Systems I

Instruction Set Architecture - I

Topics

- Introduction to instruction sets
- Y86 as a subset of the X86
- Basic elements to be implemented by hardware
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

<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

### Instruction Code vs. Function Code

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

<table>
<thead>
<tr>
<th>IA32</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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

- **Pseudo:** `jmp Dest`  
  - **Required Encodings:** 7 0  
  - **Description:** Unconditionally jumps to destination address.

### Jump When Less or Equal

- **Pseudo:** `jle Dest`  
  - **Required Encodings:** 7 1  
  - **Description:** Jumps to destination address if the source code is less than or equal to the destination code.

### Jump When Less

- **Pseudo:** `jl Dest`  
  - **Required Encodings:** 7 2  
  - **Description:** Jumps to destination address if the source code is less than the destination code.

### Jump When Equal

- **Pseudo:** `je Dest`  
  - **Required Encodings:** 7 3  
  - **Description:** Jumps to destination address if the source code is equal to the destination code.

### Jump When Not Equal

- **Pseudo:** `jne Dest`  
  - **Required Encodings:** 7 4  
  - **Description:** Jumps to destination address if the source code is not equal to the destination code.

### Jump When Greater or Equal

- **Pseudo:** `jge Dest`  
  - **Required Encodings:** 7 5  
  - **Description:** Jumps to destination address if the source code is greater than or equal to the destination code.

### Jump When Greater

- **Pseudo:** `jg Dest`  
  - **Required Encodings:** 7 6  
  - **Description:** Jumps to destination address if the source code is greater than the destination code.

- **Details:**
  - 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

- **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**
  - 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[]);
```

![Table](image)
Y86 Code Generation Example

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 `gcc -O2 -S`

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

```assembly
L18:
    incl %eax
    cmpl $0, (%edx,%eax,4)
    jne L18
```
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;
}
```

- Compile with `gcc -O2 -S`

Result

- Don’t need to do indexed addressing

```assembly
L24:
    movl (%edx),%eax
    incl %ecx
L26:
    addl $4,%edx
    testl %eax,%eax
    jne L24
```
Y86 Code Generation Example #3

IA32 Code

- Setup

len2:
  pushl %ebp
  xorl %ecx,%ecx
  movl %esp,%ebp
  movl 8(%ebp),%edx
  movl (%edx),%eax
  jmp L26

Y86 Code

- Setup

len2:
  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

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

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

Y86 Code

- Loop + Finish

L24:
  mrmovl (%edx),%eax # Get *a
  irmovl $1,%esi
  addl %esi,%ecx # len++

L26:
  # Entry:
  irmovl $4,%esi
  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        # Push argument
pushl %edx              # Call Function
halt                    # Halt
.align 4
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

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

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Assembly</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000:</td>
<td>3084000010000</td>
<td>irmovl Stack,%esp</td>
<td># Set up stack</td>
</tr>
<tr>
<td>0x006:</td>
<td>2045</td>
<td>rrmovl %esp,%ebp</td>
<td># Set up frame</td>
</tr>
<tr>
<td>0x008:</td>
<td>308218000000</td>
<td>irmovl List,%edx</td>
<td></td>
</tr>
<tr>
<td>0x00e:</td>
<td>a028</td>
<td>pushl %edx</td>
<td># Push argument</td>
</tr>
<tr>
<td>0x010:</td>
<td>80280000000</td>
<td>call len2</td>
<td># Call Function</td>
</tr>
<tr>
<td>0x015:</td>
<td>10</td>
<td>halt</td>
<td># Halt</td>
</tr>
<tr>
<td>0x018:</td>
<td></td>
<td>.align 4</td>
<td></td>
</tr>
<tr>
<td>0x018:</td>
<td></td>
<td>List:</td>
<td># List of elements</td>
</tr>
<tr>
<td>0x018:</td>
<td>b31300000</td>
<td>.long 5043</td>
<td></td>
</tr>
<tr>
<td>0x01c:</td>
<td>e1700000</td>
<td>.long 6125</td>
<td></td>
</tr>
<tr>
<td>0x020:</td>
<td>e31c0000</td>
<td>.long 7395</td>
<td></td>
</tr>
<tr>
<td>0x024:</td>
<td>00000000</td>
<td>.long 0</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

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
- 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
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
# MIPS Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>$1</td>
<td>Reserved Temp.</td>
</tr>
<tr>
<td>$2</td>
<td>Return Values</td>
</tr>
<tr>
<td>$3</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$4</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$5</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$6</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$7</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$8</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$9</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$10</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$11</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$12</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$13</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$14</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$15</td>
<td>Procedure arguments</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$17</td>
<td>May not be overwritten by called procedures</td>
</tr>
<tr>
<td>$18</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$19</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$20</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$21</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$22</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$23</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$24</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$25</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$26</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$27</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$28</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$29</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$30</td>
<td>Callee Save Temporaries:</td>
</tr>
<tr>
<td>$31</td>
<td>Callee Save Temporaries:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0</td>
<td>Caller Save Temp</td>
</tr>
<tr>
<td>$s1</td>
<td>Reserved for Operating Sys</td>
</tr>
<tr>
<td>$s2</td>
<td>Global Pointer</td>
</tr>
<tr>
<td>$s3</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>$s4</td>
<td>Callee Save Temp</td>
</tr>
<tr>
<td>$s5</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s6</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s7</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s8</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s9</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s10</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s11</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s12</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s13</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s14</td>
<td>Return Address</td>
</tr>
<tr>
<td>$s15</td>
<td>Return Address</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>Offset</th>
<th>Fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>$3,$2,$1</td>
<td># Register add: $3 = $2+$1</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>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>$3,$2, 3145</td>
<td># Immediate add: $3 = $2+3145</td>
<td></td>
</tr>
<tr>
<td>sll</td>
<td>$3,$2,2</td>
<td># Shift left: $3 = $2 &lt;&lt; 2</td>
<td></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,$2,dest</td>
<td># Branch when $3 = $2</td>
<td></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>$3,16($2)</td>
<td># Load Word: $3 = M[$2+16]</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>$3,16($2)</td>
<td># Store Word: M[$2+16] = $3</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