CS1004: Intro to CS in Java, Spring 2005

Lecture #10: Computer architecture

Janak J Parekh
janak@cs.columbia.edu

Administrivia

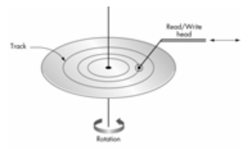
- HW#2 due Tuesday

Mass Storage

- RAM is *volatile*
  - Not useful for permanent storage, and expensive in large quantities
- Use *nonvolatile* mass storage (magnetic media, flash) for permanent storage
  - Random/direct access: hard drives, CD/DVD-ROMs
    - Uses its own addressing scheme to access data
  - Sequential access: tape drives
    - Stores data sequentially; slow
    - Primarily for backup nowadays

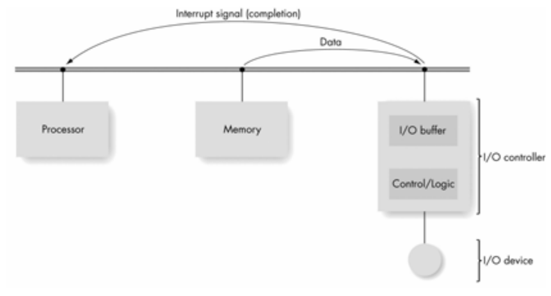
Hard Disks

- Data stored on (hard) spinning disks (*platters*)
- Disk divided into concentric rings (*tracks*)
- Read/write head moves from one ring to another while disk spins
- Access time depends on:
  - Time to move head to correct sector (seek)
  - Time for sector to spin to data location (latency)



I/O Controller

- Intermediary between central processor and I/O devices
- Hard drives are *much* slower than memory, so…
  - Processor sends request and data, then goes on with its work
  - I/O controller *interrupts* processor when request is complete
- *Memory hierarchy* of a computer (registers fastest, tape slowest)



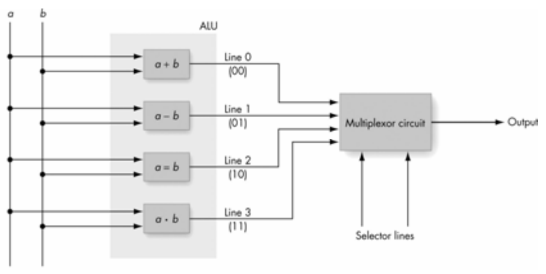
Organization of an I/O Controller (page 202)

The Arithmetic/Logic Unit

- Actual computations are performed
- Primitive operation circuits
  - Arithmetic (ADD, etc.)
  - Comparison (CE, etc.)
  - Logic (AND, etc.)
- Data inputs and results stored in registers
- Multiplexor selects desired output

ALU Process

- Values for operations copied into ALU's input register locations
- All circuits compute results for those inputs
- Multiplexor selects the one desired result from all values
- Result value copied to desired result register



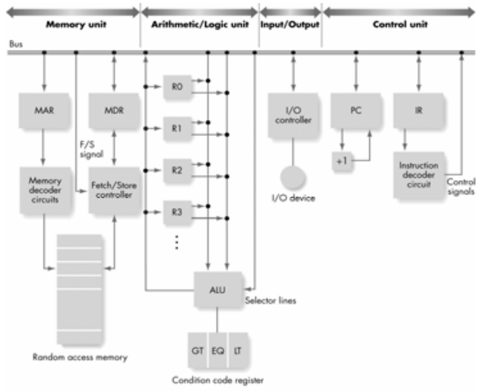
Using a Multiplexor Circuit to Select the Proper ALU Result (p. 207)

The Control Unit

- Manages stored program execution
- Task
  - *Fetch* from memory the next instruction to be executed
  - *Decode* it: determine what is to be done
  - *Execute* it: issue appropriate command to ALU, memory, and I/O controllers
- Instructions are selected from an *instruction set language*

Control Unit Components

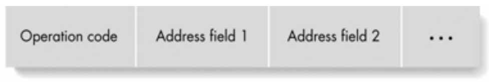
- Parts of control unit
  - Links to other subsystems (I/O controllers, etc.)
  - Instruction decoder circuit
  - Two more special registers:
    - Program Counter (PC): Stores the memory address of the next instruction to be executed
    - Instruction Register (IR): Stores the code for the current instruction
- We follow the *fetch-decode-execute* cycle repeatedly until the machine is turned off



The Organization of a Von Neumann Computer (p. 516)

Machine Language Instructions

- Can be decoded and executed by control unit
- Parts of instructions
  - Operation code (opcode): Unique unsigned-integer code assigned to each machine language operation
  - Address field(s): Memory addresses of the values on which operation will work



Main types of instructions

- Data transfer
  - Move values to and from memory and registers
  - Fetch/store operations
- Arithmetic/logic
  - Perform ALU operations that produce numeric values
- Compares
  - Set bits of compare register to hold result
- Branches
  - Jump to a new memory address to continue processing

Let's design a processor

- Flexibility as to how we design it
- RISC or CISC – how many instructions? How many operands?
  - We'll stick to one *decimal* operand for simplicity
- How many registers?
  - Book's convention: we'll use one temporary register cell ("R") for math operations
  - Rest will be to and from main memory
  - *Accumulator* architecture: early Intel CPUs derived from this
- What's the setup of the memory for data or code?
  - For simplicity's sake, we'll keep everything near each other

Book’s hypothetical machine: basic operations

Binary opcode	Operation	Meaning
0000	LOAD X	CON(X) → R
0001	STORE X	R → CON(X)
0010	CLEAR X	0 → CON(X)
0011	ADD X	R + CON(X) → R
0100	INCREMENT X	CON(X) + 1 → CON X
0101	SUBTRACT X	R – CON(X) → R
0110	DECREMENT X	CON(X) – 1 → CON(X)
0111	COMPARE X	*Sets “condition code” for JUMPs*

Dealing with compare

- We want to design *conditional* code: based on a particular result, run different pieces of code
- COMPARE will set one of three “condition codes” (in a special register) to 1, and the rest to 0
  - GT (greater than)
  - EQ (equal)
  - LT (less than)
- We can then tell the processor to *jump* to other code based on the result
- We’ll explore conditionals in much greater detail in Java

Book’s hypothetical machine: jumps and I/O

Binary opcode	Operation	Meaning
1000	JUMP X	Get next instruction from memory location X
1001	JUMPGT X	Get next instruction from X if GT = 1
1010	JUMPEQ X	Get next instruction from X if EQ = 1
1011	JUMPLT X	Get next instruction from X if LT = 1
1100	JUMPNEQ X	Get next instruction from X if EQ = 1
1101	IN X	Get input and store in X
1110	OUT X	Output (in decimal) value at X
1111	HALT	Stop program execution

Simple examples

- Practice problem 1, p. 213: set *a* to the value *b*+*c*+*d*
- Practice problem 2, p. 213: if (a = b), set *c* to the value of *d*
  - Note different use of equals in the book – we mean equality here, not assignment
- Let's assume *a* is at memory location 100, *b* is at 101, *c* at 102, *d* at 103, and that the code starts at memory location 50

The Future: Non-Von Neumann Architectures

- Physical limitations on speed of Von Neumann computers
- Non-Von Neumann architectures explored to bypass these limitations
- Parallel computing architectures can provide improvements: multiple operations occur at the same time
  - SIMD instructions: apply single instruction to a vector of data
  - MIMD instructions: essentially multiple closely coordinated processors in parallel
  - Hyperthreading, dual-core processors

Segue/next time

- Start thinking about memory and object management in Java
- Chapter 3 of Lewis/Loftus covers how to use existing classes and object in Java
- Chapter 4 will cover how to make *our own* classes in greater detail
- Memory architecture we've just discussed will help visualize how objects work