CS:APP Chapter 4
Computer Architecture
Instruction Set Architecture
Instruction Set Architecture

Assembly Language View

- Processor state
  - Registers, memory, ...

- Instructions
  - addl, pushl, ret, ...
  - How instructions are encoded as bytes

Layer of Abstraction

- Above: how to program machine
  - Processor executes instructions in a sequence

- Below: what needs to be built
  - Use variety of tricks to make it run fast
  - E.g., execute multiple instructions simultaneously
Y86 Processor State

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

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

- Program Counter
  - Indicates address of next instruction

- Program Status
  - Indicates either normal operation or some error condition

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

- Sequence of instructions
- PC points to next instruction
- Fetch & decode
  - Read instruction at PC
- Execute
  - Update registers
  - Move values to/from memory
  - Update condition codes
- Update PC
  - Default: next instruction
  - call/jmp instructions set new PC
  - goto considered your only option

<table>
<thead>
<tr>
<th>%eax</th>
<th>%esi</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>%edi</td>
</tr>
<tr>
<td>%edx</td>
<td>%esp</td>
</tr>
<tr>
<td>%ebx</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

- addl %eax, %ecx
- irmovl $0x5, %ebx
- jmp $0xaddr2
- call $0xfn_addr

ZF  SF  OF
# Y86 Instruction Set #1

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovl rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovl V, rB</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>rB</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>mrmovl D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>OPl rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Y86 Instruction Set #2

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

Keywords:
- `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 #3

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovl rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>rB</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>irmovl V, rB</td>
<td>4</td>
<td>0</td>
<td></td>
<td>rA</td>
<td>rB</td>
<td>D</td>
</tr>
<tr>
<td>rrmovl rA, D(rB)</td>
<td>5</td>
<td>0</td>
<td></td>
<td>rA</td>
<td>rB</td>
<td>D</td>
</tr>
<tr>
<td>mrmovl D(rB), rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP1 rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td>Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td>Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Y86 Instruction Set #4

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovl rA, rB</td>
<td>2 fn rA rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovl V, rB</td>
<td>3 08 rB V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>4 0 rA rB D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovl D(rB), rA</td>
<td>5 0 rA rB D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opr rA, rB</td>
<td>6 fn rA rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7 fn Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0 Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl rA</td>
<td>A 0 rA 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl rA</td>
<td>B 0 rA 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

---

CS:APP2e
Encoding Registers

Each register has 4-bit ID

| %eax | 0 |
| %ecx | 1 |
| %edx | 2 |
| %ebx | 3 |

| %esi | 6 |
| %edi | 7 |
| %esp | 4 |
| %ebp | 5 |

- Same encoding as in IA32

Register ID 15 (0xF) indicates “no register”

- Will use this in our hardware design in multiple places
Instruction Example

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

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

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

- Set with each arithmetic/logical op.
- ZF: was the result 0?
- SF: was the result <0?
- OF: did the result overflow? (two’s complement)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Values</th>
<th>ZF, SF, OF</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl rA, rB</td>
<td>0x1, 0x2</td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>addl rA, rB</td>
<td>TMIN, -1</td>
<td>0, 0, 1</td>
</tr>
<tr>
<td>addl rA, rB</td>
<td>1, -2</td>
<td>0, 1, 0</td>
</tr>
<tr>
<td>addl rA, rB</td>
<td>5, -5</td>
<td>1, 0, 0</td>
</tr>
<tr>
<td>addl rA, rB</td>
<td>TMIN, TMIN</td>
<td>1, 0, 1</td>
</tr>
</tbody>
</table>
Move Operations

- Like the IA32 movl instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

rrmovl rA, rB

Irmovl V, rB

rrmovl rA, D(rB)

mrmovl D(rB), rA

Register --> Register
Immediate --> Register
Register --> Memory
Memory --> Register
## 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>
<tr>
<td>movl $0xabcd, (%eax)</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>movl %eax, 12(%eax,%edx)</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>movl (%ebp,%eax,4),%ecx</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Move Instructions

Move Unconditionally

- **rrmovl rA, rB**  
  2 0 rA rB

Move When Less or Equal

- **cmovle rA, rB**  
  2 1 rA rB

Move When Less

- **cmovl rA, rB**  
  2 2 rA rB

Move When Equal

- **cmove rA, rB**  
  2 3 rA rB

Move When Not Equal

- **cmovne rA, rB**  
  2 4 rA rB

Move When Greater or Equal

- **cmovge rA, rB**  
  2 5 rA rB

Move When Greater

- **cmovg rA, rB**  
  2 6 rA rB

- Refer to generically as “cmovXX”
- Encodings differ only by “function code”
- Based on values of condition codes
- Variants of rrmovl instruction
  - (Conditionally) copy value from source to destination register
## Jump Instructions

### Jump Unconditionally

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp Dest</td>
<td>7 0</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Less or Equal

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>jle Dest</td>
<td>7 1</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Less

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>jl Dest</td>
<td>7 2</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Equal

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>je Dest</td>
<td>7 3</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Not Equal

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>jne Dest</td>
<td>7 4</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Greater or Equal

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>jge Dest</td>
<td>7 5</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Greater

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>jg Dest</td>
<td>7 6</td>
<td>Dest</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
  - Bottom element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer
Stack Operations

- **pushl rA**
  - A 0 rA F
  - Decrement %esp by 4
  - Store word from rA to memory at %esp
  - Like IA32

- **popl rA**
  - B 0 rA F
  - Read word from memory at %esp
  - Save in rA
  - Increment %esp by 4
  - Like IA32
Subroutine Call and Return

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

```
call Dest
```

```
ret
```

- 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
  - Encoding ensures that program hitting memory initialized to zero will halt
Status Conditions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOK</td>
<td>1</td>
</tr>
<tr>
<td>HLT</td>
<td>2</td>
</tr>
<tr>
<td>ADR</td>
<td>3</td>
</tr>
<tr>
<td>INS</td>
<td>4</td>
</tr>
</tbody>
</table>

- Normal operation
- Halt instruction encountered
- Bad address (either instruction or data) encountered
- Invalid instruction encountered

Desired Behavior

- If AOK, keep going
- Otherwise, stop program execution
Writing Y86 Code

Try to Use C Compiler as Much as Possible

- Write code in C
- Compile for IA32 with `gcc34 -O1 -S`
  - Newer versions of GCC do too much optimization
  - Use `ls /usr/bin/gcc*` to find what versions are available
- Transliterate into Y86

Coding Example

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

```plaintext
a
5043
6125
7395
0
⇒ 3
```
First Try

- Write typical array code

```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 `gcc34 -O1 -S`

Problem

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

```
L5:
incl %eax
cmp $0, (%edx,%eax,4)
jne L5
```
Y86 Code Generation Example #2

Second Try

- Write with pointer code

Result

- Don’t need to do indexed addressing

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

.L11:
    incl  %ecx
    movl  (%edx), %eax
    addl  $4, %edx
    testl  %eax, %eax
    jne .L11

- Compile with gcc34 -01 -S
Y86 Code Generation Example #3

**IA32 Code**
- Setup

```assembly
len2:
pushl %ebp
movl %esp, %ebp

movl 8(%ebp), %edx
movl $0, %ecx
movl (%edx), %eax
addl $4, %edx
testl %eax, %eax
je .L13
```

- Need constants 1 & 4
- Store in callee-save registers

**Y86 Code**
- Setup

```assembly
len2:
pushl %ebp          # Save %ebp
rrmovl %esp, %ebp   # New FP
pushl %esi          # Save
irmovl $4, %esi     # Constant 4
pushl %edi          # Save
irmovl $1, %edi     # Constant 1
mrmovl 8(%ebp), %edx # Get a
irmovl $0, %ecx     # len = 0
mrmovl (%edx), %eax # Get *a
addl %esi, %edx     # a++
andl %eax, %eax     # Test *a
je Done             # If zero, goto Done
```

- Use `andl` to test register
Y86 Code Generation Example #4

IA32 Code

- Loop

.L11:
incl %ecx
movl (%edx), %eax
addl $4, %edx
testl %eax, %eax
jne .L11

Y86 Code

- Loop

Loop:
addl %edi, %ecx       # len++
mrmovl (%edx), %eax  # Get *a
addl %esi, %edx       # a++
andl %eax, %eax       # Test *a
jne Loop              # If !0, goto Loop
Y86 Code Generation Example #5

**IA32 Code**

- Finish

```
.L13:
  movl %ecx, %eax

  leave

ret
```

**Y86 Code**

- Finish

```
Done:
  rrmovl %ecx, %eax  # return len
  popl %edi  # Restore %edi
  popl %esi  # Restore %esi
  rrmovl %ebp, %esp  # Restore SP
  popl %ebp  # Restore FP
  ret
```
Y86 Sample Program Structure #1

Program starts at address 0

Must set up stack
- Where located
- Pointer values
- Make sure don’t overwrite code!

Must initialize data

init: # Initialization
    ...
    call Main
    halt

    .align 4 # Program data
array:
    ...

Main: # Main function
    ...
    call len2
    ...

len2: # Length function
    ...

    .pos 0x100 # Placement of stack

Stack:
Y86 Program Structure #2

Init:

irmovl Stack, %esp    # Set up SP
irmovl Stack, %ebp    # Set up FP
call Main             # Execute main
halt                   # Terminate

# Array of 4 elements + terminating 0
.align 4
array:
    .long 0x000d
    .long 0x00c0
    .long 0x0b00
    .long 0xa000
    .long 0

- Program starts at address 0
- Must set up stack
- Must initialize data
- Can use symbolic names
Y86 Program Structure #3

Main:
  pushl %ebp
  rrmovl %esp,%ebp
  irmovl array,%edx
  pushl %edx            # Push array
  call len2            # Call len2(array)
  rrmovl %ebp,%esp     # Call len2(array)
  popl %ebp
  ret

Set up call to len2
- Follow IA32 procedure conventions
- Push array address as argument
Assembling Y86 Program

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

```
unix> yas len.ys
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000:</td>
<td></td>
<td>.pos 0</td>
</tr>
<tr>
<td>0x000:</td>
<td>30f400010000</td>
<td>init: irmovl Stack, %esp</td>
</tr>
<tr>
<td></td>
<td></td>
<td># Set up stack pointer</td>
</tr>
<tr>
<td>0x006:</td>
<td>30f500010000</td>
<td>irmovl Stack, %ebp</td>
</tr>
<tr>
<td></td>
<td></td>
<td># Set up base pointer</td>
</tr>
<tr>
<td>0x00c:</td>
<td>8028000000</td>
<td>call Main</td>
</tr>
<tr>
<td></td>
<td></td>
<td># Execute main program</td>
</tr>
<tr>
<td>0x011:</td>
<td>00</td>
<td>halt</td>
</tr>
<tr>
<td></td>
<td></td>
<td># Terminate program</td>
</tr>
<tr>
<td></td>
<td></td>
<td># Array of 4 elements + terminating 0</td>
</tr>
<tr>
<td>0x014:</td>
<td></td>
<td>.align 4</td>
</tr>
<tr>
<td>0x014:</td>
<td>array:</td>
<td></td>
</tr>
<tr>
<td>0x014:</td>
<td>0d000000</td>
<td>.long 0x000d</td>
</tr>
<tr>
<td>0x018:</td>
<td>c0000000</td>
<td>.long 0x00c0</td>
</tr>
<tr>
<td>0x01c:</td>
<td>000b0000</td>
<td>.long 0x0b00</td>
</tr>
<tr>
<td>0x020:</td>
<td>00a00000</td>
<td>.long 0xa000</td>
</tr>
<tr>
<td>0x024:</td>
<td>00000000</td>
<td>.long 0</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 len.yo
```

Stopped in 50 steps at PC = 0x11. Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:

- `%eax`: 0x00000000 0x00000004
- `%ecx`: 0x00000000 0x00000004
- `%edx`: 0x00000000 0x00000028
- `%esp`: 0x00000000 0x00000100
- `%ebp`: 0x00000000 0x00000100

Changes to memory:

- `0x00ec`: 0x00000000 0x000000f8
- `0x00f0`: 0x00000000 0x00000039
- `0x00f4`: 0x00000000 0x00000014
- `0x00f8`: 0x00000000 0x00000100
- `0x00fc`: 0x00000000 0x0000011
CISC Instruction Sets

- Complex Instruction Set Computer
- Dominant style through mid-80’s

Stack-oriented instruction set

- Use stack to pass arguments, save program counter
- Explicit push and pop instructions

Arithmetic instructions can access memory

- `addl %eax, 12(%ebx,%ecx,4)`
  - requires memory read and write
  - Complex address calculation

Condition codes

- Set as side effect of arithmetic and logical instructions

Philosophy

- Add instructions to perform “typical” programming tasks
RISC Instruction Sets

- Reduced Instruction Set Computer
- Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)

Fewer, simpler instructions

- Might take more to get given task done
- Can execute them with small and fast hardware

Register-oriented instruction set

- Many more (typically 32) registers
- Use for arguments, return pointer, temporaries

Only load and store instructions can access memory

- Similar to Y86 mrmovl and rmmovl

No Condition codes

- Test instructions return 0/1 in register
CISC vs. RISC

Original Debate
- Strong opinions!
- CISC proponents---easy for compiler, fewer code bytes
- RISC proponents---better for optimizing compilers, can make run fast with simple chip design

Current Status
- For desktop processors, choice of ISA not a technical issue
  - With enough hardware, can make anything run fast
  - Code compatibility more important
- For embedded processors, RISC makes sense
  - Smaller, cheaper, less power
  - Most cell phones use ARM processor
Summary

Y86 Instruction Set Architecture
- 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
- Intel has evolved from IA32 to x86-64
  - Uses 64-bit words (including addresses)
  - Adopted some features found in RISC
    - More registers (16)
    - Less reliance on stack