### Y86 Instruction Set #1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl ( rA )</td>
<td>0 0</td>
<td>Push ( rA ) to stack</td>
</tr>
<tr>
<td>popl ( rA )</td>
<td>1 0</td>
<td>Pop ( rA ) from stack</td>
</tr>
<tr>
<td>call ( Dest )</td>
<td>7 0 8</td>
<td>Call ( Dest )</td>
</tr>
<tr>
<td>cmovXX ( rA, rB )</td>
<td>2 fn rA rB</td>
<td>Conditional move ( rA ) to ( rB ) based on condition</td>
</tr>
<tr>
<td>irmovl ( V, rB )</td>
<td>3 0 8 rB V</td>
<td>Immediate move ( V ) to ( rB )</td>
</tr>
<tr>
<td>rrmovl ( rA, D(rB) )</td>
<td>4 0 rA rB D</td>
<td>Register move ( rA ) to ( D(rB) )</td>
</tr>
<tr>
<td>rmmovl ( D(rB), rA )</td>
<td>5 0 rA rB D</td>
<td>Register move ( D(rB) ) to ( rA )</td>
</tr>
<tr>
<td>rPl ( rA, rB )</td>
<td>6 fn rA rB</td>
<td>Register move ( rA ) to ( rB )</td>
</tr>
<tr>
<td>jXX ( Dest )</td>
<td>7 fn Dest</td>
<td>Jump to ( Dest )</td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td>Return from interrupt</td>
</tr>
<tr>
<td>pushl ( rA )</td>
<td>A 0 rA 8</td>
<td>Push ( rA ) to stack</td>
</tr>
<tr>
<td>popl ( rA )</td>
<td>B 0 rA 8</td>
<td>Pop ( rA ) from stack</td>
</tr>
</tbody>
</table>

- **rrmovl**
- **cmovle**
- **cmovl**
- **cmove**
- **cmovne**
- **cmovge**
- **cmovg**
Y86 Instruction Set #2

<table>
<thead>
<tr>
<th>Byte</th>
<th>Instruction</th>
<th>Function</th>
<th>Destination</th>
<th>Operand</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 fn</td>
<td>cmovXX rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 0 8</td>
<td>irmovl V, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 0</td>
<td>rmovl rA, D(rB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 0</td>
<td>rmovl D(rB), rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 fn</td>
<td>OP1 rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 fn</td>
<td>jXX Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 0</td>
<td>call Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 0</td>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 0</td>
<td>pushl rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B 0</td>
<td>popl rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- addl 6 0
- subl 6 1
- andl 6 2
- xorl 6 3

CS:APP2e
Y86 Instruction Set #3

Byte

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

Notes:
- jXX Dest: Jump based on comparison of contents of register rA and destination (Dest).
- Opr rA, rB: Argument passed to procedure specified by rB.
- jXX: Jump based on comparison of contents of register rA and address (Dest).
- nop: No operation.
- halt: Stop execution of program.
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 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

Example: $OPl \ rA, \ rB$
### Stage Computation: Arith/Log. Ops

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
</table>
| Fetch      | OPI rA, rB  
  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     | R[rB] ← valE |
| Write back | PC ← valP |
| PC update  | Read instruction byte  
  Read register byte  
  Compute next PC  
  Read operand A  
  Read operand B  
  Perform ALU operation  
  Set condition code register  
  Write back result  
  Update PC |

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions
Executing \texttt{rmmovl}

\begin{itemize}
  \item Fetch
    \begin{itemize}
      \item Read 6 bytes
    \end{itemize}
  \item Decode
    \begin{itemize}
      \item Read operand registers
    \end{itemize}
  \item Execute
    \begin{itemize}
      \item Compute effective address
    \end{itemize}
  \item Memory
    \begin{itemize}
      \item Write to memory
    \end{itemize}
  \item Write back
    \begin{itemize}
      \item Do nothing
    \end{itemize}
  \item PC Update
    \begin{itemize}
      \item Increment PC by 6
    \end{itemize}
\end{itemize}

\texttt{rmmovl rA, D(rB)}
Stage Computation: \texttt{rmmovl}

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>\texttt{rcode:ifun} \leftarrow \texttt{M}_4[PC] \newline \texttt{rA:rB} \leftarrow \texttt{M}_4[PC+1] \newline \texttt{valC} \leftarrow \texttt{M}_4[PC+2] \newline \texttt{valP} \leftarrow PC+6</td>
</tr>
<tr>
<td>Decode</td>
<td>\texttt{valA} \leftarrow R[rA] \newline \texttt{valB} \leftarrow R[rB]</td>
</tr>
<tr>
<td>Execute</td>
<td>\texttt{valE} \leftarrow \texttt{valB} + \texttt{valC}</td>
</tr>
<tr>
<td>Memory</td>
<td>\texttt{M}_4[\texttt{valE}] \leftarrow \texttt{valA}</td>
</tr>
<tr>
<td>Write back</td>
<td>\texttt{M}_4[\texttt{valE}] \leftarrow \texttt{valA}</td>
</tr>
<tr>
<td>PC update</td>
<td>\texttt{PC} \leftarrow \texttt{valP}</td>
</tr>
</tbody>
</table>

- Use ALU for address computation
Executing `popl`

**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`

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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Actions</th>
</tr>
</thead>
</table>
| Fetch          | `icode:ifun ← M_1[PC]`
                | `rA:rB ← M_1[PC+1]`
                | `valP ← PC+2`                       |
| Decode         | `valA ← R[%esp]`                             |
                | `valB ← R[%esp]`                             |
| Execute        | `valE ← valB + 4`                            |
| Memory         | `valM ← M_4[valA]`                           |
| Write back     | `R[%esp] ← valE`                             |
                | `R[rA] ← valM`                              |
| PC update      | `PC ← valP`                                 |

- Read instruction byte
- Read register byte
- Compute next PC
- Read stack pointer
- Read stack pointer
- Increment stack pointer
- Read from stack
- Update stack pointer
- Write back result
- Update PC

```plaintext
popl rA
```
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

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>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>Decode</td>
<td>Read destination address</td>
<td>Read destination address</td>
</tr>
<tr>
<td>Execute</td>
<td>Fall through address</td>
<td>Fall through address</td>
</tr>
<tr>
<td>Memory</td>
<td>Cnd ← Cond(CC,ifun)</td>
<td>Take branch?</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← Cnd ? valC : valP</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</th>
<th>0</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>return:</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>

### 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>Operation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fetch</strong></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><strong>Decode</strong></td>
<td>Read stack pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Decrement stack pointer</td>
<td></td>
</tr>
<tr>
<td><strong>Execute</strong></td>
<td>Write return value on stack</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Update stack pointer</td>
<td></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Write</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>back</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PC update</strong></td>
<td></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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td><code>icode:ifun ← M_1[PC]</code> Read instruction byte</td>
</tr>
<tr>
<td>Decode</td>
<td><code>valA ← R[esp]</code> <code>valB ← R[esp]</code> Read operand stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + 4</code> Increment stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td><code>valM ← M_4[valA]</code> Read return address</td>
</tr>
<tr>
<td>Write back</td>
<td><code>R[esp] ← valE</code> Update stack pointer</td>
</tr>
<tr>
<td>PC update</td>
<td><code>PC ← valM</code> 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>valC</td>
<td>Read register byte</td>
</tr>
<tr>
<td></td>
<td>valP</td>
<td>[Read constant word]</td>
</tr>
<tr>
<td></td>
<td></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></td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td></td>
<td>valE</td>
<td>Set condition code register</td>
</tr>
<tr>
<td></td>
<td>Cond code</td>
<td>[Memory read/write]</td>
</tr>
<tr>
<td>Memory</td>
<td>valM</td>
<td>Write back ALU result</td>
</tr>
<tr>
<td>Write back</td>
<td>dstE, srcE</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
## Computation Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Actions</th>
</tr>
</thead>
</table>
| Fetch | icode,ifun: icode:ifun ← M₁[PC]  
  rA,rB  
  valC  
  valP |
| Decode | valA, srcA  
  valB, srcB  
  valB ← R[.esp] |
| Execute | valE  
  Cond code  
  valE ← valB + –4 |
| Memory | valM  
  M₄[valE] ← valP |
| Write | dstE  
  dstM  
  R[esp] ← valE |
| PC update | PC  
  PC ← valC |

- 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
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**

- 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?

---

**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
int icode = [
    imem_error: INOP;
    1: imem_icode;
];

# Determine instruction function
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];
bool need_regids = icode in { IRRMOVL, IOPL, IPUSHL, IPOPL, IIRMOVL, IRMMOVVL, IMRMOVVL };

bool instr_valid = icode in { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVVL, IMRMOVVL, 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

```
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>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPI rA, rB</td>
<td>Write back result</td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>Conditionally write back result</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>None</td>
</tr>
<tr>
<td>popl rA</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>None</td>
</tr>
<tr>
<td>call Dest</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>ret</td>
<td>Update stack pointer</td>
</tr>
</tbody>
</table>

```c
int dstE = [
    icode in { IRRMOVL } && Cnd : rB;
    icode in { IIRMOLVL, 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 { IIRMOVL, IRMMOVL, IMRMMOVVL } : valC;
    iCode in { ICALL, IPUSHL } : -4;
    iCode in { IRET, IPOPL } : 4;
    # Other instructions don't need ALU
];
ALU Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPI rA, rB</td>
<td><strong>valE</strong> ← <strong>valB</strong> OP <strong>valA</strong></td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td><strong>valE</strong> ← 0 + <strong>valA</strong></td>
<td>Pass <strong>valA</strong> through ALU</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td><strong>valE</strong> ← <strong>valB</strong> + <strong>valC</strong></td>
<td>Compute effective address</td>
</tr>
<tr>
<td>popl rA</td>
<td><strong>valE</strong> ← <strong>valB</strong> + 4</td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>No operation</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>Decrement stack pointer</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>Increment stack pointer</td>
<td></td>
</tr>
</tbody>
</table>

```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?

#### Control Logic Diagram

![Diagram of instruction status logic](image)

---

```c
## Determine instruction status
int Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid: SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
];
```
int mem_addr = [
    icode in { IRMMOVL, IPUSHL, ICALL, IMRMVOL } : valE;
    icode in { IPOPL, IRET } : valA;
    # Other instructions don't need address
];
## Memory Read

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

```c
bool mem_read = icode in { IMRMVOL, IPOPPL, IRET };```

---

Note: The code snippet initializes a boolean variable `mem_read` based on the instruction code `icode`. The conditions `{ IMRMVOL, IPOPPL, IRET }` are used to determine if a memory read operation is present in the instruction code. This is part of a larger context that involves memory management and operation types in a computer architecture, specifically targeting memory read operations.
**PC Update Logic**

**New PC**

- Select next value of PC
<table>
<thead>
<tr>
<th>Instruction</th>
<th>PC Update</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPl rA, rB</td>
<td>PC update</td>
<td>Update PC</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>PC update</td>
<td>Update PC</td>
</tr>
<tr>
<td>popl rA</td>
<td>PC update</td>
<td>Update PC</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>PC update</td>
<td>Update PC</td>
</tr>
<tr>
<td>call Dest</td>
<td></td>
<td>Set PC to destination</td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td>Set PC to return address</td>
</tr>
</tbody>
</table>

```c
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