## Y86 Instruction Set #1

<table>
<thead>
<tr>
<th>Byte</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>halt</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>nop</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>cmovXX rA, rB</td>
<td>2 fn rA rB</td>
</tr>
<tr>
<td>3</td>
<td>irmovl V, rB</td>
<td>3 0 8 rB V</td>
</tr>
<tr>
<td>4</td>
<td>rrmovl rA, D(rB)</td>
<td>4 0 rA rB D</td>
</tr>
<tr>
<td>5</td>
<td>mrmovl D(rB), rA</td>
<td>5 0 rA rB D</td>
</tr>
<tr>
<td>6</td>
<td>OPl rA, rB</td>
<td>6 fn rA rB</td>
</tr>
<tr>
<td>7</td>
<td>jXX Dest</td>
<td>7 fn Dest</td>
</tr>
<tr>
<td>8</td>
<td>call Dest</td>
<td>8 0 Dest</td>
</tr>
<tr>
<td>9</td>
<td>ret</td>
<td>9 0</td>
</tr>
<tr>
<td>A</td>
<td>pushl rA</td>
<td>A 0 rA 8</td>
</tr>
<tr>
<td>B</td>
<td>popl rA</td>
<td>B 0 rA 8</td>
</tr>
</tbody>
</table>

### Notes
- **rrmovl**: 2 0
- **cmovle**: 2 1
- **cmovl**: 2 2
- **cmove**: 2 3
- **cmovne**: 2 4
- **cmovge**: 2 5
- **cmovg**: 2 6

---

CS:APP2e
### Y86 Instruction Set #2

<table>
<thead>
<tr>
<th>Byte</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>halt</td>
</tr>
<tr>
<td>1</td>
<td>nop</td>
</tr>
<tr>
<td>2</td>
<td>cmovXX rA, rB</td>
</tr>
<tr>
<td>3</td>
<td>irmovl V, rB</td>
</tr>
<tr>
<td>4</td>
<td>rmmovl rA, D(rB)</td>
</tr>
<tr>
<td>5</td>
<td>mrmovl D(rB), rA</td>
</tr>
<tr>
<td>6</td>
<td>OPl rA, rB</td>
</tr>
<tr>
<td>7</td>
<td>jXX Dest</td>
</tr>
<tr>
<td>8</td>
<td>call Dest</td>
</tr>
<tr>
<td>9</td>
<td>ret</td>
</tr>
<tr>
<td>A</td>
<td>pushl rA</td>
</tr>
<tr>
<td>B</td>
<td>popl rA</td>
</tr>
</tbody>
</table>

### Instruction Details
- **halt**: 0 0
- **nop**: 1 0
- **cmovXX rA, rB**: 2 fn rA rB
- **irmovl V, rB**: 3 0 8 rB V
- **rmmovl rA, D(rB)**: 4 0 rA rB D
- **mrmovl D(rB), rA**: 5 0 rA rB D
- **OPl rA, rB**: 6 fn rA rB
- **jXX Dest**: 7 fn Dest
- **call Dest**: 8 0 Dest
- **ret**: 9 0
- **pushl rA**: A 0 rA 8
- **popl rA**: B 0 rA 8

### Additional Instructions
- **addl**: 6 0
- **subl**: 6 1
- **andl**: 6 2
- **xorl**: 6 3

**Addl/Subl/Andl/Xorl**

**NO**: 0 0

**Halt**: 0 0
# Y86 Instruction Set #3

<table>
<thead>
<tr>
<th>Byte</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>halt</td>
<td>0 0</td>
</tr>
<tr>
<td>2</td>
<td>nop</td>
<td>1 0</td>
</tr>
<tr>
<td>3</td>
<td>rrmovl rA, rB</td>
<td>2 fn rA rB</td>
</tr>
<tr>
<td>4</td>
<td>irmovl V, rB</td>
<td>3 0 8 rB V</td>
</tr>
<tr>
<td>5</td>
<td>rmmovl rA, D(rB)</td>
<td>4 0 rA rB D</td>
</tr>
<tr>
<td>6</td>
<td>mrmovl D(rB), rA</td>
<td>5 0 rA rB D</td>
</tr>
<tr>
<td>7</td>
<td>OPl rA, rB</td>
<td>6 fn rA rB</td>
</tr>
<tr>
<td>8</td>
<td>jXX Dest</td>
<td>7 fn Dest</td>
</tr>
<tr>
<td>9</td>
<td>call Dest</td>
<td>8 0 Dest</td>
</tr>
<tr>
<td>0</td>
<td>ret</td>
<td>9 0</td>
</tr>
<tr>
<td>A</td>
<td>pushl rA</td>
<td>A 0 rA 8</td>
</tr>
<tr>
<td>B</td>
<td>popl rA</td>
<td>B 0 rA 8</td>
</tr>
</tbody>
</table>

Jump instructions:
- jmp 7 0
- jle 7 1
- jl 7 2
- je 7 3
- jne 7 4
- jge 7 5
- jg 7 6
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
  - \( a \land b, a \lor b, \neg a \)

- Word Comparisons
  - \( A == B, A != B, A < B, A <= B, A >= B, A > B \)

- Set Membership
  - \( A \text{ in } \{ B, C, D \} \)
    - Same as \( A == B \lor A == C \lor A == D \)

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
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 Format

- Instruction byte: iCode:iFun
- Optional register byte: rA:rB
- Optional constant word: 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

\[
\text{OP1 } rA, rB
\]

\[
6 \text{ fn } rA \text{ rB}
\]
## Stage Computation: Arith/Log. Ops

<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 ← R[rB] |
| Execute     | valE ← valB OP valA  
Set CC |
| Memory      |                                                                           |
| Write back  | R[rB] ← valE                             |
| PC update   | PC ← valP                             |

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

```
rmmovl rA, D(rB) | 4 0 | rA | rB | D
```

**Fetch**
- Read 6 bytes

**Decode**
- Read operand registers

**Execute**
- Compute effective address

**Memory**
- Write to memory

**Write back**
- Do nothing

**PC Update**
- Increment PC by 6
Stage Computation: `rmmovl`

- **Fetch**
  - `rmmovl` rA, D(rB)
  - `icode:ifun ← M_1[PC]`
  - `rA:rB ← M_1[PC+1]`
  - `valC ← M_4[PC+2]`
  - `valP ← PC+6`

- **Decode**
  - `valA ← R[rA]`
  - `valB ← R[rB]`

- **Execute**
  - `valE ← valB + valC`

- **Memory**
  - `M_4[valE] ← valA`

- **Write back**
  - `M_4[valE] ← valA`

- **PC update**
  - `PC ← valP`

- Use ALU for address computation

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B
- Compute effective address
- Write value to memory
- Update PC
Executing `popl rA`

**Fetch**
- Read 2 bytes

**Decode**
- Read stack pointer

**Execute**
- Increment stack pointer by 4

**Memory**
- Read from old stack pointer

**Write back**
- Update stack pointer
- Write result to register

**PC Update**
- Increment PC by 2
## Stage Computation: popl

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
</table>
| Fetch                  | Read instruction byte  
                          Read register byte |
| Decode                 | Compute next PC  
                          Read stack pointer  
                          Read stack pointer |
| Execute                | Increment stack pointer  
                          Read from stack  
                          Update stack pointer |
| Memory                 | Update stack pointer  
                          Write back result  
                          Update PC |
| Write back             | Update stack pointer  
                          Write back result  
                          Update PC |
| PC update              | Update PC |

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

**Fetch**
- Read 5 bytes
- Increment PC by 5

**Decode**
- Do nothing

**Execute**
- Determine whether to take branch based on jump condition and condition codes

### Instruction Format

<table>
<thead>
<tr>
<th>jXX Dest</th>
<th>7</th>
<th>fn</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>fall thru:</td>
<td>XX</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>target:</td>
<td>XX</td>
<td>XX</td>
<td></td>
</tr>
</tbody>
</table>

- **Not taken**
- **Taken**

**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

<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 destination address</td>
</tr>
<tr>
<td></td>
<td>Fall through address</td>
</tr>
<tr>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>Take branch?</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>Update PC</td>
</tr>
</tbody>
</table>

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

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

**Fetch**
- Read 5 bytes
- Increment PC by 5

**Decode**
- Read stack pointer

**Execute**
- Decrement stack pointer by 4

**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>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read instruction byte</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read destination address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Compute return point</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>Read stack pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Decrement stack pointer</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>Write return value on stack</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Update stack pointer</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>Write return value on stack</td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td>Update stack pointer</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>Set PC to destination</td>
<td></td>
</tr>
</tbody>
</table>

- Use ALU to decrement stack pointer
- Store incremented PC
Executing `ret`

**Fetch**
- Read 1 byte

**Decode**
- Read stack pointer

**Execute**
- Increment stack pointer by 4

**Memory**
- Read return address from old stack pointer

**Write back**
- Update stack pointer

**PC Update**
- Set PC to return address

`ret`  
9 0

return:  
XX XX
**Stage Computation: ret**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read instruction byte icode:ifun ← M₁[PC]</td>
</tr>
<tr>
<td>Decode</td>
<td>valA ← R[%esp]</td>
</tr>
<tr>
<td></td>
<td>valB ← R[%esp]</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + 4</td>
</tr>
<tr>
<td>Memory</td>
<td>valM ← M₄[valA]</td>
</tr>
<tr>
<td>Write back</td>
<td>R[%esp] ← valE</td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valM</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read instruction byte&lt;br&gt;Read register byte&lt;br&gt;[Read constant word]&lt;br&gt;Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>Read operand A&lt;br&gt;Read operand B&lt;br&gt;Perform ALU operation&lt;br&gt;Set condition code register</td>
</tr>
<tr>
<td>Execute</td>
<td>Read operand A&lt;br&gt;Read operand B&lt;br&gt;Perform ALU operation&lt;br&gt;Set condition code register</td>
</tr>
<tr>
<td>Memory</td>
<td>[Memory read/write]&lt;br&gt;Write back ALU result&lt;br&gt;[Write back memory result]</td>
</tr>
<tr>
<td>Write back</td>
<td>Write back ALU result&lt;br&gt;[Write back memory result]</td>
</tr>
<tr>
<td>PC update</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>Action</th>
<th>Notes</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</td>
<td>[Read register byte]</td>
</tr>
<tr>
<td></td>
<td>valC ← M₄[PC+1]</td>
<td>- Read constant word</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+5</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></td>
<td>valB ← R[%esp]</td>
<td>- Perform ALU operation</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + –4</td>
<td>- [Set condition code reg.]</td>
</tr>
<tr>
<td></td>
<td>Cond code</td>
<td>- [Memory read/write]</td>
</tr>
<tr>
<td>Memory</td>
<td>M₄[valE] ← valP</td>
<td>- [Write back ALU result]</td>
</tr>
<tr>
<td>Write back</td>
<td>dstE ← R[%esp]</td>
<td>- Write back memory result</td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valC</td>
<td>- Update PC</td>
</tr>
</tbody>
</table>

- 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 Register E
- dstM: Destination Register Register M
- valA: Register value A
- valB: Register value B

## Execute
- valE: ALU result
- Cnd: Branch/move flag

## Memory
- valM: Value from memory
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: 32-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 6 bytes (PC to PC+5)
  - Signal invalid address
- **Split**: Divide instruction byte into *icode* and *ifun*
- **Align**: Get fields for *rA*, *rB*, and *valC*
Fetch Logic

Control Logic

- Instr. Valid: Is this instruction valid?
- icode, ifun: Generate no-op if invalid address
- Need regids: Does this instruction have a register byte?
- Need valC: Does this instruction have a constant word?
# Determine instruction code

```c
int icode = [
    imem_error: INOP;
    1: imem_icode;
];
```

# Determine instruction function

```c
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];
```
bool need_regids =
    icode in { IRRMOVL, IOPL, IPUSHL, IPOPL, IIRMOVL, IRMMOVL, IMRMOVL };

bool instr_valid = icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL, IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
Decode Logic

Register File

- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

Control Logic

- srcA, srcB: read port addresses
- dstE, dstM: write port addresses

Signals

- Cnd: Indicate whether or not to perform conditional move
  - Computed in Execute stage
**A Source**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPI rA, rB</td>
<td></td>
<td>valA ← R[rA]</td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td></td>
<td>valA ← R[rA]</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td></td>
<td>valA ← R[rA]</td>
</tr>
<tr>
<td>popl rA</td>
<td></td>
<td>valA ← R[%esp]</td>
</tr>
<tr>
<td>jXX Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
int srcA = [
    icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
    icode in { IPOPL, IRET } : RESP;
    1 : RNONE;  # Don't need register
];
```
### E Destination

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
</table>
| OPI rA, rB  | Write-back | R[rB] ← valE
| cmovXX rA, rB | Write-back | R[rB] ← valE
| rmmovl rA, D(rB) | None |
| popl rA | Write-back | R[%esp] ← valE |
| jXX Dest | None |
| call Dest | Update stack pointer |
| ret | Update stack pointer |

```c
int dstE = [
    icode in { IRRMOVL } && Cnd : rB;
    icode in { IRRMOVL, IOPL} : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE;  // Don't write any register
];
```
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

- 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?
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMMOVl, IRMMOVl, IMRMMOVl } : valC;
    icode in { ICALL, IPUSHL } : -4;
    icode in { IRET, IPOPL } : 4;
    # Other instructions don't need ALU
];
### ALU Operation

- **Execute** \( \text{valE} \leftarrow \text{valB} \text{ OP } \text{valA} \)**
  - Perform ALU operation

- **cmovXX rA, rB**
  - **Execute** \( \text{valE} \leftarrow 0 + \text{valA} \)**
    - Pass \( \text{valA} \) through ALU

- **execute** \( \text{rmmovl rA, D(rB)} \)**
  - **Execute** \( \text{valE} \leftarrow \text{valB} + \text{valC} \)**
    - Compute effective address

- **popl rA**
  - **Execute** \( \text{valE} \leftarrow \text{valB} + 4 \)**
    - Increment stack pointer

- **jXX Dest**
  - No operation

- **call Dest**
  - **Execute** \( \text{valE} \leftarrow \text{valB} + -4 \)**
    - Decrement stack pointer

- **ret**
  - **Execute** \( \text{valE} \leftarrow \text{valB} + 4 \)**
    - Increment stack pointer

```c
int alufun[] = [
    icode == IOPL : 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
Instruction Status

Control Logic

- stat: What is instruction status?

## Determine instruction status

```c
int Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid: SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
];
```
## Memory Address

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPl rA, rB</td>
<td>No operation</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>Write value to memory</td>
</tr>
<tr>
<td>popl rA</td>
<td>Read from stack</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>No operation</td>
</tr>
<tr>
<td>call Dest</td>
<td>Write return value on stack</td>
</tr>
<tr>
<td>ret</td>
<td>Read return address</td>
</tr>
</tbody>
</table>

```c
int mem_addr = [
    icode in { IRMMOVVL, IPUSHL, ICALL, IMRMOVVL } : valE;
    icode in { IPOPL, IRET } : valA;
    # Other instructions don't need address
];
```
Memory Read

- **OPl rA, rB**
  - **Memory**: No operation

- **rmmovl rA, D(rB)**
  - **Memory**: Write value to memory

- **popl rA**
  - **Memory**: Read from stack

- **jXX Dest**
  - **Memory**: No operation

- **call Dest**
  - **Memory**: Write return value on stack

- **ret**
  - **Memory**: Read return address

```cpp
bool mem_read = icode in { IMRMovL, IPOPl, IReT };
```
PC Update Logic

New PC

- Select next value of PC
### PC Update

<table>
<thead>
<tr>
<th>Instruction</th>
<th>PC Update</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OPl rA, rB</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>rmmovl rA, D(rB)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jXX Dest</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>call Dest</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ret</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>PC ← valP</code></td>
<td></td>
<td>Update PC</td>
</tr>
<tr>
<td><code>PC ← valP</code></td>
<td></td>
<td>Update PC</td>
</tr>
<tr>
<td><code>PC ← valP</code></td>
<td></td>
<td>Update PC</td>
</tr>
<tr>
<td><code>PC ← Cnd ? valC : valP</code></td>
<td></td>
<td>Update PC</td>
</tr>
<tr>
<td><code>PC ← valC</code></td>
<td></td>
<td>Set PC to destination</td>
</tr>
<tr>
<td><code>PC ← valM</code></td>
<td></td>
<td>Set PC to return address</td>
</tr>
</tbody>
</table>

```plaintext
int new_pc = [  
    icode == ICALL : valC;  
    icode == IJXX && Cnd : valC;  
    icode == IRET : 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 #2

- state set according to second `irmovl` instruction
- combinational logic starting to react to state changes
SEQ Operation #3

- state set according to second irmovl instruction
- combinational logic generates results for addl instruction
SEQ Operation #4

- state set according to `addl` instruction
- combinational logic starting to react to state changes
SEQ Operation #5

- state set according to `addl` instruction
- combinational logic generates results for `je` instruction
SEQ Summary

Implementation
- Express every instruction as series of simple steps
- Follow 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