Giving credit where credit is due

- Most of slides for this lecture are based on slides created by Dr. Bryant, Carnegie Mellon University.
- I have modified them and added new slides.
Chapter Outline

Background
- Instruction sets
- Logic design

Sequential Implementation
- A simple, but not very fast processor design

Pipelining
- Get more things running simultaneously

Pipelined Implementation
- Make it work

Coverage

The Approach
- Work through designs for particular instruction set
  - Y86—a simplified version of the Intel IA32 (a.k.a. x86).
  - If you know one, you more-or-less know them all
- Work at “microarchitectural” level
  - Assemble basic hardware blocks into overall processor structure
    - Memories, functional units, etc.
  - Surround with control logic to make sure each instruction flows through properly
- Use simple hardware description language to describe control logic
  - Can extend and modify
  - Test via simulation
Topics

- Y86 ISA
- CISC vs. RISC
- High-level overview of MIPS ISA

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

- Same encoding as in IA32
- Register ID 8 indicates “no register”
  - Will use this in our hardware design in multiple places

<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>

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
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers
- e.g., addl %eax, %esi  Encoding: 60 06
### Arithmetic and Logical Operations

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

- 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

### Move Operations

<table>
<thead>
<tr>
<th>Instruction Code</th>
<th>Format</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmmovl rA, rB</td>
<td>2 0 rA</td>
<td>rB</td>
</tr>
<tr>
<td>irmovl V, rB</td>
<td>3 0 8 rB V</td>
<td>Immediate $\rightarrow$ Register</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>4 0 rA</td>
<td>rB D</td>
</tr>
<tr>
<td>mrmovl D(rB), rA</td>
<td>5 0 rA</td>
<td>rB D</td>
</tr>
</tbody>
</table>

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

- \text{movl} $0xabcd, (%eax) ---
- \text{movl} %eax, 12(%eax,%edx) ---
- \text{movl} (%ebp,%eax,4),%ecx ---

Jump Instructions

**Jump Unconditionally**

<table>
<thead>
<tr>
<th>Jump</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp Dest</td>
<td>7 0</td>
</tr>
</tbody>
</table>

**Jump When Less or Equal**

<table>
<thead>
<tr>
<th>Jump</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>jle Dest</td>
<td>7 1</td>
</tr>
</tbody>
</table>

**Jump When Less**

<table>
<thead>
<tr>
<th>Jump</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>jl Dest</td>
<td>7 2</td>
</tr>
</tbody>
</table>

**Jump When Equal**

<table>
<thead>
<tr>
<th>Jump</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>je Dest</td>
<td>7 3</td>
</tr>
</tbody>
</table>

**Jump When Not Equal**

<table>
<thead>
<tr>
<th>Jump</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>jne Dest</td>
<td>7 4</td>
</tr>
</tbody>
</table>

**Jump When Greater or Equal**

<table>
<thead>
<tr>
<th>Jump</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>jge Dest</td>
<td>7 5</td>
</tr>
</tbody>
</table>

**Jump When Greater**

<table>
<thead>
<tr>
<th>Jump</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>jg Dest</td>
<td>7 6</td>
</tr>
</tbody>
</table>

- Refer to generically as “\text{j}XX”
- 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

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

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

```plaintext
<table>
<thead>
<tr>
<th>a</th>
<th>5043</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6125</td>
</tr>
<tr>
<td></td>
<td>7395</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```

⇒ 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;
}
```

- 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
- Compile with gcc -O2 -S

/* 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

Y86 Code
L24:
    movl (%edx),%eax
    inc %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
rmovl 8(%ebp),%edx  # Get a
rmovl (%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:  | mrmmovl (%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

- irmovl Stack,%esp # Set up stack
- rrmovl %esp,%ebp  # Set up frame
- irmovl List,%edx  # Set up frame
- pushl %edx        # Push argument
- call len2         # Call Function
- halt              # Halt
- .align 4          # List of elements
- List:             
  .long 5043
  .long 6125
  .long 7395
  .long 0

- # Function
- len2:             
  ...               

- # Allocate space for stack
- .pos 0x100
- Stack:
Assembling Y86 Program

\texttt{unix> yas eg ys}

- Generates “object code” file \texttt{eg yo}
- Actually looks like disassembler output

```
0x000: 308400010000 | irmovl Stack,\esp # Set up stack
0x006: 2045 | rmovl \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

\texttt{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 0x0000000c
%ebp: 0x00000000 0x00000100
%esi: 0x00000000 0x00000004

Changes to memory:
0x0004: 0x00000000 0x00000100
0x0008: 0x00000000 0x00000015
0x000c: 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 `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>Caller Save</td>
</tr>
<tr>
<td>$5</td>
<td>Temporaries: May be overwritten by called procedures</td>
</tr>
<tr>
<td>$6</td>
<td>Caller Save Temp</td>
</tr>
<tr>
<td>$7</td>
<td>Reserved for Operating Sys</td>
</tr>
<tr>
<td>$8</td>
<td>Global Pointer</td>
</tr>
<tr>
<td>$9</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>$10</td>
<td>Callee Save Temp</td>
</tr>
<tr>
<td>$11</td>
<td>May not be overwritten by called procedures</td>
</tr>
<tr>
<td>$12</td>
<td>Return Address</td>
</tr>
</tbody>
</table>

MIPS Instruction Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu $3,$2,$1</td>
<td>Register add: $3 = $2+$1</td>
</tr>
<tr>
<td>addu $3,$2,3145</td>
<td>Immediate add: $3 = $2+3145</td>
</tr>
<tr>
<td>sll $3,$2,2</td>
<td>Shift left: $3 = $2 &lt;&lt; 2</td>
</tr>
<tr>
<td>beq $3,$2,dest</td>
<td>Branch when $3 = $2</td>
</tr>
<tr>
<td>lw $3,16($2)</td>
<td>Load Word: $3 = M[$2+16]</td>
</tr>
<tr>
<td>sw $3,16($2)</td>
<td>Store Word: M[$2+16] = $3</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