Lecture 07
Control Flow Instructions

CS213 – Intro to Computer Systems

Branden Ghena – Fall 2023

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

Northwestern

Get started on Bomb Lab right away

- Bomb lab available now
  - What should you do before the exam?
    - Phases 1-3 of Bomb Lab
    - They are good practice for the kinds of assembly problems I'll put on the exam

  - Phases 4-6 are harder and can honestly wait
    - We'll talk about stuff in lectures on "Procedures" and "Pointers, Arrays, and Structs" that will help with this part

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

- **Condition Codes**
- Branching (If/Else)
- Loops (Do While, While, For)
- Conditional Move

Condition codes

- Control is mediated via *Condition codes*
  - single-bit registers that record answers to questions about values

  - E.g., Is value x greater than value y? Are they equal? Is their sum even?
  - Let's keep "question" abstract for now. We'll see the details in a bit.

  - **Terminology**:
    - a bit is ***set*** if it is 1
    - a bit is ***cleared*** (or ***reset***) if it is 0

Conditionals at the machine level

- At machine level, conditional operations are a 2-step process:
  - Perform an operation that ***sets*** or ***clears*** condition codes (ask questions)
  - Then ***observe*** which condition codes are set, do the operation (or not)

- Can express Boolean operations, conditionals, loops, etc.
  - We will see the first today, and more control next lecture

- So now we need three things:
  1. Instructions that compare values and set condition codes
  2. Instructions that observe condition codes and do something (or not)
  3. A set of actual condition codes (what questions do we track answers to?)

Step 1: Setting condition codes

- Analogy: Asking ALL the possible questions at once
  - And recording the answers
  - We don't know yet which question is the one we care about!

- Done in one of two ways
  - **Implicitly**: all* arithmetic instructions set (and reset) condition codes in addition to producing a result
    - *except `lea`; it's not "officially" an arithmetic instruction

  - **Explicitly**: by instructions whose sole purpose is to set condition codes
    - E.g., `cmpq`
    - They don't actually produce results (in registers or memory)

  - Condition codes are left unchanged by other operations (such as `mov`)

Implicitly Setting Condition Codes

- Condition codes on x86
  - **CF** Carry Flag (for unsigned)
  - **ZF** Zero Flag
  - **PF** Parity Flag
  - **SF** Sign Flag (for signed)
  - **OF** Overflow Flag (for signed)

  - Not an arbitrary set! By combining them, can keep track of answers to many useful questions! (We'll see exactly which in a bit.)

Implicitly Setting Condition Codes

CF (Carry) **SF** (Sign) **ZF** (Zero) **OF** (Overflow) **PF** (Parity)

- Set (or reset) based on the result of arithmetic operations

  Example: `addq Src,Dest    # C-analog: t = a+b`

  - **ZF set** if `t == 0`
  - **SF set** if `t < 0` (as signed encoding)
  - **CF set** if carry out from most significant bit (unsigned overflow)

    also CF takes the value of the last bit shifted (left or right)
  - **OF set** if twos-complement (signed) overflow (pos/neg overflow)

    `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

    also, set if a 1-bit shift operation changes the sign of the result
  - **PF set** if `t` has an even number of 1 bits

Explicitly Setting Condition Codes: Compare

- **`cmp{b,w,l,q}`** ***`Src2, Src1`***
- **`cmpq src,dst`** computes **`t = dst-src`**, ignoring the result
  - And sets condition codes along the way, like **`subq`** would!
  - Follows the rules we saw on the previous slide for arithmetic instructions
  - **Beware the order of the `cmp` operands!**

- Use cases
  - **ZF set** if **`dst == src`**
  - **SF set** if **`(dst-src) < 0`** (as signed), i.e., src **`> dst`** in a signed comparison!
  - **CF and OF** used mostly in combinations with others (see in a few slides)

Explicitly Setting Condition Codes: Test

- `test{b,w,l,q}` *`Src2,Src1`*
- `testq src,dst` computes `t = dst&src`, ignoring the result!
  - And sets condition codes like `andq` would (order doesn't matter here)
  - So again, same rules as arithmetic instructions

- Use cases
  - **ZF set** when `dst&src == 0`, i.e., `a` and `b` have no bits in common
  - **SF set** when `dst&src < 0`
- Useful when doing bit masking
  - E.g., `x & 0x1`, to know whether `x` is even or odd
  - If the result of the `&` is 0, it's even, if 1, it's odd

Step 2: Reading Condition Codes

- Cannot read condition codes directly; instead observe via instructions
  - And generally observe ***combinations*** of condition codes, not individual ones

- Example: the **`setX`** family of instructions
  - Write single-byte destination register based on combinations of condition codes
    - **`set{e, ne, s, …} D`** where D is a 1-byte register
    - Example: **`sete %al`**
      - means: **`%al`**=1 if flag ZF is set, **`%al`**=0 otherwise

Using condition codes for comparison

- setle – Less than or equal (signed)
  - Combination of condition codes: `(SF^OF)|ZF`
  - SF - Sign Flag (true if negative)
  - OF – Overflow Flag (true if signed overflow occurred)
  - ZF – Zero Flag (true if result is zero)

- All of the combos expect to be run after a `cmp src,dst`
  - `dst <= src` `(runs dst-src)`
    - If:
      - The result is zero – `src` and `dst` were equal
    - OR if one but not both:
      - The result is negative (and didn't overflow) – src was larger than dst
      - The result overflowed (and is positive) – dst is negative, src is positive

Condition codes combinations

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

Note: suffixes do not indicate operand sizes, but rather conditions

These same suffixes will come back when we see other instructions that read condition codes.

Expect to be run after a **`cmp`**

Step 2: Reading Condition Codes

- `setX` (and others) read the current state of condition codes
  - Whatever it is, and whichever instruction changed it last
- So when you see (for example) `setne`, work backwards!
  - Look at previous instructions, to find the last one to change conditions
  - Then you'll know the two values that were compared
  - Ignore instructions that don't touch condition codes (like moves)
- Usually you'll see a `cmpX` (or `testX`, or arithmetic) right before
  - But not always, so know what to do in general

What do you need to know?

- 90%+ of the time
  - `cmp` instruction followed by `setX` instruction (or a branch, next lecture)
  - Don't have to think about condition codes at all!
  - Think of as `dst X src`
    - `dst <= src` or `dst != src` etc.

- 10% or less of the time
  - Arbitrary arithmetic instruction sets the condition codes
    - Or `testq` sets the condition codes
  - Followed by a `setX` or branch (next section)
  - And you actually have to think about which condition codes are set to figure out what the assembly is doing, which can be challenging

Break + Practice

`op src, dst`

- `setX` asks the question: "Is destination **X** source?"
  - Usually condition codes from "`cmp source, destination`"
  - Don't have to care about the exact values of the condition codes though
    - Just understand the logic

```
%sil = 0x01

%dil = 0xF0


cmp %sil, %dil
seta  %al
sete  %al
setge %al
```

SetX	Description
`sete`	Equal / Zero
`setne`	Not Equal / Not Zero
`sets`	Negative
`setns`	Nonnegative
`setg`	Greater (Signed)
`setge`	Greater or Equal (Signed)
`setl`	Less (Signed)
`setle`	Less or Equal (Signed)
`seta`	Above (unsigned)
`setb`	Below (unsigned)

Break + Practice

`op src, dst`

- `set`**X** asks the question: "Is destination **X** source?"
  - Usually condition codes from "`cmp source, destination`"
  - Don't have to care about the exact values of the condition codes though
    - Just understand the logic

`%sil = 0x01`

`%dil = 0xF0`

```
      cmp %sil, %dil
TRUE  seta  %al  # is %dil ABOVE %sil (unsigned)
FALSE sete  %al  # is %dil EQUAL to %sil
FALSE setge %al  # is %dil GREATER or EQUAL to %sil (signed)
```

Outline

- Condition Codes
- **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)

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

```
      movq 8(%rsp),%rdx
      movq 16(%rsp),%rax
      cmpq %rax,%rdx
      jle .L9
      movl %rdx,%rax
.L9:
      ...other instructions...
```

Jump if rdx <= rax

Label for the jump

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
`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)
`jae`	`~CF`	Above or Equal (unsigned)
`jb`	`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

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

```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;
}
```

- 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   # cmp x:y
   jle     .L2          # x <= y
   movq    %rdi, %rax
   subq    %rsi, %rax
   jmp     .L3
.L2:                    # jle target
   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    # cmp x:y
   jle      .L2           # x <= y
   movq     %rdi, %rax
   subq     %rsi, %rax
   jmp      .L3
.L2:                      # jle target
   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   # cmp x:y
   jle     .L2          # x <= y
   movq    %rdi, %rax
   subq    %rsi, %rax
   jmp     .L3
.L2:                    # jle target
   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   # cmp x:y
   jle     .L2          # x <= y
   movq    %rdi, %rax
   subq    %rsi, %rax
   jmp     .L3
.L2:                    # jle target
   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   # cmp x:y
   jle     .L2          # x <= y
   movq    %rdi, %rax
   subq    %rsi, %rax
   jmp     .L3
.L2:                    # jle target
   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    # cmp x:y
   jle      .L2           # x <= y
   movq     %rdi, %rax
   subq     %rsi, %rax
   jmp      .L3
.L2:                      # jle target
   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, $\neq$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

a→%rdi, b→%rsi, c→%rax

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

```
   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

a→%rdi, b→%rsi, c→%rax

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

```
  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

a→%rdi, b→%rsi, c→%rax

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

```
  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

a→%rdi, b→%rsi, c→%rax

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

```
  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

a→%rdi, b→%rsi, c→%rax

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

```
  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)

a→%rdi, b→%rsi, c→%rax

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

```
  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)

a→%rdi, b→%rsi, c→%rax

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

```
  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

- Condition Codes
- 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
  1. 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

  2. Rewrite that with goto
  3. 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

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

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:                     # 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
```

“Do-While” assembly translation

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

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:                     # 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
```

Which instruction sets the condition codes for `jne`?

Logical shift right (`shrq`)

"Do-While" assembly translation

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

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:                     # 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
```

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

"Do-While" assembly translation

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

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:                     # 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
```

- **`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)

General "Do-While" Translation

- Body: {
  Statement$_1$;
  Statement$_2$;
  …
  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" conversion
- More optimized compiler translation

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;
}
```

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;
}
```

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 if %rdi is 0
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 if %rdi is 0
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 if %rdi is 0
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 if %rdi is 0
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 if %rdi is 0
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 if %rdi is 0
loop:
  add %rbx, %rax     # %rax += %rbx
  add $1, %rbx       # %rbx += 1
  cmp %rdi, %rbx
  jne loop           # while %rbx != %rdi
end:
  ret                # returns %rax
```

Outline

- Condition Codes
- 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 ← 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  # res = x
   subq    %rsi, %rax  # res = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx  # alt = y-x
   cmpq    %rsi, %rdi  # cmp 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)

a→%rdi, b→%rsi, c→%rax

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

```
movq $0, %rax     # clear reg
cmp %rsi, %rdi
movq $1, %rdx
setl %al
neg %rax
cmp %rsi, %rdi
cmovg %rdx, %rax
ret               # returns %rax
```

else if and else together (%al is %rax)

select output

Outline

- Condition Codes
- 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`**)!

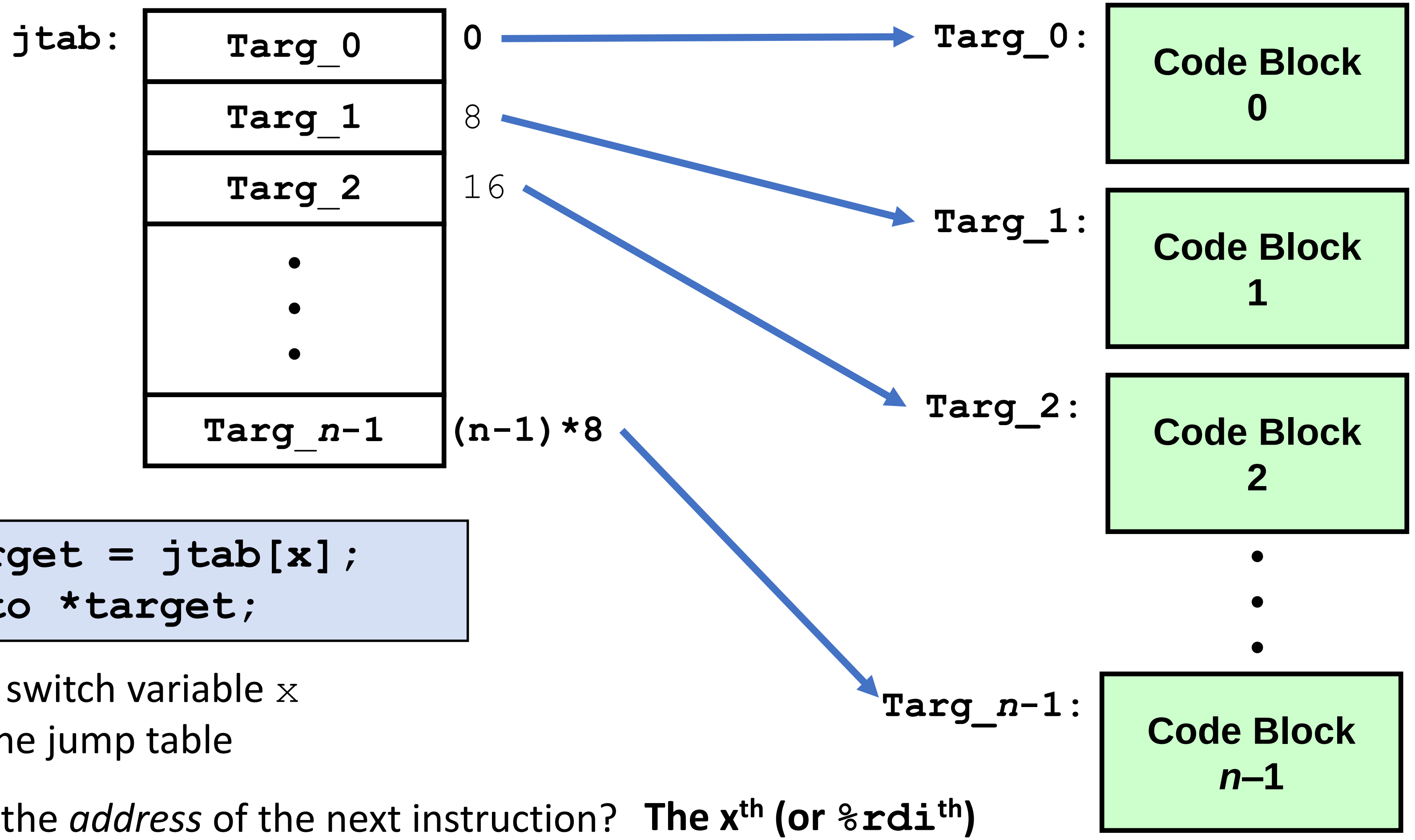
Jump tables

- 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) **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 xth (or `%rdi`th)**

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

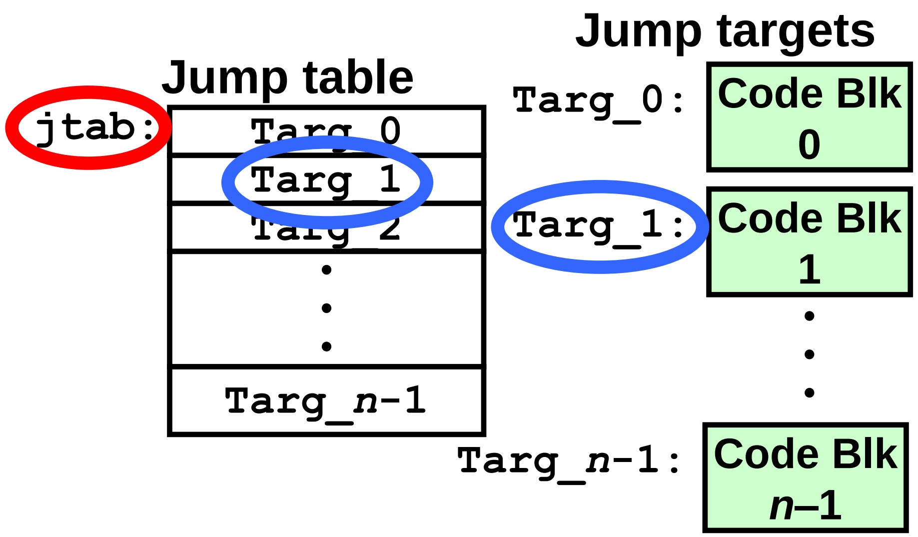
(addresses of code blocks)

```
.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;
```



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 derference)
```

Indirect jump: look up address in memory; jump there

Full assembly code for our example

```c
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;
}
```

```
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?

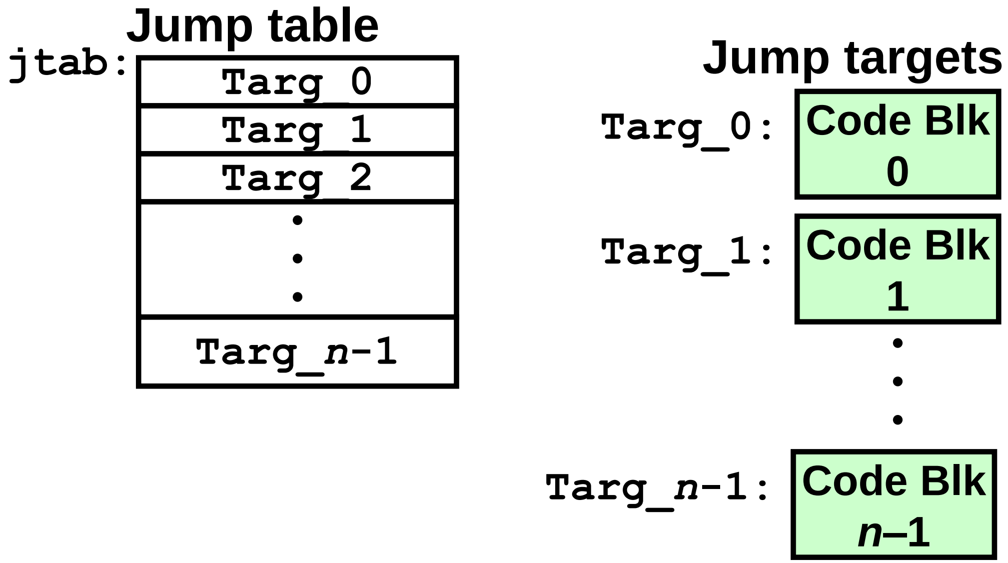
Object code: Jump Table

- 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
```

- E**x**amine **7** he**x**adecimal format "**g**iant words" (8-bytes each)
- Use command "**`help x`**" to get format documentation

```
0x400668:
    0x000000000040055c
    0x0000000000400538
    0x0000000000400540
    0x000000000040054a
    0x000000000040055c
    0x0000000000400553
    0x0000000000400553
```



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

```
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
```

```
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