Sequential Implementation

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

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

11/7/2023

CMPU 224 -- Computer Organization
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 ;`
HCL Operations

- Classify by type of value returned

- Boolean Expressions
  - Logic Operations
    - $a \land b$, $a \lor b$, $\neg a$
  - Word Comparisons
    - $A == B$, $A \neq B$, $A < B$, $A \leq B$, $A \geq B$, $A > B$
  - Set Membership
    - $A \text{ in \{ B, C, D \}}$
      - Same as $A == B \lor A == C \lor A == D$

- 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**
  - 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
  - Reads the bytes of an instruction from Instruction Memory
    - Uses the value in the Program Counter (PC) as the memory address to read
  - 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 `valA` and/or `valB`
  - Typically reads registers designated by `rA` (srcA) and `rB` (srcB)
  - For some instructions it also reads register `%rsp`
    - Which ones?
    - `push`, `pop`, `call`, `ret`
SEQ Stages -- Execute

- **Execute**
  - Compute a value (math or logic)
    - Condition codes are set
  - Compute memory address
    - `rmmovq rA, D(rB)`
  - Also handles jumps and conditional 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

11/7/2023  CMPU 224 -- Computer Organization
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
Stage Computation: Arith/Log Ops

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Fetch | Read instruction byte | Read instruction byte
|       | Read register byte | Read register byte |
|       | Compute next PC | Compute next PC |
| Decode| Read operand A | Read operand A
|       | Read operand B | Read operand B |
| Execute| Perform ALU operation | Perform ALU operation |
|        | Set condition code register | Set condition code register |
| Memory | Write back result | Write back result |
| Write back | Update PC | Update PC |

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

- 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>Actions</th>
</tr>
</thead>
</table>
| Fetch     | `icode:ifun ← M_1[PC]`
|           | `rA:rB ← M_1[PC+1]`                                                      |
|           | `valP ← PC+2`                                                           |
| Decode    | `valA ← R[%rsp]`                                                        |
|           | `valB ← R[%rsp]`                                                        |
| Execute   | `valE ← valB + 8`                                                       |
| Memory    | `valM ← M_8[valA]`                                                      |
| Write     | `R[%rsp] ← valE`                                                        |
| Back      | `R[rA] ← valM`                                                          |
| PC update | `PC ← valP`                                                              |

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

Read instruction byte
Read register byte
Compute next PC
Read stack pointer
Read stack pointer
Increment stack pointer
Read from stack
Update stack pointer
Write back result
Update PC
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

![Diagram of Execute stage](image-url)
## Stage Computation: Cond. Move

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read instruction byte</td>
<td><code>icode:ifun ← M_1[PC]</code></td>
</tr>
<tr>
<td></td>
<td>Read register byte</td>
<td><code>rA:rB ← M_1[PC+1]</code></td>
</tr>
<tr>
<td></td>
<td>Compute next PC</td>
<td><code>valP ← PC+2</code></td>
</tr>
<tr>
<td>Decode</td>
<td>Read operand A</td>
<td><code>valA ← R[rA]</code></td>
</tr>
<tr>
<td></td>
<td>Pass valA through ALU</td>
<td><code>valB ← 0</code></td>
</tr>
<tr>
<td>Execute</td>
<td>Pass valA through ALU (Disable register update)</td>
<td><code>valE ← valB + valA</code></td>
</tr>
<tr>
<td></td>
<td>If ! Cond(CC,ifun) rB ← 0xF</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>Write back result</td>
<td><code>R[rB] ← valE</code></td>
</tr>
<tr>
<td>Write back</td>
<td>Update PC</td>
<td><code>PC ← valP</code></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 $\leftarrow PC+9$ | Read instruction byte  
Read destination address  
Fall through address |
| Decode | | |
| Execute | Cnd $\leftarrow$ Cond(CC,ifun) | Take branch? |
| Memory | | |
| Write back | | |
| PC update | PC $\leftarrow$ Cnd ? valC : valP | Update PC |
### Executing call

<table>
<thead>
<tr>
<th>call Dest</th>
<th>8 0 Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>return:</td>
<td>XX XX</td>
</tr>
<tr>
<td>target:</td>
<td>XX XX</td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Fetch | `call Dest` | **call** Dest  
  
  **icode:**ifun ← $M_{1}[PC]$  
  
  valC ← $M_{8}[PC+1]$  
  
  valP ← PC+9 |
| Decode | valB ← R[$\%rsp$] | Read stack pointer  
  
  Decrement stack pointer |
| Execute | valE ← valB + –8 | Read destination address  
  
  Compute return point |
| Memory | $M_{8}[valE] ← valP$ | Write return value on stack |
| Write back | R[$\%rsp$] ← valE | Update stack pointer |
| PC update | PC ← valC | Set PC to destination |

- 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: `ret`

<table>
<thead>
<tr>
<th>Stage</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td><code>icode:ifun ← M_1[PC]</code></td>
</tr>
</tbody>
</table>
| **Decode** | `valA ← R[%rsp]`  
            | `valB ← R[%rsp]`                                                     |
| **Execute** | `valE ← valB + 8`                                                   |
| Memory  | `valM ← M_8[valA]`                                                     |
| **Write** | `R[%rsp] ← valE`                                                     |
| **back** |                                                                         |
| PC update | `PC ← valM`                                                             |

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

#### Table:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td>Read register byte</td>
</tr>
<tr>
<td></td>
<td>[Read constant word]</td>
</tr>
<tr>
<td></td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>Read operand A</td>
</tr>
<tr>
<td></td>
<td>Read operand B</td>
</tr>
<tr>
<td></td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td></td>
<td>Set/use cond. code reg</td>
</tr>
<tr>
<td></td>
<td>[Memory read/write]</td>
</tr>
<tr>
<td></td>
<td>Write back ALU result</td>
</tr>
<tr>
<td></td>
<td>[Write back memory result]</td>
</tr>
<tr>
<td></td>
<td>Update PC</td>
</tr>
</tbody>
</table>

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

| Fetch | | | | | | 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 |

### Call Dest

| icode,ifun | icode,ifun ← M₁(PC) | Read instruction byte |
| rA,rB | | [Read register byte] |
| valC | valC ← M₈(PC+1) | Read constant word |
| valP | valP ← PC+9 | Compute next PC |
| valA, srcA | valB ← R[%rsp] | [Read operand A] |
| valB, srcB | | Read operand B |
| valE | valE ← valB + –8 | Perform ALU operation |
| Cond code | | [Set /use cond. code reg] |
| valM | M₈(valE) ← valP | Memory read/write |
| Write | R[%rsp] ← valE | Write back ALU result |
| dstE | | [Write back memory result] |
| dstM | | |
| PC update | PC ← valC | Update PC |

- 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