Sequential Implementation

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Byte</th>
<th>Destination(s)</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</td>
<td>rA rB</td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3 0</td>
<td>F rB V</td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4 0</td>
<td>rA rB D</td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5 0</td>
<td>rA rB D</td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6 fn</td>
<td>rA rB</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7 fn</td>
<td>Dest</td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0</td>
<td>Dest</td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A 0</td>
<td>rA F</td>
</tr>
<tr>
<td>popq rA</td>
<td>B 0</td>
<td>rA F</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 ;`
HCL Operations

• Classify by type of value returned

• Boolean Expressions
  • 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
  • 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

ONE CLOCK CYCLE

Diagram of instruction flow through stages:

- Fetch: Instruction memory, PC increment
- Decode: iCode, ifun, rA, rB, valC
- Memory: Addr, Data, valE, valM
- Write back: newPC, valE, valM

Diagram notation:
- PC
- Instruction memory
- ALU
- Register file
- Data memory
- Addr, Data
- PC increment
- iCode, ifun, rA, rB, valC
- Cnd, aluA, aluB
- srcA, srcB, dstE, dstM
- valA, valB, valE, valM
- newPC

4/5/2023

CMPU 224 -- Computer Organization
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 \texttt{icode:ifun}
  - Optional register byte \texttt{rA:rB}
  - Optional constant word \texttt{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 \textbf{valA} and/or \textbf{valB}
  • Typically reads registers designated by \texttt{rA} (srcA) and \texttt{rB} (srcB)
  • For some instructions it also reads register \texttt{\%rsp}
    • Which ones?
    • \texttt{push}, \texttt{pop}, \texttt{call}, \texttt{ret}
SEQ Stages -- Execute

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

- Memory
  - Either reads or writes data to memory
  - \texttt{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
Stage Computation: Arith/Log Ops

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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>icode:ifun ← M₁[PC]</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td>rA:rB ← M₁[PC+1]</td>
<td>Read register byte</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+2</td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>valA ← R[rA]</td>
<td>Read operand A</td>
</tr>
<tr>
<td></td>
<td>valB ← R[rB]</td>
<td>Read operand B</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB OP valA</td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td></td>
<td>Set CC</td>
<td>Set condition code register</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td>R[rB] ← valE</td>
<td>Write back result</td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valP</td>
<td>Update PC</td>
</tr>
</tbody>
</table>

OPq rA, rB
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

---

`rmmovq rA, D(rB)`
Stage Computation: rmmovq

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>rmmovq rA, D(rB)</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read register byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read displacement D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>icode:ifun ← M₁[PC]</td>
<td>Read operand A</td>
</tr>
<tr>
<td></td>
<td>rA:rB ← M₁[PC+1]</td>
<td>Read operand B</td>
</tr>
<tr>
<td></td>
<td>valC ← M₈[PC+2]</td>
<td>Compute effective address</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+10</td>
<td>Write value to memory</td>
</tr>
<tr>
<td>Execute</td>
<td>valA ← R[rA]</td>
<td>Update PC</td>
</tr>
<tr>
<td></td>
<td>valB ← R[rB]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>valE ← valB + valC</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>M₈[valE] ← valA</td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td>PC ← valP</td>
<td></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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>( \text{icode}: \text{ifun} \leftarrow M_1[\text{PC}] ) ( \text{rA} : \text{rB} \leftarrow M_1[\text{PC}+1] ) ( \text{valP} \leftarrow \text{PC}+2 )</td>
</tr>
<tr>
<td>Decode</td>
<td>( \text{valA} \leftarrow R[%rsp] ) ( \text{valB} \leftarrow R[%rsp] )</td>
</tr>
<tr>
<td>Execute</td>
<td>( \text{valE} \leftarrow \text{valB} + 8 )</td>
</tr>
<tr>
<td>Memory</td>
<td>( \text{valM} \leftarrow M_8[\text{valA}] )</td>
</tr>
<tr>
<td>Write back</td>
<td>( R[%rsp] \leftarrow \text{valE} ) ( R[\text{rA}] \leftarrow \text{valM} )</td>
</tr>
<tr>
<td>PC update</td>
<td>( \text{PC} \leftarrow \text{valP} )</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

```cmovXX rA, rB```

```2 fn rA rB```
### Stage Computation: Cond. Move

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read instruction byte&lt;br&gt;Read register byte&lt;br&gt;Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>Read operand A&lt;br&gt;Pass valA through ALU&lt;br&gt;(Disable register update)</td>
</tr>
<tr>
<td>Execute</td>
<td>Write back result</td>
</tr>
<tr>
<td>Memory</td>
<td>Update PC</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td></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>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>icode:ifun ← M₁[PC]</td>
</tr>
<tr>
<td></td>
<td>valC ← M₈[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+9</td>
</tr>
<tr>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>Cnd ← Cond(CC,ifun)</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← Cnd ? valC : valP</td>
</tr>
</tbody>
</table>

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: `ret`

<table>
<thead>
<tr>
<th>Stage</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fetch</strong></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`                                                     |
| **PC update** | `PC ← valM`                                                                    |

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>icode, ifun</td>
</tr>
<tr>
<td></td>
<td>rA, rB</td>
</tr>
<tr>
<td></td>
<td>valC</td>
</tr>
<tr>
<td></td>
<td>valP</td>
</tr>
<tr>
<td>Decode</td>
<td>valA, srcA</td>
</tr>
<tr>
<td></td>
<td>valB, srcB</td>
</tr>
<tr>
<td>Execute</td>
<td>valE</td>
</tr>
<tr>
<td></td>
<td>Cond code</td>
</tr>
<tr>
<td>Memory</td>
<td>valM</td>
</tr>
<tr>
<td>Write</td>
<td>dstE</td>
</tr>
<tr>
<td></td>
<td>dstM</td>
</tr>
<tr>
<td>PC update</td>
<td>PC</td>
</tr>
</tbody>
</table>

- All instructions follow same general pattern
- Differ in what gets computed on each step
# 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</td>
<td>call Dest</td>
</tr>
<tr>
<td>rA, rB</td>
<td>valB, srcB</td>
<td>valE</td>
<td></td>
<td>dstM</td>
<td></td>
</tr>
<tr>
<td>valC</td>
<td></td>
<td>valE ← valB + –8</td>
<td>M_8[valE] ← valP</td>
<td></td>
<td>PC ← valC</td>
</tr>
<tr>
<td>valP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

- 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