

Machine Programming II: C to assembly

- Move instructions, registers, and operands
- Complete addressing mode, address computation (`leal`)
- Arithmetic operations (including some x86-64 instructions)
- Condition codes
- Control, unconditional and conditional branches
- While loops

Three Kinds of Instructions

- Perform arithmetic function on register or memory data
 - $c = a + b;$
- Transfer data between memory and register
 - Load data from memory into register
 - `%reg = Mem[address]`
 - Store register data into memory
 - `Mem[address] = %reg`
- Transfer control (control flow)
 - Unconditional jumps to/from procedures
 - Conditional branches

Moving Data: IA32

■ Moving Data

- `movx Source, Dest`
- `x` is one of {`b`, `w`, `l`}

- `movl Source, Dest:`
Move 4-byte “long word”
- `movw Source, Dest:`
Move 2-byte “word”
- `movb Source, Dest:`
Move 1-byte “byte”

■ Lots of these in typical code

<code>%eax</code>
<code>%ecx</code>
<code>%edx</code>
<code>%ebx</code>
<code>%esi</code>
<code>%edi</code>
<code>%esp</code>
<code>%ebp</code>

Moving Data: IA32

■ Moving Data

`movl Source, Dest:`

■ 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”

<code>%eax</code>
<code>%ecx</code>
<code>%edx</code>
<code>%ebx</code>
<code>%esi</code>
<code>%edi</code>
<code>%esp</code>
<code>%ebp</code>

movl Operand Combinations

	Source	Dest	Src, Dest	C Analog
movl	Imm	Reg	movl \$0x4, %eax	
		Mem	movl \$-147, (%eax)	
	Reg	Reg	movl %eax, %edx	
		Mem	movl %eax, (%edx)	
	Mem	Reg	movl (%eax), %edx	

Cannot do memory-memory transfer with a single instruction.

How do you copy from a memory location to another then?

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;

Memory vs. registers

- Why both?
- Performance?
- Usage difference?

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

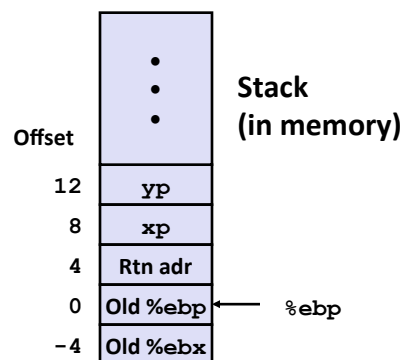
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax, (%edx)
movl %ebx, (%ecx)   } Body

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

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
```

```
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
```

```
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
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	
	-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

		Address
	123	0x124
	456	0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	
	-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

```

```
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
```

```
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
```



```
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
```

x86-64 Integer Registers

- 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 generate 32-bit results,

- What about the other 32 bits in the register?
- Set higher order bits of destination register to 0
- Example: `addl`

Swap in 32-bit Mode

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

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

    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
```

Swap in 64-bit Mode

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

```
swap:
    movl    (%rdi), %edx
    movl    (%rsi), %eax
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)
    retq
```

- **Operands passed in registers (why useful?)**
 - First (**x**p) in %rdi, second (**y**p) in %rsi
 - 64-bit pointers
- **No stack operations required**
- **32-bit data**
 - Data held in registers %eax and %edx
 - **movl** operation

Swap Long Ints in 64-bit Mode

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

```
swap_l:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    retq
```

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

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

%edx	0xf000
%ecx	0x100

(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]]**
D(Rb) **Mem[Reg[Rb] +D]**

Expression	Address Computation	Address
0x8 (%edx)		
(%edx,%ecx)		
(%edx,%ecx,4)		
0x80(,%edx,2)		

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*i`
 - `k = 1, 2, 4, or 8`

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

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

<code>incl Dest</code>	$Dest = Dest + 1$
<code>decl Dest</code>	$Dest = Dest - 1$
<code>negl Dest</code>	$Dest = -Dest$
<code>notl Dest</code>	$Dest = \sim Dest$

■ See book for more instructions

Using `leal` for Arithmetic Expressions

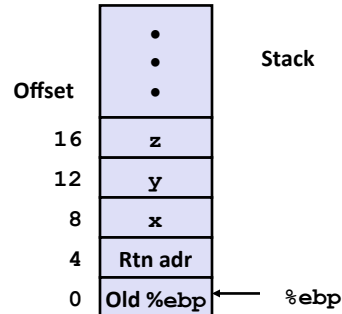
```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

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

} Set Up
 } Body
 } 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;
}
```

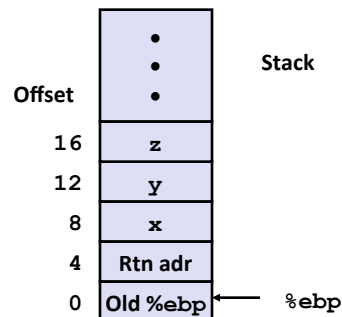


```
movl 8(%ebp), %eax
movl 12(%ebp), %edx
leal (%edx,%eax), %ecx
leal (%edx,%edx,2), %edx
sall $4, %edx
addl 16(%ebp), %ecx
leal 4(%edx,%eax), %eax
imull %ecx, %eax
```

What does each of these instructions mean?

Understanding arith

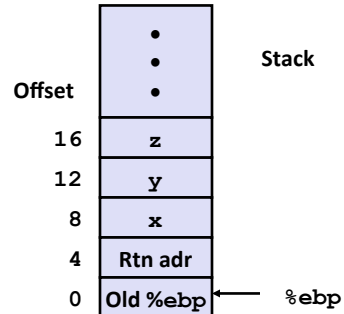
```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```



```
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

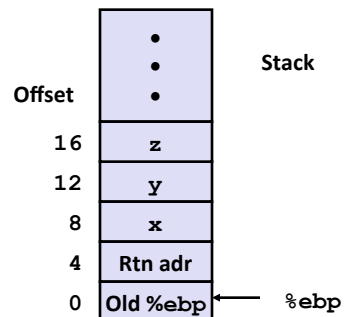
```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```



```
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

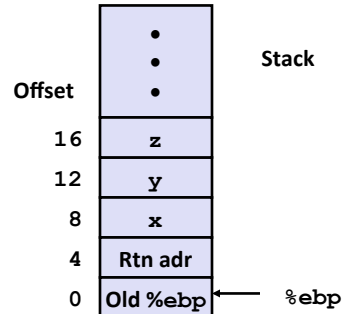
```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```



```
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;
}
```

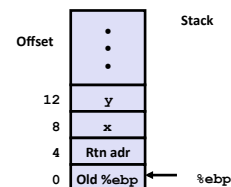
logical:

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

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

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

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

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

<code>movl 8(%ebp),%eax</code>	<code>eax = x</code>
<code>xorl 12(%ebp),%eax</code>	<code>eax = x^y (t1)</code>
<code>sarl \$17,%eax</code>	<code>eax = t1>>17 (t2)</code>
<code>andl \$8185,%eax</code>	<code>eax = t2 & 8185</code>

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   }
    sarl $17,%eax        } Body
    andl $8185,%eax

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

<code>movl 8(%ebp),%eax</code>	<code>eax = x</code>
<code>xorl 12(%ebp),%eax</code>	<code>eax = x^y (t1)</code>
<code>sarl \$17,%eax</code>	<code>eax = t1>>17 (t2)</code>
<code>andl \$8185,%eax</code>	<code>eax = t2 & 8185</code>

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;
}
```

$2^{13} = 8192$, $2^{13} - 7 = 8185$
 ...0010000000000000, ...0001111111111001

<code>movl 8(%ebp), %eax</code>	<code>eax = x</code>
<code>xorl 12(%ebp), %eax</code>	<code>eax = x^y (t1)</code>
<code>sarl \$17, %eax</code>	<code>eax = t1>>17 (t2)</code>
<code>andl \$8185, %eax</code>	<code>eax = t2 & 8185</code>

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

Control-Flow/Conditionals

■ Unconditional

```
while(true) {
    do_something;
}
...
```

■ Conditional

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

Conditionals and Control Flow

- **A test / conditional branch is sufficient to implement most control flow constructs offered in higher level languages**
 - if (condition) then {...} else {...}
 - while(condition) {...}
 - do {...} while (condition)
 - for (initialization; condition; iterative) {...}
- **Unconditional branches implemented some related control flow constructs**
 - break, continue

Jumping

■ jX Instructions

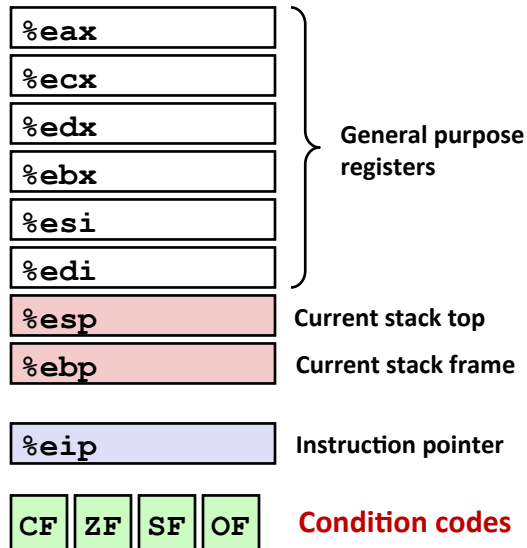
- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
j _e	ZF	Equal / Zero
j _{ne}	~ZF	Not Equal / Not Zero
j _s	SF	Negative
j _{ns}	~SF	Nonnegative
j _g	~(SF^OF) & ~ZF	Greater (Signed)
j _{ge}	~(SF^OF)	Greater or Equal (Signed)
j _l	(SF^OF)	Less (Signed)
j _{le}	(SF^OF) ZF	Less or Equal (Signed)
j _a	~CF & ~ZF	Above (unsigned)
j _b	CF	Below (unsigned)

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 (beware!)

■ Full documentation (IA32) <http://www.jegerlehner.ch/intel/IntelCodeTable.pdf>

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`
- `testl %eax, %eax`
 - Sets SF and ZF, check if `eax` is +,0,-

Reading Condition Codes

■ SetX Instructions

- Set a 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>	$(SF \wedge OF) \mid ZF$	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
```

What does each of these instructions do?

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;
}
```

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

Body

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

Jumping

■ jX Instructions

- Jump to different part of code depending on condition codes

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

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
```

Diagram illustrating the assembly code for the `absdiff` function, grouped into sections:

- Setup:** `pushl %ebp`, `movl %esp, %ebp`
- Body1:** `movl 8(%ebp), %edx`, `movl 12(%ebp), %eax`, `cmpl %eax, %edx`, `jle .L7`, `subl %eax, %edx`, `movl %edx, %eax`
- Finish:** `.L8:`, `leave`, `ret`
- Body2:** `.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
```

- 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;
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
- How would you make this efficient?

Conditionals: x86-64

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

```
absdiff: # x in %edi, y in %esi
    movl    %edi, %eax    # eax = x
    movl    %esi, %edx    # edx = y
    subl    %esi, %eax    # eax = x - y
    subl    %edi, %edx    # edx = y - x
    cmpl    %esi, %edi    # x:y
    cmovle  %edx, %eax    # eax=edx if <=
    ret
```

Conditional move instruction

- **cmovC** src, dest
- Move value from src to dest if condition *C* holds
- More efficient than conditional branching (simple control flow)
- But overhead: both branches are evaluated

PC Relative Addressing

0x100	cmp	r2, r3	0x1000
0x102	je	0x70	0x1002
0x104	...		0x1004
...	...		...
0x172	add	r3, r4	0x1072



- PC relative branches are relocatable
- Absolute branches are not

Compiling Loops

C/Java code:

```
while ( sum != 0 ) {
    <loop body>
}
```

Machine code:

```
loopTop:  cmp    r3, $0
          be     loopDone
          <loop body code>
          jmp    loopTop
loopDone:
```

- How to compile other loops should be straightforward
 - The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop
- Q: How is `for(i=0; i<100; i++)` implemented?

Machine Programming II: Instructions (cont'd)

- Move instructions, registers, and operands
- Complete addressing mode, address computation (`leal`)
- Arithmetic operations (including some x86-64 instructions)
- Condition codes
- Control, unconditional and conditional branches
- While loops
- For loops
- Switch statements

“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

Translation?

“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*₁;
 *Statement*₂;
 ...
 *Statement*_{*n*};
 }

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

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

- Used by GCC for both IA32 & x86-64
- First iteration jumps over body computation within loop straight to test

“While” Loop 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
```

Quick Review

■ Complete memory addressing mode

- `(%eax), 17(%eax), 2(%ebx, %ecx, 8), ...`

■ Arithmetic operations that do set condition codes

- `subl %eax, %ecx` # `ecx = ecx + eax`
- `sall $4, %edx` # `edx = edx << 4`
- `addl 16(%ebp), %ecx` # `ecx = ecx + Mem[16+ebp]`
- `imull %ecx, %eax` # `eax = eax * ecx`

■ Arithmetic operations that do NOT set condition codes

- `leal 4(%edx, %eax), %eax` # `eax = 4 + edx + eax`

Quick Review

■ x86-64 vs. IA32

- Integer registers: 16 x 64-bit vs. 8 x 32-bit
- `movq, addq, ...` vs. `movl, addl, ...`
 - `movq` -> "move quad word" or 4*16-bits
- Better support for passing function arguments in 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

■ Control

- Condition code registers
- Set as side effect or by `cmp, test`
- Used:
 - Read out by `setx` instructions (`setg, setle, ...`)
 - Or by conditional jumps (`jle .L4, je .L10, ...`)

CF	ZF	SF	OF
----	----	----	----

Quick Review

■ Do-While loop

C Code

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

Goto Version

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

■ While-Do loop

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:
```

or

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

“For” Loop Example: Square-and-Multiply

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}
```

■ Algorithm

- Exploit bit representation: $p = p_0 + 2p_1 + 2^2p_2 + \dots + 2^{n-1}p_{n-1}$
- Gives: $x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \dots \cdot \underbrace{((z_{n-1}^2)^2) \dots^2}_{n-1 \text{ times}}$
 - $z_i = 1$ when $p_i = 0$
 - $z_i = x$ when $p_i = 1$
- Complexity $O(\log p)$

Example

$$\begin{aligned} 3^{10} &= 3^2 * 3^8 \\ &= 3^2 * ((3^2)^2)^2 \end{aligned}$$

ipwr Computation

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}
```

before iteration	result	x=3	p=10
1	1	3	10=1010 ₂
2	1	9	5= 101 ₂
3	9	81	2= 10 ₂
4	9	6561	1= 1 ₂
5	59049	43046721	0 ₂

“For” Loop Example

```
int result;
for (result = 1; p != 0; p = p>>1)
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

General Form

```
for (Init; Test; Update)
    Body
```

Test

```
p != 0
```

Init

```
result = 1
```

Update

```
p = p >> 1
```

Body

```
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

“For” → “While”

For Version

```
for (Init; Test; Update )
    Body
```



While Version

```
Init;
while (Test) {
    Body
    Update ;
}
```



Goto Version

```
Init;
goto middle;
loop:
    Body
    Update ;
middle:
    if (Test)
        goto loop;
done:
```

For-Loop: Compilation

For Version

```
for (Init; Test; Update )
    Body
```



Goto Version

```
Init;
goto middle;
loop:
    Body
    Update ;
middle:
    if (Test)
        goto loop;
done:
```

```
for (result = 1; p != 0; p = p>>1)
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```



```
result = 1;
goto middle;
loop:
    if (p & 0x1)
        result *= x;
    x = x*x;
    p = p >> 1;
middle:
    if (p != 0)
        goto loop;
done:
```

```
long switch_eg
(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

- Multiple case labels
 - Here: 5, 6
- Fall through cases
 - Here: 2
- Missing cases
 - Here: 4
- Lots to manage, we need a "jump table"

Jump Table Structure

Switch Form

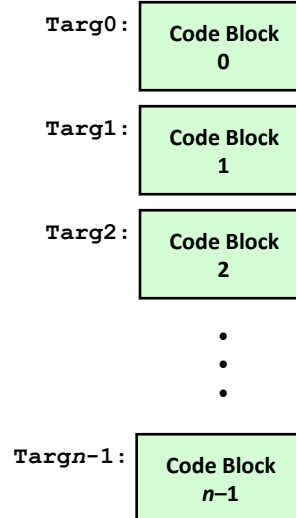
```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    . . .
  case val_n-1:
    Block n-1
}
```

Jump Table

jtab:

Targ0
Targ1
Targ2
•
•
•
Targn-1

Jump Targets



Approximate Translation

```
target = JTab[x];
goto *target;
```

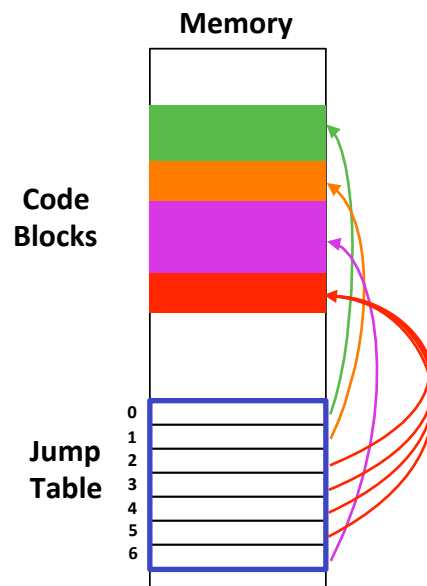
Jump Table Structure

C code:

```
switch(x) {
  case 0: <some code>
    break;
  case 1: <some code>
    break;
  case 6: <some code>
    break;
  default: <some code>
    break;
}
```

We can use the jump table when $x \leq 6$:

```
if (x <= 6)
  target = JTab[x];
  goto *target;
else
  goto default;
```



Jump Table

Jump table

```
.section .rodata
.align 4
.L62:
.long .L61 # x = 0
.long .L56 # x = 1
.long .L57 # x = 2
.long .L58 # x = 3
.long .L61 # x = 4
.long .L60 # x = 5
.long .L60 # x = 6
```

```
switch(x) {
case 1:      // .L56
    w = y*z;
    break;
case 2:      // .L57
    w = y/z;
    /* Fall Through */
case 3:      // .L58
    w += z;
    break;
case 5:
case 6:      // .L60
    w -= z;
    break;
default:    // .L61
    w = 2;
}
```

Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

```
Setup:  switch_eg:
        pushl %ebp           # Setup
        movl %esp, %ebp     # Setup
        pushl %ebx          # Setup
        movl $1, %ebx
        movl 8(%ebp), %edx
        movl 16(%ebp), %ecx
        cmpl $6, %edx
        ja .L61
        jmp *.L62(, %edx, 4)
```

Jump table

```
.section .rodata
.align 4
.L62:
.long .L61 # x = 0
.long .L56 # x = 1
.long .L57 # x = 2
.long .L58 # x = 3
.long .L61 # x = 4
.long .L60 # x = 5
.long .L60 # x = 6
```

Translation?

Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Jump table

```
.section .rodata
.align 4
.L62:
    .long .L61 # x = 0
    .long .L56 # x = 1
    .long .L57 # x = 2
    .long .L58 # x = 3
    .long .L61 # x = 4
    .long .L60 # x = 5
    .long .L60 # x = 6
```

```
Setup:  switch_eg:
        pushl %ebp                # Setup
        movl %esp, %ebp          # Setup
        pushl %ebx               # Setup
        movl $1, %ebx            # w = 1
        movl 8(%ebp), %edx        # edx = x
        movl 16(%ebp), %ecx       # ecx = z
        cmpl $6, %edx            # x:6
        ja .L61                  # if > goto default
        jmp *.L62(, %edx, 4)      # goto JTab[x]
```

Indirect
jump →

Assembly Setup Explanation

■ Table Structure

- Each target requires 4 bytes
- Base address at .L62

Jump table

```
.section .rodata
.align 4
.L62:
    .long .L61 # x = 0
    .long .L56 # x = 1
    .long .L57 # x = 2
    .long .L58 # x = 3
    .long .L61 # x = 4
    .long .L60 # x = 5
    .long .L60 # x = 6
```

■ Jumping

- Direct:** `jmp .L61`
- Jump target is denoted by label .L61

Indirect: `jmp *.L62(, %edx, 4)`

- Start of jump table: .L62
- Must scale by factor of 4 (labels are 32-bits = 4 bytes on IA32)
- Fetch target from effective Address `.L62 + edx*4`
 - Only for $0 \leq x \leq 6$

Jump Table

Jump table

```
.section .rodata
.align 4
.L62:
.long .L61 # x = 0
.long .L56 # x = 1
.long .L57 # x = 2
.long .L58 # x = 3
.long .L61 # x = 4
.long .L60 # x = 5
.long .L60 # x = 6
```

```
switch(x) {
case 1:      // .L56
    w = y*z;
    break;
case 2:      // .L57
    w = y/z;
    /* Fall Through */
case 3:      // .L58
    w += z;
    break;
case 5:
case 6:      // .L60
    w -= z;
    break;
default:     // .L61
    w = 2;
}
```

Code Blocks (Partial)

```
switch(x) {
    . . .
case 2:      // .L57
    w = y/z;
    /* Fall Through */
case 3:      // .L58
    w += z;
    break;
    . . .
default:     // .L61
    w = 2;
}
```

```
.L61: // Default case
    movl $2, %ebx    # w = 2
    movl %ebx, %eax  # Return w
    popl %ebx
    leave
    ret
.L57: // Case 2:
    movl 12(%ebp), %eax # y
    cld                # Div prep
    idivl %ecx         # y/z
    movl %eax, %ebx    # w = y/z
# Fall through
.L58: // Case 3:
    addl %ecx, %ebx    # w+= z
    movl %ebx, %eax    # Return w
    popl %ebx
    leave
    ret
```

Code Blocks (Rest)

```
switch(x) {
case 1:      // .L56
    w = y*z;
    break;
    . . .
case 5:
case 6:      // .L60
    w -= z;
    break;
    . . .
}
```

```
.L60: // Cases 5&6:
    subl    %ecx, %ebx # w -= z
    movl    %ebx, %eax # Return w
    popl    %ebx
    leave
    ret

.L56: // Case 1:
    movl    12(%ebp), %ebx # w = y
    imull    %ecx, %ebx    # w*= z
    movl    %ebx, %eax # Return w
    popl    %ebx
    leave
    ret
```

IA32 Object Code

■ Setup

- Label `.L61` becomes address `0x08048630`
- Label `.L62` becomes address `0x080488dc`

Assembly Code

```
switch_eg:
    . . .
    ja     .L61          # if > goto default
    jmp    *.L62(, %edx, 4) # goto JTab[x]
```

Disassembled Object Code

```
08048610 <switch_eg>:
    . . .
08048622: 77 0c                ja     8048630
08048624: ff 24 95 dc 88 04 08 jmp    *0x80488dc(, %edx, 4)
```

IA32 Object Code (cont.)

■ Jump Table

- Doesn't show up in disassembled code
- Can inspect using GDB

```
gdb asm-cntl
```

```
(gdb) x/7xw 0x080488dc
```

- Examine 7 hexadecimal format "words" (4-bytes each)
- Use command "**help x**" to get format documentation

```
0x080488dc:
```

```
0x08048630
```

```
0x08048650
```

```
0x0804863a
```

```
0x08048642
```

```
0x08048630
```

```
0x08048649
```

```
0x08048649
```

Disassembled Targets

8048630:	bb 02 00 00 00	mov \$0x2,%ebx
8048635:	89 d8	mov %ebx,%eax
8048637:	5b	pop %ebx
8048638:	c9	leave
8048639:	c3	ret
804863a:	8b 45 0c	mov 0xc(%ebp),%eax
804863d:	99	cld
804863e:	f7 f9	idiv %ecx
8048640:	89 c3	mov %eax,%ebx
8048642:	01 cb	add %ecx,%ebx
8048644:	89 d8	mov %ebx,%eax
8048646:	5b	pop %ebx
8048647:	c9	leave
8048648:	c3	ret
8048649:	29 cb	sub %ecx,%ebx
804864b:	89 d8	mov %ebx,%eax
804864d:	5b	pop %ebx
804864e:	c9	leave
804864f:	c3	ret
8048650:	8b 5d 0c	mov 0xc(%ebp),%ebx
8048653:	0f af d9	imul %ecx,%ebx
8048656:	89 d8	mov %ebx,%eax
8048658:	5b	pop %ebx
8048659:	c9	leave
804865a:	c3	ret

Matching Disassembled Targets

0x08048630	8048630:	bb 02 00 00 00	mov
0x08048650	8048635:	89 d8	mov
0x0804863a	8048637:	5b	pop
0x08048642	8048638:	c9	leave
0x08048630	8048639:	c3	ret
0x08048649	804863a:	8b 45 0c	mov
0x08048649	804863d:	99	cld
	804863e:	f7 f9	idiv
	8048640:	89 c3	mov
	8048642:	01 cb	add
	8048644:	89 d8	mov
	8048646:	5b	pop
	8048647:	c9	leave
	8048648:	c3	ret
	8048649:	29 cb	sub
	804864b:	89 d8	mov
	804864d:	5b	pop
	804864e:	c9	leave
	804864f:	c3	ret
	8048650:	8b 5d 0c	mov
	8048653:	0f af d9	imul
	8048656:	89 d8	mov
	8048658:	5b	pop
	8048659:	c9	leave
	804865a:	c3	ret

Question

- Would you implement this with a jump table?

```
switch(x) {
  case 0:      <some code>
              break;
  case 10:     <some code>
              break;
  case 52000:  <some code>
              break;
  default:     <some code>
              break;
}
```

- Probably not:
 - Don't want a jump table with 52000 entries (too big)

Summarizing

■ C Control

- if-then-else
- do-while
- while, for
- switch

■ Assembler Control

- Conditional jump
- Conditional move
- Indirect jump
- Compiler
- Must generate assembly code to implement more complex control

■ Standard Techniques

- Loops converted to do-while form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees (see text)

■ Conditions in CISC

- CISC machines generally have condition code registers