Review

Intel Assembly Language

Objective: *simple reading knowledge* so that

→ examples in B&O are intelligible

→ you can see what's going on in your code

Topics:

- Instruction format: # of operands, typing, "direction"
- Data types
- Registers
- **Move** instructions – addressing modes, **lea**
- Flow control – condition codes, **cmp, test, jxx**

Assembly Language Formats

For those of you familiar with MIPS and other RISC assembly languages, Intel AL may be a little jarring (see next slide).

But first: In assembly language in general there are possible two formats:

dest. on the left → "Intel" *- also what we used for MIPS except SW*

ADD DestReg, Src1Reg, Src2Reg

dest. on the right → "AT&T" – used by B&O

ADD Src1Reg, Src2Reg, DestReg

The textbook uses AT&T, which is also the default for the gcc compiler. For the icc compiler, however, the default is Intel (surprise!). To force one or the other, use the following (along with the –S switch to generate the AL code file).

```
gcc -S -masm=att code.c
gcc -S -masm=intel code.c
```

Intel ≠ MIPS (or other RISC AL)

Some things to watch out for: ***>> You should know and understand these points***

begin every bullet point with "unlike MIPS" like so:

- (unlike MIPS) Direction of the data flow in the instructions: left-to-right
- … in Intel Registers have funny names: RAX, ESI, etc. In the 64-bit version most of the historic distinctions among these registers disappear.
- … you can use pieces of a register: in an 8 byte register, you can use 1, 2, 4, or 8 bytes (depending on the register, different of these are available, but 4 is always an option).
- … there are many data types. In Intel format these are shown in the opcode (l, q, etc.): **addl** means "add" on a "long" (4 bytes). **addq** means "add" on a "quad" (8 bytes).
- … instructions have at most two operands (not three). One doubles as a source and dest.
- … single instructions can both access memory *and* do arithmetic, a RISC no-no. In fact, these *complex* instructions actually get split into multiple instructions within the CPU, but that is not programmer visible. *But we will be interpolating this CPU-level instr. set.*
- … instructions generate implicit *conditions* as side-effects that are recorded in a *condition code register*. In MIPS a general purpose register is used explicitly (recall SLT).
- … there are complex addressing modes, rather than only *displacement*. And there are real complex instructions: **push, pop, lea, leave,** etc. Again, in both cases, these get broken into MIPS-like instructions within the CPU, but they are still real parts of the AL.

Intel’s 64-Bit Architecture

- **Intel Attempted Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- **AMD Stepped in with Evolutionary Solution**
  - x86-64 (now called “AMD64”)
- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- **All but low-end x86 processors support x86-64**
  - But, lots of code still runs in 32-bit mode

Data Representations: IA32 + x86-64

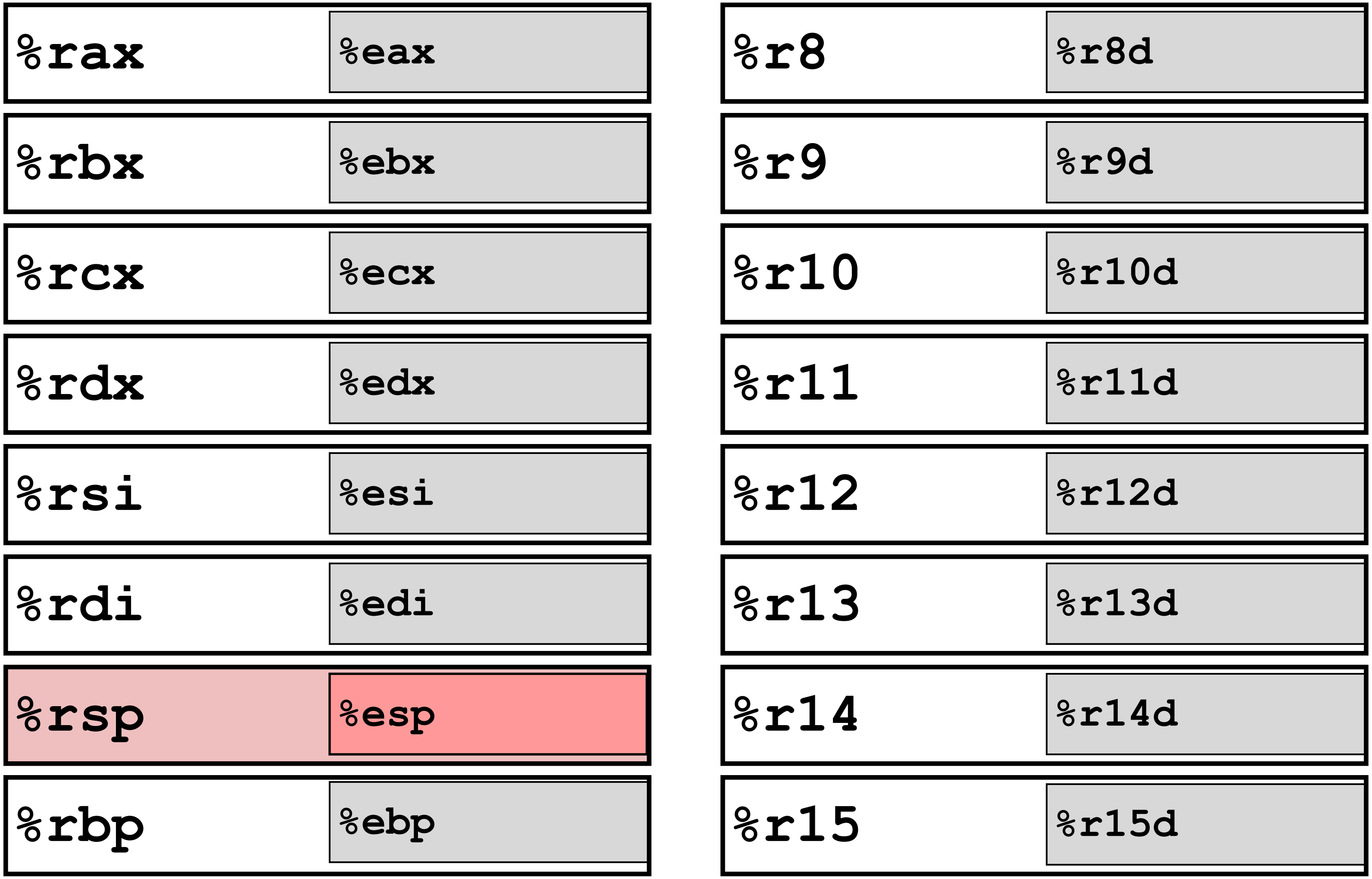
- **Sizes of C Objects (in Bytes)**

C Data Type	*Generic 32-bit*	*Intel IA32*	*x86-64*
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	16
char *	4	4	8

– *Or any other pointer*

Not the Assembly Language Data Type – “l” = 4 “q” = 8

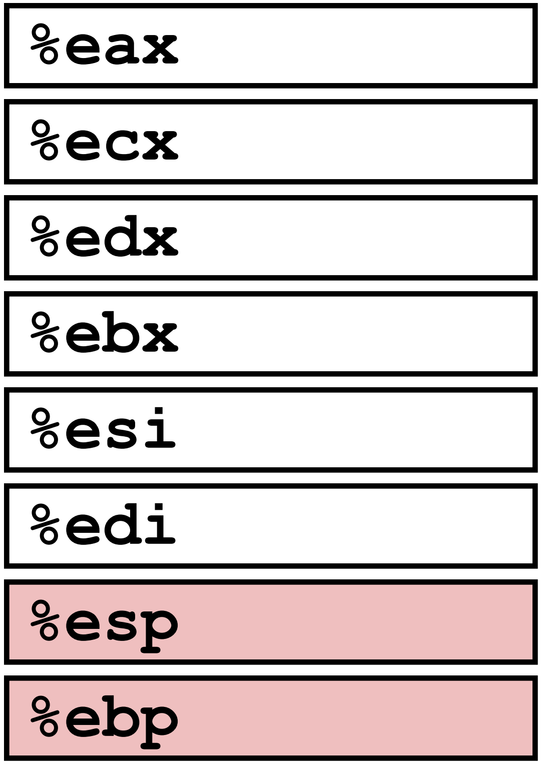
x86-64 Integer Registers



%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

- Extend existing registers in IA32 (on left). Add 8 new ones (on right).
- Make **`%ebp/%rbp`** general purpose

Moving Data: IA32



- **Moving Data**

  `movl` ***Source*, *Dest*:** *In IA32 “l” means 4 bytes*

- **Operand Types**
  - ***Immediate:*** Constant integer data
    - Example: `$0x400`, `$-533`
    - Like C constant, but prefixed with `‘$’`
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 8 integer registers
    - Example: `%eax`, `%edx`
    - But `%esp` and `%ebp` reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 4 consecutive bytes of memory at address given by register
    - Simplest example: `(%eax)`
    - Various other “address modes”

`movl` Operand Combinations

	Source	Dest	Src, Dest	C Analog
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- **Normal** (R) Mem[Reg[R]]
  - Register R specifies memory address

```
movl (%ecx),%eax     # %ecx has address
```

- **Displacement** D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movl 8(%ebp),%edx    # address = 8+%ebp
```

These should look familiar to MIPS programmers, except for the direction (l→r) and "l" for size

Using Simple Addressing Modes

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

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

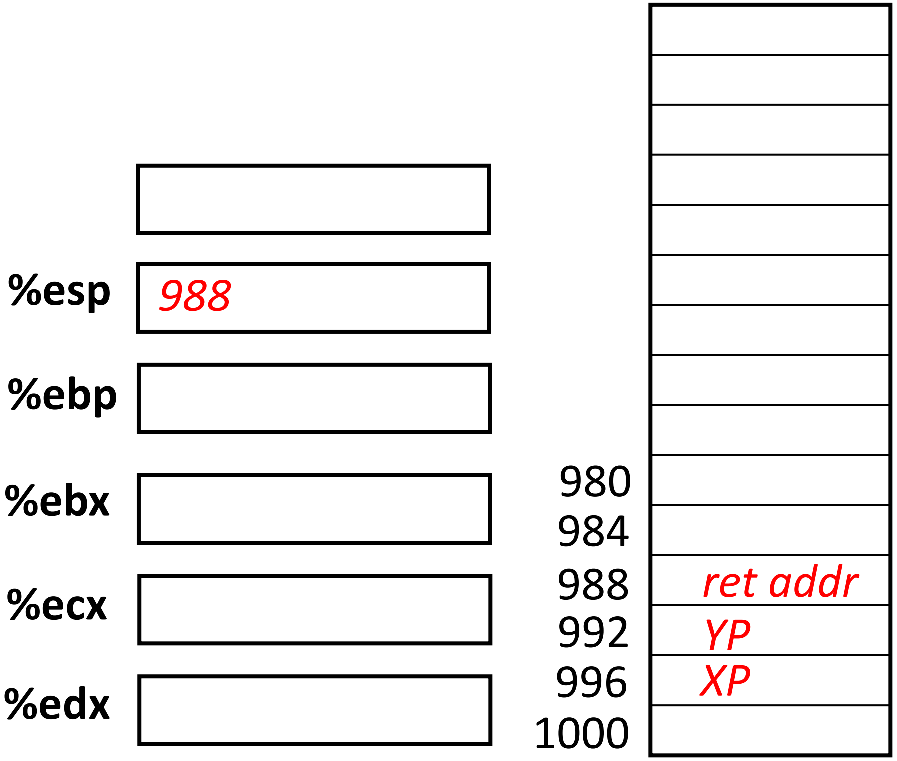
Set Up

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

Body

```
   popl   %ebx
   popl   %ebp
   ret
```

Finish



Complete Memory Addressing Modes

Most General Form	**What it means**
D(Rb,Ri,S)	**Mem[Reg[Rb] + S*Reg[Ri] + D]**

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 8 integer registers
- Ri: Index register: Any, except for `%esp`
  - Unlikely you’d use `%ebp`, either
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**

(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]]**
D(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]+D]**
(Rb,Ri,S)	**Mem[Reg[Rb]+S*Reg[Ri]]**

64-bit code for swap (on 32-bit data)

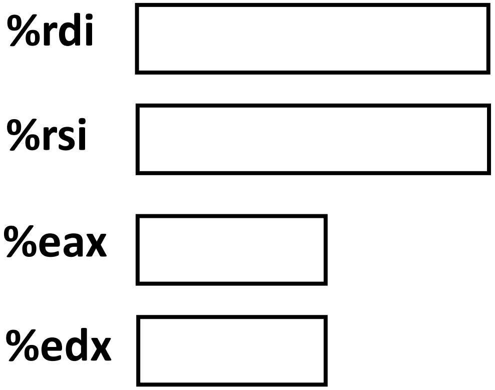
```c
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
                                      Set Up

    movl    (%rdi), %edx
    movl    (%rsi), %eax              Body
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)

    ret                               Finish
```

- **Operands passed in registers (why useful?)**
  - First op **`xp`** in **`%rdi`**, second op **`yp`** in **`%rsi`**
  - Both are 64-bit pointers
- **No stack operations required**
- **32-bit data**
  - Data held in registers **`%eax`** and **`%edx`**
  - **`movl`** operation



64-bit code for long int swap (on 64-bit data)

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

```
swap_l:

   movq     (%rdi), %rdx
   movq     (%rsi), %rax
   movq     %rax, (%rdi)
   movq     %rdx, (%rsi)

   ret
```

Set Up

Body

Finish

- **64-bit data**
  - Data held in registers `%rax` and `%rdx`
  - `movq` operation
    - “q” stands for quad-word

Address Computation Instruction

- `leal` *Src, Dest*   *# load effective address long*
  - *Src* is address mode expression
  - Set *Dest* to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- **Example**

```
int mul12(int x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leal (%eax,%eax,2), %eax   ;t <- x+x*2
sall $2, %eax              ;return t<<2
```

Some Arithmetic Operations

- **Two Operand Instructions:**

Format	*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	

- **Watch out for argument order!**
- **No distinction between signed and unsigned int (why?)**
- **One Operand Instructions**

`incl`	*Dest*	Dest = Dest + 1	*# No eq in MIPS*
`decl`	*Dest*	Dest = Dest – 1	
`negl`	*Dest*	Dest = – Dest	
`notl`	*Dest*	Dest = ~Dest	

Arithmetic Expression Example

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

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

Set Up

```
  movl   8(%ebp), %ecx
  movl   12(%ebp), %edx
  leal   (%edx,%edx,2), %eax
  sall   $4, %eax
  leal   4(%ecx,%eax), %eax
  addl   %ecx, %edx
  addl   16(%ebp), %edx
  imull  %edx, %eax
```

Body

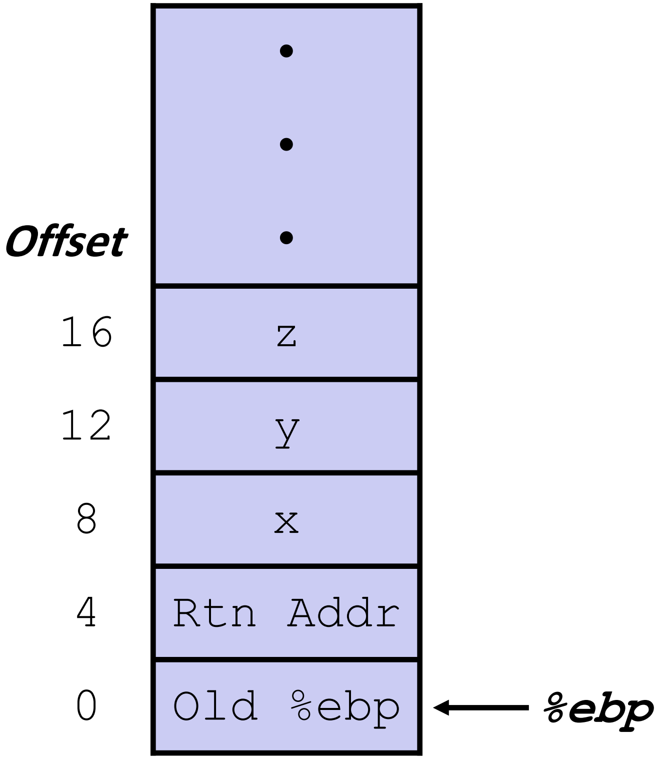
```
  popl   %ebp
  ret
```

Finish

%ecx ← X	
%edx ← Y	
%eax ← 3 * Y	
%eax ← 16 * %eax	(t4)
%eax ← 4 + %ecx + %eax	(t5)
%edx ← %ecx + %edx	(t1)
%edx ← t1 + z	(t2)
%eax ← t2 * t5	(t5 = rval)

Understanding `arith`

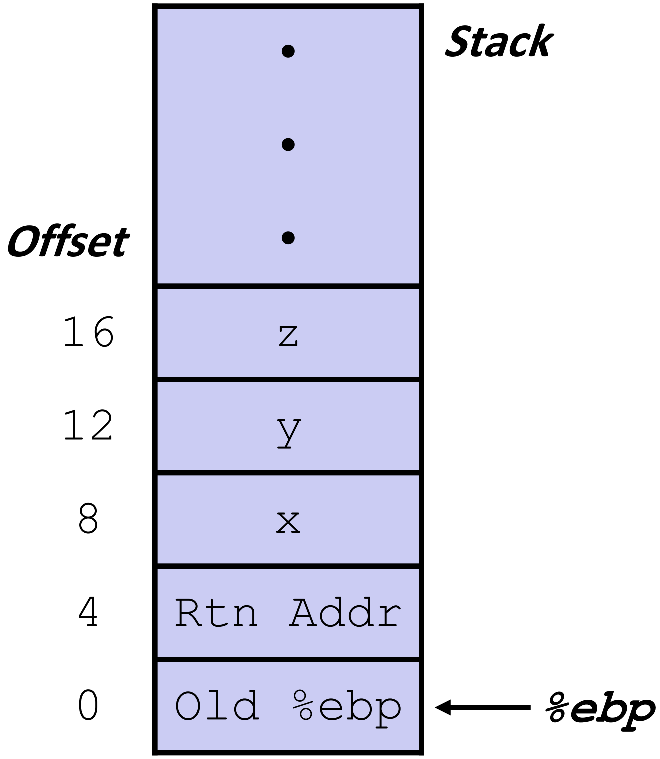
```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```



```
movl    8(%ebp), %ecx         %ecx ← X
movl    12(%ebp), %edx        %edx ← Y
leal    (%edx,%edx,2), %eax   %eax ← 3 * Y
sall    $4, %eax              %eax ← 16 * %eax               (t4)
leal    4(%ecx,%eax), %eax    %eax ← 4 + %ecx + %eax         (t5)
addl    %ecx, %edx            %edx ← %ecx + %edx             (t1)
addl    16(%ebp), %edx        %edx ← t1 + z                  (t2)
imull   %edx, %eax            %eax ← t2 * t5                 (t5 = rval)
```

Understanding arith

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```



Stack

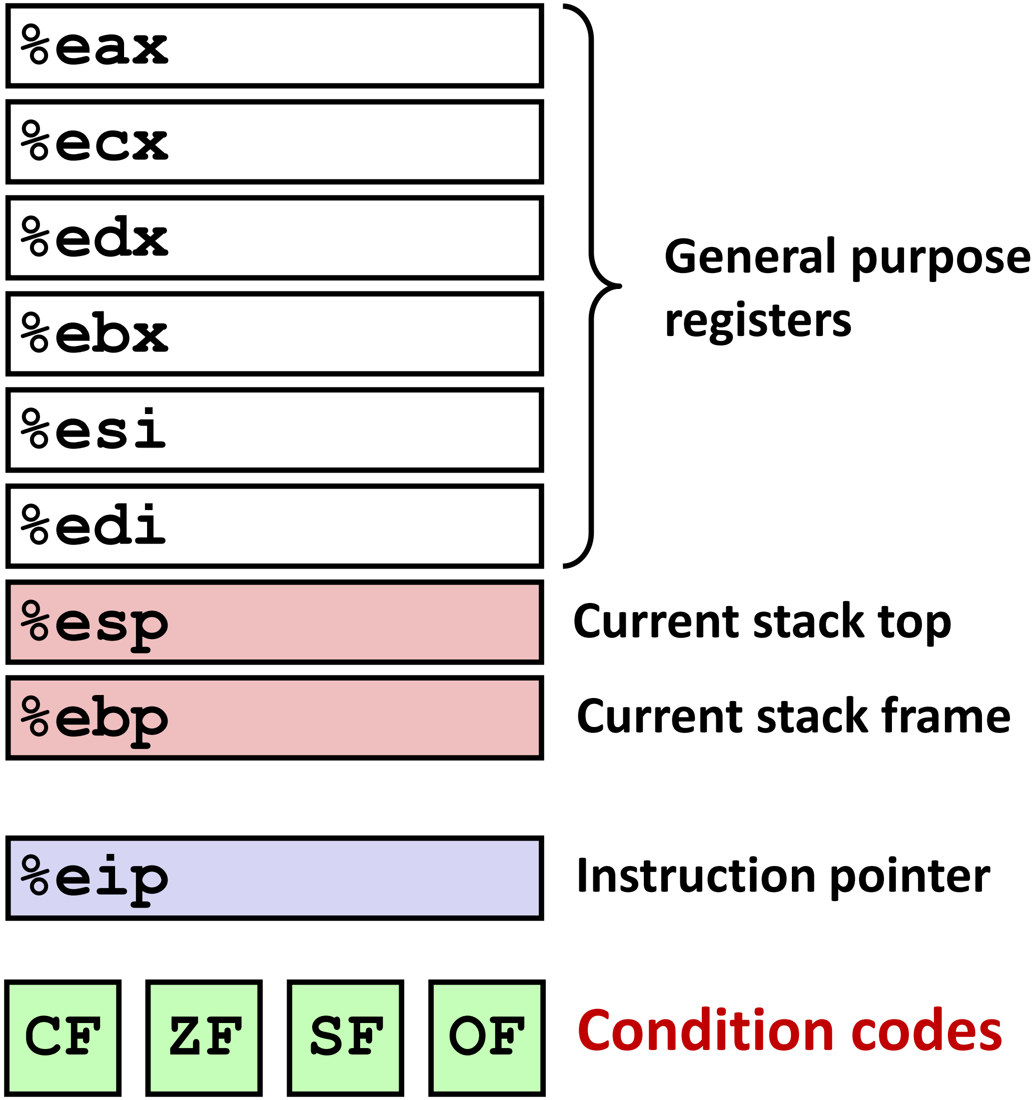
Offset

Offset	Stack
	•
	•
	•
16	z
12	y
8	x
4	Rtn Addr
0	Old %ebp ← %ebp

```
movl   8(%ebp), %ecx          # ecx = x
movl   12(%ebp), %edx         # edx = y
leal   (%edx,%edx,2), %eax    # eax = y*3
sall   $4, %eax               # eax *= 16       (t4)
leal   4(%ecx,%eax), %eax     # eax = t4 +x+4   (t5)
addl   %ecx, %edx             # edx = x+y       (t1)
addl   16(%ebp), %edx         # edx += z        (t2)
imull  %edx, %eax             # eax = t2 * t5   (rval)
```

Processor State (IA32, Partial)

- **Information about currently executing program**
  - Temporary data (**`%eax`**, …)
  - Location of runtime stack (**`%ebp`**,**`%esp`**)
  - Location of current code control point (**`%eip`**, …)
  - Status of recent tests (**CF, ZF, SF, OF**)



Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
  - **ZF** Zero Flag **OF** Overflow Flag (for signed)
- **Implicitly set (think of it as *side effect*) by arithmetic operations**

  Example: `addl/addq` *Src,Dest* ↔ `t = a+b`

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if `t == 0`

  SF set if `t < 0` (as signed)

  OF set if two's-complement (signed) overflow
  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- **Not set by `lea` instruction**

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - **`cmpl/cmpq`** *Src2*, *Src1*
  - **`cmpl b,a`** like computing **`a-b`** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction – used for *logicals***
  - `testl/testq` *Src2*, *Src1*

    `testl b,a`   # like computing `a&b` without setting destination

  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask

  - **ZF set** when `a&b == 0`
  - **SF set** when `a&b < 0`

Conditional Branch Example

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

```
absdiff:
    pushl   %ebp                   Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6                    Body1
    subl    %eax, %edx
    movl    %edx, %eax             Body2a
    jmp .L7
.L6:
    subl %edx, %eax                Body2b
.L7:
    popl %ebp
    ret                            Finish
```

Generally do not worry about CCs – used implicitly

Conditional Branch Example (Cont.)

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

```
absdiff:
   pushl   %ebp
   movl    %esp, %ebp          Setup
   movl    8(%ebp), %edx
   movl    12(%ebp), %eax
   cmpl    %eax, %edx
   jle     .L6                 Body1
   subl    %eax, %edx
   movl    %edx, %eax          Body2a
   jmp .L7
.L6:
   subl %edx, %eax             Body2b
.L7:
   popl %ebp
   ret                         Finish
```

- **C allows "goto" as means of transferring control**
  - Closer to machine-level programming style
- **Generally considered bad coding style**

Conditional Branch Example (Cont.)

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

```
absdiff:
   pushl   %ebp                      Setup
   movl    %esp, %ebp
   movl    8(%ebp), %edx
   movl    12(%ebp), %eax
   cmpl    %eax, %edx
   jle     .L6                       Body1
   subl    %eax, %edx
   movl    %edx, %eax                Body2a
   jmp .L7
.L6:
   subl %edx, %eax                   Body2b
.L7:
   popl %ebp                         Finish
   ret
```

Conditional Branch Example (Cont.)

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

```
absdiff:
   pushl   %ebp
   movl    %esp, %ebp              Setup
   movl    8(%ebp), %edx
   movl    12(%ebp), %eax
   cmpl    %eax, %edx
   jle     .L6                     Body1
   subl    %eax, %edx
   movl    %edx, %eax              Body2a
   jmp .L7
.L6:
   subl %edx, %eax                 Body2b
.L7:
   popl %ebp
   ret                             Finish
```

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp                    Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6                     Body1
    subl    %eax, %edx
    movl    %edx, %eax              Body2a
    jmp .L7
.L6:
    subl %edx, %eax                 Body2b
.L7:
    popl %ebp
    ret                             Finish
```

General Conditional Expression Translation

C Code

```
val = Test ? Then_Expr : Else_Expr;    # C Conditional
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
   nt = !Test;
   if (nt) goto Else;
   val = Then_Expr;
   goto Done;
Else:
   val = Else_Expr;
Done:
   . . .
```

- Test is expression returning integer
    - = 0 interpreted as false
    - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves in AL

Conditional Move Instructions

- Instruction supports:

  if (Test) Dest ← Src
- Supported in post-1995 x86 processors
- GCC does not always use them
  - Wants to preserve compatibility with ancient processors
  - Enabled for x86-64
  - Use switch `-march=686` for IA32

Why?

- **Branches are very disruptive to instruction flow through pipelines**
- **Conditional moves do not require control transfer**

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
  tval = Then_Expr;
  result = Else_Expr;
  t = Test;
  if (t) result = tval;
  return result;
```

Conditional Move Example: x86-64

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

Start w/
x in %edi
y in %esi

```
absdiff:
  movl   %edi, %edx    # copy x into %edx
  subl   %esi, %edx    # tval = x-y
  movl   %esi, %eax    # copy y into %eax
  subl   %edi, %eax    # result = y-x (speculate)
  cmpl   %esi, %edi    # Compare x:y
  cmovg  %edx, %eax     # If >, result = tval
  ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed - so
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed - so
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed - so
- Must be side-effect free