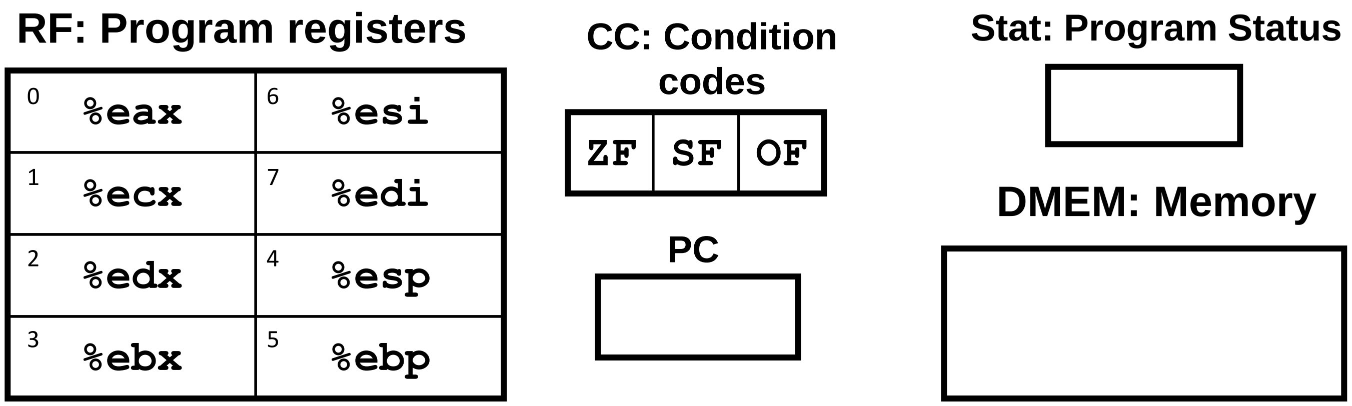
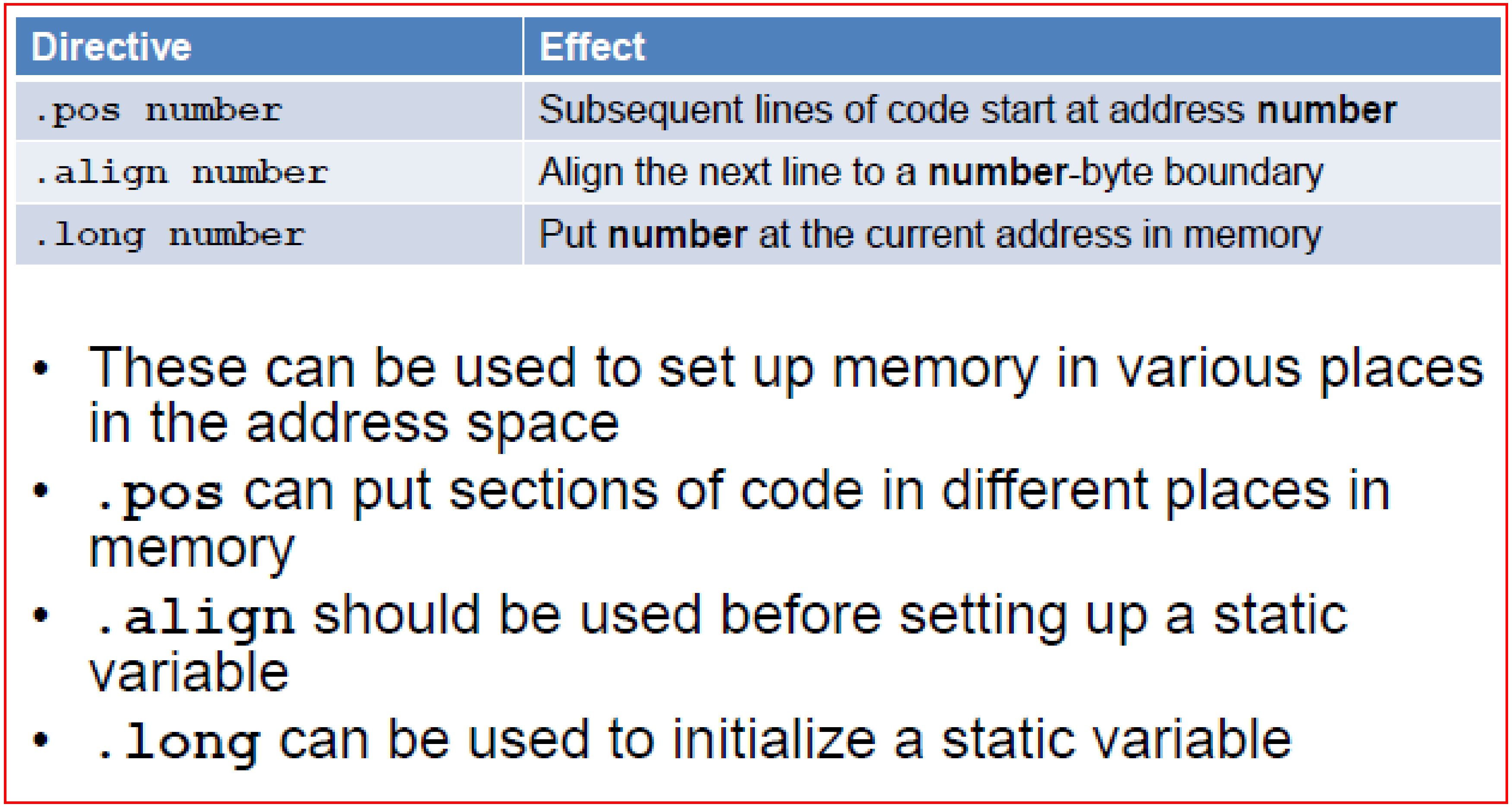
Y86 Notes

- **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



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

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 ★

Jump Instruction Types

- **Unconditional jumps**
  - jmp Dest PC ← Dest

What about checking OF?

- **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**

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

Stack Operations

stack for Y86 works just the same as with IA32

`pushl rA`	A	0	rA	F

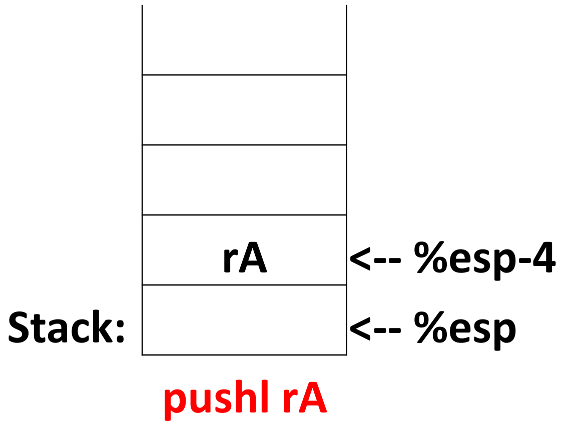
R[%esp]←R[%esp]-4
M[R[%esp]]←R[rA]

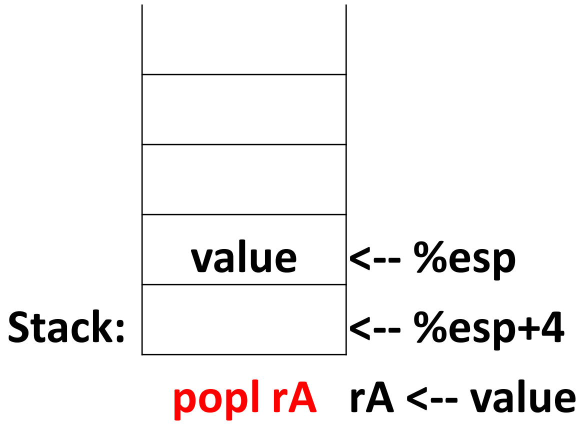
- Decrement `%esp` by 4
- Store word from rA to memory at `%esp`
- Like IA32

`popl rA`	B	0	rA	F

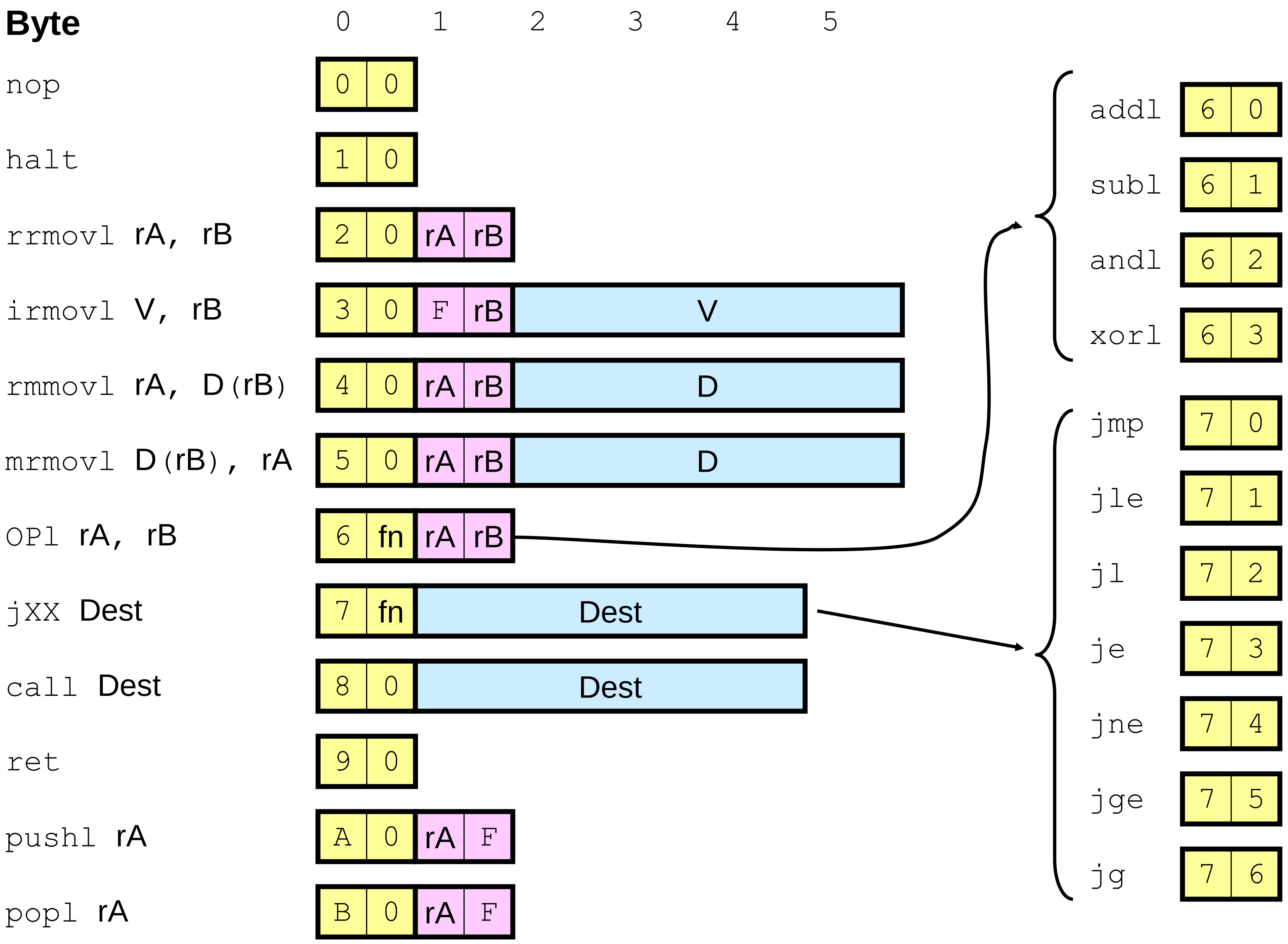
- Read word from memory at `%esp`
- Save in rA
- Increment `%esp` by 4
- Like IA32

R[rA]←M[R[%esp]]
R[%esp]←R[%esp]+4





Instruction Encoding (32-bit)



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	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

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

Instruction Encoding (64-bit)

Byte	0		1		2	3	4	5	6	7	8	9
`halt`	0	0										
`nop`	1	0										
`cmovXX` rA, rB	2	fn	rA	rB								
`irmovq` V, rB	3	0	F	rB	V							
`rmmovq` rA, D(rB)	4	0	rA	rB	D							
`mrmovq` D(rB), rA	5	0	rA	rB	D							
`OPq` rA, rB	6	fn	rA	rB								
`jXX` Dest	7	fn	Dest									
`call` Dest	8	0	Dest									
`ret`	9	0										
`pushq` rA	A	0	rA	F								
`popq` rA	B	0	rA	F								

CS:APP3e

Instruction Encoding

Operations

`addq`	6	0
`subq`	6	1
`andq`	6	2
`xorq`	6	3

Branches

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

Moves

`rrmovq`	2	0	`cmovne`	2	4
`cmovle`	2	1	`cmovge`	2	5
`cmovl`	2	2	`cmovg`	2	6
`cmove`	2	3			

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