

# Introduction to Computer Systems

15-213, fall 2009

5<sup>th</sup> Lecture, Sep. 7<sup>th</sup>

## **Instructors:**

Majd Sakr and Khaled Harras

# Last Time: Machine Programming, Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly (IA32):
  - Registers
  - Operands
  - Move (what's the `l` in `movl`?)

```
movl $0x4, %eax
```

```
movl %eax, %edx
```

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

```
%eax
```

```
%ecx
```

```
%edx
```

```
%ebx
```

```
%esi
```

```
%edi
```

```
%esp
```

```
%ebp
```

# Today

- Addressing modes, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

# Simple Memory Addressing Modes

## ■ Normal (R) Mem[Reg[R]]

- Register R specifies memory address

```
movl (%ecx) , %eax
```

## ■ Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movl 8(%ebp) , %edx
```

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

} Body

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

} Finish

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

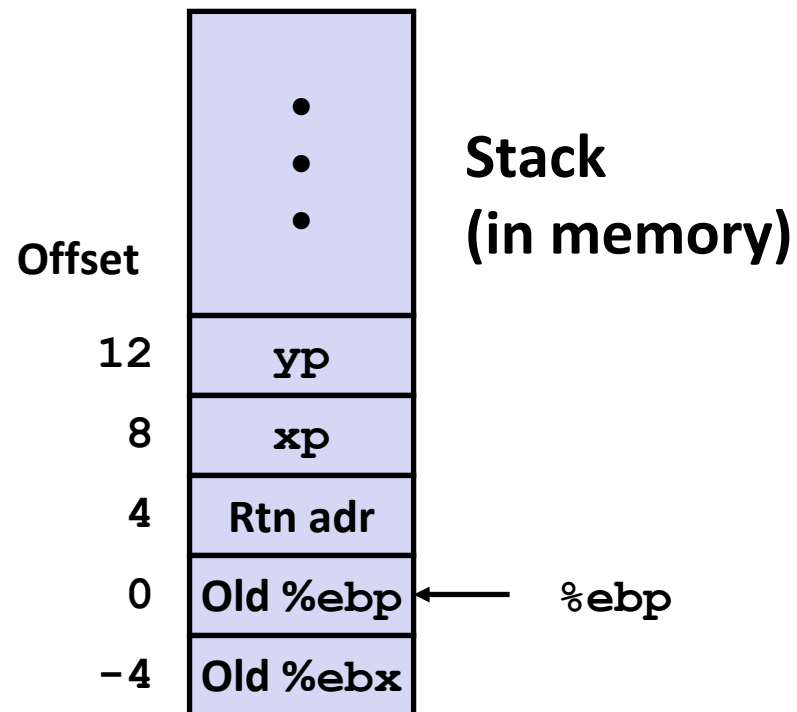
Body

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

Finish

# Understanding Swap

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



Register	Value
%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 (t1)
```

```
movl (%edx), %ebx # ebx = *xp (t0)
```

```
movl %eax, (%edx) # *xp = eax
```

```
movl %ebx, (%ecx) # *yp = ebx
```







# Understanding Swap

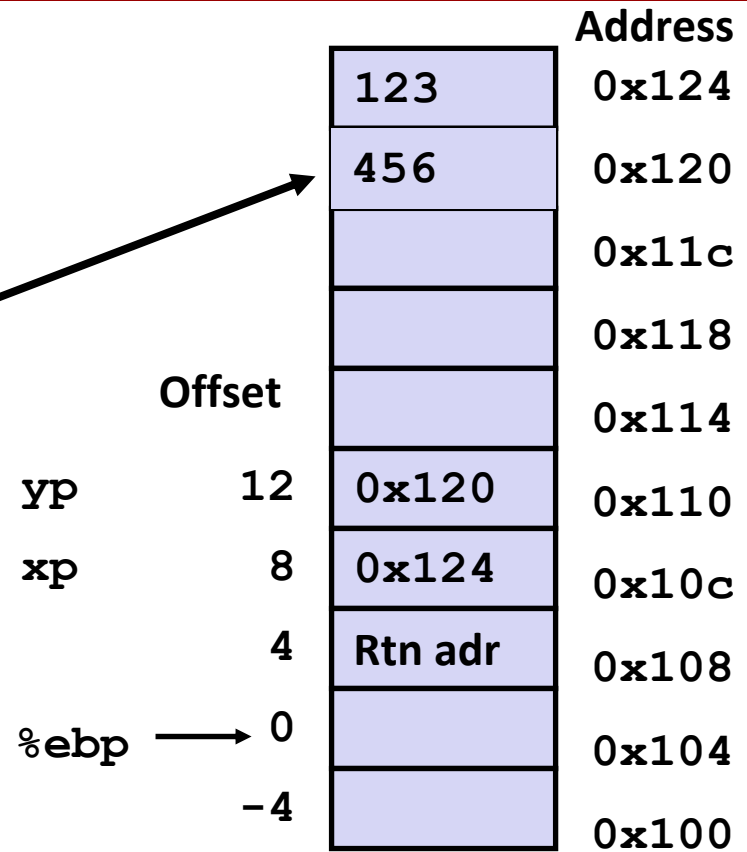
%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address	
		123	0x124
		456	0x120
			0x11c
			0x118
Offset			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```
movl 12(%ebp), %ecx      # ecx = yp
movl 8(%ebp), %edx       # edx = xp
movl (%ecx), %eax        # eax = *yp (t1)
movl (%edx), %ebx        # ebx = *xp (t0)
movl %eax, (%edx)        # *xp = eax
movl %ebx, (%ecx)        # *yp = ebx
```

# Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104



```

movl 12(%ebp), %ecx      # ecx = yp
movl 8(%ebp), %edx       # edx = xp
movl (%ecx), %eax      # eax = *yp (t1)
movl (%edx), %ebx        # ebx = *xp (t0)
movl %eax, (%edx)        # *xp = eax
movl %ebx, (%ecx)        # *yp = ebx

```

# Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Offset		Address
			123	0x124
			456	0x120
				0x11c
				0x118
				0x114
yp	12		0x120	0x110
xp	8		0x124	0x10c
	4		Rtn adr	0x108
%ebp	→ 0			0x104
	-4			0x100

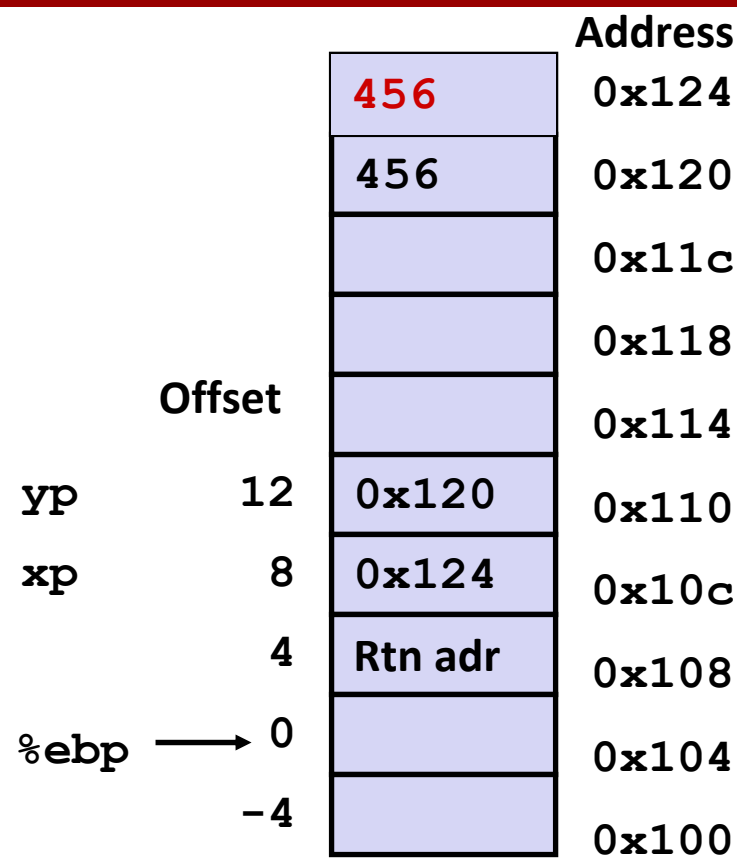
```

movl 12(%ebp), %ecx      # ecx = yp
movl 8(%ebp), %edx      # edx = xp
movl (%ecx), %eax      # eax = *yp (t1)
movl (%edx), %ebx    # ebx = *xp (t0)
movl %eax, (%edx)      # *xp = eax
movl %ebx, (%ecx)      # *yp = ebx

```

# Understanding Swap

<code>%eax</code>	456
<code>%edx</code>	0x124
<code>%ecx</code>	0x120
<code>%ebx</code>	123
<code>%esi</code>	
<code>%edi</code>	
<code>%esp</code>	
<code>%ebp</code>	0x104



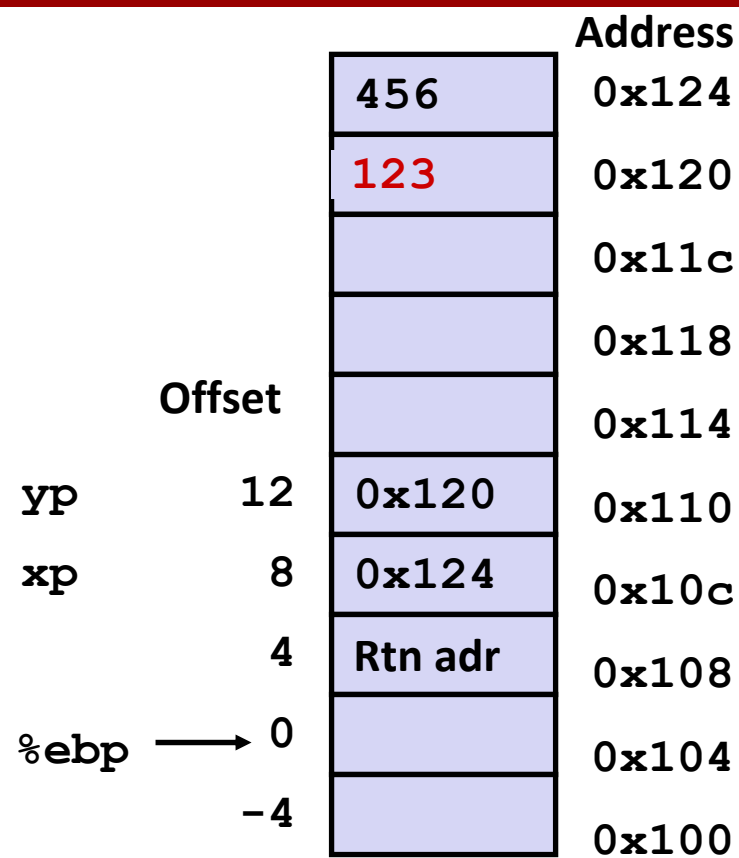
```

movl 12(%ebp), %ecx      # ecx = yp
movl 8(%ebp), %edx      # edx = xp
movl (%ecx), %eax       # eax = *yp (t1)
movl (%edx), %ebx       # ebx = *xp (t0)
movl %eax, (%edx)     # *xp = eax
movl %ebx, (%ecx)       # *yp = ebx

```

# Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104



```

movl 12(%ebp), %ecx      # ecx = yp
movl 8(%ebp), %edx       # edx = xp
movl (%ecx), %eax        # eax = *yp (t1)
movl (%edx), %ebx        # ebx = *xp (t0)
movl %eax, (%edx)        # *xp = eax
movl %ebx, (%ecx)      # *yp = ebx

```

# Complete Memory Addressing Modes

## ■ Most General Form

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

- 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

**$(Rb, Ri)$                                        $Mem[Reg[Rb]+Reg[Ri]]$**

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

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

# Address Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x100</code>

Expression	Address Computation	Address
<code>0x8(%edx)</code>	<b>will disappear blackboard?</b>	
<code>(%edx,%ecx)</code>		
<code>(%edx,%ecx,4)</code>		
<code>0x80(,%edx,2)</code>		



# Address Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x100</code>

Expression	Address Computation	Address
<code>0x8(%edx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%edx,%ecx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%edx,%ecx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%edx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

# Address Computation Instruction

## ■ `leal Src, Dest`

- *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

# Today

- Complete addressing mode, address computation (`leal`)
- **Arithmetic operations**
- `x86-64`
- Control: Condition codes
- Conditional branches
- While loops

# Some Arithmetic Operations

## ■ Two Operand Instructions:

### *Format*

**addl** *Src, Dest*

**subl** *Src, Dest*

**imull** *Src, Dest*

**sall** *Src, Dest*

**sarl** *Src, Dest*

**shrl** *Src, Dest*

**xorl** *Src, Dest*

**andl** *Src, Dest*

**orl** *Src, Dest*

### *Computation*

$Dest = Dest + Src$

$Dest = Dest - Src$

$Dest = Dest * Src$

$Dest = Dest \ll Src$

$Dest = Dest \gg Src$

$Dest = Dest \gg Src$

$Dest = Dest \wedge Src$

$Dest = Dest \& Src$

$Dest = Dest | Src$

*Also called shll*

*Arithmetic*

*Logical*

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

# Some Arithmetic Operations

## ■ One Operand Instructions

`incl Dest`                     $Dest = Dest + 1$

`decl Dest`                     $Dest = Dest - 1$

`negl Dest`                     $Dest = -Dest$

`notl Dest`                     $Dest = \sim Dest$

## ■ See book for more instructions

# Using `leal` for Arithmetic Expressions

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

} Body

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

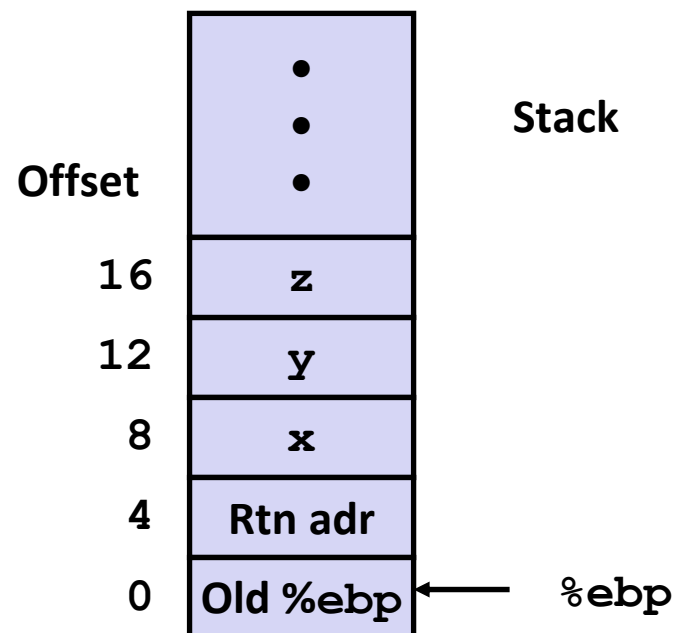
} Finish

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

```

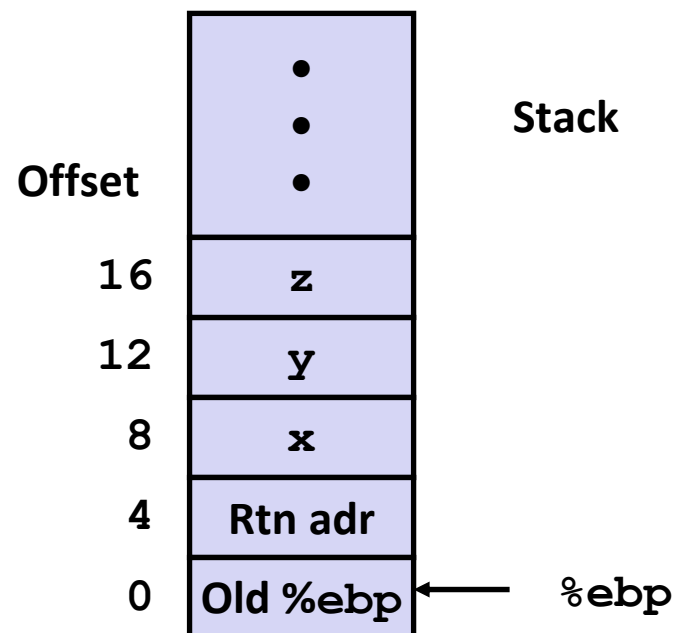
will disappear  
blackboard?

# 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),%eax      # eax = x
movl 12(%ebp),%edx     # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx # edx = 3*y
sall $4,%edx          # edx = 48*y (t4)
addl 16(%ebp),%ecx     # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax       # eax = t5*t2 (rval)

```

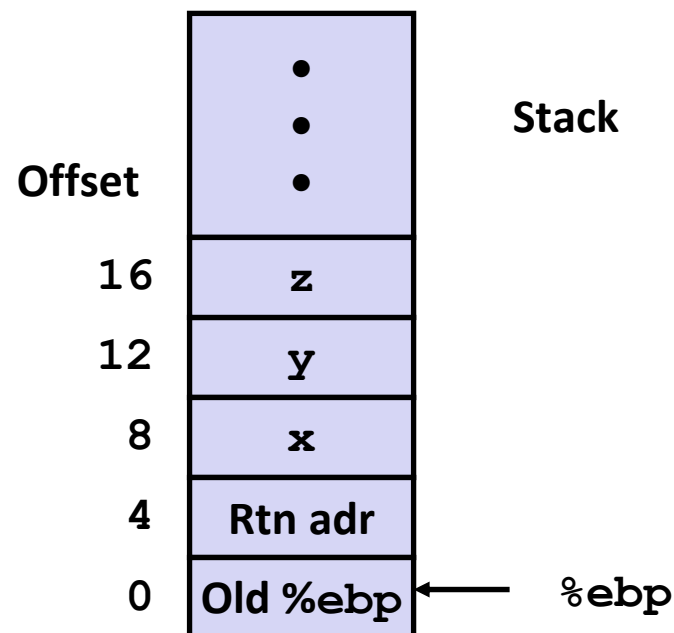


# Understanding arith

```

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), %eax      # eax = x
movl 12(%ebp), %edx     # edx = y
leal (%edx, %eax), %ecx # ecx = x+y (t1)
leal (%edx, %edx, 2), %edx # edx = 3*y
sall $4, %edx          # edx = 48*y (t4)
addl 16(%ebp), %ecx    # ecx = z+t1 (t2)
leal 4(%edx, %eax), %eax # eax = 4+t4+x (t5)
imull %ecx, %eax       # eax = t5*t2 (rval)

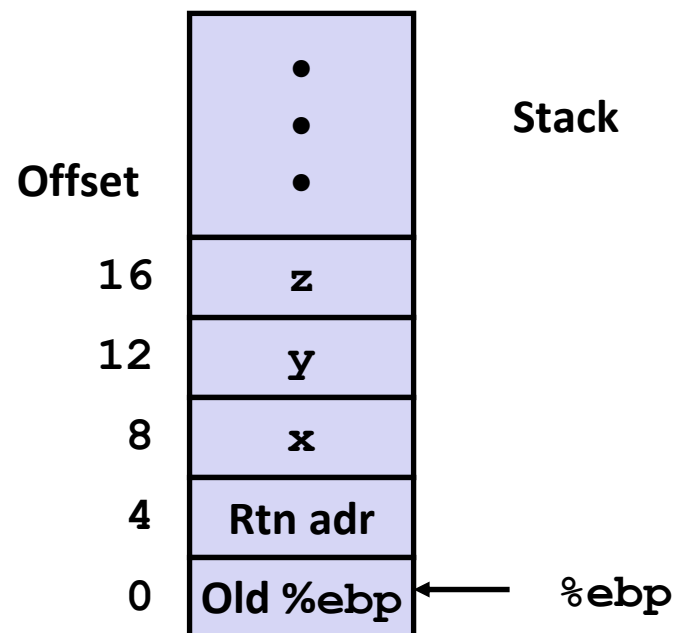
```

# 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),%eax      # eax = x
movl 12(%ebp),%edx     # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx # edx = 3*y
sall $4,%edx          # edx = 48*y (t4)
addl 16(%ebp),%ecx     # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax       # eax = t5*t2 (rval)

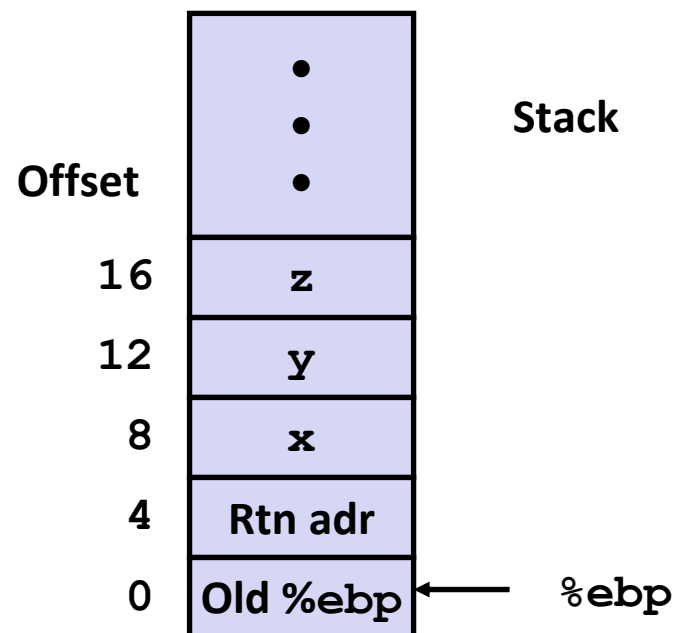
```

# 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),%eax      # eax = x
movl 12(%ebp),%edx     # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx # edx = 3*y
sall $4,%edx          # edx = 48*y (t4)
addl 16(%ebp),%ecx     # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax       # eax = t5*t2 (rval)

```

# Another Example

```
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          } Set
    movl  %esp,%ebp    } Up

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

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

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

```
# eax = x
# eax = x^y
# eax = t1>>17
# eax = t2 & 8185
```

# Another Example

```
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          } Set
    movl  %esp,%ebp    } Up

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

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

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

```
eax = x
eax = x^y      (t1)
eax = t1>>17  (t2)
eax = t2 & 8185
```

# Another Example

```
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          } Set
    movl  %esp,%ebp    } Up

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

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

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

```
eax = x
eax = x^y      (t1)
eax = t1>>17  (t2)
eax = t2 & 8185
```

# Another Example

```
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;
}
```

$$2^{13} = 8192, 2^{13} - 7 = 8185$$

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

logical:

```
pushl %ebp
movl %esp, %ebp
```

} Set  
Up

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

} Body

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

} Finish

```
eax = x
eax = x^y      (t1)
eax = t1>>17  (t2)
eax = t2 & 8185
```

# Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- **x86-64**
- Control: Condition codes
- Conditional branches
- While loops



# Data Representations: IA32 + x86-64

## ■ Sizes of C Objects (in Bytes)

<i>C Data Type</i>	<i>Typical 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*

# x86-64 Integer Registers

<b>%rax</b>	<b>%eax</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. Add 8 new ones.
- Make **%ebp/%rbp** general purpose

# Instructions

- Long word `l` (4 Bytes)  $\leftrightarrow$  Quad word `q` (8 Bytes)
- New instructions:
  - `movl`  $\rightarrow$  `movq`
  - `addl`  $\rightarrow$  `addq`
  - `sall`  $\rightarrow$  `salq`
  - etc.
- **32-bit instructions that generate 32-bit results**
  - Set higher order bits of destination register to 0
  - Example: `addl`

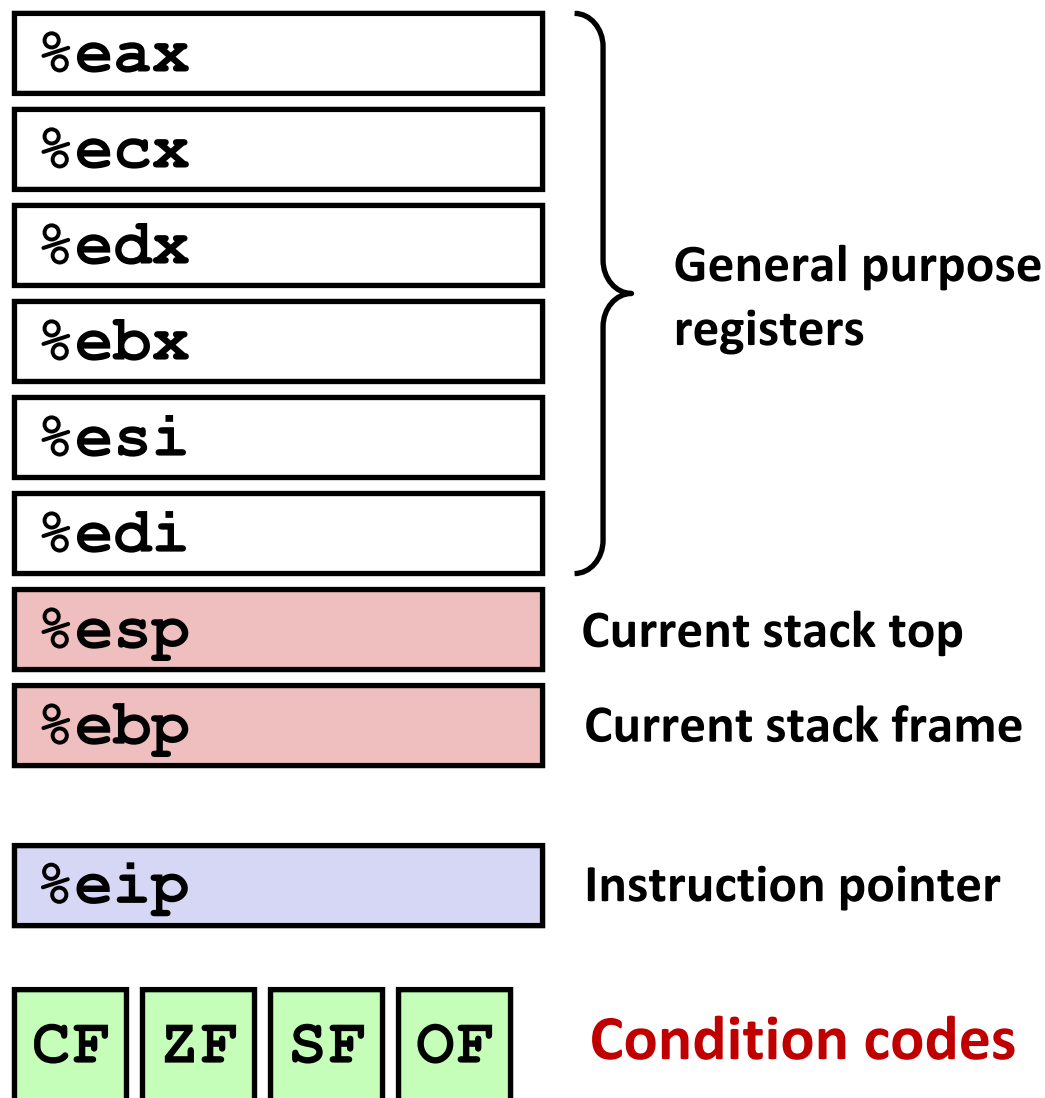
# Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- **Control: Condition codes**
- Conditional branches
- While loops

# 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

## ■ Full documentation (IA32), link also on course website

# 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

`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`



# Reading Condition Codes

## ■ SetX Instructions

- Set single byte based on combinations of condition codes

SetX	Condition	Description
<code>sete</code>	$ZF'$	Equal / Zero
<code>setne</code>	$\sim ZF'$	Not Equal / Not Zero
<code>sets</code>	$SF'$	Negative
<code>setns</code>	$\sim SF'$	Nonnegative
<code>setg</code>	$\sim (SF^{\wedge}OF) \ \& \ \sim ZF'$	Greater (Signed)
<code>setge</code>	$\sim (SF^{\wedge}OF)$	Greater or Equal (Signed)
<code>setl</code>	$(SF^{\wedge}OF)$	Less (Signed)
<code>setle</code>	$(SF^{\wedge}OF) \   \ ZF'$	Less or Equal (Signed)
<code>seta</code>	$\sim CF' \ \& \ \sim ZF'$	Above (unsigned)
<code>setb</code>	$CF'$	Below (unsigned)

# Reading Condition Codes (Cont.)

## ■ SetX Instructions:

Set single byte based on combination of condition codes

## ■ One of 8 addressable byte registers

- Does not alter remaining 3 bytes
- Typically use `movzbl` to finish job

```
int gt (int x, int y)
{
    return x > y;
}
```

<code>%eax</code>	<code>%ah</code>	<code>%al</code>
<code>%ecx</code>	<code>%ch</code>	<code>%cl</code>
<code>%edx</code>	<code>%dh</code>	<code>%dl</code>
<code>%ebx</code>	<code>%bh</code>	<code>%bl</code>
<code>%esi</code>		
<code>%edi</code>		
<code>%esp</code>		
<code>%ebp</code>		

## Body

```
movl 12(%ebp), %eax
cmpl %eax, 8(%ebp)
setg %al
movzbl %al, %eax
```

Will disappear  
Blackboard?

# Reading Condition Codes (Cont.)

## ■ SetX Instructions:

Set single byte based on combination of condition codes

## ■ One of 8 addressable byte registers

- Does not alter remaining 3 bytes
- Typically use `movzbl` to finish job

```
int gt (int x, int y)
{
    return x > y;
}
```

<code>%eax</code>	<code>%ah</code>	<code>%al</code>
<code>%ecx</code>	<code>%ch</code>	<code>%cl</code>
<code>%edx</code>	<code>%dh</code>	<code>%dl</code>
<code>%ebx</code>	<code>%bh</code>	<code>%bl</code>
<code>%esi</code>		
<code>%edi</code>		
<code>%esp</code>		
<code>%ebp</code>		

## Body

```
movl 12(%ebp), %eax    # eax = y
cmpl %eax, 8(%ebp)    # Compare x and y ←
setg %al              # al = x > y
movzbl %al, %eax      # Zero rest of %eax
```

Note  
inverted  
ordering!

# Jumping

## ■ jX Instructions

- Jump to different part of code depending on condition codes

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

# Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- **Conditional branches**
- While loops

# 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    .L7
    subl   %eax, %edx
    movl   %edx, %eax
.L8:
    leave
    ret
.L7:
    subl   %edx, %eax
    jmp    .L8

```

} Setup  
 } Body1  
 } Finish  
 } Body2

# Conditional Branch Example (Cont.)

```
int goto_ad(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 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   .L7
    subl  %eax, %edx
    movl  %edx, %eax
.L8:
    leave
    ret
.L7:
    subl  %edx, %eax
    jmp  .L8
```

# Conditional Branch Example (Cont.)

```
int goto_ad(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
```



# Conditional Branch Example (Cont.)

```
int goto_ad(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
```

# Conditional Branch Example (Cont.)

```
int goto_ad(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
```

# Conditional Branch Example (Cont.)

```
int goto_ad(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
```

# General Conditional Expression Translation

## C Code

```
val = Test ? Then-Expr : Else-Expr;
```

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

## Goto Version

```
nt = !Test;  
if (nt) goto Else;  
val = Then-Expr;  
Done:  
. . .  
Else:  
val = Else-Expr;  
goto 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

# General Form with Conditional Move

## C Code

```
val = Test ? Then-Expr : Else-Expr;
```

## Conditional Move Version

```
val1 = Then-Expr;  
val2 = Else-Expr;  
val1 = val2 if !Test;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn't hold
- **Don't use when:**
  - Then or else expression have side effects
  - Then and else expression are too expensive

# Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- **While loops**

# “Do-While” Loop Example

## C Code

```
int fact_do(int x)
{
    int result = 1;
    do {
        result *= x;
        x = x-1;
    } while (x > 1);

    return result;
}
```

## Goto Version

```
int fact_goto(int x)
{
    int result = 1;
loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto loop;
    return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds

# “Do-While” Loop Compilation

## Goto Version

```
int
fact_goto(int x)
{
    int result = 1;

loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto loop;

    return result;
}
```

## Assembly

```
fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx

.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11

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

Registers:

%edx	x
%eax	result

Will disappear  
Blackboard?



# “Do-While” Loop Compilation

## Goto Version

```
int
fact_goto(int x)
{
    int result = 1;

loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto loop;

    return result;
}
```

## Assembly

```
fact_goto:
    pushl %ebp                # Setup
    movl %esp,%ebp          # Setup
    movl $1,%eax            # eax = 1
    movl 8(%ebp),%edx        # edx = x

.L11:
    imull %edx,%eax         # result *= x
    decl %edx                # x--
    cmpl $1,%edx            # Compare x : 1
    jg .L11                 # if > goto loop

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

## Registers:

%edx	x
%eax	result

# General “Do-While” Translation

## C Code

```
do  
    Body  
while (Test);
```

## Goto Version

```
loop:  
    Body  
    if (Test)  
        goto loop
```

- *Body*:  
 {  
 *Statement*<sub>1</sub>;  
 *Statement*<sub>2</sub>;  
 ...  
 *Statement*<sub>*n*</sub>;  
 }

- *Test* returns integer  
 = 0 interpreted as false  
 ≠0 interpreted as true

# “While” Loop Example

## C Code

```
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {

        result *= x;
        x = x-1;
    };

    return result;
}
```

## Goto Version #1

```
int fact_while_goto(int x)
{
    int result = 1;
loop:
    if (!(x > 1))
        goto done;
    result *= x;
    x = x-1;
    goto loop;
done:
    return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

# Alternative “While” Loop Translation

## C Code

```
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    };
    return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

## Goto Version #2

```
int fact_while_goto2(int x)
{
    int result = 1;
    if (!(x > 1))
        goto done;
loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto loop;
done:
    return result;
}
```

# General “While” Translation

## While version

```
while (Test)  
  Body
```



## Do-While Version

```
if (!Test)  
  goto done;  
do  
  Body  
  while (Test) ;  
done :
```



## Goto Version

```
if (!Test)  
  goto done;  
loop :  
  Body  
  if (Test)  
    goto loop;  
done :
```

# New Style “While” Loop Translation

## C Code

```
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    };
    return result;
}
```

## Goto Version

```
int fact_while_goto3(int x)
{
    int result = 1;
    goto middle;
loop:
    result *= x;
    x = x-1;
middle:
    if (x > 1)
        goto loop;
    return result;
}
```

- Recent technique for GCC
  - Both IA32 & x86-64
- First iteration jumps over body computation within loop

# Jump-to-Middle While Translation

## C Code

```
while (Test)  
    Body
```



## Goto Version

```
goto middle;  
loop:  
    Body  
middle:  
    if (Test)  
        goto loop;
```

- Avoids duplicating test code
- Unconditional goto incurs no performance penalty
- for loops compiled in similar fashion

## Goto (Previous) Version

```
if (!Test)  
    goto done;  
loop:  
    Body  
    if (Test)  
        goto loop;  
done:
```

# Jump-to-Middle Example

```
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x--;
    };
    return result;
}
```

```
# x in %edx, result in %eax
    jmp    .L34          # goto Middle
.L35:                # Loop:
    imull %edx, %eax   # result *= x
    decl  %edx         # x--
.L34:                # Middle:
    cmpl  $1, %edx    # x:1
    jg    .L35         # if >, goto Loop
```



# Implementing Loops

## ■ IA32

- All loops translated into form based on “do-while”
- IA32 compiler developed for machine where all operations costly