
WELCOME
TO
15-213
18-213
15-513

Quiz

DEVICES AWAY.
UNTIL EVERYONE IS DONE.

q2: yes all numbers are decimal

AI, bringing pencil and
paper back since 2024

```
int x = 27;
x += (~x) + 15
```

```
int x = 27;
x += (~x + 1) + 14
```

```
int x = 27;
x += -x + 14
```

```
x = 14
```

```
rdx = 5000
rax = 67
rdi = 16
mov %rax, 22(%rdx, %rdi, 2)

mov D(Rb, Ri, S) →
   *(D + Rb + Ri * S)

22 + 5000 + 16 * 2 → 5054
```

Previous lecture question followup

Gnu assembly == AT&T syntax

Alternative is **Intel** syntax

You *can* address low-order 32 and
16 bits of r8-r15: `r8d`, `r8w`

but *not* the ‘ah’, ‘al’ 8 bit chunks you
can with rax

Activities are posted on the website (we won't have time today, but you can do after lecture)

On shark machines:

```
wget http://www.cs.cmu.edu/~213/activities/machine-control.pdf
wget http://www.cs.cmu.edu/~213/activities/machine-control.tar
tar xf machine-control.tar
cd machine-control
```

Today’s slides differ from the ones I posted before class

(quiz solution…)

But content overall is the same.

Some review from last time, for your records

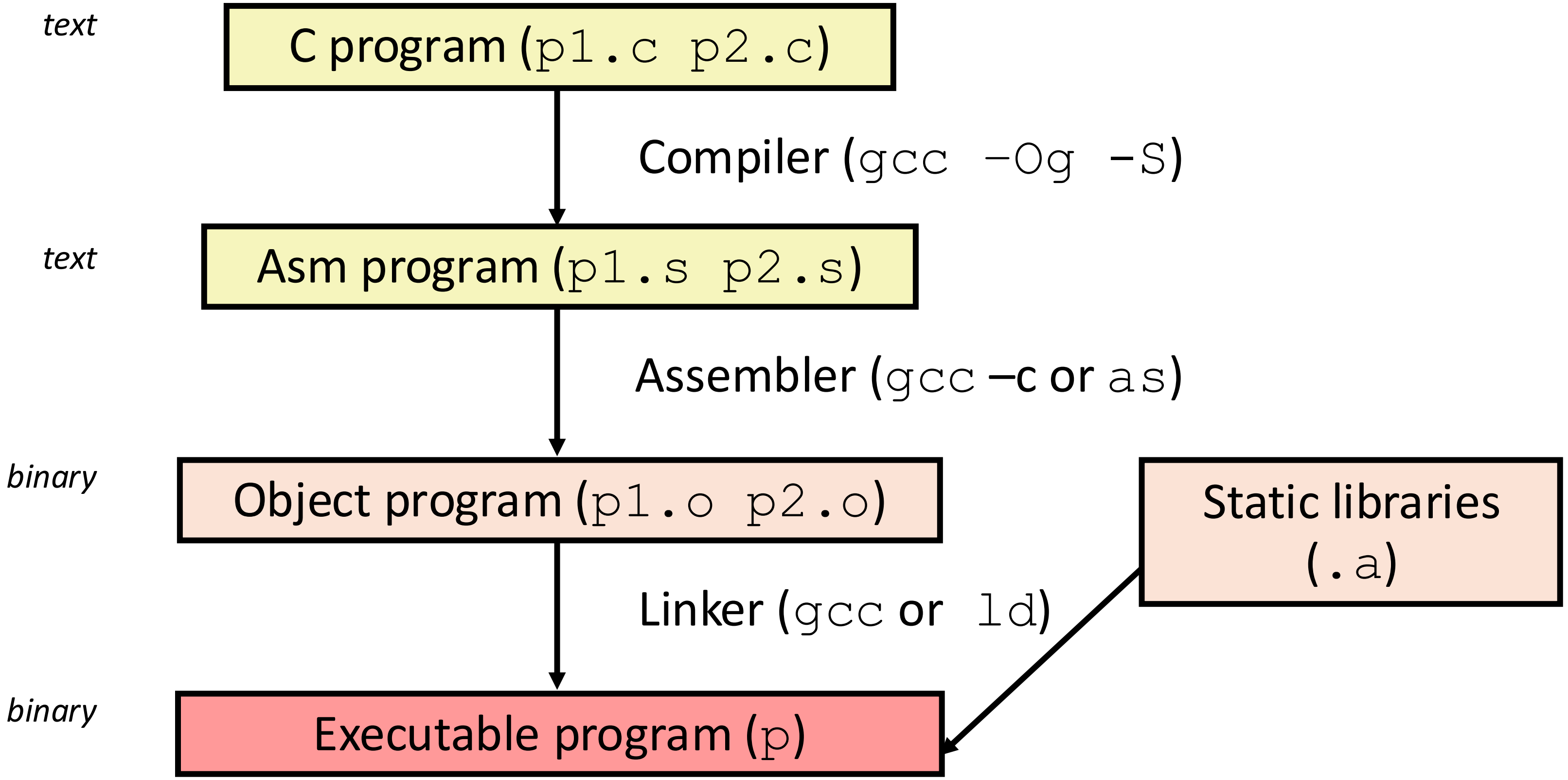
- Programs can do arithmetic on the same register!

```
imul %rax, %rax
```

- C translation to assembly
- An x86 program's view
- Instruction register destinations
- Addressing modes
- `lea`

C code is translated into assembly code by a compiler (ex: `gcc`).

Compile **`p1.c`** and **`p2.c`** with command: **`gcc –Og p1.c p2.c -o p`**



The specification for that assembly code is defined by the instruction set architecture (ISA).

The ISA we learn in this class is x86-64.

Assembly code is a plain text version of what will eventually be object code.

Example:

`gcc -Og –S sum.c`

```
sumstore:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

The specification for that assembly code is defined by the instruction set architecture (ISA).

The ISA we learn in this class is x86-64.

Assembly code is a plain text version of what will eventually be object code.

Example:

```
gcc -Og –S sum.c
```

```
sumstore:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

Object code can be disassembled into assembly using a disassembler (`objdump -d` or `gdb`).

```
>> gdb sum
(gdb) disas sumstore
Dump of assembler code for function sumstore:
   0x0000000000001144 <+4>:     push   %rbp
   0x0000000000001145 <+5>:     mov    %rsp,%rbp
   0x0000000000001148 <+8>:     sub    $0x28,%rsp
   0x000000000000114c <+12>:    mov    %rdi,-0x18(%rbp)
   0x0000000000001150 <+16>:    mov    %rsi,-0x20(%rbp)
   0x0000000000001154 <+20>:    mov    %rdx,-0x28(%rbp)
   0x0000000000001158 <+24>:    mov    -0x20(%rbp),%rdx
   0x000000000000115c <+28>:    mov    -0x18(%rbp),%rax
```

On these slides you will sometimes see the “gcc compiled” version of assembly code and sometimes the “objump” version of assembly code. (Some points are easier to illustrate with one rather than the other.)

Let’s examine the translation of C to x86-64:

C:

```c
long plus(long x, long y);

void sumstore(long x, long y, long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

x86-64:

```
0000000000001133 <sumstore>:
    1137:   53                  push   %rbx
    1138:   48 89 d3            mov    %rdx,%rbx
    113b:   e8 e9 ff ff ff      call   1129 <plus>
    1140:   48 89 03            mov    %rax,(%rbx)
    1143:   5b                  pop    %rbx
    1144:   c3                  ret
```

An x86-64 program's view…

CPU

Program Counter (PC)

`%rpi`	`0x00ff`

Condition Codes

ZF	0
SF	1
OF	1
CF	0

16 "General purpose" Registers

`%rsp`	`0xff88`
`%rsi`	1
`%rdi`	10
`%rax`	11
`...`	

Memory (Virtual)

…. Memory addresses …..

`0:` `1:` `2:`	**Operating system**
`...` (`%rpi` →)	**Code:** function instructions stored here
	Data: global variables stored here
	Heap: dynamically allocated memory grows as program allocates memory
`max:`	**Stack:** local variables and parameters stored here grows as program calls functions shrinks on return from functions

An x86-64 program's view…

CPU

Program Counter (PC)

%rpi	0x00ff

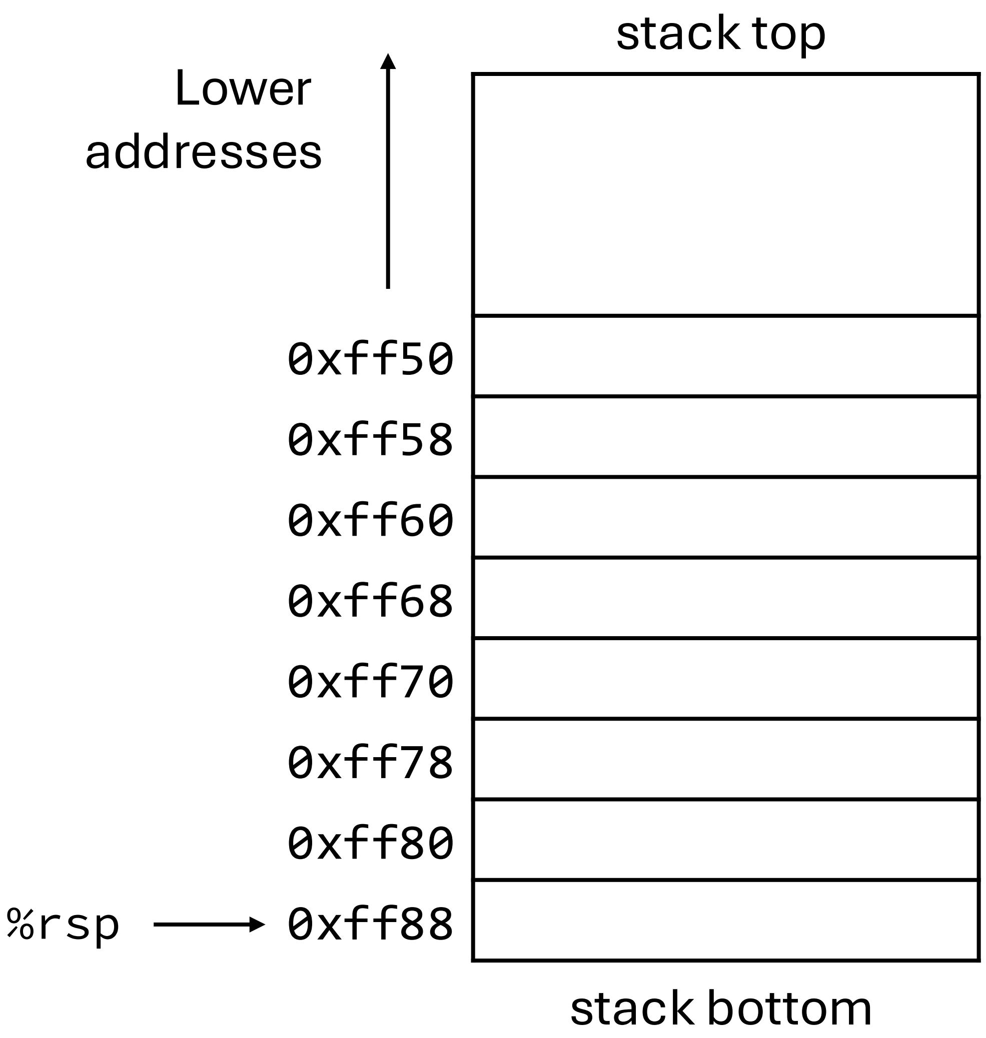
Condition Codes

ZF	0
SF	1
OF	1
CF	0

16 "General purpose" Registers

%rsp	0xff88
%rsi	1
%rdi	10
%rax	11
...	

Stack



Sometimes an instruction may only change portions of the register destination.

63	31	15	7 0
`%rax`	`%eax`	`%ax`	`%al`

Lower order portions of integer registers can be accessed as byte, word (2-byte), double word (4-byte), and quad word (8-byte).

```
movabsq   $0x0011223344556677, %rax        %rax = 0011223344556677
movb      $-1, %al                         %rax = 00112233445566FF
movw      $-1, %ax                         %rax = 001122334455FFFF
movl      $-1, %eax                        %rax = 00000000FFFFFFFF
movq      $-1, %rax                        %rax = FFFFFFFFFFFFFFFF
```

Convention: Any instruction that generates a 32-bit value for a register also sets upper 32 bits to 0.

63	31	15	7	0
%rax	**%eax**	**%ax**	**%al**	

Lower order portions of integer registers can be accessed as byte, word (2-byte), double word (4-byte), and quad word (8-byte).

```
movabsq   $0x0011223344556677, %rax        %rax = 0011223344556677
movb      $-1, %al                         %rax = 00112233445566FF
movw      $-1, %ax                         %rax = 001122334455FFFF
movl      $-1, %eax                        %rax = 00000000FFFFFFFF
movq      $-1, %rax                        %rax = FFFFFFFFFFFFFFFF
```

There are several “addressing modes” that allow the CPU to interact with memory through addresses contained in registers.

Example with `%rsi`, `%rdi`, and `%rax`

General form:

```
D(%rsi, %rdi, S) = Memory[%rsi + %rdi*S + D]
```

Special Cases

```
(%rsi)              Memory[%rsi]
(%rsi, %rdi)        Memory[%rsi + %rdi]
D(%rsi, %rdi)       Memory[%rsi + %rdi + D]
(%rsi, %rdi, S)     Memory[%rsi + %rdi*S]
```

- D is “displacement”, a constant in 1,2, or 4 bytes
- %rsi is a **base register**
  - Could be an of 16 integer registers

%rsi is an “index register”

- Any, except for %rsp

S is scale is 1, 2, 4, 8

Address computation examples

%rdx	**0xf000**
%rcx	**0x0100**

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**

Load effective addressing instruction (`lea`) does math and does not access memory.

Example of instructions that access memory:

Assembly	C equivalent
mov 6(%rbx,%rdi,8), %ax	ax = *(rbx + rdi*8 + 6)
add 6(%rbx,%rdi,8), %ax	ax += *(rbx + rdi*8 + 6)
xor %ax, 6(%rbx,%rdi,8)	*(rbx + rdi*8 + 6) ^= ax

`lea` is special and does not access memory:

Assembly	C equivalent
`lea 6(%rbx,%rdi,8), %rax`	`rax = rbx + rdi*8 + 6`

Why use `lea`?

Compiler authors often use it for ordinary arithmetic

- It can do complex calculations in one instruction
- It's one of the only three-operand instructions the x86 has
- It doesn't touch the condition codes (we'll come back to this)

```c
long m12(long x)
{
  return x*12;
}
```

```
leaq (%rdi,%rdi,2), %rax   # t = x+2*x
salq $2, %rax              # return t<<2
```

Today: How does x86-64 implement C structures that change control flow?

- Condition codes
- Conditional branching
- Loops
- Switch statements (we won't have time for this)

Today: How does x86-64 implement C structures that change control flow?

- **Condition codes**
- Conditional branching
- Loops
- Switch statements (we won’t have time for this)

Every arithmetic and logical operation (except for `lea`) implicitly updates special single-bit registers called "condition codes".

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

GDB prints these as one "eflags" register

`eflags 0x246 [ PF ZF IF ]` *Z set, CSO clear*

CPU

Program Counter (PC)

`%rpi`	`0x00ff`

Condition Codes

`ZF`	`0`
`SF`	`1`
`OF`	`1`
`CF`	`0`

16 "General purpose" Registers

`%rsp`	`0xff88`
`%rsi`	`1`
`%rdi`	`10`
`%rax`	`11`
`...`	

Example:

addq *Src,Dest* `t = a + b`

ZF 1 if `t == 0` (otherwise 0) | 000000000000…00000000000

SF `t < 0` (signed) | 1xxxxxxxxxxxx…xxxxxxxxxxxx

CF (unsigned) `t <` (unsigned) `a`

OF `(a < 0 == b < 0) && (t < 0 != a < 0)`

CF set when unsigned overflow:

```
    1xxxxxxxxxxxxx...
+   1xxxxxxxxxxxxx...
-----------------------
  1 xxxxxxxxxxxxxx...
```

Carry

```
  1 0xxxxxxxxxxxxx...
−   1xxxxxxxxxxxxx...
-----------------------
    1xxxxxxxxxxxxx...
```

Borrow

OF set when signed overflow:

```
  wxxxxxxxxxxxx...
+ yxxxxxxxxxxxx...
------------------
  zxxxxxxxxxxxx...
```

```
w==y && w!=z
```

Example:

addq *Src,Dest* `t = a+b`

ZF 1 if t == 0 (otherwise 0)
SF 1 if t < 0 (as signed)
OF 1 if two's-complement (signed overflow)
CF 1 if carry out from most significant bit (unsigned overflow)

Before `sub` instruction:

```
sub  %rsi, %rax
```

a in `%rsi`

b in `%rax`

CPU

Program Counter (PC)

`%rpi`	`0x00f0`

Condition Codes

ZF	0
SF	0
OF	0
CF	0

16 “General purpose” Registers

`%rsp`	`0xff80`
`%rsi`	a
`%rdi`	0
`%rax`	b
...	

After `sub` instruction:

```
sub  %rsi, %rax
```

a in `%rsi`, b in `%rax`

compute `b-a` and store in `%rax`

CPU

Program Counter (PC)

%rpi	0x00f8

Condition Codes

ZF	**1**
SF	0
OF	0
CF	0

16 “General purpose” Registers

%rsp	0xff80
%rsi	a
%rdi	0
%rax	**b-a**
...	

cmp instruction computes subtraction but does not change second operand.

```
cmp  %rsi, %rax
```

a in `%rsi`, b in `%rax`

computes y-x (no store!)

used to compute
`if ( a < b ) { … }`

CPU

Program Counter (PC)

%rpi	0x00f8

Condition Codes

ZF	**1**
SF	0
OF	0
CF	0

16 “General purpose” Registers

%rsp	0xff80
%rsi	a
%rdi	0
%rax	b
...	

Why use a `cmp` instruction instead of a `sub` instruction to compare two?

`test` instruction computes & but does not change second operand.

```
test %rdi, %rdi
```

z (which equals 0) in `%rdi`

computes z & z (no store!)

only updates ZF and SF!

used to check if `%rdi` is zero

CPU

Program Counter (PC)

`%rpi`	`0x00f8`

Condition Codes

ZF	**1**
SF	**1**
OF	0
CF	0

16 “General purpose” Registers

`%rsp`	`0xff80`
`%rsi`	a
`%rdi`	0
`%rax`	b
...	

`test` instruction computes & but does not change second operand.

```
test %rsi, %rax
```

a in `%rsi`, b in `%rax`

computes a & b (no store!)

used to check if any of the 1-bits in %rax are also set in %rsi (and vice versa)

CPU

Program Counter (PC)

`%rpi`	`0x00f8`

Condition Codes

ZF	**1**
SF	**1**
OF	0
CF	0

16 “General purpose” Registers

`%rsp`	`0xff80`
`%rsi`	a
`%rdi`	0
`%rax`	b
...	

Set instructions read condition codes and set a single byte in the destination.

Instruction	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)
`sete`	`ZF`	Equal / Zero

Jump instructions let programs jump to different parts of code depending on condition codes.

Instruction	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)

Jump instructions let programs jump to different parts of code depending on condition codes.

x86-64 Reference Sheet (GNU assembler format)

Instructions

Data movement

`movq` Src, Dest	Dest = Src
`movsbq` Src,Dest	Dest (quad) = Src (byte), sign-extend
`movzbq` Src,Dest	Dest (quad) = Src (byte), zero-extend

Conditional move

`cmove` Src, Dest	Equal /
`cmovne` Src, Dest	Not equ
`cmovs` Src, Dest	Negativ
`cmovns` Src, Dest	Nonneg
`cmovg` Src, Dest	Greater
`cmovge` Src, Dest	Greater
`cmovl` Src, Dest	Less (si
`cmovle` Src, Dest	Less or
`cmova` Src, Dest	Above (
`cmovae` Src, Dest	Above o
`cmovb` Src, Dest	Below (
`cmovbe` Src, Dest	Below o

Control transfer

`cmpq` Src2, Src1	Sets CCs Src1 Src2
`testq` Src2, Src1	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed ≥)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed ≤)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq` Src	%rsp = %rsp 8, Mem[%rsp] = Src
`popq` Dest	Dest = Mem[%rsp], %rsp = %rsp + 8
`call label`	push address of next instruction, `jmp label`
`ret`	%rip = Mem[%rsp], %rsp = %rsp + 8

Arithmetic operations

`leaq` Src, Dest	Dest = address of Src
`incq` Dest	Dest = Dest + 1
`decq` Dest	Dest = Dest − 1
`addq` Src, Dest	Dest = Dest + Src
`subq` Src, Dest	Dest = Dest − Src
`imulq` Src, Dest	Dest = Dest ∗ Src

`movq $7, %rax`

- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  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 %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long (4 bytes)
`q`	quad (8 bytes)

%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

You don't need to memorize every x86 instruction, use a reference sheet like this.

To implement conditionals, programs use `set` and `jmp` instructions.

`set` instructions read condition codes and set a single byte in the destination

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

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

Program Counter (PC)

%rpi	0x00f8

Condition Codes

ZF	0
SF	0
OF	0
CF	0

16 “General purpose” Registers

%rsp	0xff80
%rsi	y
%rdi	x
%rax	Return value
...	

To implement conditionals, programs use `set` and `jmp` instructions.

`set` instructions
codes and set a
destination

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

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

Zero rest of %rax

purpose"
ters

f80

urn value

a move + zero extension:

movzbl (and others)

`movzbl %al, %eax`

0x00000000	0x000000	%al

Zapped to all 0's

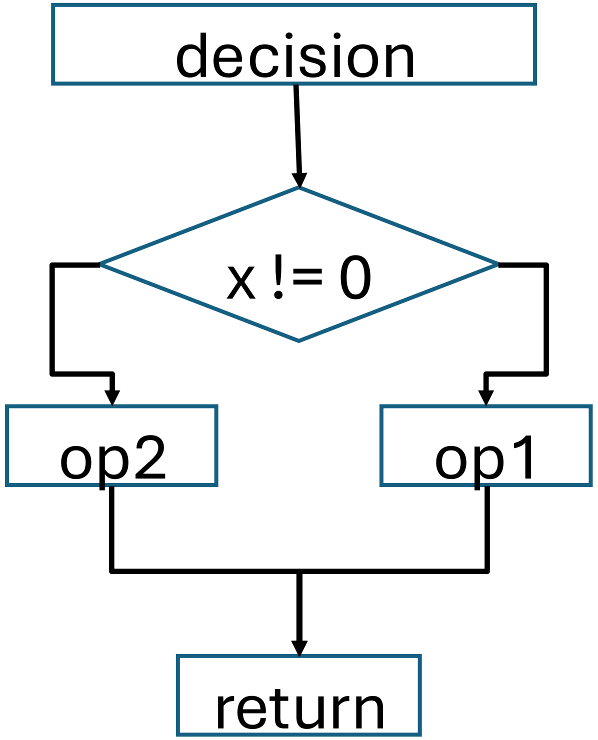
Today: How does x86-64 implement C structures that change control flow?

- Condition codes
- **Conditional branches**
- Loops
- Switch statements (we won’t have time for this)

Programs often need to change control flow based on conditionals.

```c
extern void op1(void);
extern void op2(void);

void decision(int x) {
    if (x) {
        op1();
    } else {
        op2();
    }
}
```



Control flow in x86 is all done with "goto code"

```c
extern void op1(void);
extern void op2(void);

void decision(int x) {
    if (x) {
        op1();
    } else {
        op2();
    }
}
```

```
decision:
        subq    $8, %rsp
        testl   %edi, %edi
        je      .L2
        call    op1
        jmp     .L1
.L2:
        call    op2
.L1:
        addq    $8, %rsp
        ret
```

Useful to be able to know translation of code to goto style.

```
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 FROG;
      result = x-y;
      goto Done;
FROG:
     result = y-x;
Done:
     return result;
}
```

Jumps are implemented by updating the pointer to the next instruction (%rip)

```
1   long absdiff_j
2     (long x, long y)
3   {
4       long result;
5       int ntest = x <= y;
6       if (ntest) goto Else;
7       result = x-y;
8       goto Done;
9   Else:
10       result = y-x;
11  Done:
12       return result;
13  }
```

```
1  0x112d <+4>:     cmp    %rsi,%rdi
2  0x1130 <+7>:     jle    0x1139 <absdiff+16>
3  0x1132 <+9>:     mov    %rdi,%rax
4  0x1135 <+12>:    sub    %rsi,%rax
5  0x1138 <+15>:    ret
6  0x1139 <+16>:    mov    %rsi,%rax
7  0x113c <+19>:    sub    %rdi,%rax
8  0x113f <+22>:    ret
```

Before executing line 1:

%rdi	x
%rsi	y
%rax	result
%rip	**0x112d**

Jumps are implemented by updating the pointer to the next instruction (`%rip`)

```c
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;
}
```

```
1  0x112d <+4>:    cmp    %rsi,%rdi
2  0x1130 <+7>:    jle    0x1139 <absdiff+16>
3  0x1132 <+9>:    mov    %rdi,%rax
4  0x1135 <+12>:   sub    %rsi,%rax
5  0x1138 <+15>:   ret
6  0x1139 <+16>:   mov    %rsi,%rax
7  0x113c <+19>:   sub    %rdi,%rax
8  0x113f <+22>:   ret
```

After executing line 1:

`%rdi`	`x`
`%rsi`	`y`
`%rax`	`result`
`%rip`	`0x1130`

Jumps are implemented by updating the pointer to the next instruction (`%rip`)

```c
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;
}
```

```
1  0x112d <+4>:     cmp    %rsi,%rdi
2  0x1130 <+7>:     jle    0x1139 <absdiff+16>
3  0x1132 <+9>:     mov    %rdi,%rax
4  0x1135 <+12>:    sub    %rsi,%rax
5  0x1138 <+15>:    ret
6  0x1139 <+16>:    mov    %rsi,%rax
7  0x113c <+19>:    sub    %rdi,%rax
8  0x113f <+22>:    ret
```

After executing line 2:

%rdi	x
%rsi	y
%rax	result
%rip	0x1139

General Conditional Expression Translation (Using Branches)

C Code

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

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

Goto Version

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

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

Conditional Move Instructions

- 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?

- 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;
  eval = Else_Expr;
  nt = !Test;
  if (nt) result = eval;
  return result;
```

Alternative to conditional branching with conditional move

```c
1  long absdiff
2    (long x, long y)
3  {
4      long result;
5      if (x > y)
6          result = x-y;
7      else
8          result = y-x;
9      return result;
10 }
```

```
1 absdiff:
2     movq    %rdi, %rax   # x
3     subq    %rsi, %rax   # result = x-y
4     movq    %rsi, %rdx
5     subq    %rdi, %rdx   # eval = y-x
6     cmpq    %rsi, %rdi   # x:y
7     cmovle  %rdx, %rax   # if <=, result = eval
8     ret
```

Before executing line 2:

%rdi	x
%rsi	y
%rdx	
%rax	

Alternative to conditional branching with conditional move

```c
1  long absdiff
2    (long x, long y)
3  {
4      long result;
5      if (x > y)
6          result = x-y;
7      else
8          result = y-x;
9      return result;
10 }
```

```
1  absdiff:
2     movq    %rdi, %rax   # x
3     subq    %rsi, %rax   # result = x-y
4     movq    %rsi, %rdx
5     subq    %rdi, %rdx   # eval = y-x
6     cmpq    %rsi, %rdi   # x:y
7     cmovle  %rdx, %rax   # if <=, result = eval
8     ret
```

After executing line 2:

%rdi	x
%rsi	y
%rdx	
%rax	x

Alternative to conditional branching with conditional move

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

```
1 absdiff:
2    movq     %rdi, %rax   # x
3    subq     %rsi, %rax   # result = x-y
4    movq     %rsi, %rdx
5    subq     %rdi, %rdx   # eval = y-x
6    cmpq     %rsi, %rdi   # x:y
7    cmovle   %rdx, %rax   # if <=, result = eval
8    ret
```

After executing line 3:

%rdi	x
%rsi	y
%rdx	
%rax	x - y

Alternative to conditional branching with conditional move

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

```
1 absdiff:
2    movq    %rdi, %rax   # x
3    subq    %rsi, %rax   # result = x-y
4    movq    %rsi, %rdx
5    subq    %rdi, %rdx   # eval = y-x
6    cmpq    %rsi, %rdi   # x:y
7    cmovle  %rdx, %rax   # if <=, result = eval
8    ret
```

After executing line 4:

%rdi	x
%rsi	y
%rdx	y
%rax	x-y

Alternative to conditional branching with conditional move

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

```
1 absdiff:
2    movq     %rdi, %rax   # x
3    subq     %rsi, %rax   # result = x-y
4    movq     %rsi, %rdx
5    subq     %rdi, %rdx   # eval = y-x
6    cmpq     %rsi, %rdi   # x:y
7    cmovle   %rdx, %rax   # if <=, result = eval
8    ret
```

After executing line 5:

%rdi	x
%rsi	y
%rdx	y-x
%rax	x-y

Alternative to conditional branching with conditional move

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

```
1 absdiff:
2     movq    %rdi, %rax   # x
3     subq    %rsi, %rax   # result = x-y
4     movq    %rsi, %rdx
5     subq    %rdi, %rdx   # eval = y-x
6     cmpq    %rsi, %rdi   # x:y
7     cmovle  %rdx, %rax   # if <=, result = eval
8     ret
```

After executing line 6 and 7:

%rdi	x
%rsi	y
%rdx	y-x
%rax	result

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

Bad Performance

Risky Computations

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

Both values get computed

May have undesirable effects

Unsafe

Computations with side effects

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

Both values get computed

Must be side-effect free

Illegal

Today: How does x86-64 implement C structures that change control flow?

- Condition codes
- Conditional branches
- **Loops**
- Switch statements (we won't have time for this)

do { … body … } while (condition)

while (condition) { … body … }

for (init; condition; update) { … body … }

Generic do … while goto conversion

C Code

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

Goto Version

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

- **Body:**

```
{
   Statement1;
   Statement2;
     …
   Statementn;
}
```

“Do-While” Loop example: Count number of 1s in argument x

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

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

“Do-While” Loop Compilation

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

```

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

Loop iteration #1:

Before executing line 3:

%rdi	x
%rax	0
%rdx	

“Do-While” Loop Compilation

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

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

Loop iteration #1:

After executing line 3:

%rdi	x
%rax	0
%rdx	x

“Do-While” Loop Compilation

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

```

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

Loop iteration #1:

After executing line 4:

`%rdi`	`x`
`%rax`	0
`%rdx`	`x & 0x1`

“Do-While” Loop Compilation

```
1   long pcount_goto
2     (unsigned long x) {
3     long result = 0;
4   loop:
5     result += x & 0x1;
6     x >>= 1;
7     if(x) goto loop;
8     return result;
9   }
10
```

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

Loop iteration #1:

After executing line 5:

%rdi	x
%rax	x & 0x1
%rdx	x & 0x1

“Do-While” Loop Compilation

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

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

Loop iteration #1:

After executing line 6:

%rdi	x >> 1
%rax	x & 0x1
%rdx	x & 0x1

“Do-While” Loop Compilation

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

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

Loop iteration #2:
After executing line 6 & 7:
(goto .L2)

%rdi	x >> 1
%rax	x & 0x1
%rdx	x & 0x1

“Do-While” Loop Compilation

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

```

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

Loop iteration #2:

After executing line 3:

%rdi	x >> 1
%rax	x & 0x1
%rdx	x >> 1

%rdi stores loop control variable x

%rax stores return value result

“jump-to-middle” while (test) loop implementation

- **Used with –Og**

While version

```
while (Test)
  Body
```

Goto Version

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

While Loop Example #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;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

“guarded-do” do-while loop implementation

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 #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;
}
```

- **Compare to do-while version of function**
- **Initial conditional guards entrance to loop**

"For" Loop → Do-While Loop

For version

```
for (Init; Test; Update )

    Body
```

Do-While Version

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

Goto Version

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

- **Initial test can often be optimized away – why?**

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

Reverse engineering loops is challenging!

- Compiler may use variables in assembly code that have no C equivalent and vice-versa
- Compiler may “optimize” away conditional checks
- Compiler may reuse registers

If you remember nothing else from this lecture…

There are three ways to set condition codes:
- Arithmetic and logical operations (not lea)
- Test
- Cmp

There are many ways to do things different depending on condition condes:
- Set bytes
- Jumps
- Conditional moves

You can mix and match these combinations. You'll understand the details as you do the labs, attend recitation and lecture in the next few weeks.

x86-64 code reading tips..

- Use an x86-64 reference while reading code (you don't need to memorize everything!)
- You can use gdb hex to decimal conversions!

```
(gdb) print /x 0x8 + 0x8
0x10
```

- Put a breakpoint before the function that that you want to inspect

```
(gdb) break phase_1
```

- Code trace with simulated inputs like what happens if x is in `%rsi` and y is `%rdi`, etc. Write things down and draw things like register state after each instruction.

Today: How does x86-64 implement C structures that change control flow?

- Condition codes
- Conditional branches
- Loops
- **Switch statements (we won’t have time for this)**

Switch Statement Example

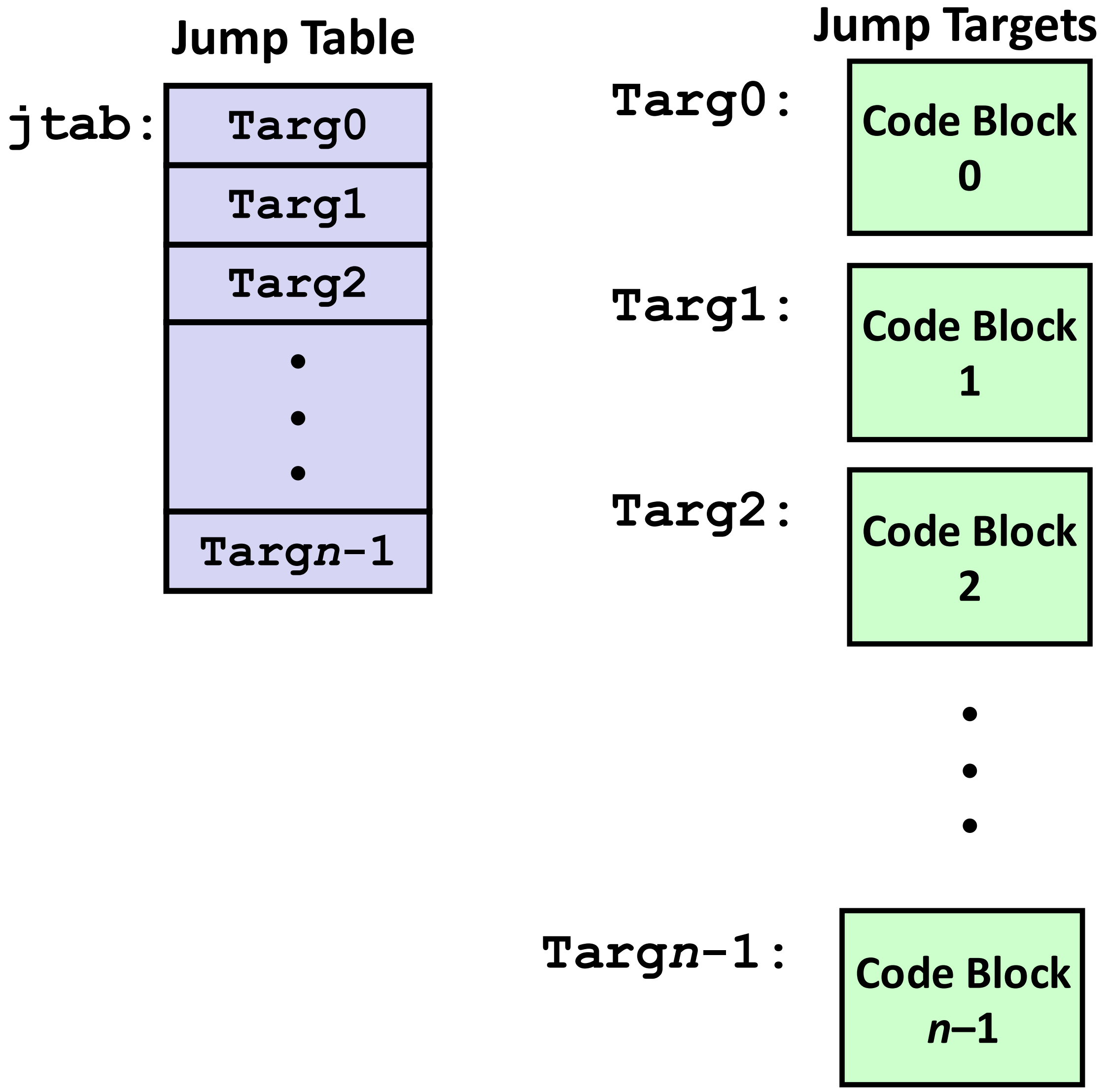
```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
       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

- **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
}
```



Translation (Extended C)

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

Switch Statement Example

```
long my_switch
   (long x, long y, long z)
{
    long w = 1;
    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;
    }
    return w;
}
```

```
my_switch:
   cmpq    $6, %rdi    # x:6
   ja      .L8    # if x > 6 jump
                  # to default
   jmp     *.L4(,%rdi,8)
```

```
.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
```

Assembly Setup Explanation

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

- **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 $0 \le \mathbf{x} \le 6$

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
```

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

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

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

```c
case 3:
        w = 1;

merge:
        w += z;
```

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

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

```
.L5:                        # Case 2
    movq    %rsi, %rax
    cqto
    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

Finding Jump Table in Binary

```
 00000000004005e0 <switch_eg>:
4005e0:       48 89 d1                mov    %rdx,%rcx
4005e3:       48 83 ff 06             cmp    $0x6,%rdi
4005e7:       77 2b                   ja     400614 <switch_eg+0x34>
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
4005f0:       48 89 f0                mov    %rsi,%rax
4005f3:       48 0f af c2             imul   %rdx,%rax
4005f7:       c3                      retq
4005f8:       48 89 f0                mov    %rsi,%rax
4005fb:       48 99                   cqto
4005fd:       48 f7 f9                idiv   %rcx
400600:       eb 05                   jmp    400607 <switch_eg+0x27>
400602:       b8 01 00 00 00          mov    $0x1,%eax
400607:       48 01 c8                add    %rcx,%rax
40060a:       c3                      retq
40060b:       b8 01 00 00 00          mov    $0x1,%eax
400610:       48 29 d0                sub    %rdx,%rax
400613:       c3                      retq
400614:       b8 02 00 00 00          mov    $0x2,%eax
400619:       c3                      retq
```

Finding Jump Table in Binary

```
 00000000004005e0 <switch_eg>:
. . .
4005e9:       ff 24 fd f0 07 40 00     jmpq   *0x4007f0(,%rdi,8)
. . .
```

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
(gdb)
```

Which numbers are pointers?

- They aren't labeled
- You have to figure it out from context

```
(gdb) info registers
rax     0x40057d         4195709
rbx     0x0              0
rcx     0x4005e0         4195808
rdx     0x7fffffffdc28   140737488346152
rsi     0x7fffffffdc18   140737488346136
rdi     0x1              1
rbp     0x0              0x0
rsp     0x7fffffffdb38   0x7fffffffdb38
r8      0x7ffff7dd5e80   140737351868032
r9      0x0              0
r10     0x7fffffffd7c0   140737488345024
r11     0x7ffff7a2f460   140737348039776
r12     0x400490         4195472
r13     0x7fffffffdc10   140737488346128
r14     0x0              0
r15     0x0              0
rip     0x40057d         0x40057d
```

Which numbers are pointers?

- They aren't labeled
- You have to figure it out from context

- %rsp and %rip always hold pointers

```
(gdb) info registers
rax     0x40057d         4195709
rbx     0x0              0
rcx     0x4005e0         4195808
rdx     0x7fffffffdc28   140737488346152
rsi     0x7fffffffdc18   140737488346136
rdi     0x1              1
rbp     0x0              0x0
rsp     0x7fffffffdb38   0x7fffffffdb38
r8      0x7ffff7dd5e80   140737351868032
r9      0x0              0
r10     0x7fffffffd7c0   140737488345024
r11     0x7ffff7a2f460   140737348039776
r12     0x400490         4195472
r13     0x7fffffffdc10   140737488346128
r14     0x0              0
r15     0x0              0
rip     0x40057d         0x40057d
```

Which numbers are pointers?

- They aren't labeled
- You have to figure it out from context

- %rsp and %rip always hold pointers
    - Register values that are "close" to %rsp or %rip are *probably* also pointers

```
(gdb) info registers
rax      0x40057d           4195709
rbx      0x0                0
rcx      0x4005e0           4195808
rdx      0x7fffffffdc28     140737488346152
rsi      0x7fffffffdc18     140737488346136
rdi      0x1                1
rbp      0x0                0x0
rsp      0x7fffffffdb38     0x7fffffffdb38
r8       0x7ffff7dd5e80     140737351868032
r9       0x0                0
r10      0x7fffffffd7c0     140737488345024
r11      0x7ffff7a2f460     140737348039776
r12      0x400490           4195472
r13      0x7fffffffdc10     140737488346128
r14      0x0                0
r15      0x0                0
rip      0x40057d           0x40057d
```

Which numbers are pointers?

- If a register is being *used* as a pointer…

```
Dump of assembler code for function main:
=> 0x40057d <+0>:  sub   $0x8,%rsp
   0x400581 <+4>:  mov   (%rsi),%rsi
   0x400584 <+7>:  mov   $0x400670,%edi
   0x400589 <+12>: mov   $0x0,%eax
   0x40058e <+17>: call  0x400460
```

Which numbers are pointers?

- If a register is being *used* as a pointer…
  - mov (%rsi), %rsi
  - …Then its value is *expected* to be a pointer.
    - There might be a bug that makes its value incorrect.

```
Dump of assembler code for function main:
=> 0x40057d <+0>:   sub    $0x8,%rsp
   0x400581 <+4>:   mov    (%rsi),%rsi
   0x400584 <+7>:   mov    $0x400670,%edi
   0x400589 <+12>:  mov    $0x0,%eax
   0x40058e <+17>:  call   0x400460
```

Which numbers are pointers?

- If a register is being *used* as a pointer...
  - mov (%rsi), %rsi
  - ...Then its value is *expected* to be a pointer.
    - There might be a bug that makes its value incorrect.

```
Dump of assembler code for function main:
=> 0x40057d <+0>:   sub    $0x8,%rsp
   0x400581 <+4>:   mov    (%rsi),%rsi
   0x400584 <+7>:   mov    $0x400670,%edi
   0x400589 <+12>:  mov    $0x0,%eax
   0x40058e <+17>:  call   0x400460
```

- Not as obvious with complicated address “modes”
  - (%rsi, %rbx) – *One* of these is a pointer, we don’t know which.
  - (%rsi, %rbx, 2) – %rsi is a pointer, %rbx isn’t (why?)
  - 0x400570(, %rbx, 2) – 0x400570 is a pointer, %rbx isn’t (why?)
  - lea (anything), %rax – (anything) *may or may not* be a pointer

Assembly Syntax

- Intel versus AT&T

    In this class we will be using the AT&T syntax

Feature	AT&T Syntax	Intel Syntax
Operand Order	source, destination	destination, source
Register Prefix	% (e.g., %eax)	None (e.g., eax)
Immediate Value Prefix	$ (e.g., $10)	None (e.g., 10)
Memory Addressing	displacement(base, index, scale)	[base + index*scale + displacement]
Operand Size Suffix	b, w, l, q (e.g., movl)	Inferred or ptr prefixes (e.g., dword ptr)