Structured Computer Organization

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 shift maintains the sign
    - 1-bit arithmetic shift left multiplies by 2
    - 1-bit arithmetic shift right divides by 2
  - Logical shift 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

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

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 specified its successor
  - Not executed in order stored in control store
- Accessing the microprogram
  - MicroProgram Counter similar to memory address register
  - MicroInstruction Register similar to memory data register

Microinstruction Notation

- \(B\) field is source register
- \(Mem\) field for initiating memory operation
- \(C\) field sets 0, 1, 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

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 operands can or cannot be in memory?

• Data type and size

• Operations
  – what are supported

• Successor instruction
  – jumps, conditions, branches
  – fetch-decode-execute is implicit!
Instruction Types (from Tanenbaum)

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

IJVM Instruction Set (page 222)

<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>0x19</td>
<td>DUP</td>
<td>Copy top word on stack and push onto stack</td>
</tr>
<tr>
<td>0x17</td>
<td>GOTO offset</td>
<td>Unconditional branch</td>
</tr>
<tr>
<td>0x50</td>
<td>IADD</td>
<td>Pop two words from stack, push their sum</td>
</tr>
<tr>
<td>0x55</td>
<td>IAND</td>
<td>Pop two words from stack, push Boolean AND</td>
</tr>
<tr>
<td>0x99</td>
<td>IFNE offset</td>
<td>Pop word from stack and branch if it is zero</td>
</tr>
<tr>
<td>0x3B</td>
<td>IFLT offset</td>
<td>Pop word from stack and branch if it is less than zero</td>
</tr>
<tr>
<td>0x5F</td>
<td>IF_ICMPEQ offset</td>
<td>Pop two words from stack, branch if equal</td>
</tr>
<tr>
<td>0x84</td>
<td>IINVOKEVIRTUAL</td>
<td>Add a constant to a local variable</td>
</tr>
<tr>
<td>0x15</td>
<td>ILOAD</td>
<td>Push local variable onto stack</td>
</tr>
<tr>
<td>0x18</td>
<td>IVOKEVIRTUAL, dup</td>
<td>Invoke a method</td>
</tr>
<tr>
<td>0x20</td>
<td>IPOP</td>
<td>Pop two words from stack, push Boolean OR</td>
</tr>
<tr>
<td>0x4C</td>
<td>ireturn</td>
<td>Return from method with integer value</td>
</tr>
<tr>
<td>0x36</td>
<td>ISTORE</td>
<td>Pop word from stack and store in local variable</td>
</tr>
<tr>
<td>0x44</td>
<td>ISUB</td>
<td>Pop two words from stack, push their difference</td>
</tr>
<tr>
<td>0x13</td>
<td>LDC, Y index</td>
<td>Push constant from constant pool onto stack</td>
</tr>
<tr>
<td>0x09</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>0x55</td>
<td>SWAP</td>
<td>Swap the top two words on the stack</td>
</tr>
<tr>
<td>0x24</td>
<td>WIDE</td>
<td>Prefix instruction; next instruction has a 16-bit index</td>
</tr>
</tbody>
</table>

Basic ISA Classes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>R4 ← R4+R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>R4 ← R4+3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>R4 ← R4+Mem[R1+100]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>R4 ← R4+Mem[R1]</td>
</tr>
<tr>
<td>Base-Indexed</td>
<td>Add R3,(R1+R2)</td>
<td>R3 ← R3+Mem[R1+R2]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>R1 ← R1+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>R1 ← R1+Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Post-increment</td>
<td>Add R1,(R2)+</td>
<td>R1 ← R1+Mem[R2]; R2 ← R2+d</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>Add R1,−(R2)</td>
<td>R2 ← R2−d; R1 ← R1−Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>R1 ← R1+Mem[R1+100+R2+R3+d]</td>
</tr>
</tbody>
</table>

Addressing Modes

Adapted from John Kubiatowicz’s CS 152 lecture notes. Copyright © 2003 UCB.
Addressing Modes

- **Register**
  - Specify the register where the operand is located

- **Immediate**
  - Operand is specified within the instruction

- **Displacement (indexed)**
  - Addressing memory by giving a register (explicit or implicit) plus a constant offset

- **Register indirect**
  - Register contains a pointer for the memory address of the operand

- **Based-indexed**
  - Adds two registers plus an optional offset

- **Direct or absolute**
  - Give full memory address of operand

- **Stack**
  - Memory referenced with a Last-In First-Out (LIFO) queue

Generic Examples: Instruction Format Widths

<table>
<thead>
<tr>
<th>Variable:</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fixed:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Hybrid:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Modern CPU Design

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

Conventional Pipelined Execution Representation

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

Pipelining is not quite that easy!

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

Three Generic Data Hazards

- Read After Write (RAW)
  Instr_j tries to read operand before Instr_i writes it
  \[
  I: \text{add } r1, r2, r3 \\
  J: \text{sub } r4, r1, r3
  \]

- Caused by a “dependence” (in compiler nomenclature). This hazard results from an actual need for communication.
Three Generic Data Hazards

• Write After Read (WAR)
  Instr$_j$ writes operand \textit{before} Instr$_i$ reads it

  \begin{align*}
  I: & \text{ sub } r4, r1, r3 \\
  J: & \text{ add } r1, r2, r3 \\
  K: & \text{ mul } r6, r1, r7
  \end{align*}

• Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.

• Can’t happen in MIPS 5-stage pipeline because:
  – All instructions take 5 stages, and
  – Reads are always in stage 2, and
  – Writes are always in stage 5

Adapted from John Kubiatowicz’s CS 252 lecture notes. Copyright © 2003 UCB.

Three Generic Data Hazards

• Write After Write (WAW)
  Instr$_j$ writes operand \textit{before} Instr$_i$ writes it.

  \begin{align*}
  I: & \text{ sub } r1, r4, r3 \\
  J: & \text{ add } r1, r2, r3 \\
  K: & \text{ mul } r6, r1, r7
  \end{align*}

• Called an “output dependence” by compiler writers. This also results from the reuse of name “r1”.

• Can’t happen in MIPS 5-stage pipeline because:
  – All instructions take 5 stages, and
  – Writes are always in stage 5

• Will see WAR and WAW in more complicated pipes

Adapted from John Kubiatowicz’s CS 252 lecture notes. Copyright © 2003 UCB.

Control Hazards

• The IFU pipelines with sequential instruction stream in method area

• A branch may indicate another control flow

  \begin{align*}
  \text{ LLOAD } i \\
  \text{ BIPUSH } 3 \\
  \text{ IF_ICMPEQ L1} \\
  \text{ LLOAD } j \\
  \text{ BIPUSH } 1 \\
  \text{ ISTORE } j \\
  \text{ ISUB} \\
  \text{ ISTORE } k \\
  \text{ GOTO } L2 \\
  \text{ L1: } \text{ BIPUSH } 0 \\
  \text{ ISTORE } k \\
  \text{ L2: } \text{ ...}
  \end{align*}

  \textbf{The pipeline issues the next sequential instruction before resolving if the branch is taken or not}

Branches

• Unconditional
  – Must decide where to fetch before knowing what instruction it just got
  – Can insert a “delay slot” after unconditional branch
    • Try to fill it with useful work, but typically a NOP

• Conditional
  – Must test the condition before knowing what instruction to fetch
  – Can stall the pipeline until the branch is resolved
    • Defeats the purpose of pipelining!
Branch Prediction

• **Dynamic prediction**
  – CPU maintains a history of branch locations and their previous behavior
  – Algorithms include always taking backward conditional branches or predicting based on what the branch did the last time

• **Static prediction**
  – Compiler uses branch instructions that specify the branch’s normal outcome (Taken or Not Taken)
  – Program is simulated to profile the branch behavior

Precise Interrupts/Exceptions

• An interrupt or exception is considered **precise** if there is a single instruction (or interrupt point) for which all instructions before that instruction have committed their state and no following instructions including the interrupting instruction have modified any state.
  – This means, effectively, that you can restart execution at the interrupt point and “get the right answer”
  – Implicit in the example below of a device interrupt:
    • Interrupt point is at first lw instruction

Precise Interrupts/Exceptions

• An interrupt or exception is considered **precise** if there is a single instruction (or interrupt point) for which all instructions before that instruction have committed their state and no following instructions including the interrupting instruction have modified any state.
  – This means, effectively, that you can restart execution at the interrupt point and “get the right answer”
  – Implicit in the example below of a device interrupt:
    • Interrupt point is at first lw instruction

ISA Examples

• **IA-32**
  – Pentium II
  – CISC

• **Version 9 SPARC**
  – UltraSPARC II
  – RISC

• **Java Virtual Machine**
  – picoJava II
  – Stack machine

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

<table>
<thead>
<tr>
<th>Parity bits (powers of 2)</th>
<th>Data bits (remaining bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  2  3  4  5  6  7  8</td>
<td>9  10 11 12</td>
</tr>
</tbody>
</table>

Parity Bit Assignment

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

- yes (hit)
  - Line in cache?
    - yes (hit)
      - Read bytes from cache
    - no (miss)
      - 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

- **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

<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>

Read 0x2464

Causes conflict!

---

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

```
.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
// 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
```

**Address Bytecode**

```
<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>0xFFFF</td>
</tr>
<tr>
<td>F</td>
<td>0xFF</td>
</tr>
</tbody>
</table>
```

Stepping Through Microinstructions

1. PC = PC + 1; fetch; goto (MBR)

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

Summary

- Computer Organization has been a GREAT class
- You have been GREAT students
- Final Exam is **cumulative**
  - Don’t forget relevant concepts since Exam 3
- Main ideas
  - Digital logic level
  - Microarchitecture level
  - Instruction Set Architecture (ISA) level
- Study hard and good luck!