Sequential Implementation

CMPU 224 – Computer Organization
Jason Waterman
## Y86-64 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Byte</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2 fn rA rB</td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3 0 F rB V</td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4 0 rA rB D</td>
<td></td>
</tr>
<tr>
<td>mrmmovq D(rB), rA</td>
<td>5 0 rA rB D</td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6 fn rA rB</td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7 fn Dest</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0 Dest</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A 0 rA F</td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B 0 rA F</td>
<td></td>
</tr>
</tbody>
</table>
Building Blocks

- **Combinational Logic**
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control

- **Storage Elements**
  - Store bits
  - Registers
  - Addressable memories
  - Loaded only as clock rises
Hardware Control Language

• Very simple hardware description language
  • Can only express limited aspects of hardware operation
    • Parts we want to explore and modify
  • Boolean operations have syntax similar to C logical operations
  • We’ll use it to describe control logic for processors

• Data Types
  • bool: Boolean
    • a, b, c, ...
  • int: words
    • A, B, C, ...
    • Does not specify word size---bytes, 64-bit words, ...

• Statements
  • bool a = bool-expr ;
  • int A = int-expr ;

4/14/2021
HCL Operations

• Classify by type of value returned

• Boolean Expressions – evaluate to a Boolean
  • Logic Operations
    • a && b, a || b, !a
  • Word Comparisons
    • A == B, A != B, A < B, A <= B, A >= B, A > B
  • Set Membership
    • A in { B, C, D }
      • Same as A == B || A == C || A == D

• Word Expressions
  • Case expressions
    • [ a : A; b : B; c : C ]
    • Evaluate test expressions a, b, c, ... in sequence
    • Return word expression A, B, C, ... for first successful test
SEQ Hardware Structure

• State
  • Program counter register (PC)
  • Condition code register (CC)
    • ZF: Zero
    • SF: Negative
    • OF: Overflow
• Register File
• Memories
  • Access same memory space
  • Data: for reading/writing program data
  • Instruction: for reading instructions
SEQ Hardware Structure

- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter
SEQ Stages

- Fetch
  - Read an instruction from Instruction Memory
- Decode
  - Gets values for the operands rA and rB
- Execute
  - Operation or address calculation
  - Sets Condition Codes
- Memory
  - Read or write memory
- Write Back
  - Update registers
- PC
  - Update program counter with next instruction address
Instruction Format

• Instruction Format
  • Instruction byte $icode : ifun$
  • Optional register byte $rA : rB$
  • Optional constant word $valC$
SEQ Stages -- Fetch

- Fetch
  - Read an instruction from Instruction Memory
  - Reads the bytes of an instruction from memory, using the Program Counter (PC) as the memory address
  - Extracts the $icode$ and $ifun$ values from the instruction
  - Optionally extracts register operand specifiers $rA$ and $rB$
  - Optionally extracts 8-byte constant word $valC$
  - Computes the address of the instruction following the current one as $valP$ (PC + length of the fetched instruction)
SEQ Stages -- Decode

• Decode
  • Reads up to two operands from the register file, giving values \texttt{valA} and/or \texttt{valB}
  • Typically reads registers designated by \texttt{rA} and \texttt{rB}
  • For some instructions it reads register \%\texttt{rsp}
    • Which ones?
    • \texttt{push, pop, call, ret}

4/14/2021
SEQ Stages -- Execute

- **Execute**
  - Compute value (math or logic)
    - Condition codes are possibly set
  - Compute memory address
    - `rmmovq rA, D(rB)`
  - Also handles conditional jumps and moves
  - **valE**, the value or address computed
SEQ Stages -- Memory

- Memory
  - Either reads or writes data to memory
  - valM, the data read from memory
SEQ Stages -- Write Back

- **Write Back**
  - Writes up to two results to the register file: valE, valM
  - Why would you need to update two registers?
    - popq %rax
    - Need to update %rax and %rsp
SEQ Stages -- PC

- PC
  - Update program counter to the address of the next instruction
Executing Arithmetic/Logical Operation

- **Fetch**
  - Read 2 bytes
- **Decode**
  - Read operand registers: rA, rB
- **Execute**
  - Perform operation
  - Set condition codes
- **Memory**
  - Do nothing
- **Write back**
  - Update register: rB
- **PC Update**
  - Increment PC by 2

\[
\text{OPq rA, rB} \quad 6 \quad \text{fn} \quad \text{rA rB}
\]
### Stage Computation: Arith/Log Ops

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fetch</strong></td>
<td><code>icode:ifun ← M_1[PC]</code>&lt;br&gt;<code>rA:rB ← M_1[PC+1]</code>&lt;br&gt;<code>valP ← PC+2</code></td>
</tr>
<tr>
<td><strong>Decode</strong></td>
<td><code>valA ← R[rA]</code>&lt;br&gt;<code>valB ← R[rB]</code></td>
</tr>
<tr>
<td><strong>Execute</strong></td>
<td><code>valE ← valB OP valA</code>&lt;br&gt;Set CC</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Write</strong></td>
<td><code>R[rB] ← valE</code></td>
</tr>
<tr>
<td><strong>back</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PC update</strong></td>
<td><code>PC ← valP</code></td>
</tr>
</tbody>
</table>

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions
Executing rmmovq

- Fetch
  - Read 10 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
- Write back
  - Do nothing
- PC Update
  - Increment PC by 10
Stage Computation: `rmmovq`

<table>
<thead>
<tr>
<th>Stage</th>
<th>Operation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td><code>rmmovq rA, D(rB)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>icode:ifun ← M₁[PC]</code></td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td><code>rA:rB ← M₁[PC+1]</code></td>
<td>Read register byte</td>
</tr>
<tr>
<td></td>
<td><code>valC ← M₈[PC+2]</code></td>
<td>Read displacement D</td>
</tr>
<tr>
<td></td>
<td><code>valP ← PC+10</code></td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td><code>valA ← R[rA]</code></td>
<td>Read operand A</td>
</tr>
<tr>
<td></td>
<td><code>valB ← R[rB]</code></td>
<td>Read operand B</td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + valC</code></td>
<td>Compute effective address</td>
</tr>
<tr>
<td>Memory</td>
<td><code>M₈[valE] ← valA</code></td>
<td>Write value to memory</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td><code>PC ← valP</code></td>
<td>Update PC</td>
</tr>
</tbody>
</table>

- Use ALU for address computation
Executing popq

- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer (%rsp)
- Execute
  - Increment stack pointer by 8
- Memory
  - Read value at address from old stack pointer
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2
Stage Computation: popq

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fetch</strong></td>
<td>iode:ifun ← M_{1}[PC] rA:rB ← M_{1}[PC+1]</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+2</td>
<td>Read register byte</td>
</tr>
<tr>
<td><strong>Decode</strong></td>
<td>valA ← R[%rsp] valB ← R[%rsp]</td>
<td>Compute next PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read stack pointer</td>
</tr>
<tr>
<td><strong>Execute</strong></td>
<td>valE ← valB + 8</td>
<td>Read stack pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>valM ← M_{8}[valA]</td>
<td>Read from stack</td>
</tr>
<tr>
<td><strong>Write</strong></td>
<td>R[%rsp] ← valE</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td></td>
<td>R[rA] ← valM</td>
<td>Write back result</td>
</tr>
<tr>
<td><strong>PC update</strong></td>
<td>PC ← valP</td>
<td>Update PC</td>
</tr>
</tbody>
</table>

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer
Executing Conditional Moves

- **Fetch**
  - Read 2 bytes

- **Decode**
  - Read operand registers

- **Execute**
  - If !cnd, then set destination register to 0xF

- **Memory**
  - Do nothing

- **Write back**
  - Update register (or not)

- **PC Update**
  - Increment PC by 2

![Conditional Move Diagram](image)
### Stage Computation: Cond. Move

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td><code>cmovXX rA, rB</code></td>
<td>Read instruction byte, Read register byte, Compute next PC</td>
</tr>
<tr>
<td></td>
<td><code>icode:ifun ← M_1[PC]</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>rA:rB ← M_1[PC+1]</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>valP ← PC+2</code></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td><code>valA ← R[rA]</code></td>
<td>Read operand A</td>
</tr>
<tr>
<td></td>
<td><code>valB ← 0</code></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + valA</code></td>
<td>Pass valA through ALU (Disable register update)</td>
</tr>
<tr>
<td></td>
<td><code>If ! Cond(CC,ifun) rB ← 0xF</code></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td><code>R[rB] ← valE</code></td>
<td>Write back result</td>
</tr>
<tr>
<td>Write back</td>
<td><code>PC ← valP</code></td>
<td>Update PC</td>
</tr>
</tbody>
</table>

- Read register rA and pass through ALU
- Cancel move by setting destination register to 0xF
  - If condition codes & move condition indicate no move
Executing Jumps

- **Fetch**
  - Read 9 bytes
  - Increment PC by 9
- **Decode**
  - Do nothing
- **Execute**
  - Determine whether to take branch based on jump condition and condition codes
- **Memory**
  - Do nothing
- **Write back**
  - Do nothing
- **PC Update**
  - Set PC to Dest if branch taken or to incremented PC if not branch
Stage Computation: Jumps

• Compute both addresses
• Choose based on setting of condition codes and branch condition

<table>
<thead>
<tr>
<th>Stage</th>
<th>Operation</th>
</tr>
</thead>
</table>
| Fetch | icode:ifun $\leftarrow M_{1}[PC]$  
        |          
        | valC $\leftarrow M_{8}[PC+1]$  
        | valP $\leftarrow PC+9$  
| Decode|           |
| Execute| Cnd $\leftarrow \text{Cond}(CC,\text{ifun})$  
        |           |
| Memory|           |
| Write |           |
| back  |           |
| PC update | PC $\leftarrow \text{Cnd ? valC : valP}$  

- Read instruction byte
- Read destination address
- Fall through address
- Take branch?
- Update PC
Executing call

- **Fetch**
  - Read 9 bytes
  - Increment PC by 9

- **Decode**
  - Read stack pointer

- **Execute**
  - Decrement stack pointer by 8

- **Memory**
  - Write incremented PC to new value of stack pointer

- **Write back**
  - Update stack pointer

- **PC Update**
  - Set PC to Dest
Stage Computation: call

- Use ALU to decrement stack pointer
- Store incremented PC
Executing ret

- Fetch
  - Read 1 byte
- Decode
  - Read %rsp
- Execute
  - Calculate %rsp + 8
- Memory
  - Read return address M[%rsp]
- Write back
  - Update %rsp
- PC Update
  - Set PC to return address
Stage Computation: \texttt{ret}

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>\texttt{icode:ifun} $\leftarrow$ ( M_{1}[PC] )</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>Decode</td>
<td>\texttt{valA} $\leftarrow$ ( R[%rsp] ) \texttt{valB} $\leftarrow$ ( R[%rsp] )</td>
<td>Read operand stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td>\texttt{valE} $\leftarrow$ ( \texttt{valB} + 8 )</td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td>\texttt{valM} $\leftarrow$ ( M_{8}[\texttt{valA}] )</td>
<td>Read return address</td>
</tr>
<tr>
<td>Write</td>
<td>( R[%rsp] ) $\leftarrow$ \texttt{valE}</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>( \texttt{PC} \leftarrow \texttt{valM} )</td>
<td>Set PC to return address</td>
</tr>
</tbody>
</table>

- Use ALU to increment stack pointer
- Read return address from memory
### Computation Steps

All instructions follow same general pattern
- Differ in what gets computed on each step

<table>
<thead>
<tr>
<th>Fetch</th>
<th>OPq rA, rB</th>
</tr>
</thead>
<tbody>
<tr>
<td>rA, rB</td>
<td>rA::rB ← M₁[PC+1]</td>
</tr>
<tr>
<td>valC</td>
<td>valP ← PC+2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode</th>
<th>Read instruction byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>valA, srcA</td>
<td>Read operand A</td>
</tr>
<tr>
<td>valB, srcB</td>
<td>Read operand B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execute</th>
<th>Perform ALU operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>valE</td>
<td>valE ← valB OP valA</td>
</tr>
<tr>
<td>Cond code</td>
<td>Set CC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>[Memory read/write]</th>
</tr>
</thead>
<tbody>
<tr>
<td>valM</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write back</th>
<th>Write back ALU result</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstE</td>
<td>R[rB] ← valE</td>
</tr>
<tr>
<td>dstM</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC update</th>
<th>Update PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>

- Read instruction byte
- Read register byte
- [Read constant word]
- Compute next PC
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- [Memory read/write]
- Write back ALU result
- [Write back memory result]
- Update PC
## Computation Steps

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write</th>
<th>PC update</th>
</tr>
</thead>
<tbody>
<tr>
<td>iCode,ifun</td>
<td>valA, srcA</td>
<td>valE</td>
<td>valM</td>
<td>dstE &amp; dstM</td>
<td>PC</td>
</tr>
<tr>
<td>rA,rB</td>
<td>valB, srcB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>valC</td>
<td>valE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>valP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Call Dest

- **Read instruction byte**
- **Read register byte**
- **Read constant word**
- **Compute next PC**
- **Read operand B**
- **Perform ALU operation**
- **[Set /use cond. code reg]**
- **Memory read/write**
- **Write back ALU result**
- **[Write back memory result]**
- **Update PC**

### Instructions

- **Call Dest**

### Computation Details

- **Fetch**
  - Read instruction byte
  - [Read register byte]
  - Read constant word
  - Compute next PC

- **Decode**
  - Read operand A
  - Read operand B

- **Execute**
  - Perform ALU operation
  - [Set /use cond. code reg]

- **Memory**
  - Memory read/write

- **Write**
  - Write back ALU result
  - [Write back memory result]

- **PC update**
  - Update PC

### Notes

- All instructions follow same general pattern
- Differ in what gets computed on each step
### Computed Values

#### Fetch
- **icode**: Instruction code
- **ifun**: Instruction function
- **rA**: Instr. Register A
- **rB**: Instr. Register B
- **valC**: Instruction constant
- **valP**: Incremented PC

#### Decode
- **srcA**: Register ID A
- **srcB**: Register ID B
- **dstE**: Destination Register E
- **dstM**: Destination Register M
- **valA**: Register value A
- **valB**: Register value B

#### Execute
- **valE**: ALU result
- **Cnd**: Branch/move flag

#### Memory
- **valM**: Value from memory