Architecture
Y86 ISA

CSCI 2021: Machine Architecture and Organization

Antonia Zhai

Department Computer Science and Engineering

University of Minnesota

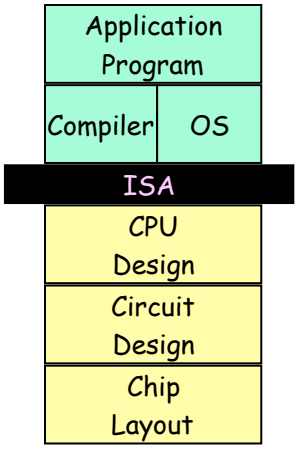
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O'Hallaron

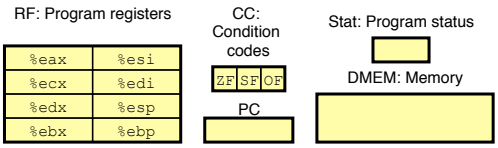
UNIVERSITY OF MINNESOTA

Instruction Set Architecture

- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addl`, `pushl`, `ret`, …
    - 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



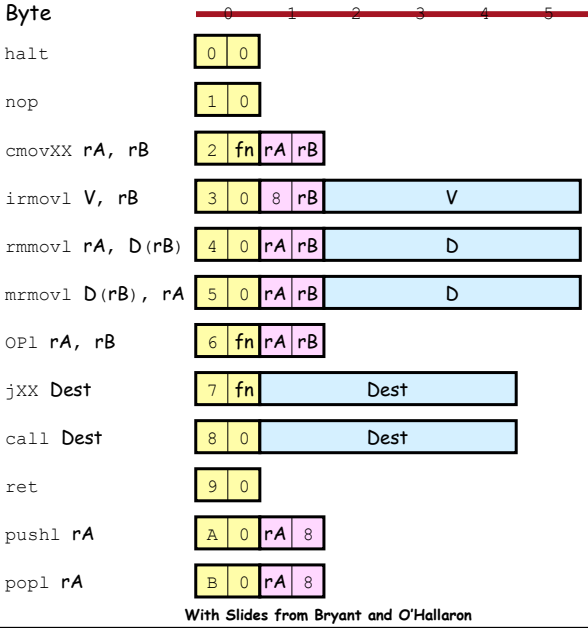
Y86 Processor State



- Program Registers
  - Same 8 as with IA32. Each 32 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - 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

With Slides from Bryant and O'Hallaron

Y86 Instruction Set #1



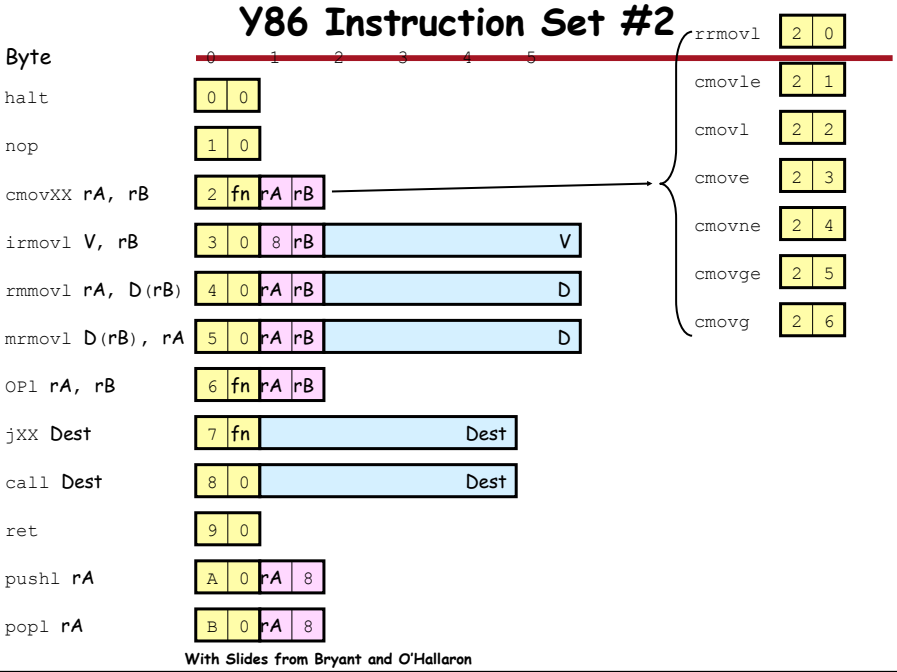
With Slides from Bryant and O'Hallaron

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

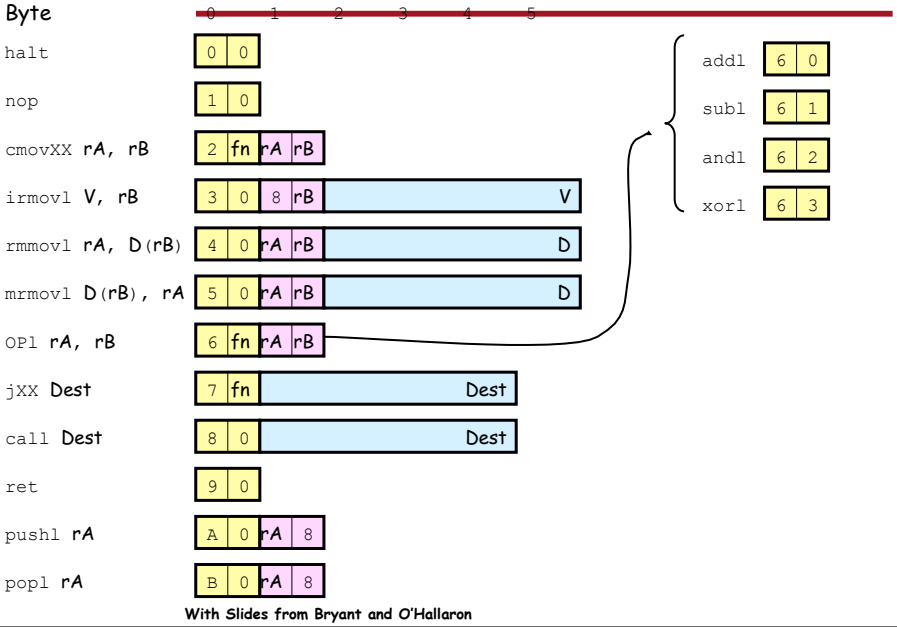
With Slides from Bryant and O'Hallaron

Y86 Instruction Set #2



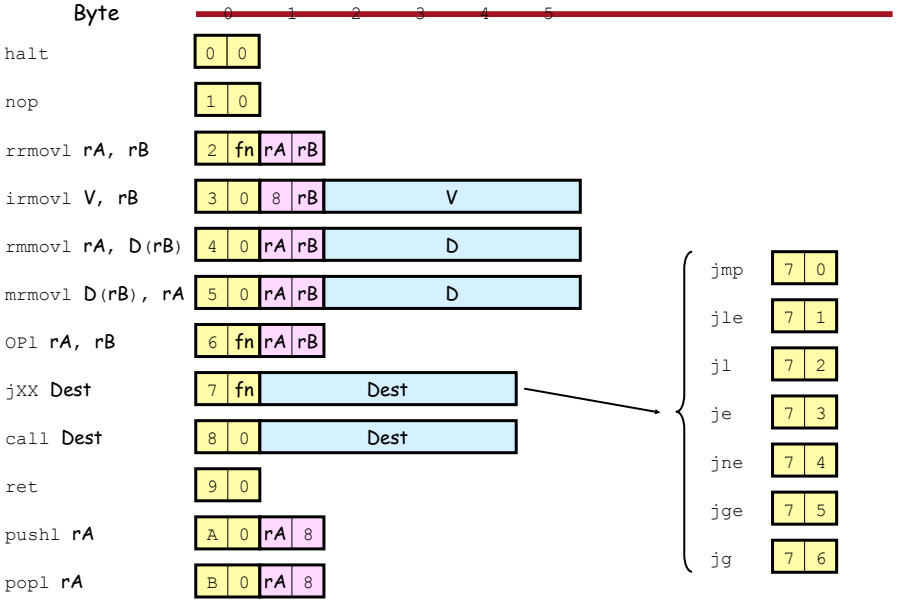
With Slides from Bryant and O'Hallaron

Y86 Instruction Set #3



With Slides from Bryant and O'Hallaron

Y86 Instruction Set #4



With Slides from Bryant and O'Hallaron

Encoding Registers

- Each register has 4-bit ID

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

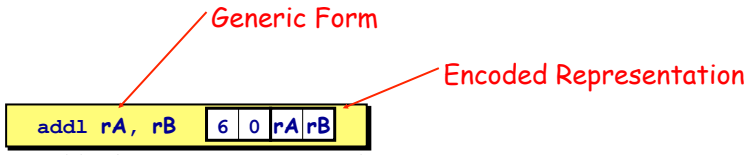
%esi	6
%edi	7
%esp	4
%ebp	5

  - Same encoding as in IA32
- Register ID 15 (0xF) indicates "no register"
  - Will use this in our hardware design in multiple places

With Slides from Bryant and O'Hallaron

Instruction Example

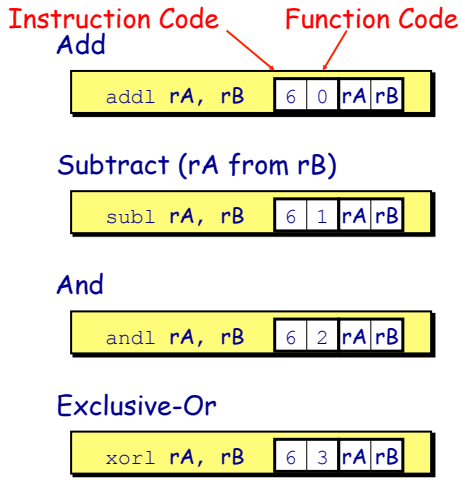
- Addition Instruction



  - 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

With Slides from Bryant and O'Hallaron

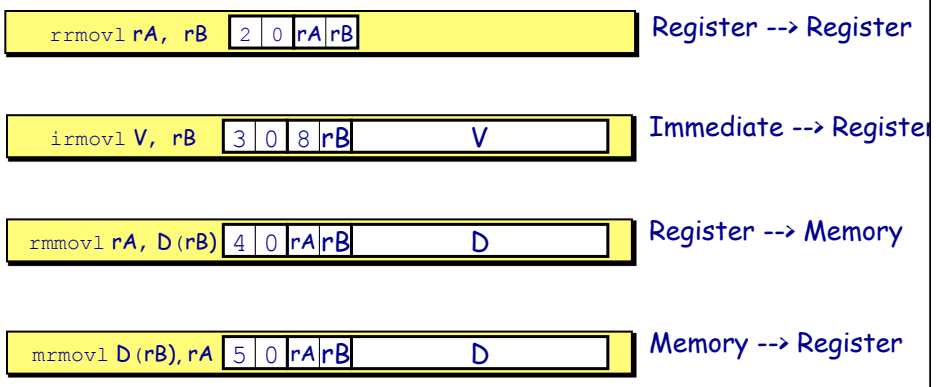
Arithmetic and Logical Operations



- 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

With Slides from Bryant and O'Hallaron

Move Operations



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

With Slides from Bryant and O'Hallaron

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`	—

With Slides from Bryant and O'Hallaron

Conditional Move Instructions

Move Unconditionally

`rrmovl` rA, rB — 2 0 rA rB

Move When Less or Equal

`cmovle` rA, rB — 2 1 rA rB

Move When Less

`cmovl` rA, rB — 2 2 rA rB

Move When Equal

`cmove` rA, rB — 2 3 rA rB

Move When Not Equal

`cmovne` rA, rB — 2 4 rA rB

Move When Greater or Equal

`cmovge` rA, rB — 2 5 rA rB

Move When Greater

`cmovg` rA, rB — 2 6 rA rB

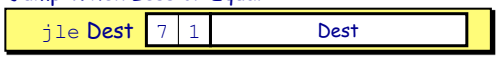
- Refer to generically as "`cmovXX`"
- Encodings differ only by "function code"
- Based on values of condition codes
- Variants of `rrmovl` instruction
  - (Conditionally) copy value from source to destination register

With Slides from Bryant and O'Hallaron

Jump Instructions

Jump Unconditionally

Jump When Less or Equal



Jump When Less

Jump When Equal

Jump When Not Equal

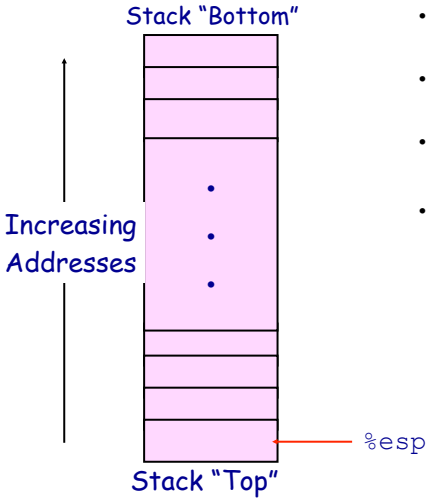
Jump When Greater or Equal

Jump When Greater

- 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

With Slides from Bryant and O'Hallaron

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
  - After popping, increment stack pointer

With Slides from Bryant and O'Hallaron

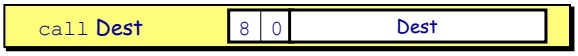
Stack Operations

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

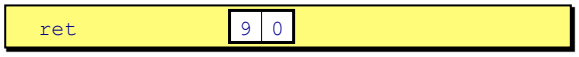
- Read word from memory at %esp
- Save in rA
- Increment %esp by 4
- Like IA32

With Slides from Bryant and O'Hallaron

Subroutine Call and Return



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

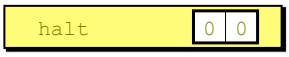


- Pop value from stack
- Use as address for next instruction
- Like IA32

With Slides from Bryant and O'Hallaron

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
- Encoding ensures that program hitting memory initialized to zero will halt

With Slides from Bryant and O'Hallaron

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

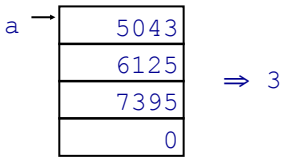
- Desired Behavior
  - If AOK, keep going
  - Otherwise, stop program execution

With Slides from Bryant and O'Hallaron

Writing Y86 Code

- Try to Use C Compiler as Much as Possible
  - Write code in C
  - Compile for IA32 with `gcc34 -O1 -S`
    - Newer versions of GCC do too much optimization
    - Use `ls /usr/bin/gcc*` to find what versions are available
  - Transliterate into Y86
- Coding Example
  - Find number of elements in null-terminated list

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



With Slides from Bryant and O'Hallaron

Y86 Code Generation Example

- First Try
  - Write typical array code
  - Compile with `gcc34 -O1 -S`
- Problem
  - Hard to do array indexing on Y86
    - Since don't have scaled addressing modes

```
/* Find number of elements in
   null-terminated list */
int len1(int a[])
{
  int len;
  for (len = 0; a[len]; len++)
      ;
  return len;
}
```

```
L5:
   incl   %eax
   cmpl   $0, (%edx,%eax,4)
   jne L5
```

With Slides from Bryant and O'Hallaron

Y86 Code Generation Example #2

- Second Try
  - Write with pointer code
  - Compile with `gcc -O1 -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

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

With Slides from Bryant and O'Hallaron

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**

With Slides from Bryant and O'Hallaron

Y86 Code Generation Example #4

- IA32 Code
  - Loop

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

- Y86 Code
  - Loop

```
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
```

With Slides from Bryant and O'Hallaron

Y86 Code Generation Example #5

- IA32 Code
  - Finish

```
.L13:
  movl %ecx, %eax



  leave

  ret
```

- Y86 Code
  - Finish

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

With Slides from Bryant and O'Hallaron

Y86 Sample Program Structure #1

```
init:                   # Initialization
   . . .
   call Main
   halt

   .align 4             # Program data
array:
   . . .

Main:                   # Main function
   . . .
   call len2
   . . .

len2:                   # Length function
   . . .

   .pos 0x100           # Placement of stack
Stack:
```

- Program starts at address 0
- Must set up stack
  - Where located
  - Pointer values
  - Make sure don't overwrite code!
- Must initialize data

With Slides from Bryant and O'Hallaron

Y86 Program Structure #2

```
init:
   irmovl Stack, %esp  # Set up SP
   irmovl Stack, %ebp  # Set up FP
   call Main           # Execute main
   halt                # Terminate

# Array of 4 elements + terminating
0
   .align 4
array:
   .long 0x000d
   .long 0x00c0
   .long 0x0b00
   .long 0xa000
   .long 0
```

- Program starts at address 0
- Must set up stack
- Must initialize data
- Can use symbolic names

With Slides from Bryant and O'Hallaron

Y86 Program Structure #3

```
Main:
   pushl %ebp
   rrmovl %esp,%ebp
   irmovl array,%edx
   pushl %edx        # Push array
   call len2         # Call
len2(array)
   rrmovl %ebp,%esp
   popl %ebp
   ret
```

- Set up call to len2
  - Follow IA32 procedure conventions
  - Push array address as argument

With Slides from Bryant and O'Hallaron

Assembling Y86 Program

```
unix> yas len.ys
```

- Generates "object code" file `len.yo`
  - Actually looks like disassembler output

```
   0x000:              |   .pos 0
   0x000: 30f400010000 | init:   irmovl Stack, %esp  # Set up stack pointer
   0x006: 30f500010000 |   irmovl Stack, %ebp     # Set up base pointer
   0x00c: 8028000000   |   call Main              # Execute main program
   0x011: 00           |   halt                   # Terminate program
                       |
                       | # Array of 4 elements + terminating 0
   0x014:              |   .align 4
   0x014:              | array:
   0x014: 0d000000     |   .long 0x000d
   0x018: c0000000     |   .long 0x00c0
   0x01c: 000b0000     |   .long 0x0b00
   0x020: 00a00000     |   .long 0xa000
   0x024: 00000000     |   .long 0
```

With Slides from Bryant and O'Hallaron

Simulating Y86 Program

```
unix> yis len.yo
```

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

```
Stopped in 50 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:                   0x00000000      0x00000004
%ecx:                   0x00000000      0x00000004
%edx:                   0x00000000      0x00000028
%esp:                   0x00000000      0x00000100
%ebp:                   0x00000000      0x00000100

Changes to memory:
0x00ec:                 0x00000000      0x000000f8
0x00f0:                 0x00000000      0x00000039
0x00f4:                 0x00000000      0x00000014
0x00f8:                 0x00000000      0x00000100
0x00fc:                 0x00000000      0x00000011
```


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

With Slides from Bryant and O'Hallaron

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
    - Most cell phones use ARM processor

With Slides from Bryant and O'Hallaron

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

With Slides from Bryant and O'Hallaron