IA32

9/22/16

From last time…

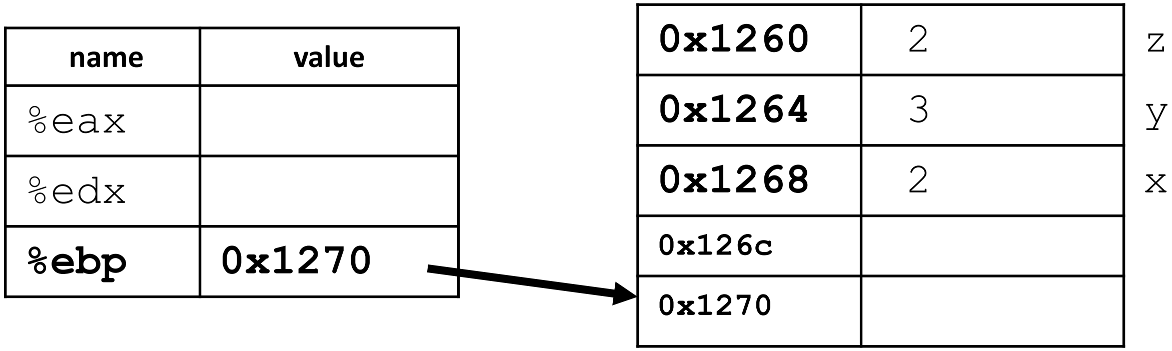
```
movl %ebp, %ecx
subl $16, %ecx
movl (%ecx), %eax
orl  %eax, -8(%ebp)
negl %eax
movl %eax, 4(%ecx)
```

name	value
%eax	?
%ecx	?
%ebp	**0x456C**

address	value
0x455C	7
0x4560	11
0x4564	5
0x4568	3
0x456C	
…	

How would you do this in IA32?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`



name	value
%eax	
%edx	
%ebp	**0x1270**

0x1260	2	z
0x1264	3	y
0x1268	2	x
0x126c		
0x1270		

```
C code: z = x ^ y
```

How would you do this in IA32?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

name	value
%eax	
%edx	
%ebp	**0x1270**

0x1260	2	z
0x1264	3	y
0x1268	2	x
0x126c		
0x1270		

```
C code:  z = x ^ y
```

A:

```
movl -8(%ebp), %eax
movl -12(%ebp), %edx
xorl %eax, %edx
movl %eax, -16(%ebp)
```

B:

```
movl -8(%ebp), %eax
movl -12(%ebp), %edx
xorl %edx, %eax
movl %eax, -16(%ebp)
```

C:

```
movl -8(%ebp), %eax
movl -12(%ebp), %edx
xorl %eax, %edx
movl %eax, -8(%ebp)
```

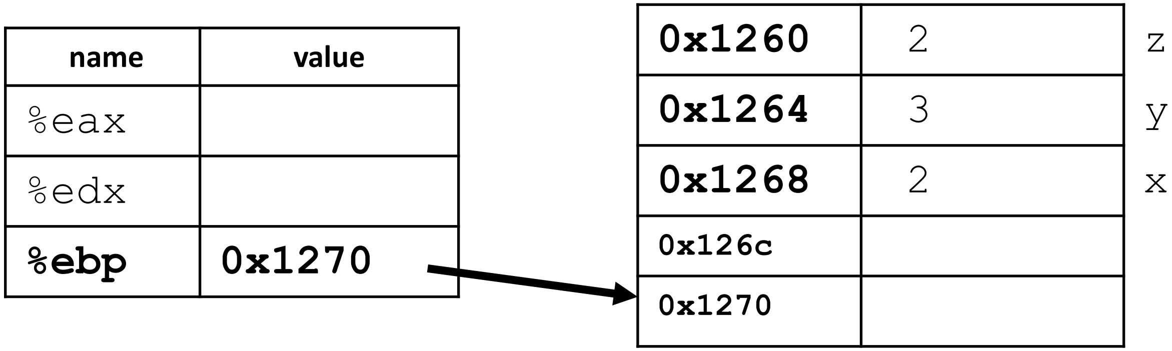
D:

```
movl -16(%ebp), %eax
movl -12(%ebp), %edx
xorl %edx, %eax
movl %eax, -8(%ebp)
```

E: none of these implements `z = x ^ y`

How would you do this in IA32?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`



name	value
`%eax`	
`%edx`	
`%ebp`	**`0x1270`**

`0x1260`	2	z
`0x1264`	3	y
`0x1268`	2	x
`0x126c`		
`0x1270`		

```
x = y >> 3 | x * 8
```

name	value
%eax	
%edx	
%ebp	**0x1270**

0x1260		z
0x1264		y
0x1268		x
0x126c		
0x1270		

```
(1)z = x ^ y
   movl -8(%ebp), %eax    # R[%eax] ← x
   movl -12(%ebp), %edx   # R[%edx] ← y
   xorl %edx, %eax        # R[%eax] ← x ^ y
   movl %eax, -16(%ebp)   # M[R[%ebp-16]] ← x^y

(2)x = y >> 3 | x * 8
      movl -8(%ebp), %eax    # R[%eax] ← x
      imull $8, %eax         # R[%eax] ← x*8
      movl -12(%ebp), %edx   # R[%edx] ← y
      rshl $3, %edx          # R[%edx] ← y >> 3
      orl  %eax, %edx        # R[%edx] ← y>>3 | x*8
      movl %edx, -8(%ebp)    # M[R[%ebp-8]] ← result
```

Recall Memory Operands

- `displacement(%reg)`
  - **e.g.**, `addl %eax, -8(%ebp)`

- IA32 allows a memory operand as the source or destination, but NOT BOTH
  - One of the operands must be a register

- This would not be allowed:
  - `addl -4(%ebp), -8(%ebp)`
  - If you wanted this, `movl` one value into a register first

Unconditional Jumping / Goto

```
int main() {
  int a = 10;
  int b = 20;

  goto label1;
  a = a + b;

label1:
  return;
```

A label is a place you <u>might</u> jump to.

Labels are ignored except for goto/jumps.

(Skipped over if encountered)

```
 int x = 20;
L1:
  int y = x + 30;
L2:
  printf(“%d, %d\n”, x, y);
```

Unconditional Jumping / Goto

```
int main() {
  int a = 10;
  int b = 20;

  goto label1;
  a = a + b;

label1:
  return;
```

```
 push    %ebp
 mov  %esp, %ebp
 sub  $16, %esp
 movl $10, -8(%ebp)
 movl $20, -4(%ebp)
 jmp label1
 movl -4(%ebp), $eax
 addl $eax, -8(%ebp)
 movl -8(%ebp), %eax
label1:
 leave
```

`jmp` isn’t very useful by itself…

We’d like to use branch instructions for:

- if/else
- switch
- for loops
- while loops

But if `jmp` were our only branch instruction, the closest we could get would be an infinite loop.

We need *conditional* jumps.

Condition Codes (or Flags)

- Set in two ways:
    1. As “side effects” produced by ALU
    2. In response to explicit comparison instructions

- IA-32, condition codes tell you:
    - If the result is zero (ZF)
    - If the result’s first bit is set (negative if signed) (SF)
    - If the result overflowed (assuming unsigned) (CF)
    - If the result overflowed (assuming signed) (OF)

Processor State in Registers

- Temporary data

  `%eax - %edi`

- Location of runtime stack

  `%ebp, %esp`

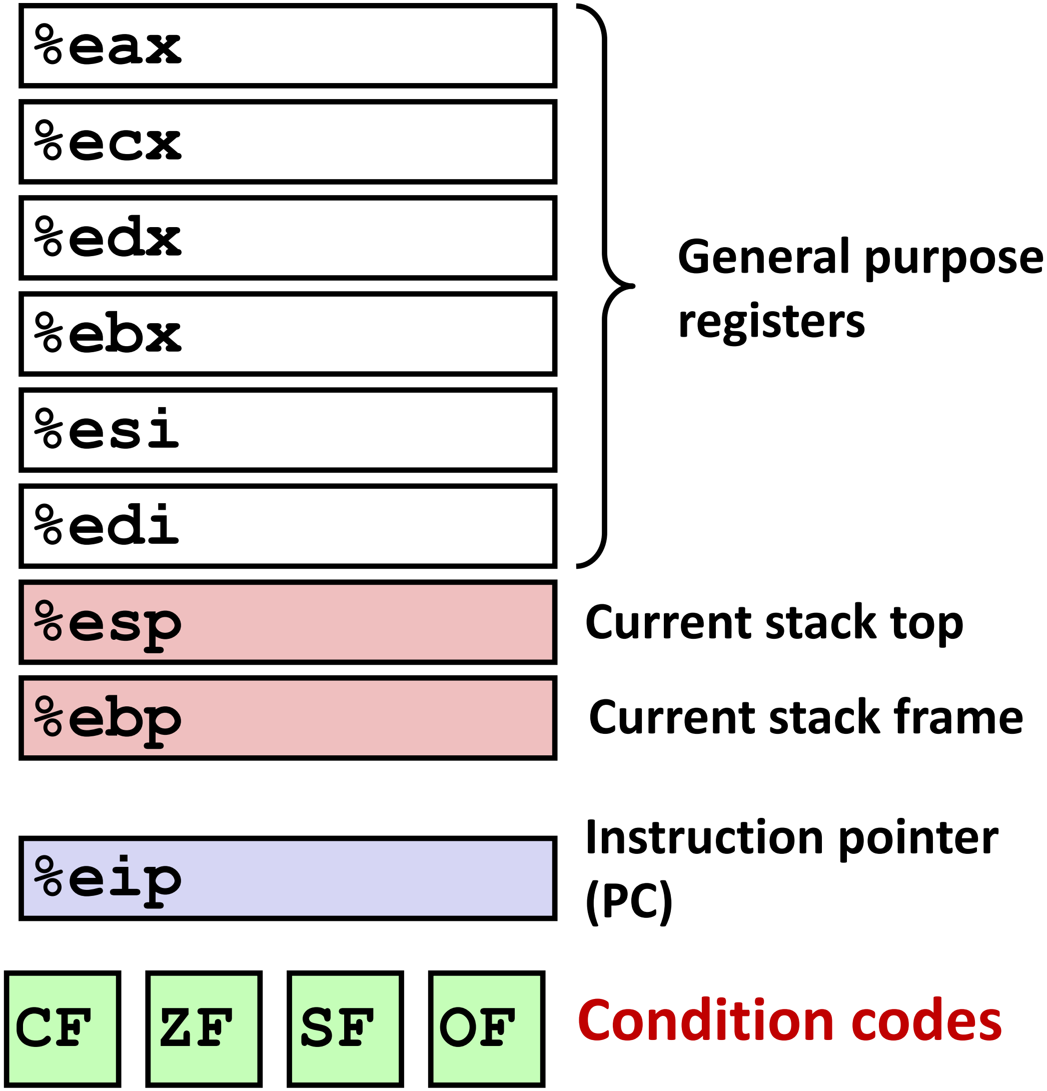
- Location next instruction

  `%eip`

- Status of recent tests

  `%EFLAGS:`

  `CF, ZF, SF, OF`



%eax

%ecx

%edx

%ebx

%esi

%edi

General purpose registers

%esp

Current stack top

%ebp

Current stack frame

%eip

Instruction pointer (PC)

CF

ZF

SF

OF

Condition codes

Instructions that set condition codes

1. Arithmetic/logic side effects (addl, subl, orl, etc.)

2. `CMP` and `TEST`:

   `cmpl b,a` like computing **`a-b`** without storing result

   - Sets `OF` if overflow, Sets `CF` if carry-out,
     Sets `ZF` if result zero, Sets `SF` if results is negative

   `testl b,a` like computing **`a&b`** without storing result

   - Sets `ZF` if result is zero, sets `SF` if `a&b < 0`
     `OF` and `CF` flags are zero (no overflow with `&`)

Which flags would this `subl` set?

- Suppose `%eax` holds 5, `%ecx` holds 7

```
subl $5, %eax
```

If the result is zero (ZF)
If the result's first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF

Which flags would this `cmpl` set?

- Suppose `%eax` holds 5, `%ecx` holds 7

```
cmpl %ecx, %eax
```

If the result is zero (ZF)
If the result's first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF

Conditional Jumping

- Jump based on which condition codes are set

Jump Instructions: (fig. 3.12)

You do not need to memorize these.

	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned jg)
`jb`	`CF`	Below (unsigned)

Example Scenario

```
int userval;
scanf(“%d”, &userval);

if (userval == 42) {
  userval += 5;
} else {
  userval -= 10;
}
…
```

- Suppose user gives us a value via scanf

- We want to check to see if it equals 42
  - If so, add 5
  - If not, subtract 10

How would we use jumps/CCs for this?

```
int userval;
scanf(“%d”, &userval);

if (userval == 42) {
  userval += 5;
} else {
  userval -= 10;
}
…
```

Assume userval is stored in %eax at this point.

How would we use jumps/CCs for this?

```
int userval;
scanf("%d", &userval);
```

Assume userval is stored in %eax at this point.

```
if (userval == 42) {
  userval += 5;
} else {
  userval -= 10;
}
```

…

(A)

```
  cmpl $42, %eax
  je L2
L1:
  subl $10, %eax
  jmp DONE
L2:
  addl $5, %eax
DONE:
  …
```

(B)

```
  cmpl $42, %eax
  jne L2
L1:
  subl $10, %eax
  jmp DONE
L2:
  addl $5, %eax
DONE:
  …
```

(C)

```
  cmpl $42, %eax
  jne L2
L1:
  addl $5, %eax
  jmp DONE
L2:
  subl $10, %eax
DONE:
  …
```

Loops via `goto`

Goal: translate for loops and while loops to IA32.

- We know how to translate a for loop to a while loop, so let’s focus on while loops.
- Intermediate step: translate c code with a while loop into c code with `goto` statements.

Translate `while` → `goto`

```
int i=1, j=100, k=0;
while(i < j){
  i *= 2;
  j -= i;
}
k = j + i;
```

Translate goto → IA32

~~int i=1, j=100, k=0;~~

```
L1:
   if(i >= j) goto L2;
   i *= 2;
   j -= i;
   goto L1;

L2:
   k = j + i;
```

0x8B00	2	k
0x8B04	3	j
0x8B08	2	i
0x8B0c		
0x8B10		(%ebp)

Hint:

cmpl
jge
jmp