Lecture 11:
Sequential processor design

Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
Last time

- Operators
- Function pointers
- Typedefs and structures
- `goto`
- Assertions
- Are arrays the same as pointers?
- `setjmp()`/`longjmp()`
- Coroutines
Today

- Y86 instruction set architecture
  - Processor state
  - Instruction encoding
  - Compiling, assembling, and simulation
- Sequential processor design
  - Fetch, decode, execute, memory, write back, PC update
  - Performance implications
Why a fictitious ISA?

• Simpler than X86, but close to it
• Illustrate instruction encoding with concrete examples
• Basis for assembler / simulator exercises
• Simple enough to design
  – Sequential processor (this time)
  – Pipelined processor (next that)
Y86 Processor State

- Program registers
  - Same 8 as with IA32. Each 32 bits

- Condition codes
  - Single-bit flags set by arithmetic or logical instructions
    - OF: Overflow
    - ZF: Zero
    - SF: Negative

- Program counter
  - Indicates address of instruction

- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order
Y86 Instructions

• Format
  – 1-6 bytes of information read from memory
    • Can determine instruction length from first byte
    • Not as many instruction types, and simpler encoding than with IA32
  – Each accesses and modifies some part(s) of the program state
Encoding Registers

- Each register has 4-bit ID

<table>
<thead>
<tr>
<th>Register</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0</td>
</tr>
<tr>
<td>%ecx</td>
<td>1</td>
</tr>
<tr>
<td>%edx</td>
<td>2</td>
</tr>
<tr>
<td>%ebx</td>
<td>3</td>
</tr>
<tr>
<td>%esi</td>
<td>6</td>
</tr>
<tr>
<td>%edi</td>
<td>7</td>
</tr>
<tr>
<td>%esp</td>
<td>4</td>
</tr>
<tr>
<td>%ebp</td>
<td>5</td>
</tr>
</tbody>
</table>

- Same encoding as in IA32

- Register ID 8 indicates “no register”
  - Will use this in our hardware design in multiple places
Addition instruction

- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86 only allows addition to be applied to register data
- Set condition codes based on result
- e.g., addl %eax, %esi Encoding: 60 06
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers
Arithmetic and Logical Operations

- Refer to generically as “OP1”
- Encodings differ only by “function code”
  - Low-order 4 bytes in first instruction word
- Set condition codes as side effect

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction Code</th>
<th>Function Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>addl rA, rB</td>
<td>6 0 rA rB</td>
</tr>
<tr>
<td>Subtract</td>
<td>subl rA, rB</td>
<td>6 1 rA rB</td>
</tr>
<tr>
<td>And</td>
<td>andl rA, rB</td>
<td>6 2 rA rB</td>
</tr>
<tr>
<td>Exclusive-Or</td>
<td>xorl rA, rB</td>
<td>6 3 rA rB</td>
</tr>
</tbody>
</table>
Move Operations

- Like the IA32 `movl` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct
## Move Instruction Examples

<table>
<thead>
<tr>
<th>IA32</th>
<th>Y86</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0xabcd, %edx</td>
<td>irmovl $0xabcd, %edx</td>
<td>30 82 cd ab 00 00</td>
</tr>
<tr>
<td>movl %esp, %ebx</td>
<td>rrmovl %esp, %ebx</td>
<td>20 43</td>
</tr>
<tr>
<td>movl -12(%ebp),%ecx</td>
<td>mrmovl -12(%ebp),%ecx</td>
<td>50 15 f4 ff ff ff ff</td>
</tr>
<tr>
<td>movl %esi,0x41c(%esp)</td>
<td>rmmovl %esi,0x41c(%esp)</td>
<td>40 64 1c 04 00 00</td>
</tr>
</tbody>
</table>

| movl $0xabcd, (%eax) | — |
| movl %eax, 12(%eax,%edx) | — |
| movl (%ebp,%eax,4),%ecx | — |
Jump Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Length</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Jump Unconditionally</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jmp Dest</code></td>
<td>7 0</td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td><strong>Jump When Less Than or Equal</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jle Dest</code></td>
<td>7 1</td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td><strong>Jump When Less Than</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jl Dest</code></td>
<td>7 2</td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td><strong>Jump When Equal</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>je Dest</code></td>
<td>7 3</td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td><strong>Jump When Not Equal</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jne Dest</code></td>
<td>7 4</td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td><strong>Jump When Greater Than or Equal</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jge Dest</code></td>
<td>7 5</td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td><strong>Jump When Greater Than</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jg Dest</code></td>
<td>7 6</td>
<td>Dest</td>
<td></td>
</tr>
</tbody>
</table>

- Refer to generically as “jXX”
- Encodings differ only by “function code”
- Based on values of condition codes
- Same as IA32 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in IA32
Y86 Program Stack

- Region of memory holding program data
- Used in Y86 (and IA32) for supporting procedure calls
- Stack top indicated by `%esp`
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - When popping, increment stack pointer
Stack Operations

- Decrement `%esp` by 4
- Store word from `rA` to memory at `%esp`
- Like IA32

```
pushl rA  | a 0 rA 8
```

- Read word from memory at `%esp`
- Save in `rA`
- Increment `%esp` by 4
- Like IA32

```
popl rA  | b 0 rA 8
```
Subroutine Call and Return

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like IA32

<table>
<thead>
<tr>
<th>call Dest</th>
<th>8 0 Dest</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ret</th>
<th>9 0</th>
</tr>
</thead>
</table>

- Pop value from stack
- Use as address for next instruction
- Like IA32
Miscellaneous Instructions

- **nop**
  - Don’t do anything

- **halt**
  - Stop executing instructions
  - IA32 has comparable instruction, but can’t execute it in user mode
  - We will use it to stop the simulator
Writing Y86 Code

• Try to use C compiler as much as possible
  – Write code in C
  – Compile for IA32 with gcc -S
  – Transliterate into Y86

• Coding example
  – Find number of elements in null-terminated list
    
    ```c
    int len1(int a[]);
    ```

    ```
    a →
    5043
    6125
    7395
    0
    ⇒ 3
    ```
Y86 code generation example

First try
  - Write typical array code

Problem
  - Hard to do array indexing on Y86
    • Since don’t have scaled
      addressing modes

```c
/* Find number of elements in null-terminated list */
int len1(int a[])
{
  int len;
  for (len = 0; a[len]; len++)
    ;
  return len;
}
```

- Compile with `gcc -O2 -S`
Y86 code generation example #2

- Second try
  - Write with pointer code

```c
/* Find number of elements in null-terminated list */
int len2(int a[])
{
    int len = 0;
    while (*a++)
        len++;
    return len;
}
```

- Result
  - Don’t need to do indexed addressing

```
L24:
    movl (%edx),%eax
    incl %ecx
L26:
    addl $4,%edx
    testl %eax,%eax
    jne L24
```

- Compile with `gcc -O2 -S`
Y86 code generation example #3

- IA32 code
  - Setup

- Y86 code
  - Setup

**len2:**

```asm
pushl %ebp
xorl %ecx,%ecx
movl %esp,%ebp
movl 8(%ebp),%edx
movl (%edx),%eax
jmp L26
```

**len2:**

```asm
pushl %ebp  
# Save %ebp
xorl %ecx,%ecx  
# len = 0
rrmovl %esp,%ebp  
# Set frame
mrmovl 8(%ebp),%edx  
# Get a
mrmovl (%edx),%eax  
# Get *a
jmp L26  
# Goto entry
```
Y86 code generation example #4

- IA32 code
  - loop + finish

- Y86 code
  - loop + finish

L24:
  movl (%edx),%eax
  incl %ecx

L26:
  addl $4,%edx
  testl %eax,%eax
  jne L24
  movl %ebp,%esp
  movl %ecx,%eax
  popl %ebp
  ret

L24:
  mrmovl (%edx),%eax # Get *a
  irmovl $1,%esi
  addl %esi,%ecx # len++
  irmovl $4,%esi # Entry:
  addl %esi,%edx # a++
  andl %eax,%eax # *a == 0?
  jne L24 # No--Loop
  rrmovl %ebp,%esp # Pop
  rrmovl %ecx,%eax # Rtn len
  popl %ebp
  ret
Y86 Program Structure

- Program starts at address 0
- Must set up stack
  - Make sure don’t overwrite code!
- Must initialize data
- Can use symbolic names

```
irmovl Stack, %esp  # Set up stack
rrmovl %esp, %ebp  # Set up frame
irmovl List, %edx  # Set up frame
pushl %edx        # Push argument
call len2         # Call Function
halt              # Halt
.align 4          # List of elements
List:             # List of elements
   .long 5043
   .long 6125
   .long 7395
   .long 0

# Function
len2:
   ...

# Allocate space for stack
.pos 0x100
Stack:
```
Assembling Y86 Program

- Generates “object code” file `eg.yo`
  - Actually looks like disassembler output

```plaintext
unix> yas eg.yo

<table>
<thead>
<tr>
<th>Address</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>30</td>
<td>84</td>
<td>00</td>
<td>00</td>
<td><code>irmovl Stack, %esp</code></td>
<td># Set up stack</td>
</tr>
<tr>
<td>0x006</td>
<td>20</td>
<td>45</td>
<td></td>
<td></td>
<td><code>rrmovl %esp, %ebp</code></td>
<td># Set up frame</td>
</tr>
<tr>
<td>0x008</td>
<td>30</td>
<td>82</td>
<td>18</td>
<td>00</td>
<td><code>irmovl List, %edx</code></td>
<td># Push argument</td>
</tr>
<tr>
<td>0x00e</td>
<td>a0</td>
<td>28</td>
<td></td>
<td></td>
<td><code>pushl %edx</code></td>
<td># Call Function</td>
</tr>
<tr>
<td>0x010</td>
<td>80</td>
<td>28</td>
<td>00</td>
<td>00</td>
<td><code>call len2</code></td>
<td># Halt</td>
</tr>
<tr>
<td>0x015</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td><code>halt</code></td>
<td></td>
</tr>
<tr>
<td>0x018</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>.align 4</code></td>
<td></td>
</tr>
<tr>
<td>0x018</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>List:</code></td>
<td></td>
</tr>
<tr>
<td>0x018</td>
<td>b3</td>
<td>13</td>
<td>00</td>
<td>00</td>
<td><code>.long 5043</code></td>
<td># List of elements</td>
</tr>
<tr>
<td>0x01c</td>
<td>e3</td>
<td>17</td>
<td>00</td>
<td>00</td>
<td><code>.long 6125</code></td>
<td></td>
</tr>
<tr>
<td>0x020</td>
<td>e3</td>
<td>1c</td>
<td>00</td>
<td>00</td>
<td><code>.long 7395</code></td>
<td></td>
</tr>
<tr>
<td>0x024</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td><code>.long 0</code></td>
<td></td>
</tr>
</tbody>
</table>
```
Simulating Y86 Program

- Instruction set simulator
  - Computes effect of each instruction on processor state
  - Prints changes in state from original

```
unix> yis eg.yo
```

```
Stopped in 41 steps at PC = 0x16. Exception 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax: 0x00000000 0x00000003
%ecx: 0x00000000 0x00000003
%edx: 0x00000000 0x00000028
%esp: 0x00000000 0x000000fc
%ebp: 0x00000000 0x00000100
%esi: 0x00000000 0x00000004

Changes to memory:
0x00f4: 0x00000000 0x00000010
0x00f8: 0x00000000 0x00000015
0x00fc: 0x00000000 0x00000018
```
Summary

• Y86 instruction set architecture (ISA)
  – Similar state and instructions as IA32
  – Simpler encodings
  – Somewhere between CISC and RISC

• How important is ISA design?
  – Less now than before
    • With enough hardware, can make almost anything go fast
    • Added complexity of little-used instructions is not many transistors
  – Internally, IA32 is almost RISC anyway
    • Decode unit can generate stream of “RISC-like” instructions
Y86 Instruction Set

<table>
<thead>
<tr>
<th>Byte</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>nop</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>halt</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>rrmovl V,rB</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>irmovl V,rB</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>rmmovl rA,D(rB)</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>mrmovl D(rB),rA</td>
</tr>
<tr>
<td>6</td>
<td>fn</td>
<td>OP1 rA,rB</td>
</tr>
<tr>
<td>7</td>
<td>fn</td>
<td>jXX Dest</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>call Dest</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>ret</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>pushl rA</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>popl rA</td>
</tr>
<tr>
<td>addl</td>
<td>6 0</td>
<td></td>
</tr>
<tr>
<td>subl</td>
<td>6 1</td>
<td></td>
</tr>
<tr>
<td>andl</td>
<td>6 2</td>
<td></td>
</tr>
<tr>
<td>xorl</td>
<td>6 3</td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>7 0</td>
<td></td>
</tr>
<tr>
<td>jle</td>
<td>7 1</td>
<td></td>
</tr>
<tr>
<td>jl</td>
<td>7 2</td>
<td></td>
</tr>
<tr>
<td>je</td>
<td>7 3</td>
<td></td>
</tr>
<tr>
<td>jne</td>
<td>7 4</td>
<td></td>
</tr>
<tr>
<td>jge</td>
<td>7 5</td>
<td></td>
</tr>
<tr>
<td>jg</td>
<td>7 6</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
  - 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
      - \( A == B, A != B, A < B, A \leq B, A \geq B, A > B \)
    - Set membership
      - \( A \in \{ B, C, D \} \)
        - 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 \textit{icode}:\textit{ifun}
  - Optional register byte \textit{rA}:\textit{rB}
  - Optional constant word \textit{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

---

<table>
<thead>
<tr>
<th>OP1</th>
<th>rA, rB</th>
<th>6 fn rA rB</th>
</tr>
</thead>
</table>
Stage Computation: Arith/Log. Ops

<table>
<thead>
<tr>
<th>Stage</th>
<th>Operation</th>
<th>Details</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[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 OP valA</td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td></td>
<td>Set CC</td>
<td>Set condition code register</td>
</tr>
<tr>
<td>Memory</td>
<td>R[rB] ← valE</td>
<td>Write back result</td>
</tr>
<tr>
<td>Write back</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valP</td>
<td>Update PC</td>
</tr>
</tbody>
</table>

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions
Executing `rmmovl` rA,D(rB)

- 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`

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read instruction byte, Read register byte, Read displacement D, Compute next PC</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>Read operand A, Read operand B, Compute effective address</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>Compute effective address</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>Write value to memory</td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td>Update PC</td>
<td></td>
</tr>
</tbody>
</table>

### Code

- `rmmovl rA, D(rB)`

### Instructions

- `icode:ifun ← M_1[PC]`
- `rA:rB ← M_1[PC+1]`
- `valC ← M_4[PC+2]`
- `valP ← PC+6`
- `valA ← R[rA]`
- `valB ← R[rB]`
- `valE ← valB + valC`
- `M_4[valE] ← valA`
- `PC ← valP`

---

- Use ALU for address computation

---

*Systems@ETH Zürich*
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

```
popl rA  b 0 rA 8
```
## Stage Computation: popl

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction Code</th>
<th>Register Access</th>
<th>Memory Access</th>
<th>PC Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>icode:ifun ← M₁[PC]</td>
<td>rA:rB ← M₁[PC+1]</td>
<td>valP ← PC+2</td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>valA ← R[%esp]</td>
<td>valB ← R [%esp]</td>
<td></td>
<td>Read from stack</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + 4</td>
<td></td>
<td></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td>valM ← M₄[valA]</td>
<td></td>
<td></td>
<td>Write back result</td>
</tr>
<tr>
<td>Write back</td>
<td>R[%esp] ← valE</td>
<td>R[rA] ← valM</td>
<td></td>
<td>Update PC</td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Use ALU to increment stack pointer
- Must update two registers: Popped value, New SP
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

![Diagram showing the execution of jumps with fields for fall through and target destinations.](image-url)
## 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</td>
<td></td>
</tr>
<tr>
<td>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

- **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>Action</th>
<th>Description</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></td>
<td><code>valC ← M_4[PC+1]</code></td>
<td>Read destination address</td>
</tr>
<tr>
<td></td>
<td><code>valP ← PC+5</code></td>
<td>Compute return point</td>
</tr>
<tr>
<td>Decode</td>
<td><code>valB ← R[.esp]</code></td>
<td>Read stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + –4</code></td>
<td>Decrement stack pointer</td>
</tr>
<tr>
<td>Memory</td>
<td><code>M_4[valE] ← valP</code></td>
<td>Write return value on stack</td>
</tr>
<tr>
<td>Write back</td>
<td><code>R[esp] ← valE</code></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>PC update</td>
<td><code>PC ← valC</code></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 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

Return: `xx xx`
**Stage Computation: `ret`**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td><code>icode::ifun ← M_1[PC]</code></td>
</tr>
</tbody>
</table>
| Decode     | \( \text{valA} ← R[\%\text{esp}] \)
|            | \( \text{valB} ← R[\%\text{esp}] \)                                   |
| Execute    | \( \text{valE} ← \text{valB} + 4 \)                                   |
| Memory     | \( \text{valM} ← M_4[\text{valA}] \)                                  |
| Write back | \( R[\%\text{esp}] ← \text{valE} \)                                   |
| PC update  | \( \text{PC} ← \text{valM} \)                                         |

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

All instructions follow same general pattern

- Differ in what gets computed on each step

<table>
<thead>
<tr>
<th>Fetch</th>
<th>OP1 rA, rB</th>
</tr>
</thead>
<tbody>
<tr>
<td>icode,ifun</td>
<td>icode:ifun ← M₁[PC]</td>
</tr>
<tr>
<td>rA,rB</td>
<td>rA:rB ← M₁[PC+1]</td>
</tr>
<tr>
<td>valC</td>
<td></td>
</tr>
<tr>
<td>valP</td>
<td>valP ← PC+2</td>
</tr>
<tr>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td>valA, srcA</td>
<td>valA ← R[rA]</td>
</tr>
<tr>
<td>valB, srcB</td>
<td>valB ← R[rB]</td>
</tr>
<tr>
<td>Execute</td>
<td></td>
</tr>
<tr>
<td>valE</td>
<td>valE ← valB OP valA</td>
</tr>
<tr>
<td>Cond code</td>
<td>Set CC</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>valM</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>dstE</td>
<td>R[rB] ← valE</td>
</tr>
<tr>
<td>back</td>
<td></td>
</tr>
<tr>
<td>dstM</td>
<td></td>
</tr>
<tr>
<td>PC update</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>

- Read instruction byte
- Read register byte
- [Read constant word]
- Compute next PC
- Read operand A
- Read operand B
- Perform ALU operation
- Set condition code register
- [Memory read/write]
- Write back ALU result
- [Write back memory result]
- Update PC
## Computation Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>iCode, ifun</td>
<td>$\text{icode:ifun} \leftarrow M_1[PC]$</td>
<td>Read instruction byte</td>
</tr>
<tr>
<td></td>
<td>rA, rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>valC</td>
<td>$\text{valC} \leftarrow M_4[PC+1]$</td>
<td>Read constant word</td>
</tr>
<tr>
<td></td>
<td>valP</td>
<td>$\text{valP} \leftarrow PC+5$</td>
<td>Compute next PC</td>
</tr>
<tr>
<td>Decode</td>
<td>valA, srcA</td>
<td></td>
<td>[Read operand A]</td>
</tr>
<tr>
<td></td>
<td>valB, srcB</td>
<td>$\text{valB} \leftarrow R[%\text{esp}]$</td>
<td>Read operand B</td>
</tr>
<tr>
<td>Execute</td>
<td>valE</td>
<td>$\text{valE} \leftarrow \text{valB} + -4$</td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td></td>
<td>Cond code</td>
<td></td>
<td>[Set condition code reg.]</td>
</tr>
<tr>
<td>Memory</td>
<td>valM</td>
<td>$M_4[\text{valE}] \leftarrow \text{valP}$</td>
<td>[Memory read/write]</td>
</tr>
<tr>
<td>Write</td>
<td>dstE</td>
<td>$R[%\text{esp}] \leftarrow \text{valE}$</td>
<td>[Write back ALU result]</td>
</tr>
<tr>
<td></td>
<td>dstM</td>
<td></td>
<td>Write back memory result</td>
</tr>
<tr>
<td>PC update</td>
<td>PC</td>
<td>$\text{PC} \leftarrow \text{valC}$</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 E
  - dstM: Destination Register M
  - valA: Register value A
  - valB: Register value B

- **Execute**
  - valE: ALU result
  - Bch: Branch 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)
  - **Split**: Divide instruction byte into icode and ifun
  - **Align**: Get fields for rA, rB, and valC
• **Control Logic**
  
  – Instr. Valid: Is this instruction valid?
  
  – Need regids: Does this instruction have a register bytes?
  
  – Need valC: Does this instruction have a constant word?
bool need_regids = icode in { IRRMOVL, IOPL, IPUSHL, IPOPL, 
   IIRMOVL, IRMMOVVL, IMRMMOVVL };

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

• Register File
  – Read ports A, B
  – Write ports E, M
  – Addresses are register IDs or 8 (no access)

• Control Logic
  – srcA, srcB: read port addresses
  – dstA, dstB: write port addresses
### A Source

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Decode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OP1 rA, rB</code></td>
<td>valA ← R[rA]</td>
<td>Read operand A</td>
<td></td>
</tr>
<tr>
<td><code>rmmovl rA, D(rB)</code></td>
<td>valA ← R[rA]</td>
<td>Read operand A</td>
<td></td>
</tr>
<tr>
<td><code>popl rA</code></td>
<td>valA ← R[%esp]</td>
<td>Read stack pointer</td>
<td></td>
</tr>
<tr>
<td><code>jXX Dest</code></td>
<td></td>
<td>No operand</td>
<td></td>
</tr>
<tr>
<td><code>call Dest</code></td>
<td></td>
<td>No operand</td>
<td></td>
</tr>
<tr>
<td><code>ret</code></td>
<td>valA ← R[%esp]</td>
<td>Read stack pointer</td>
<td></td>
</tr>
</tbody>
</table>

```c
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>Operation</th>
<th>Write-back</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP1 rA, rB</td>
<td></td>
<td>⎯</td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td>R[rB] ← valE</td>
<td></td>
<td>Write back result</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>Write-back</td>
<td>R[%esp] ← valE</td>
<td></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>popl rA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td>R[%esp] ← valE</td>
<td></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>jXX Dest</td>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>Write-back</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td></td>
<td></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>Write-back</td>
<td>R[%esp] ← valE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>Write-back</td>
<td>R[%esp] ← valE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
int dstE [ ] {
    icode in { IRRMOVL, IIRMVOL, IOPL} : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE;  # Don't need register
};
```
Execute Logic

• Units
  – ALU
    • Implements 4 required functions
    • Generates condition code values
  – CC
    • Register with 3 condition code bits
  – bcond
    • Computes branch 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?
## ALU A Input

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OP1 rA, rB</strong></td>
<td>valE ← valB OP valA</td>
</tr>
<tr>
<td><strong>rmmovl rA, D(rB)</strong></td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td><strong>popl rA</strong></td>
<td>valE ← valB + 4</td>
</tr>
<tr>
<td><strong>jXX Dest</strong></td>
<td>No operation</td>
</tr>
<tr>
<td><strong>call Dest</strong></td>
<td>valE ← valB + 4</td>
</tr>
<tr>
<td><strong>ret</strong></td>
<td>Increment stack pointer</td>
</tr>
</tbody>
</table>

```
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMOVL, IRMMOVL, IMRMOMVL } : 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>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OP1 rA, rB</code></td>
<td><code>valE ← valB OP valA</code></td>
</tr>
<tr>
<td>Execute</td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td><code>rmmovl rA, D(rB)</code></td>
<td><code>valE ← valB + valC</code></td>
</tr>
<tr>
<td>Execute</td>
<td>Compute effective address</td>
</tr>
<tr>
<td><code>popl rA</code></td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + 4</code></td>
</tr>
<tr>
<td><code>jXX Dest</code></td>
<td>No operation</td>
</tr>
<tr>
<td>Execute</td>
<td></td>
</tr>
<tr>
<td><code>call Dest</code></td>
<td>Decrement stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + -4</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>Execute</td>
<td><code>valE ← valB + 4</code></td>
</tr>
</tbody>
</table>

```c
int alufun = [
    icode == IOPL : ifun;
    1 : ALUADD;
];
```
Memory Logic

• Memory
  – Reads or writes memory word

• Control Logic
  – Mem. read: should word be read?
  – Mem. write: should word be written?
  – Mem. addr.: Select address
  – Mem. data.: Select data
## Memory Address

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP1 rA, rB</td>
<td>No operation</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>Write value to memory</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>M₄[valE] ← valA</td>
<td>Read from stack</td>
</tr>
<tr>
<td>popl rA</td>
<td>Read return address</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>valM ← M₄[valA]</td>
<td>Write return value on stack</td>
</tr>
<tr>
<td>jxx Dest</td>
<td>No operation</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>Write return value on stack</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>M₄[valE] ← valP</td>
<td>Read return address</td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>valM ← M₄[valA]</td>
<td></td>
</tr>
</tbody>
</table>

```c
int mem_addr = [
  icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : 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>OP1 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 { IMRMOVVL, 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><strong>OP</strong> <code>rA, rB</code></td>
<td>PC $\leftarrow$ valP</td>
<td>Update PC</td>
</tr>
<tr>
<td><strong>rmmovl</strong> <code>rA, D(rB)</code></td>
<td>PC $\leftarrow$ valP</td>
<td>Update PC</td>
</tr>
<tr>
<td><strong>popl</strong> <code>rA</code></td>
<td>PC $\leftarrow$ valP</td>
<td>Update PC</td>
</tr>
<tr>
<td><strong>jXX</strong> <code>Dest</code></td>
<td>PC $\leftarrow$ Bch ? valC : valP</td>
<td>Update PC</td>
</tr>
<tr>
<td><strong>call</strong> <code>Dest</code></td>
<td>PC $\leftarrow$ valC</td>
<td>Set PC to destination</td>
</tr>
<tr>
<td><strong>ret</strong></td>
<td>PC $\leftarrow$ valM</td>
<td>Set PC to return address</td>
</tr>
</tbody>
</table>

```c
int new_pc = [
    icode == ICALL : valC;
    icode == IJXX && Bch : 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