

Chapter 4: Y86 Instruction Set

These slides derived from some provided by the authors of our textbook.

Simplified version of the IA32 Instruction set

- Fewer Data Types
- Fewer Instructions
- Fewer Addressing Modes
- Simpler *machine language encoding*

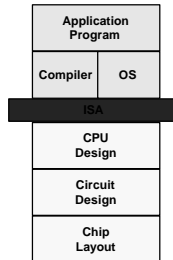
Instruction Set Architecture

Assembly Language View

- Processor state
 - Registers, memory, ...
- Instructions
 - `addl, movl, leal, ...`
 - How instructions are encoded as bytes

Layer of Abstraction

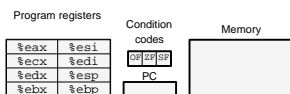
- Above: how to program machine
 - Processor executes instructions in a sequence
- Below: what needs to be built
 - Use variety of tricks to make it run fast
 - E.g., execute multiple instructions simultaneously



CompOrg Fall 2002 - Y86 Instruction Set

2

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 (Instruction Pointer)
 - Indicates address of instruction
- Memory
 - Byte-addressable storage array
 - Words stored in little-endian byte order

CompOrg Fall 2002 - Y86 Instruction Set

3

Y86 Instructions (machine code)

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 | %esi | 6 |
| %ecx | 1 | %edi | 7 |
| %edx | 2 | %esp | 4 |
| %ebx | 3 | %ebp | 5 |

- Same encoding as in IA32

Register ID 8 indicates "no register"

- Will use this in our hardware design in multiple places

Instruction Encoding

| Byte | 0 | 1 | 2 | 3 | 4 | 5 |
|-------------------|---|---|---|---|---|------|
| nop | | | | | | |
| halt | | | | | | |
| rirmovl rA, rB | | | | | | |
| irmovl V, rB | | | | | V | |
| rirmovl rA, D(rB) | | | | | | D |
| mirmovl D(rB), rA | | | | | | D |
| OP1 rA, rB | | | | | | |
| jXX Dest | | | | | | Dest |
| call Dest | | | | | | Dest |
| ret | | | | | | |
| pushl rA | | | | | | |
| popl rA | | | | | | |

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

- 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

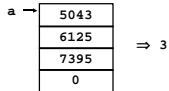
Writing Y86 Code

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
- ```
int len1(int a[]);
```



---

---

---

---

---

---

---

---

## Y86 Code Generation Example

First Try

- Write typical array code

```
/* 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

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

---

---

---

---

---

---

---

---

## Y86 Code Generation Example #2

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

```
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++
 # Entry:

L26:
 irmovl $4,%esi
 addl %esi,%edx
 andl %eax,%eax # *a == 0?
 jne L24 # No--Loop
 rrmovl %ebp,%esp # Pop
 rrmovl %ecx,%eax # Rtn len
 popl %ebp
 ret
```

---

---

---

---

---

---

---

---

---

---

### Y86 Program Structure

```
 irmovl Stack,%esp # Set up stack
 rrmovl %esp,%ebp # Set up frame
 irmovl List,%edx # Push argument
 pushl %edx # Call Function
 call len2 # 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:
```

- Program starts at address 0
- Must set up stack
  - Make sure don't overwrite code!
- Must initialize data
- Can use symbolic names

---

---

---

---

---

---

---

---

---

---

## Assembling Y86 Program

```
unix> yas eg.y8
```

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

```
0x000: 308400010000 | irmovl Stack,%esp | # Set up stack
0x006: 2045 | rrmovl %esp,%ebp | # Set up frame
0x008: 308218000000 | irmovl List,%edx |
0x00e: a028 | pushl %edx | # Push argument
0x010: 8028000000 | call len2 | # Call Function
0x015: 10 | halt | # Halt
0x018: | .align 4 |
0x018: | List: | # List of elements
0x018: b3130000 | .long 5043 |
0x01c: ed170000 | .long 6125 |
0x020: e31c0000 | .long 7395 |
0x024: 00000000 | .long 0 |
```

---

---

---

---

---

---

---

---

---

---

## Simulating Y86 Program

```
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 0x000000fc
%ebp: 0x00000000 0x00000100
%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

---

---

---

---

---

---

---

---

---

---



## 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

---

---

---

---

---

---

---

---