Instruction Set Architecture

Assembly Language View
- Processor state
  - Registers, memory, ...
- Instructions
  - addl, movl, leal, ...
  - 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
    - 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

- eax 0
- ecx 1
- edx 2
- ebx 3
- esi 6
- edi 7
- esp 4
- ebp 5

- Same encoding as in IA32
- Register ID 8 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

Add
- 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

Subtract (rA from rB)

And

Exclusive-Or

Move Operations

Register --> Register

Immediate --> Register

Register --> Memory

Memory --> Register

- 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 FE FF FF</td>
</tr>
<tr>
<td>movl %esi,0x41c(%esp)</td>
<td>rrmmovl %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>

Jump Instructions

- **Jump Unconditionally**
  - jmp Dest
  - Encoding: 7 0

- **Jump When Less or Equal**
  - jle Dest
  - Encoding: 7 1

- **Jump When Less**
  - jl Dest
  - Encoding: 7 2

- **Jump When Equal**
  - je Dest
  - Encoding: 7 3

- **Jump When Not Equal**
  - jne Dest
  - Encoding: 7 4

- **Jump When Greater or Equal**
  - jge Dest
  - Encoding: 7 5

- **Jump When Greater**
  - jg Dest
  - Encoding: 7 6

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

- **pushl rA**
  - a 0 rA 8
  - Decrement %esp by 4
  - Store word from rA to memory at %esp
  - Like IA32

- **popl rA**
  - b 0 rA 8
  - Read word from memory at %esp
  - Save in rA
  - Increment %esp by 4
  - Like IA32
Subroutine Call and Return

- `call Dest 8 0 Dest`
  - Push address of next instruction onto stack
  - Start executing instructions at Dest
  - Like IA32

- `ret 9 0`
  - Pop value from stack
  - Use as address for next instruction
  - Like IA32

Miscellaneous Instructions

- `nop 0 0`
  - Don’t do anything

- `halt 1 0`
  - 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[]);
  int len;
  for (len = 0; a[len]; len++)
      ;
  return len;
  ```

  ```y86
  L18:
  incl %eax
  cmpl $0,(%edx,%eax,4)
  jne L18
  ```

Y86 Code Generation Example

First Try

- Write typical array code

```y86
/* Find number of elements in null-terminated list */
int len1(int a[])
{
    int len;
    for (len = 0; a[len]; len++)
        ;
    return len;
}
```
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;
}

- Compile with gcc -O2 -S

Y86 Code Generation Example #3

IA32 Code
- Setup

Y86 Code
- Setup

len2:
    pushl %ebp  # Save %ebp
    xorl %ecx,%ecx  # len = 0
    movl %esp,%ebp # Set frame
    movl 8(%ebp),%edx # Get a
    movl (%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

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

.align 4
Stats:
    .long 0x100
    .pos 0x100
Assembling Y86 Program

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

```
unix> yas eg.ys
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000:</td>
<td>irmovl Stack,%esp</td>
<td>Set up stack</td>
</tr>
<tr>
<td>0x006:</td>
<td>rrmovl %esp,%ebp</td>
<td>Set up frame</td>
</tr>
<tr>
<td>0x008:</td>
<td>irmovl List,%edx</td>
<td>Push %edx</td>
</tr>
<tr>
<td>0x00e:</td>
<td>pushl %edx</td>
<td>Push argument</td>
</tr>
<tr>
<td>0x010:</td>
<td>call len2</td>
<td>Call function</td>
</tr>
<tr>
<td>0x015:</td>
<td>halt</td>
<td>Halt</td>
</tr>
<tr>
<td>0x018:</td>
<td>.align 4</td>
<td></td>
</tr>
<tr>
<td>0x018:</td>
<td>List:</td>
<td>List of elements</td>
</tr>
<tr>
<td>0x018:</td>
<td>b3130000</td>
<td>.long 5043</td>
</tr>
<tr>
<td>0x01c:</td>
<td>ed170000</td>
<td>.long 6125</td>
</tr>
<tr>
<td>0x020:</td>
<td>e31c0000</td>
<td>.long 7395</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 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 0x0000028
  - %esp: 0x00000000 0x00000000
  - %ebp: 0x00000000 0x00000010
  - %esi: 0x00000000 0x00000004

- Changes to memory:
  - 0x00f4: 0x00000000 0x00000100
  - 0x00f8: 0x00000000 0x00000015
  - 0x00fc: 0x00000000 0x00000018

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 irmovl and rmmovl

No Condition codes
- Test instructions return 0/1 in register
### MIPS Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>$1</td>
<td>Return Values</td>
</tr>
<tr>
<td>$2</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$3</td>
<td>Caller Save Temporaries:</td>
</tr>
<tr>
<td></td>
<td>May be overwritten by called procedures</td>
</tr>
<tr>
<td>$4</td>
<td>Caller Save Temporaries:</td>
</tr>
<tr>
<td></td>
<td>May not be overwritten by called procedures</td>
</tr>
<tr>
<td>$5</td>
<td>Reserved for Operating Sys</td>
</tr>
<tr>
<td>$6</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>$7</td>
<td>Caller Save Temp</td>
</tr>
<tr>
<td>$8</td>
<td>Global Pointer</td>
</tr>
<tr>
<td>$9</td>
<td>Return Address</td>
</tr>
<tr>
<td>$10</td>
<td>$11</td>
</tr>
<tr>
<td>$12</td>
<td>$13</td>
</tr>
<tr>
<td>$14</td>
<td>$15</td>
</tr>
<tr>
<td>$16</td>
<td>$17</td>
</tr>
<tr>
<td>$18</td>
<td>$19</td>
</tr>
<tr>
<td>$20</td>
<td>$21</td>
</tr>
<tr>
<td>$22</td>
<td>$23</td>
</tr>
<tr>
<td>$24</td>
<td>$25</td>
</tr>
<tr>
<td>$26</td>
<td>$27</td>
</tr>
<tr>
<td>$28</td>
<td>$29</td>
</tr>
<tr>
<td>$30</td>
<td>$31</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>Fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>$3</td>
<td>$2</td>
<td>$1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**R-I**

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>$3</td>
<td>$2</td>
<td>3145</td>
</tr>
</tbody>
</table>

**Branch**

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>$3</td>
<td>$2</td>
<td>dest</td>
</tr>
</tbody>
</table>

**Load/Store**

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>$3</td>
<td>$2</td>
<td>16</td>
</tr>
<tr>
<td>sw</td>
<td>$3</td>
<td>$2</td>
<td>16</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

### 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 is moving away from IA32
  - Does not allow enough parallel execution
  - Introduced IA64
    - 64-bit word sizes (overcome address space limitations)
    - Radically different style of instruction set with explicit parallelism
    - Requires sophisticated compilers