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

<table>
<thead>
<tr>
<th>RF: Program registers</th>
<th>CC: Condition codes</th>
<th>Stat: Program status</th>
<th>DMEM: Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ecx</td>
<td>%edx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%esi</td>
<td>%edi</td>
<td>%esp</td>
<td>%ebp</td>
</tr>
<tr>
<td>ZF</td>
<td>SF</td>
<td>OF</td>
<td></td>
</tr>
</tbody>
</table>

- 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 Instruction Set #1

<table>
<thead>
<tr>
<th>Byte</th>
<th>halt</th>
<th>nop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovX x, r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imov V, r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmov x, D, r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>smov D(r), r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp x, r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl rA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl rA</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 #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>nol</td>
<td>cmovXX rA, rB</td>
<td>imovl V, rB</td>
<td>cmovl D(rB), rA</td>
<td>cmovl D(rB), rA</td>
<td>cmovl D(rB), rA</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>nol</td>
<td>cmovXX rA, rB</td>
<td>imovl V, rB</td>
<td>cmovl D(rB), rA</td>
<td>cmovl D(rB), rA</td>
<td>cmovl D(rB), rA</td>
</tr>
</tbody>
</table>

---

**Encoding Registers**

Each register has 4-bit ID

- 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>addl rA, rB</td>
<td>6 0 [A]B</td>
</tr>
<tr>
<td>subl rA, rB</td>
<td>6 1 [A]B</td>
</tr>
<tr>
<td>andl rA, rB</td>
<td>6 2 [A]B</td>
</tr>
<tr>
<td>xorl rA, rB</td>
<td>6 3 [A]B</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

---

**Move Operations**

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

<table>
<thead>
<tr>
<th>Instruction Code</th>
<th>Function Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl rA, rB</td>
<td>8 0 [A]B</td>
</tr>
<tr>
<td>movl V, rB</td>
<td>7 0 [V]B</td>
</tr>
<tr>
<td>movl D(rB), rA</td>
<td>7 1 [D]B</td>
</tr>
<tr>
<td>movl D(rB), rA</td>
<td>7 2 [D]B</td>
</tr>
</tbody>
</table>

- Register -> Register
- Immediate -> Register
- Register -> Memory
- Memory -> Register
### Move Instruction Examples

<table>
<thead>
<tr>
<th>IA32</th>
<th>Y68</th>
<th>Y68 Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0xabcd, tegade</td>
<td>irmovl $0xabcd, teda</td>
<td>30 82 cd ab 00 00</td>
</tr>
<tr>
<td>movl $0xabcd, teda</td>
<td>movl $0xabcd, teda</td>
<td>20 43</td>
</tr>
<tr>
<td>movl (%ebp), tedge</td>
<td>irmovl (%ebp), teda</td>
<td>50 15 64 66 66</td>
</tr>
<tr>
<td>movl (%ebp), tedge</td>
<td>movl (%ebp), teda</td>
<td>50 15 64 66 66</td>
</tr>
</tbody>
</table>

### Conditional Move Instructions

- **Move Unconditionally**
  - cmovl $0xabcd, (%eax)
  - cmovl (%eax), $0xabcd

- **Move When Less or Equal**
  - cmovle $0xabcd, (%eax)
  - cmovle (%eax), $0xabcd (Conditionally) copy value from source to destination register

- **Move When Less**
  - cmovl $0xabcd, (%eax)
  - cmovl (%eax), $0xabcd

- **Move When Equal**
  - cmovl $0xabcd, (%eax)
  - cmovl (%eax), $0xabcd

- **Move When Greater or Equal**
  - cmovge $0xabcd, (%eax)
  - cmovge (%eax), $0xabcd

- **Move When Greater**
  - cmovg $0xabcd, (%eax)
  - cmovg (%eax), $0xabcd

### Jump Instructions

- **Jump Unconditionally**
  - jmp Dest
  - jle Dest
  - jl Dest
  - je Dest
  - jne Dest
  - jge Dest
  - jg Dest

### Y86 Program Stack

- Region of memory holding program data
- Used in Y68 (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

### 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

- Don't do anything
- 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

Administrative Break

- Assignment II: due beginning of Friday’s lecture
  - Late submission period shortened to end Sunday at noon
  - Full solutions also posted Sunday at noon
- Friday lecture: quiz 1 review session
- Quiz 1: in class Monday
  - Open book, any paper notes or printouts allowed
  - No electronics, calculators, phones, etc.
- Buffer lab: starts Friday

Writing Y86 Code

Try to Use C Compiler as Much as Possible

- Write code in C
- Compile for IA32 with `gcc -O1 -S`
  - Older versions of GCC do better (less optimization)
  - Use `module avail` to find what versions are available
  - Transliterate into Y86

Coding Example

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

First Try

- Write typical array code
- Compile with `gcc34 -O1 -S`

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

Problem

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

Second Try

- Write with pointer code
- Compile with `gcc34 -O1 -S`

```
/* 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

```
  5043
  6125
  7395
  0
```
### Y86 Code Generation Example #3

**IA32 Code**
- Setup
  - pushl %ebp
  - movl %ebp, %esp
  - movl %0(%ebp), %edx
  - movl %0, %ecx
  - addl %4, %edx
  - testl %eax, %eax
  - je .L13

**Y86 Code**
- Setup
  - pushl %ebp
  - movl %ebp, %esp
  - movl %0(%ebp), %edx
  - movl %0, %ecx
  - addl %4, %edx
  - testl %eax, %eax
  - addl %eax, %eax
  - je Done

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

### Y86 Code Generation Example #4

**IA32 Code**
- Loop
  - incl %ecx
  - movl (%edx), %eax
  - addl %4, %edx
  - testl %eax, %eax
  - jne .L11

**Y86 Code**
- Loop
  - incl %ecx
  - movl (%edx), %eax
  - addl %edx, %eax
  - andl %eax, %eax
  - jne Loop

### Y86 Code Generation Example #5

**IA32 Code**
- Finish
  - movl %ecx, %eax
  - leave
  - ret

**Y86 Code**
- Finish
  - movl %ecx, %eax
  - ret

### Y86 Sample Program Structure #1

**init:**
- .
- call Main
- halt

**Main:**
- .
- call len2

**len2:**
- .
- pos 0x100

**Stack:**
- .
- .align 4
- .long 0x0d
- .long 0xc0
- .long 0xb0
- .long 0x00
- .long 0x00

### Y86 Program Structure #2

**init:**
- irmovl Stack, %esp # Set up SP
- irmovl Stack, %ebp # Set up FP
- call Main # Execute main
- halt # Terminate
- 
**Array:**
- .align 4
- .long 0x000d
- .long 0x00c0
- .long 0xb000
- .long 0xa000
- .long 0

### Y86 Program Structure #3

**Main:**
- pushl %ebp
- irmovl %esp, %ebp
- pushl %ebp
- irmovl array, %edx
- call len2 # 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

Simulating Y86 Program

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

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, %ebx
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

MIPS Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 - $7</td>
<td>Constant 0</td>
</tr>
<tr>
<td>$8 - $15</td>
<td>Reserved Temp.</td>
</tr>
<tr>
<td>$16 - $23</td>
<td>Return Values</td>
</tr>
<tr>
<td>$24 - $29</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$30 - $31</td>
<td>Caller Save</td>
</tr>
</tbody>
</table>

MIPS Instruction Examples

R-R

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Rd</th>
<th>00000</th>
<th>Ph</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $3,$2,$1</td>
<td>Register add: $3 = $2+$1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-I

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $3,$2, 3145</td>
<td>Immediate add: $3 = $2+3145</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sl $3,$2, 2</td>
<td>Shift left: $3 = $2 &lt;&lt; 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq $3,$2,d0</td>
<td># Branch when $3 = $2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/Store</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $3,16($2)</td>
<td># Load Word: $3 = $2+16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $3,16($2)</td>
<td># Store Word: $M($2+16) = $3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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