HARVARD
School of Engineering and Applied Sciences

Assembly Programming: Data operations, and control flow

CS61, Lecture 3

Prof. Stephen Chong

September 9, 2010

Announcements

- Sections
  - Signup at https://www.section.fas.harvard.edu/ by **5pm Friday**
- Lab 1 released!
  - See website for details.
  - Due Tues 21 Sep
- Office hours start next week
  - Office hours in Science Center Computer Lab
- See website for lecture notes, announcements, etc.

Topics for today

- More on assembly language!
  - mov example
  - Arrays
  - LEAL: Load Effective Address
  - Data operations
  - x86-64
  - Control flow

Last lecture

- `mov`*x source, dest*
  - *x* is one of `b`, `w`, `l`
  - Move data from *source* to *dest*
- Addressing modes
  - Immediate
    - $*num*
    - e.g., `$0xdeadbeef`, `$42`, `$-536`
  - Register
    - e.g, `%eax`, `%ebx`
  - Memory
    - *Imm*(*base*, *index*, *scale*)
    - e.g., `0x804974c`, `(%eax)`, `0x8(%ebp)`

Example: swap

```c
void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t0;
  *yp = t1;
}
```

```
swap:
  pushl %ebp
  pushl %ebx
  movl %esp,%ebp

  movl 12(%ebp),%ecx
  movl 8(%ebp),%edx
  movl (%ecx),%eax
  movl (%edx),%ebx
  movl %eax,(%edx)
  movl %ebx,(%ecx)

  movl -4(%ebp),%ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

Set up

Body

Finish

Example: swap

```c
void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t0;
  *yp = t1;
}
```

```
swap:
  pushl %ebp
  pushl %ebx
  movl %esp,%ebp

  movl 12(%ebp),%ecx
  movl 8(%ebp),%edx
  movl (%ecx),%eax
  movl (%edx),%ebx
  movl %eax,(%edx)
  movl %ebx,(%ecx)

  movl -4(%ebp),%ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

Set up

Body

Finish

Understanding swap

```
void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t0;
  *yp = t1;
}
```

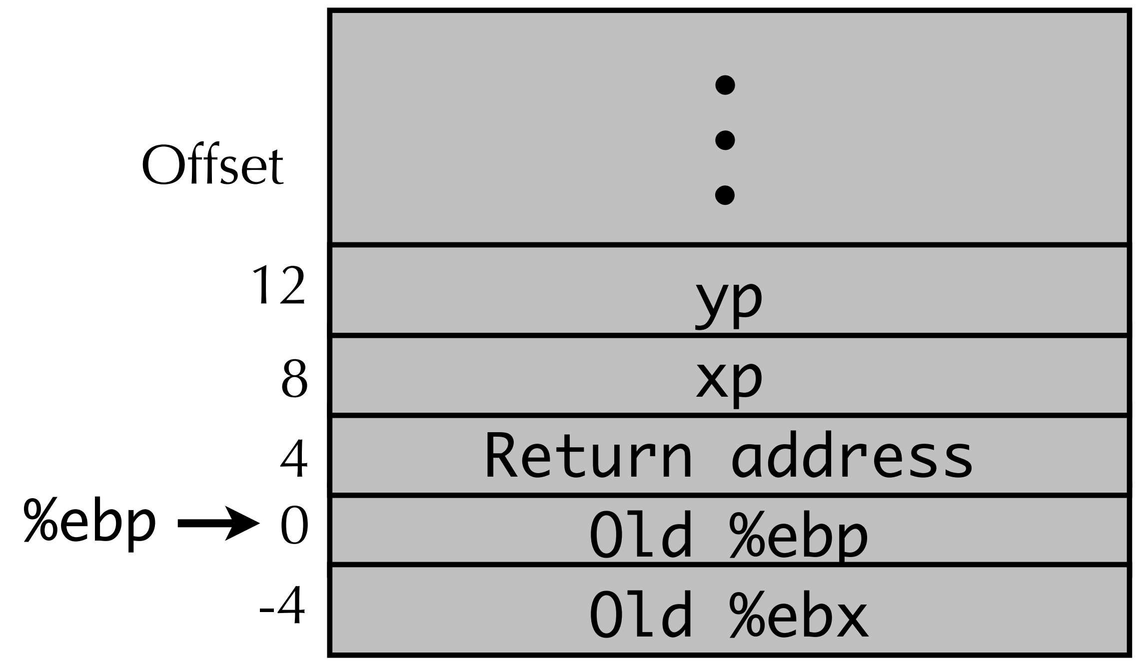
```
  movl 12(%ebp),%ecx
  movl 8(%ebp),%edx
  movl (%ecx),%eax
  movl (%edx),%ebx
  movl %eax,(%edx)
  movl %ebx,(%ecx)
```

Body

Understanding swap

```
void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t0;
  *yp = t1;
}
```

Stack



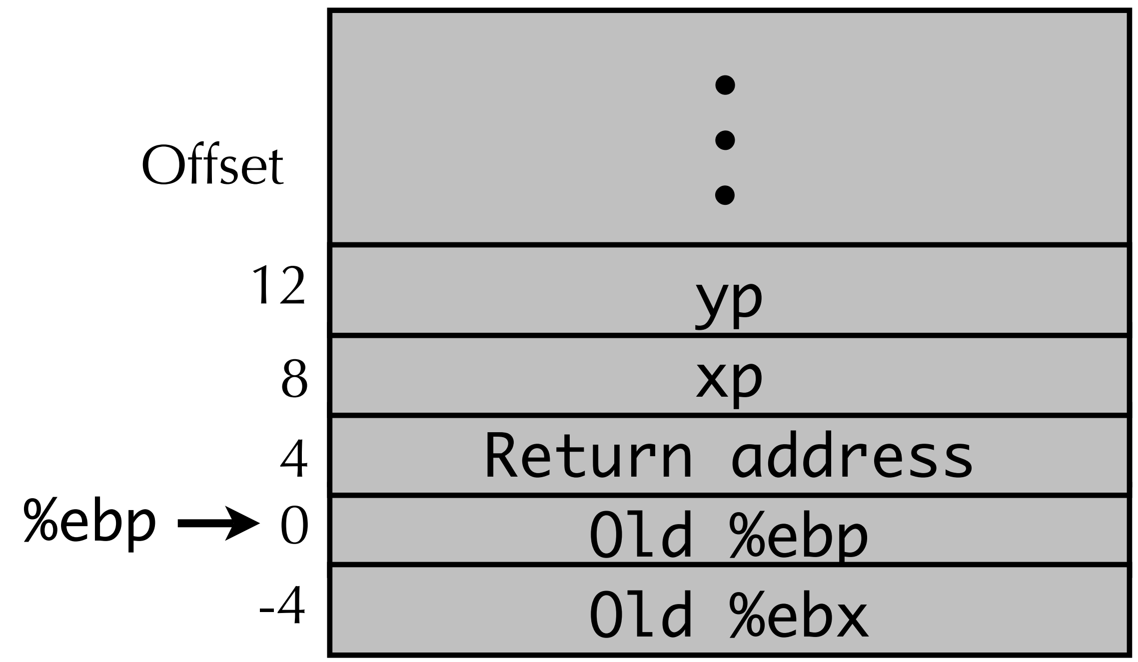

```
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
```

Body

Understanding swap

```c
void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t0;
  *yp = t1;
}
```

Stack

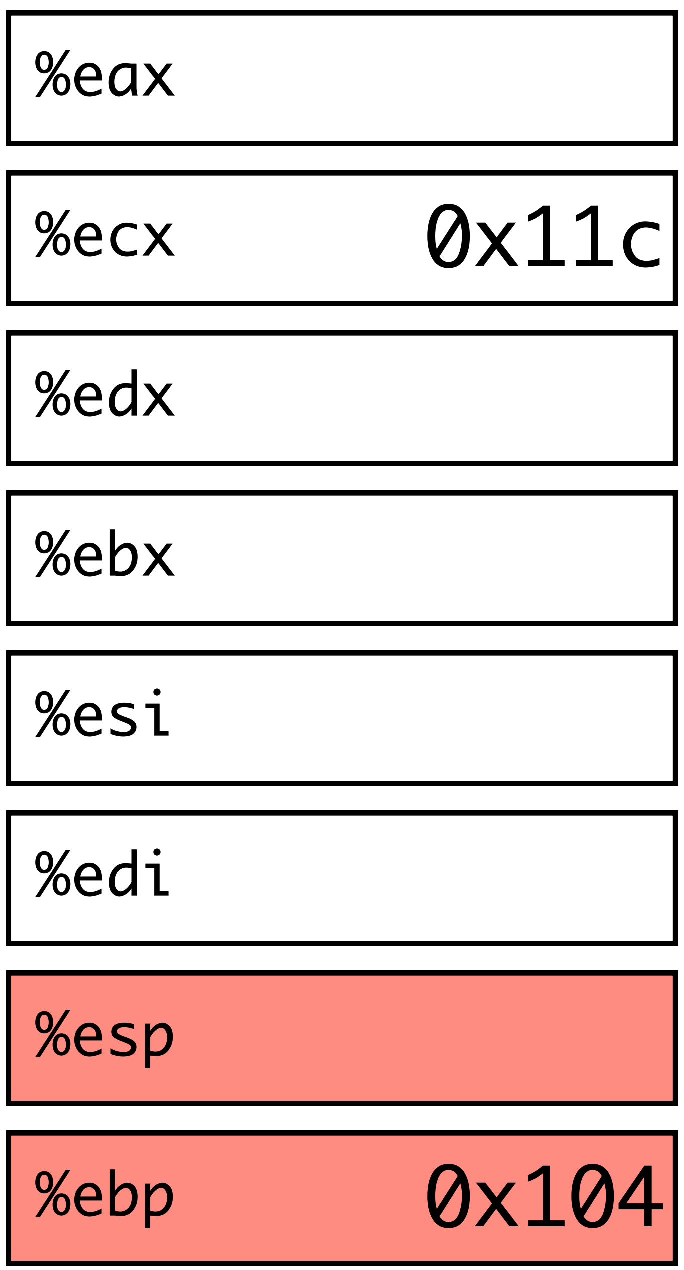


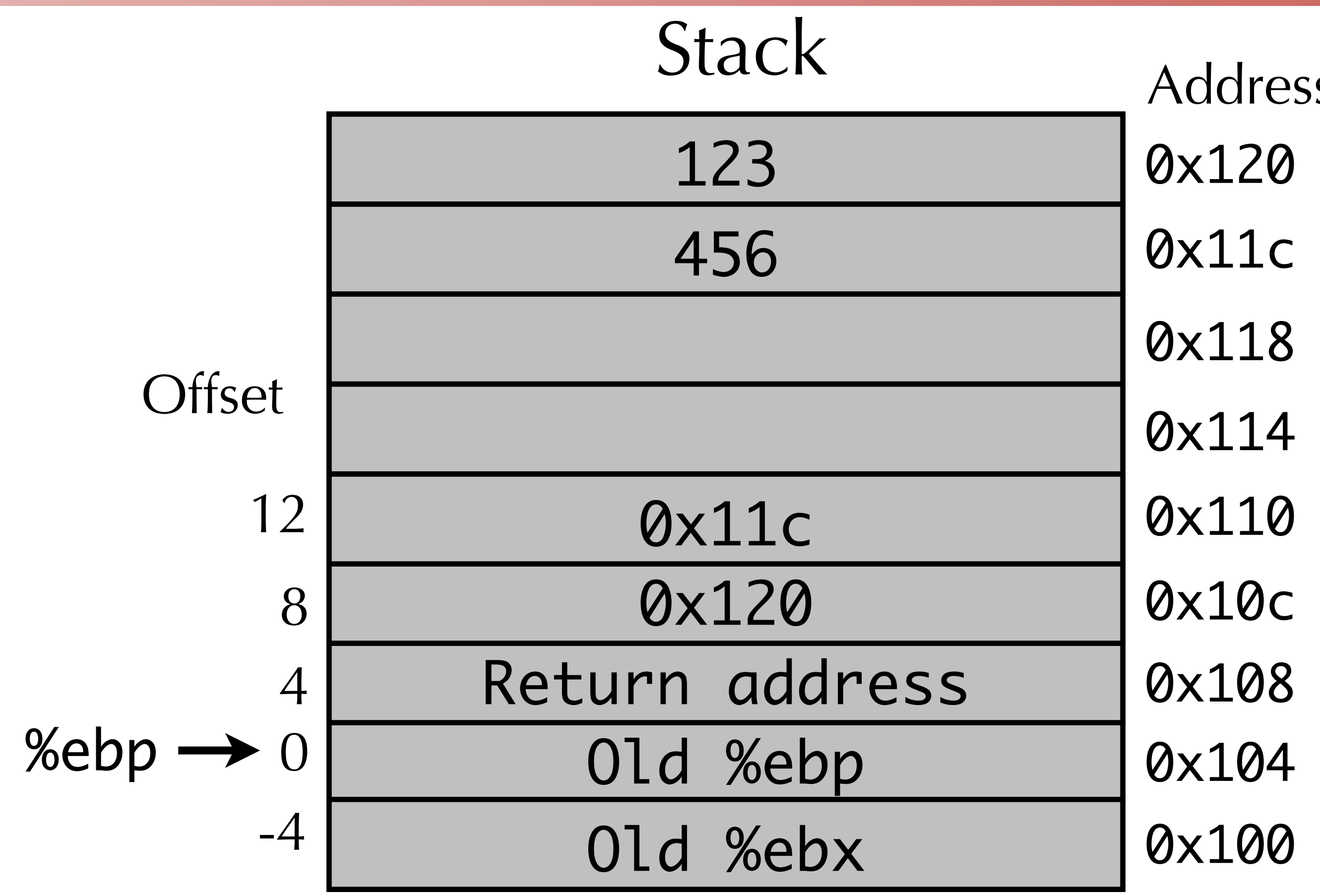
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Body

Understanding swap

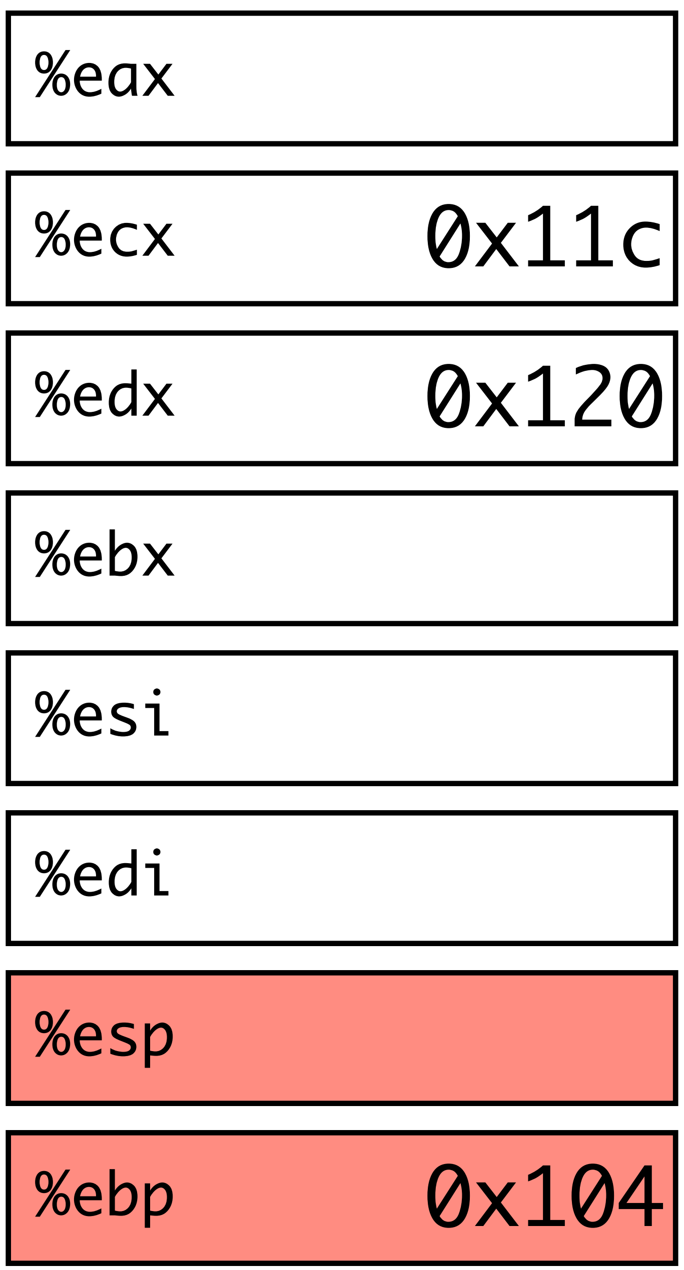


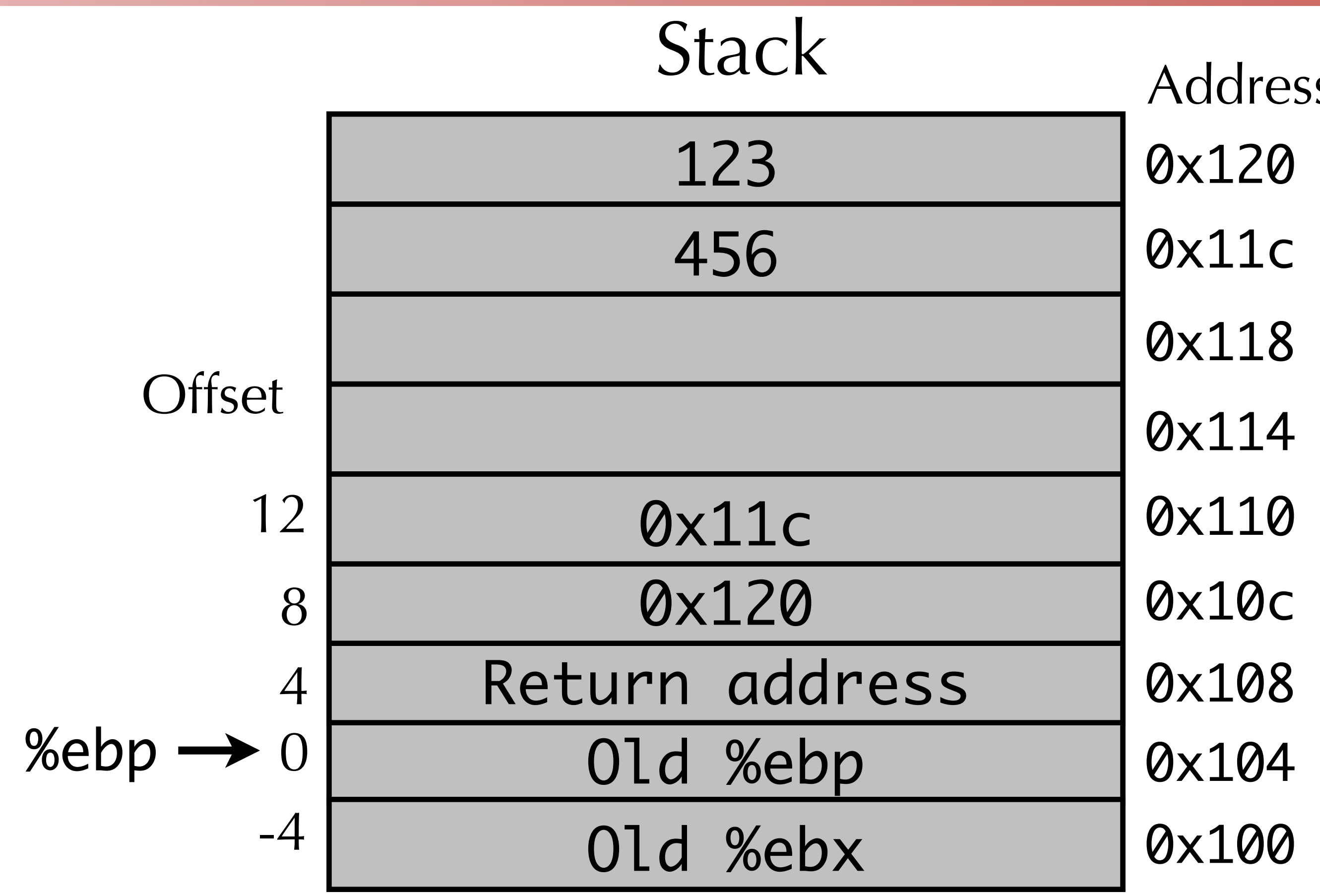



```
movl 12(%ebp),%ecx    # %ecx = yp
movl 8(%ebp),%edx     # %edx = xp
movl (%ecx),%eax      # %eax = *yp
movl (%edx),%ebx      # %ebx = *xp
movl %eax,(%edx)      # *xp = %eax
movl %ebx,(%ecx)      # *yp = %ebx
```

Body

Understanding swap






```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Body

Understanding swap

%eax	456
%ecx	0x11c
%edx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Stack

	Offset	Stack	Address
		123	0x120
		456	0x11c
			0x118
			0x114
	12	0x11c	0x110
	8	0x120	0x10c
	4	Return address	0x108
%ebp →	0	Old %ebp	0x104
	-4	Old %ebx	0x100

Body

```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Understanding swap

%eax	456
%ecx	0x11c
%edx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Stack

Offset	Stack	Address
	123	0x120
	456	0x11c
		0x118
		0x114
12	0x11c	0x110
8	0x120	0x10c
4	Return address	0x108
%ebp → 0	Old %ebp	0x104
-4	Old %ebx	0x100

```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Body

Understanding swap

%eax	456
%ecx	0x11c
%edx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

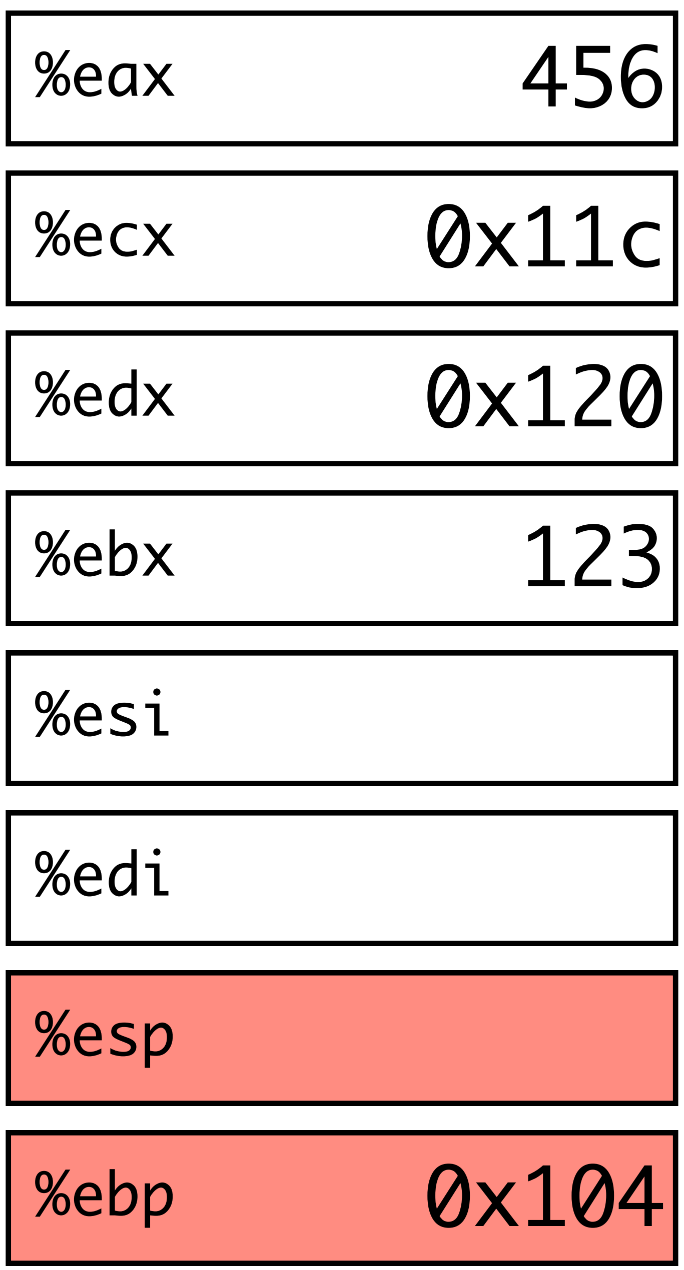
Stack

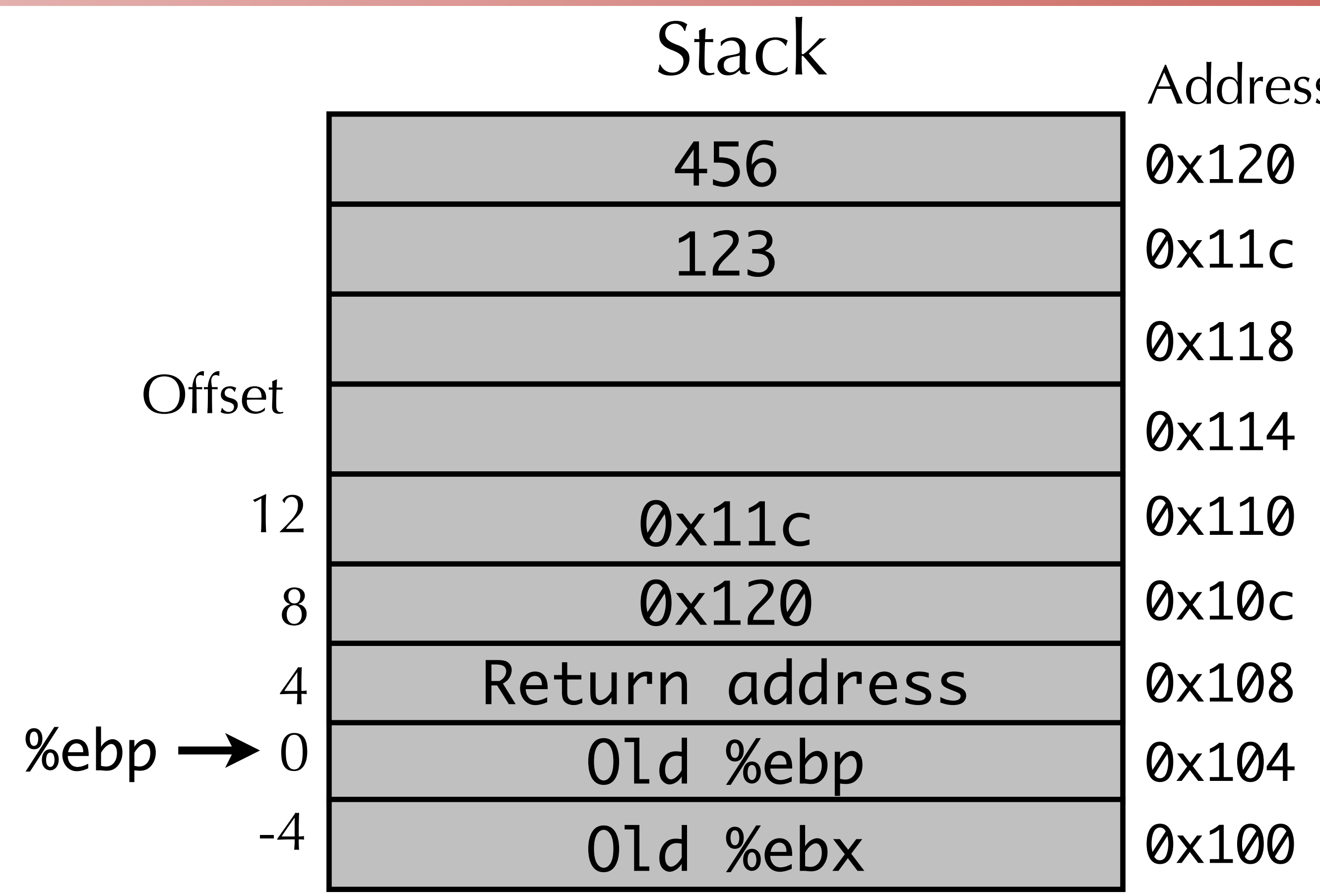
Offset	Stack	Address
	456	0x120
	456	0x11c
		0x118
		0x114
12	0x11c	0x110
8	0x120	0x10c
4	Return address	0x108
%ebp → 0	Old %ebp	0x104
-4	Old %ebx	0x100

Body

```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Understanding swap






```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Body

Arrays

Arrays

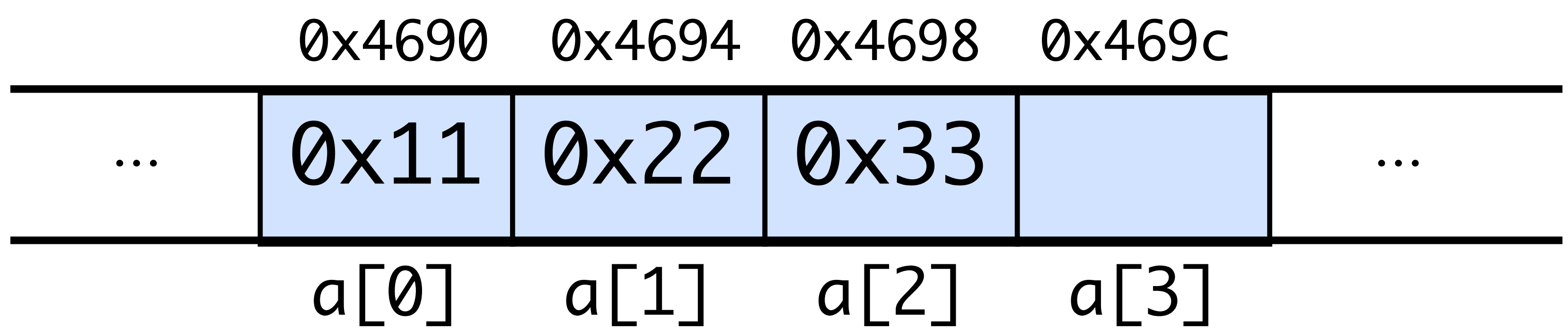
- An array can be thought of as a pointer to the first element of the array
    - ```
      int a[3];
      a[0] = 0x11;
      a[1] = 0x22;
      a[2] = 0x33;
      ```
      ```

Arrays

- An array can be thought of as a pointer to the first element of the array
    - ```
      int a[3];
      a[0] = 0x11;
      a[1] = 0x22;
      a[2] = 0x33;
      ```
- In x86 assembly this becomes
    - ```
      movl $0x4690, %eax      ; Set %eax to address of 'a'
      movl $0x11, (%eax)      ; a[0] = 0x11
      movl $0x22, 4(%eax)     ; a[1] = 0x22
      movl $0x33, 8(%eax)     ; a[2] = 0x33
      ```
- Why do we add 4 to `%eax` each time?

Arrays

- An array can be thought of as a pointer to the first element of the array
  - ```
    int a[3];
    a[0] = 0x11;
    a[1] = 0x22;
    a[2] = 0x33;
    ```



- In x86 assembly this becomes
  - ```
    movl $0x4690, %eax       ; Set %eax to address of 'a'
    movl $0x11, (%eax)       ; a[0] = 0x11
    movl $0x22, 4(%eax)      ; a[1] = 0x22
    movl $0x33, 8(%eax)      ; a[2] = 0x33
    ```
- Why do we add 4 to `%eax` each time?

Arrays

- An array can be thought of as a pointer to the first element of the array
  - ```
    int a[3];
    a[0] = 0x11;
    a[1] = 0x22;
    a[2] = 0x33;
    ```

	0x4690	0x4694	0x4698	0x469c	
…	0x11	0x22	0x33		…
	a[0]	a[1]	a[2]	a[3]	

- In x86 assembly this becomes
  - ```
    movl $0x4690, %eax       ; Set %eax to address of 'a'
    movl $0x11, (%eax)       ; a[0] = 0x11
    movl $0x22, 4(%eax)      ; a[1] = 0x22
    movl $0x33, 8(%eax)      ; a[2] = 0x33
    ```
- Why do we add 4 to `%eax` each time?
  - Each integer is represented with 4 bytes, so the address of a[1] and a[2] differ by 4

Address computation instruction

Address computation instruction

- `leal` *src*`,` *dest*
  - *src* is address mode expression
    - e.g., `(%eax)` or `0x8(%ebp)`
    - Most generally, *Imm*(*base*, *index*, *scale*)
  - Set *dest* to the address denoted by expression *src*
  - “Load effective address”
- Can compute an address without a memory reference
  - E.g., compilation of C code `p = &x[i]`

```
leal (%eax, %ebx, 4), %edx
```

Address computation instruction

- `leal` *src*`,` *dest*
  - *src* is address mode expression
    - e.g., `(%eax)` or `0x8(%ebp)`
    - Most generally, *Imm*(*base*, *index*, *scale*)
  - Set *dest* to the address denoted by expression *src*
  - "Load effective address"
- Can compute an address without a memory reference
  - E.g., compilation of C code `p = &x[i]`

    ```
    leal (%eax, %ebx, 4), %edx
    ```
- Can also be used to compute arithmetic expressions!
  - Anything of the form $x + y*k$, where $k = 1, 2, 4,$ or 8

Some arithmetic operations

- Two operand instructions

Format		**Equivalent C computation**	
`addl`	*Src,Dest*	*Dest = Dest + Src*	
`subl`	*Src,Dest*	*Dest = Dest - Src*	
`imull`	*Src,Dest*	*Dest = Dest * Src*	
`sall`	*Src,Dest*	*Dest = Dest << Src*	Also called `shll`
`sarl`	*Src,Dest*	*Dest = Dest >> Src*	Arithmetic
`shrl`	*Src,Dest*	*Dest = Dest >> Src*	Logical
`xorl`	*Src,Dest*	*Dest = Dest ^ Src*	
`andl`	*Src,Dest*	*Dest = Dest & Src*	
`orl`	*Src,Dest*	*Dest = Dest | Src*	

- No distinction between signed and unsigned int. Why?

Some more arithmetic operations

- One operand instructions

Format	**Equivalent C computation**
`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* - 1
`negl` *Dest*	*Dest* = -*Dest*
`notl` *Dest*	*Dest* = ~*Dest*

- See textbook for more instructions

Example: logical

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set up

Body

Finish

Example: logical

```c
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set up

Body

Finish

Example: logical

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set up

Body

Finish

```
movl 8(%ebp),%eax        #   %eax = x
```

Example: logical

```c
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set up

Body

Finish

```
movl 8(%ebp),%eax      #   %eax = x
xorl 12(%ebp),%eax     #   %eax = x^y (t1)
```

Example: logical

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set up

Body

Finish

```
movl 8(%ebp),%eax      #   %eax = x
xorl 12(%ebp),%eax     #   %eax = x^y (t1)
sarl $17,%eax          #   %eax = t1>>17 (t2)
```

Example: logical

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set up

Body

Finish

```
movl 8(%ebp),%eax      #   %eax = x
xorl 12(%ebp),%eax     #   %eax = x^y (t1)
sarl $17,%eax          #   %eax = t1>>17 (t2)
andl $8185,%eax        #   %eax = t2 & 8185
```

Example: logical

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set up

Body

Finish

```
movl 8(%ebp),%eax     #   %eax = x
xorl 12(%ebp),%eax    #   %eax = x^y (t1)
sarl $17,%eax         #   %eax = t1>>17 (t2)
andl $8185,%eax       #   %eax = t2 & 8185
```

$1<<13 = 2^{13} = 8192$

$8192 - 7 = 8185$

Topics for today

- More on assembly language!
  - mov example
  - Arrays
  - LEAL: Load Effective Address
  - Data operations
  - x86-64
  - Control flow

x86-64

- x86 (aka IA32) instruction set defined in about 1985
  - Has been dominant instruction format for many years
- x86-64 extends x86 to 64 bits
  - Originally developed by AMD (Advanced Micro Devices), Intel's competitor
  - Also referred to as AMD64, Intel64, and x64
- Currently in transition from 32 bits to 64 bits
  - Most new machines you buy will be 64 bits

Differences between x86 and x86-64

- Data types

C declaration	Intel data type	Assembly code suffix	32-bit	64-bit
`char`	Byte	b	1	1
`short int`	Word	w	2	2
`int`	Double word	l	4	4
`long int`	Quad word	q	4	8
`long long int`	Quad word	q	8	8
`char *`	Quad word	q	4	8
`float`	Single precision	s	4	4
`double`	Double precision	d	8	8
`long double`	Extended precision	t	10/16	10/12

Differences between x86 and x86-64

- Registers
  - x86 has 8 general purpose registers
  - x86-64 has 16 general purpose registers
    - Extend existing registers and add new ones
    - Make `%ebp`/`%rbp` general purpose

64-bit	32-bit	64-bit	32-bit
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

x86-64 instructions

- Long word `l` (4 Bytes) ↔ Quad word `q` (8 Bytes)
- New instructions:
  - `movl` → `movq`
  - `addl` → `addq`
  - `sall` → `salq`
  - etc.
- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - E.g., `addl`

Topics for today

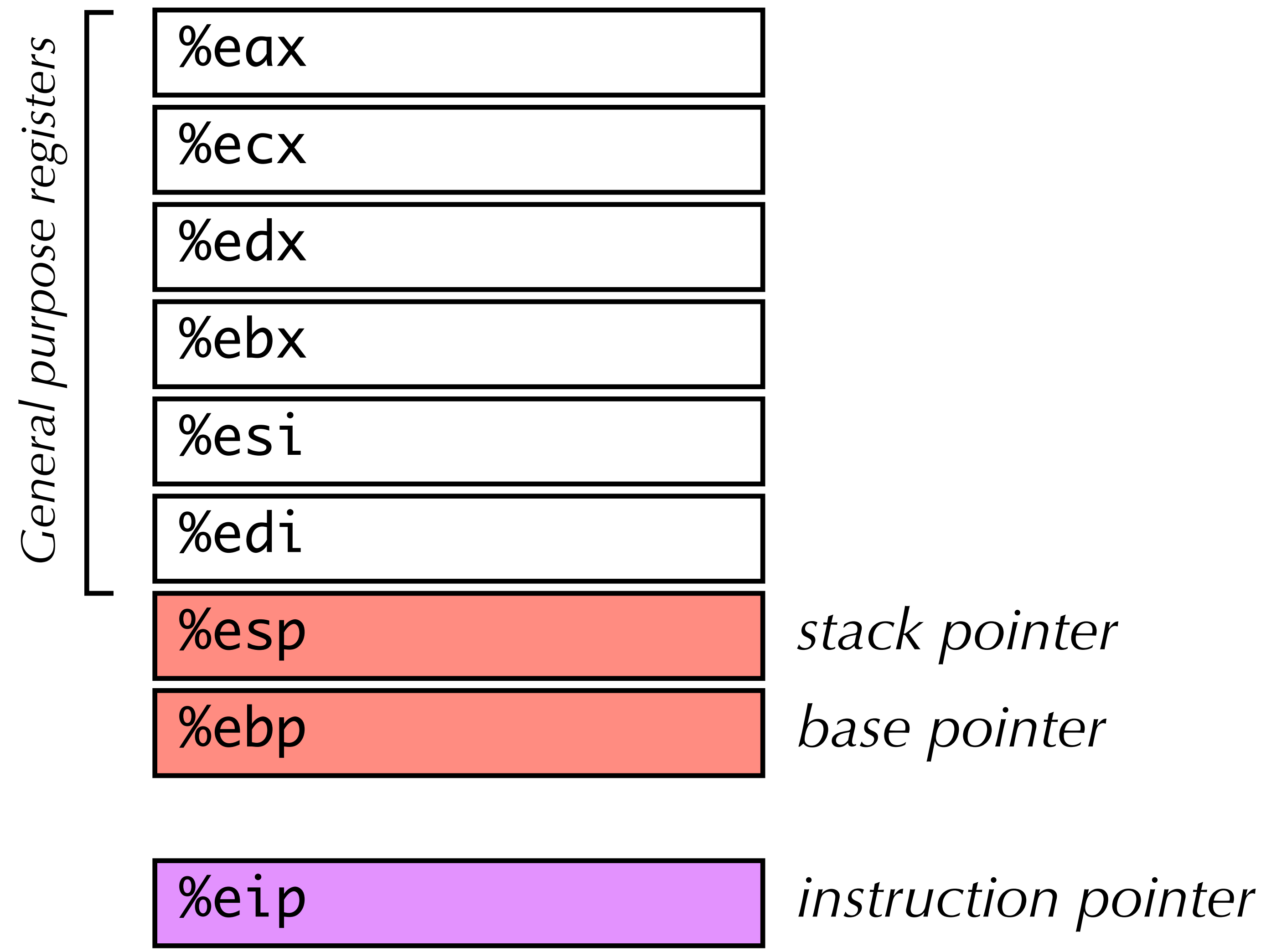
- More on assembly language!
  - Arrays
  - Stack
  - LEAL: Load Effective Address
  - Data operations
  - x86-64
  - Control flow

Control flow

- **Control flow** is the general term for any code that controls which parts of a program are executed.
- Examples in C
  - `if (`*expr*`) { ... } else { ... }`
  - `do { .... } while (`*expr*`);`
  - `while (`*expr*`) { .... }`
  - `for (`*expr1*`; `*expr2*`; `*expr3*`) { ... }`
  - C "`goto`" statement
  - `switch (`*expr*`) { case `*val1*`: ...; case `*val1*`: ...; default: ...; }`

Processor state

- Control flow is about how the value of the instruction pointer changes



Simplest case: jmp instruction

- Operation `jmp` *label* causes processor to "jump" to a new instruction and execute from there

```
.L3:
        movl    12(%ebp), %eax        eax = arg2
        addl    8(%ebp), %eax         eax += arg1
        movl    %eax, -4(%ebp)        temp = eax
        jmp     .L6                   goto .L6
.L5:
        movl    12(%ebp), %eax        eax = arg2
        imull   8(%ebp), %eax         eax *= arg1
        movl    %eax, -4(%ebp)        temp = eax
.L6:
        movl    -4(%ebp), %eax        eax = temp
        leave                         return
        ret
```

- The label (".L6") is just a symbolic reference to a specific instruction in the program.
- Once compiled to a binary, the .L6 will be replaced by a memory address.

Simplest case: jmp instruction

- Operation `jmp` *label* causes processor to "jump" to a new instruction and execute from there

```
.L3:
        movl    12(%ebp), %eax        eax = arg2
        addl    8(%ebp), %eax         eax += arg1
        movl    %eax, -4(%ebp)        temp = eax
        jmp     .L6                   goto .L6
.L5:
        movl    12(%ebp), %eax        eax = arg2
        imull   8(%ebp), %eax         eax *= arg1
        movl    %eax, -4(%ebp)        temp = eax
.L6:
        movl    -4(%ebp), %eax        eax = temp
        leave                         return
        ret
```

- The label (".L6") is just a symbolic reference to a specific instruction in the program.
- Once compiled to a binary, the .L6 will be replaced by a memory address.

Symbolic labels

- The assembler replaces symbolic labels with the actual memory address when converting to machine code.
  - They are just "placeholders" for memory addresses.

Output from gcc -S

```
.L3:
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        jmp     .L6
.L5:
        movl    12(%ebp), %eax
        imull   8(%ebp), %eax
        movl    %eax, -4(%ebp)
.L6:
        movl    -4(%ebp), %eax
        leave
        ret
```

Output from objdump

```
08048476: movl    12(%ebp), %eax
08048477: addl    8(%ebp), %eax
08040479: movl    %eax, -4(%ebp)
0804847c: jmp     08048489

0804847f: movl    12(%ebp), %eax
08048481: imull   8(%ebp), %eax
08048487: movl    %eax, -4(%ebp)

08048489: movl    -4(%ebp), %eax
0804848d: leave
0804848e: ret
```

Symbolic labels

- The assembler replaces symbolic labels with the actual memory address when converting to machine code.
  - They are just "placeholders" for memory addresses.

Output from gcc -S

```
.L3:
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        jmp     .L6
.L5:
        movl    12(%ebp), %eax
        imull   8(%ebp), %eax
        movl    %eax, -4(%ebp)
.L6:
        movl    -4(%ebp), %eax
        leave
        ret
```

Output from objdump

```
08048476: movl    12(%ebp), %eax
08048477: addl    8(%ebp), %eax
08040479: movl    %eax, -4(%ebp)
0804847c: jmp     08048489

0804847f: movl    12(%ebp), %eax
08048481: imull   8(%ebp), %eax
08048487: movl    %eax, -4(%ebp)

08048489: movl    -4(%ebp), %eax
0804848d: leave
0804848e: ret
```

Conditional branching

Conditional branching

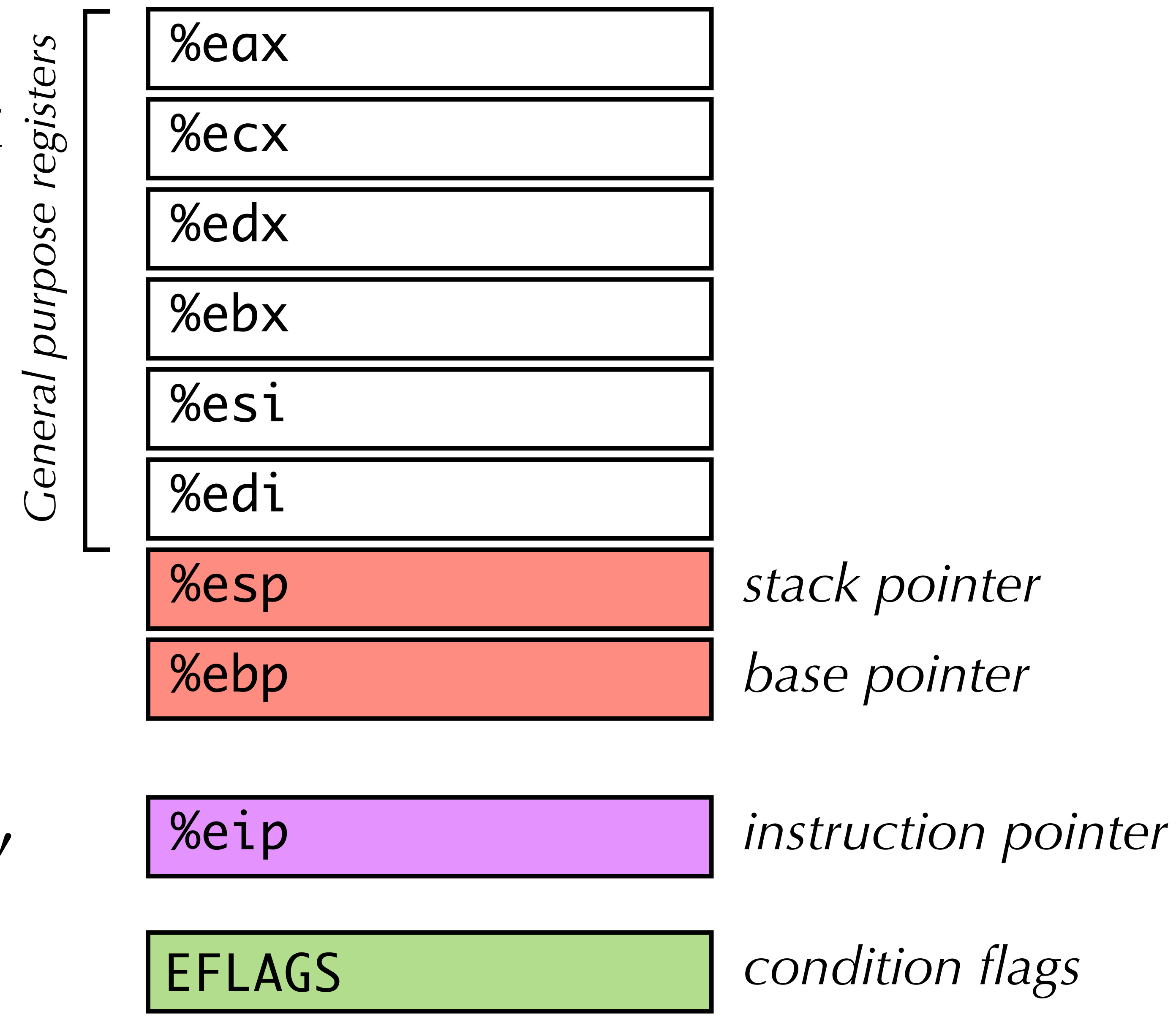
- `jmp` basically implements goto
  - Always same control flow
- How do we implement if statements, loops, etc?
  - Not always the same control flow

Conditional branching

- `jmp` basically implements goto
  - Always same control flow
- How do we implement if statements, loops, etc?
  - Not always the same control flow
- Two kinds of instructions
- Comparison instructions (`cmpl`, `testl`, etc.)
  - Compare values of two registers
  - Set **condition flags** based on result
- Conditional branch instructions (`je`, `jne`, `jg`, etc.)
  - Jump depending on current value of condition flags

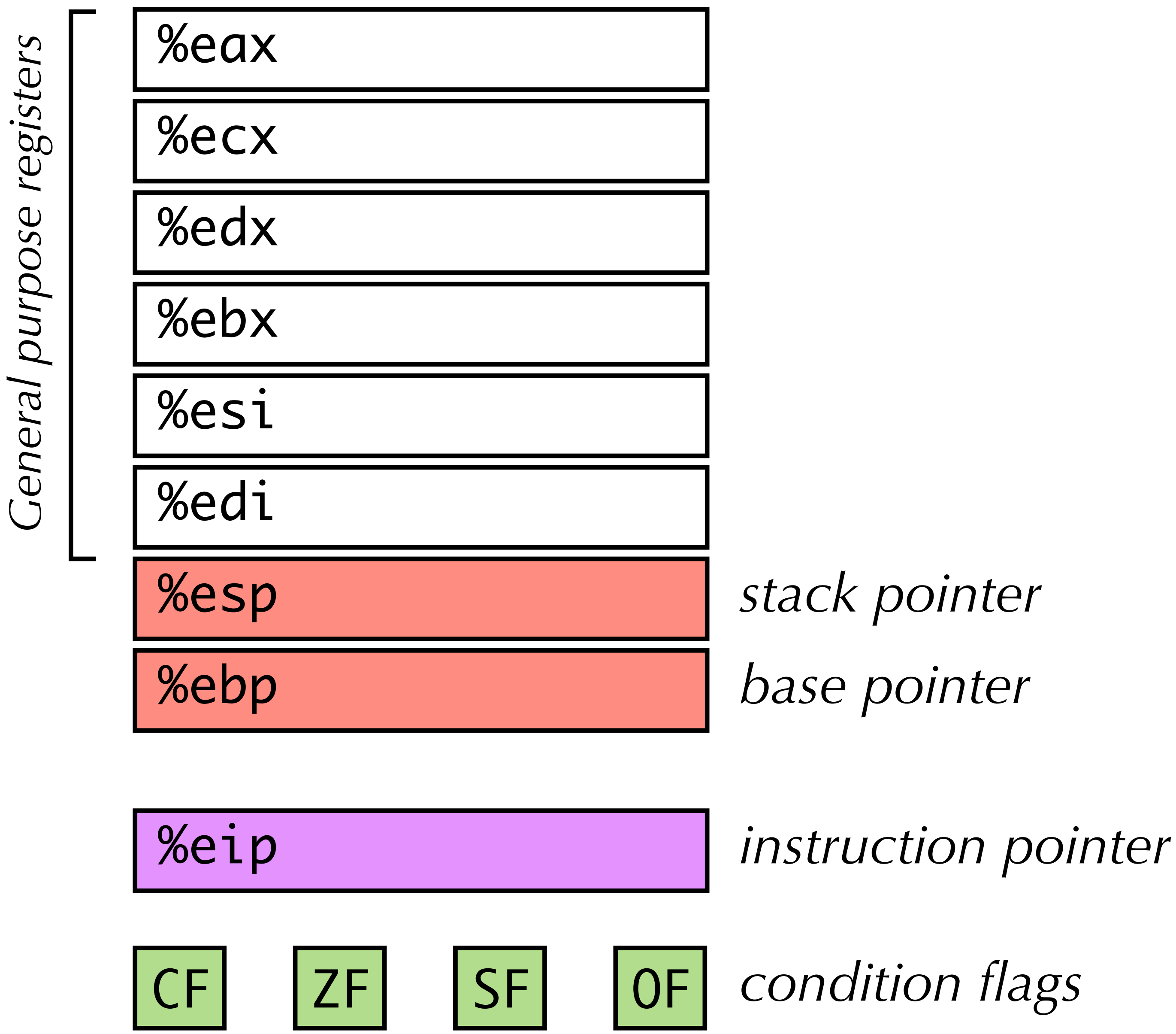
Condition flags

- Bits maintained by the processor representing result of previous arithmetic instruction
- Used for many purposes: To determine if there has been overflow, whether the result is zero, etc.
- Stored in a special "EFLAGS" register within processor



Condition flags

- Bits maintained by the processor representing result of previous arithmetic instruction
- Used for many purposes: To determine if there has been overflow, whether the result is zero, etc.
- Stored in a special "EFLAGS" register within processor



Some x86 condition flags

Some x86 condition flags

- CF: Carry Flag
  - Set if the MSB of the arithmetic operation resulted in a carry bit being generated
  - Indicates an overflow when performing unsigned integer arithmetic

Some x86 condition flags

- CF: Carry Flag
  - Set if the MSB of the arithmetic operation resulted in a carry bit being generated
  - Indicates an overflow when performing unsigned integer arithmetic
- OF: Overflow flag
  - Set if result is too large or too small (negative) to fit in the destination
  - Indicates an overflow when performing signed integer arithmetic

Some x86 condition flags

- CF: Carry Flag
  - Set if the MSB of the arithmetic operation resulted in a carry bit being generated
  - Indicates an overflow when performing unsigned integer arithmetic
- OF: Overflow flag
  - Set if result is too large or too small (negative) to fit in the destination
  - Indicates an overflow when performing signed integer arithmetic
- SF: Sign flag
  - Set to MSB of result; which indicates the sign of a two's complement integer
  - 0 means result was positive, 1 means negative

Some x86 condition flags

- CF: Carry Flag
  - Set if the MSB of the arithmetic operation resulted in a carry bit being generated
  - Indicates an overflow when performing unsigned integer arithmetic
- OF: Overflow flag
  - Set if result is too large or too small (negative) to fit in the destination
  - Indicates an overflow when performing signed integer arithmetic
- SF: Sign flag
  - Set to MSB of result; which indicates the sign of a two's complement integer
  - 0 means result was positive, 1 means negative
- ZF: Zero flag
  - Set if the result of an arithmetic operation is zero; cleared otherwise

Some x86 condition flags

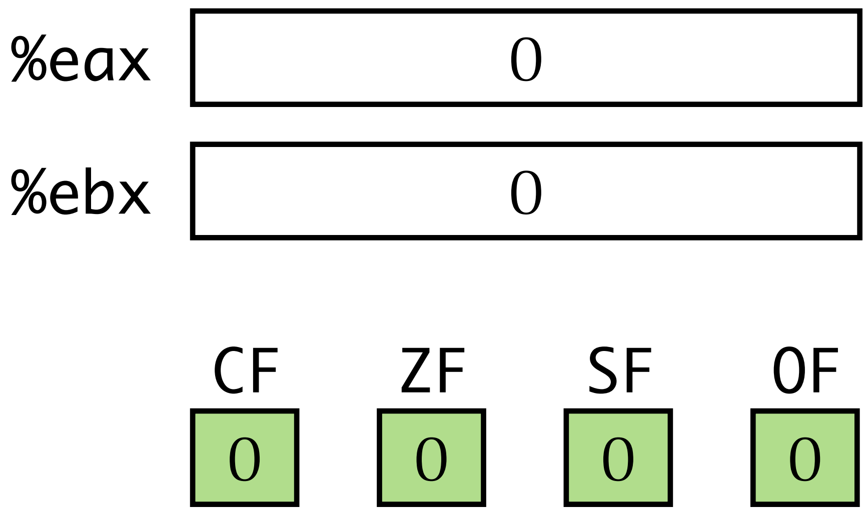
- CF: Carry Flag
  - Set if the MSB of the arithmetic operation resulted in a carry bit being generated
  - Indicates an overflow when performing unsigned integer arithmetic
- OF: Overflow flag
  - Set if result is too large or too small (negative) to fit in the destination
  - Indicates an overflow when performing signed integer arithmetic
- SF: Sign flag
  - Set to MSB of result; which indicates the sign of a two's complement integer
  - 0 means result was positive, 1 means negative
- ZF: Zero flag
  - Set if the result of an arithmetic operation is zero; cleared otherwise
- Condition flags are set **implicitly** by every arithmetic operation.
- Can also be set **explicitly** by comparison instructions.

Comparison instructions

- `cmpl` *src1, src2*
  - Compares value of *src1* and *src2*
  - *src1*, *src2* can be registers, immediate values, or contents of memory.
  - Computes (*src2* – *src1*) without modifying either operand
    - like "`subl` *src1, src2*" without changing *src2*
  - But, sets the condition flags based on the result of the subtraction.
- `testl` *src1, src2*
  - Like `cmpl`, but computes (*src1* & *src2*) instead of subtracting them.

Condition flags example

```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
```



Condition flags example

```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
```

%eax	42
%ebx	0

CF	ZF	SF	OF
0	0	0	0

Condition flags example

```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
```

%eax	42
%ebx	6

CF	ZF	SF	OF
0	0	0	0

Condition flags example

```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
```

%eax	42
%ebx	42

CF	ZF	SF	OF
0	0	0	0

Condition flags example

```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
```

%eax	42
%ebx	42

CF	ZF	SF	OF
0	1	0	0

- Recall: `cmpl %eax, %ebx` computes `%ebx` – `%eax` and sets the flags
  - `%ebx` is not modified by the `cmpl` instruction (unlike `subl`)
- Condition flags are set after every instruction!
  - See x86 manual or textbook for details of which flags are set by each instruction
  - In this example, the flags were only changed by the `cmpl` instruction

Another example

Another example

- Consider `cmpl %eax, %ebx`
  - computes %ebx – %eax
- Suppose `%ebx` is greater than `%eax`
- Which condition flags are set? ZF? OF? SF?

Another example

- Consider `cmpl %eax, %ebx`
  - computes %ebx – %eax
- Suppose `%ebx` is greater than `%eax`
- Which condition flags are set? ZF? OF? SF?
- Since %ebx > %eax result cannot be zero ⇒ ZF is not set

Another example

- Consider `cmpl %eax, %ebx`
  - computes %ebx – %eax
- Suppose `%ebx` is greater than `%eax`
- Which condition flags are set? ZF? OF? SF?
- Since %ebx > %eax result cannot be zero ⇒ ZF is not set
- Suppose no overflow occurs (i.e., OF is not set)

Another example

- Consider `cmpl %eax, %ebx`
  - computes %ebx – %eax
- Suppose `%ebx` is greater than `%eax`
- Which condition flags are set? ZF? OF? SF?
- Since %ebx > %eax result cannot be zero ⇒ ZF is not set
- Suppose no overflow occurs (i.e., OF is not set)
  - %ebx > %eax if and only if result is positive
  - ⇒ SF is not set (indicating positive)

Another example

- Consider `cmpl %eax, %ebx`
  - computes %ebx – %eax
- Suppose `%ebx` is greater than `%eax`
- Which condition flags are set? ZF? OF? SF?
- Since %ebx > %eax result cannot be zero ⇒ ZF is not set
- Suppose no overflow occurs (i.e., OF is not set)
  - %ebx > %eax if and only if result is positive
  - ⇒ SF is not set (indicating positive)
- Suppose overflow occurs (i.e., OF is set)

Another example

- Consider `cmpl %eax, %ebx`
  - computes %ebx – %eax
- Suppose `%ebx` is greater than `%eax`
- Which condition flags are set? ZF? OF? SF?
- Since %ebx > %eax result cannot be zero ⇒ ZF is not set
- Suppose no overflow occurs (i.e., OF is not set)
  - %ebx > %eax if and only if result is positive
  - ⇒ SF is not set (indicating positive)
- Suppose overflow occurs (i.e., OF is set)
  - %ebx > %eax if and only if result is negative
  - ⇒ SF is set (indicating negative)

Another example

Another example

- Consider `cmpl %eax, %ebx`
  - computes %ebx – %eax
- Suppose `%ebx` is greater than `%eax`
- Which condition flags are set? ZF? OF? SF?

Another example

- Consider `cmpl %eax, %ebx`
  - computes %ebx – %eax
- Suppose `%ebx` is greater than `%eax`
- Which condition flags are set? ZF? OF? SF?
- `%ebx` is greater than `%eax`

  if and only if ZF not set, and SF equal to OF

  if and only if ~(SF ^ OF) & ~ZF

Reading condition flags

- Operation `setX` *dest* sets single byte based on condition code

SetX	Synonyms	Condition	Description
`sete`	`setz`	dest = ZF	Equal/zero
`setne`	`setnz`	dest = ~ZF	Not equal/non-zero
`sets`		dest = SF	Negative
`setns`		dest = ~SF	Not negative
`setg`	`setnle`	dest = ~(SF ^ OF) & ~ZF	Greater than (signed >)
`setge`	`netnl`	dest = ~(SF ^ OF)	Greater than or equal (signed ≥)
`setl`	`setnge`	dest = SF ^ OF	Less than (signed <)
`setle`	`setng`	dest = (SF ^ OF) | ZF	Less than or equal (signed ≤)
`seta`	`setnbe`	dest = ~CF & ~ZF	Above (unsigned >)
`setb`	`setnae`	dest = CF	Below (unsigned <)

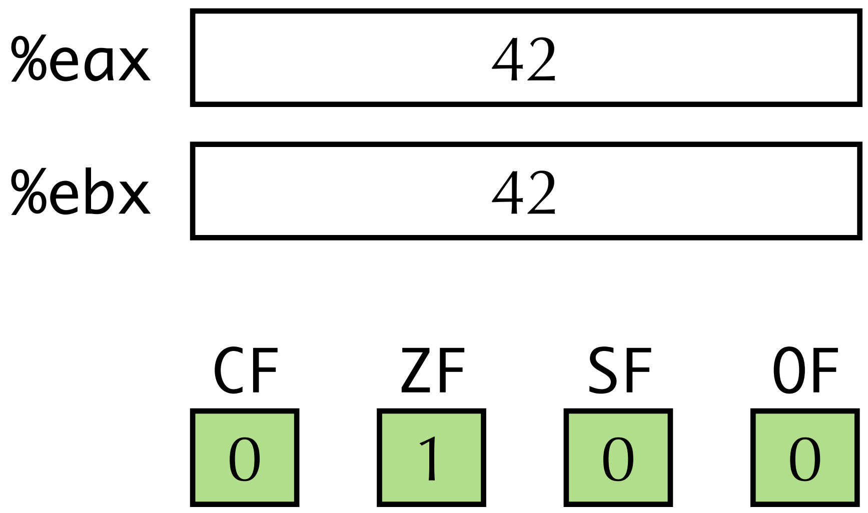
Conditional jumps

- Operation `jX` *label* jumps to label if condition *X* is satisfied

Instruction	Synonyms	Jump condition	Description
`jmp`		1	
`je`	`jz`	ZF	Equal/zero
`jne`	`jnz`	~ZF	Not equal/non-zero
`js`		SF	Negative
`jns`		~SF	Not negative
`jg`	`jnle`	~(SF ^ OF) & ~ZF	Greater than (signed >)
`jge`	`jnl`	~(SF ^ OF)	Greater than or equal (signed ≥)
`jl`	`jnge`	SF ^ OF	Less than (signed <)
`jle`	`jng`	(SF ^ OF) | ZF	Less than or equal (signed ≤)
`ja`	`jnbe`	~CF & ~ZF	Above (unsigned >)
`jb`	`jnae`	CF	Below (unsigned <)

Condition flags example

```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
```



Condition flags example

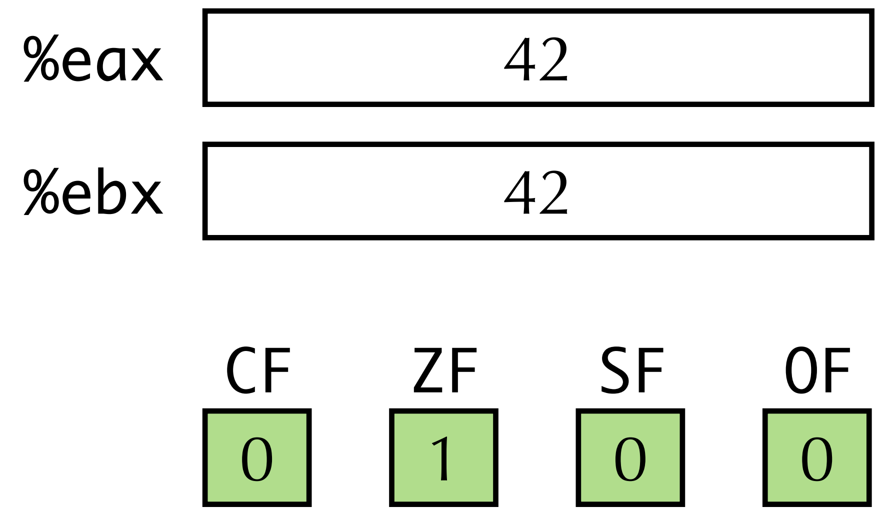
```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
jz   0x80459845
movl $33, %eax
…
```

%eax	42
%ebx	42

CF	ZF	SF	OF
0	1	0	0

Condition flags example

```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
jz   0x80459845
movl $33, %eax
...
```



- Instruction `jz 0x80459845` will set instruction pointer to `0x80459845` if ZF is set.
- Otherwise, execution continues after the instruction
  - i.e., with `movl $33, %eax`

More examples

- What do the following examples do?

More examples

- What do the following examples do?

```
movl $0xffffffff, %eax
addl $0x1, %eax
jz 0x08045900
...
```

More examples

- What do the following examples do?

```
movl $0xffffffff, %eax
addl $0x1, %eax
jz 0x08045900
...
```

Jump if -1 + 1 is zero

More examples

- What do the following examples do?

```
movl $0xffffffff, %eax
addl $0x1, %eax
jz 0x08045900
...
```

Jump if -1 + 1 is zero

```
movl $6, %eax
subl $10, %eax
jl 0x08045900
...
```

More examples

- What do the following examples do?

```
movl $0xffffffff, %eax
addl $0x1, %eax
jz 0x08045900
...
```

Jump if -1 + 1 is zero

```
movl $6, %eax
subl $10, %eax
jl 0x08045900
...
```

Jump if 6 is less than 10

More examples

- What do the following examples do?

```
movl $0xffffffff, %eax
addl $0x1, %eax
jz 0x08045900
...
```

Jump if -1 + 1 is zero

```
movl $6, %eax
subl $10, %eax
jl 0x08045900
...
```

Jump if 6 is less than 10

Given `cmpl` *src*, *dest*, what is the relationship of *dest* to *src*?

More examples

- What do the following examples do?

```
movl $0xffffffff, %eax
addl $0x1, %eax
jz 0x08045900
...
```

Jump if -1 + 1 is zero

```
movl $6, %eax
subl $10, %eax
jl 0x08045900
...
```

Jump if 6 is less than 10

Given `cmpl` *src*, *dest*, what is the relationship of *dest* to *src*?

```
movl $0x42, %eax
movl $0x77, %ebx
subl %ebx, %eax
js 0x08045900
...
```

More examples

- What do the following examples do?

```
movl $0xffffffff, %eax
addl $0x1, %eax
jz 0x08045900
...
```

Jump if -1 + 1 is zero

```
movl $6, %eax
subl $10, %eax
jl 0x08045900
...
```

Jump if 6 is less than 10

Given `cmpl` *src*, *dest*, what is the relationship of *dest* to *src*?

```
movl $0x42, %eax
movl $0x77, %ebx
subl %ebx, %eax
js 0x08045900
...
```

Jump if 0x42 – 0x77 is negative

Example: absdiff

```c
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Set up

Body 1

Finish

Body 2

Example: absdiff

```c
int absdiff_goto(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
  Exit:
    return result;
  Else:
    result = y-x;
    goto Exit;
}
```

- C allows “goto” as a control flow mechanism
  - Closer to machine-level programming
  - Generally considered bad programming style

```
absdiff:
    pushl   %ebp                    Set up
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx              Body 1
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave                           Finish
    ret
.L7:
    subl    %edx, %eax              Body 2
    jmp     .L8
```

Example: absdiff

```
int absdiff_goto(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
  Exit:
    return result;
  Else:
    result = y-x;
    goto Exit;
}
```

- C allows "goto" as a control flow mechanism
  - Closer to machine-level programming
  - Generally considered bad programming style

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Example: absdiff

```c
int absdiff_goto(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
  Exit:
    return result;
  Else:
    result = y-x;
    goto Exit;
}
```

- C allows “goto” as a control flow mechanism
  - Closer to machine-level programming
  - Generally considered bad programming style

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

%edx = x
%eax = y

Example: absdiff

```
int absdiff_goto(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
  Exit:
    return result;
  Else:
    result = y-x;
    goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

%edx = x
%eax = y

- C allows “goto” as a control flow mechanism
  - Closer to machine-level programming
  - Generally considered bad programming style

Example: absdiff

```c
int absdiff_goto(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
  Exit:
    return result;
  Else:
    result = y-x;
    goto Exit;
}
```

- C allows "goto" as a control flow mechanism
  - Closer to machine-level programming
  - Generally considered bad programming style

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

%edx = x
%eax = y

Example: absdiff

```c
int absdiff_goto(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
  Exit:
    return result;
  Else:
    result = y-x;
    goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

%edx = x
%eax = y

- C allows "goto" as a control flow mechanism
  - Closer to machine-level programming
  - Generally considered bad programming style

Example: absdiff

```
int absdiff_goto(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
  Exit:
    return result;
  Else:
    result = y-x;
    goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

%edx = x
%eax = y

- C allows "goto" as a control flow mechanism
  - Closer to machine-level programming
  - Generally considered bad programming style

Next lecture

- Loops
- Switch statements
- Procedures

- Reminder!
  - Lab 1 released!
  - Sign up for sections by Friday 5pm
  - See website for more details