JDEP 284H
Foundations of Computer Systems

Processor Architecture I:
Y86 Instruction Set Architecture

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/JDEP284

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

Application Program	
Compiler	OS
ISA	
CPU Design	
Circuit Design	
Chip Layout	
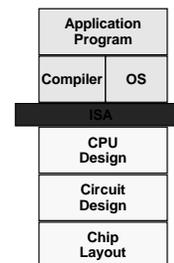
Y86 Processor State

Program registers		Condition codes	Memory
%eax	%esi	OF ZF SF	
%ecx	%edi	PC	
%edx	%esp		
%ebx	%ebp		
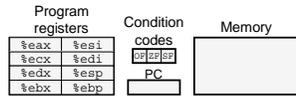
- **Program Registers**
  - Same 8 as with IA32. Each 32 bits
- **Condition Codes**
  - Single-bit flags set by arithmetic or logical instructions
    - » OF: Overflow ZF: Zero SF:Negative
- **Program Counter**
  - Indicates address of instruction
- **Memory**
  - Byte-addressable storage array
  - Words stored in little-endian byte order

7

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**

8

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**

9

Instruction Example

Addition Instruction

Generic Form

Encoded Representation

`addl rA, rB`   `6 0 rA rB`
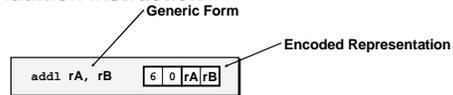
- **Add value in register rA to that in register rB**
  - Store result in register rB
  - Note that Y86 only allows addition to be applied to register data
- **Set condition codes based on result**
- **Two-byte encoding**
  - First indicates instruction type
  - Second gives source and destination registers
- **e.g.,** `addl %eax,%esi`   **Encoding:** `60 06`

10

Arithmetic and Logical Operations

Instruction Code   Function Code

Add

`addl rA, rB`   `6 0 rA rB`

Subtract (rA from rB)

`subl rA, rB`   `6 1 rA rB`

And

`andl rA, rB`   `6 2 rA rB`

Exclusive-Or

`xorl rA, rB`   `6 3 rA rB`
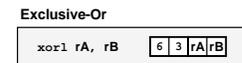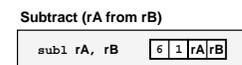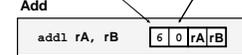
- **Refer to generically as "OP1"**
- **Encodings differ only by "function code"**
  - Low-order 4 bytes in first instruction word
- **Set condition codes as side effect**

11

Move Operations

`rrmovl rA, rB`	`2 0 rA rB`	Register --> Register
`irmovl V, rB`	`3 0 8 rB` V	Immediate --> Register
`rmmovl rA, D(rB)`	`4 0 rA rB` D	Register --> Memory
`mrmovl D(rB), rA`	`5 0 rA rB` D	Memory --> Register
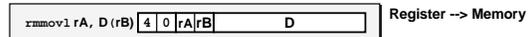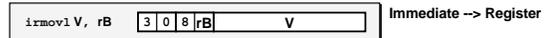
- **Like the IA32 `movl` instruction**
- **Simpler format for memory addresses**
- **Give different names to keep them distinct**

12

Move Instruction Examples

IA32	Y86	Encoding
movl $0xabcd, %edx	irmovl $0xabcd, %edx	30 82 cd ab 00 00
movl %esp, %ebx	rrmovl %esp, %ebx	20 43
movl -12(%ebp),%ecx	mrmovl -12(%ebp),%ecx	50 15 f4 ff ff ff
movl %esi,0x41c(%esp)	rmmovl %esi,0x41c(%esp)	40 64 1c 04 00 00

movl $0xabcd, (%eax)	—	
movl %eax, 12(%eax,%edx)	—	
movl (%ebp,%eax,4),%ecx	—	

13

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 |

- ■ **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**

14

Y86 Program Stack

Stack "Bottom"

Increasing Addresses

%esp
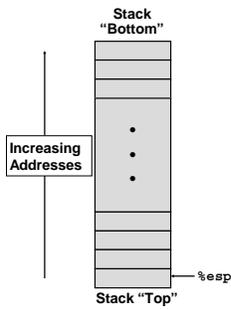
Stack "Top"

- ■ **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**

15

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**

16

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 |

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

17

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**

18

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[]);
  ```

```
a →  5043
     6125      ⇒ 3
     7395
      0
```
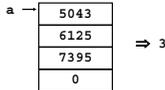
19

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;
}
```

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
```

- Compile with `gcc -O2 -S`

20

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;
}
```

Result
- Don't need to do indexed addressing

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

- Compile with `gcc -O2 -S`

21

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
```

22

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++
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
```

23

Y86 Program Structure

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

24

Page 4

Assembling Y86 Program

```
unix> yas eg.ys
```

- **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**

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

$0	$0	Constant 0	$16	$s0	
$1	$at	Reserved Temp.	$17	$s1	
$2	$v0	Return Values	$18	$s2	Callee Save Temporaries: May not be overwritten by called procedures
$3	$v1		$19	$s3	
$4	$a0		$20	$s4	
$5	$a1	Procedure arguments	$21	$s5	
$6	$a2		$22	$s6	
$7	$a3		$23	$s7	
$8	$t0		$24	$t8	Caller Save Temp
$9	$t1		$25	$t9	
$10	$t2	Caller Save Temporaries: May be overwritten by called procedures	$26	$k0	Reserved for Operating Sys
$11	$t3		$27	$k1	
$12	$t4		$28	$gp	Global Pointer
$13	$t5		$29	$sp	Stack Pointer
$14	$t6		$30	$s8	Callee Save Temp
$15	$t7		$31	$ra	Return Address

MIPS Instruction Examples

R-R

Op	Ra	Rb	Rd	00000	Fn

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

R-I

Op	Ra	Rb	Immediate

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

Branch

Op	Ra	Rb	Offset

```
beq $3,$2,dest    # Branch when $3 = $2
```

Load/Store

Op	Ra	Rb	Offset

```
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

31

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

32