Topics

- Cumulative Final Exam: Saturday, December 20
  - FGH 306, 2pm – 5pm
  - All material covered in the class is fair game
  - Closed-book and closed-notes
  - Study old tests

- Review Course Highlights

- TA Evaluation Forms
  - Last 10 minutes of class

Structured Computer Organization

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Problem-oriented language level</td>
</tr>
<tr>
<td></td>
<td>Translation (compiler)</td>
</tr>
<tr>
<td>4</td>
<td>Assembly language level</td>
</tr>
<tr>
<td></td>
<td>Translation (assembler)</td>
</tr>
<tr>
<td>3</td>
<td>Operating system machine level</td>
</tr>
<tr>
<td></td>
<td>Partial interpretation (operating system)</td>
</tr>
<tr>
<td>2</td>
<td>Instruction set architecture level</td>
</tr>
<tr>
<td></td>
<td>Interpretation (microprogram) or direct execution</td>
</tr>
<tr>
<td>1</td>
<td>Microarchitecture level</td>
</tr>
<tr>
<td></td>
<td>Hardware</td>
</tr>
<tr>
<td>0</td>
<td>Digital logic level</td>
</tr>
</tbody>
</table>
Boolean Algebra Defined

• Provides the operations and the rules for working with the binary set \{0,1\}

• Used in the study of electronic switches
  – “0” represents “off”, “low”, or “false”
  – “1” represents “on”, “high”, or “true”

Boolean functions are represented using variables and operators

Logic Gates

• AND, NAND

• OR, NOR

• XOR, XNOR

• NOT
Multiplexers/Demultiplexers

- **MUX**
  - $2^n$ data inputs, 1 data output, n control signals
  - Binary code on select determines which input is routed to output

- **DEMUX**
  - 1 data input, $2^n$ data outputs, n control signals
  - Binary code on select routes a single input signal to one of $2^n$ outputs

Decoders/Encoders

- **Decoder**
  - Takes an n-bit number as input
  - Selects (sets to “1”) exactly 1 of $2^n$ outputs

- **Encoder**
  - Input is a group of parallel bits
  - Output is the binary code assigned to asserted input
Comparators and Shifters

• Comparator
  – Determine if two input words are equal
  – Based upon XOR gate

• Shifter
  – Arithmetic maintains the sign
    • 1-bit arithmetic shift left multiplies by 2
    • 1-bit arithmetic shift right divides by 2
  – Logical fills empty bits with “0”
    • Shift left logical 8 from the Mic-1

Adders

• Half adder
  – Inputs A, B
  – Outputs $C_{out}$, Sum

• Full adder
  – Inputs A, B, $C_{in}$
  – Outputs $C_{out}$, Sum

• Ripple-carry adder
• Carry-select adder
Block Diagram of Mic-1 Microarchitecture

**Datapath**
Part of CPU containing ALU, its inputs, and its outputs

**Control Section**
Part of CPU containing the H/W necessary to direct the datapath

**Purpose**
Implement the ISA level above it (*macro-architecture*)

---

**Datapath Registers**

- Memory **Address Register**
- Memory **Data Register**
- Program **Counter**
- Memory **Buffer Register**
- **Stack Pointer**
- Local **Variable pointer**
- Constant **Pool Pointer**
- **Top Of Stack**
- **OPC**
- **Holding register**

Hold either an **address** or a **data value**
Datapath Timing

- Has a finite propagation time
  - Signals travel along wires, through transistors, etc.

- Implicit clock subcycles
  - Set up control signals to drive datapath
  - Register loaded onto B bus
  - ALU and shifter operate
  - Result propagates along C bus to registers

Timing Diagram for One Datapath Cycle
Control Store (ROM)

- Memory that holds the microprogram
- Contains 512 words, each a 36-bit microinstruction
- Each microinstruction specifies its successor
  - Not executed in order stored in control store
- Accessing the microprogram
  - MicroProgram Counter holds address for next microinstruction
  - MicroInstruction Register holds the current microinstruction

Microinstruction Notation

- **B** field is source register
- **Mem** field for initiating memory operation
- **C** field sets one (or more) destination registers
- **ALU** field determines the ALU operation
- **JAM** field determines the branching
- **Addr** field explicitly names the successor of the current microinstruction
Instruction Set Architecture Level

- Interface between software and hardware
- Both compilers and hardware must understand ISA
  - Compilers translate high-level language into object code
  - Hardware must directly execute or interpret ISA

Why Design a Good ISA?

- Can last for a long time
  - Customers will want backward compatibility
- Important for performance considerations
  - Execute programs quickly
- Important for hardware implementations
  - Manufacturing cost
- Important for emerging domains/markets
  - Embedded systems, multimedia processors
Factors for a Good ISA

Define a set of instructions that can be implemented efficiently in current and future technologies

- Technology trends
  - IC logic technology
    • Improves transistor density, die size, device speed
  - Semiconductor DRAM
    • Higher densities increases memory capacity
  - Magnetic disk technology
    • Larger hard drives and improved access times
  - Network technology
    • Improved switches and transmission systems

Factors for a Good ISA

Provides a clean target for compiled code

- Regularity and completeness
  - Provide a range of alternatives
  - Easy to generate good code for high-level language
  - Compiler must make the best choice among alternatives
Instruction Types

- Data movement
- Dyadic (two operands)
- Monadic (one operand)
- Comparisons and conditional branches
- Procedure calls
- Loop control
- Input/output

IJVM Instruction Set

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>BIPUSH byte</td>
<td>Push byte onto stack</td>
</tr>
<tr>
<td>0x50</td>
<td>DUP</td>
<td>Copy top word on stack and push onto stack</td>
</tr>
<tr>
<td>0x47</td>
<td>GOTO offset</td>
<td>Unconditional branch</td>
</tr>
<tr>
<td>0x69</td>
<td>IADD</td>
<td>Pop two words from stack; push their sum</td>
</tr>
<tr>
<td>0x7E</td>
<td>IAND</td>
<td>Pop two words from stack; push Boolean AND</td>
</tr>
<tr>
<td>0x99</td>
<td>IF_EQ offset</td>
<td>Pop word from stack and branch if it is zero</td>
</tr>
<tr>
<td>0x98</td>
<td>IF_LT offset</td>
<td>Pop word from stack and branch if it is less than zero</td>
</tr>
<tr>
<td>0x9F</td>
<td>IF_ICMPGE offset</td>
<td>Pop two words from stack; branch if equal</td>
</tr>
<tr>
<td>0x84</td>
<td>IINC varnum const</td>
<td>Add a constant to a local variable</td>
</tr>
<tr>
<td>0x15</td>
<td>ILOAD varnum</td>
<td>Push local variable onto stack</td>
</tr>
<tr>
<td>0xB6</td>
<td>INVOKEVIRTUAL disp</td>
<td>Invoke a method</td>
</tr>
<tr>
<td>0x80</td>
<td>IOR</td>
<td>Pop two words from stack; push Boolean OR</td>
</tr>
<tr>
<td>0xAC</td>
<td>IRETURN</td>
<td>Return from method with integer value</td>
</tr>
<tr>
<td>0x36</td>
<td>ISTORE varnum</td>
<td>Pop word from stack and store in local variable</td>
</tr>
<tr>
<td>0x84</td>
<td>ISUB</td>
<td>Pop two words from stack; push their difference</td>
</tr>
<tr>
<td>0x13</td>
<td>LDC_W index</td>
<td>Push constant from constant pool onto stack</td>
</tr>
<tr>
<td>0x03</td>
<td>NOP</td>
<td>Do nothing</td>
</tr>
<tr>
<td>0x57</td>
<td>POP</td>
<td>Delete word on top of stack</td>
</tr>
<tr>
<td>0x5F</td>
<td>SWAP</td>
<td>Swap the two top words on the stack</td>
</tr>
<tr>
<td>0xC4</td>
<td>WIDE</td>
<td>Prefix instruction; next instruction has a 16-bit index</td>
</tr>
</tbody>
</table>

Figure 4.11: The JVM instruction set. The operands byte, cons, and varnum are 1 byte. The operands disp, index, and offset are 2 bytes.
Modern CPU Design

- Superscalar – contains multiple functional units
- Pipelined – instructions execute in stages

Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Use different resources to execute multiple tasks simultaneously
- Potential speedup = Number pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

Adapted from David Patterson’s CS 252 lecture notes. Copyright 2001 UCB.
Conventional Pipelined Execution Representation

• Assume each stage is one cycle
• # of cycles = # of instructions + (# of stages – 1)
  – Assuming full utilization of pipeline

Data Hazards

• Read After Write (RAW)
  – True data dependence (most common type)
  – Must preserve program order to ensure correct execution

• Write After Read (WAR)
  – Output dependence
  – Only present in pipelines that
    • write in more than one pipe stage
    • allow instructions to proceed after previous instructions stalls

• Write After Write (WAW)
  – Anti-dependence
  – Can occur when instructions are reordered
Control Hazards

• The IFU pipelines with sequential instruction stream in method area

• A branch may indicate another control flow
  
  ILOAD i
  BIPUSH 3
  IF_ICMPEQ L1
  ILOAD j
  BIPUSH 1
  ISUB
  ISTORE j
  GOTO L2

  L1: BIPUSH 0
  ISTORE k

  L2: ...

  The pipeline issues the next sequential instruction before resolving if the branch is taken or not

Dynamic Branch Prediction

• Branch History Table (BHT)
  – CPU maintains a history of branch locations and their previous behavior

• Branch Target Address Cache (BTAC)
  – Stores the target location for branches predicted taken

• Use 1 bit to store history
  – Stores what the branch did the last time
  – With loops, you will always mispredict entering and exiting

• Use 2 bits to store history
  – Maintains a strong and weak prediction state
  – Must mispredict twice to change the prediction
Hamming Algorithm

- In a Hamming code
  - $r$ parity bits added to $m$-bit word
  - Forms codeword with length $(m + r)$ bits

- Bit numbering
  - Starts at 1 with leftmost (high-order) bit
  - All powers of 2 are parity bits
  - Remaining bits are for data

Bit Numbering for Hamming Algorithm

Given an 8-bit data word to encode
Parity Bit Assignment

<table>
<thead>
<tr>
<th>Bit No.</th>
<th>Bit No. in Binary</th>
<th>Encoded by</th>
<th>Encodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0 0 0 1</td>
<td></td>
<td>3, 5, 7, 9, 11</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td></td>
<td>3, 6, 7, 10, 11</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>2, 1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td></td>
<td>5, 6, 7, 12</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>4, 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>4, 2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>4, 2, 1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td></td>
<td>9, 10, 11, 12</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>8, 1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0</td>
<td>8, 2</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>8, 2, 1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0</td>
<td>8, 4</td>
<td></td>
</tr>
</tbody>
</table>

Basic Cache Algorithm

- CPU read (memory reference)
- Line in cache? (yes (hit) / no (miss))
- Read bytes from cache
- Read line from memory into cache
- Extract desired bytes
- Return data to CPU
Terminology

- **Hit** – memory location found in cache

- **Miss** – memory location not in cache
  - Compulsory
    - Cache is empty at the start of a program
  - Conflict
    - Another valid cache line is currently stored in the location
  - Capacity
    - Cache isn’t large enough to hold the entire working set of a program

Why Do Caches Work?

- **Spatial locality** – words in close physical proximity to the word being read will probably be read also.

- **Temporal locality** – a recently read word will probably be read again soon.
Terminology

N-bit address

- **TAG field**
  - Indicates the address tag for comparison

- **LINE field**
  - Indicates the proper cache entry

- **WORD field**
  - Indicates which word referenced within a line

- **BYTE field**
  - Indicates a single byte, but not normally used

Direct-Mapped Cache

**Read 0x2464 Causes conflict!**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>Y</td>
<td>0x24</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>7</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
Reverse Polish Notation (RPN)

- **Method to write arithmetic expressions**
  - Avoids the use of brackets to define priorities for evaluation of operators

- **Devised by Jan Lucasiewicz**
  - Polish philosopher and mathematician
  - In his notation, the operators preceded their arguments
  - The “reverse” places operators after arguments

- **For more info**
  - [http://www-stone.ch.cam.ac.uk/documentation/rrf/rpn.html](http://www-stone.ch.cam.ac.uk/documentation/rrf/rpn.html)

---

Basic Program Format

```plaintext
..constant
// all constants declared within .constant and .end-constant
..end-constant

..main

..var
// all variables declared within .var and .end-var
..end-var

// Main program goes here
// Program execution is terminated with a HALT statement
..end-main

..method methodname(parameter_a, parameter_b, etc.)
// all methods declared within .method and .end-method
..var
// local variables for method
..end-var

// method code goes here
// use IRETURN to go back to the main program
..end-method
```
Convert to Assembly and Java Bytecode

**High-level language:**
for (int i = 3; i = = 0; i--)

**Assembly language:**
BIPUSH 3
ISTORE i
L1: ILOAD i
IFEQ L2
IINC i -1
GOTO L1
L2: HALT

<table>
<thead>
<tr>
<th>Address</th>
<th>Bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x10</td>
</tr>
<tr>
<td>1</td>
<td>0x03</td>
</tr>
<tr>
<td>2</td>
<td>0x36</td>
</tr>
<tr>
<td>3</td>
<td>0x01</td>
</tr>
<tr>
<td>4</td>
<td>0x15</td>
</tr>
<tr>
<td>5</td>
<td>0x01</td>
</tr>
<tr>
<td>6</td>
<td>0x99</td>
</tr>
<tr>
<td>7</td>
<td>0x00</td>
</tr>
<tr>
<td>8</td>
<td>0x09</td>
</tr>
<tr>
<td>9</td>
<td>0x84</td>
</tr>
<tr>
<td>A</td>
<td>0x01</td>
</tr>
<tr>
<td>B</td>
<td>0xFF</td>
</tr>
<tr>
<td>C</td>
<td>0xA7</td>
</tr>
<tr>
<td>D</td>
<td>0xFF</td>
</tr>
<tr>
<td>E</td>
<td>0xF8</td>
</tr>
<tr>
<td>F</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

Stepping Through Microinstructions

- Main loop is executed
- Multiway branch to BIPUSH microinstruction sequence