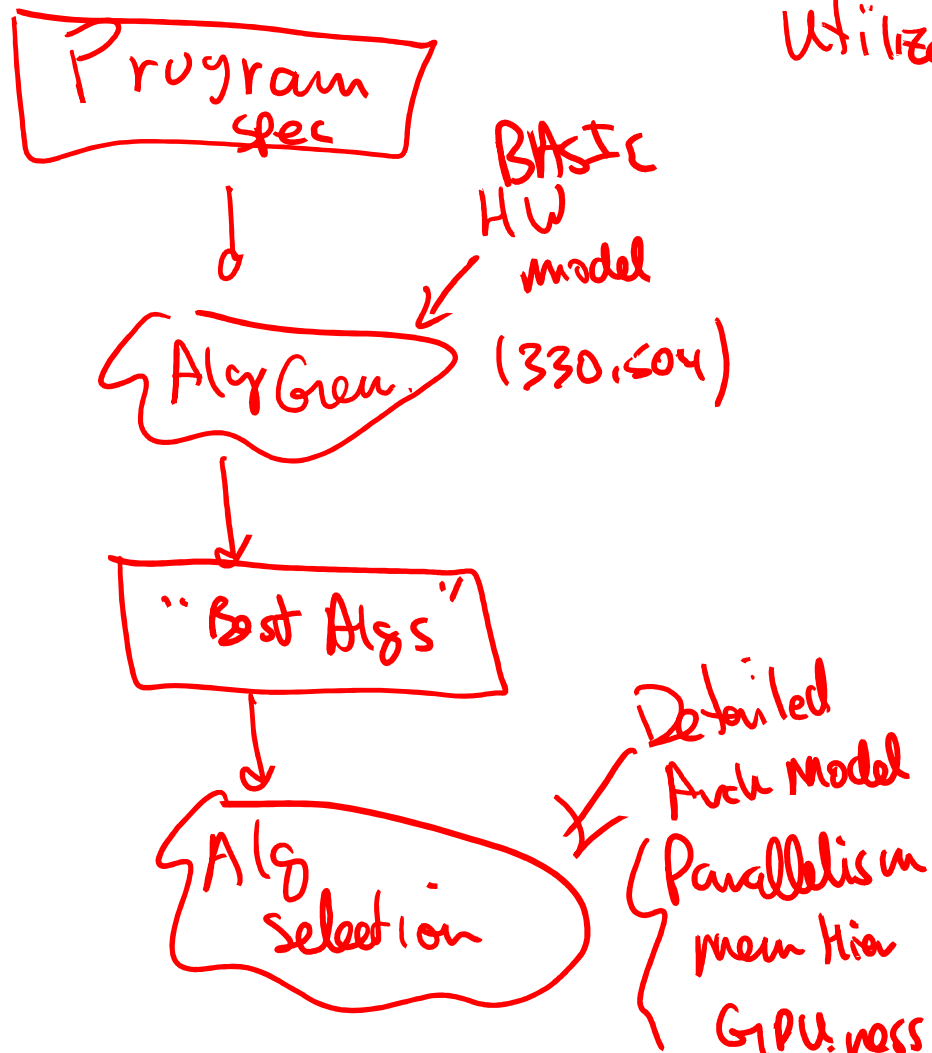
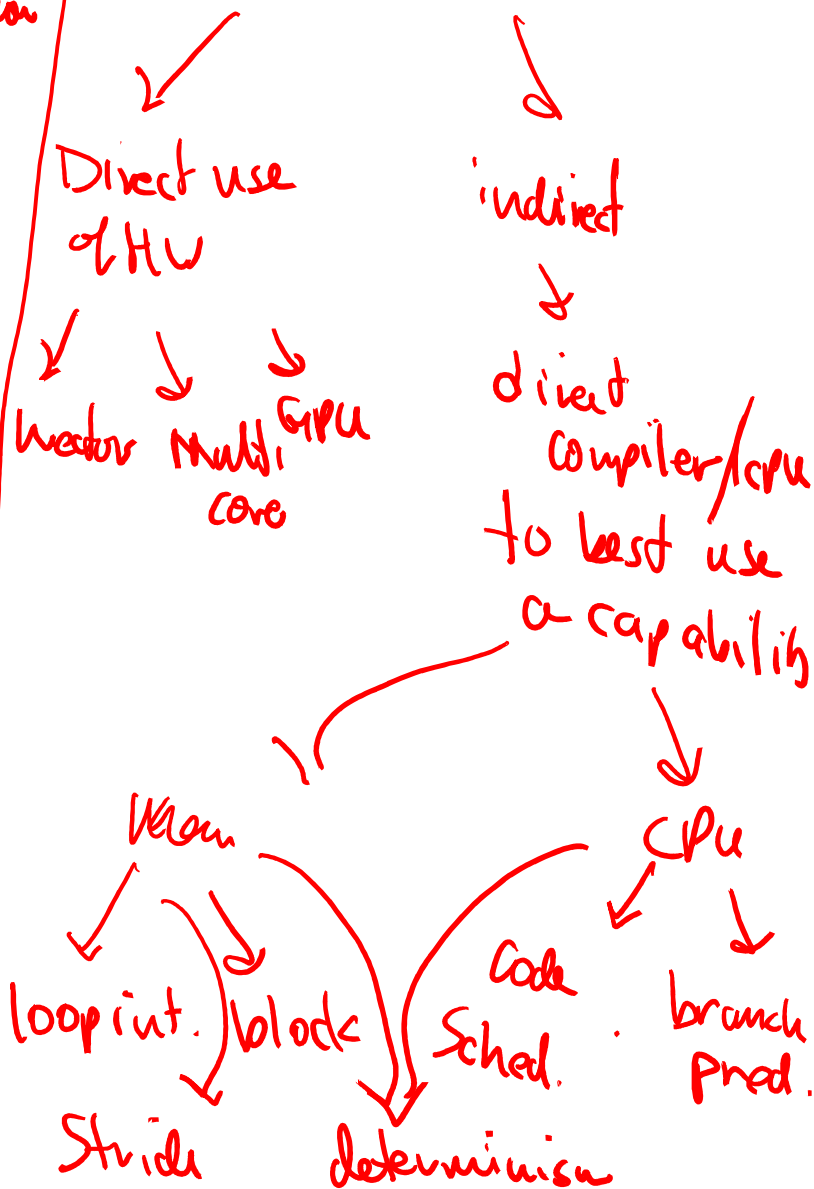


# Review

Overall Good = 100% Utilization



## Optimization



# 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 – *For Intel!*

*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: C to x86-64

## ■ Sizes of C Objects (in Bytes)

■ <i>C Data Type</i>	<i>Generic 32-bit</i>	<i>Intel IA32</i>	<i>x86-64</i>
▪ 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

64 bit

<b>%rax</b>	<b>%eax</b>	<b>AH</b>	<b>AL</b>
<b>%rbx</b>	<b>%ebx</b>		
<b>%rcx</b>	<b>%ecx</b>		
<b>%rdx</b>	<b>%edx</b>		
<b>%rsi</b>	<b>%esi</b>		
<b>%rdi</b>	<b>%edi</b>		
<b>%rsp</b>	<b>%esp</b>		
<b>%rbp</b>	<b>%ebp</b>		

<b>%r8</b>	<b>%r8d</b>
<b>%r9</b>	<b>%r9d</b>
<b>%r10</b>	<b>%r10d</b>
<b>%r11</b>	<b>%r11d</b>
<b>%r12</b>	<b>%r12d</b>
<b>%r13</b>	<b>%r13d</b>
<b>%r14</b>	<b>%r14d</b>
<b>%r15</b>	<b>%r15d</b>

- 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”

`%eax`

`%ecx`

`%edx`

`%ebx`

`%esi`

`%edi`

`%esp`

`%ebp`



# 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, <u>(%edx)</u>	*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 ✓
```

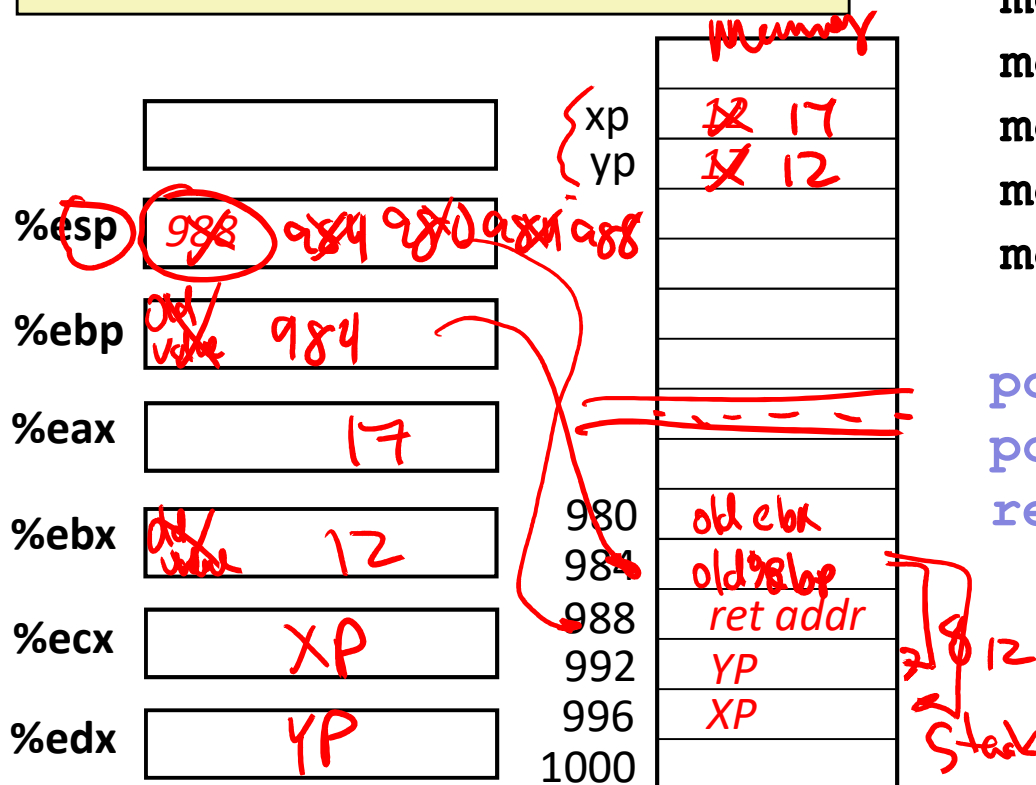
Set Up

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

Body

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

Finish



# Complete Memory Addressing Modes

## Most General Form

**D(Rb,Ri,S)**

- 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?*)

## What it means

**Mem[Reg[Rb] + S\*Reg[Ri] + D]**

*Rb Ri D S*  
*{ R7, R8, 64, 8 }*

*( R7 + 8 \* R8 + 64 )*

## ■ 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)

swap:

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

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

ret

} Set Up  
} Body  
} Finish

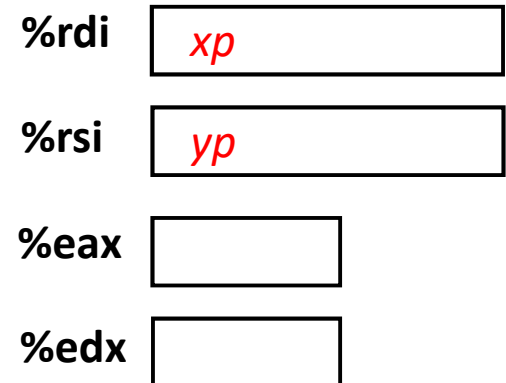
## ■ Operands passed in registers (why useful?)

- 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\_1:

```
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  $x + k*y$ 
  - $k = 1, 2, 4, \text{ or } 8$

## ■ 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:

<i>Format</i>	<i>Computation</i>	
<code>addl</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} + \text{Src}$
<code>subl</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} - \text{Src}$
<code>imull</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} * \text{Src}$
<code>sall</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \ll \text{Src}$ <i>Also called shll</i>
<code>sarl</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \gg \text{Src}$ <i>Arithmetic</i>
<code>shrl</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \gg \text{Src}$ <i>Logical</i>
<code>xorl</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \wedge \text{Src}$
<code>andl</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest} \& \text{Src}$
<code>orl</code>	<i>Src, Dest</i>	$\text{Dest} = \text{Dest}   \text{Src}$

## ■ Watch out for argument order!

## ■ No distinction between signed and unsigned int (why?)

## ■ One Operand Instructions

<code>incl</code>	<i>Dest</i>	$\text{Dest} = \text{Dest} + 1$	<i># No eq in MIPS</i>
<code>decl</code>	<i>Dest</i>	$\text{Dest} = \text{Dest} - 1$	
<code>negl</code>	<i>Dest</i>	$\text{Dest} = -\text{Dest}$	
<code>notl</code>	<i>Dest</i>	$\text{Dest} = \sim \text{Dest}$	



# 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

```

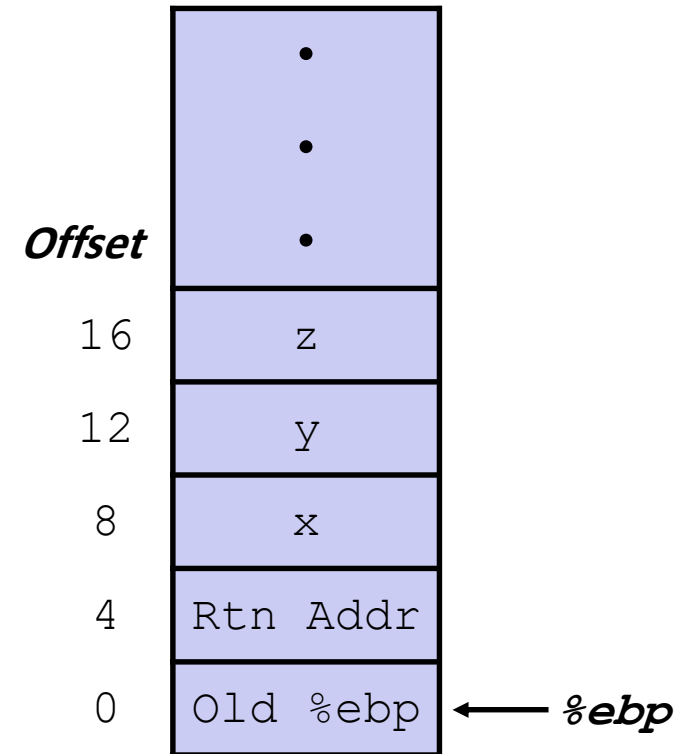
$\%ecx \leftarrow X$   
 $\%edx \leftarrow Y$   
 $\%eax \leftarrow 3 * Y$   
 $\%eax \leftarrow 16 * \%eax$  (t4)  
 $\%eax \leftarrow 4 + \%ecx + \%eax$  (t5)  
 $\%edx \leftarrow \%ecx + \%edx$  (t1)  
 $\%edx \leftarrow t1 + z$  (t2)  
 $\%eax \leftarrow t2 * t5$  (t5 = rval)

# 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;
}
```

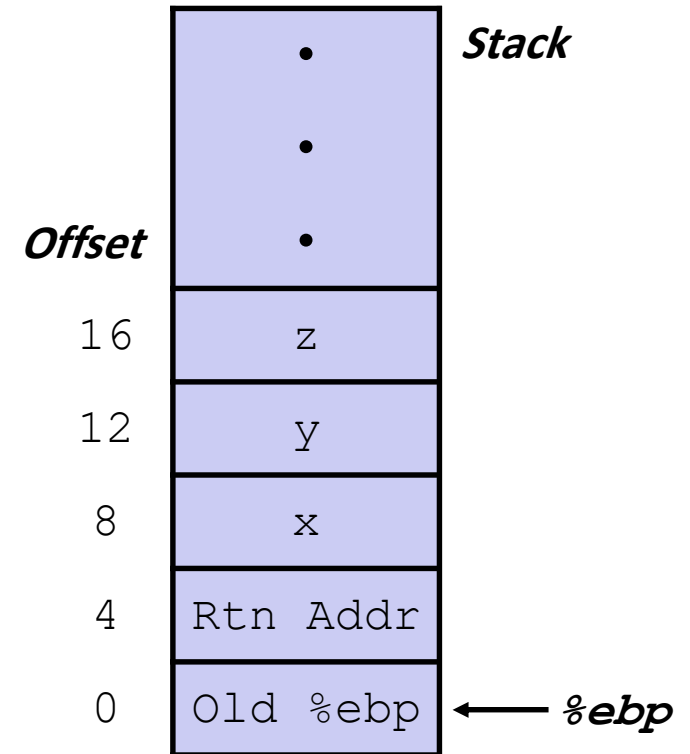
```
movl    8(%ebp), %ecx
movl    12(%ebp), %edx
leal    (%edx,%edx,2), %eax
sall    $4, %eax
leal    4(%ecx,%eax), %eax
addl    %ecx, %edx
addl    16(%ebp), %edx
imull   %edx, %eax
```

```
%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;
}
```

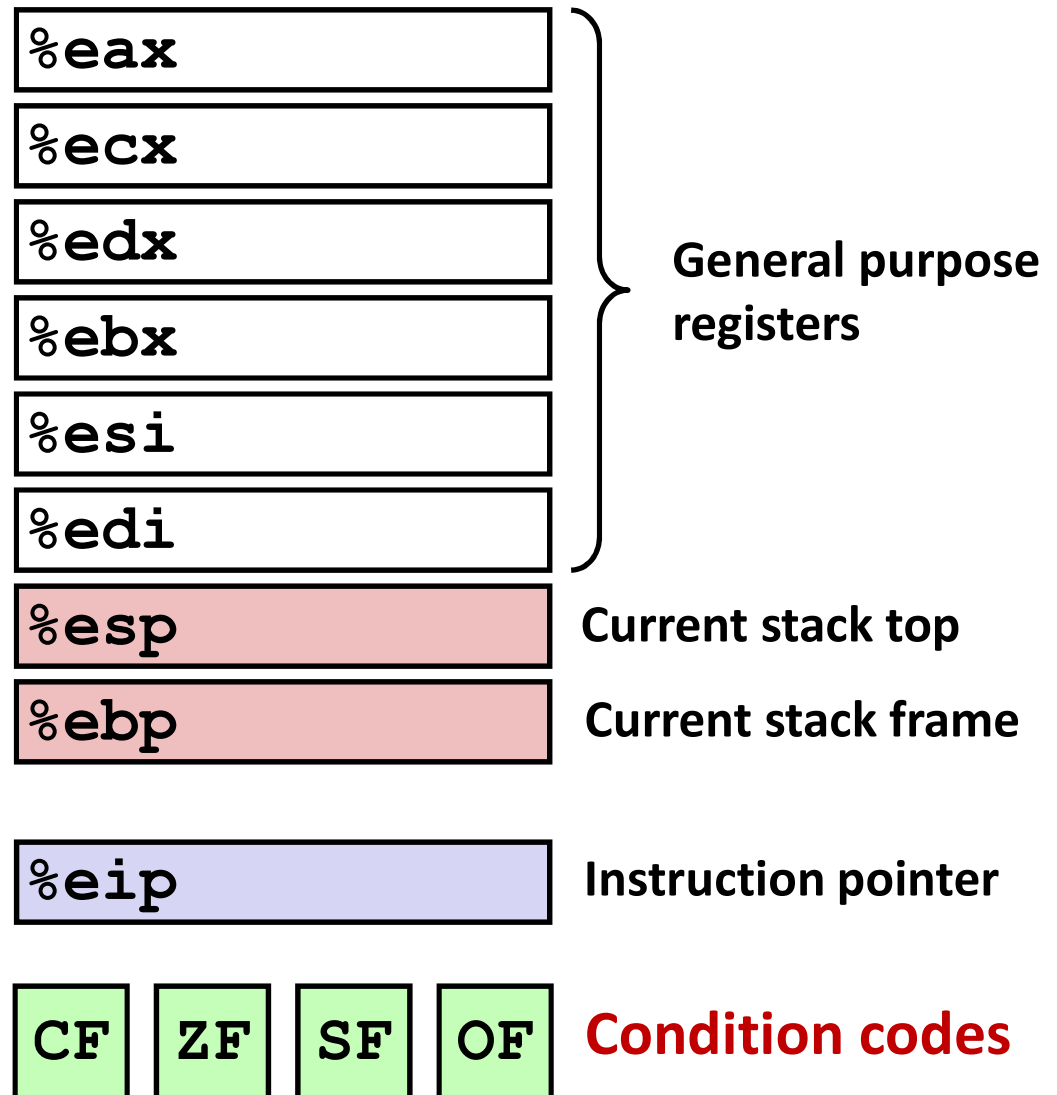


movl	8(%ebp), %ecx	# ecx = x
movl	12(%ebp), %edx	# edx = y
leal	(%edx,%edx,2), %eax	# eax = y*3
sall	\$4, %eax	# eax *= 16 (t4)
leal	4(%ecx,%eax), %eax	# eax = t4 + x + 4 (t5)
addl	%ecx, %edx	# edx = x + y (t1)
addl	16(%ebp), %edx	# edx += z (t2)
imull	%edx, %eax	# eax = t2 * t5 (rval)

# 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`  $\leftrightarrow$  `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 `leal` 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
    movl     %esp, %ebp
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    cmpl     %eax, %edx
    jle      .L6
    subl     %eax, %edx
    movl     %edx, %eax
    jmp      .L7
.L6:
    subl     %edx, %eax
.L7:
    popl     %ebp
    ret
```

Setup

Body1

Body2a

Body2b

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;
}
```

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

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

Diagram illustrating the assembly code structure with labels and control flow:

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

Diagram illustrating the assembly code structure with labels and branches:

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

Setup

Body1

Body2a

Body2b

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
    movl     %esp, %ebp
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    cmpl     %eax, %edx
    jle      .L6
    subl     %eax, %edx
    movl     %edx, %eax
    jmp      .L7
.L6:
    subl     %edx, %eax
.L7:
    popl     %ebp
    ret
```

Setup

Body1

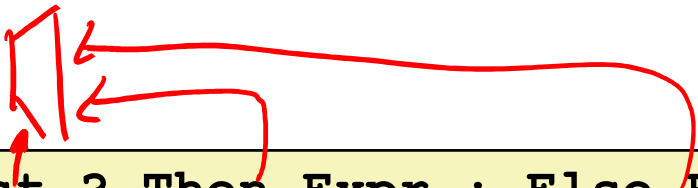
Body2a

Body2b

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  $\leftarrow$  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;  
}
```

absdiff:

Start w/ x in %edi y in %esi	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
Conditional →	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