12-y86-notes

Y86 Assembly Language

Agenda

- 0. Re-Orienting
- 1. Assembly Language
- 2. Basic Instructions
- 3. Arithmetic and Logic
- 4. Conditional Instructions
- 5. The Stack
- 6. Subroutines

Reading:

- 3.2.0, 3.2.1: general background on ISA
- 3.4.0, 3.4.1: addressing---but for the more general x86/IA32 case
- 3.4.2: discussing the stack!
- 4.1.4, 4.1.5: Y86 examples

---

0. Re-Orienting

The computer scientist.... all the way down to the laws of physics

the ISA

manipulating the registers and such with instructions

assembling the instructions into sequences of machine code

- by hand
- or with an assembler

---

1. Assembly Language

```
        .pos 0

        # immediate addressing!
                irmovl $0x11223344, %ecx

        # register addressing!
                rrmovl %ecx, %edx

        # indirect addressinng!
                irmovl target2, %ebx
                rmmovl %ecx, (%ebx)

        # absolute addressing!
                rmmovl %ecx, 0x50

        # base + displacement
                rmmovl %ecx, 4(%ebx)

        # goodbye!
                halt

        # a region for the first write
                .pos 0x50
                target1: .long 0xdeadbeef

                .align 16
                target2: .long 0xfeedcafe
```

Comments

Directives. In yas:

- .pos
- .align
- .long (4 bytes)
- .word (apparently 2 bytes)
- .byte

Labels

Commands

- Immediates should start with $, but the assembler appears to be happy if you skip that
- It looks like you can't make a hex number negative

Nits I've noticed:

- If you don't start with a .pos directive, the assembler *sometimes* messes up the first instruction

- Similarly, having comments after the last instruction/directive sometimes confuses the assembler.

---

2. Basic Instructions

---

nop

halt

rrmovl, irmovl, rmmovl, mrmovl

---

jmp

```
        .pos 0
        ping:
                jmp pong
                nop
                nop
                nop
                nop
        pong:
                jmp ping
```

jumper.ys (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/jumper.zip)

Status codes (Fig 4.5)

- "We limit the maximum address (the exact limit varies by implementation), and any access to an address beyond this limit will trigger an ADR exception."
- simguide.pdf claims: 0x0FFF

How might we trigger an ADR?

How might we trigger an INS?

codes.ys (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/codes.zip)

---

3. Arithmetic and Logic

Operations

addl	6	0
subl	6	1
andl	6	2
xorl	6	3

```
opl rA, rB    #  rB <- rB opl rA
```

[sub.ys (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/sub.zip)](https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/sub.zip)

Condition Codes

- ZF: zero
- SF: sign (msb is one)
- OF: overflow?

How can we set these flags?

[flags.ys (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/flags.zip)](https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/flags.zip)

---

4. Conditional Jumps and Moves

The Y86 provides conditional jumps and moves.

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					

Fig 4.3 (excerpt)

But if you recall from Fig 3.12, these are inherited from the x86... which tested these conditions via various strange Boolean combinations of the condition codes.

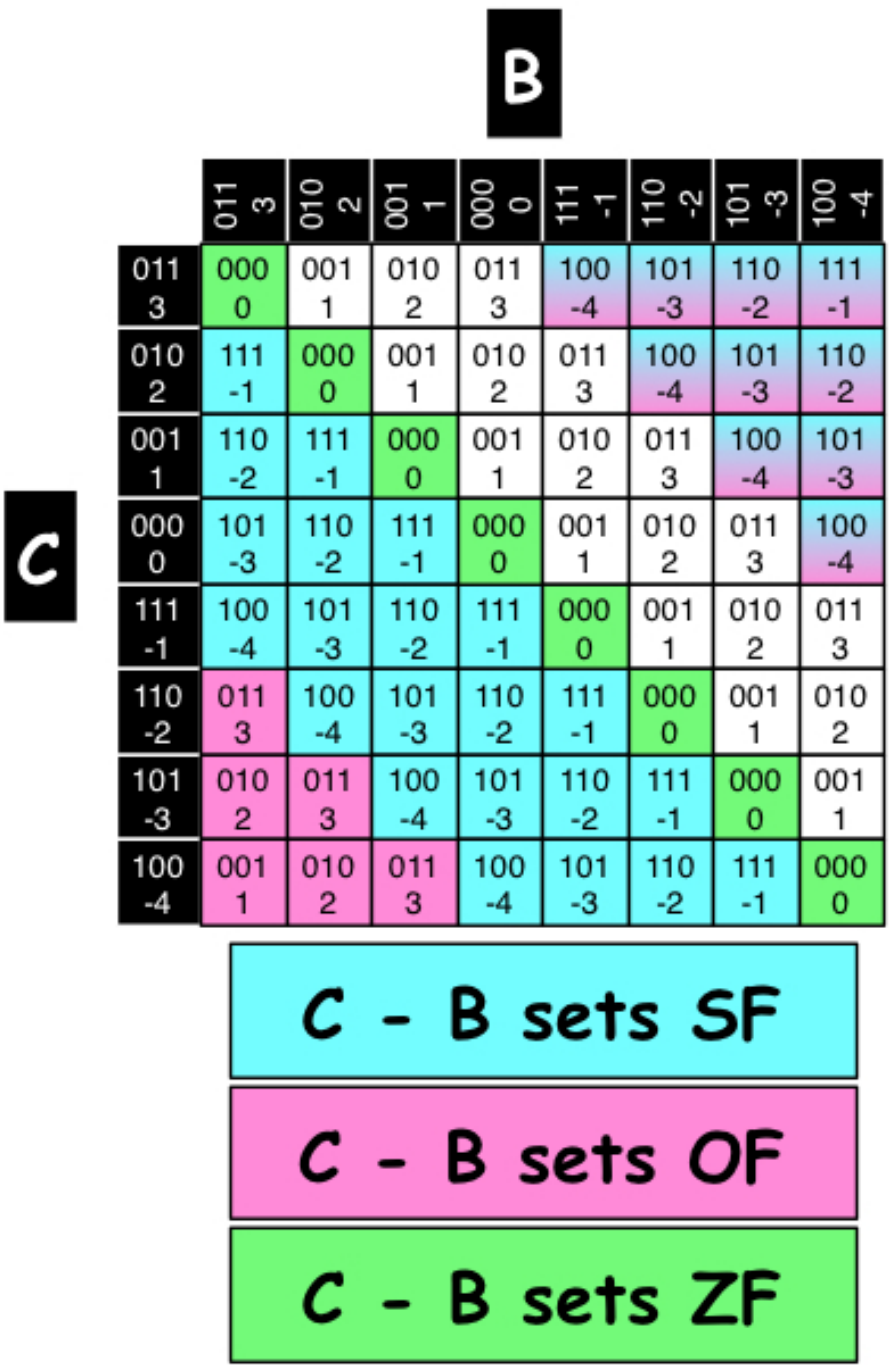
Can we derive these conditions?

Let's consider "c-b" (except we'll stick with 3-bit 2's complement integers).

```
        subl %ebx, %ecx
```

C \ B	011 3	010 2	001 1	000 0	111 -1	110 -2	101 -3	100 -4
011 3	000 0	001 1	010 2	011 3	100 -4	101 -3	110 -2	111 -1
010 2	111 -1	000 0	001 1	010 2	011 3	100 -4	101 -3	110 -2
001 1	110 -2	111 -1	000 0	001 1	010 2	011 3	100 -4	101 -3
000 0	101 -3	110 -2	111 -1	000 0	001 1	010 2	011 3	100 -4
111 -1	100 -4	101 -3	110 -2	111 -1	000 0	001 1	010 2	011 3
110 -2	011 3	100 -4	101 -3	110 -2	111 -1	000 0	001 1	010 2
101 -3	010 2	011 3	100 -4	101 -3	110 -2	111 -1	000 0	001 1
100 -4	001 1	010 2	011 3	100 -4	101 -3	110 -2	111 -1	000 0

How do the flags get set?



Can we derive the rules in Fig 3-12 for je, jne, jg, jge jl, jle?

Hence....

```
        subl %ebx,%ecx
        jl foobar  # if %ecx < %ebx
```

[comp.ys (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/comp.zip)](https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/comp.zip)

(Gosh, wouldn't it be nice to have a non-destructive "compare" instruction that just pretended to subtract, but set the flags appropriately?)

---

5. The Stack

There are four more Y86 instructions we haven't explained....

The **stack** is a LIFO data structure..

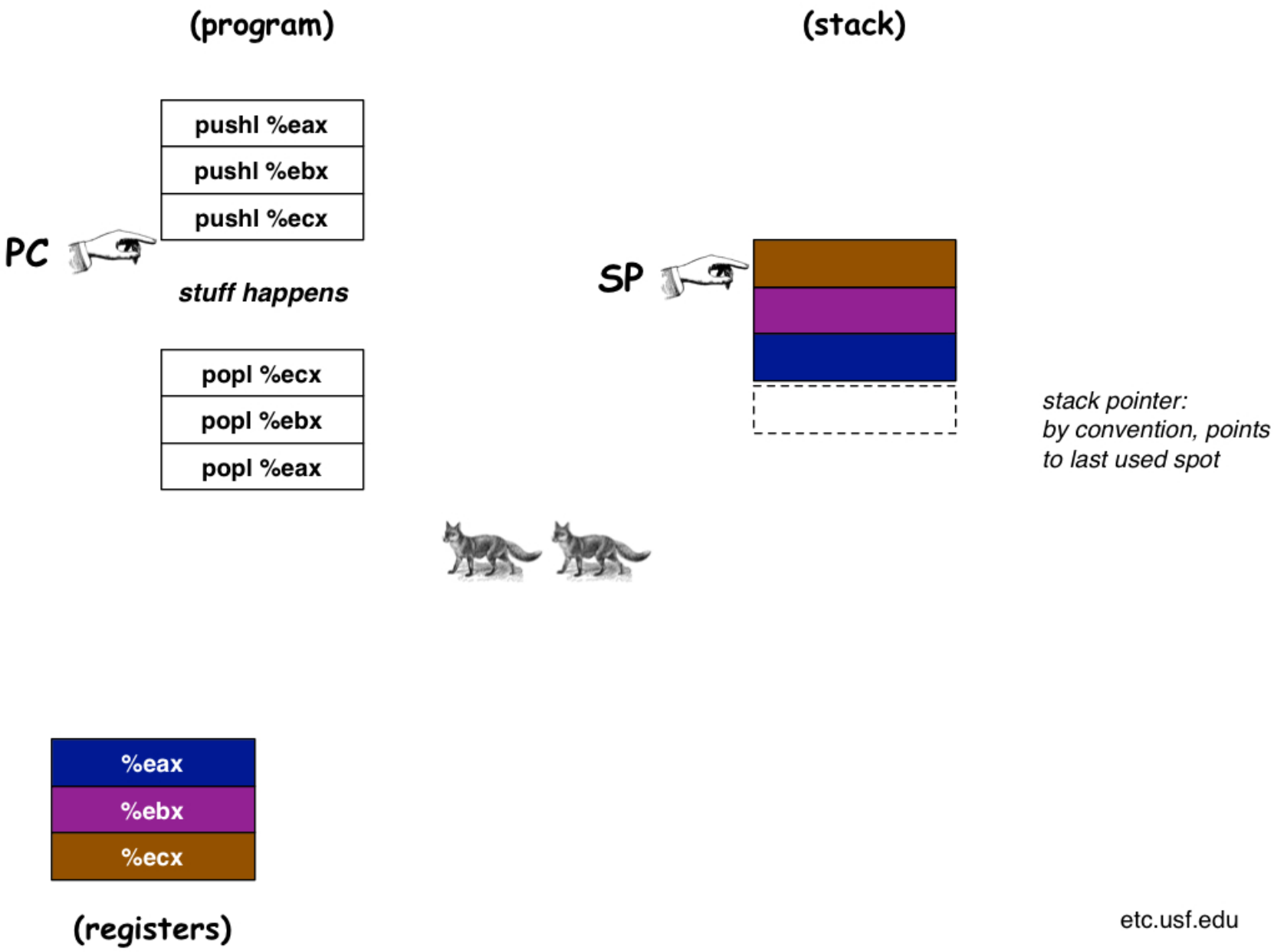
Most ISAs give direct hardware support to work with one.

- We set up a region in memory for it.
- We decide on a direction it should grow. (Convention: to lower addresses)
- We dedicate a register as a **stack pointer** to the top of the stack. (Convention: this is the address of the last thing pushed on the stack). In Y86, this is %esp
- We then have push and pop instructions

```
pushl %eax
# equiv to
#   subl $4, %esp   (if we had immediate addressing for sub!)
#   movl %eax, (%esp)
```

```
popl %edx
# equiv to
#   movl (%esp), %edx
#   addl $4, %esp   (if we had immediates!)
```

stack1.ys (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/stack1.zip)



See Fig 3.5 for another illustration (but note they have "foxes on top"---higher addresses on the top of the page).

Q: what if we push the stack pointer itself?

stack2.ys (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/12-y86/stack2.zip)
