Chapter 4: Processor Architecture

Topics (Ch 4.1)

- Y86 Instruction Set Architecture

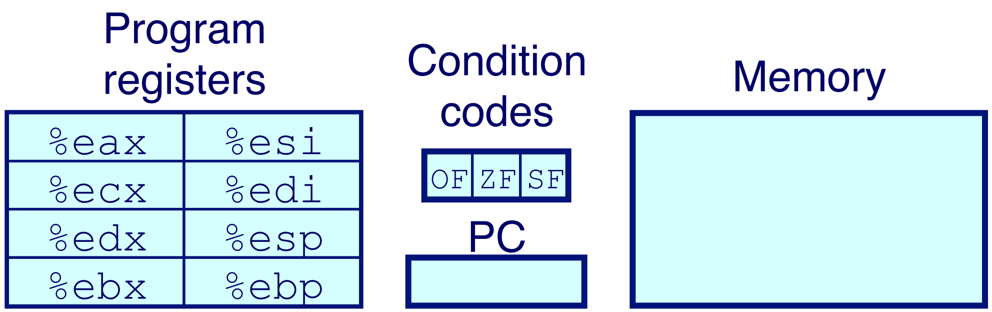
Binary Object Code

```
00401040 <_sum>:
   0:        55                 push   %ebp
   1:        89 e5              mov    %esp,%ebp
   3:        8b 45 0c           mov    0xc(%ebp),%eax
   6:        03 45 08           add    0x8(%ebp),%eax
   9:        89 ec              mov    %ebp,%esp
   b:        5d                 pop    %ebp
   c:        c3                 ret
   d:        8d 76 00           lea    0x0(%esi),%esi
```

Encoding & Execution

- How is it that “55” represents “push %ebp”
- How is it that “03 45 08” represents “add 0x8(%ebp), %eax”?
- Note how the encodings are variable length
- How does the CPU execute each instruction?

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

`%eax`	**0**
`%ecx`	**1**
`%edx`	**2**
`%ebx`	**3**

`%esi`	**6**
`%edi`	**7**
`%esp`	**4**
`%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

`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
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers
- e.g., `addl %eax,%esi` Encoding: `60 06`
- Set condition codes based on result

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 "`OPl`"
- Encodings differ only by "function code"
  - Low-order 4 bytes in first instruction word
- Operate only on register data, not memory
  - Unlike IA32, where one of operands could be a memory location
  - Separate move instructions to operate on memory
- Set condition codes as side effect

Move Operations

Instruction						Description
`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
- Supports only simple addressing mode: D(rX)

Move Instruction Examples

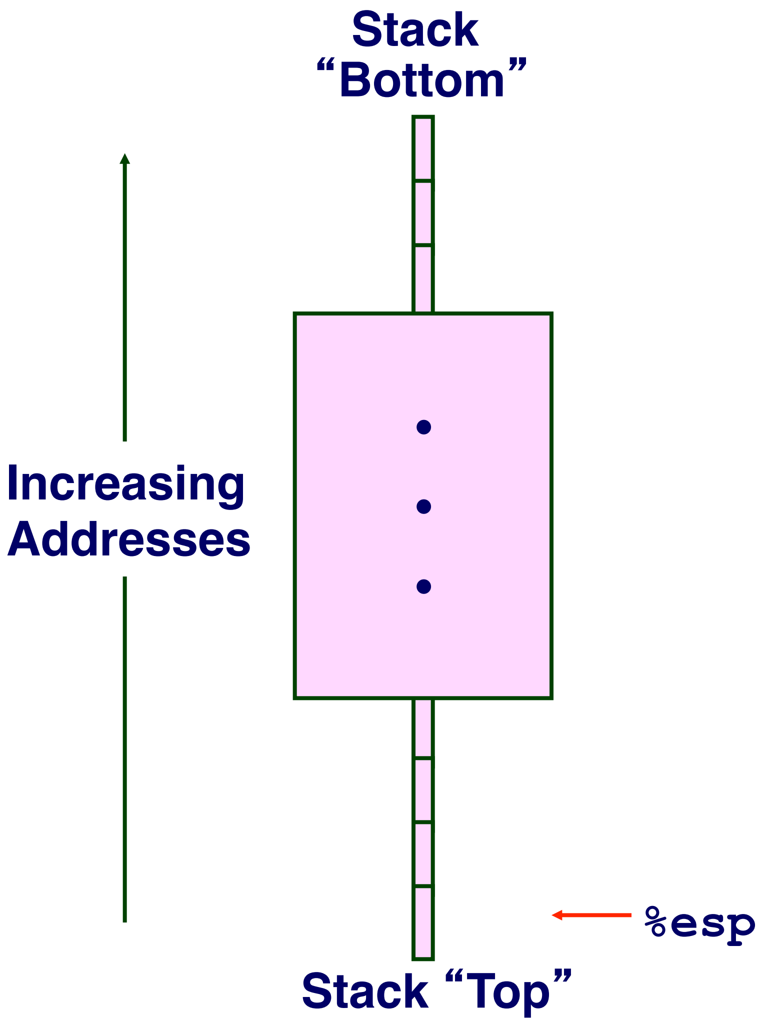
IA32	Y86	Encoding
`movl $0xabcd, %edx`	`irmovl $0xabcd, %edx`	`30 82 cd ab 00 00`
`movl %esp, %ebx`	`rrmovl %esp, %ebx`	`20 43`
`movl -12(%ebp),%ecx`	`mrmovl -12(%ebp),%ecx`	`50 15 f4 ff ff ff`
`movl %esi,0x41c(%esp)`	`rmmovl %esi,0x41c(%esp)`	`40 64 1c 04 00 00`
`movl $0xabcd, (%eax)`	—	
`movl %eax, 12(%eax,%edx)`	—	
`movl (%ebp,%eax,4),%ecx`	—	

Jump Instructions

Description	Instruction	Byte 0		Bytes 1–4
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**

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`	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

- Pop value from stack
- 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

Y86 Instruction Set

Byte	0	1	2	3	4	5
`nop`	0 0					
`halt`	1 0					
`rrmovl` rA, rB	2 0	rA rB				
`irmovl` V, rB	3 0	8 rB	V			
`rmmovl` rA, D(rB)	4 0	rA rB	D			
`mrmovl` D(rB), rA	5 0	rA rB	D			
`OPl` rA, rB	6 fn	rA rB				
`jXX` Dest	7 fn	Dest				
`call` Dest	8 0	Dest				
`ret`	9 0					
`pushl` rA	A 0	rA 8				
`popl` rA	B 0	rA 8				

`addl`	6 0
`subl`	6 1
`andl`	6 2
`xorl`	6 3

`jmp`	7 0
`jle`	7 1
`jl`	7 2
`je`	7 3
`jne`	7 4
`jge`	7 5
`jg`	7 6

Writing Y86 Code

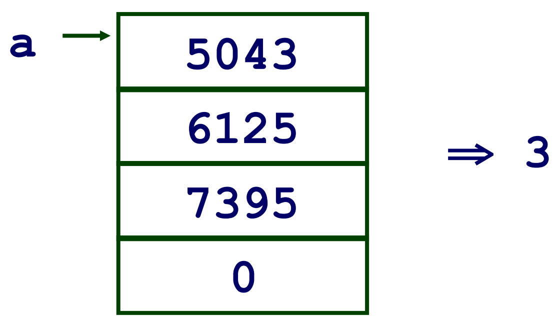
Try to Use C Compiler as Much as Possible

- Write code in C and compile for IA32 with `gcc -S`
- Transliterate into Y86
- Wrote an assembler called YAS

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
```

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

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++
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
```

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
```

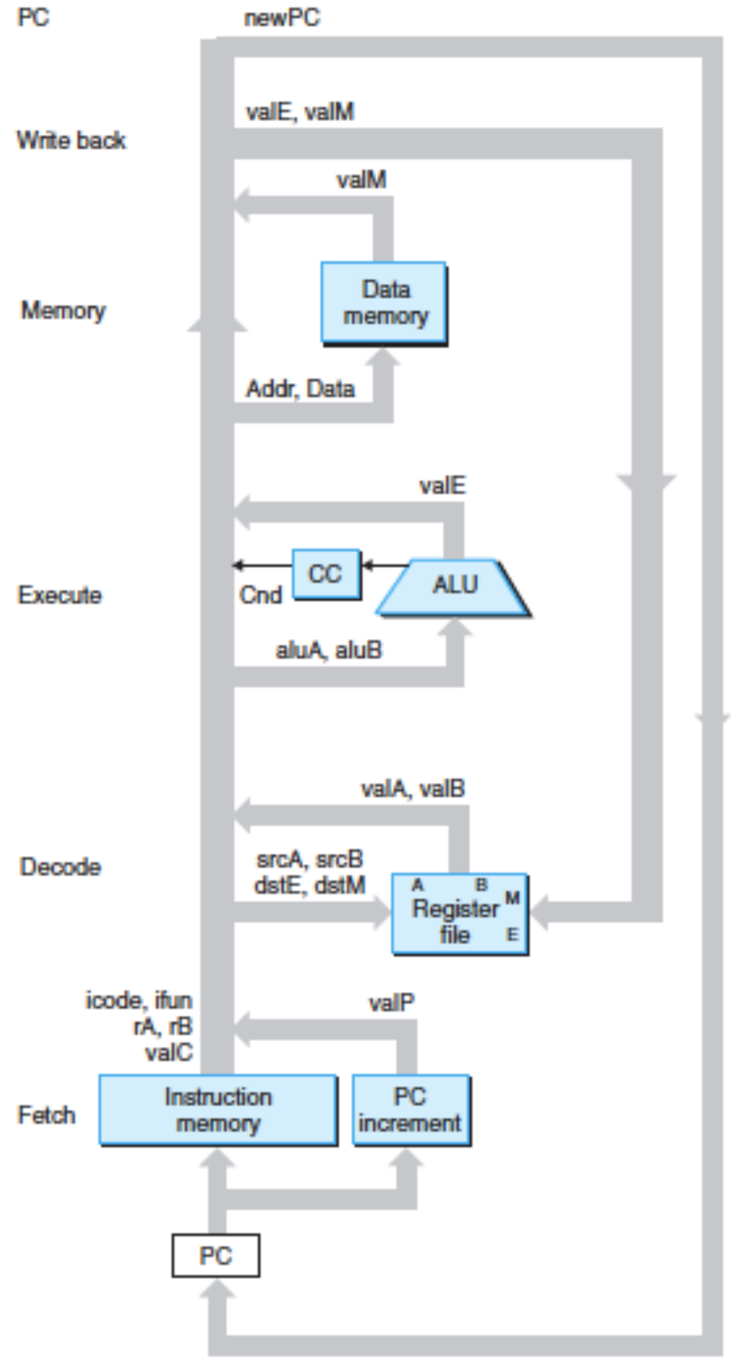
- Instruction set simulator
  - Computes effect of each instruction on processor state
  - Prints changes in state from original

```
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
```

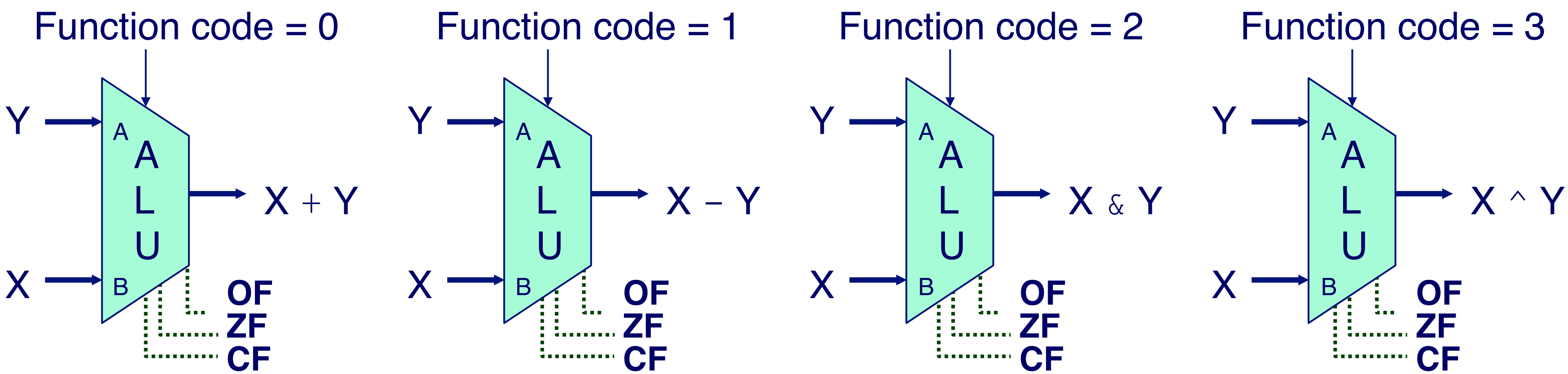
Executing CPU Instructions

Each Y86 instruction can be divided into 6 stages of execution, e.g. addl %edx,%ecx and mrmovl 4(%edx), %ecx

- Fetch instruction
- Decode Instruction
- Execute Instruction
- Store results to memory or retrieve values from memory
- Write back results to registers
- Update CPU state
  - Update PC



Arithmetic Logic Unit



- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86
- Also computes values for condition codes

Supplementary Slides

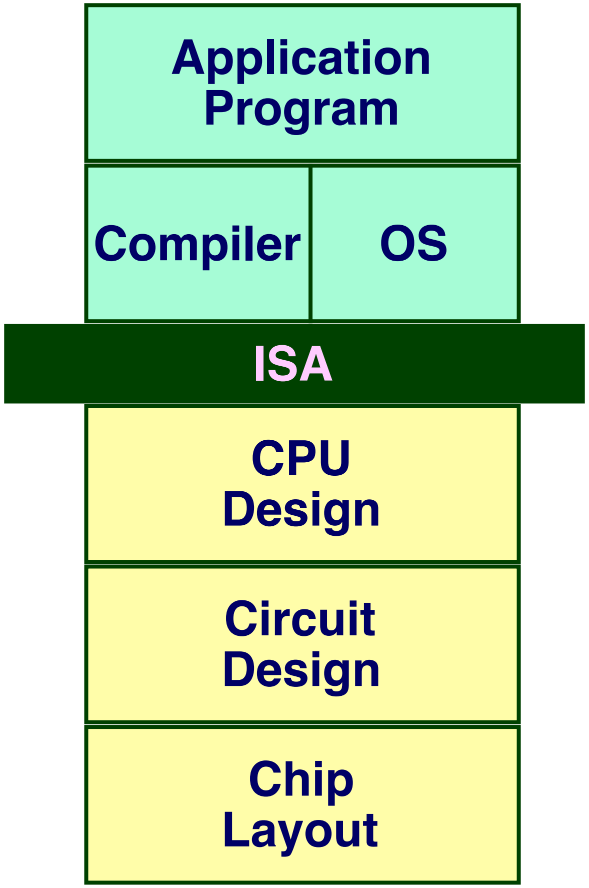
Instruction Set Architecture

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 Program Structure

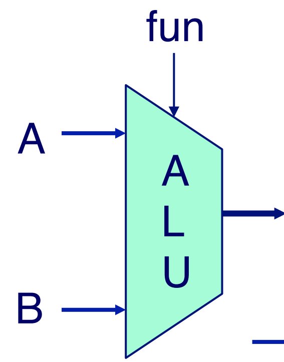
```
   irmovl Stack,%esp   # Set up stack
   rrmovl %esp,%ebp    # Set up frame
   irmovl List,%edx
   pushl %edx          # Push argument
   call len2           # Call Function
   halt                # Halt
.align 4
List:                  # List of elements
   .long 5043
   .long 6125
   .long 7395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

# Function
len2:
   . . .

# Allocate space for stack
.pos 0x100
Stack:
```

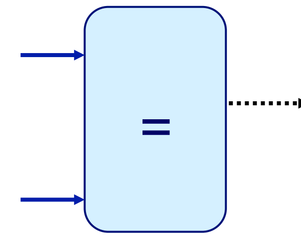
- Program starts at address 0
- Must set up stack
  - Make sure don't overwrite code!
- Must initialize data
- Can use symbolic names

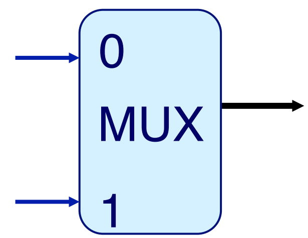
Building Blocks

Combinational Logic

- Compute Boolean functions of inputs
- Continuously respond to input changes
- Operate on data and implement control







Storage Elements

- Store bits
- Addressable memories
- Non-addressable registers
    - Loaded only as clock rises