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
```

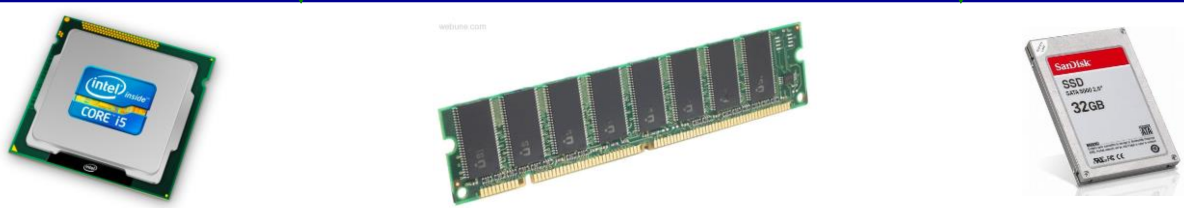
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:



Next x86 topics

- x86 basics: registers
- Move instructions, registers, and operands
- Memory addressing modes
- `swap` example
- 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

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`

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Some History: IA32 Registers – 32 bits wide

Assembly Data Types

- **"Integer" data of 1, 2, 4, or 8 bytes**
  - Data values
  - Addresses (untyped pointers)

- **Floating point data of 4, 8, or 10 bytes**

- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

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

- **Moving Data**

  `movq` ***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 16 integer registers
    - Example: **`%rax, %r13`**
    - But **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other "address modes"

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

`movq, movl, movw, movb`

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

  - **`movq`** ***Source, Dest*:**

    Move 8-byte "quad word"
  - **`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**

`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`var_a = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p_a = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`var_d = var_a;`
		Mem	`movq %rax,(%rdx)`	`*p_d = var_a;`
	Mem	*Reg*	`movq (%rax),%rdx`	`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
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **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

```
movq 8(%rbp),%rdx
```

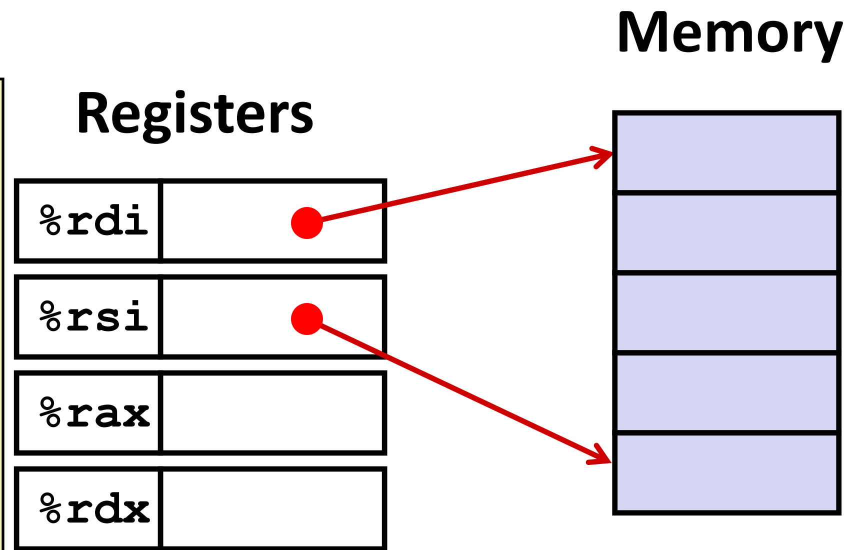
Example of Basic Addressing Modes

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

```
swap:
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

Understanding Swap()

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



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()

Registers

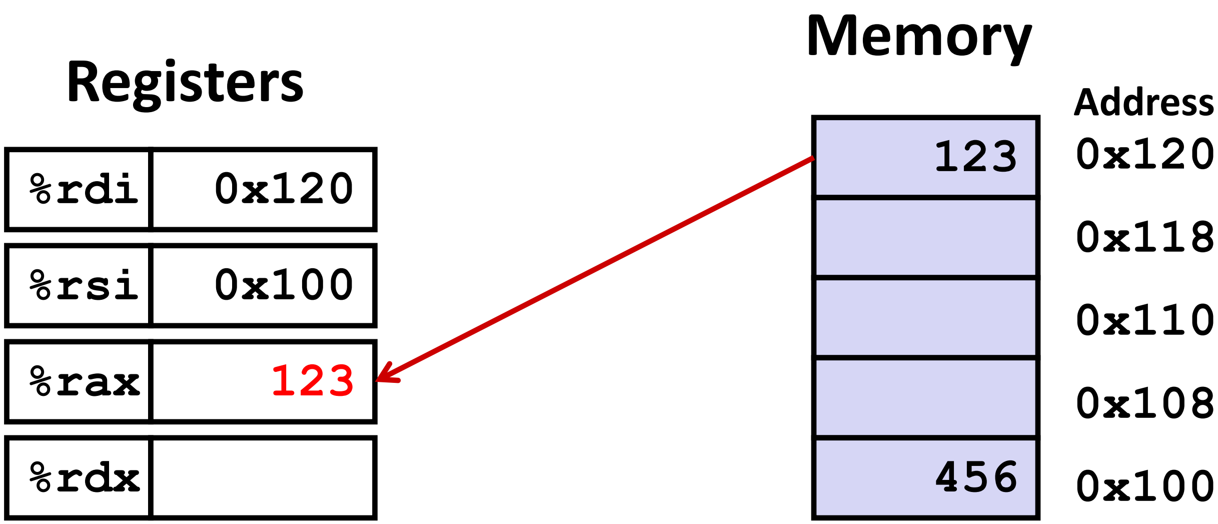
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

	Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

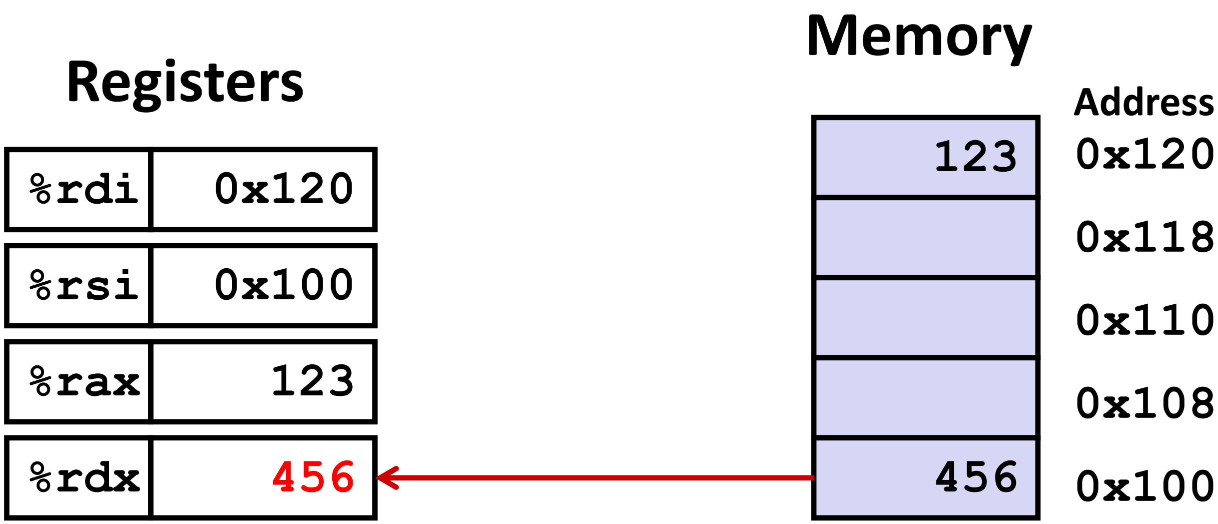
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



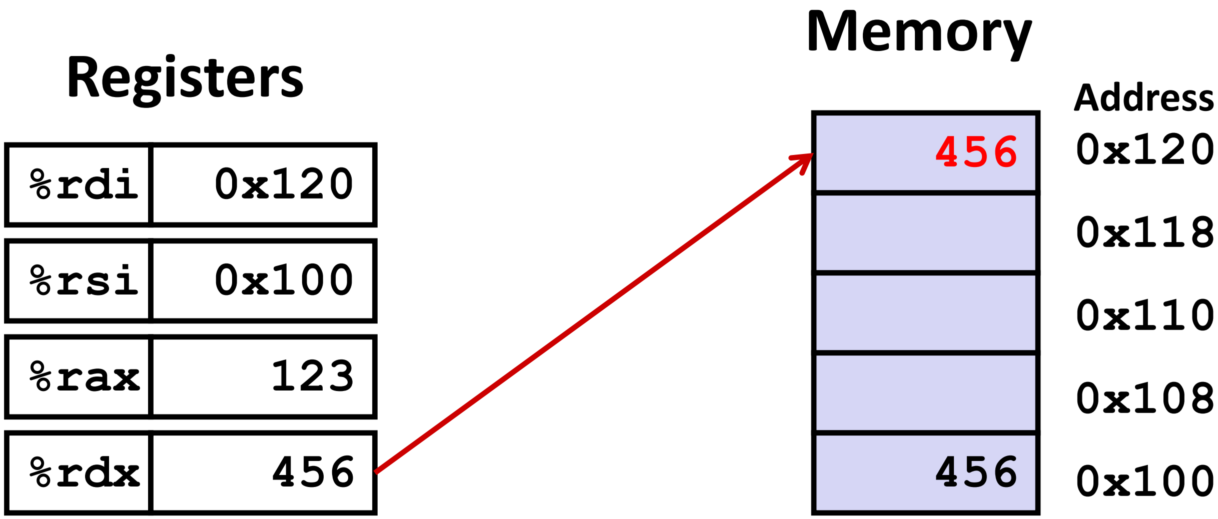

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



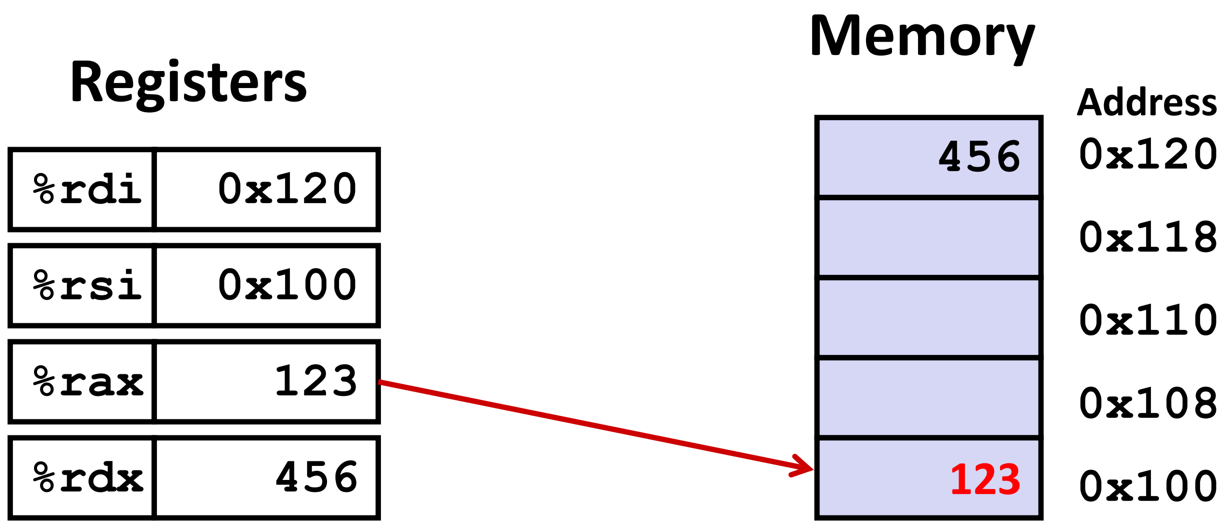

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

Memory Addressing Modes: Basic

- **Indirect (R) Mem[Reg[R]]**
  - Register R specifies the memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **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

```
movq 8(%rbp),%rdx
```

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 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - 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

`%rdx`	`0xf000`
`%rcx`	`0x0100`

(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(%rdx)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

(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(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Address Computation Instruction

- `leaq` *Src*,*Dest*
  - *Src* is address expression (Any of the formats we just discussed!)
  - *Dest* is a register
  - Set *Dest* to address computed by expression
    - (lea stands for *load effective address)*
  - Example: `leaq (%rdx,%rcx,4), %rax`

- **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

`leaq` vs. `movq` example

Registers

`%rax`	
`%rbx`	
`%rcx`	`0x4`
`%rdx`	`0x100`
`%rdi`	
`%rsi`	

Memory

	Address
0x400	`0x120`
0xf	`0x118`
0x8	`0x110`
0x10	`0x108`
0x1	`0x100`

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

`leaq` vs. `movq` example (solution)

Registers

`%rax`	`0x110`
`%rbx`	`0x8`
`%rcx`	`0x4`
`%rdx`	`0x100`
`%rdi`	`0x100`
`%rsi`	`0x1`

Memory

	Address
0x400	`0x120`
0xf	`0x118`
0x8	`0x110`
0x10	`0x108`
0x1	`0x100`

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

Some Arithmetic Operations

- **Two Operand (Binary) Instructions:**

Format		*Computation*	
`addq`	*Src,Dest*	*Dest = Dest + Src*	
`subq`	*Src,Dest*	*Dest = Dest − Src*	
`imulq`	*Src,Dest*	*Dest = Dest * Src*	
`shlq`	*Src,Dest*	*Dest = Dest << Src*	***Also called salq***
`sarq`	*Src,Dest*	*Dest = Dest >> Src*	***Arithmetic***
`shrq`	*Src,Dest*	*Dest = Dest >> Src*	***Logical***
`xorq`	*Src,Dest*	*Dest = Dest ^ Src*	
`andq`	*Src,Dest*	*Dest = Dest & Src*	
`orq`	*Src,Dest*	*Dest = Dest	Src*	

- **Watch out for argument order! (especially `subq`)**
- **No distinction between signed and unsigned int (why?)**
  - except arithmetic vs. logical shift right

Some Arithmetic Operations

- **One Operand (Unary) Instructions**

`incq` *Dest*	*Dest* = *Dest* + 1	increment
`decq` *Dest*	*Dest* = *Dest* − 1	decrement
`negq` *Dest*	*Dest* = −*Dest*	*negate*
`notq` *Dest*	*Dest* = ~*Dest*	*bitwise complement*

- **See textbook section 3.5.5 for more instructions: `mulq`, `cqto`, `idivq`, `divq`**

The `leaq` Instruction

- "lea" stands for *load effective address*
- Example: `leaq (%rdx,%rcx,4), %rax`

Does the `leaq` instruction go to memory?

NO

"`leaq` – it just does math"

Using `leaq` for Arithmetic Expressions

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

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once instead of twice

Understanding `arith`

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

```
arith:
  leaq    (%rdi,%rsi), %rax     # t1
  addq    %rdx, %rax            # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx              # t4
  leaq    4(%rdi,%rdx), %rcx    # t5
  imulq   %rcx, %rax            # rval
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

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)**

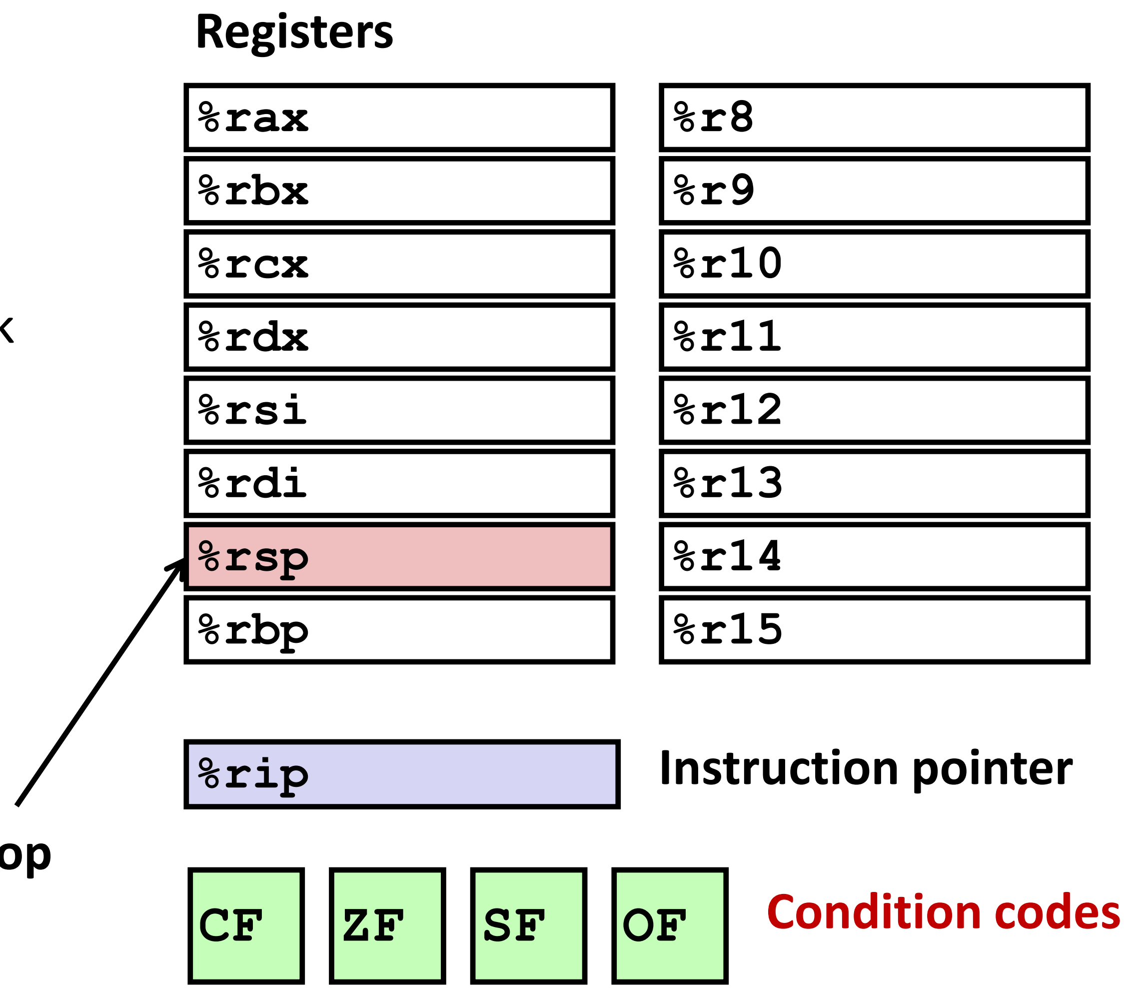
Jumping

- **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 (x86-64, Partial)

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



Current stack top

Condition Codes (Implicit Setting)

- **Implicitly set by arithmetic operations**
  - (think of it as side effect)

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

Single bit registers

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

- **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 `leaq` instruction (beware!)

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**

  `cmpq` *Src2,Src1*

  `cmpq b,a` like computing `a-b` without setting destination

Single bit registers

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

- **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**

  `testq` *Src2,Src1*

  `testq 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

Single bit registers

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

- ZF set if `a&b == 0`
- SF set if `a&b < 0`

- `testq %rax, %rax`
  - Sets SF and ZF, check if rax is +,0,-

Reading Condition Codes

- **SetX Instructions**
  - Set a low-order byte to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

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)

x86-64 Integer Registers

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

- Can reference low-order byte

Reading Condition Codes (Cont.)

- **`SetX` Instructions:**
  - Set single byte to 0 or 1 based on combination of condition codes
  - Operand is one of the byte registers (eg. **`al`**, **`dl`**) or a byte in memory
- **`Set` instruction does not alter remaining bytes in register**
  - Typically use **`movzbl`** to finish job - Sets upper 32 bits to zero
    - Aside: In x86-64, any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0.

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

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
cmpq   %rsi, %rdi
setg   %al
movzbl %al, %eax
ret
```

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
  - Operand is one of the byte registers (eg. **`al`**, **`dl`**) or a byte in memory
- **`Set` instruction does not alter remaining bytes in register**
  - Typically use **`movzbl`** to finish job - Sets upper 32 bits to zero
    - Aside: In x86-64, any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0.

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

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
    cmpq   %rsi, %rdi    # Compare x:y
    setg   %al           # al = x > y
    movzbl %al, %eax     # Zero rest of %rax
    ret
```

Aside: `movz` and `movs` examples

`movzbl` *Src, RegisterDest* — *Move with zero extension*

`movsbl` *Src, RegisterDest* — *Move with sign extension*

- For use when copying a smaller source value to a larger destination
- Source can be memory or register; Destination must be a register

`movz`<u>***SD***</u> – fills out remaining bytes of the destination with zeroes

`movs`<u>***SD***</u> – fills out remaining bytes of the destination by sign extension, replicating the most significant bit of the source

<u>***S***</u> – can be b=byte, w=16-bit word

<u>***D***</u> – can be w=16-bit word, l=32-bit long word, q=64-bit quad word

- **Note**: In x86-64, <u>***any instruction***</u> that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0.
- Good example in the "Aside" on p. 184 in 3e CS-APP (our text)

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 (Old Style)

- Generation

```
gcc –Og -S –fno-if-conversion control.c
```

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

```
absdiff:
   cmpq    %rsi, %rdi   # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:               # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rax`	Return value

Expressing with Goto Code

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

```
long absdiff_j
   (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

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

General Conditional Expression Translation (Using Branches)

C Code

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

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

Example:

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

Goto Version

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

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

Using Conditional Moves

- **Conditional Move Instructions**
  - **`cmovC`** src, dest
  - Move value from src to dest if condition *C* holds
  - Instruction supports:

    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be **safe**
- **Why is this useful?**
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

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

Goto Version

```
  result = Then_Expr;
  else_val = Else_Expr;
  nt = !Test;
  if (nt) result = else_val;
  return result;
```

Conditional Move Example

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

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rax`	Return value

```
absdiff:
   movq    %rdi, %rax   # x
   subq    %rsi, %rax   # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx   # else_val = y-x
   cmpq    %rsi, %rdi   # x:y
   cmovle  %rdx, %rax   # if <=, result = else_val
   ret
```

Bad Cases for Conditional Move

Expensive Computations

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

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

Risky Computations

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

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

Computations with side effects

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

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

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

```
long pcount_do(unsigned long x)
{
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto(unsigned long x)
{
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

- Count number of 1's in argument **x** ("popcount")
- Use backward branch to continue looping
- Only take branch when "while" condition holds

Do-While Loop Compilation

Goto Version

```
long pcount_goto(unsigned long x)
{
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
    movl    $0, %eax     #  result = 0
.L2:                     # loop:
    movq    %rdi, %rdx
    andl    $1, %edx     #  t = x & 0x1
    addq    %rdx, %rax   #  result += t
    shrq    %rdi         #  x >>= 1
    jne     .L2          #  if (x) goto loop
    rep; ret
```

General Do-While Loop 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

General <u>While Loop</u> - Translation #1

- "Jump-to-middle" translation
- Used with –Og

While version

```
while (Test)
  Body
```

Goto Version

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

While Loop Example – Translation #1

C Code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
  (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- Used with –Og
- Compare to do-while version of function
- Initial goto starts loop at test

General While Loop - Translation #2

While version

```
while (Test)
   Body
```

- "Do-while" conversion
- Used with –O1

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:
```

While Loop Example – Translation #2

C Code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Do-While Version

```
long pcount_goto_dw
  (unsigned long x) {
  long result = 0;
  if (!x) goto done;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
 done:
  return result;
}
```

- Used with –O1
- Compare to do-while version of function
- Initial conditional guards entrance to loop

For Loop Form

General Form

```
for (Init; Test; Update )

    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

For Loop → While Loop

For Version

```
for (Init; Test; Update)

      Body
```

While Version

```
Init;

while (Test) {

      Body

      Update;

}
```

For Loop-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

For Loop Do-While Conversion

C Code

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

- Initial test can be optimized away

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                      Init
  if (!(i < WSIZE))           ! Test
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;         Body
    result += bit;
  }
  i++;                        Update
  if (i < WSIZE)              Test
    goto loop;
 done:
  return result;
}
```

```
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

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
• • •
Targ*n*-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targ*n*-1: Code Block *n*–1

Approximate Translation

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

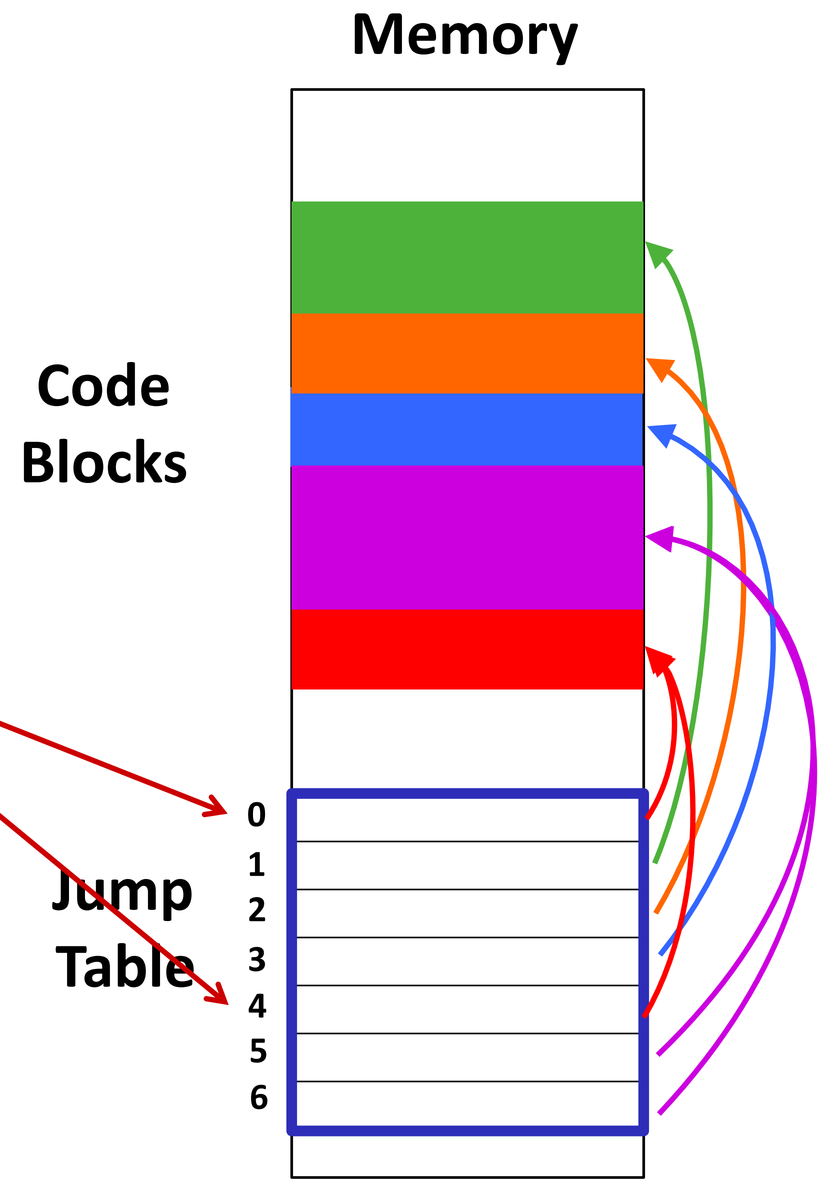
Jump Table Structure

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;
```



Switch Statement Example

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

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

What range of values takes default?

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	Return value

Note that **w** not initialized here

Switch Statement Example

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

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Setup:

jump above (like jg, but unsigned)

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump

Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes
  - Base address at `.L4`

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

- **Jumping**
  - **Direct:** `jmp .L8`
  - Jump target is denoted by label `.L8`
  - **Indirect:** `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address `.L4 + x*8`
    - Only for $0 \leq$ **x** ≤ 6

Jump Table

declaring data, not instructions

8-byte memory alignment

Jump table

```
.section .rodata
  .align 8
.L4:
  .quad   .L8 # x = 0
  .quad   .L3 # x = 1
  .quad   .L5 # x = 2
  .quad   .L9 # x = 3
  .quad   .L8 # x = 4
  .quad   .L7 # x = 5
  .quad   .L7 # x = 6
```

```
switch(x) {
case 1:         // .L3
    w = y*z;
    break;
case 2:         // .L5
    w = y/z;
    /* Fall Through */
case 3:         // .L9
    w += z;
    break;
case 5:
case 6:         // .L7
    w -= z;
    break;
default:        // .L8
    w = 2;
}
```

Code Blocks (x == 1)

```
  switch(x) {
  case 1:       // .L3
        w = y*z;
        break;
   . . .
}
```

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	Return value

Handling Fall-Through

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:   // .L5
    w = y/z;
    /* Fall Through */
case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
        w = 1;

merge:
        w += z;
```

Code Blocks (x == 2, x == 3)

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:   // .L5
    w = y/z;
    /* Fall Through */
case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
.L5:                      # Case 2
   movq    %rsi, %rax #   y in rax
   cqto                   #   Div prep
   idivq   %rcx           #   y/z
   jmp     .L6            #   goto merge
.L9:                      # Case 3
   movl    $1, %eax       #   w = 1
.L6:                      # merge:
   addq    %rcx, %rax #   w += z
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

Code Blocks (x == 5, x == 6, default)

```
switch(x) {
    . . .
    case 5:   // .L7
    case 6:   // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                      # Case 5,6
  movl   $1, %eax         #  w = 1
  subq   %rdx, %rax #  w -= z
  ret
.L8:                      # Default:
  movl   $2, %eax         #  2
  ret
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

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