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)
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

<table>
<thead>
<tr>
<th>Operation code</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>...</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Binary opcode</th>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LOAD X</td>
<td>CON(X) (\rightarrow) R</td>
</tr>
<tr>
<td>0001</td>
<td>STORE X</td>
<td>R (\rightarrow) CON(X)</td>
</tr>
<tr>
<td>0010</td>
<td>CLEAR X</td>
<td>0 (\rightarrow) CON(X)</td>
</tr>
<tr>
<td>0011</td>
<td>ADD X</td>
<td>R + CON(X) (\rightarrow) R</td>
</tr>
<tr>
<td>0100</td>
<td>INCREMENT X</td>
<td>(\text{CON(X) + 1} \rightarrow \text{CON(X)})</td>
</tr>
<tr>
<td>0101</td>
<td>SUBTRACT X</td>
<td>(\text{R} - \text{CON(X)} \rightarrow \text{R})</td>
</tr>
<tr>
<td>0110</td>
<td>DECREMENT X</td>
<td>(\text{CON(X)} - 1 \rightarrow \text{CON(X)})</td>
</tr>
<tr>
<td>0111</td>
<td>COMPARE X</td>
<td>Sets &quot;condition code&quot; for JUMP/s</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Binary opcode</th>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>JUMP X</td>
<td>Get next instruction from memory location X</td>
</tr>
<tr>
<td>1001</td>
<td>JUMP GT X</td>
<td>Get next instruction from X if GT = 1</td>
</tr>
<tr>
<td>1010</td>
<td>JUMP EQ X</td>
<td>Get next instruction from X if EQ = 1</td>
</tr>
<tr>
<td>1011</td>
<td>JUMP LT X</td>
<td>Get next instruction from X if LT = 1</td>
</tr>
<tr>
<td>1100</td>
<td>JUMP NEQ X</td>
<td>Get next instruction from X if NEQ = 1</td>
</tr>
<tr>
<td>1101</td>
<td>IN X</td>
<td>Get input and store in X</td>
</tr>
<tr>
<td>1110</td>
<td>OUT X</td>
<td>Output (in decimal) value at X</td>
</tr>
<tr>
<td>1111</td>
<td>HALT</td>
<td>Stop program execution</td>
</tr>
</tbody>
</table>
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