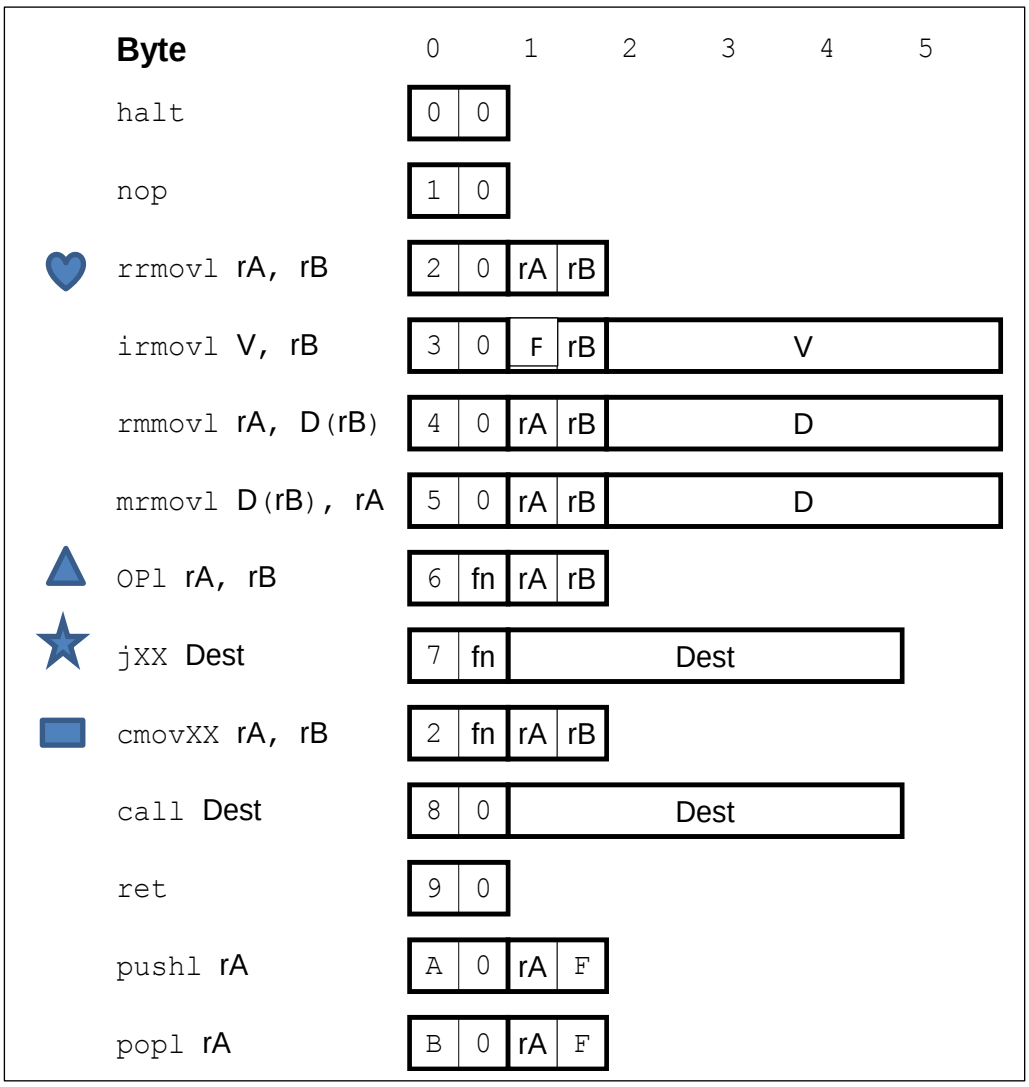
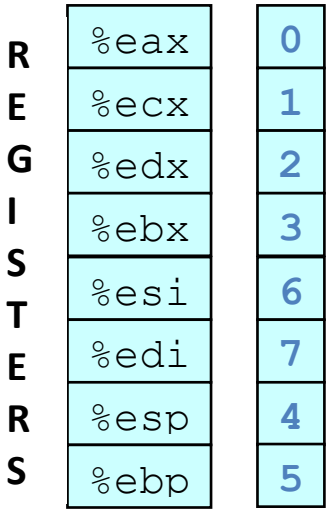
Y86 Instruction Set





Operations

addl	6 0
subl	6 1
andl	6 2
xorl	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

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

COMPLETE PROGRAM FILE WRITTEN IN Y86 ASSEMBLY CODE

The program contains both data and instructions. Directives indicate where to place code or data and how to align it. The program also specifies issues such as stack placement, data initialization, program initialization and program termination.

```
1  # Execution begins at address 0
2          .pos 0
3  init:   irmovl Stack, %esp      # Set up stack pointer
4          irmovl Stack, %ebp      # Set up base pointer
5          call Main               # Execute main program
6          halt                    # Terminate program
7
8  # Array of 4 elements
9          .align 4
10 array:  .long 0xd
11         .long 0xc0
12         .long 0xb00
13         .long 0xa000
14
15 Main:   pushl %ebp
16         rrmovl %esp,%ebp
17         irmovl $4,%eax
18         pushl %eax              # Push 4
19         irmovl array,%edx
20         pushl %edx              # Push array
21         call Sum                # Sum(array, 4)
22         rrmovl %ebp,%esp
23         popl %ebp
24         ret
25
26         # int Sum(int *Start, int Count)
27 Sum:    pushl %ebp
28         rrmovl %esp,%ebp
29         mrmovl 8(%ebp),%ecx     # ecx = Start
30         mrmovl 12(%ebp),%edx    # edx = Count
31         xorl %eax,%eax          # sum = 0
32         andl   %edx,%edx        # Set condition codes
33         je     End
34 Loop:   mrmovl (%ecx),%esi      # get *Start
35         addl %esi,%eax          # add to sum
36         irmovl $4,%ebx          #
37         addl %ebx,%ecx          # Start++
38         irmovl $-1,%ebx         #
39         addl %ebx,%edx          # Count--
40         jne    Loop             # Stop when 0
41 End:    rrmovl %ebp,%esp
42         popl %ebp
43         ret
44
45 # The stack starts here and grows to lower addresses
46         .pos 0x100
47 Stack:
```

Start address for all Y86 programs
init → we are the OS now since Y86 doesn't have one!

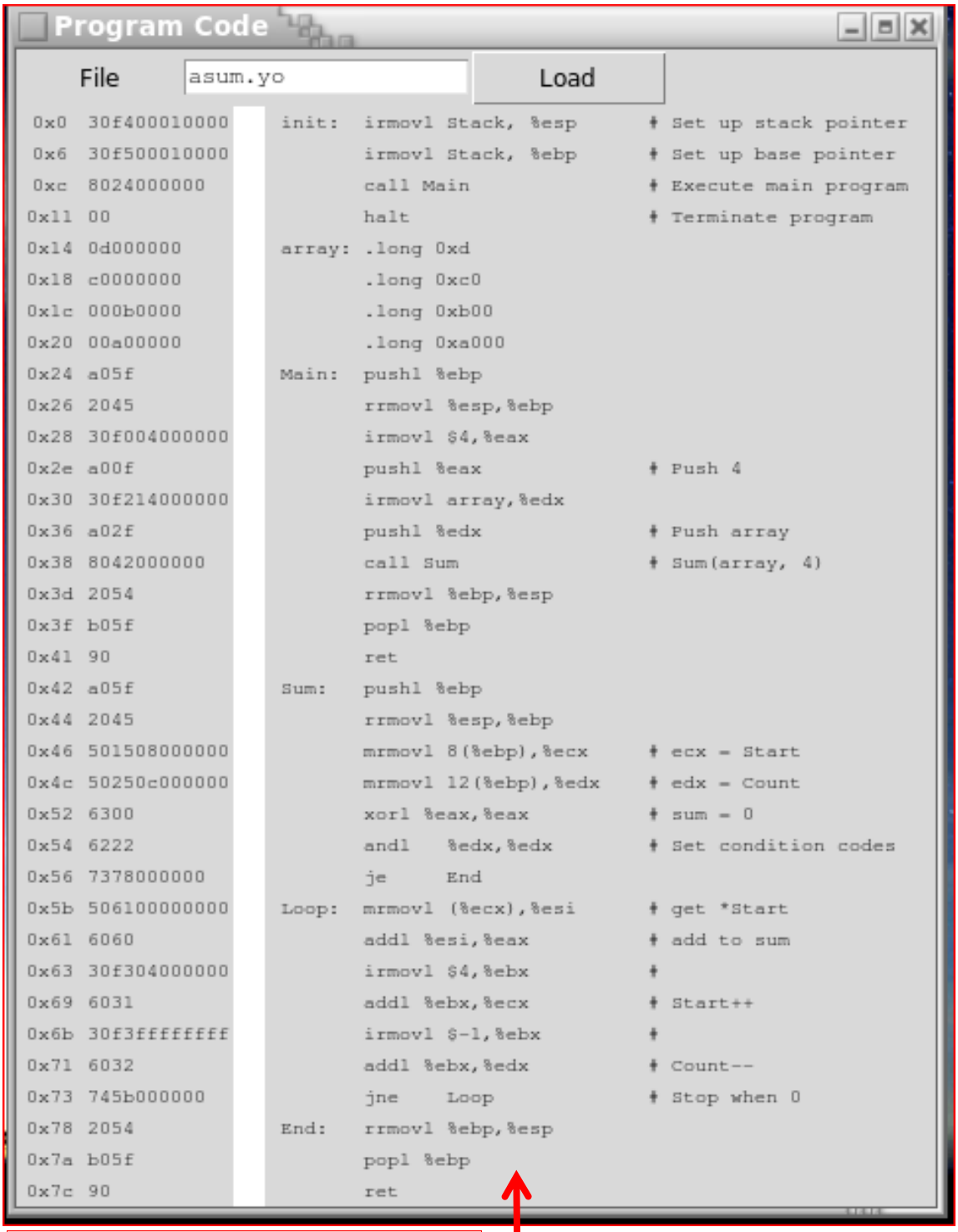
Declare an array of 4 words starting on a 4 byte boundary

Save "init" base pointer
Set up new frame for main routine
Add parameters to the stack to a call to the "Sum" function

Sum up the values in the array

Need to make sure that the stack does not grow so large that it overwrites the code or other program data

Figure 4.7: **Sample program written in Y86 assembly code.** The `Sum` function is called to compute the sum of a four-element array.



PROGRAM WALKTHRU

%esp=0x100
%ebp=0x100
%esp = 0xfc = 0x11

%esp=0xf8=0x100
%ebp=0xf8
%eax=0x4
%esp=0xf4=4
%edx=0x14
%esp=0xf0=0x14
%esp=0xec=0x3d

%esp=0xe8=0xf8
%ebp=0xe8
%ecx=0x14
%edx=4
%eax=0
If %edx=0
Jump to End
%esi=0x14 value=0xd
%eax=0xd+0=0xd
%ebx=4
%ecx=0x14+4=0x18
%ebx=0xffffffff
%edx=-1+4=3
Zero flag not set,
Loop

ETC

```
Changes to registers:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%ebx:   0x00000000      0xffffffff
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000a000
```

```
Changes to memory:
0x00e8: 0x00000000      0x000000f8
0x00ec: 0x00000000      0x0000003d
0x00f0: 0x00000000      0x00000014
0x00f4: 0x00000000      0x00000004
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

Old %ebp (sum)
Ret addr (sum)
Start param
Count param
Old %ebp (init)
Ret addr (main)

0x0100: (top of stack)

To put the above in a different way:

address	STACK	description
	PROG et all typically up here	
	ETC	
0x0e8	**0xf8**	<-- **%esp** (push %ebp), %ebp
0x0ec	**0x3d**	<-- %esp (call **Sum**)
0x0f0	**0x14**	<-- %esp (push %edx)
0x0f4	**0x4**	<-- %esp (push %eax)
0x0f8	**0x100**	<-- %esp (push %ebp), **%ebp**
0x0fc	**0x11**	<-- **%esp** (call **Main**)
stack: 0x100		<-- %esp, **%ebp**
	<------ 4-bytes ------>	

Y86 REGISTERS	**VALUE**
%eax=	**0x4**, 0, 0xd, 0xcd, 0xbcd, 0xabcd
%ebx=	0x4, 0xffffffff, 0x4, 0xffffffff, 0x4, 0xffffffff, 0x4, 0xffffffff
%ecx=	0x14, 0x18, 0x1c, 0x20, 0x24
%edx=	**0x14**, 0x4, 0x3, 0x2, 0x1, 0x0
%esi=	0xd, 0xc0, 0xb00, 0xa000
%edi=	
%esp=	stack pointer
%ebp=	base pointer

QUESTIONS

Why use "xorl %eax,%eax" instead of "irmovl $0, %eax"? to set condition code bits

What are CC bits… for xorl and for irmovl? Z=1, S=0, O=0 for xorl

for irmovl… starting bits?

Why do "andl %edx,%edx"? to set CC bits for “je” instruction

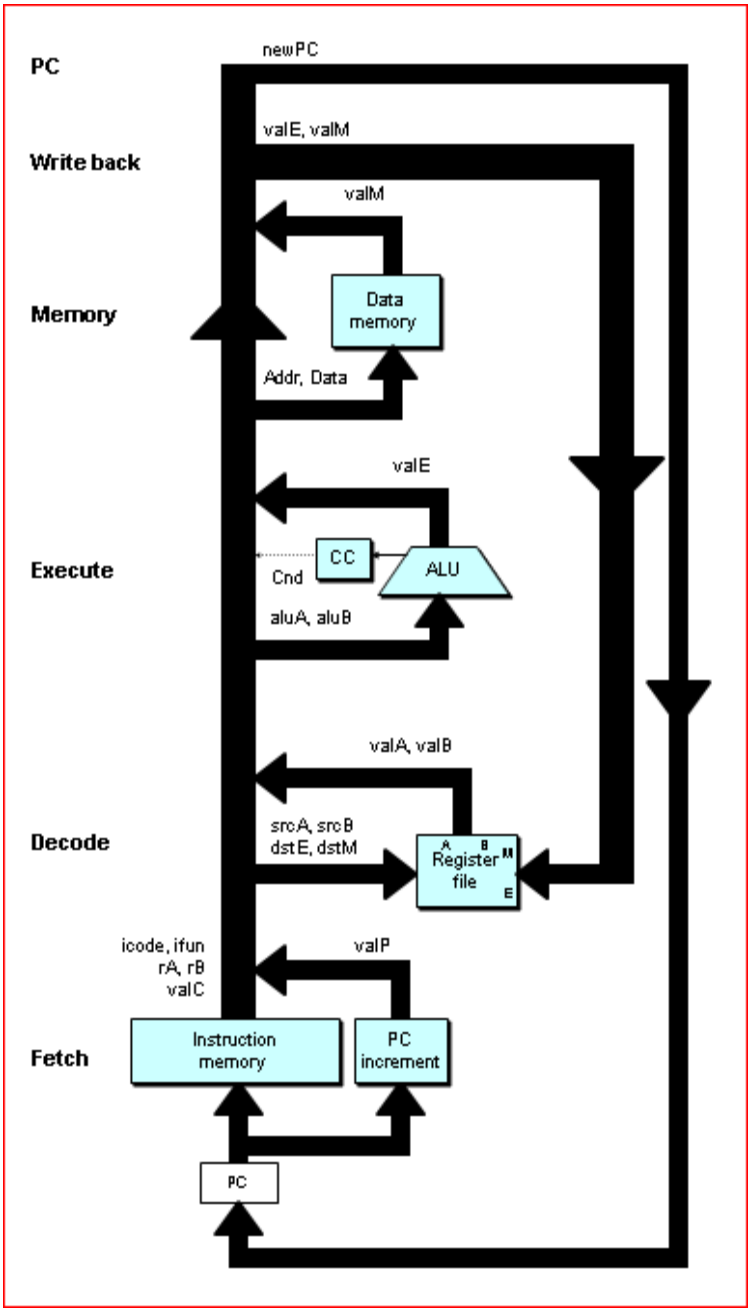
What are CC bits? Z=0, S=0, O=0

How many registers used just to sum a list of numbers? only %edi not used

What if stack bumps into where the program is stored? uh oh!

how make this happen? stack .pos < 0x7c + stack size

ORGANIZING PROCESSING INTO STAGES



PC Update: The **PC** is set to the address of the next instruction

Write back: Writes up to two results to the register file

Memory: Can write data to memory, or may read data from memory. The value read is referred to as **valM**

Execute:The ALU either:

1. Performs the operations specified (according to the value of ifun)
2. Computes the effective address of a memory reference
3. Increments/decrements the stack pointer

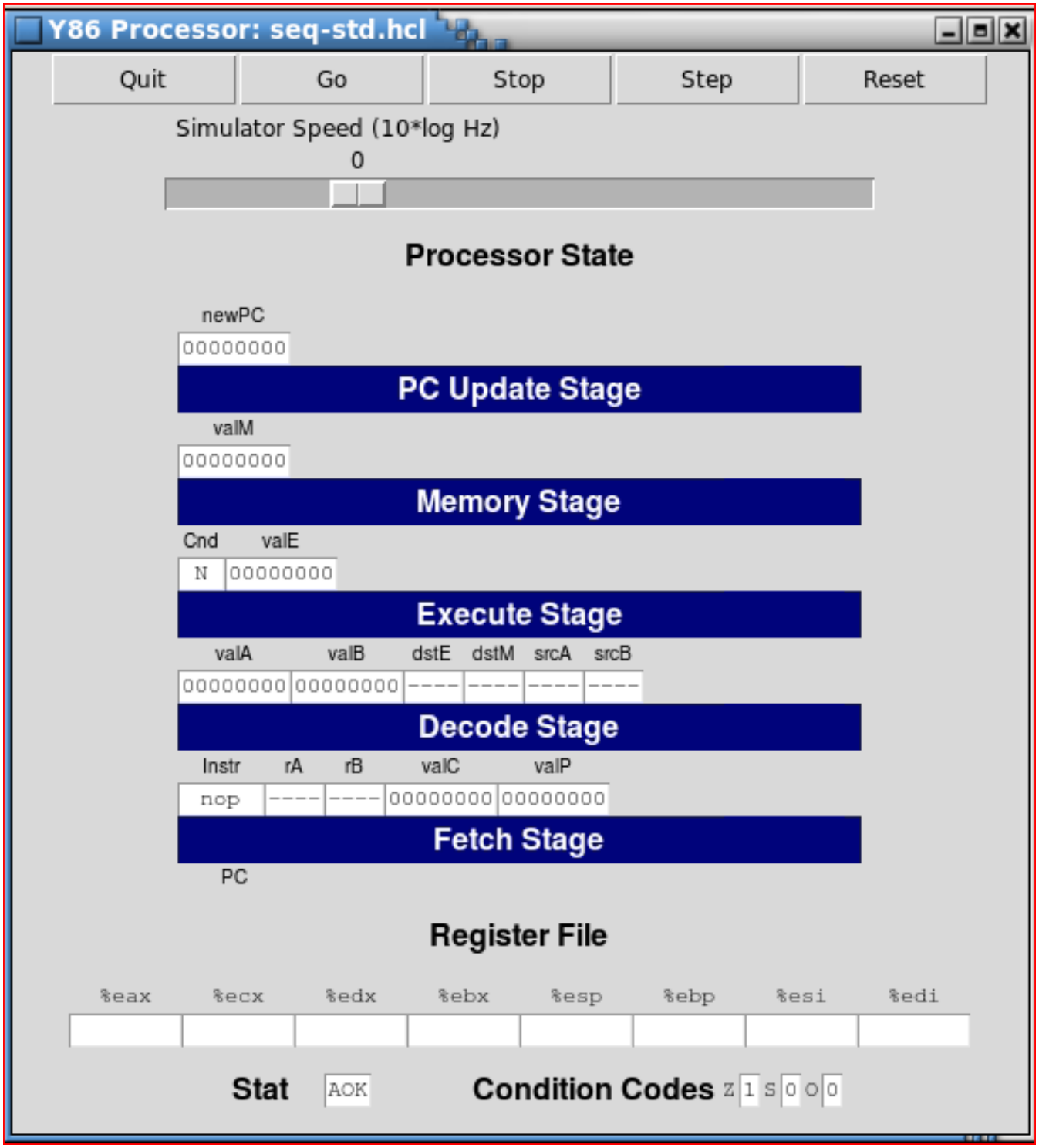
The resulting value is referred to as **valE**
The conditions codes are possibly set
The jump instruction tests the condition codes and branch condition (given by **ifun**) to see whether or not to take the branch

Decode: Reads up to two operands from the register file, giving values **valA** and/or **valB**.
Note: for some instructions it reads register %esp

Fetch:

- Reads the bytes of an instruction from memory using the **PC** as the memory address.
- **icode** = instruction code (high nibble of first byte)
- **ifun** = instruction function (low nibble of first byte)
- assigns **rA** and **rB** if applicable i.e. if one or both given as register operands
- **valC** = 4-byte constant word (if applicable)
- **valP** = computed to be the address of the instruction following the current one in sequential order i.e. value of PC + length of the fetched instruction

The processor loops indefinitely, performing these stages. In our simplified implementations, the processor will stop when any exception occurs: it executes a “halt” or invalid instruction, or it attempts to read or write an invalid address. In a more complete design, the processor would enter an exception-handling mode and begin executing special code determined by the type of exception.


Y86 Processor: seq-std.hcl
Quit
Go
Stop
Step
Reset
Simulator Speed (10*log Hz)
0
Processor State
newPC
00000000
PC Update Stage
valM
00000000
Memory Stage
Cnd
valE
N
00000000
Execute Stage
valA
valB
dstE
dstM
srcA
srcB
00000000
00000000

Decode Stage
Instr
rA
rB
valC
valP
nop

00000000
00000000
Fetch Stage
PC
Register File
%eax
%ecx
%edx
%ebx
%esp
%ebp
%esi
%edi
Stat
AOK
Condition Codes
Z 1 S 0 O 0

Stage	OPl rA, rB	rrmovl rA, rB	irmovl V, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1]	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1]	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_4[PC + 2]
	valP ← PC + 2	valP ← PC + 2	valP ← PC + 6
Decode	valA ← R[rA] valB ← R[rB]	valA ← R[rA]	
Execute	valE ← valB OP valA Set CC	valE ← 0 + valA	valE ← 0 + valC
Memory			
Write back	R[rB] ← valE	R[rB] ← valE	R[rB] ← valE
PC update	PC ← valP	PC ← valP	PC ← valP

Stage	Generic ret
Fetch	icode:ifun ← M_1[PC]
	valP ← PC + 1
Decode	valA ← R[%esp] valB ← R[%esp]
Execute	valE ← valB + 4
Memory	valM ← M_4[valA]
Write back	R[%esp] ← valE
PC update	PC ← valM

Stage	rmmovl rA, D(rB)	mrmovl D(rB), rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_4[PC + 2] valP ← PC + 6	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_4[PC + 2] valP ← PC + 6
Decode	valA ← R[rA] valB ← R[rB]	valB ← R[rB]
Execute	valE ← valB + valC	valE ← valB + valC
Memory	M_4[valE] ← valA	valM ← M_4[valE]
Write back		R[rA] ← valM
PC update	PC ← valP	PC ← valP

Stage	pushl rA	popl rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1]	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1]
	valP ← PC + 2	valP ← PC + 2
Decode	valA ← R[rA] valB ← R[%esp]	valA ← R[%esp] valB ← R[%esp]
Execute	valE ← valB + (−4)	valE ← valB + 4
Memory	M_4[valE] ← valA	valM ← M_4[valA]
Write back	R[%esp] ← valE	R[%esp] ← valE R[rA] ← valM
PC update	PC ← valP	PC ← valP

tage	jXX Dest	call Dest
'etch	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]
	valC ← M_4[PC + 1] valP ← PC + 5	valC ← M_4[PC + 1] valP ← PC + 5
)ecode		valB ← R[%esp]
:xecute	Cnd ← Cond(CC, ifun)	valE ← valB + (−4)
lemory		M_4[valE] ← valP
Vrite back		R[%esp] ← valE
C update	PC ← Cnd ? valC : valP	PC ← valC

Stage	cmovXX rA, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valP ← PC + 2
Decode	valA ← R[rA]
Execute	valE ← 0 + valA
Memory	
Write back	R[rB] ← valE
PC update	PC ← valP