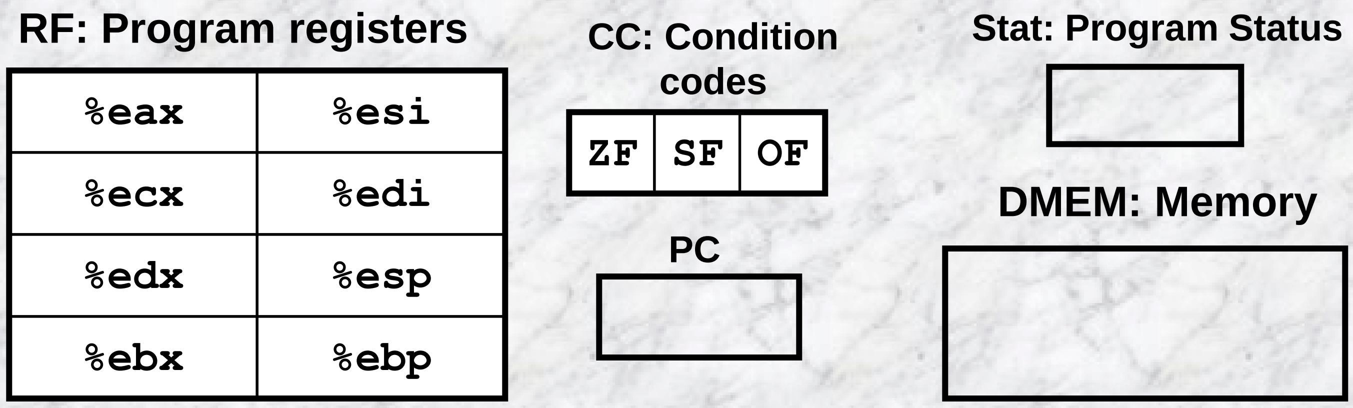
Y86 programmer-visible state

- Y86 is an assembly language instruction set simpler than but similar to IA32; but not as compact (as we will see)
- 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 - 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 (4096 = 2^12)



Looking ahead and comparing

- Y86 is:
  - Little endian
  - Load/store
    - Can only access memory on read/write
    - On move statements in Y86
  - Combination of CISC and RISC
  - Word = 4 bytes
- IA32 is:
  - Little endian
  - NOT load/store
  - CISC
  - Byte (1 byte), word (2 bytes), long (4 bytes)

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
- 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

- 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)
- 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.

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

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

Move operation (cont)

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

Supported OPs and Jump

- OP1 (opcode = 6)
  - Only take registers as operands
  - Only work on 32 bits
  - Note: no “or” and “not” ops
  - Only instructions to set CC
- 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

fn	operation
0	addl
1	subl
2	andl
3	xorl

Conditional move

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

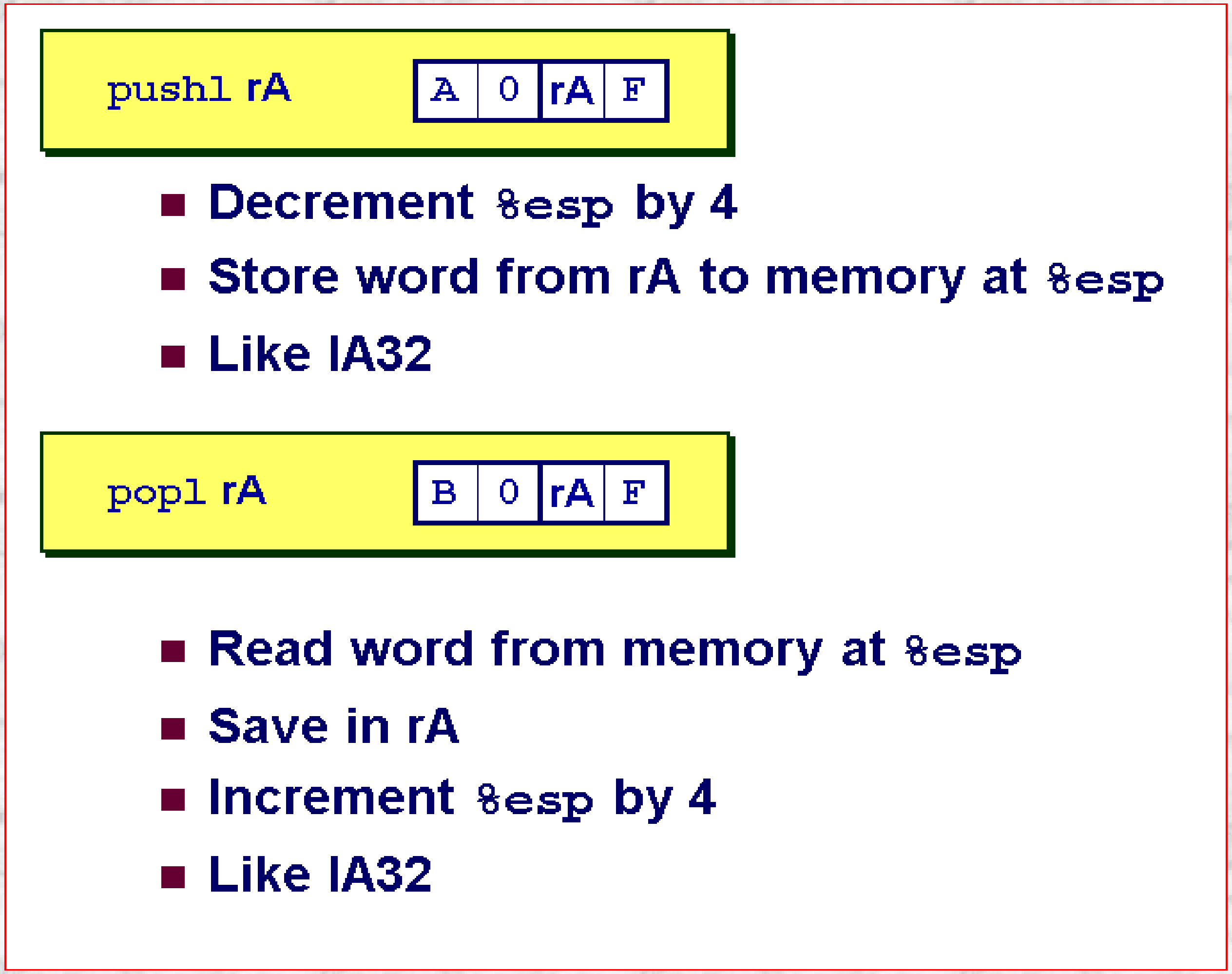
cmovge rA, rB	2	5	rA	rB

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

Stack Operations



`pushl rA`

A	0	rA	F

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

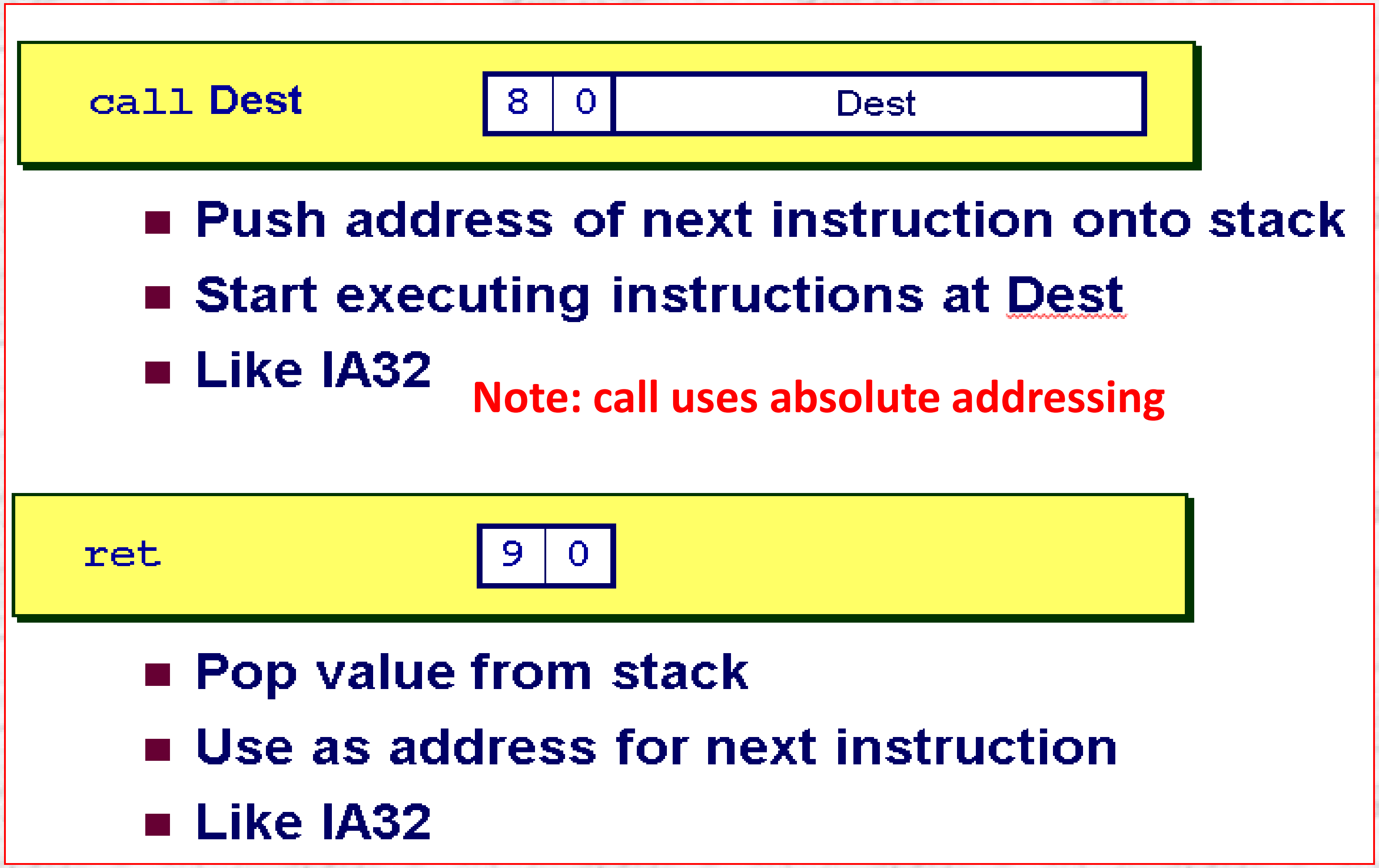
`popl rA`

B	0	rA	F

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

- Stack for Y86 works just the same as with IA32

Subroutine call and return



`call` **Dest**

8	0	Dest

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

Note: call uses absolute addressing

`ret`

9	0

- **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

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 (cont)

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

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

Now you try:

- 0x200: a06f 80080200000030f30a00000090
- 0x400: 6113730004000000

Summary

Important property of any instruction set

THE BYTE ENCODINGS MUST HAVE A UNIQUE INTERPRETATION

which

ENSURES THAT A PRCESSOR CAN EXECUTE AN OBJECT-CODE PROGRAM WITHOUT ANY AMBIGUITY ABOUT THE MEANING OF THE CODE