Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

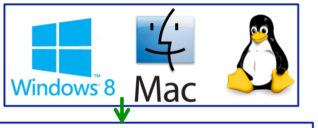
Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

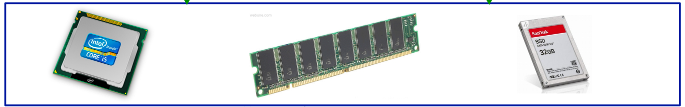
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

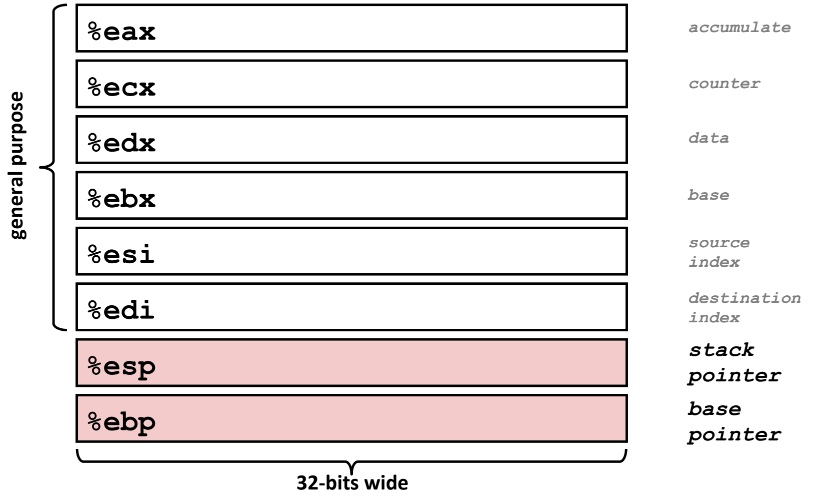
Next x86 topics

- Move instructions, registers, and operands
- Memory addressing modes
- swap example: 32-bit vs. 64-bit
- Arithmetic operations

What Is A Register (again)?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)
- Registers have names, not addresses.
- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but *especially* x86

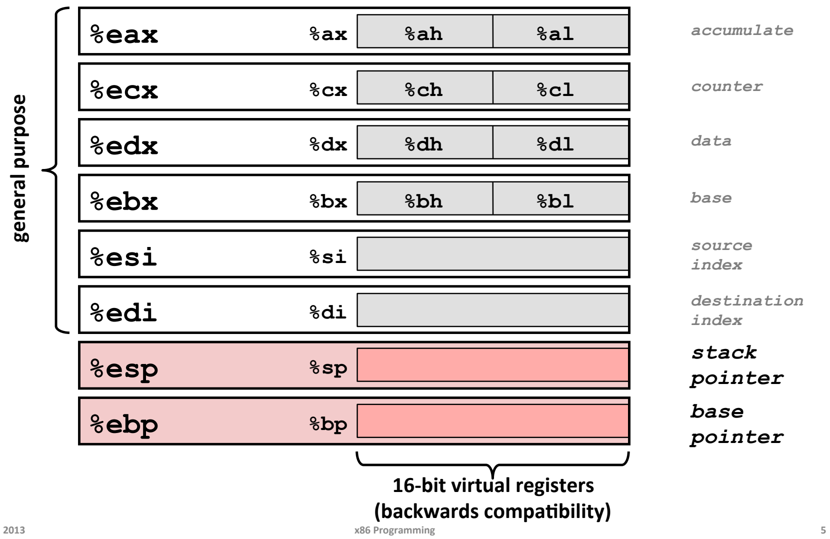
Integer Registers (IA32)



		Origin (mostly obsolete)
general purpose	%eax	*accumulate*
	%ecx	*counter*
	%edx	*data*
	%ebx	*base*
	%esi	*source index*
	%edi	*destination index*
	%esp	***stack pointer***
	%ebp	***base pointer***

32-bits wide

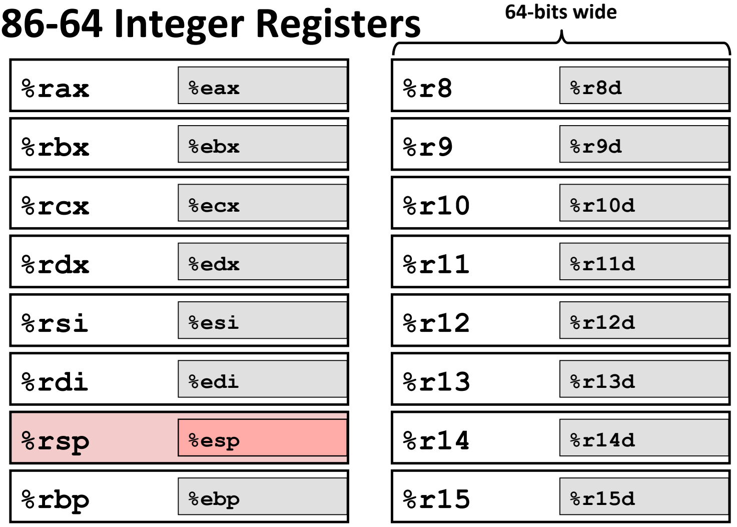
Integer Registers (IA32)



					Origin (mostly obsolete)
general purpose	%eax	%ax	%ah	%al	accumulate
	%ecx	%cx	%ch	%cl	counter
	%edx	%dx	%dh	%dl	data
	%ebx	%bx	%bh	%bl	base
	%esi	%si			source index
	%edi	%di			destination index
	%esp	%sp			stack pointer
	%ebp	%bp			base pointer

16-bit virtual registers (backwards compatibility)

x86-64 Integer Registers



64-bits wide

%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, and add 8 new ones; *all* accessible as 8, 16, 32, 64 bits.

Assembly Data Types

- **"Integer" data of 1, 2, 4 (IA32), or 8 (just in x86-64) bytes**
  - Data values
  - Addresses (untyped pointers)
- **Floating point data of 4, 8, or 10 bytes**
- **What about "aggregate" types such as arrays?**
  - Just contiguous memory locations

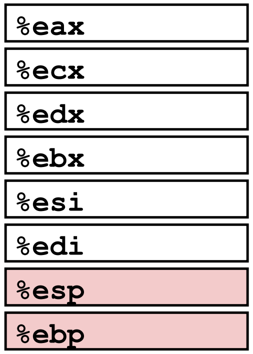
Three Basic Kinds of Instructions

- **Transfer data between memory and register**
  - *Load* data from memory into register
    - %reg = Mem[address]
  - *Store* register data into memory
    - Mem[address] = %reg

Remember: memory is indexed just like an array[] of bytes!

- **Perform arithmetic function on register or memory data**
  - c = a + b; z = x << y; i = h & g;
- **Transfer control: what instruction to execute next**
  - Unconditional jumps to/from procedures
  - Conditional branches

Moving Data: IA32



%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp

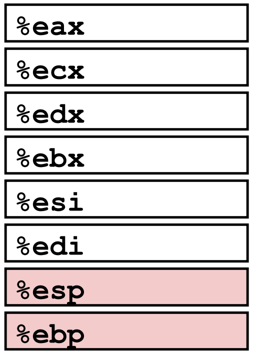
- Moving Data
  - `movx` *Source*, *Dest*
  - `x` is one of {`b, w, l`}

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

confusing historical terms…
not the current machine word size

- **Lots of these in typical code**

Moving Data: IA32



%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp

- 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"

`movl` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`var_a = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p_a = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`var_d = var_a;`
		Mem	`movl %eax,(%edx)`	`*p_d = var_a;`
	Mem	*Reg*	`movl (%eax),%edx`	`var_d = *p_a;`

Cannot do memory-memory transfer with a single instruction.

How would you do it?

Memory vs. registers

- **What is the main difference?**
- **Addresses vs. Names**
- **Big vs. Small**

Memory Addressing Modes: Basic

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

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

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies a memory address
    - (e.g. the start of some memory region)
  - Constant displacement D specifies the offset from that address

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

Using Basic 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)
```

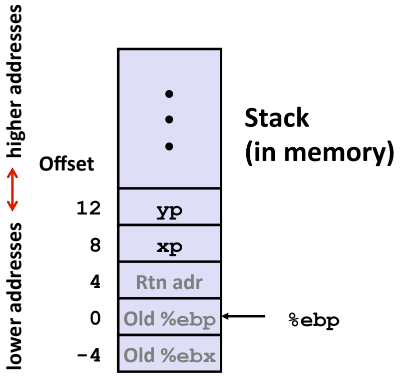
Body

```
   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;
}
```



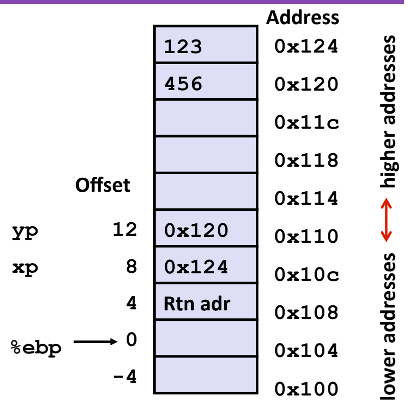
Register	Value
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

register <-> variable mapping

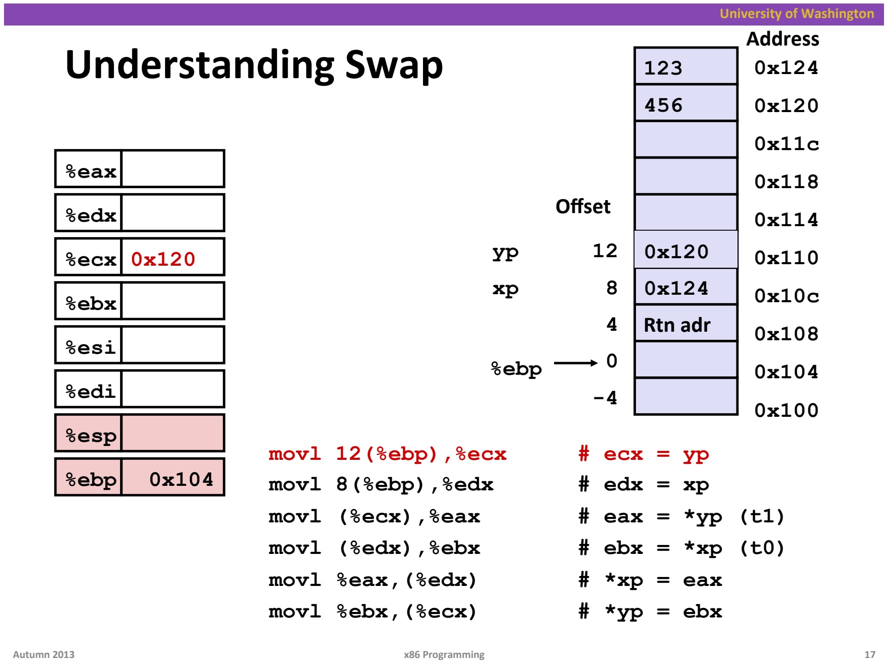
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap

%eax	
%edx	
%ecx	
%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
```

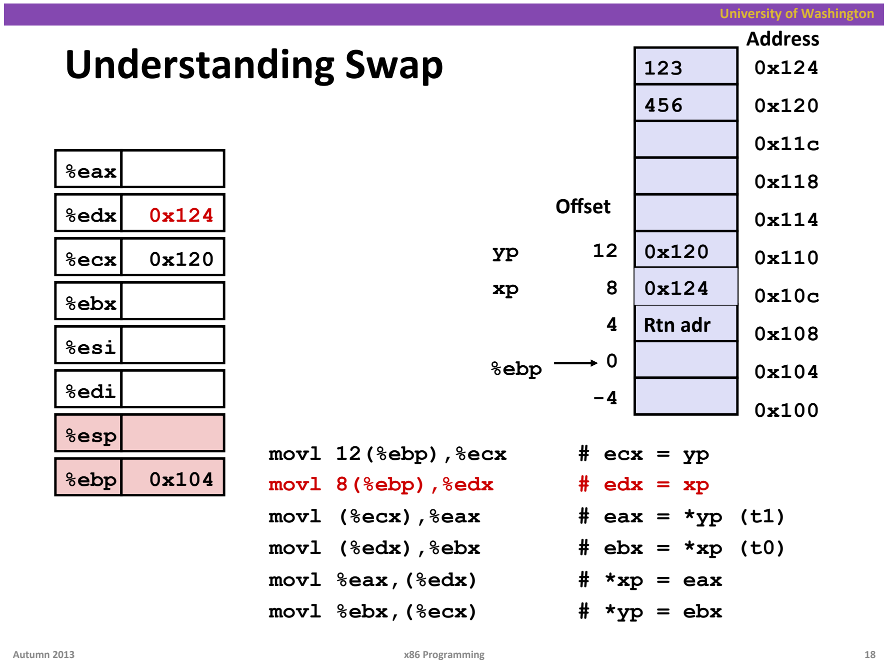


Understanding Swap

%eax	
%edx	
%ecx	0x120
%ebx	
%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
```

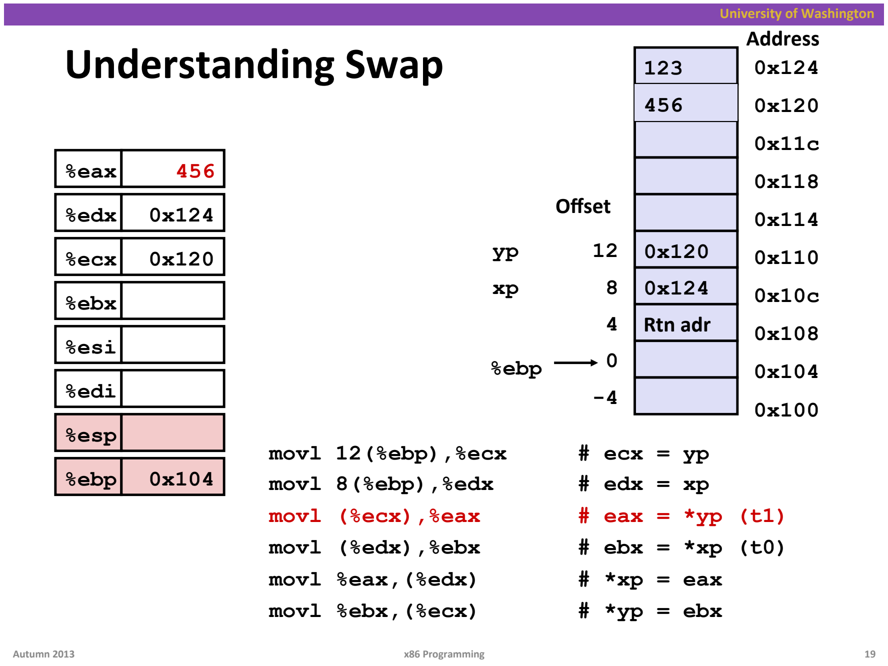


Understanding Swap

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%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
```

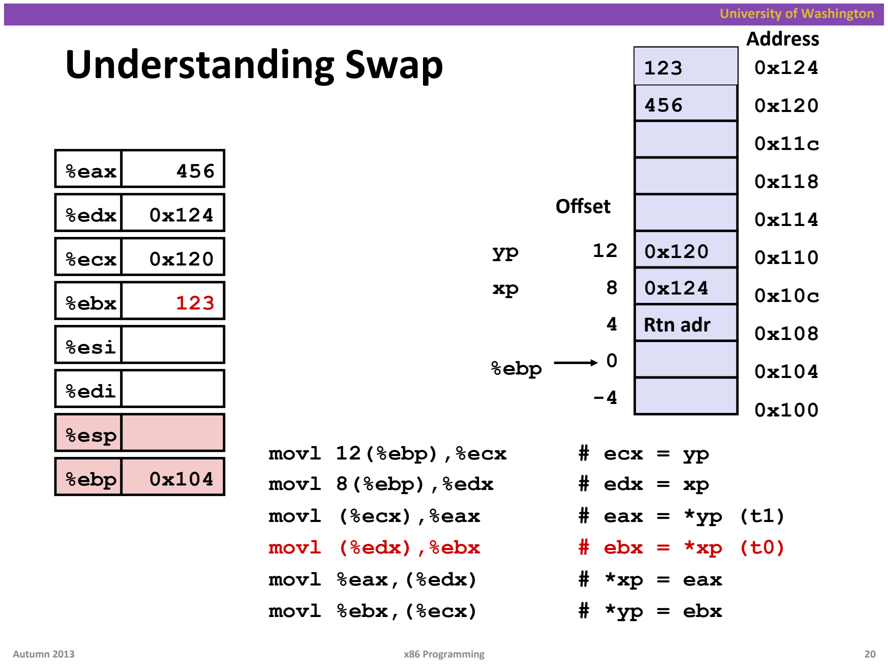


Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%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

%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

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

	Offset		Address
		456	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

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

	Offset		Address
		456	0x124
		123	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
```

x86-64 Integer Registers

64-bits wide

%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, and add 8 new ones; *all* accessible as 8, 16, 32, 64 bits.

32-bit vs. 64-bit operands

- Long word `l` (4 Bytes) ↔ Quad word `q` (8 Bytes)

again, confusing historical terms…
not the current machine word size

- New instruction forms:
  - `movl` → `movq`
  - `addl` → `addq`
  - `sall` → `salq`
  - etc.

- x86-64 can still use 32-bit instructions that generate 32-bit results
  - Higher-order bits of destination register are just set to 0
  - Example: `addl`

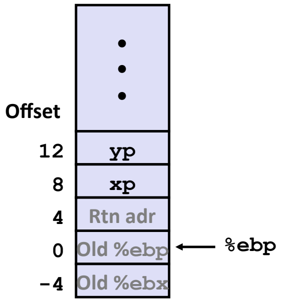
Swap Ints in 32-bit Mode

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

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

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

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



Swap Ints 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
```

- **Arguments passed in registers (why useful?)**
  - First (**xp**) in **%rdi**, second (**yp**) in **%rsi**
  - 64-bit pointers
- **No stack operations required: faster**
- **32-bit data**
  - Data held in registers **%eax** and **%edx**
  - **movl** operation (the **l** refers to data width, not address width)

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

- **Remember, the addresses used for accessing memory in mov (and other) instructions can be computed in several different ways**
- **Most General Form:**

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

  - D: Constant "displacement" value represented in 1, 2, or 4 bytes
  - Rb: Base register: Any of the 8/16 integer registers
  - Ri: Index register: Any, except for %esp or %rsp; **%ebp unlikely**
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)
- **Special Cases: can use any combination of D, Rb, Ri and S**

(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]]**	**(S=1, D=0)**
D(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]+D]**	**(S=1)**
(Rb,Ri,S)	**Mem[Reg[Rb]+S*Reg[Ri]]**	**(D=0)**

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x100`

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

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

Address Computation Instruction

- `leal` *Src,Dest*
  - *Src* is address mode expression
  - Set *Dest* to address computed by expression
    - (lea stands for *load effective address)*
  - Example: `leal (%edx,%ecx,4), %eax`
- **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 (Binary) Instructions:**

Format	*Computation*	
`addl` *Src,Dest*	*Dest* = *Dest* + *Src*	
`subl` *Src,Dest*	*Dest* = *Dest* − *Src*	
`imull` *Src,Dest*	*Dest = Dest * Src*	
`shll` *Src,Dest*	*Dest = Dest << Src*	*Also called sall*
`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! (especially `subl`)**
- **No distinction between signed and unsigned int (why?)**
  - except arithmetic vs. logical shift right

Some Arithmetic Operations

- **One Operand (Unary) Instructions**

`incl` *Dest*	*Dest* = *Dest* + 1	increment
`decl` *Dest*	*Dest* = *Dest* − 1	decrement
`negl` *Dest*	*Dest* = −*Dest*	*negate*
`notl` *Dest*	*Dest* = ~*Dest*	*bitwise complement*

- **See textbook section 3.5.5 for more instructions: `mull`, `cltd`, `idivl`, `divl`**

Using `leal` for Arithmetic Expressions (IA32)

```
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;
}
```

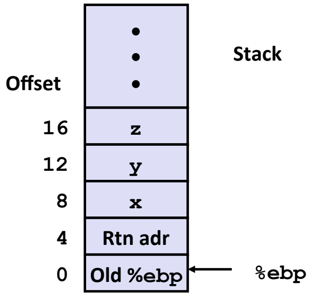
```
arith:
   pushl %ebp                       Set
   movl %esp,%ebp                   Up

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

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

Understanding `arith` (IA32)

```
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;
}
```



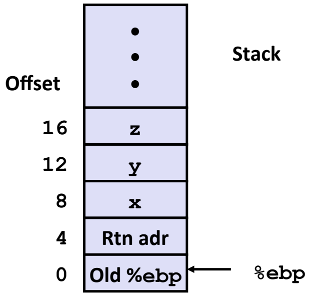
Stack

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

```
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 = y + 2*y = 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` (IA32)

```
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;
}
```



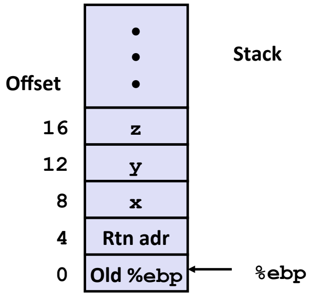
Stack

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

```
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 = y + 2*y = 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` (IA32)

```
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;
}
```



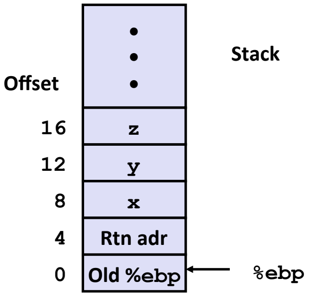
Stack

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

```
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 = y + 2*y = 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` (IA32)

```
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;
}
```



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

```
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 = y + 2*y = 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)
```

Observations about `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;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
- `(x+y+z)*(x+4+48*y)`

```
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 = y + 2*y = 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 (IA32)

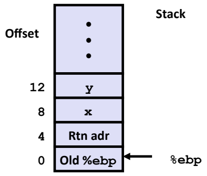
```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

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

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



Offset	Stack
	• • •
12	y
8	x
4	Rtn adr
0	Old %ebp ← %ebp

Another Example (IA32)

```
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              }
    andl $8185,%eax            } Body

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

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

Another Example (IA32)

```
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           }
   andl  $8185,%eax         } Body

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

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

Another Example (IA32)

```
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

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

   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      (t1)
sarl  $17,%eax          eax = t1>>17   (t2)
andl  $8185,%eax        eax = t2 & 8185
```

compiler optimization

Topics: control flow

- Condition codes
- Conditional and unconditional branches
- Loops

Conditionals and Control Flow

- **A 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 implement some related control flow constructs**
  - break, continue
- **In x86, we'll refer to branches as "jumps" (either conditional or unconditional)**

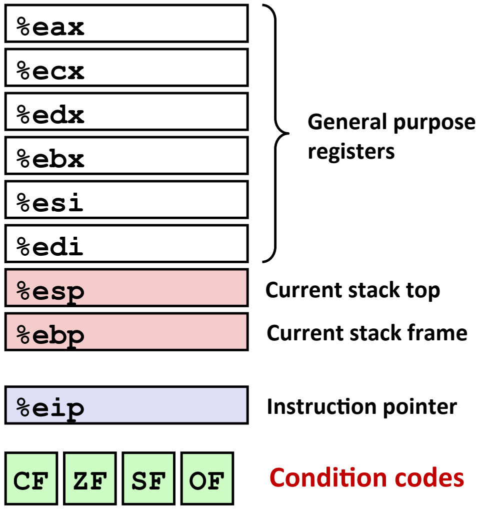
Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes
  - Takes address as argument

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



`%eax`
`%ecx`
`%edx`
`%ebx`
`%esi`
`%edi`
General purpose registers

`%esp` **Current stack top**

`%ebp` **Current stack frame**

`%eip` **Instruction pointer**

`CF` `ZF` `SF` `OF` **Condition codes**

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 (beware!)**
- **Full documentation (IA32): http://www.jegerlehner.ch/intel/IntelCodeTable.pdf**

Condition Codes (Explicit Setting: Compare)

- **Single-bit registers**

  `CF` Carry Flag (for unsigned) `SF` Sign Flag (for signed)

  `ZF` Zero Flag `OF` Overflow Flag (for signed)

- **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)

- **Single-bit registers**

CF	Carry Flag (for unsigned)	**SF**	Sign Flag (for signed)
ZF	Zero Flag	**OF**	Overflow Flag (for signed)

- **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 if **a&b == 0**
  - SF set if **a&b < 0**

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

Reading Condition Codes

- **SetX Instructions**
  - Set a single byte to 0 or 1 based on combinations of condition codes

SetX	Condition	Description
sete	**ZF**	Equal / Zero
setne	**~ZF**	Not Equal / Not Zero
sets	**SF**	Negative
setns	**~SF**	Nonnegative
setg	**~(SF^OF)&~ZF**	Greater (Signed)
setge	**~(SF^OF)**	Greater or Equal (Signed)
setl	**(SF^OF)**	Less (Signed)
setle	**(SF^OF)|ZF**	Less or Equal (Signed)
seta	**~CF&~ZF**	Above (unsigned)
setb	**CF**	Below (unsigned)

Reading Condition Codes (Cont.)

- **SetX Instructions:**
  Set single byte to 0 or 1 based on combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use **movzbl** to finish job

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

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

Body: y at 12(%ebp), x at 8(%ebp)

```
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 to 0 or 1 based on combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use **movzbl** to finish job

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

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

Body: y at 12(%ebp), x at 8(%ebp)

```
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
  - Takes address as argument

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`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                 } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx        } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave                        } Finish
    ret
.L7:
    subl    %edx, %eax           } Body2
    jmp     .L8
```

Conditional Branch Example (Cont.)

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

```
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

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
```

`int x`	`%edx`
`int y`	`%eax`

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;
}
```

int x	%edx
int y	%eax

```
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;
}
```

int x	%edx
int y	%eax

```
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;
}
```

int x	%edx
int y	%eax

```
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;
}
```

int x	%edx
int y	%eax

```
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;
```

```
if (Test)
  val = Then-Expr;
else
  val = Else-Expr;
```

```
result = 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 might you make this more 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**
  - **cmov*C*** src, dest
  - Move value from src to dest if condition *C* holds
  - *Why is this good?*

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**
  - **cmov*C*** 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 <u>relocatable</u>**
- **Absolute branches are not**

Compiling Loops

C/Java code:

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

Machine code:

```
loopTop:    cmpl  $0, %eax
            je    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

“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

Registers:

`%edx`	`x`
`%eax`	`result`

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
```

Translation?

“Do-While” Loop Compilation

Registers:

`%edx`	`x`
`%eax`	`result`

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
```

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
```

"For" Loop Example: Square-and-Multiply

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

x^m * x^n = x^{m+n}

0 … 0 1 1 0 1 = 13

1$^{2\text{^}31}$ * … * 1^{16} * x^8 * x^4 * 1^2 * x^1 = x^{13}

1 = x^0 x = x^1

- **Algorithm**
  - Exploit bit representation: $p = p_0 + 2p_1 + 2^2p_2 + \ldots\ 2^{n-1}p_{n-1}$
  - Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot \underbrace{(\ldots((z_{n-1}{}^2)^2)\ldots)^2}_{n-1 \text{ times}}$

    $z_i = 1$ when $p_i = 0$

    $z_i = x$ when $p_i = 1$
  - Complexity $\mathrm{O}(\log p) = \mathrm{O}(\mathrm{sizeof}(p))$

Example

3^{10} = 3^2 * 3^8

= 3^2 * ((3^2)^2)^2

`ipwr` Computation

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int 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 (Initialize; Test; Update)
    Body
```

Init

```
result = 1
```

Test

```
p != 0
```

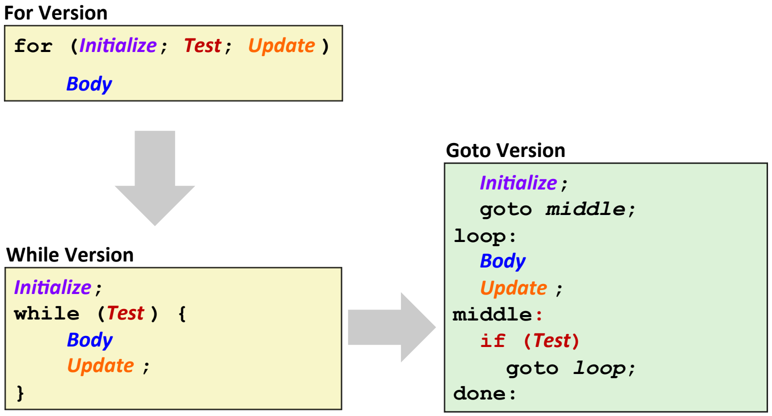
Update

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

Body

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

"For"→ "While"



For Version

```
for (Initialize; Test; Update )

      Body
```

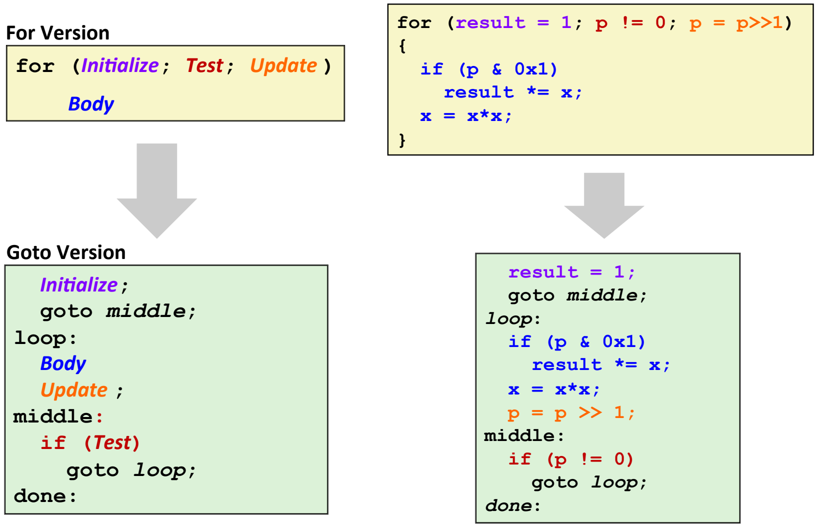
While Version

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

Goto Version

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

For-Loop: Compilation



For Version

```
for (Initialize; Test; Update )

      Body
```

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

Goto Version

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

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

Switch Statement Example

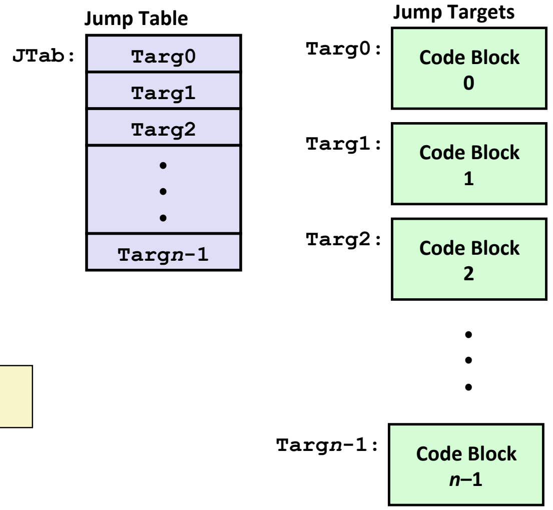
```
long switch_eg (unsigned
   long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default: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

- 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
}
```



Approximate Translation

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

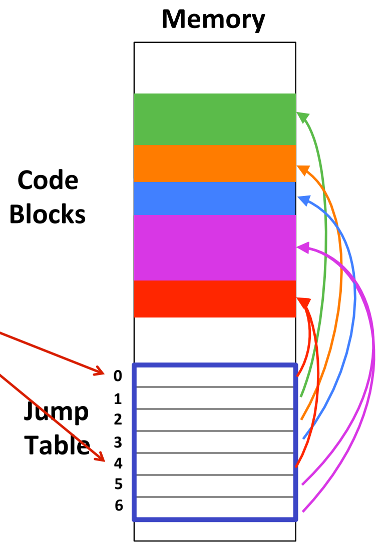
Jump Table Structure

C code:

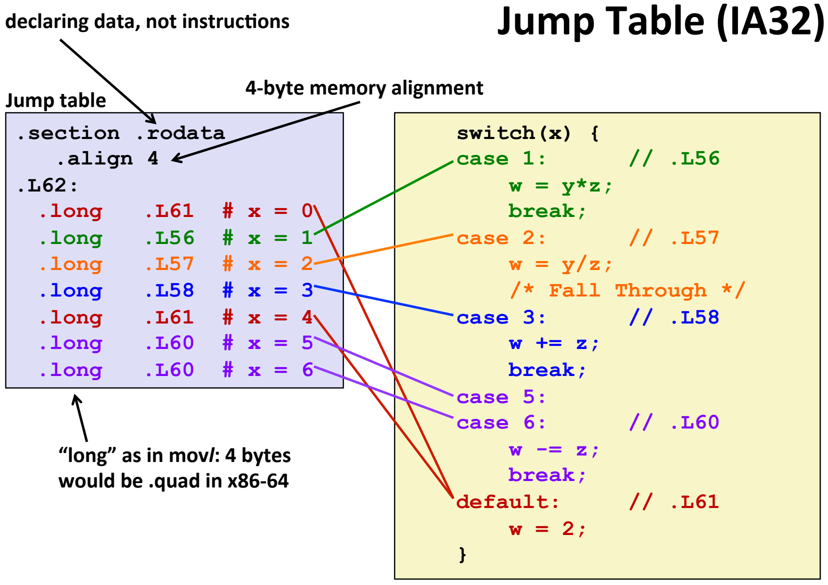
```
switch(x) {
  case 1: <some code>
          break;
  case 2: <some code>
  case 3: <some code>
          break;
  case 5:
  case 6: <some code>
          break;
  default: <some code>
}
```

We can use the jump table when x <= 6:

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



Jump Table (IA32)



```
.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(unsigned 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
    ja    .L61
    jmp   *.L62(,%edx,4)
```

Translation?

Switch Statement Example (IA32)

```
long switch_eg(unsigned 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 > 6 goto default
    jmp   *.L62(,%edx,4)   # goto JTab[x]
```

jump above (like jg, but unsigned)

Indirect jump

Assembly Setup Explanation (IA32)

- **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: different address modes for target**

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`
  - `target = JTab[x]; goto target;` (only for 0 ≤ **x** ≤ 6)

Code Blocks (Partial)

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

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   jmp .L63
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd               # Div prep
   idivl %ecx         # y/z
   movl  %eax, %ebx  # w = y/z
# Fall through – no jmp
.L58:  // Case 3:
   addl  %ecx, %ebx  # w+= z
   jmp .L63
...
.L63
   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;
  . . .
}
return w;
```

```
.L60: // Cases 5&6:
   subl  %ecx, %ebx  # w -= z
   jmp .L63
.L56: // Case 1:
   movl  12(%ebp), %ebx # w = y
   imull %ecx, %ebx     # w*= z
   jmp .L63


...
.L63
   movl  %ebx, %eax  # return w
   popl  %ebx
   leave
   ret
```

Code Blocks (Partial, return inlined)

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

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

```
.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
   cltd               # Div prep
   idivl %ecx         # y/z
   movl  %eax, %ebx # w = y/z
# Fall through – no jmp
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```

Code Blocks (Rest, return inlined)

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

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

```
.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
```

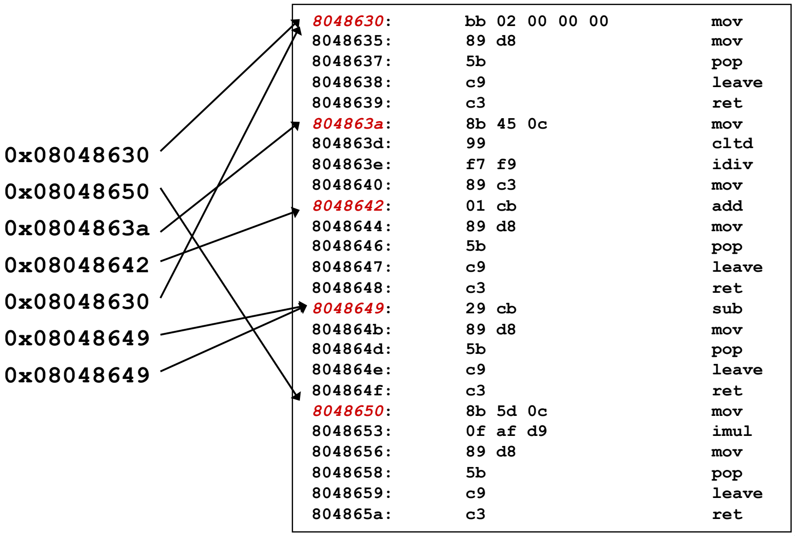
  - Examine 7 hexadecimal format “words” (4-bytes each)
  - Use command “**help x**” to get format documentation

```
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                  cltd
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
0x08048650
0x0804863a
0x08048642
0x08048630
0x08048649
0x08048649
```

```
8048630:     bb 02 00 00 00      mov
8048635:     89 d8               mov
8048637:     5b                  pop
8048638:     c9                  leave
8048639:     c3                  ret
804863a:     8b 45 0c            mov
804863d:     99                  cltd
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 52001 entries for only 4 cases (too big)
  - about 200KB = 200,000 bytes
  - text of this switch statement = about 200 bytes

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
  - x86-64: better support for passing function arguments in registers

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

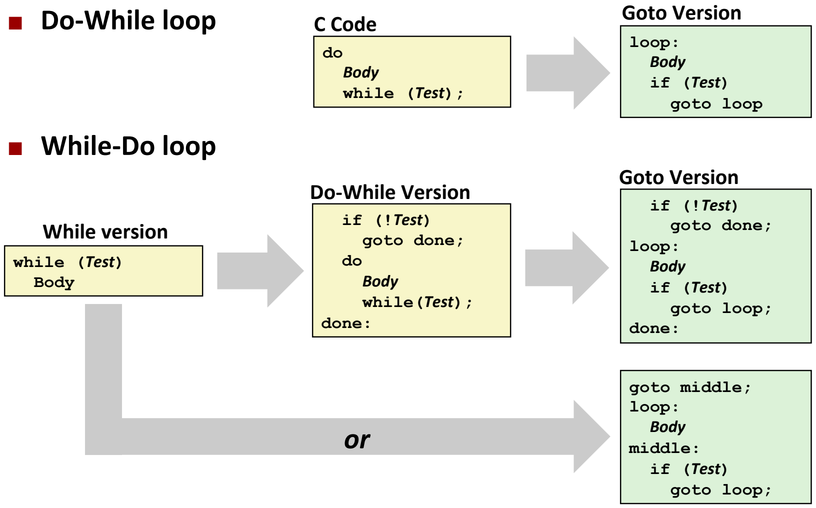
- Complete memory addressing mode
  - `(%eax), 17(%eax), 2(%ebx, %ecx, 8),` …
- Immediate (constant), Register, and Memory Operands
  - `subl %eax, %ecx          # ecx = ecx + eax`
  - `sall $4,%edx             # edx = edx << 4`
  - `addl 16(%ebp),%ecx       # ecx = ecx + Mem[16+ebp]`
  - `imull %ecx,%eax          # eax = eax * ecx`

Quick Review

- Control
  - 1-bit condition code registers CF ZF SF OF
  - Set as side effect by arithmetic instructions or by `cmp, test`
  - Used:
    - Read out by setx instructions (`setg, setle`, …)
    - Or by conditional jumps (`jle .L4, je .L10,` …)
    - Or by conditional moves (`cmovle %edx, %eax`)
- Arithmetic operations also set condition codes
  - subl, addl, imull, shrl, etc.
- Load Effective Address does NOT set condition codes
  - `leal 4(%edx,%eax),%eax   # eax = 4 + edx + eax`

Quick Review

- Do-While loop
- While-Do loop



C Code

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

Goto Version

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

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

Control Flow Summary

- 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 textbook)
- Conditions in CISC
  - CISC machines generally have condition code registers