Topics

- Review Sheet
- Microarchitecture
- IJVM ISA

Block Diagram of Mic-1 Microarchitecture

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

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

**Control Section**
Part of CPU containing the H/W necessary to direct the datapath
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>0x59</td>
<td>DUP</td>
<td>Copy top word on stack and push onto stack</td>
</tr>
<tr>
<td>0xA7</td>
<td>GOTO offset</td>
<td>Unconditional branch</td>
</tr>
<tr>
<td>0x89</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>IFEQ offset</td>
<td>Pop word from stack and branch if it is zero</td>
</tr>
<tr>
<td>0x9F</td>
<td>IF, JCMP EQ offset</td>
<td>Pop two words from stack; branch if equal</td>
</tr>
<tr>
<td>0x84</td>
<td>INCR 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>0x86</td>
<td>INVOCVIRTUAL disp</td>
<td>Invoke a method</td>
</tr>
<tr>
<td>0x8D</td>
<td>IOR</td>
<td>Pop two words from stack; push Boolean OR</td>
</tr>
<tr>
<td>0x8C</td>
<td>IRETURN</td>
<td>Return from method with integer value</td>
</tr>
<tr>
<td>0x8E</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>0x00</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 IJVM instruction set. The operands byte, const, and varnum are 1 byte. The operands disp, index, and offset are 2 bytes.

Parts of IJVM Memory

- CPP, LV, and SP registers are pointers to **words**
- PC contains a **byte address**
Mic-1 Simulator Window

- **Microinstruction** – current microinstruction being executed
- **NextMicroinstruction** – next microinstruction to be executed
- **MPC** – current location in the control store
- **Registers** – value stored in each microarchitecture register

Mic-1 Simulator Special Instructions

- **HALT**
  - Instruction used to terminate a program

- **IN**
  - Reads a character from the key buffer
  - Pushes its **ASCII** value onto the stack
  - If no key has been pressed, zero will be pushed onto the stack

- **OUT**
  - Pops a word off the stack
  - Prints it to the standard output text area
  - Can only output **ASCII** values

The ASCII character set is found in Fig. 2-41 on p. 110 of your text
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

---

Basic Program Format

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

.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
```
Conversion to Java 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>0xF8</td>
</tr>
<tr>
<td>F</td>
<td>0xFF</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
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 "<< 8" to them to shift the result left by 1 byte. For example, a common operation is $H = MBR << 8$
Assignment and ALU Operations

Invalid Operations
- MDR = SP + MDR
- H = H - MDR
- SP = H + MAR
- Etc.

Multiple assignments
- SP = MDR = SP + 1
- MDR = TOS = MBR
- Etc.

Assignment and ALU Operations

<table>
<thead>
<tr>
<th>F6</th>
<th>F5</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>1</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>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A + B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A + B + 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A + 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>B - 1</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>0</td>
<td>A OR B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

DEST = H
DEST = SOURCE
DEST = H
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE - H
DEST = SOURCE - 1
DEST = -H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = -1
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**)
Stepping Through Microinstructions

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction: PC = PC + 1; fetch; goto (MBR)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Start of cycle</td>
</tr>
<tr>
<td></td>
<td>Method Area</td>
</tr>
<tr>
<td>MAR</td>
<td></td>
</tr>
<tr>
<td>MDR</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>1</td>
</tr>
<tr>
<td>MBR</td>
<td>0x10</td>
</tr>
<tr>
<td>SP</td>
<td>4</td>
</tr>
<tr>
<td>LV</td>
<td>600</td>
</tr>
<tr>
<td>CPP</td>
<td>100</td>
</tr>
<tr>
<td>TOS</td>
<td>102</td>
</tr>
<tr>
<td>OPC</td>
<td>101</td>
</tr>
<tr>
<td>H</td>
<td>100</td>
</tr>
</tbody>
</table>

- Main loop is executed
- Multiway branch to BIPUSH microinstruction sequence
Stepping Through Microinstructions

**Step 2**

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

**Step 3**

- Program Counter is updated
- Initiate a fetch operation
Stepping Through Microinstructions

1. Operand is stored in Top Of Stack and Memory Data Register
2. Next byte is fetched into Memory Buffer Register

Stepping Through Microinstructions

1. Main loop is executed
2. Multiway branch to ISTORE microinstruction sequence
### 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></td>
<td></td>
<td>MAR = MBRU + H</td>
<td></td>
</tr>
</tbody>
</table>

- **Copy LV pointer into Holding register**
- **Next byte is fetched into Memory Buffer Register**

---

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Start of cycle</th>
<th>End of cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>H = LV</td>
<td></td>
</tr>
</tbody>
</table>

---

- **Computer ISTORE address**
Stepping Through Microinstructions

- Copy TOS into Memory Data Register
- Initiate memory write

Stepping Through Microinstructions

- Decrement Stack Pointer
- Initiate a read for new TOS value
Stepping Through Microinstructions

- Program Counter is updated
- Initiate a fetch operation

Stepping Through Microinstructions

- Copy new TOS value
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

Adapted from David Patterson’s CS 252 lecture notes. Copyright 2001 UCB.
Classic Five-Stage Pipeline

- **I-fetch**: 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

Conventional Pipelined Execution Representation

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

• Created whenever there is a dependence between instructions where pipelining overlap changes the order of access

• True dependence: Read After Write (RAW)
  – Instructions are dependent upon previous data results that aren’t available
  – Causes stalls in the pipeline to wait for data

• Other hazard types include WAW and WAR
  – Can be handled with register renaming
  – Note that RAR is not a hazard

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

---

2-Bit Branch Prediction FSM

```
Strongly predicts taken | Weakly predicts taken
---|---
Strongly predicts taken |
Weakly predicts not taken |
Predict taken |
Taken |
Not taken |
Predict not taken |
Taken |
Not taken |
Predict taken |
Taken |
Not taken |
Predict not taken |
Taken |
Not taken |
Strongly predicts not taken |
```

37

38
Static Branch Prediction

• Compiler uses branch instructions that specify the branch’s normal outcome
  – Based upon the code to be compiled
  – Taken or Not Taken

• Program is simulated to profile the branch behavior
  – Branch statistics are given to the compiler