Lecture 7A
Computer Architecture I
Instruction Set Architecture

Assembly Language View
- Processor state
  - Registers, memory, ...
- Instructions
  - `addl`, `movl`, and `...
  - 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 Instruction Set

<table>
<thead>
<tr>
<th>Program registers</th>
<th>Condition Codes</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax %esi</td>
<td>OF: Overflow</td>
<td>PC</td>
</tr>
<tr>
<td>%ecx %edi</td>
<td>ZF: Zero</td>
<td></td>
</tr>
<tr>
<td>%edx %esp</td>
<td>SF: Negative</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
    - OF: Overflow
    - ZF: Zero
    - SF: Negative
- Program Counter
  - Indicates address of instruction
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

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 Example

Addition Instruction

Generic Form

Encoded Representation

- 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

**Register → Register**

```
addl rA, rB  # 6 0 rA|rB
subl rA, rB  # 6 1 rA|rB
andl rA, rB  # 6 2 rA|rB
xorl rA, rB  # 6 3 rA|rB
```

Move Operations

**Register → Register**

```
rrmovl rA, rB  # 2 0 rA|rB
```

**Immediate → Register**

```
irmovl V, rB  # 3 0 8 rB V
```

**Register → Memory**

```
rmovl rA, D(rB)  # 4 0 rA|rB D
```

**Memory → Register**

```
rmovl D(rB), rA  # 5 0 rA|rB D
```

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

Move Instruction Examples

**IA32**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0xabcd, %edx</td>
<td>6 0 06 1f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f</td>
</tr>
<tr>
<td>movl %esp, %ebx</td>
<td>6 0 06 2d 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f</td>
</tr>
<tr>
<td>movl -12(%ebp), %ecx</td>
<td>6 0 f4 89 01 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f</td>
</tr>
<tr>
<td>movl %esi,0x41c(%esp)</td>
<td>6 0 44 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f</td>
</tr>
</tbody>
</table>

**Y86**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0xabcd, %edx</td>
<td>3 0 89 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f</td>
</tr>
<tr>
<td>movl %esp, %ebx</td>
<td>3 0 89 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f</td>
</tr>
<tr>
<td>movl -12(%ebp), %ecx</td>
<td>3 0 89 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f</td>
</tr>
<tr>
<td>movl %esi,0x41c(%esp)</td>
<td>3 0 89 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f</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**

```
jeq 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
```

- 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

Subroutine Call and Return

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like IA32

Stack Operations

- Decrement `esp` by 4
- Store word from rA to memory at `esp`
  - Like IA32

- Read word from memory at `esp`
- Save in rA
- Increment `esp` by 4
  - 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
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[]);

  G# 1
  
  a  5043
  6125
  7395
  0  => 3

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

```c
L18:
  incl #eax
  cmp %ebx, %eax, 4
  jne L18
```

Compile with gcc -O2 -S

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;
}
```

Result
- Don’t need to do indexed addressing

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

Compile with gcc -O2 -S

Y86 Code Generation Example #3

IA32 Code
- Setup

```c
len2:
  pushl %ebp
  xorl %ecx, %ecx
  movl %esp, %ebp
  movl $0(%ebp), %edx
  jmp L26
```

Y86 Code
- Setup

```c
len2:
  pushl %ebp
  # Save %ebp
  xorl %ecx, %ecx
  # len = 0
  rmovl %esp, %ebp
  # Set frame
  movl (%edx), %eax
  jmp L26
```

# Go to entry
**Y86 Code Generation Example #4**

**IA32 Code**

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

**Y68 Code**

```
L24:  mrmoi (%edx),%eax # Get *a
     mrmoi $1,%esi
     addl %esi,%ecx # len++
L26:  # Entry:
     mrmoi $4,%esi
     addl %esi,%edx # a++
     andl %eax,%eax # *a == 0?
     jne L24 # No--Loop
     rmmovl %ebp,%esp # Pop
     rmmovl %ecx,%eax # Rtn len
     popl %ebp
     ret
```

---

**Assembling Y86 Program**

```
unix> yas eg.y
```

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

```
0x000: 3084000010000000 movl stack,esp # Set up stack
0x006: 2045  rmmovl %esp,%ebp # Set up frame
0x008: 3082180000000000  irmovl %ebp
0x010: e028 pushl %edx # Push argument
0x012: 80280000 call len2 # Call Function
0x014: 0f5510 halt # Halt
0x016: .align 4 # List of elements
0x018: 83130000  .long 5043
0x01c: d170000  .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 = 014f. Exception 'HL2', CC 0-1 = 2-0 = 0
Changes to registers:
  %eax: 0x00000000 0x00000003
  %ecx: 0x00000000 0x00000001
  %edx: 0x00000000 0x00000002
  %esp: 0x00000000 0x00000003
  %ebp: 0x00000000 0x00000002
  %esi: 0x00000000 0x00000004

Changes to memory:
0x000: 0x00000000 0x00000001
0x008: 0x00000000 0x00000001
0x000c: 0x00000000 0x00000001
```
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 `movl` and `rmovl`

No Condition codes

- Test instructions return 0/1 in register
- SPARC has condition codes

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