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 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 Instruction Set #1

<table>
<thead>
<tr>
<th>Byte</th>
<th>Label</th>
<th>Opcode</th>
<th>RF: Program registers</th>
<th>CC: Condition codes</th>
<th>Stat: Program status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>halt</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cmovXX rA, rB</td>
<td>2 fn rA rB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>irmovl V, rB</td>
<td>3 0 8 rB V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rmovl rA, D(rB)</td>
<td>4 0 rA rB D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mrmovl D(rB), rA</td>
<td>5 0 rA rB D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OP1 rA, rB</td>
<td>6 fn rA rB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>jXX Dest</td>
<td>7 fn Dest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>call Dest</td>
<td>8 0 Dest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ret</td>
<td>9 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pushl rA</td>
<td>A 0 rA B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>popl rA</td>
<td>B 0 rA B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With Slides from Bryant and O’Hallaron
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

With Slides from Bryant and O'Hallaron

Y86 Instruction Set #2

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

With Slides from Bryant and O'Hallaron
Y86 Instruction Set #3

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

Y86 Instruction Set #4

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

With Slides from Bryant and O'Hallaron
Encoding Registers

- Each register has 4-bit ID
  
  \[
  \begin{array}{c|c}
  \%eax & 0 \\
  \%ecx & 1 \\
  \%edx & 2 \\
  \%ebx & 3 \\
  \%esi & 6 \\
  \%edi & 7 \\
  \%esp & 4 \\
  \%ebp & 5 \\
  \end{array}
  \]

- 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
  
  \textbf{Generic Form}  
  \[\text{addl } rA, rB\]

  \textbf{Encoded Representation}  
  \[60 06\]

- 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., \texttt{addl \%eax,\%esi}  
  Encoding: 60 06
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers
Arithmetic and Logical Operations

Instruction Code  Function Code
Add
- 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

Add
- \texttt{addl rA, rB}
  - 0 \texttt{rA-rB}

Subtract (rA from rB)
- \texttt{subl rA, rB}
  - 0 \texttt{rA-rB}

And
- \texttt{andl rA, rB}
  - 0 \texttt{rA-rB}

Exclusive-Or
- \texttt{xorl rA, rB}
  - 0 \texttt{rA-rB}

Move Operations

Register --> Register
- \texttt{rrmovl rA, rB}
  - 2 \texttt{rA-rB}

Immediate --> Register
- \texttt{irmovl V, rB}
  - 3 \texttt{V}

Register --> Memory
- \texttt{rmovl rA, D(rB)}
  - 4 \texttt{rA-rB} \texttt{D}

Memory --> Register
- \texttt{mrmovl D(rB), rA}
  - 5 \texttt{rA-rB} \texttt{D}

- 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>rmovl -12(%ebp),%ecx</td>
<td>50 15 f4 ff ff ff</td>
</tr>
<tr>
<td>movl %esi,0x41c(%esp)</td>
<td>rrmovl %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**
  - `rmovl 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 `rmovl` instruction
  - (Conditionally) copy value from source to destination register

With Slides from Bryant and O’Hallaron
### Jump Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding (function code)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump Unconditionally</td>
<td><code>jmp Dest 7 0 Dest</code></td>
<td>Refer to generically as &quot;jxx&quot;</td>
</tr>
<tr>
<td>Jump When Less or Equal</td>
<td><code>jle Dest 7 1 Dest</code></td>
<td>Encodings differ only by &quot;function code&quot;</td>
</tr>
<tr>
<td>Jump When Less</td>
<td><code>jl Dest 7 2 Dest</code></td>
<td>Based on values of condition codes</td>
</tr>
<tr>
<td>Jump When Equal</td>
<td><code>je Dest 7 3 Dest</code></td>
<td>Same as IA32 counterparts</td>
</tr>
<tr>
<td>Jump When Not Equal</td>
<td><code>jne Dest 7 4 Dest</code></td>
<td>Encode full destination address</td>
</tr>
<tr>
<td>Jump When Greater or Equal</td>
<td><code>jge Dest 7 5 Dest</code></td>
<td>Unlike PC-relative addressing seen in IA32</td>
</tr>
<tr>
<td>Jump When Greater</td>
<td><code>jg Dest 7 6 Dest</code></td>
<td></td>
</tr>
</tbody>
</table>

### 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
  - After popping, increment stack pointer

*With Slides from Bryant and O'Hallaron*
Stack Operations

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

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

Subroutine Call and Return

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

- **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 -01 -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
    int len1(int a[]);

    a ─  5043
       6125
       7395
       0

    ⇒ 3

Y86 Code Generation Example

• First Try
  • Write typical array code
  • Compile with gcc34 -01 -S

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

    L5:
    incl %eax
    cmp $0, (%edx,%eax,4)
    jne L5

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

With Slides from Bryant and O’Hallaron
Y86 Code Generation Example #2

• Second Try
  • Write with pointer code
  • Compile with `gcc -O1 -S`

/* 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

Y86 Code Generation Example #3

• IA32 Code
  • Setup

<table>
<thead>
<tr>
<th>len2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %ebp</td>
</tr>
<tr>
<td>movl %esp, %ebp</td>
</tr>
<tr>
<td>movl 8(%ebp), %edx</td>
</tr>
<tr>
<td>movl $0, %ecx</td>
</tr>
<tr>
<td>movl (%edx), %eax</td>
</tr>
<tr>
<td>addl $4, %edx</td>
</tr>
<tr>
<td>testl %eax, %eax</td>
</tr>
<tr>
<td>je .L13</td>
</tr>
</tbody>
</table>

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

• Y86 Code
  • Setup

<table>
<thead>
<tr>
<th>len2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %ebp</td>
</tr>
<tr>
<td>rrmovl %esp, %ebp</td>
</tr>
<tr>
<td>pushl %esi</td>
</tr>
<tr>
<td>irmovl $4, %esi</td>
</tr>
<tr>
<td>pushl %edi</td>
</tr>
<tr>
<td>irmovl $1, %edi</td>
</tr>
<tr>
<td>mrmovl 8(%ebp), %edx</td>
</tr>
<tr>
<td>irmovl $0, %ecx</td>
</tr>
<tr>
<td>mrmovl (%edx), %eax</td>
</tr>
<tr>
<td>addl %esi, %edx</td>
</tr>
<tr>
<td>addl %eax, %eax</td>
</tr>
<tr>
<td>je Done</td>
</tr>
</tbody>
</table>

- Use andl to test register

---

With slides from Bryant and O'Hallaron
**Y86 Code Generation Example #4**

• IA32 Code  
  • Loop  

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

• Y86 Code  
  • Loop  

```assembly
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
```

With Slides from Bryant and O’Hallaron

**Y86 Code Generation Example #5**

• IA32 Code  
  • Finish  

```assembly
.L13:
  movl %ecx, %eax
  leave
  ret
```

• Y86 Code  
  • Finish  

```assembly
Done:
  rrmovl %ecx, %eax  # return len
  popl %edi  # Restore %edi
  popl %esi  # Restore %esi
  rrmovl %ebp, %esp  # Restore SP
  popl %ebp  # Restore FP
  ret
```

With Slides from Bryant and O’Hallaron
Y86 Sample Program Structure #1

init: # Initialization
    ...
    call Main
    halt

    .align 4 # Program data
array: ...

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

len2: # Length function
    ...
    .pos 0x100 # Placement of stack
Stack:

- Program starts at address 0
- Must set up stack
  - Where located
  - Pointer values
  - Make sure don’t overwrite code!
- Must initialize data

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
rmovl %esp,%ebp
movl array,%edx
pushl %edx  # Push array
call len2  # Call
len2(array)
rmovl %ebp,%esp
popl %ebp
ret

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

Assembling Y86 Program

unix> yas len.ys

- Generates "object code" file len.yo
  - Actually looks like disassembler output
Simulating Y86 Program

```
unix> yis len.yo
```

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

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Original Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00ec</td>
<td>0x00000000</td>
<td>0x000000f8</td>
</tr>
<tr>
<td>0x00f0</td>
<td>0x00000000</td>
<td>0x00000039</td>
</tr>
<tr>
<td>0x00f4</td>
<td>0x00000000</td>
<td>0x00000014</td>
</tr>
<tr>
<td>0x00f8</td>
<td>0x00000000</td>
<td>0x00000100</td>
</tr>
<tr>
<td>0x00fc</td>
<td>0x00000000</td>
<td>0x00000011</td>
</tr>
</tbody>
</table>

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:

- 0x00000000 → 0x000000f8
- 0x00000000 → 0x00000039
- 0x00000000 → 0x00000014
- 0x00000000 → 0x00000100
- 0x00000000 → 0x00000011

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

With Slides from Bryant and O'Hallaron
**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