# Outline

- Organization overview
- Instruction Set Architecture (ISA)
- Instruction set design principles
- ISA classes



# A Look Ahead

Chapter 3

- Overview of computer organization
  - Fetch/Execute cycle
  - Computer operations
  - addressing modes
- Instruction set design principles
- Overview of MIPS architecture
- Stored program concept
- Learn a subset of MIPS assembly language
- Show how MIPS instructions are represented in machine language
- Contrast MIPS architecture with other X86 and PowerPC architectures

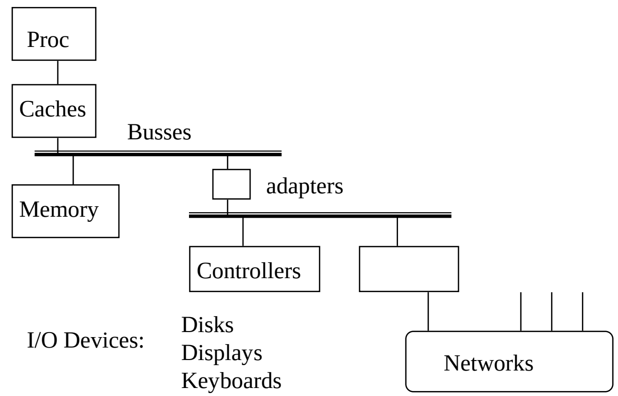


# Organization Review

- All computers consist of five components
  - Processor: (1) datapath and (2) control
  - (3) Memory
  - (4) Input devices and (5) Output devices
- Not all “memory” are created equally
  - Cache: fast (expensive) memory are placed closer to the processor
  - Main memory: less expensive memory--we can have more
- Input and output (I/O) devices have the messiest organization
  - Wide range of speed: graphics vs. keyboard
  - Wide range of requirements: speed, standard, cost ...
  - Least amount of research (so far)



## Summary: Computer System Components




- All have interfaces & organizations

# Instruction Set Architecture

*Instruction Set Architecture* of a machine is an abstraction Interface between the hardware and the lowest-level software

•Includes anything that programmers need to know to make a binary machine program work correctly, including

✓Instructions (categories, format, ..)

✓I/O devices, # of registers, memory access schemes , …

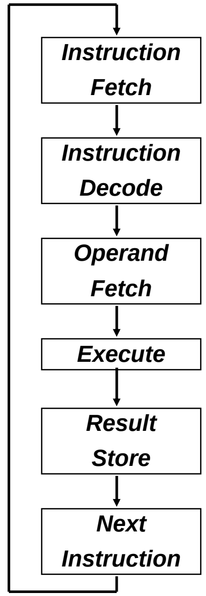
## ISA design principles

•Simplicity favors regularity

•Smaller is faster

•Make the common case fast






# ISA: What must be specified?

- Instruction Format or Encoding
  – how is it decoded?
- Location of operands and result
  – where other than memory?
  – how many explicit operands?
  – how are memory operands located?
  – which can or cannot be in memory?
- Data type and Size
- Operations
  – what are supported
- Successor instruction
  – jumps, conditions, branches



# Registers

•A storage device (in MIPS holds 32 bits, a *word*)

•Registers are built into the processor, in particular in the section labeled "Data Path"

•Faster than main memory

•The basic ISA classes are distinguished by

➢what kinds of registers they have

➢what are the functions of the registers



# Basic ISA Classes

Accumulator known as 1-operand machines

Use 1 register for source as well as destination operands

| | | |
|---|---|---|
| 1-operand | Add A | acc ← acc + mem[A] |
| | Load A | acc ← mem[A] |
| | Store B | mem[B] ← acc |

Stack known as 0-operand machines

Operands are pushed and popped off an internal stack

All other operations remove their operands from the stack and replace them with the result

| | | |
|---|---|---|
| | Push A | push (mem[A]) |
| | Push B | push (mem[B]) |
| 0-operand | Add | push (pop+pop) |
| | Pop C | mem[A] ← pop |

# Basic ISA Classes

General Purpose Register

Limited number of registers to store data for any purpose

Memory-Memory or Register-Memory

(arithmetic instructions can use data in memory for operands)

| | | |
|---|---|---|
| 1-operand | Add Ra, B | Ra ← Ra + mem[B] |
| 2-operand | Add A,B | mem[A] ← mem[A] + mem[B] |
| 3-operand | Add A,B,C | mem[A] ← mem[B] + mem[C] |

Load/Store (arithmetic instructions can only use data in registers)

(register-register machines)

| | |
|---|---|
| Add Ra,Rb,Rc | Ra ← Rb + Rc |
| Load Ra,A | Ra ← mem[A] |
| Store Ra,B | mem[B] ← Ra |



# Comparison of ISA classes

Code sequence for C = A + B for classes of instruction sets:

| Stack | Accumulator | Register |
|---|---|---|
| | | (load-store) |
| Push A | Load A | Load R1,A |
| Push B | Add B | Load R2,B |
| Add | Store C | Add R3,R1,R2 |
| Pop C | | Store C,R3 |



# Which ISA dominates?

General purpose registers dominate

1975-1995 all machines use general purpose registers

Advantages of registers

- registers are faster than memory
- registers are easier for a compiler to use
  e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
- registers can hold variables
  - memory traffic is reduced, so program is sped up (since registers are faster than memory)
  - code density improves (since register named with fewer bits than memory location