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
- %ebx: 2
- %ecx: 1
- %edx: 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

- 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 format 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>rmovl %esp, %ebx</td>
<td>20 43</td>
</tr>
<tr>
<td>movl -12(%ebp),%ecx</td>
<td>mrmovl -12(%ebp),%ecx</td>
<td>50 15 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>

Jump Instructions

- Jump Unconditionally
  - `jmp Dest` 7 0 Dest
- Jump When Less or Equal
  - `jle Dest` 7 1 Dest
- Jump When Less
  - `jl Dest` 7 2 Dest
- Jump When Equal
  - `je Dest` 7 3 Dest
- Jump When Not Equal
  - `jne Dest` 7 4 Dest
- Jump When Greater or Equal
  - `jge Dest` 7 5 Dest
- Jump When Greater
  - `jg Dest` 7 6 Dest

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` 4 0 [RA 8]
  - Decrement %esp by 4
  - Store word from rA to memory at %esp
  - Like IA32

- `popl rA` 8 0 [RA 8]
  - 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

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 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;
}
```

Problem

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

- Compile with gcc -O2 -S

```c
L18:
incl %eax
cmp $0,(%edx,%eax,4)
jne L18
```
Second Try
- Write with pointer code
  /* 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

Example #2

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
  
  L24:
  movl (%edx),%eax
  incl %ecx
  
  L26:
  addl $4,%edx
  testl %eax,%eax
  jne L24

Y86 Code
- Setup
  
  len2:
  pushl %ebp
  # Save %ebp
  xorl %ecx,%ecx
  # len = 0
  rrmovl %esp,%ebp
  # Set frame
  mmovl 8(%ebp),%edx
  # Get a
  mmovl (%edx),%eax
  # Get *a
  jmp L26
  # Goto entry

Y86 Code Generation Example #4

IA32 Code
- Loop + Finish
  
  L24:
  mmovl (%edx),%eax
  # Get *a
  irmovl $1,%esi
  addl %esi,%ecx
  # len++
  
  L26:
  irmovl $4,%esi
  addl %esi,%edx
  # a++
  andl %eax,%eax
  # *a == 0?
  jne L24
  # No--Loop
  
  # Entry:
  irmovl %ebp,%esp
  # Pop
  rrmovl %ebp,%esp
  # Return
  irmovl %ecx,%eax
  # Rtn len
  popl %ebp
  ret

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

Example #3

Y86 Code
- Setup
  
  len2:
  pushl %ebp
  # Save %ebp
  xorl %ecx,%ecx
  # len = 0
  rrmovl %esp,%ebp
  # Set frame
  mmovl 8(%ebp),%edx
  # Get a
  mmovl (%edx),%eax
  # Get *a
  jmp L26
  # Goto entry

IA32 Code
- Loop + Finish
  
  L24:
  mrmovl (%edx),%eax
  # Get *a
  irmovl $1,esi
  addl esi,ecx
  # len++
  
  L26:
  mrmovl $4,esi
  addl esi,edx
  # a++
  andl eax,eax
  # *a == 0?
  jne L24
  # No--Loop
  
  # Entry:
  mrmovl %ebp,esp
  # Pop
  rrmovl %ebp,esp
  # Return
  mrmovl %ecx,eax
  # Rtn len
  popl %ebp
  ret
Assembling Y86 Program

```plaintext
unix> yas eg.ys

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

```
0x000: 308400010000 | irmovl Stack,%es | $ Set up stack
0x006: 2045 | rmovl %esp,%ebp | $ Set up frame
0x008: 308218000000 | irmovl List,%edx
0x00e: a028 | pushl %edx | $ Push argument
0x010: 802800000000 | call len2 | $ Call Function
0x015: 10 | halt | $ Halt
0x018: | .align 4 | $ List: $ List of elements
0x018: b3130000 | .long 5043
0x01c: ed170000 | .long 6125
0x020: a31c0000 | .long 7395
0x024: 00000000 | .long 0
```

Simulating Y86 Program

```plaintext
unix> yis eg.yo

- 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 0x0000000f
%ebp: 0x00000000 0x00000100
%esi: 0x00000000 0x00000004
Changes to memory:
0x00f8: 0x00000000 0x00000010
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 rmovl 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>Sat</td>
</tr>
<tr>
<td>$2$</td>
<td>Return Values</td>
</tr>
<tr>
<td>$3$</td>
<td>SV0</td>
</tr>
<tr>
<td>$4$</td>
<td>SV1</td>
</tr>
<tr>
<td>$5$</td>
<td>SA1</td>
</tr>
<tr>
<td>$6$</td>
<td>Procedure arguments</td>
</tr>
<tr>
<td>$7$</td>
<td>SA2</td>
</tr>
<tr>
<td>$8$</td>
<td>$t0$</td>
</tr>
<tr>
<td>$9$</td>
<td>$t1$</td>
</tr>
<tr>
<td>$10$</td>
<td>$t2$</td>
</tr>
<tr>
<td>$11$</td>
<td>$t3$</td>
</tr>
<tr>
<td>$12$</td>
<td>$t4$</td>
</tr>
<tr>
<td>$13$</td>
<td>$t5$</td>
</tr>
<tr>
<td>$14$</td>
<td>$t6$</td>
</tr>
<tr>
<td>$15$</td>
<td>$t7$</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>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>$3$, $2$, $1$</td>
<td>Register add: $3 = 2+1$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**R-I**

Op | Ra | Rb | Immediate |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></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>slt</td>
<td>$3$, $2$, $2$</td>
<td>Shift left: $3 = 2 &lt;&lt; 2$</td>
<td></td>
</tr>
</tbody>
</table>

**Branch**

Op | Ra | Rb | Offset |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></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**

Op | Ra | Rb | Offset |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>$3$, $16$, $dest$</td>
<td>Load Word: $3 = M[dest]$</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>$3$, $16$, $dest$</td>
<td>Store Word: $M[dest] = 3$</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

**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