# CS:APP Chapter 4
# Computer Architecture

# Instruction Set Architecture

**Randal E. Bryant**

**adapted by Jason Fritts**

`http://csapp.cs.cmu.edu`

# Hardware Architecture - using Y86 ISA

## For learning aspects of hardware architecture design, we'll be using the Y86 ISA

- x86 is a CISC language
  - too complex for educational purposes

## Y86 Instruction Set Architecture

- a pseudo-language based on x86 (IA-32)
- similar state, but simpler set of instructions
- simpler instruction formats and addressing modes
- more RISC-like ISA than IA-32

## Format

- 1–6 bytes of information read from memory
  - can determine instruction length from first byte

# 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`

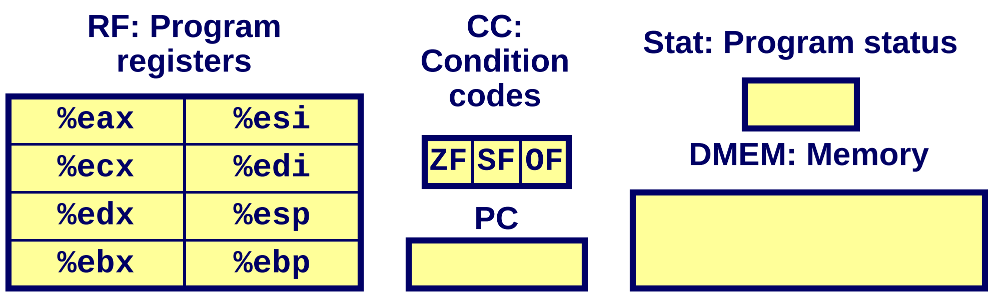
## No Condition codes

- Test instructions return 0/1 in register

# Y86 Instruction Set and Formatting

| Byte | 0 | | 1 | | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|
| `halt` | 0 | 0 | | | | | | |
| `nop` | 1 | 0 | | | | | | |
| `rrmovl rA, rB` | 2 | 0 | rA | rB | | | | |
| `irmovl V, rB` | 3 | 0 | F | 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 | F | | | | |
| `popl rA` | B | 0 | rA | F | | | | |

# Y86 Processor State




- **Program Registers**
  - **Same 8 as with IA32.  Each 32 bits**
- **Condition Codes**
  - **Single-bit flags set by arithmetic or logical instructions**
    - » ZF: Zero  SF:Negative  OF: Overflow
- **Program Counter**
  - **Indicates address of next instruction**
- **Program Status**
  - **Indicates either normal operation or some error condition**
- **Memory**
  - **Byte-addressable storage array**
  - **Words stored in little-endian byte order**

# Y86 Instruction Set #2

| Byte | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| `halt` | 0 0 | | | | | |
| `nop` | 1 0 | | | | | |
| `rrmovl rA, rB` | 2 0 | rA rB | | | | |
| `irmovl V, rB` | 3 0 | F 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 F | | | | |
| `popl rA` | B 0 | rA F | | | | |

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

# Y86 Instruction Set #3

| Byte | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| halt | 0 0 | | | | | |
| nop | 1 0 | | | | | |
| rrmovl rA, rB | 2 0 | rA rB | | | | |
| irmovl V, rB | 3 0 | F 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 F | | | | |
| popl rA | B 0 | rA F | | | | |

| jXX | Byte 0 |
|---|---|
| jmp | 7 0 |
| jle | 7 1 |
| jl | 7 2 |
| je | 7 3 |
| jne | 7 4 |
| jge | 7 5 |
| jg | 7 6 |

# Encoding Registers

## Each register has 4-bit ID

| | |
|---|---|
| %eax | 0 |
| %ecx | 1 |
| %edx | 2 |
| %ebx | 3 |

| | |
|---|---|
| %esi | 6 |
| %edi | 7 |
| %esp | 4 |
| %ebp | 5 |

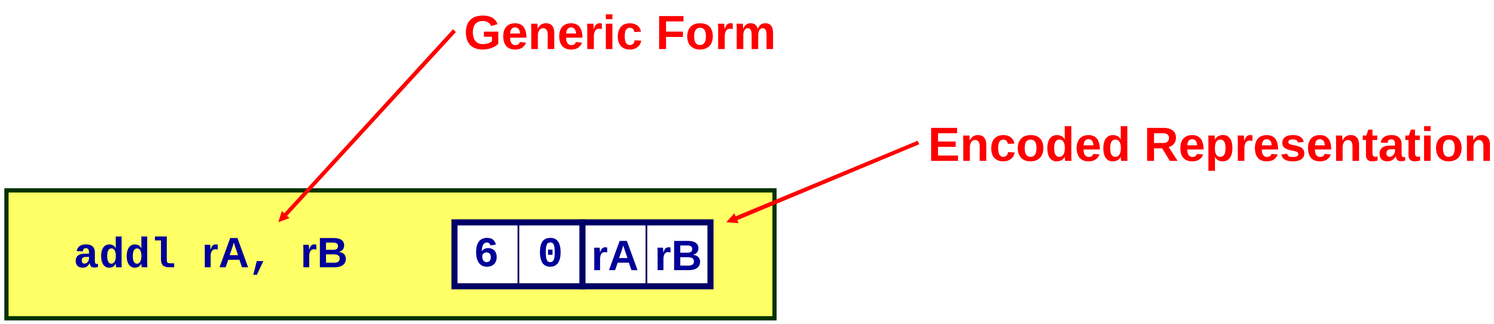
- **Same encoding as in IA32**

## Register ID 15 (0xF) indicates "no register"

- **Will use this in our hardware design in multiple places**

# 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

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
- Set condition codes as side effect

# Move Operations

| Instruction | Encoding | | | | | Description |
|---|---|---|---|---|---|---|
| `rrmovl rA, rB` | 2 | 0 | rA | rB | | Register --> Register |
| `irmovl V, rB` | 3 | 0 | F | 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

# Move Instruction Examples

| IA32 | Y86 | Encoding |
|---|---|---|
| `movl $0xabcd, %edx` | `irmovl $0xabcd, %edx` | `30 f2 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

**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

# Stack Operations

`pushl rA`

| | | | |
|---|---|---|---|
| A | 0 | rA | F |

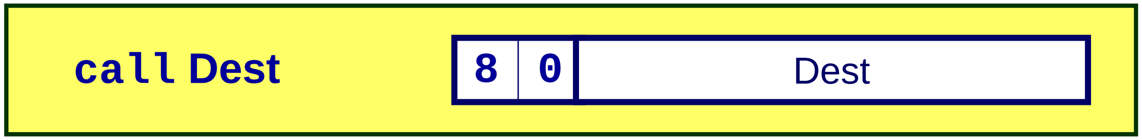
- Decrement %esp by 4
- Store word from rA to memory at %esp
- Like IA32

`popl rA`

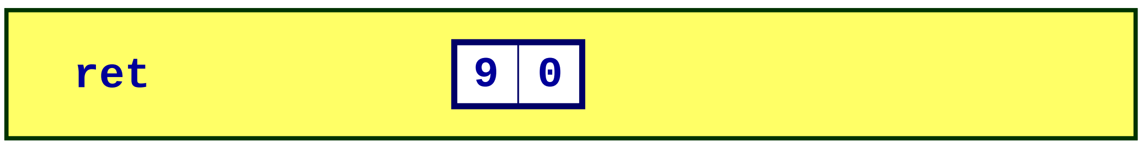
| | | | |
|---|---|---|---|
| B | 0 | rA | F |

- 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




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

# Miscellaneous Instructions

| | | |
|---|---|---|
| `nop` | 1 | 0 |

- Don't do anything

| | | |
|---|---|---|
| `halt` | 0 | 0 |

- Stop executing instructions
- IA32 has comparable instruction, but can't execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

# Status Conditions

| Mnemonic | Code |
|---|---|
| AOK | 1 |

- Normal operation

| Mnemonic | Code |
|---|---|
| HLT | 2 |

- Halt instruction encountered

| Mnemonic | Code |
|---|---|
| ADR | 3 |

- Bad address (either instruction or data) encountered

| Mnemonic | Code |
|---|---|
| INS | 4 |

- Invalid instruction encountered

## Desired Behavior

- If AOK, keep going
- Otherwise, stop program execution

# 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 `gcc34 –O1 -S`

## Result

- Don't need to do indexed addressing

```
.L11:
   incl      %ecx
   movl      (%edx), %eax
   addl      $4, %edx
   testl     %eax, %eax
   jne .L11
```

# Y86 Code Generation Example #3

## IA32 Code

- Setup

```
len2:
   pushl %ebp
   movl %esp, %ebp



   movl 8(%ebp), %edx
   movl $0, %ecx
   movl (%edx), %eax
   addl $4, %edx
   testl %eax, %eax
   je   .L13
```

- Need constants 1 & 4
- Store in callee-save registers

## Y86 Code

- Setup

```
len2:
   pushl %ebp              # Save %ebp
   rrmovl %esp, %ebp       # New FP
   pushl %esi              # Save
   irmovl $4, %esi         # Constant 4
   pushl %edi              # Save
   irmovl $1, %edi         # Constant 1
   mrmovl 8(%ebp), %edx        # Get a
   irmovl $0, %ecx             # len = 0
   mrmovl (%edx), %eax         # Get *a
   addl %esi, %edx             # a++
   andl %eax, %eax             # Test *a
   je Done     # If zero, goto Done
```

- Use `andl` to test register

# Y86 Code Generation Example #4

## IA32 Code

- Loop & Exit

```
.L11:
   incl %ecx
   movl (%edx), %eax
   addl $4, %edx
   testl %eax, %eax
   jne .L11

.L13:
   movl %ecx, %eax


   leave

   ret
```

## Y86 Code

- Loop & Exit

```
Loop:
   addl %edi, %ecx          # len++
   mrmovl (%edx), %eax      # Get *a
   addl %esi, %edx          # a++
   andl %eax, %eax          # Test *a
   jne Loop     # If !0, goto Loop

Done:
   rrmovl %ecx, %eax   # return len
   popl %edi     # Restore %edi
   popl %esi     # Restore %esi
   rrmovl %ebp, %esp # Restore SP
   popl %ebp         # Restore FP
   ret
```

# 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 has evolved from IA32 to x86-64**
  - **Uses 64-bit words (including addresses)**
  - **Adopted some features found in RISC**
    - » More registers (16)
    - » Less reliance on stack