Chapter 4
Processor Architecture: Y86
(Sections 4.1 & 4.3)

with material from Dr. Bin Ren, College of William & Mary

Outline

- Introduction to assembly programing
- Introduction to Y86
- Y86 instructions, encoding and execution

Assembly

- The CPU uses machine language to perform all its operations
- Machine code (pure numbers) is generated by translating each instruction into binary numbers that the CPU uses
- This process is called "assembling"; conversely, we can take assembled code and disassemble it into (mostly) human readable assembly language
- Assembly is a much more readable translation of machine language, and it is what we work with if we need to see what the computer is doing
- There are many different kinds of assembly languages; we'll focus on the Y86/IA32 language as defined in the text and on our system (also SPARC and MIPS)

Assembly Operations

- **Perform arithmetic function on register or memory data**
- **Transfer data between memory and register**
  - Load data from memory into register (read)
  - Store register data into memory (write)
- **Transfer control**
  - Unconditional jumps to/from procedures (calls)
  - Conditional branches (if, switch, for, while, etc)

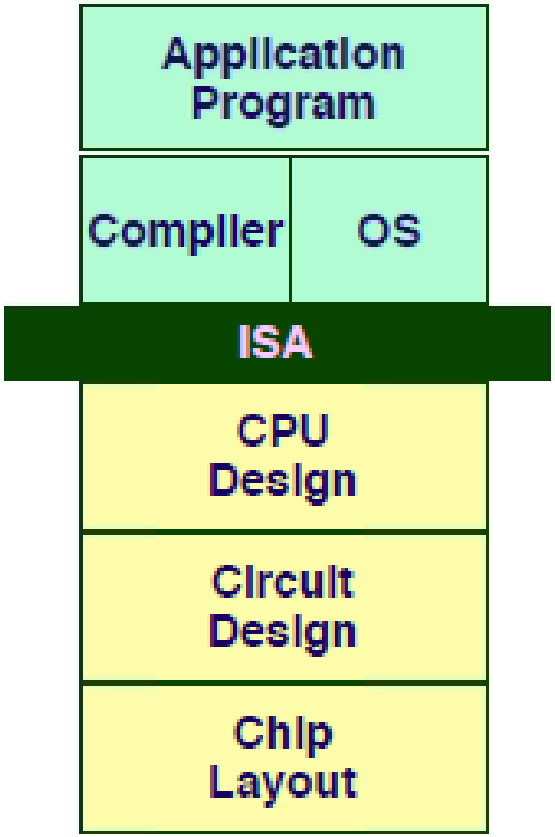
ISA – Instruction Set Architecture

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



ISA-More explanations

- **ISA – instruction set architecture**
  - Format and behavior of a machine level program
  - Defines
    - The processor state (see the CPU fetch-execute cycle)
    - The format of the instructions
    - The effect of each of these instructions on the state
  - Abstractions
    - Instruction executed “in sequence”
      - Technically defined to be completing one instruction before starting the next
      - Pipelining
      - Concurrent execution (but not really)
    - Memory addresses are virtual addresses
      - Very large byte-addressable array
      - Address space managed by the OS (virtual → physical)
      - Contains both executable code of the program AND its data
        - Run-time stack
        - Block of memory for user (global and heap)

Generic Instruction Cycle

An instruction cycle is the basic operation cycle of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction requires, and carries out those actions. This cycle is repeated continuously by the central processing unit (CPU), from bootup to when the computer is shut down.

1. **Fetching the instruction**
2. **Decode the instruction**
3. **Memory and addressing issues**
4. **Execute the instruction**

Hardware abstractions

- **Program Counter (PC)**
  - Register %eip (X86)
  - Address in memory of the next instruction to be executed
- **Integer Register File**
  - Contains eight named locations for storing 32-bit values
    - Can hold addresses (C pointers) or integer data
    - Have other special duties
- **Floating point registers**
- **Condition Code registers**
  - Hold status information
    - About arithmetic or logical instruction executed
      - CF (carry flag)
      - OF (overflow flag)
      - SF (sign flag)
      - ZF (zero flag)
- **Memory**

Machine instruction example

- **C code**
  - Add two signed integers
- **Assembly**
  - Add 2 4-byte integers
- **Operands**
  - X: register %eax
  - Y: memory M[%ebp+8]
  - T: register %eax
  - Return function value in %eax
- **Object code**
  - 3 byte instruction
  - Stored at address: 0x????????

```
int  t = x + y;

addl 8(%ebp),%eax

03 45 08
```

Outline

- Introduction to assembly programing
- Introduction to Y86
- Y86 instructions, encoding and execution

Y86: A Simpler Instruction Set

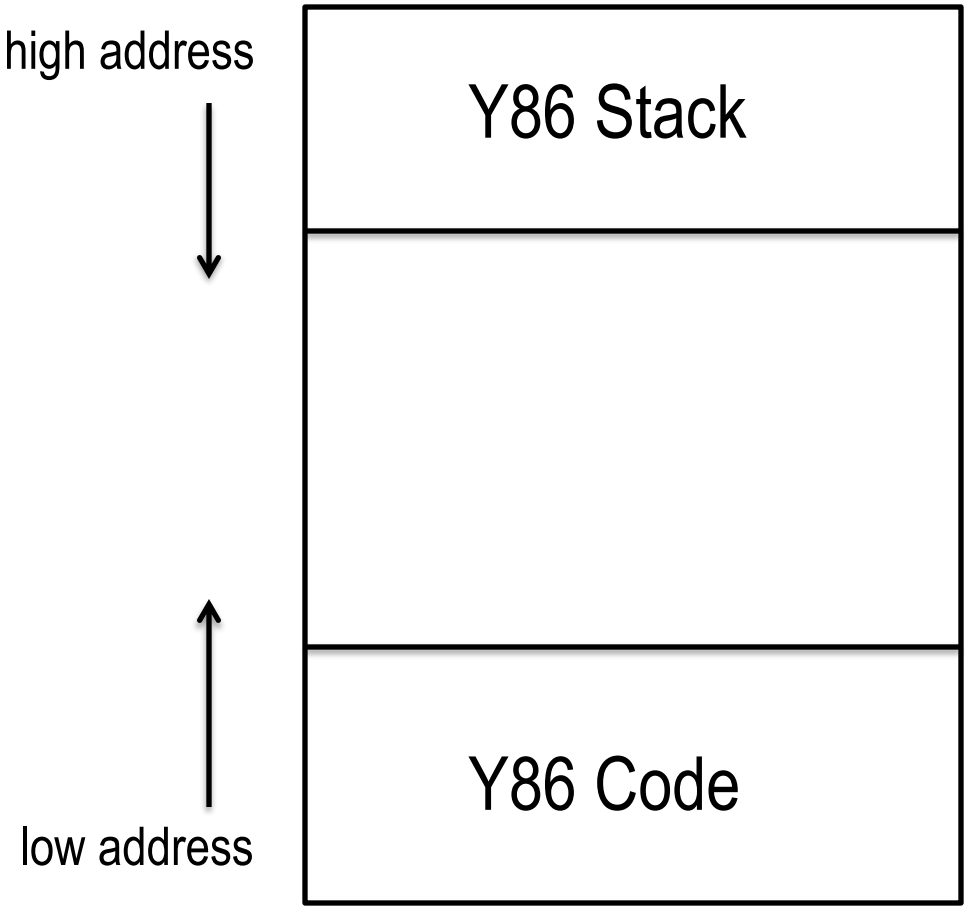
- **IA32 has a lot more instructions**
- **IA32 has a lot of quirks**
- **Y86 is a subset of IA32 instructions**
- **Y86 has a simpler encoding scheme than IA32**
- **Y86 is easier**
  - to reason about hardware
  - first-time programming in assembly language

Y86 abstractions

- **The Y86 has**
  - 8 32-bit registers with the same names as the IA32 32-bit registers
  - 3 condition codes: ZF, SF, OF
    - no carry flag
    - interprets integers as signed
  - a program counter (PC)
  - a program status byte: AOK, HLT, ADR, INS
  - memory: up to 4 GB to hold program and data
- **The Y86 does not have**
  - floating point registers or instructions

http://voices.yahoo.com/the-y86-processor-simulator-770435.html?cat=15
http://y86tutoring.wordpress.com/

Y86 Memory and Stack



1. A huge array of bytes;
2. Set the bottom of the stack far enough away from the code;
3. The location of your code should always start from 0x0.

How to set up the starting point of stack and code?

directive: .pos address-in-hex

yis and yas and the Y86 Simulator

- **check on**
  - how to set up $PATH
  - how to connect Linux with X display
- **add the following variables to $PATH**
  - /home/bren/Software/sim/misc
  - /home/bren/Software/sim/pipe
  - /home/bren/Software/sim/seq
- **otherwise, use absolute path**
  - /home/bren/Software/sim/misc/yas
  - /home/bren/Software/sim/misc/yis
- **example code was assembled during the build process and is in**
  - /home/bren/Software/sim/y86-code

YIS and YAS and the Y86 Simulator

- **how to assemble and run code**
  - % yas prog.ys
    - assembles program
    - creates *.yo file
  - % yis prog.yo
    - instruction set simulator – gives output and changes
      - changes shown as original value/new value pairs
  - % ssim -g prog.yo &
    - graphical simulator
    - SimGuide: http://csapp.cs.cmu.edu/public/simguide.pdf

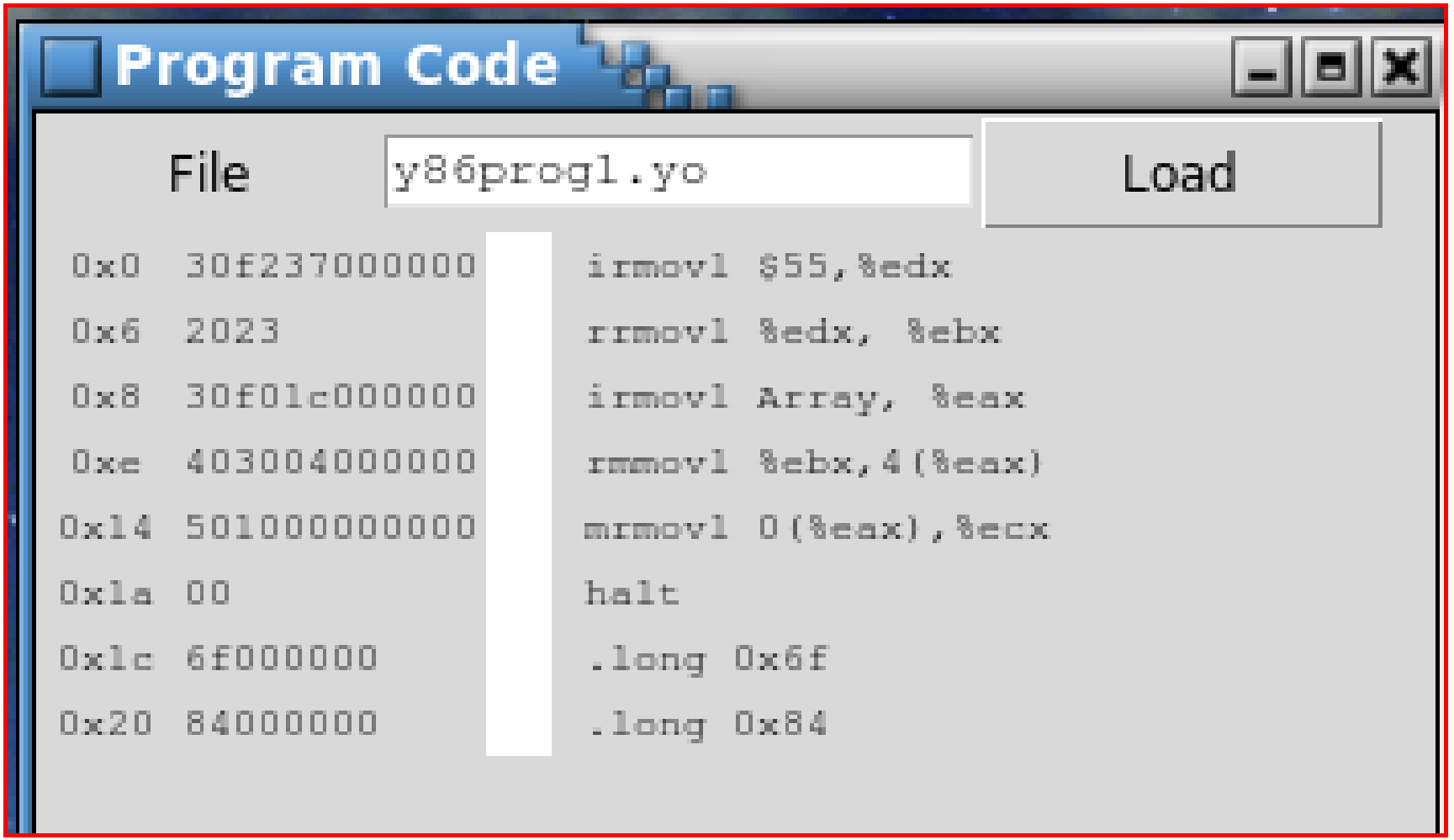
Run Y86 program

```
irmovl $55,%edx
rrmovl %edx, %ebx
irmovl Array, %eax
rmmovl %ebx,4(%eax)
mrmovl 0(%eax),%ecx
halt

.align 4
Array:
.long 0x6f
.long 0x84
```

y86prog1.ys

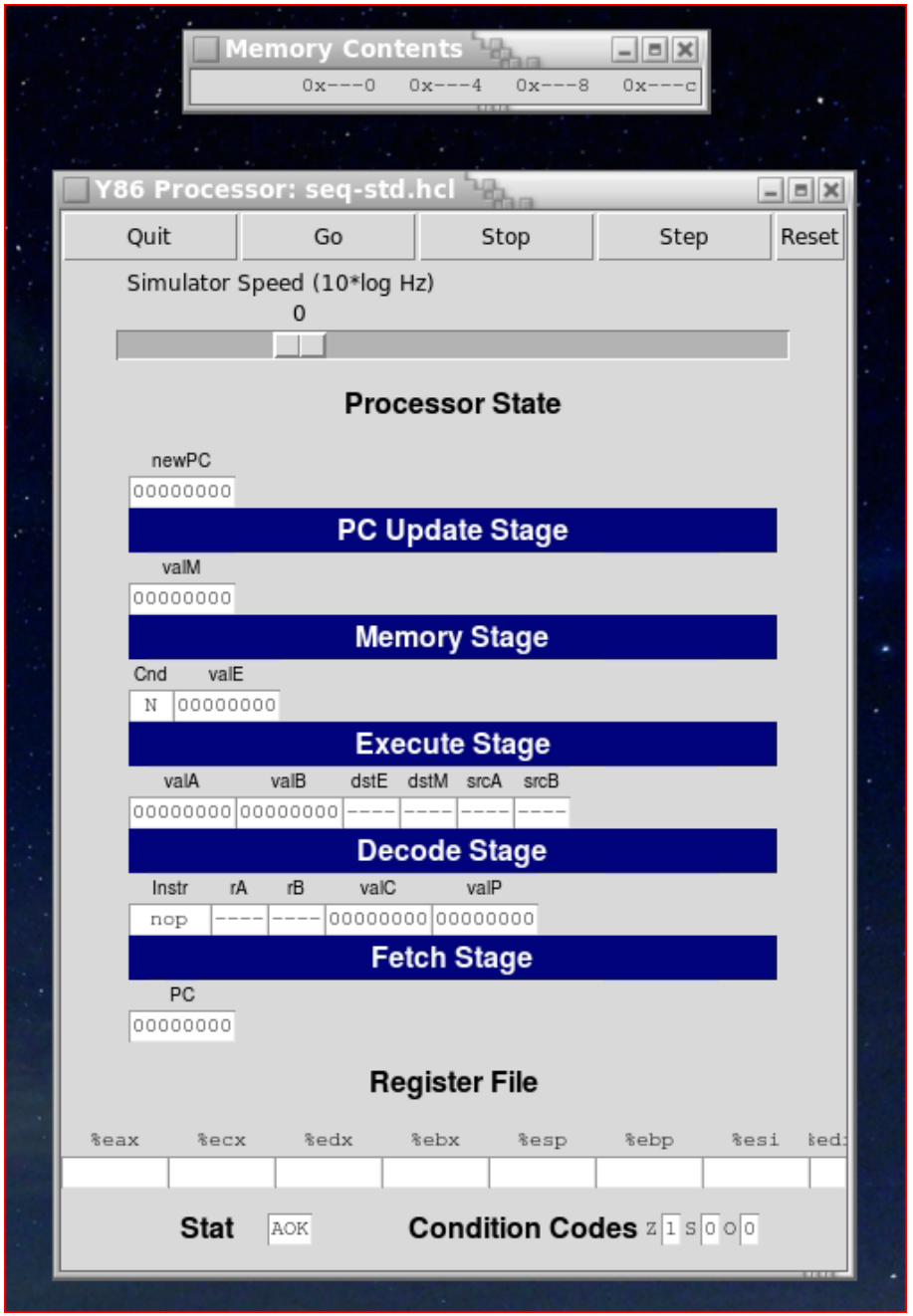
```
% yas y86prog1.ys
% yis y86prog1.yo
Stopped in 6 steps at PC = 0x1a.
Status 'HLT'
CC Z=1 S=0 O=0
Changes to registers:
%eax:    0x00000000          0x0000001c
%ecx:    0x00000000          0x0000006f
%edx:    0x00000000          0x00000037
%ebx:    0x00000000          0x00000037

Changes to memory:
0x0020:  0x00000084          0x00000037
```

Y86 Simulator program code



Y86 Simulator

- **displays**
  - contents of memory
  - processor state
  - fetch-execute loop
  - register file
  - status
  - condition codes



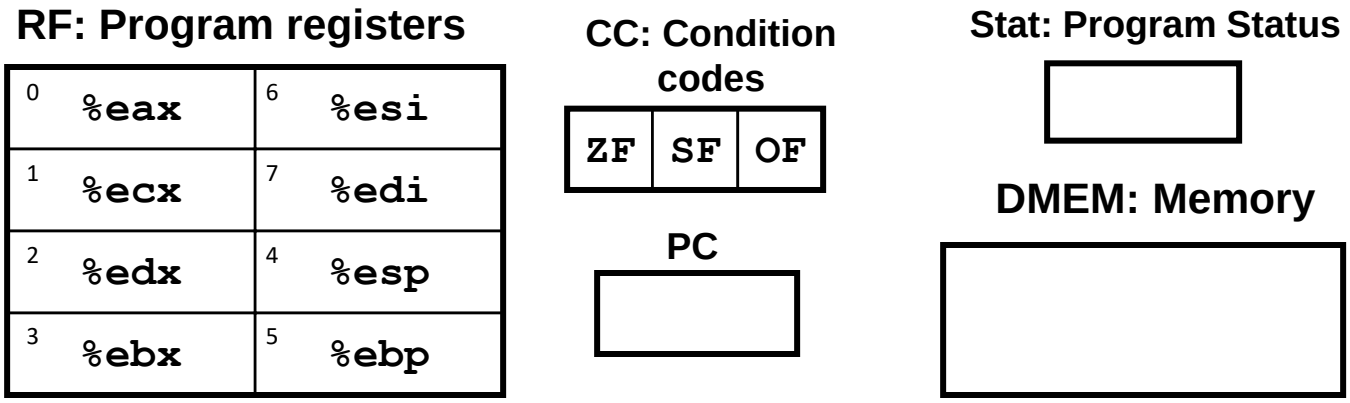
Y86 Notes

- **Y86 is an assembly language instruction set**
  - simpler than but similar to IA32
  - but not as compact (as we will see)

- **Y86 features**
  - 8 32-bit registers with the same names as the IA32 32-bit registers
  - 3 condition codes: ZF, SF, OF
    - no carry flag - interpret integers as signed
  - a program counter (PC)
    - holds the address of the instruction currently being executed
  - a program status byte: AOK, HLT, ADR, INS
    - state of program execution
  - memory: up to 4 GB to hold program and data

Y86 Notes

- **Y86 features**
  - 8 32-bit registers with the same names as the IA32 32-bit registers
    - different names from those in text, such as **`%rax`**, which are 64-bit
    - F indicates no register
  - 3 condition codes: ZF, SF, OF
  - a program counter (PC)
  - a program status byte: AOK, HLT, ADR, INS
  - memory: up to 4 GB to hold program and data



Outline

- Introduction to assembly programing
- Introduction to Y86
- Y86 instructions, encoding and execution

Learning Y86

- **Assembler Directives**
- **Status conditions and Exceptions**
- **Instructions**
  - Operations
  - Branches
  - Moves
- **Addressing Modes**
- **Stack Operations**
- **Subroutine Call/Return**
- **How to encode and execute each instruction**

Y86 Assembler Directives

Directive	Effect
`.pos number`	Subsequent lines of code start at address **number**
`.align number`	Align the next line to a **number**-byte boundary
`.long number`	Put **number** at the current address in memory

- These can be used to set up memory in various places in the address space
- `.pos` can put sections of code in different places in memory
- `.align` should be used before setting up a static variable
- `.long` can be used to initialize a static variable

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 Exceptions

- **What happens when an invalid assembly instruction is found?**
  - how would this happen?
  - generates an exception

- **In Y86 an exception halts the machine**
  - stops executing
  - on a real system, this would be handled by the OS and only the current process would be terminated

Y86 Exceptions

- **What are some possible causes of exceptions?**
  - invalid operation
  - divide by 0
  - sqrt of negative number
  - memory access error (address too large)
  - hardware error

- **Y86 handles 3 types of exceptions:**
  - HLT instruction executed
  - invalid address encountered
  - invalid instruction encountered

  - in each case, the status is set

Y86 Instructions

- **Each accesses and modifies some part(s) of the program state**

- **largely a subset of the IA32 instruction set**
  - includes only 4-byte integer operations → “word”
  - has fewer addressing modes
  - smaller set of operations

Y86 Instructions

- **format**
  - 1–6 bytes of information read from memory
    - Can determine the type of instruction from first byte
    - Can determine instruction length from first byte
    - Not as many instruction types
    - Simpler encoding than with IA32

- **registers**
  - **rA** or **rB** represent one of the registers (0-7)
  - 0xF denotes no register (when needed)
  - no partial register options (must be a byte)

Move Operation

Instruction	Effect	Description
`irmovl V,R`	Reg[R] ← V	Immediate-to-register move
`rrmovl rA,rB`	Reg[rB] ← Reg[rA]	Register-to-register move
`rmmovl rA,D(rB)`	Mem[Reg[rB]+D] ← Reg[rA]	Register-to-memory move
`mrmovl D(rA),rB`	Reg[rB] ← Mem[Reg[rA]+D]	Memory-to-register move

- **`irmovl`** is used to place known numeric values (labels or numeric literals) into registers
- **`rrmovl`** copies a value between registers
- **`rmmovl`** stores a word in memory
- **`mrmovl`** loads a word from memory
- **`rmmovl`** and **`mrmovl`** are the only instructions that access memory - Y86 is a load/store architecture

Move Operation

- **different opcodes for 4 types of moves**
  - register to register (opcode = 2)
    - notice conditional move has opcode 2 as well
  - immediate to register (opcode = 3)
  - register to memory (opcode = 4)
  - memory to register (opcode = 5)

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

Move Operation

- the only memory addressing mode is base register + displacement
- memory operations always move 4 bytes (no byte or word memory operations, i.e., no 8/16-bit move)
- source or destination of memory move must be a register

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

CORRECTION = F

Supported Arithmetic Operations

- **OP1 (opcode = 6)**
  - only takes registers as operands
  - only work on 32 bits
  - note: no “or” and “not” ops
  - only instructions to set CC
    - starting point ZF=1, SF=0, OF=0

- **arithmetic instructions**
  - addl rA, rB R[rB] ← R[rB] + R[rA]
  - subl rA, rB R[rB] ← R[rB] – R[rA]
  - andl rA, rB R[rB] ← R[rB] & R[rA]
  - xorl rA, rB R[rB] ← R[rB] ^ R[rA]

```
# y86cc.ys
.pos 0x0
irmovl   $1, %eax
irmovl   $0, %ebx
irmovl   $1, %ecx
addl %eax, %eax
andl %ebx, %ebx
subl %eax, %ecx
irmovl  $0x7fffffff, %edx
addl %edx, %edx
halt
```

Jump Instructions

- **jump instructions (opcode = 7)**
  - fn = 0 for unconditional jump
  - fn =1-6 for <= < = != >= >
  - 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

Jump Instruction Types

- **Unconditional jumps**
  - jmp Dest PC ← Dest
- **Conditional jumps**
  - jle Dest PC ← Dest if last result ≤ 0
    - SF=1 or ZF=1
  - jl Dest PC ← Dest if last result < 0
    - SF=1 **and ZF=0**
  - je Dest PC ← Dest if last result = 0
    - ZF=1
  - jne Dest PC ← Dest if last result ≠ 0
    - ZF=0
  - jge Dest PC ← Dest if last result ≥ 0
    - SF=0 or ZF=1
  - jg Dest PC ← Dest if last result > 0
    - SF=0 **and ZF=0**

What about checking OF?

If the last result is not what is specified, then the jump is not taken; and the next sequential instruction is executed, i.e., PC = PC + jump instruction size

Y86 Example Program with Loop

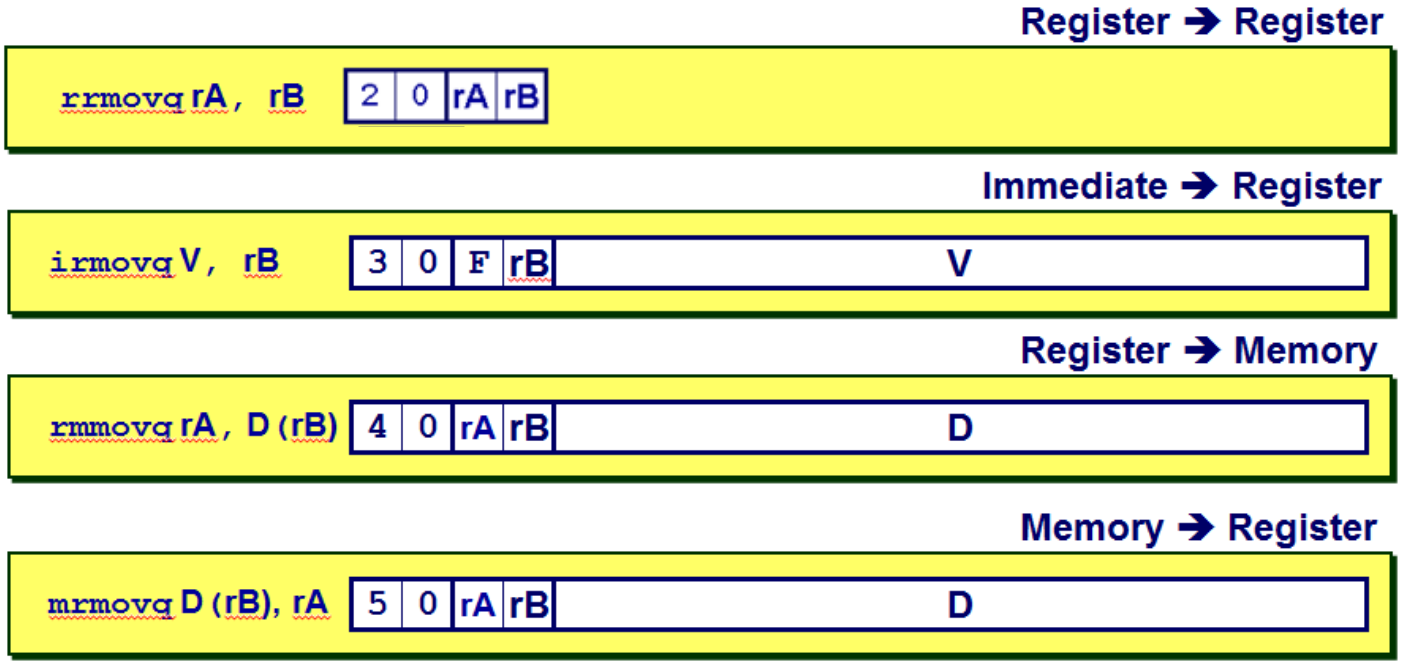
```
# y86loop.ys
.pos 0x0
        irmovl $0,%eax          # sum = 0
        irmovl $1,%ecx          # num = 1
Loop:   addl %ecx,%eax          # sum += num
        irmovl $1,%edx          # tmp = 1
        addl %edx,%ecx          # num++
        irmovl $1000,%edx       # lim = 1000
        subl %ecx,%edx          # if lim - num >= 0
        jge Loop                # loop again
        halt
```

Which instructions set the CC bits?

What does this code do?

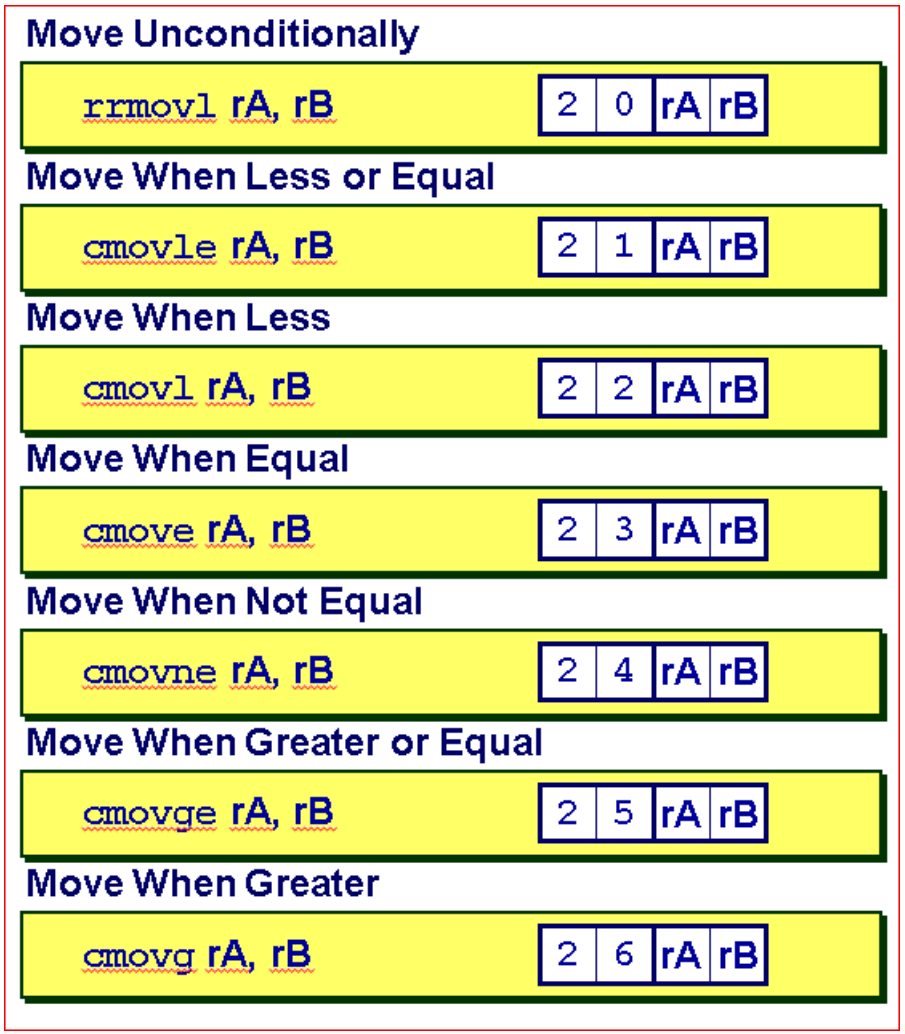
What are the flags set to for each instruction?

Move



- simple move commands
- note register bytes and bytes to store immediate or displacement values

Conditional Move



- 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

Conditional Move Examples

```
# y86ccmov.ys
.pos 0x0
irmovl   $1, %eax
cmove  %eax,%ecx
irmovl    0, %ebx
addl %eax, %eax
cmovg %eax, %ebx
andl %ebx, %ebx
subl %eax, %ecx
cmovg %ecx, %edx
irmovl  $0x7fffffff, %edx
addl %edx, %edx
halt
```

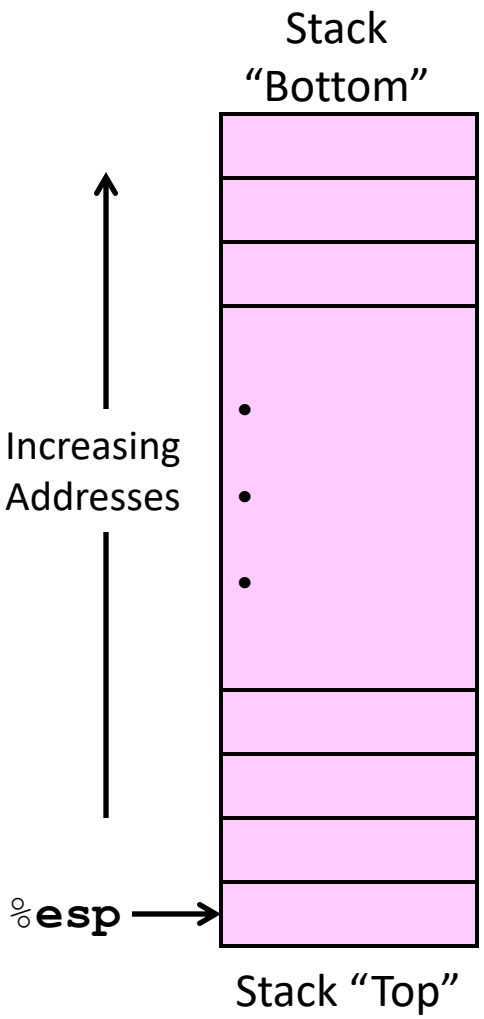
The cmovxx statement only moves the source register value to the destination register if the condition is true

If the condition is equal, that means the CC bits have the ZF set to 1, i.e., the previous result was equal to zero

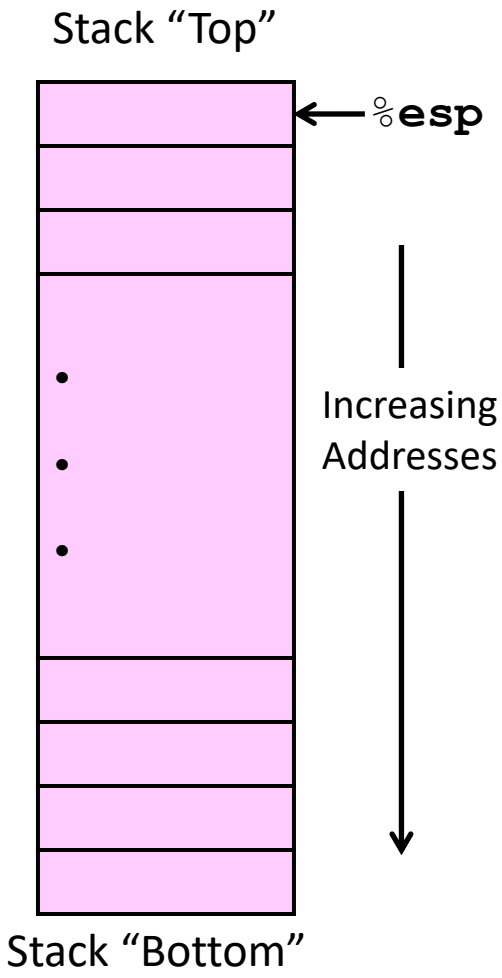
cmovg – checks if the previous result was greater than zero (i.e. SF=0) and if so, moves the source register value to the destination register

similar for other instructions

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

stack for Y86 works just the same as with IA32

`pushl rA` | A | 0 | rA | F |

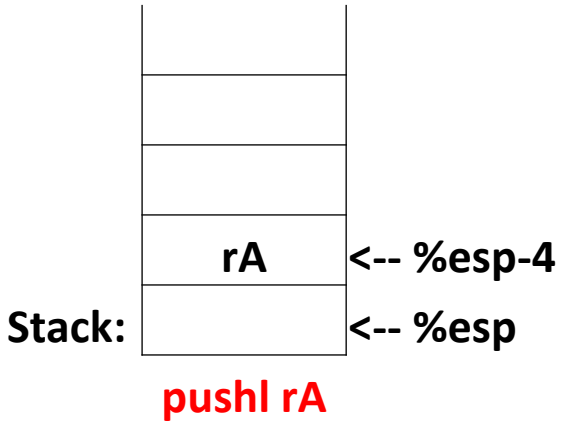
R[%esp]←R[%esp]-4
M[R[%esp]]←R[rA]

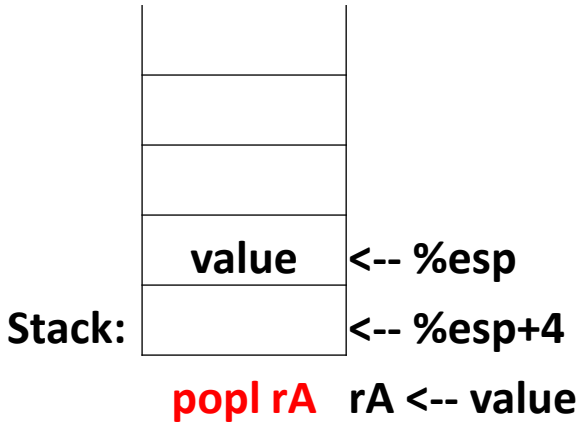
- Decrement `%esp` by 4
- Store word from rA to memory at `%esp`
- Like IA32

`popl rA` | B | 0 | rA | F |

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

R[rA]←M[R[%esp]]
R[%esp]←R[%esp]+4





Subroutine Call and Return

`call Dest`

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

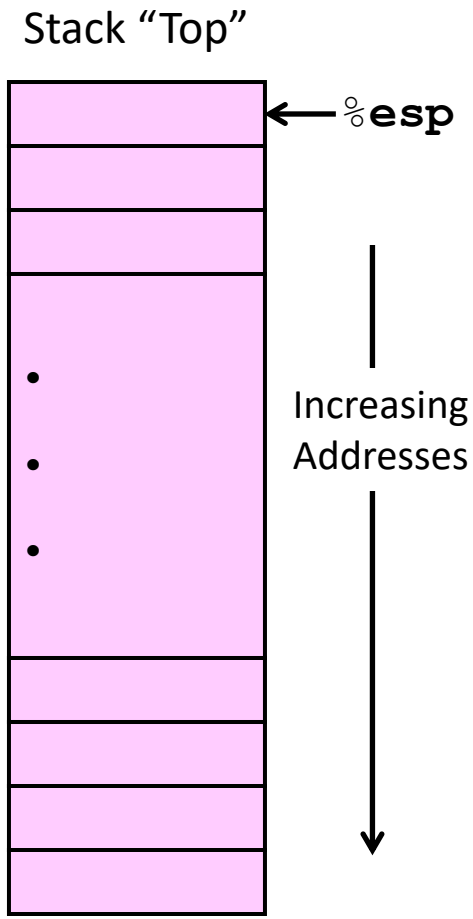
Note: call uses absolute addressing

`ret`

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

Procedure Call and Return

- **call pushes the PC value (point to next instruction) onto the top of the stack**
  - Dest R[%esp]←R[%esp]-4
    - make space on the stack
  - M[R[%esp]]←PC
    - move the value of the PC, which has been incremented to the next instruction, and store it in the memory location pointed to by reg %esp
  - PC←Dest
    - Move the destination address of the routine being called into the PC
- **ret**
  - PC←M[R[%esp]]
    - get the return address off the stack
  - R[%esp]←R[%esp]+4
    - adjust the stack pointer



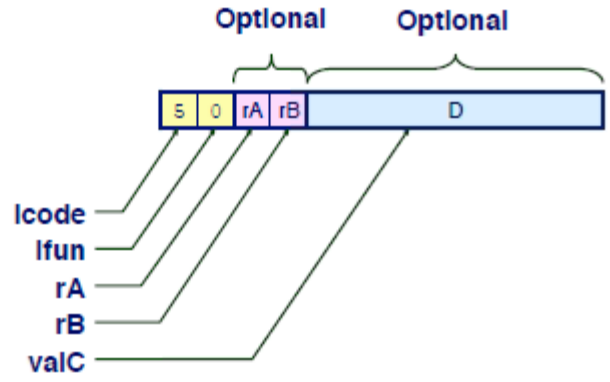
Miscellaneous Instructions

- 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

Y86 Instruction Set

- encoding of each instruction
- SEQ Hardware Structure



Instruction Format

- Instruction byte — icode:ifun
- Optional register byte — rA:rB
- Optional constant word — valC

Recall Memory is a Bunch of Bits!

How do we know if it is an instruction or not?

How do we know which instruction, which operands, etc.

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

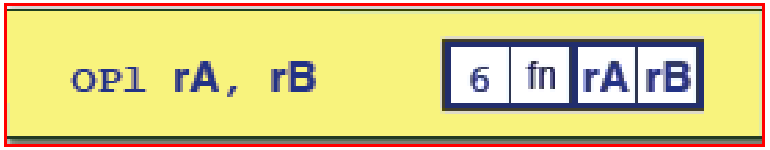
SEQ Hardware Structure Abstract and Stages

- State
  - Program counter reg (PC)
  - Condition code reg (CC)
  - Register File
  - Memories
    Data: read and write
    Instruction: read
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

- Fetch
  - Read instruction from instruction memory
  - If PC points to it, we view it as instruction
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter

Executing → Arithmetic/Logical Ops

- **Fetch**
  - Read 2 bytes
- **Decode**
  - Read operand registers
- **Execute**
  - Perform operation
  - Set condition codes
- **Memory**
  - Do nothing
- **Write back**
  - Update register
- **PC Update**
  - Increment PC by 2

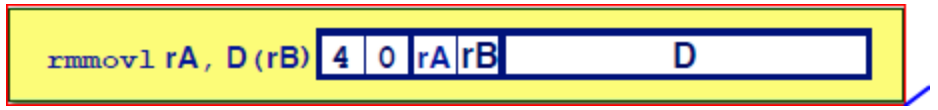


	OPl rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions; often called Register Transfer Language (RTL)

Executing → rmmovl

- **Fetch**
  - Read 6 bytes
- **Decode**
  - Read operand registers
- **Execute**
  - Compute effective address
- **Memory**
  - Write to memory
- **Write back**
  - Do nothing
- **PC Update**
  - Increment PC by 6

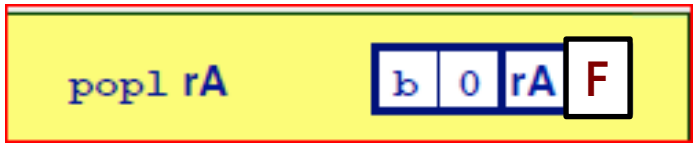


	rmmovl rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_4[PC+2]	Read displacement D
	valP ← PC+6	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_4[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- Use ALU for address computation

Executing → popl

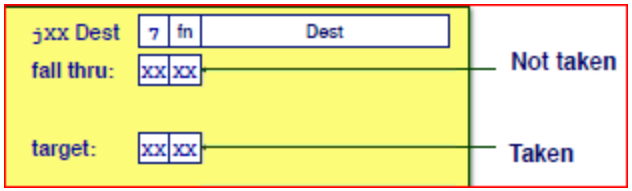
- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read from old stack pointer
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2



	popl rA	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[%esp] valB ← R [%esp]	Read stack pointer Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read from stack
Write back	R[%esp] ← valE R[rA] ← valM	Update stack pointer Write back result
PC update	PC ← valP	Update PC

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

Executing → Jumps

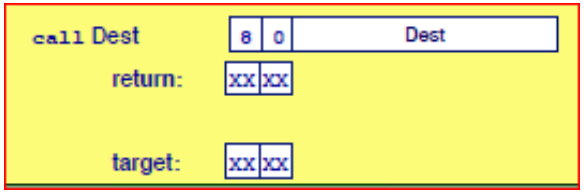


- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

	jXX Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	Read instruction byte Read destination address Fall through address
Decode		
Execute	Bch ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Bch ? valC : valP	Update PC

- **Compute both addresses**
- **Choose based on setting of condition codes and branch condition**

Executing → Call

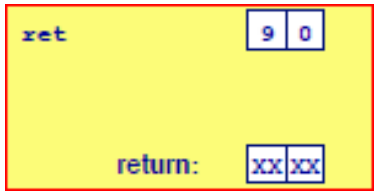


- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 4
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Compute return point
Decode	valB ← R[%esp]	Read stack pointer
Execute	valE ← valB + –4	Decrement stack pointer
Memory	M_4[valE] ← valP	Write return value on stack
Write back	R[%esp] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- **Use ALU to decrement stack pointer**
- **Store Incremented PC**

Executing → ret

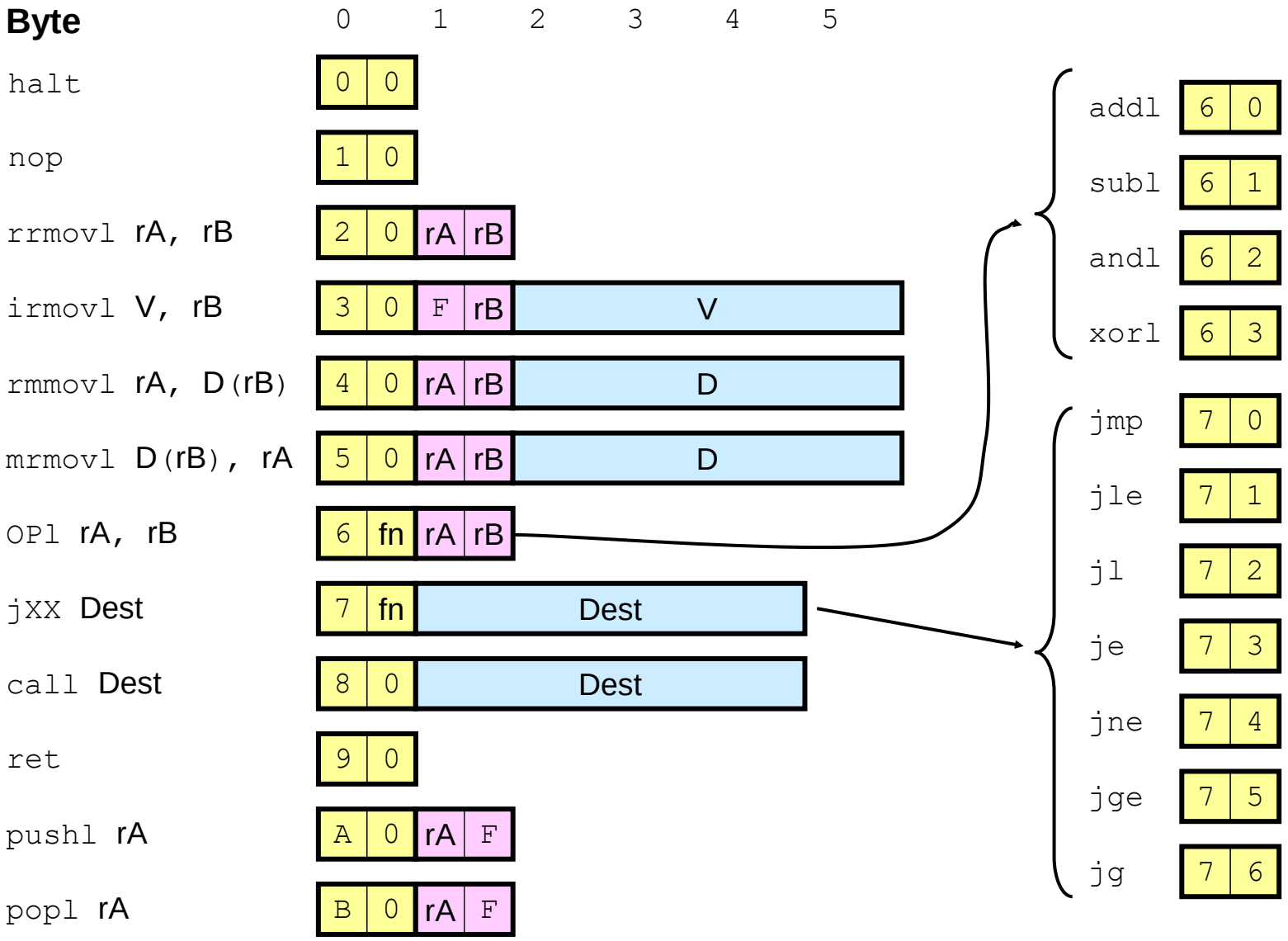


- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 4
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

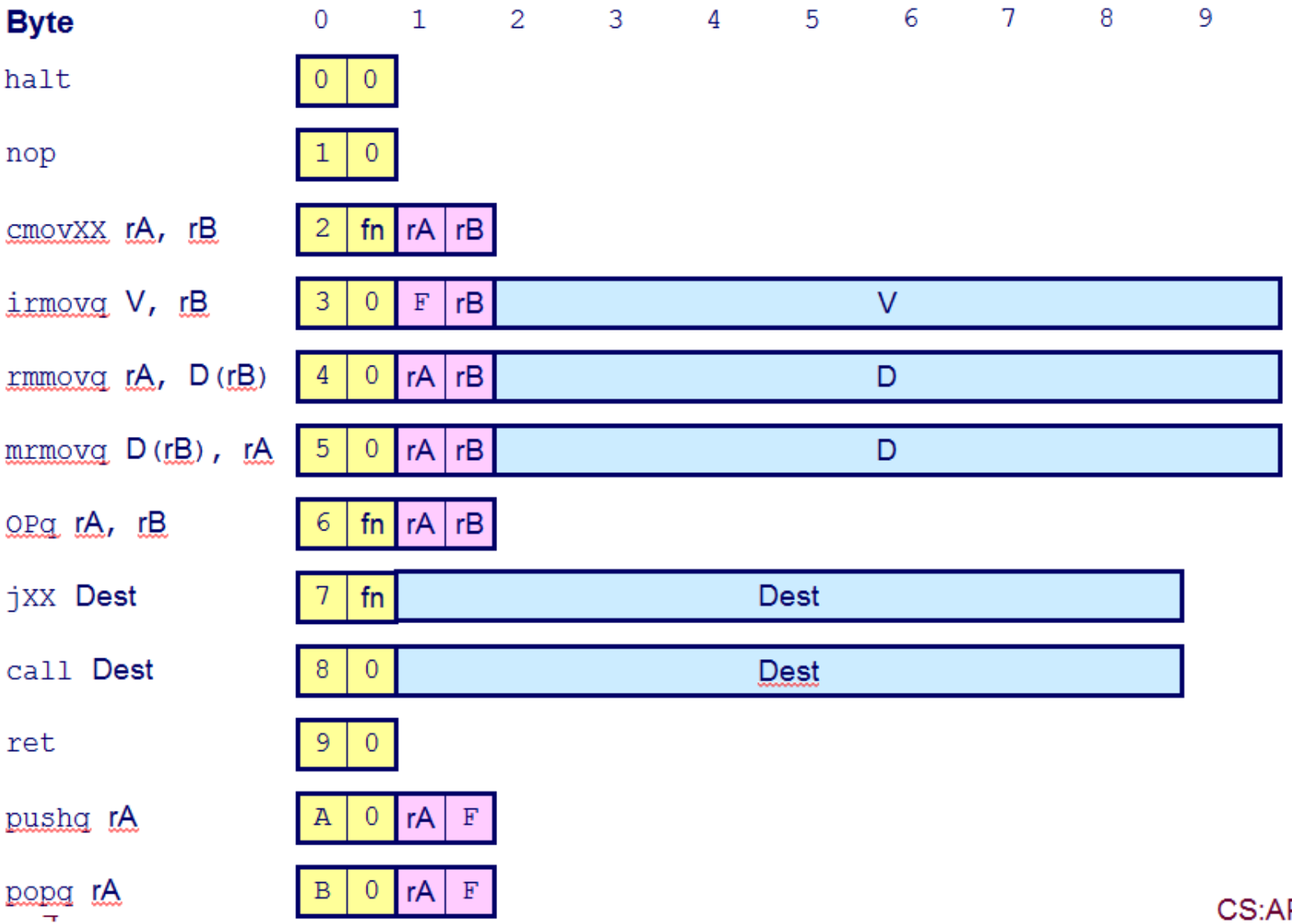
	ret	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[%esp] valB ← R[%esp]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read return address
Write back	R[%esp] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- **Use ALU to Increment stack pointer**
- **Read return address from memory**

Instruction Encoding (32-bit)

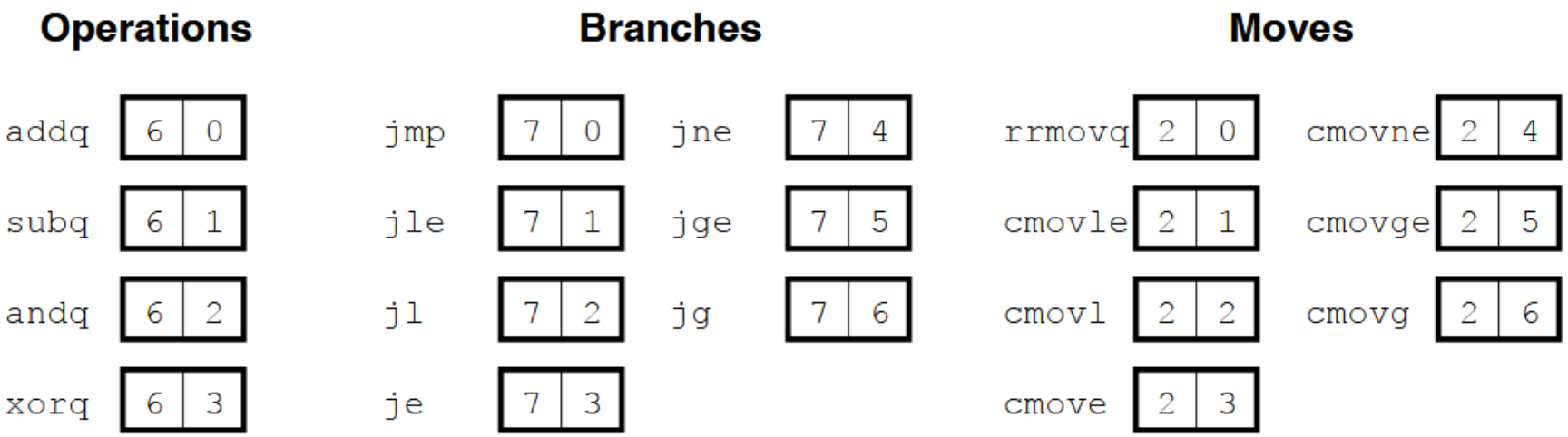


Instruction Encoding (64-bit)



CS:APP3e

Instruction Encoding



0	%eax	6	%esi
1	%ecx	7	%edi
2	%edx	4	%esp
3	%ebx	5	%ebp

Instruction Encoding Practice

Determine the byte encoding of the following Y86 instruction sequence given ".pos 0x100" specifies the starting address of the object code to be 0x100 (practice problem 4.1)

```
.pos 0x100 # start code at address 0x100
    irmovl    $15, %ebx          # load 15 into %ebx
    rrmovl    %ebx, %ecx         # copy 15 to %ecx
loop:
    rmmovl    %ecx, -3(%ebx)     # save %ecx at addr 15-3=12
    addl      %ebx, %ecx         # increment %ecx by 15
    jmp        loop              # goto loop
```

Instruction Encoding Practice

```
0x100:                | .pos 0x100 # start code at address 0x100
0x100: 30f30f000000 |     irmovl $15, %ebx         # load 15 into %ebx
0x106: 2031         |     rrmovl %ebx, %ecx        # copy 15 to %ecx
0x108:                | loop:
0x108: 4013fdffffff |     rmmovl %ecx, -3(%ebx)    # save %ecx at addr 15-3=12
0x10e: 6031         |     addl   %ebx, %ecx        # increment %ecx by 15
0x110: 7008010000   |     jmp    loop              # goto loop
```

Instruction Encoding Practice

- **0x100: 30f3fcffffff 406300080000 00**

0x100: 30f3fcffffff	irmovl $-4, %ebx
0x106: 406300080000	rmmovl %esi, 0x800(%ebx)
0x10c: 00	halt

Now try

- **0x200: a06f80080200000030f30a00000090**

0x200: a06f	push %esi
0x202: 8008020000	call 0x00000208
0x207: 00	halt
0x208: 30f3a0000000	irmovl $a0, %ebx
0x20e: 90	ret

- **0x400: 6113730004000000**

0x400: 6113	subl %ecx, %ebx
0x402: 7300040000	je 0x00040000
0x407: 00	halt

Summary

Important property of any instruction set

THE BYTE ENCODINGS MUST HAVE A UNIQUE INTERPRETATION

which

ENSURES THAT A PROCESSOR CAN EXECUTE

AN OBJECT-CODE PROGRAM WITHOUT ANY AMBIGUITY
ABOUT THE MEANING OF THE CODE

Conditional Statements

- simple if statement

  C code:

```
if (x == 2)
    x += 1;
// program continues
```

  Y86 code:

```
    irmovl $2,   %ecx  # get ready to compare x to 2
    rrmovl %eax, %edx  # move x to temp register
    subl   %ecx, %edx  # set condition codes: tmpx -= 2
    jne    progcont    # jump over if block when x != 2
    irmovl $1,   %edx  # get ready to add 1 to x
    addl   %edx, %eax  # add 1 to x
                       # fall into rest of program
progcont:
```

Conditional Statements

- simple if-else statement

C code:

```
if (x > 5)
    x += 1;
else
    x -= 2;
// program continues
```

Y86 code:

```
     irmovl $5,   %ecx  # get ready to compare x to 5
     rrmovl %eax, %edx  # move x to temp register
     subl   %ecx, %edx  # set condition codes: tmpx -= 5
     jle    else        # jump over if block when x <= 5
     irmovl $1,   %edx  # get ready to add 1 to x
     addl   %edx, %eax  # add 1 to x
     jmp    progcont    # jump over else
else:
     irmovl $2,   %edx  # get ready to add 1 to x
     subl   %edx, %eax  # subtract 2 from x
                        # fall into rest of program
progcont:
```

Conditional Statements

- always test for the opposite of the conditional statement
  - jump over the if block if opposite is true
  - otherwise, fall into the if block
  - keeps assembly instructions in same order as C statements

- since code is in same order, easier to map if statements to assembly
  - conditional jump => if statement
  - unconditional jump => jump over block to avoid executing it