Computer Organization
CS 231-01

Exam 3 Review

Dr. William H. Robinson
November 15, 2004

http://eecs.vanderbilt.edu/courses/cs231/

Topics

“If you study to remember, you will forget, but, if you study to understand, you will remember.”
– Unknown

• Review, review, review
  – Ask questions if you have them
  – IJVM
  – Microinstructions
  – Improving performance

• Today’s 50-minute review is NOT comprehensive

Structured Computer Organization

Level 5
Problem-oriented language level
Translation (compiler)

Level 4
Assembly language level
Translation (assembler)

Level 3
Operating system/machine level
Partial interpretation (operating systems)

Level 2
Instruction set architecture level
Interpretation (microprogram) or direct execution

Level 1
Microarchitecture level
Hardware

Level 0
Digital logic level

• Digital logic builds microarchitecture
• Microarchitecture implements the ISA
• ISA is in machine language
• Assembly language allows us to use ISA

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)
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>0x18</td>
<td>IADD</td>
<td>Pop two words from stack; push their sum</td>
</tr>
<tr>
<td>0x16</td>
<td>INVOKEVIRTUAL</td>
<td>Invoke a method</td>
</tr>
<tr>
<td>0x19</td>
<td>IOR</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>0x35</td>
<td>ISTORE varnum</td>
<td>Pop word from stack and store in local variable</td>
</tr>
<tr>
<td>0x15</td>
<td>LOAD varnum</td>
<td>Push local variable onto stack</td>
</tr>
<tr>
<td>0x3A</td>
<td>NOP</td>
<td>Do nothing</td>
</tr>
<tr>
<td>0x3F</td>
<td>POP</td>
<td>Delete word on top of stack</td>
</tr>
<tr>
<td>0x68</td>
<td>SWAP</td>
<td>Swap the two top words on the stack</td>
</tr>
<tr>
<td>0x44</td>
<td>WIDE</td>
<td>Prefix instruction; next instruction has a 16-bit index</td>
</tr>
</tbody>
</table>

Using Offset

- **Operand of all branch instructions**
  - GOTO, IFEQ, IFLT, IF_ICMPEQ

- **Changes the value of PC**
  - 16-bit signed value
    - Use 2's complement notation
    - First byte fetched from method area is the "high" byte
    - Second byte fetched from method area is the "low" byte
  
  - Added to the PC value at the start of the instruction
    - Not the value after fetching the 2 offset bytes

Java Bytecode (page 226)

```java
i = j + k;  // 1
1  ILOAD j  // i = j + k  0x15 0x02
if (i == 3)  // 2
2  ILOAD k  0x15 0x03
3  IADD  0x00 0x02
4  ISTORE i  0x36 0x01
j = j - 1;  // 5
5  ILOAD i  // j = j - 1  0x15 0x01
6  BIPUSH 3  0x10 0x03
7  IF_ICMPEQ L1  0x10 0x00 0x0D
8  ILOAD j  // j = j - 1  0x15 0x02
9  BIPUSH 1  0x10 0x01
10  ISUB  0x06 0x14
11  ISTORE j  0x36 0x02
12  GOTO L2  0xA7 0x00 0x0D7
13  L1  0x00 0x00 0x00 0x00
14  L2  0x36 0x03
```

Assume:

- `i` is varnum 1
- `j` is varnum 2
- `k` is varnum 3

Figure 4.14. (a) A Java fragment. (b) The corresponding Java assembly language. (c) The JVM program in hexadecimal.
Parts of IJVM Memory

- CPP, LV, and SP registers are pointers to words
- PC contains a byte address

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

Assignment and ALU Operations

- Possible sources
  - Connected to B bus
  - MDR, PC, MBR, MBRU, SP, LV, CPP, TOS, or OPC
- Possible destinations
  - Connected to C bus
  - MAR, MDR, PC, SP, LV, CPP, TOS, OPC, or H

Figure 4.16. All permitted operations. Any of the above operations may be extended by adding “ shifting” to them to shift the results left by 1 byte. For example, a common operation is: $H = MBR << 1$.
Assignment and ALU Operations

<table>
<thead>
<tr>
<th>F&lt;sub&gt;x&lt;/sub&gt;</th>
<th>F&lt;sub&gt;y&lt;/sub&gt;</th>
<th>ENA</th>
<th>ENB</th>
<th>INVA</th>
<th>INC</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A + B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A * B + 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>B + 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>B - A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A AND B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A OR B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Memory Access

- **rd, wr**
  - Uses MAR/MDR to access constant pool, current local variable frame and current operand stack

- **fetch**
  - Uses PC/MBR to access method area

- **Initiated at the end of the cycle**
  - After the C bus is valid

- **Data availability for rd and fetch**
  - At the end of the next cycle

Branching

- **Unconditional**
  - goto label
  - Can explicitly name a successor for unconditional branch

- **Conditional**
  - Use ALU flags Z and N
    - Program Status Word
    - Set according to the result of the ALU operation
    - Ex: Z = TOS
      - If (Z) goto L1; else L2
      - If (N) goto L1; else L2

- **Multiway branch for next opcode**
  - goto (MBR OR value)

Determining Next Microinstruction

- **When JAM bits are all zeros**
  - NEXT_ADDRESS field is the next microinstruction

- **When JAMN or JAMZ are set**
  - The value of NEXT_ADDRESS
  - The value of NEXT_ADDRESS with high-order bit ORed with 1

- **WHEN JMPC is set**
  - MBR is bitwise ORed with 8 low-order bits of NEXT_ADDRESS field
Stepping Through Microinstructions

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Start of cycle</th>
<th>End of cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x10</td>
<td>600</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>0x07</td>
<td>600</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>0x36</td>
<td>600</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>0x01</td>
<td>600</td>
<td>100</td>
</tr>
</tbody>
</table>

- Stack Pointer and Memory Address Register updated
- Next byte is fetched into Memory Buffer Register

Improving Speed with Architecture Design

**Three basic approaches**

- Reduce number of clock cycles needed to execute an instruction
- Simplify organization to shorten clock cycle
- Overlap execution of instructions
  - Finding “independent” operations

Generally have some serial operations that affect cycle time

Datapath of Mic-2 Microarchitecture

- **Three-bus architecture**
  - Possible to add any register to any other register in a single cycle
- **Instruction Fetch Unit (IFU)**
  - Efficiently fetch and process instruction stream
- **Merge interpreter Main loop with microcode (Fig 4-30)**
  - Fully utilize instruction

Datapath of Mic-3 Microarchitecture

- **Add latches on A bus, B bus, and C bus**
- **Benefits**
  - Speed up the clock cycle because maximum delay is shorter
  - Use all parts of datapath during every cycle
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

Classic Five-Stage Pipeline

- **Ifetch**: Instruction Fetch
  – Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Perform calculation for instruction
- **Mem**: Access the data from the Data Memory
- **Wr**: Write the data back to the register file

Visualizing Pipelining

- 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: add r1,r2,r3
  J: sub r4,r1,r3

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

- **Write After Read (WAR)**
  Instr$_j$ writes operand before Instr$_i$ reads it.

  I: sub r4,r1,r3
  J: add r1,r2,r3
  K: mul r6,r1,r7

- 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

- Will see WAR and WAW in more complicated pipes.

Three Generic Data Hazards

- **Write After Write (WAW)**
  Instr$_j$ writes operand before Instr$_i$ writes it.

  I: sub r1,r4,r3
  J: add r1,r2,r3
  K: mul r6,r1,r7

- 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.

Control Hazards

- The IFU pipelines with sequential instruction stream in method area.

- A branch may indicate another control flow:
  - 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

In-Order Execution

• **Issue instructions in program order**

• **Retire (complete) instructions in program order**

• **May not give optimal performance because of instruction dependencies**
  – Created whenever there is a dependence between instructions where pipelining overlap changes the order of access
    • RAW
    • WAW
    • WAR

Scoreboarding: O-O-O Issue/Completion

<table>
<thead>
<tr>
<th>Cy</th>
<th>Decoded</th>
<th>Iss.</th>
<th>Ret.</th>
<th>Registers being read</th>
<th>Registers being written</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R1=R2</td>
<td>0</td>
<td>1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>2</td>
<td>R2=R3</td>
<td>0</td>
<td>1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>3</td>
<td>R3=R4</td>
<td>0</td>
<td>1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>4</td>
<td>R4=R5</td>
<td>0</td>
<td>1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>5</td>
<td>R5=R6</td>
<td>0</td>
<td>1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>6</td>
<td>R6=R7</td>
<td>0</td>
<td>1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>7</td>
<td>R7=R8</td>
<td>0</td>
<td>1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>8</td>
<td>R8=R9</td>
<td>0</td>
<td>1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>9</td>
<td>R9=R10</td>
<td>0</td>
<td>1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

*Figure 4-44.* Operation of a superscalar CPU with out-of-order issues and in-order completion.
Precise Interrupt

- Need the capability to store the CPU state
- With out-of-order completion
  - If an interrupt occurred, it would be difficult to save current state
  - Not possible to say all instructions up to some address had been executed and all instructions beyond it had not
- In-order completion ensures precise interrupts

Example Microarchitectures

- IA-32 (Pentium II)
  - CISC
- Version 9 SPARC (UltraSPARC II)
  - RISC
- Java Virtual Machine (picoJava II)
  - Stack machine

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

Example:

```
add   r1, r2, r3
subi  r4, r1, #4
slli  r4, r4, #2
lw    r2, 0(r4)
lw    r3, 4(r4)
add   r2, r2, r3
sw    8(r4), r2
```

External Interrupt
PC saved
Disable All Ints
Supervisor Mode
Restore PC
User Mode
Int handler

Summary

- Computer Organization is still a GREAT class
- Dr. Robinson is still a GREAT professor
- Questions focus on material since Exam 2
  - Don’t forget relevant concepts from previous exams
- Main ideas
  - High-level language ↔ JVM assembly ↔ JVM bytecode
  - Mic-1 Microarchitecture interprets JVM using microinstructions
  - Hazards reduce architectural performance
- Study hard and good luck!