Sequential Implementation
Y86-64 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0 0</td>
</tr>
<tr>
<td>nop</td>
<td>1 0</td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2 fn rA rB</td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3 0 F rB V</td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4 0 rA rB D</td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5 0 rA rB D</td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6 fn rA rB</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7 fn Dest</td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0 Dest</td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
</tr>
<tr>
<td>pushq rA</td>
<td>A 0 rA F</td>
</tr>
<tr>
<td>popq rA</td>
<td>B 0 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 – 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 $\text{valA}$ and/or $\text{valB}$
  - Typically reads registers designated by $rA$ and $rB$
  - For some instructions it reads register $\%\text{rsp}$
    - Which ones?
    - $\text{push}$, $\text{pop}$, $\text{call}$, $\text{ret}$

---

**Diagram:**

- **Register file**

- **Fetch**
  - Instruction memory
  - PC increment

- **Decode**
  - Icode, Ifun, RA, RB, ValC
  - A, B, M, E

- **Execute**
  - ALU
  - Cnd
  - AluA, AluB

- **Memory**
  - Addr, Data
  - ValM

- **Write back**
  - ValA, ValM

- **PC**
  - NewPC

---

**Date:** 10/29/2019

**Course:** CMPU 224 -- Computer Organization
SEQ Stages -- Execute

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

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

- Write Back
  - Writes up to two results to the register file: valE, valM
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>Actions</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

<table>
<thead>
<tr>
<th>Fetch</th>
<th>rmmovq rA, D(rB)</th>
<th>Read instruction byte</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>icode:ifun ← M_1[PC]</td>
<td>Read register byte</td>
</tr>
<tr>
<td></td>
<td>rA:rB ← M_1[PC+1]</td>
<td>Read displacement D</td>
</tr>
<tr>
<td></td>
<td>valC ← M_8[PC+2]</td>
<td>Compute next PC</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+10</td>
<td></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 + valC</td>
<td>Compute effective address</td>
</tr>
<tr>
<td>Memory</td>
<td>M_8[valE] ← valA</td>
<td>Write value to memory</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valP</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>Description</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read instruction byte, Read register byte</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>Decode</td>
<td>Read stack pointer, Read stack pointer, Compute next PC</td>
<td><code>valA ← R[%rsp]</code>&lt;br&gt;<code>valB ← R[%rsp]</code></td>
</tr>
<tr>
<td>Execute</td>
<td>Increment stack pointer</td>
<td><code>valE ← valB + 8</code></td>
</tr>
<tr>
<td>Memory</td>
<td>Read from stack</td>
<td><code>valM ← M_8[valA]</code></td>
</tr>
<tr>
<td>Write</td>
<td>Update stack pointer, Write back result</td>
<td><code>R[%rsp] ← valE</code>&lt;br&gt;<code>R[rA] ← valM</code></td>
</tr>
<tr>
<td>PC update</td>
<td>Update PC</td>
<td><code>PC ← valP</code></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
```
Stage Computation: Cond. Move

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
</table>
| Fetch          | icode:ifun ← M₁[PC]  
                 | rA:rB ← M₁[PC+1]  
                 | valP ← PC+2                                                           |
| Decode         | valA ← R[rA]                                                            
                 | valB ← 0                                                                 |
| Execute        | valE ← valB + valA  
                 | If ! Cond(CC,ifun) rB ← 0xF                                           |
| Memory         |                                                                             |
| Write back     | R[rB] ← valE                                                           |
| PC update      | PC ← valP                                                             |

- 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
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>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>( \text{icode}: \text{ifun} \leftarrow M_1[PC] )</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td>( \text{valC} \leftarrow M_8[PC+1] )</td>
<td>Read destination address</td>
</tr>
<tr>
<td></td>
<td>( \text{valP} \leftarrow PC+9 )</td>
<td>Compute return point</td>
</tr>
<tr>
<td>Decode</td>
<td>( \text{valB} \leftarrow R[%rsp] )</td>
<td>Read stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td>( \text{valE} \leftarrow \text{valB} + -8 )</td>
<td>Decrement stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td>( M_8[\text{valE}] \leftarrow \text{valP} )</td>
<td>Write return value on stack</td>
</tr>
<tr>
<td>Write back</td>
<td>( R[%rsp] \leftarrow \text{valE} )</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>PC update</td>
<td>( \text{PC} \leftarrow \text{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/Operation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td><code>icode:ifun ← M_1[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 ← M_8[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>Fetch</th>
<th>Fetch</th>
<th>Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>icode, ifun</td>
<td>rA, rB</td>
<td>OPq rA, rB</td>
</tr>
<tr>
<td>valA</td>
<td>valB</td>
<td>valC</td>
</tr>
<tr>
<td>valP</td>
<td>valP</td>
<td>valP ← PC+2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode</th>
<th>Decode</th>
<th>Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>valA, srcA</td>
<td>valB, srcB</td>
<td>icode, ifun</td>
</tr>
<tr>
<td>valA</td>
<td>valB</td>
<td>M1[PC]</td>
</tr>
<tr>
<td>valA</td>
<td>valB</td>
<td>rA:rB ← M1[PC+1]</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>valM</td>
<td>valM</td>
<td>valM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write</th>
<th>Write</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstE</td>
<td>dstE</td>
<td>dstE</td>
</tr>
<tr>
<td>dstM</td>
<td>dstM</td>
<td>dstM</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Steps</th>
<th>Activity</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></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

- icode,ifun $\leftarrow M_{1}[PC]$  
- valC $\leftarrow M_{8}[PC+1]$  
- valP $\leftarrow PC+9$  
- valB $\leftarrow R[\%rsp]$  
- valE $\leftarrow valB + –8$  
- valM $\leftarrow M_{8}[valE]$  
- dstE $\leftarrow R[\%rsp]$  
- dstM $\leftarrow valE$  
- PC $\leftarrow valC$
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