavily on one-address based instructions, as it makes the CPU much simpler to design. As you can see, it does become somewhat more unwieldy to program.

Zero Operand Instructions

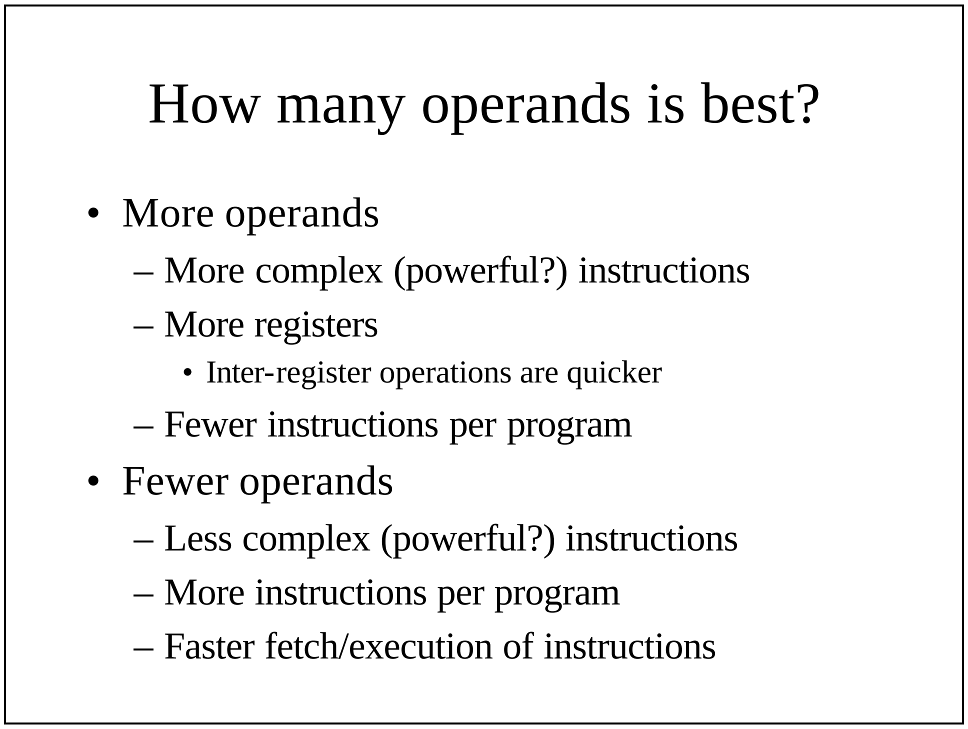
- In some cases we can have zero operand instructions
- Uses the **Stack**
  - Section of memory where we can add and remove items in LIFO order
  - Last In, First Out
  - Envision a stack of trays in a cafeteria; the last tray placed on the stack is the first one someone takes out
  - The stack in the computer behaves the same way, but with data values
    - PUSH A ; Places A on top of stack
    - POP A ; Removes value on top of stack and puts result in A
    - ADD ; Pops top two values off stack, pushes result back on



Stack-Based Instructions

Y = (A-B) / (C + (D * E))

– Instruction	Stack Contents (top to left)
– PUSH B	; B
– PUSH A	; B, A
– SUB	; (A-B)
– PUSH E	; (A-B), E
– PUSH D	; (A-B), E, D
– MUL	; (A-B), (E*D)
– PUSH C	; (A-B), (E*D), C
– ADD	; (A-B), (E*D+C)
– DIV	; (A-B) / (E*D+C)



How many operands is best?

- More operands
  - More complex (powerful?) instructions
  - More registers
    - Inter-register operations are quicker
  - Fewer instructions per program
- Fewer operands
  - Less complex (powerful?) instructions
  - More instructions per program
  - Faster fetch/execution of instructions

Design Tradeoff Decisions

- Operation repertoire
  - How many ops?
  - What can they do?
  - How complex are they?
- Data types
  - What types of data should ops perform on?
- Registers
  - Number of registers, what ops on what registers?
- Addressing
  - Mode by which an address is specified (more on this later)

RISC vs. CISC

- RISC – Reduced Instruction Set Computer
  - Advocates fewer and simpler instructions
  - CPU can be simpler, means each instruction can be executed quickly
  - Benchmarks: indicate that most programs spend the majority of time doing these simple instructions, so make the common case go fast!
  - Downside: uncommon case goes slow (e.g., instead of a single SORT instruction, need lots of simple instructions to implement a sort)
  - Sparc, Motorola, Alpha
- CISC – Complex Instruction Set Computer
  - Advocates many instructions that can perform complex tasks
  - E.g. SORT instruction
  - Additional complexity in the CPU
    - This complexity typically makes ALL instructions slower to execute, not just the complex ones
  - Fewer instructions needed to write a program using CISC due to richness of instructions available
  - Intel x86