Instruction Set Architecture

Introduction to Computer Systems
4th Lecture, Sep. 10, 2015

Instructor:
Younghoon Kim
Turning C into Object code

```bash
>> gcc p1.c p2.c -o p
```
Instruction Set Architecture #1

What is it?
- Assembly Language Abstraction
- Machine Language Abstraction

What does it provide?
- An abstraction of the real computer, hide the details of implementation
  - The syntax of computer instructions
  - The semantics of instructions
  - The execution model
  - Programmer-visible computer status
Instruction Set Architecture #2

- **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 tricks to make it run fast
    - E.g., execute multiple instructions simultaneously
Instruction Set Architecture #3

- ISA define the processor family
  - Two main kind: RISC and CISC
    - RISC: SPARC, MIPS, PowerPC, ARM
    - CISC: X86 (or called IA32)
- Under same ISA, there are many different processors
  - From different manufacturers
    - X86 from Intel, AMD and VIA
  - Different models (microarchitectures)
    - 8086, 80386, Pentium, Pentium 4
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
    ▪ Simpler encoding than with IA32 (1~17 bytes)
  ▪ Each accesses and modifies some part(s) of the program state

■ Categories
  ▪ Arithmetic & logical
  ▪ Move (immediate, register, memory)
  ▪ Jump
  ▪ Stack operations (push, pop)
  ▪ Subroutine (call, return)
  ▪ Misc.
Encoding Registers

- Each register has 4-bit ID

<table>
<thead>
<tr>
<th></th>
<th>%eax</th>
<th>0</th>
<th>%esi</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%ecx</td>
<td>1</td>
<td>%edi</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>%edx</td>
<td>2</td>
<td>%esp</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>%ebx</td>
<td>3</td>
<td>%ebp</td>
<td>5</td>
</tr>
</tbody>
</table>

- Same encoding as in IA32, but IA32 using only 3-bit ID

- Register ID 8 indicates “no register”
  - Will appear in instruction encodings which do not use register fields
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 “OP1”
- Encodings differ only by “function code”
  - Low-order 4 bytes in first instruction word
- Set condition codes as side effect
- Notice: no multiply or divide operation

### Instruction Code

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

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

- `rrmovl rA, rB` (Register --> Register)
- `irmovl V, rB` (Immediate --> Register)
- `rmmovl rA, D(rB)` (Register --> Memory)
- `mrmovl D(rB), rA` (Memory --> Register)
## 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>

movl $0xabcd, (%eax) —

movl %eax, 12(%eax,%edx) —

movl (%ebp,%eax,4),%ecx —
Jump Instructions

Jump Unconditionally

\[
\text{jmp Dest} \quad 7 \quad 0 \quad \text{Dest}
\]

Jump When Less or Equal

\[
\text{jle Dest} \quad 7 \quad 1 \quad \text{Dest}
\]

Jump When Less

\[
\text{jl Dest} \quad 7 \quad 2 \quad \text{Dest}
\]

Jump When Equal

\[
\text{je Dest} \quad 7 \quad 3 \quad \text{Dest}
\]

Jump When Not Equal

\[
\text{jne Dest} \quad 7 \quad 4 \quad \text{Dest}
\]

Jump When Greater or Equal

\[
\text{jge Dest} \quad 7 \quad 5 \quad \text{Dest}
\]

Jump When Greater

\[
\text{jg Dest} \quad 7 \quad 6 \quad \text{Dest}
\]

- 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 \( \% \text{esp} \)
  - 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

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

```
call Dest 8 0 Dest
```

- Pop value from stack
- Use as address for next instruction
- Like IA32

```
ret 9 0
```
### 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
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 instructions 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**

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</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>Caller Save</td>
</tr>
<tr>
<td>$5</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$6</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$7</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$8</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$9</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$10</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$11</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$12</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$13</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$14</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$15</td>
<td>Caller Save</td>
</tr>
<tr>
<td>$16</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$17</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$18</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$19</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$20</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$21</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$22</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$23</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$24</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$25</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$26</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$27</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$28</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$29</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$30</td>
<td>Callee Save</td>
</tr>
<tr>
<td>$31</td>
<td>Callee Save</td>
</tr>
</tbody>
</table>

- **Constant 0**
- **Reserved Temp.**
- **Return Values**
- **Procedure arguments**
- **Caller Save**:
  - May be overwritten by called procedures
- **Callee Save**:
  - May not be overwritten by called procedures
- **Reserved for Operating Sys**
- **Global Pointer**
- **Stack Pointer**
- **Callee Save Temp**
- **Return Address**
## 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>
</table>

- **addu $3,$2,$1**  # Register add: $3 = $2+$1

### R-I

<table>
<thead>
<tr>
<th>Op</th>
<th>Ra</th>
<th>Rb</th>
<th>Immediate</th>
</tr>
</thead>
</table>

- **addu $3,$2, 3145**  # Immediate add: $3 = $2+3145
- **sll $3,$2,2**  # Shift left: $3 = $2 << 2

### Branch

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

- **beq $3,$2,dest**  # Branch when $3 = $2

### Load/Store

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

- **lw $3,16($2)**  # Load Word: $3 = M [$2+16]
- **sw $3,16($2)**  # Store Word: M [$2+16] = $3
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