Computer Architecture: Y86-64 Sequential Implementation

CSci 2021: Machine Architecture and Organization
Lecture #19, March 4th, 2016
Your instructor: Stephen McCamant
Based on slides originally by:
Randy Bryant and Dave O’Hallaron

Building Blocks

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

Storage Elements
- Store bits
- Addressable memories
- Non-addressable registers
- 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

Data Types
- bool: Boolean
  - a, b, c, ...
- int: words
  - A, B, C, ...
  - Does not specify word size—bytes, 32-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
- 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)
- Register File
- Memories
  - Access same memory space
  - Data: for reading/writing program data
  - Instruction: for reading instructions

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

- **Fetch**
  - Read instruction from instruction memory
- **Decode**
  - Read program registers
- **Execute**
  - Compute value or address
- **Memory**
  - Read or write data
- **Write Back**
  - Write program registers
- **PC**
  - Update program counter

**Instruction Decoding**

- Instruction byte $\text{icode:ifun}$
- Optional register byte $rA:rB$
- Optional constant word $\text{valC}$

**Instruction Format**

- Instruction byte $\text{icode:ifun}$
- Optional register byte $rA:rB$
- Optional constant word $\text{valC}$

**Executing Arith./Logical Operation**

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

**Stage Computation: Arith/Log. Ops**

- **Fetch**
  - Read instruction byte
  - Read register byte
- **Decode**
  - Compute next PC
- **Execute**
  - Perform ALU operation
  - Set condition code register
- **Memory**
  - Write to memory
- **Write back**
  - Write back result
- **PC update**
  - Update PC

- **Executing rmmovq**

- **Stage Computation: rmmovq**

- **Executing rmmovq**

- **Stage Computation: rmmovq**

- **Executing rmmovq**
### Executing `popq`

**Stage Computation: popq**

- **Fetch**
  - Read 2 bytes
- **Decode**
  - Read stack pointer
- **Execute**
  - Increment stack pointer by 8
- **Memory**
  - Read from old stack pointer
- **Write back**
  - Update stack pointer
- **Write result to register**
- **PC Update**
  - Increment PC by 2

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

**Stage Computation: popq**

- **Fetch**
  - Read instruction byte
- **Decode**
  - Read register byte
- **Execute**
  - Compute next PC
- **Memory**
  - Read from stack
- **Write back**
  - Update stack pointer
- **Write back result**
- **PC Update**
  - Update PC

### Executing Conditional Moves

**Stage Computation: Cond. Move**

- **Fetch**
  - Read register A and pass through ALU
- **Execute**
  - If condition codes & move condition indicate no move
- **Memory**
  - Do nothing
- **Write back**
  - Do nothing
- **PC Update**
  - Set PC to destination if branch taken or to incremented PC if not branch

### Executing Jumps

**Stage Computation: Jumps**

- **Fetch**
  - Read 9 bytes
- **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 destination if branch taken or to incremented PC if not branch

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read destination address</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
<tr>
<td>target:</td>
<td>Set PC to destination</td>
</tr>
</tbody>
</table>

Fetch
- Read 9 bytes
- Increment PC by 9

Decode
- Read stack pointer
- Write incremented PC to new value of stack pointer
- Write incremented PC to PC+9

Execute
- Decrement stack pointer by 8
- Update stack pointer
- Set PC to Dest

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>destM</td>
<td>Read destination address</td>
</tr>
<tr>
<td>PC update</td>
<td>Set PC to destination</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC

Stage Computation: ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td>return:</td>
<td>Store incremented PC</td>
</tr>
</tbody>
</table>

Fetch
- Read instruction byte
- Read operand stack pointer
- Increment stack pointer
- Compute next PC

Decode
- Read operand A
- Read operand B
- Perform ALU operation
- Set/use cond. code reg
- Memory read/write

Execute
- Write back ALU result
- Update PC
## Computed Values

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Execute</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>icode</code> Instruction code</td>
<td><code>valE</code> ALU result</td>
<td><code>valM</code> Value from memory</td>
</tr>
<tr>
<td><code>ifun</code> Instruction function</td>
<td><code>Cnd</code> Branch/move flag</td>
<td></td>
</tr>
<tr>
<td><code>rA</code> Instr. Register A</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>rB</code> Instr. Register B</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>valC</code> Instruction constant</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>valP</code> Incremented PC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>srcA</code> Register ID A</td>
<td></td>
</tr>
<tr>
<td><code>srcB</code> Register ID B</td>
<td></td>
</tr>
<tr>
<td><code>dstE</code> Destination Register E</td>
<td></td>
</tr>
<tr>
<td><code>dstM</code> Destination Register M</td>
<td></td>
</tr>
<tr>
<td><code>valA</code> Register value A</td>
<td></td>
</tr>
<tr>
<td><code>valB</code> Register value B</td>
<td></td>
</tr>
</tbody>
</table>

## SEQ Hardware

### Key
- Blue boxes: predesigned hardware blocks
  - E.g., memories, ALU
- Gray boxes: control logic
  - Describe in HCL
- White ovals: labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values

## Fetch Logic

### Predefined Blocks
- **PC**: Register containing PC
- **Instruction memory**: Read 10 bytes (PC to PC+9)
- **Signal invalid address**
- **Split**: Divide instruction byte into `icode` and `ifun`
- **Align**: Get fields for `rA`, `rB`, and `valC`

### Fetch Control Logic in HCL

```hcl
# Determine instruction code
int icode = [
    imem_error: INOP;
    1: imem_icode;
];

# Determine instruction function
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];
```

## Midterm grade distribution

<table>
<thead>
<tr>
<th>Frequency</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>More</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>More</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Mean**: 47.9  
**Median**: 47.5  
**Will return at end of lecture**
### Fetch Control Logic in HCL

- **ALU compute?**
- **cond**
- **Jump/move flag**
- **CC**
- **Register with 3 condition code bits**
- **Set CC: Should condition code register be loaded?**
- **ALU A: Input A to ALU**
- **ALU B: Input B to ALU**
- **ALU fun: What function should ALU compute?**

### Decode Logic

- **Control Logic**
  - srcA, srcB: read port addresses
  - dstE, dstM: write port addresses
- **Register File**
  - Read ports A, B
  - Write ports E, M
  - Addresses are register IDs or 15 (0xF) (no access)

### Execute Logic

- **Units**
  - ALU: Implements 4 required functions
  - Generates condition code values
  - CC: Register with 3 condition code bits
  - cond: Computes conditional jump/move flag
- **Control Logic**
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?
### ALU Operation

- **Execute**
  - `valE = valB OP valA`
  - `movXX rA, rB`
  - `valE = valA`
  - `movl crA, D(rB)`
  - `valE = valB + valC`
  - `popq rA`
  - `valE = 0 + valA`
  - `cmovXX rA, rB`

### Instruction Status

**Control Logic**
- **stat**: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data

```c
int alufun = {
    icode == IOPQ ? ifun; 1 : ALUADD;
};
```

### Memory Logic

**Memory**
- Reads or writes memory word

**Control Logic**
- stat: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data

### Memory Address

- **OPq rA, rB**
  - No operation
- **movq rA, D(rB)**
  - Write value to memory
- **valM = valA**
  - Read from stack
- **popq rA**
  - No operation
- **valM = M{valA}**
  - Write return value on stack
- **ret**
  - Read return address

```c
int mem_addr = {
    icode in {IMRMOVQ, IPOPQ, IRET } : valA;
    # Other instructions don't need address
};
```

### Memory Read

- **OPq rA, rB**
  - No operation
- **movq rA, D(rB)**
  - Write value to memory
- **valM = M{valA}**
  - Read from stack
- **popq rA**
  - No operation
- **valM = M{valA}**
  - Write return value on stack
- **ret**
  - Read return address

```c
bool mem_read = icode in {IMRMOVQ, IPOPQ, IRET };
```
Data memory

Combinational logic

State
- PC register
- Cond. Code register
- Data memory
- Register file

All updated as clock rises

Combinational Logic
- ALU
- Control logic
- Memory reads
  - Instruction memory
  - Register file
  - Data memory

SEQ Operation

State

- PC register
- Cond. Code register
- Data memory
- Register file

All updated as clock rises

Combinational Logic
- ALU
- Control logic
- Memory reads
  - Instruction memory
  - Register file
  - Data memory

SEQ Operation

Int new_pc = {
    iicode = ZCALS ; valC;
    iicode = ZZXX && Cond : valC;
    iicode = INST : valM;
    1 : valP;
};

SEQ Operation

State

- PC register
- Cond. Code register
- Data memory
- Register file

All updated as clock rises

Combinational Logic
- ALU
- Control logic
- Memory reads
  - Instruction memory
  - Register file
  - Data memory

SEQ Operation

Int new_pc = {
    iicode = ZCALS ; valC;
    iicode = ZZXX && Cond : valC;
    iicode = INST : valM;
    1 : valP;
};

SEQ Operation

Int new_pc = {
    iicode = ZCALS ; valC;
    iicode = ZZXX && Cond : valC;
    iicode = INST : valM;
    1 : valP;
};

SEQ Operation

Int new_pc = {
    iicode = ZCALS ; valC;
    iicode = ZZXX && Cond : valC;
    iicode = INST : valM;
    1 : valP;
};

SEQ Operation

Int new_pc = {
    iicode = ZCALS ; valC;
    iicode = ZZXX && Cond : valC;
    iicode = INST : valM;
    1 : valP;
};
SEQ Summary

Implementation
- Express every instruction as a series of simple steps
- Follow the same general flow for each instruction type
- Assemble registers, memories, predesigned combinational blocks
- Connect with control logic

Limitations
- Too slow to be practical
- In one cycle, must propagate through instruction memory, register file, ALU, and data memory
- Would need to run clock very slowly
- Hardware units only active for fraction of clock cycle

Midterms turn back

By first letter of last name:

A - H  J - M  N - S  T - Z