Sequential Implementation

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

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

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

• Word Expressions
  • Case expressions
    • $[ a : A; b : B; c : C ]$
    • Evaluate test expressions $a, b, c, \ldots$ in sequence
    • Return word expression $A, B, C, \ldots$ for first successful test
HCL Word-Level Examples

- Find minimum of three input words
- HCL case expression
- Final case guarantees match

- Select one of 4 inputs based on two control bits
- HCL case expression
- Simplify tests by assuming sequential matching

Minimum of 3 Words

```c
int Min3 = [
    A < B && A < C : A;
    B < C          : B;
    1              : C;
];
```

4-Way Multiplexor

```c
int Out4 = [
    !s1&&!s0: D0;
    !s1     : D1;
    !s0     : D2;
    1       : D3;
];
```
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

ONE CLOCK CYCLE

![Diagram of SEQ Hardware Structure]
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 \( \text{icode:ifun} \)
  - Optional register byte \( rA:rB \)
  - Optional constant word \( \text{valC} \)
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 `valA` and/or `valB`
  - Typically reads registers designated by `rA` and `rB`
  - For some instructions it reads register `%rsp`
    - Which ones?
    - `push`, `pop`, `call`, `ret`
SEQ Stages -- Execute

• Execute
  • Compute value (math or logic)
    • Condition codes are possibly set
  • Compute memory address
    • \texttt{rmmovq rA, 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
  - $\text{valM}$, 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

<table>
<thead>
<tr>
<th>Stage</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>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>Execute</td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td></td>
<td>Set condition code register</td>
</tr>
<tr>
<td>Memory</td>
<td>Write back result</td>
</tr>
<tr>
<td>Write back</td>
<td>Update PC</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`

#### Code: `rmmovq rA, D(rB)`

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
<th>Instruction Details</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>Read displacement D</td>
<td><code>valC ← M_8[PC+2]</code></td>
</tr>
<tr>
<td></td>
<td>Compute next PC</td>
<td><code>valP ← PC+10</code></td>
</tr>
<tr>
<td>Decode</td>
<td>Read operand A</td>
<td><code>valA ← R[rA]</code></td>
</tr>
<tr>
<td></td>
<td>Read operand B</td>
<td><code>valB ← R[rB]</code></td>
</tr>
<tr>
<td>Execute</td>
<td>Compute effective address</td>
<td><code>valE ← valB + valC</code></td>
</tr>
<tr>
<td>Memory</td>
<td>Write value to memory</td>
<td><code>M_8[valE] ← valA</code></td>
</tr>
<tr>
<td>Write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>Update PC</td>
<td><code>PC ← valP</code></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

### popq rA

<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[½rsp]</td>
<td>Read stack pointer</td>
</tr>
<tr>
<td></td>
<td>valB ← R[½rsp]</td>
<td>Read stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + 8</td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td>valM ← M₈[valA]</td>
<td>Read from stack</td>
</tr>
<tr>
<td>Write back</td>
<td>R[½rsp] ← valE</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td></td>
<td>R[rA] ← valM</td>
<td>Write back result</td>
</tr>
<tr>
<td>PC update</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
**Stage Computation: Cond. Move**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Operation</th>
</tr>
</thead>
</table>
| Fetch | Read instruction byte  
Read register byte  
Compute next PC |
| Decode | Read operand A  
Pass valA through ALU (Disable register update) |
| Execute | Pass valA through ALU  
If ! Cond(CC,ifun) rB ← 0xF |
| Memory | Write back result |
| Write back | Write back result |
| PC update | Update PC |

- 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

![Diagram showing jump execution steps]
Stage Computation: Jumps

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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction/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</td>
<td></td>
</tr>
<tr>
<td>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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read instruction byte</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td>Read destination address</td>
<td>Read destination address</td>
</tr>
<tr>
<td></td>
<td>Compute return point</td>
<td>Compute return point</td>
</tr>
<tr>
<td>Decode</td>
<td>Read stack pointer</td>
<td>Read stack pointer</td>
</tr>
<tr>
<td></td>
<td>Decrement stack pointer</td>
<td>Decrement stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td>Write return value on stack</td>
<td>Write return value on stack</td>
</tr>
<tr>
<td></td>
<td>Update stack pointer</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td>M_8[valE] ← valP</td>
<td>Write return value on stack</td>
</tr>
<tr>
<td>Write</td>
<td>R[%rsp] ← valE</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valC</td>
<td>Set PC to destination</td>
</tr>
</tbody>
</table>

- 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>Instruction</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td><code>icode:ifun ← M₁[PC]</code></td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>Decode</td>
<td><code>valA ← R[%rsp]</code></td>
<td>Read operand stack pointer</td>
</tr>
<tr>
<td></td>
<td><code>valB ← R[%rsp]</code></td>
<td>Read operand stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + 8</code></td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td><code>valM ← M₈[valA]</code></td>
<td>Read return address</td>
</tr>
<tr>
<td>Write</td>
<td><code>R[%rsp] ← valE</code></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td><code>PC ← valM</code></td>
<td>Set PC to return address</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>iCode, iFun, rA, rB</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td>rA, rB</td>
<td>Read register byte</td>
</tr>
<tr>
<td></td>
<td>valC</td>
<td>[Read constant word]</td>
</tr>
<tr>
<td></td>
<td>valP</td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>valA, srcA, valB, srcB</td>
<td>Read operand A</td>
</tr>
<tr>
<td></td>
<td>valB</td>
<td>Read operand B</td>
</tr>
<tr>
<td></td>
<td>valE</td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td></td>
<td>Cond code</td>
<td>Set/use cond. code reg</td>
</tr>
<tr>
<td>Memory</td>
<td>valM</td>
<td>[Memory read/write]</td>
</tr>
<tr>
<td>Write</td>
<td>dstE</td>
<td>Write back ALU result</td>
</tr>
<tr>
<td></td>
<td>dstM</td>
<td>[Write back memory result]</td>
</tr>
<tr>
<td>PC update</td>
<td>PC</td>
<td>Update 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>Step</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>icode, ifun</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td>rA, rB</td>
<td>[Read register byte]</td>
</tr>
<tr>
<td></td>
<td>valC</td>
<td>Read constant word</td>
</tr>
<tr>
<td></td>
<td>valP</td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>valA, srcA</td>
<td>[Read operand A]</td>
</tr>
<tr>
<td></td>
<td>valB, srcB</td>
<td>Read operand B</td>
</tr>
<tr>
<td>Execute</td>
<td>valE</td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td></td>
<td>Cond code</td>
<td>[Set/use cond. code reg]</td>
</tr>
<tr>
<td>Memory</td>
<td>valM</td>
<td>Memory read/write</td>
</tr>
<tr>
<td>Write</td>
<td>dstE</td>
<td>Write back ALU result</td>
</tr>
<tr>
<td></td>
<td>dstM</td>
<td>[Write back memory result]</td>
</tr>
<tr>
<td>PC update</td>
<td>PC</td>
<td>Update PC</td>
</tr>
</tbody>
</table>

- All instructions follow the 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