

Lecture 07

Control Instructions

CS213 – Intro to Computer Systems
Branden Ghen a – Winter 2022

Slides adapted from:

St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Administrivia

- Exam details will be posted later today
 - Class time next week Tuesday (Feb 1st)
 - Content is through **today's material**
- Exam will happen in class during lecture time
 - I have already emailed you if you are at an alternate time
- I will release a practice exam later today
 - Expect different questions but a similar *style*
 - Also good practice: Homework 2 (due Thursday), phases 1-3 of Bomb Lab

Today's Goals

- Understand converting C control flow statements to assembly
 - If, If-else, While, For, etc.
- Discuss multiple ways to represent code
 - Often an efficiency tradeoff

Outline

- **Viewing Assembly Code**
- Branching (If/Else)
- Loops (Do While, While, For)
- Conditional Move

How to Get Your Hands on Assembly

- From C source code, using a compiler
 - `gcc -O1 -S sum.c`
 - Produces file `sum.s`
 - Online compiler, shows asm output: <https://godbolt.org>
 - **Warning:** May get very different results on different machines due to different versions of gcc and different compiler settings

C Code: sum.c

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

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

Generated x86-64 assembly: sum.s

```
sum:
    pushq    %rbx
    movq     %rdx, %rbx
    call     plus
    movq     %rax, (%rbx)
    popq     %rbx
    ret
```

How to Get Your Hands on Assembly

- From machine code, using a disassembler
 - `objdump -d sum.o`
 - Within the gdb Debugger

```
linux> gdb prog
(gdb) disassemble sum
```

 - gdb tutorial coming soon!
 - **Warning:** Disassemblers are approximate; some information is lost during translation from assembly to machine code
 - Label names are lost, what is just data (vs code) is lost, etc.
- Useful if you don't have the source

```
0000000000400595 <sum>:
400595: 53                push    %rbx
400596: 48 89 d3          mov     %rdx,%rbx
400599: e8 f2 ff ff ff    callq   400590 <plus>
40059e: 48 89 03          mov     %rax, (%rbx)
4005a1: 5b                pop     %rbx
4005a2: c3                retq
```

Godbolt

Ignore
“_dl_relocate_static_pie”

Play around with this to
try stuff on your own

<https://godbolt.org/>

The screenshot displays the Godbolt Compiler Explorer interface. The left pane shows the C source code with three functions: `square`, `add_inputs_plus_two`, and `check_greater`. The right pane shows the generated assembly for x86-64 gcc 10.2 with optimization level -O1. The assembly includes instructions for `_dl_relocate_static_pie`, `endbr64`, `retq`, `nopw`, `imul`, `mov`, `leaq`, `cmp`, `setg`, and `retq`. The assembly is color-coded to match the source code, with function names and labels highlighted in green.

```
C source #1 X
A Save/Load + Add new... Vim C
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
5
6
7 int add_inputs_plus_two(int a, int b) {
8     return a+b+2;
9 }
10
11
12 char check_greater(int a, int b) {
13     return (a > b);
14 }

x86-64 gcc 10.2 (Editor #1, Compiler #1) C X
x86-64 gcc 10.2 -O1
A Output... Filter... Libraries + Add new... Add tool...
_dl_relocate_static_pie:
f3 0f 1e fa
401050 endbr64
c3
401054 retq
66 2e 0f 1f 84 00 00 00 00 00
401055 nopw %cs:0x0(%rax,%rax,1)
90
40105f nop
square:
0f af ff
401102 imul %edi,%edi
89 f8
401105 mov %edi,%eax
c3
401107 retq
add_inputs_plus_two:
8d 44 37 02
401108 leaq 0x2(%rdi,%rsi,1),%eax
c3
40110c retq
check_greater:
39 f7
40110d cmp %esi,%edi
0f 9f c0
40110f setg %al
c3
401112 retq
```

- Godbolt example!

Outline

- Viewing Assembly Code
- **Branching (If/Else)**
- Loops (Do While, While, For)
- Conditional Move

What can instructions do?

- Move data: ✓
- Arithmetic: ✓
- **Transfer control**
 - Instead of executing next instruction, go somewhere else

```
if (x > y)
    result = x-y;
else
    result = y-x;
```

```
while (x > y)
    result = x-y;
return result;
```

- Sometimes we want to go from the red code to the green code
- But the blue code is what's next!
- Need to transfer control! Execute an instruction that is not the next one
- And ***conditionally***, too! (i.e., based on a condition)

Conditional operations in (x86-64) assembly

- First, an instruction sets condition codes
 - Implicitly: any arithmetic (not `leaq`)
 - Explicitly: `cmp`, `test`
- Second, another instruction observes condition codes
 - And does one thing or another depending on what it sees
- In the second category, we saw the `setX` instructions
 - `set{e, ne, s, ...} D` evaluates condition, writes 0 or 1 to `D`

SetX	Condition	Description
<code>sete</code>	<code>ZF</code>	Equal / Zero
<code>setne</code>	<code>~ZF</code>	Not Equal / Not Zero
<code>setg</code>	<code>~(SF^OF) & ~ZF</code>	Greater (Signed)
<code>setge</code>	<code>~(SF^OF)</code>	Greater or Equal (Signed)
<code>...</code>	<code>...</code>	<code>...</code>

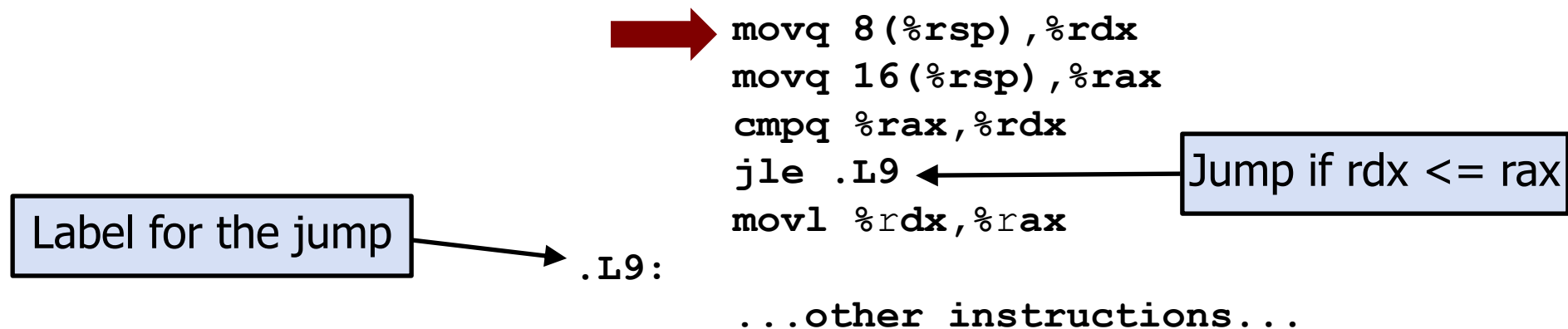
`cmp SRC, DST`
`setX`

Asks question:

Is `DST` `X` `SRC`?

Breaking with sequential execution

- “Normal” execution follows instructions in listed (sequential) order
- To move to a different location – jump
 - Jump to different part of code depending on condition codes
 - Destination of a jump – label: particular address at which we find code
 - Label addresses are determined when generating the object code



Jumping

- jX Instructions
 - Jump to different part of code depending on condition codes
 - `jmp` has two options
 - **Direct:** to a label (literal address)
 - **Indirect:** based on a register
 - Direct is the most common

jX	Condition	Description
<code>jmp</code>	1	Unconditional
<code>je</code>	ZF	Equal / Zero
<code>jne</code>	$\sim ZF$	Not Equal / Not Zero
<code>js</code>	SF	Negative
<code>jns</code>	$\sim SF$	Nonnegative
<code>jg</code>	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
<code>jge</code>	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
<code>j1</code>	$(SF \wedge OF)$	Less (Signed)
<code>jle</code>	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
<code>ja</code>	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
<code>jae</code>	$\sim CF$	Above or Equal (unsigned)
<code>jb</code>	CF	Below (unsigned)
...


Key idea: building C constructs with assembly

- Jump will let us build the flow control statements in C
 - If, While, For, Switch, etc.
- But the translation isn't always obvious
 - Might switch ordering, or negate the logical condition
 - Maintains the same result when it runs, but easier for assembly
- Steps
 1. Transform C into something simpler (closer to assembly)
 2. Transform simpler C into assembly

The “something simpler” is goto

- C allows **goto** as means of transferring control
 - Closer to machine-level programming style
 - Place labels wherever you want in code
 - Goto “jumps” to the referenced label
- Generally considered bad programming style
 - Makes it really difficult to understand what code is doing

```
int i = 0;
start:
    if (i >= 3) { goto end; }
    ++i;
    printf("Hello ");
    goto start;
end:
    printf("World!\n");
```




Prints:

```
"Hello Hello Hello World!\n"
```

Conditional Branch Example

```
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;
}
```

- Translate an if statement into a "simpler" goto statement
 - Makes the if statement closer to machine code because goto can translate to jumps

Conditional Branch Example

Goto Version

```
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;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle     .L2           # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    jmp     .L3
.L2:
    # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ret
```

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

Conditional Branch Example

Goto Version

```
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;
}
```

Asm Version

```
absdiff:
→ cmpq    %rsi, %rdi
→ jle     .L2          # x <= y
    movq   %rdi, %rax
    subq   %rsi, %rax
    jmp    .L3
.L2:      # x <= y
    movq   %rsi, %rax
    subq   %rdi, %rax
.L3:
    ret
```

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

Conditional Branch Example

Goto Version

```
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;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle     .L2           # x <= y
    → movq   %rdi, %rax
    → subq   %rsi, %rax
    jmp     .L3
.L2:
    # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ret
```

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

Conditional Branch Example

Goto Version

```
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;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle     .L2           # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    → jmp    .L3
.L2:
    # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ret
```

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

Conditional Branch Example

Goto Version

```
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;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle     .L2           # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    jmp     .L3
.L2:
    ➔ movq    %rsi, %rax
    ➔ subq    %rdi, %rax
.L3:
    ret
```

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

Conditional Branch Example

Goto Version

```
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;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle     .L2           # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    jmp     .L3
.L2:
    # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ➡ ret
```

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

General “if-then-else” translation

C Code

```
if (test-expr)  
    then-statement  
else  
    else-statement
```

Goto Version

```
ntest = !(test-expr) ;  
if (ntest) {  
    goto Else;  
}  
then-statement;  
goto done;  
Else:  
    else-statement;  
done:
```

- *test-expr* is an expression returning integer
 - = 0 interpreted as false, ≠0 interpreted as true
- Only one of the two statements is executed
 - i.e. only one of the two *branches* of code
- That's one translation; there are others
 - E.g., flipping the order of the blocks instead of flipping the test
- Conditional expressions ($x \ ? \ y \ : \ z$) can use the same translation

If statement - bigger example

```
long test(long a, long b) {  
    long c;  
    if (a > b) {  
        c = 1;  
    } else if (a < b) {  
        c = -1;  
    } else {  
        c = 0;  
    }  
    return c;  
}
```


If statement - bigger example

```
long test(long a, long b) {  
    long c;  
    if (a > b) {  
        c = 1;  
    } else if (a < b) {  
        c = -1;  
    } else {  
        c = 0;  
    }  
    return c;  
}
```

a→%rdi, b→%rsi, c→%rax

```
    cmp %rsi, %rdi  
    jle elif          # !(a > b)  
    movq $1, %rax  
    jmp end  
elif:  
    cmp %rsi, %rdi  
    jge else          # !(a < b)  
    movq $-1, %rax  
    jmp end  
else:  
    movq $0, %rax  
end:  
    ret                # returns %rax
```

If statement - bigger example

```
long test(long a, long b) {  
    long c;  
    if (a > b) {  
        c = 1;  
    } else if (a < b) {  
        c = -1;  
    } else {  
        c = 0;  
    }  
    return c;  
}
```

a→%rdi, b→%rsi, c→%rax

```
cmp %rsi, %rdi  
jle elif          # !(a > b)  
movq $1, %rax  
jmp end
```

elif:

```
cmp %rsi, %rdi  
jge else          # !(a < b)  
movq $-1, %rax  
jmp end
```

else:

```
movq $0, %rax
```

end:

```
ret                # returns %rax
```

If statement - bigger example

```
long test(long a, long b) {  
    long c;  
    if (a > b) {  
        c = 1;  
    } else if (a < b) {  
        c = -1;  
    } else {  
        c = 0;  
    }  
    return c;  
}
```

a→%rdi, b→%rsi, c→%rax

```
    cmp %rsi, %rdi  
    jle elif          # !(a > b)  
    movq $1, %rax  
    jmp end  
elif:  
    cmp %rsi, %rdi  
    jge else          # !(a < b)  
    movq $-1, %rax  
    jmp end  
else:  
    movq $0, %rax  
end:  
    ret                # returns %rax
```

If statement - bigger example

```
long test(long a, long b) {  
    long c;  
    if (a > b) {  
        c = 1;  
    } else if (a < b) {  
        c = -1;  
    } else {  
        c = 0;  
    }  
    return c;  
}
```

```
a→%rdi, b→%rsi, c→%rax  
  
    cmp %rsi, %rdi  
    jle elif          # !(a > b)  
    movq $1, %rax  
    jmp end  
elif:  
    cmp %rsi, %rdi  
    jge else          # !(a < b)  
    movq $-1, %rax  
    jmp end  
else:  
    movq $0, %rax  
end:  
    ret              # returns %rax
```

If statement - bigger example

```
long test(long a, long b) {  
    long c;  
    if (a > b) {  
        c = 1;  
    } else if (a < b) {  
        c = -1;  
    } else {  
        c = 0;  
    }  
    return c;  
}
```

```
a→%rdi, b→%rsi, c→%rax  
  
    cmp %rsi, %rdi  
    jle elif          # !(a > b)  
    movq $1, %rax  
    jmp end  
elif:  
    cmp %rsi, %rdi # unnecessary  
    jge else          # !(a < b)  
    movq $-1, %rax  
    jmp end  
else:  
    movq $0, %rax  
end:  
    ret                # returns %rax
```

Break + Optimization (O1)

```
long test(long a, long b) {  
    long c;  
    if (a > b) {  
        c = 1;  
    } else if (a < b) {  
        c = -1;  
    } else {  
        c = 0;  
    }  
    return c;  
}
```

a→%rdi, b→%rsi, c→%rax

```
movq $1, %rax  
cmp %rsi, %rdi  
jg end  
{  
    setl %al  
    movzbq %al, %rax  
    neg %rax  
}  
end:  
ret # returns %rax
```

What is the yellow code block doing above?

Break + Optimization (O1)

```
long test(long a, long b) {  
    long c;  
    if (a > b) {  
        c = 1;  
    } else if (a < b) {  
        c = -1;  
    } else {  
        c = 0;  
    }  
    return c;  
}
```

a→%rdi, b→%rsi, c→%rax

```
movq $1, %rax  
cmp %rsi, %rdi  
jg end  
[  
    setl %al  
    movzbq %al, %rax  
    neg %rax  
]  
end:  
ret # returns %rax
```

else if and else
together

What is the yellow code block doing above?
Generates 0 (not less) or -1 (less)

Indirect jump

- `jmp *0x40000(%rdi, %rdx, 8)`
 - Calculate memory address: $0x40000 + \%rdi + 8 * \%rdx$
 - Load value from memory address
 - Jump to *that* value
- Indirect jumps jump to the address loaded from memory
 - Essentially a function pointer
 - Or used for a Jump Table: efficient switch statements (see bonus slides)
- The `*` lets you know that something tricky is going on
 - Displacement could be a label rather than a value

Outline

- Viewing Assembly Code
- Branching (If/Else)
- **Loops (Do While, While, For)**
- Conditional Move

Loops

- C provides different looping constructs
 - `while`, `do ... while`, `for`
- No corresponding instruction in machine code
- Most compilers
 - Transform general loops into `do ... while`

```
do  
    body-statement  
while (test-expr);
```

Do-while:

Same idea as a while loop, but the body always runs at least once

- Rewrite that with `goto`
- Then compile them into machine code

“Do-While” Loop Compilation

- Running example: count number of 1s in x (“popcount”)
 - We’ll write it with different kinds of loops
 - What the body of the loop does is not our focus; we’ll just ignore it
- Use conditional branch to either continue looping or to exit loop

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;
}
```

“Do-While” assembly translation

Goto Version

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

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

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

“Do-While” assembly translation

Goto Version

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

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

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

Which instruction sets the condition codes for `jne`?

Logical shift right (`shrq`)

“Do-While” assembly translation

Goto Version

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

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

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

- `rep` instruction repeats string operations following it **What?!!**

“Do-While” assembly translation

Goto Version

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

```

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

- `rep` instruction repeats string operations following it **What?!!**
- `rep; ret` uses `rep` as a no-op (a.k.a nop, an operation that does nothing)
 - Example of a compiler optimization that you might run into in real assembly code
 - AMD recommends this to speed up execution when there is a jump before a return
 - See CE361 and CE452 for more details (Computer Architecture courses)

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

General “Do-While” Translation

- Body: {

Statement₁;

Statement₂;

...

Statement_n;

}

C Code

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

Goto Version

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

- Test returns integer

- = 0 interpreted as false
- ≠ 0 interpreted as true

General “While” Translation #1

- “Jump-to-middle” translation
- Most straightforward match to how “while” works

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;
}
```

- Initial goto starts loop at test

Comparing while to do-while

While with goto (jump to middle)

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

Do While with goto

```
long pcount_dowhile_goto
(unsigned long x)
{
    long result = 0;

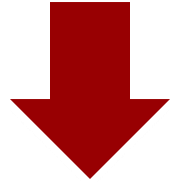
loop:
    result += x & 0x1;
    x >>= 1;

    if (x) {goto loop;}
    return result;
}
```

General “While” Translation #2

While version

```
while ( Test )  
    Body
```



Do-While Version

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

- “Do-while” conversion
- More optimized compiler translation



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;
}
```

Goto 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;
}
```

- Initial conditional guards entrance to loop

Comparing jump-to-middle and guarded-do-while

While with goto (jump to middle)

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

    return result;
}
```

While with goto (guarded do-while)

```
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;
}
```

"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" → "While" → "Do-While" → "Goto"

For Version

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



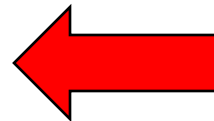
While Version

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



Do-While Version

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



Goto Version

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


"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
long pcount_for(unsigned 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!
(0 always < WSIZE)

Goto Version

```
#define WSIZE 8*sizeof(int)
long pcount_for_gt(unsigned x)
{
    size_t i;
    long result = 0;
    i = 0; Init
    if (!(i < WSIZE)) ! Test
    goto done;
loop:
    { Body
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    i++; Update
    if (i < WSIZE) Test
        goto loop;
done:
    return result;
}
```

Break + Assembly to loop

What does this
function do?

```
my_function:           # %rdi is argument1
    mov $0, %rax
    mov $0, %rbx
    test %rdi, %rdi
    je end
loop:
    add %rbx, %rax
    add $1, %rbx
    cmp %rdi, %rbx
    jne loop
end:
    ret                # returns %rax
```

Assembly to loop

```
my_function:          # %rdi is argument1
    mov $0, %rax      # clear variables
    mov $0, %rbx
    test %rdi, %rdi
    je end            # skip loop
loop:
    add %rbx, %rax
    add $1, %rbx
    cmp %rdi, %rbx
    jne loop
end:
    ret               # returns %rax
```

Assembly to loop

```
my_function:      # %rdi is argument1
    mov $0, %rax  # clear variables
    mov $0, %rbx
    test %rdi, %rdi
    je end        # skip loop
loop:
    add %rbx, %rax
    add $1, %rbx
    cmp %rdi, %rbx
    jne loop
end:
    ret           # returns %rax
```

Assembly to loop

```
my_function:      # %rdi is argument1
    mov $0, %rax  # clear variables
    mov $0, %rbx
    test %rdi, %rdi
    je end        # skip loop
loop:
    add %rbx, %rax # %rax += %rbx
    add $1, %rbx   # %rbx += 1
    cmp %rdi, %rbx
    jne loop
end:
    ret           # returns %rax
```

Assembly to loop

```
my_function:      # %rdi is argument1
    mov $0, %rax   # clear variables
    mov $0, %rbx
    test %rdi, %rdi
    je end         # skip loop
loop:
    add %rbx, %rax  # %rax += %rbx
    add $1, %rbx    # %rbx += 1
    cmp %rdi, %rbx
    jne loop        # while %rbx != %rdi
end:
    ret            # returns %rax
```

Assembly to loop

```
long my_function(long rdi) {  
    long rax = 0;  
    long rbx = 0;  
  
    while (rbx != rdi) {  
        rax += rbx;  
        rbx += 1;  
    }  
  
    return rax;  
}
```

```
my_function:        # %rdi is argument1  
    mov $0, %rax    # clear variables  
    mov $0, %rbx  
  
    test %rdi, %rdi  
    je end          # skip loop  
  
loop:  
    add %rbx, %rax  # %rax += %rbx  
    add $1, %rbx    # %rbx += 1  
  
    cmp %rdi, %rbx  
    jne loop        # while %rbx != %rdi  
  
end:  
  
    ret            # returns %rax
```

Assembly to loop

```
long my_function(long rdi) {  
    long rax = 0;  
    long rbx = 0;  
  
    while (rbx != rdi) {  
        rax += rbx;  
        rbx += 1;  
    }  
  
    return rax;  
}
```

```
long my_function(long max) {  
    long result = 0;  
    for (int i=0; i<max; i++){  
        result += i;  
    }  
  
    return result;  
}
```

```
my_function:        # %rdi is argument1  
    mov $0, %rax    # clear variables  
    mov $0, %rbx  
  
    test %rdi, %rdi  
    je end          # skip loop  
  
loop:  
    add %rbx, %rax  # %rax += %rbx  
    add $1, %rbx    # %rbx += 1  
  
    cmp %rdi, %rbx  
    jne loop        # while %rbx != %rdi  
  
end:  
  
    ret            # returns %rax
```


Outline

- Viewing Assembly Code
- Branching (If/Else)
- Loops (Do While, While, For)
- **Conditional Move**

The Problem with Conditional Jumps

- Conditional jumps = conditional *transfer of control*
 - i.e., forget what you thought you were going to do, do this other thing instead
- Modern processors like to do work “ahead of time”
 - Keywords: ***pipelining, branch prediction, speculative execution***
 - Transfer of control may mean throwing that work away
 - That’s inefficient
- Solution: conditional *moves*
 - We still get to do something conditionally
 - But no transfer of control necessary
 - “Ahead of time” work can always be kept

Conditional Moves

cmovX	Description
cmove S, D	equal / Zero
cmovne S, D	not equal / Not zero
comvs S, D	negative
cmovns S, D	nonnegative
comvg S, D	greater (Signed)
cmovge S, D	greater or equal (Signed)
cmovl S, D	less (Signed)
cmovle S, D	less or equal (Signed)
cmova S, D	above (Unsigned)
cmovae S, D	above or equal (Unsigned)
cmovb S, D	below (Unsigned)
cmovbe S, D	below or equal (Unsigned)

*$D \leftarrow S$ only if
test condition
is true*

Conditional Move Example

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

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

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

Look Ma, no branching!

Must compute both results, though, which is not always possible or desirable...

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;
```

- A `cmov` requires that both values get computed
- Could trigger a fault (compiler must use jumps instead)

Computations with side effects

```
val = x > 0 ? x++ : x--;
```

- Both values get computed
- Needs use extra temporary registers to hold intermediate results

If, else if, else – optimized (O3)

```
long test(long a, long b) {  
    long c;  
    if (a > b) {  
        c = 1;  
    } else if (a < b) {  
        c = -1;  
    } else {  
        c = 0;  
    }  
    return c;  
}
```

a→%rdi, b→%rsi, c→%rax

```
movq $0, %rax    # clear reg  
cmp %rsi, %rdi  
movq $1, %rdx  
setl %al  
neg %rax          # else if and else  
                  # together  
                  # (%al is %rax)  
cmp %rsi, %rdi  
cmovg %rdx, %rax  # select output  
ret              # returns %rax
```

Outline

- Viewing Assembly Code
- Branching (If/Else)
- Loops (Do While, While, For)
- Conditional Move

- Bonus Slides
 - Switch Statements and Jump Tables

Switch statements

- A multi-way branching capability based on the value of an integer
- Useful when many possible outcomes
- Switch cases
 - Fall through cases:
 - Here 1
 - Missing cases:
 - Here 3, 4, 5, 6
 - Multiple case labels:
 - Here 7 & 8
- Easier to read C code and more efficient implementation with jump tables

```
long switch_fun
(long x, long y, long z, long w) {
    switch(x) {
        case 0:
            w += y;
            break;
        case 1:
            w -= y;
            /* FALL THROUGH */
        case 2:
            w += z;
            break;
        /* MISSING CASES */
        case 7:
        case 8: /* MULTIPLE CASES */
            w -= z;
            break;
        default:
            w = 2;
            break;
    }
    w += 5;
    return w;
}
```

Target code blocks

```
case 0:
    w += y;
    break;
case 1:
    w -= y;
    /* FALL THROUGH */
case 2:
    w += z;
    break;
case 7:
case 8: /* MULTIPLE CASES */
    w -= z;
    break;
default:
    w = 2;
    break;
```

One code block per case!

```
.L7:                                # case 0
    addq    %rsi, %rcx
    jmp     .L2                      # break

.L6:                                # case 1
    subq    %rsi, %rcx
    # FALL THROUGH

.L5:                                # case 2
    addq    %rdx, %rcx
    jmp     .L2                      # break

.L3:                                # cases 7 and 8
    subq    %rdx, %rcx
    jmp     .L2                      # break

.L8:                                # default
    movl    $2, %ecx
    jmp     .L2                      # break

.L2:
    ...
```

%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rcx	Argument w
%rax	Return value

break becomes a jump to after the **switch (.L2)**!

Jump tables

- Definition: An array where entry i is the address of the code segment to run when the switch variable equals i

Switch statement

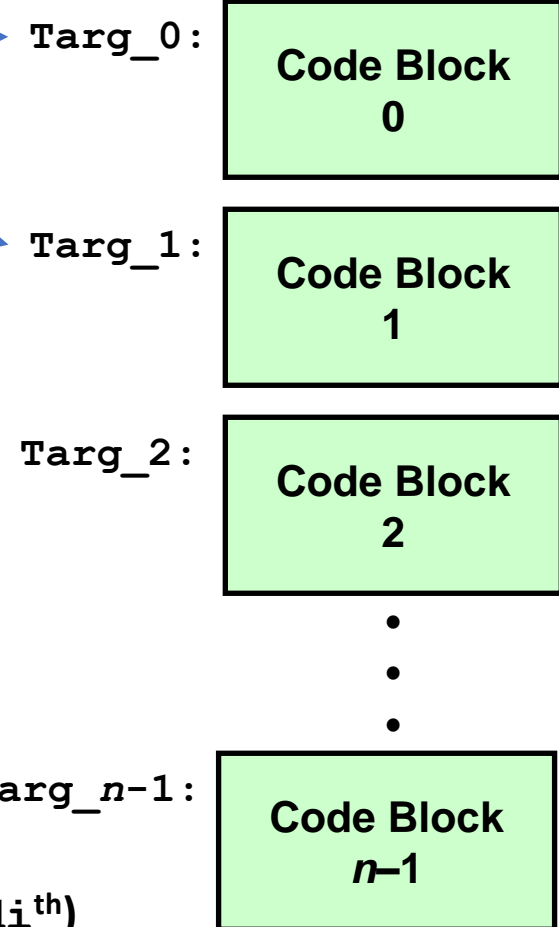
```
switch(x) {  
  case 0:  
    Block 0  
  case 1:  
    Block 1  
    . . .  
  case  $n-1$ :  
    Block  $n-1$   
}
```

Jump table (data in memory)

jtab:

Targ_0	0
Targ_1	8
Targ_2	16
.	
.	
.	
Targ_ $n-1$	$(n-1) * 8$

Jump targets (code in memory)



Approx. translation:

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

- Register `%rdi` holds the switch variable x
- `jtab` is the address of the jump table

Q1: which *table entry* holds the *address* of the next instruction? The x^{th} (or `%rdith`)

Q2: what is the memory address of *that entry*? `jtab + %rdi*8`

Q3: what is the address of the *next instruction* to execute? `M[jtab + %rdi*8]`

Jump table for our example

Jump table

(addresses of code blocks)

.L4:	
.quad	.L7
.quad	.L6
.quad	.L5
.quad	.L8
.quad	.L8
.quad	.L8
.quad	.L8
.quad	.L3
.quad	.L3

At address .L4, store a
quad word of data, then
another, then another...

These quad words are
the addresses of each
of these code blocks.

```
case 0:
    w += y;
    break;
case 1:
    w -= y;
    /* FALL THROUGH */
case 2:
    w += z;
    break;
case 7:
case 8: /* MULTIPLE CASES */
    w -= z;
    break;
default:
    w = 2;
    break;
```

Jump targets

jtab:	Targ_0	Targ_0:	Code Blk 0
	Targ_1	Targ_1:	Code Blk 1
	Targ_2		
	⋮		⋮
	⋮		⋮
	Targ_n-1	Targ_n-1:	Code Blk n-1

Putting it all Together

Jump table (addresses of code blocks)

```
long switch_fun (...)
{
    switch(x) {
        // cases 0,1,2,7,8
        // and default
    }
    w += 5;
    return w;
}
```

%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rcx	Argument w
%rax	Return value

```
.L4:
    .quad    .L7 # x=0
    .quad    .L6 # x=1
    .quad    .L5 # x=2
    .quad    .L8 # x=3
    .quad    .L8 # x=4
    .quad    .L8 # x=5
    .quad    .L8 # x=6
    .quad    .L3 # x=7
    .quad    .L3 # x=8
```

switch_fun:

```
    cmpq     $8, %rdi    # compare x to 8
    ja       .L8         # above 8 (outside table!) -> default
    jmp      *.L4(, %rdi, 8) # goto *Jtab[x], a.k.a M[.L4 + x*8]
                                # * means an indirect jump (like deference)
```



Indirect jump: look up address in memory; jump there

```

long switch_fun
(long x, long y, long z, long w) {
    switch(x) {
    case 0:
        w += y;
        break;
    case 1:
        w -= y;
        /* FALL THROUGH */
    case 2:
        w += z;
        break;
    /* MISSING CASES */
    case 7:
    case 8: /* MULTIPLE CASES */
        w -= z;
        break;
    default:
        w = 2;
        break;
    }
    w += 5;
    return w;
}

```

Full assembly code for our example

```

switch_fun:
    cmpq    $8, %rdi
    ja      .L8
    jmp     *.L4(, %rdi, 8)
.L4:
    .quad   .L7
    .quad   .L6
    .quad   .L5
    .quad   .L8
    .quad   .L8
    .quad   .L8
    .quad   .L8
    .quad   .L3
    .quad   .L3
.L7:
    addq    %rsi, %rcx
    jmp     .L2

```

```

.L6:
    subq    %rsi, %rcx
    # FALL THROUGH
.L5:
    addq    %rdx, %rcx
    jmp     .L2
.L3:
    subq    %rdx, %rcx
    jmp     .L2
.L8:
    movl    $2, %ecx
    jmp     .L2
.L2:
    leaq    5(%rcx), %rax
    ret

```

Another Jump Table Example: starting with assembly

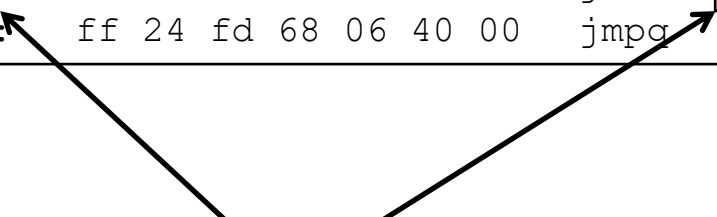
- QUIZ: find the address of the jump table and code blocks

- `linux> objdump -d prog`

- The jump table starts at address `0x400668`

- The **default** code block is at address `0x40055c`

```
0000000000400528 <switch_eg>:
400528:  48 89 d1                mov     %rdx,%rcx
40052b:  48 83 ff 06            cmp     $0x6,%rdi
40052f:  77 2b                  ja      40055c <switch_eg+0x34>
400531:  ff 24 fd 68 06 40 00    jmpq    *0x400668(,%rdi,8)
```



Note: these are hex values (memory addresses for instructions)
objdump does not put 0x in front of instruction addresses when it disassembles

How would you find the address of the other code blocks?

Object code: Jump Table

- Jump table
 - Doesn't show up in disassembled code
 - Can inspect using GDB: examine data starting at address **0x400668**

gdb prog

(gdb) x/7xg 0x400668

- Examine **7** hexadecimal format "**g**iant words" (8-bytes each)
- Use command "**help x**" to get format documentation

0x400668:

```
0x00000000000040055c
0x000000000000400538
0x000000000000400540
0x00000000000040054a
0x00000000000040055c
0x000000000000400553
0x000000000000400553
```

Jump table

jtab:

Targ 0
Targ 1
Targ 2
⋮
Targ _{n-1}

Jump targets

Targ_0: **Code Blk 0**

Targ_1: **Code Blk 1**

⋮

Targ_{n-1}: **Code Blk n-1**

How can you see the code for each one of the target code blocks?

Object code: Disassemble targets

0x40055c

0x400538

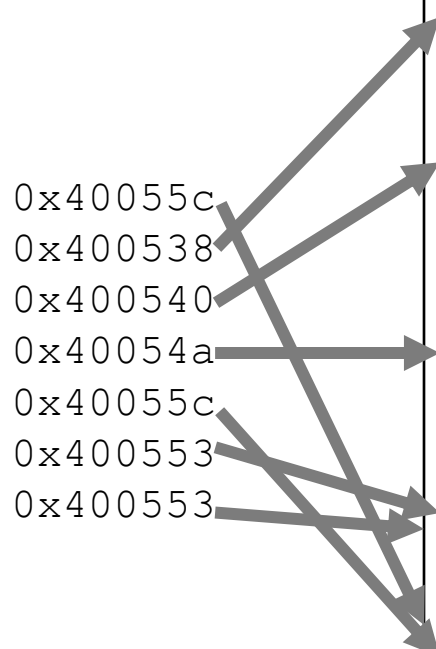
0x400540

0x40054a

0x40055c

0x400553

0x400553



```
400538: 48 89 f0      mov  %rsi,%rax
40053b: 48 0f af c2    imul %rdx,%rax
40053f: c3            retq
400540: 48 89 f0      mov  %rsi,%rax
400543: 48 99         cqto
400545: 48 f7 f9      idiv %rcx
400548: eb 05         jmp  40054f <switch_eg+0x27>
40054a: b8 01 00 00 00 mov  $0x1,%eax
40054f: 48 01 c8      add  %rcx,%rax
400552: c3            retq
400553: b8 01 00 00 00 mov  $0x1,%eax
400558: 48 29 d0      sub  %rdx,%rax
40055b: c3            retq
40055c: b8 02 00 00 00 mov  $0x2,%eax
400561: c3            retq
```

```
.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
```

linux> gdb prog

(gdb) disassemble 0x400538,0x400562

Object code: Disassemble targets

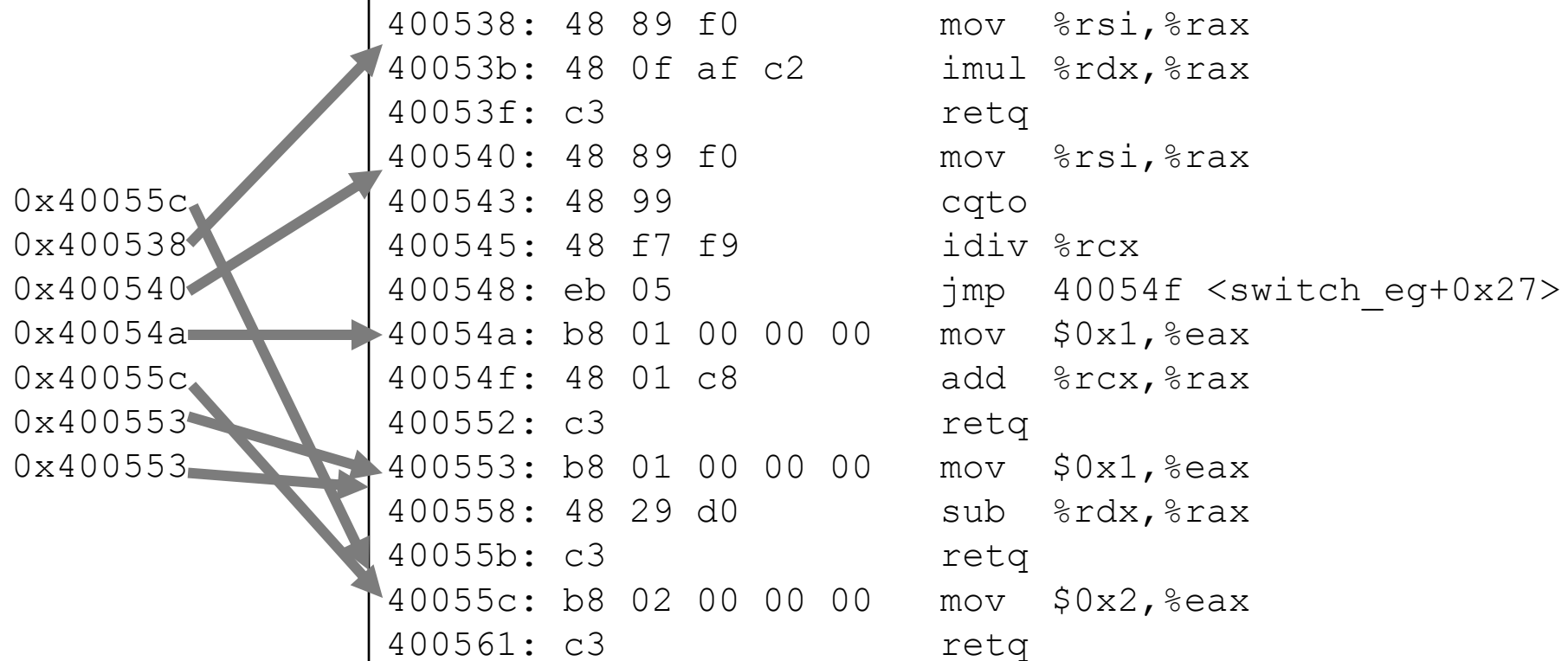


Diagram illustrating the mapping of C code labels to assembly instructions:

- 0x40055c → 400538: 48 89 f0 (mov %rsi,%rax)
- 0x400538 → 40053b: 48 0f af c2 (imul %rdx,%rax)
- 0x400540 → 400540: 48 89 f0 (mov %rsi,%rax)
- 0x40054a → 40054a: b8 01 00 00 00 (mov \$0x1,%eax)
- 0x40055c → 40054f: 48 01 c8 (add %rcx,%rax)
- 0x400553 → 400552: c3 (retq)
- 0x400553 → 400553: b8 01 00 00 00 (mov \$0x1,%eax)
- 0x400553 → 400558: 48 29 d0 (sub %rdx,%rax)
- 0x400553 → 40055b: c3 (retq)
- 0x400553 → 40055c: b8 02 00 00 00 (mov \$0x2,%eax)

Address	Hex	Assembly
400538:	48 89 f0	mov %rsi,%rax
40053b:	48 0f af c2	imul %rdx,%rax
40053f:	c3	retq
400540:	48 89 f0	mov %rsi,%rax
400543:	48 99	cqto
400545:	48 f7 f9	idiv %rcx
400548:	eb 05	jmp 40054f <switch_eg+0x27>
40054a:	b8 01 00 00 00	mov \$0x1,%eax
40054f:	48 01 c8	add %rcx,%rax
400552:	c3	retq
400553:	b8 01 00 00 00	mov \$0x1,%eax
400558:	48 29 d0	sub %rdx,%rax
40055b:	c3	retq
40055c:	b8 02 00 00 00	mov \$0x2,%eax
400561:	c3	retq

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

Object code: Memory View

