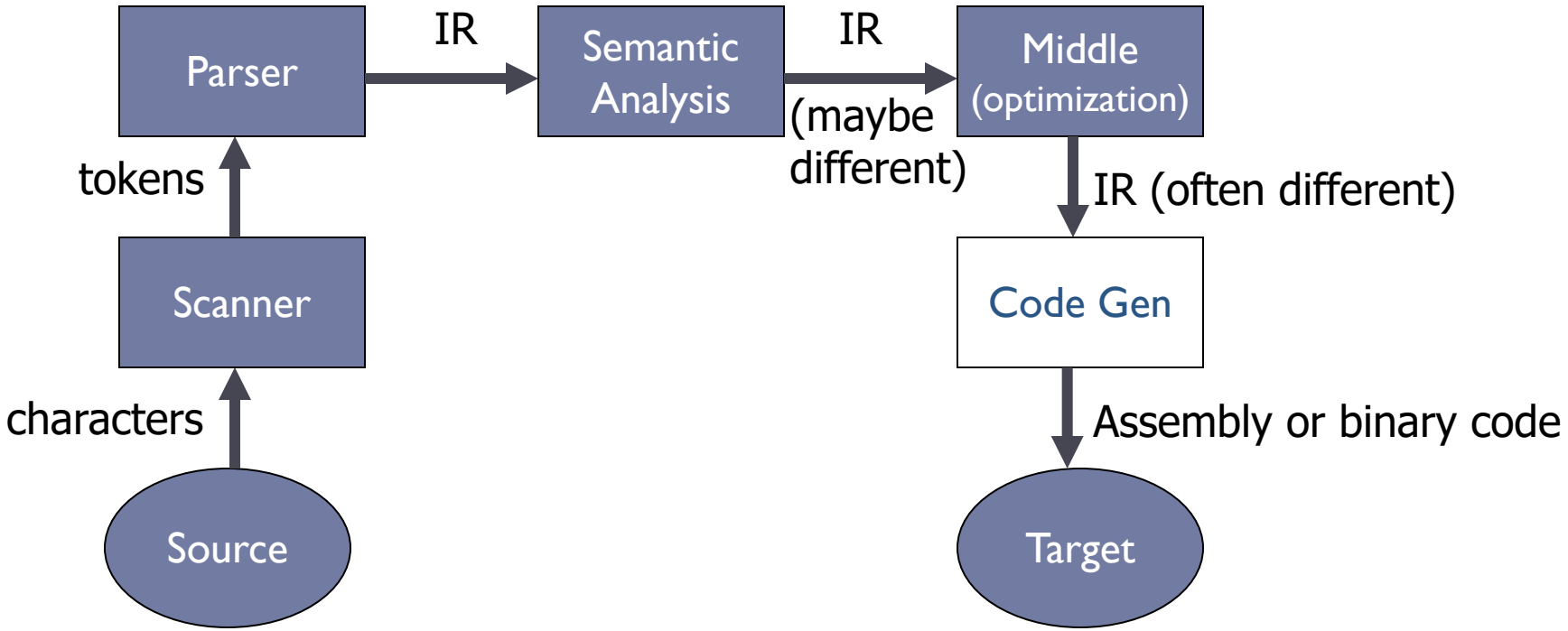
CSC 348 – Intro to Compilers
Lecture 7

Dr. David Zaretsky
david.zaretsky@depaul.edu

Agenda

- Code Generation for the x86-64 architecture
- Programming Assignment 5: Code Translation

Compiler Structure (review)



Code Translation Approach

- Code Generation can be quite complex.
- Best approach: focus on smaller constructs first (constants, identifiers, expressions, etc), then build up to larger constructs (statements, loops, functions).
- Code Explorer (https://godbolt.org) provides a quick and easy way to evaluate code translations from C/C++ to X86_64 architecture.
  - You can use this tool to help in setting up your code structures
  - Java & C/C++ have many similar constructs
- X64 Cheat Sheet – Brown University

Review: Variables

- For us, all data will be either:
  - In a stack frame (method local variables)
  - In an object (instance variables)
- Local variables accessed via %rbp
  - movq -16(%rbp),%rax
- Object instance variables accessed via an offset from an object address in a register

Conventions for Examples

- Examples show code snippets in isolation
- Register %rax used here as a generic example
  - Rename as needed for more complex code using multiple registers
- 64-bit data used everywhere
- A few peephole optimizations shown for a flavor of what's possible
  - Some might be easy to do in the compiler project

What we're skipping for now

- Real code generator needs to deal with many things like:
  - Which registers are busy at which point in the program
  - Which registers to spill into memory when a new register is needed and no free ones are available
  - Dealing with different sizes of data
  - Exploiting the full instruction set

Code Generation for Constants

- Source
  - 17
- x86-64

```
movq    $17, %rax
```

  - Idea: realize constant value in a register

- Optimization: if constant is 0

```
~~movq    $0, %rax~~
xorq    %rax,%rax
```

(some processors do better with movq $0,%rax – but this has changed over time)

Assignment Statement

- Source

  var = exp;

- x86-64

  <code to evaluate exp into, say, %rax>

  movq %rax, $offset_{var}$(%rbp)

Unary Minus

- Source

  -exp

- x86-64

  <code evaluating exp into %rax>

  negq %rax

- Optimization
  - Collapse -(-exp) to exp
  - Unary plus is a no-op

Binary +

- Source

  exp_1 + exp_2

- x86-64

  <code evaluating exp_1 into %rax>

  <code evaluating exp_2 into %rdx>

  addq %rdx,%rax

- Optimizations
- If exp_2 is a simple variable or constant, don't need to load it into a register. Instead:

  addq exp_2, %rax

- Change exp_1 + (-exp_2) into exp_1-exp_2
- If exp_2 is 1

  incq %rax

Binary -, *

- Same as +
  - Use subq for – (but not commutative!)
  - Use imulq for *
- Some optimizations
  - Use left shift to multiply by powers of 2
  - If your multiplier is slow or you've got free scalar units and multiplier is busy, you can do 10*x = (8*x)+(2*x)
    - But might be slower depending on microarchitecture
  - Use x+x instead of 2*x, etc. (often faster)
  - Use decq for x-1 (but check: subq $1 might be faster)

Signed Integer Division

- Ghastly on x86-64
  - Only works on 128-bit int divided by 64-bit int
    - (similar instructions for 64-bit divided by 32-bit in 32-bit x86)
  - Requires use of specific registers
- Source

  exp_1 / exp_2
- x86-64

```
<code evaluating exp1 into %rax ONLY>
<code evaluating exp2 into %rbx>
cqto                # extend to %rdx:%rax, clobbers %rdx
idivq  %rbx         # quotient in %rax, remainder in %rdx
```

Boolean Expressions

- What do we do with this?

  x > y

- It is an expression that evaluates to true or false
  - Could generate the value (0/1 or whatever the local convention is)
  - But normally we don't want/need the value – we're only trying to decide whether to jump
  - (Although for our project we might simplify and always produce the value)

Code for exp1 > exp2

- Basic idea: Generated code depends on context:
  - What is the jump target?
  - Jump if the condition is true or if false?
- Example: evaluate exp1 > exp2, jump on false, target if jump taken is L123

```
<evaluate exp1 to %rax>
<evaluate exp2 to %rdx>
cmpq    %rdx,%rax        # dst-src = exp1-exp2
jng     L123
```

Boolean Operators: !

- Source

  ! exp

- Context: evaluate exp and jump to L123 if false (or true)
- To compile ! just reverse the sense of the test: evaluate exp and jump to L123 if true (or false)

Boolean Operators: && and ||

- In C/C++/Java/C#/many others, these are short-circuit operators
  - Right operand is evaluated only if needed
- Basically, generate the if statements that jump appropriately and only evaluate operands when needed

Example: Code for &&

- Source

  if (exp_1 && exp_2) stmt

- x86-64

```
        <code for exp1>
        jfalse skip
        <code for exp2>
        jfalse skip
        <code for stmt>
skip:
```

Example: Code for ||

- Source

  if (exp_1 || exp_2) stmt

- x86-64

  <code for exp_1>
  j_{true} doit
  <code for exp_2>
  j_{false} skip
  doit: <code for stmt>
  skip:

Realizing Boolean Values

- If a boolean value needs to be stored in a variable or method call parameter, generate code needed to actually produce it
- Typical representations: 0 for false, +1 or -1 for true
  - C specifies 0 and 1; we'll use that
  - Best choice can depend on machine instructions & language; normally some convention is picked during the primeval history of the architecture

Boolean Values: Example

- Source

  var = bexp;

- x86-64

```
                <code for bexp>
                jfalse    genFalse
                movq    $1,%rax
                jmp     storeIt
genFalse:
                movq    $0,%rax                 # or xorq
storeIt:
                movq    %rax,offset_var(%rbp)   # generated by asg stmt
```

Better, If Enough Registers

- Source

  var = bexp;

- x86-64

```
                xorq    %rax,%rax                   # or movq $0,%rax
                <code for bexp>
                jfalse  store
                incq    %rax                        # or movq $1,%rax
store:
                movq    %rax,offsetvar(%rbp)    # generated by asg
```

  - Better: use movecc instruction to avoid conditional jump
  - Can also use conditional move instruction for sequences like x = y<z ? y : z

Better yet: setcc

- Source

  var = x < y;

- x86-64

```
movq    offset_x(%rbp),%rax     # load x
cmpq    offset_y(%rbp),%rax     # compare to y
setl    %al                       # set low byte %rax to 0/1
movzbq  %al,%rax                  # zero-extend to 64 bits
movq    %rax,offset_var(%rbp)   # gen. by asg stmt
```

Control Flow

- Basic idea: decompose higher level operation into conditional and unconditional gotos
- In the following, j_{false} is used to mean jump when a condition is false
  - No such instruction on x86-64
  - Will have to realize with appropriate sequence of instructions to set condition codes followed by conditional jumps
  - Normally don't need to actually generate the value "true" or "false" in a register
    - But this is a useful hack for the project

Control Flow

```
long max(long x, long y)
{
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

```
max:
   ???
   movq    %rdi, %rax
   ???
   ???
   movq    %rsi, %rax
   ???
   ret
```

Control Flow

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

```
long max(long x, long y)
{
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

Conditional jump

Unconditional jump

```
max:
  if x <= y then jump to else
  movq    %rdi, %rax
  jump to done
else:
  movq    %rsi, %rax
done:
  ret
```

Conditionals and Control Flow

- Conditional branch/*jump*
  - Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- Unconditional branch/*jump*
  - *Always* jump when you get to this instruction

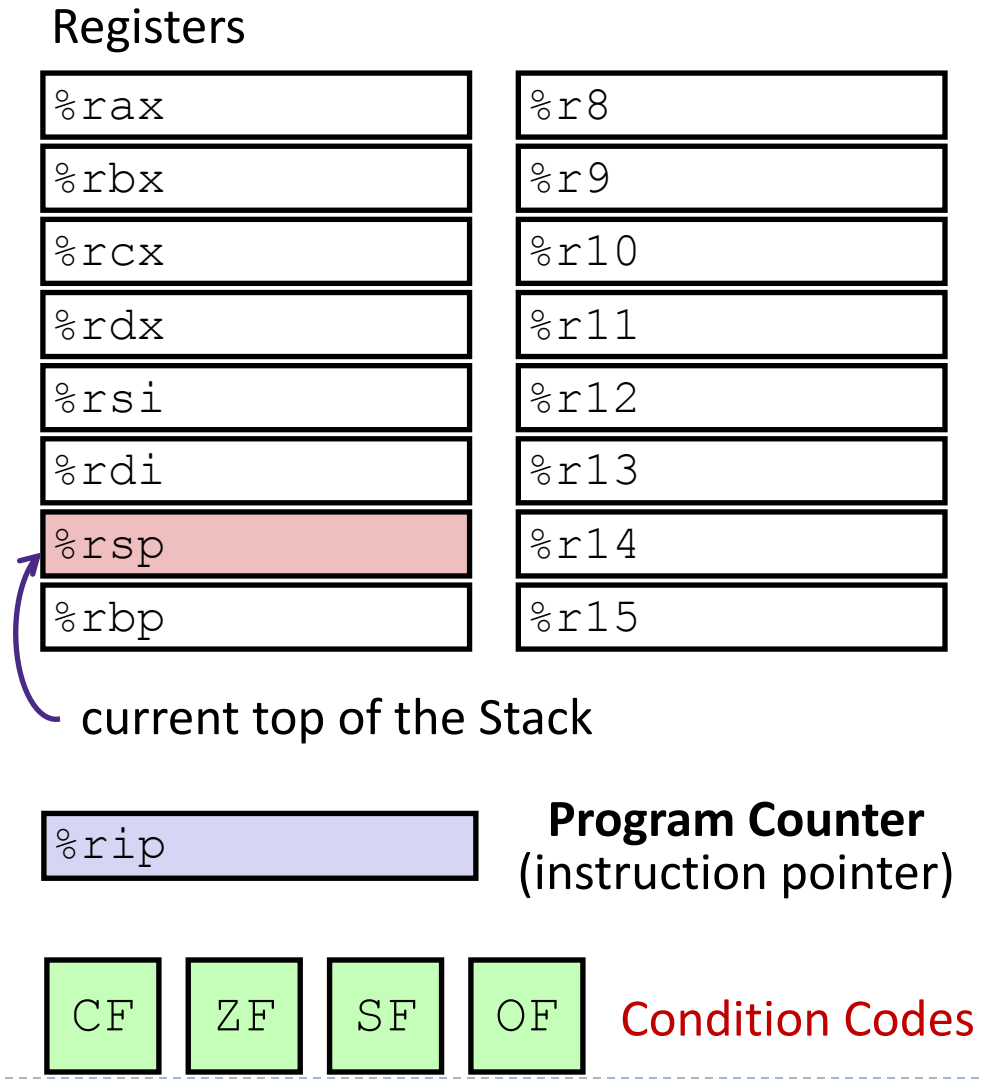
- Together, they can implement most control flow constructs in high-level languages:
  - `if` (*condition*) `then` {…} `else` {…}
  - `while` (*condition*) {…}
  - `do` {…} `while` (*condition*)
  - `for` (*initialization*; *condition*; *iterative*) {…}
  - `switch` {…}

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches

Processor State (x86-64, partial)

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



Condition Codes (Implicit Setting)

- *Implicitly* set by **arithmetic** operations
  - (think of it as side effects)
  - Example: **addq** `src, dst` ↔ `r = d+s`

  - **CF=1** if carry out from MSB (unsigned overflow)
  - **ZF=1** if `r==0`
  - **SF=1** if `r<0` (assuming signed, actually just if MSB is 1)
  - **OF=1** if two's complement (signed) overflow
    `(s>0 && d>0 && r<0)||(s<0 && d<0 && r>=0)`
  - *Not* set by `lea` instruction (beware!)

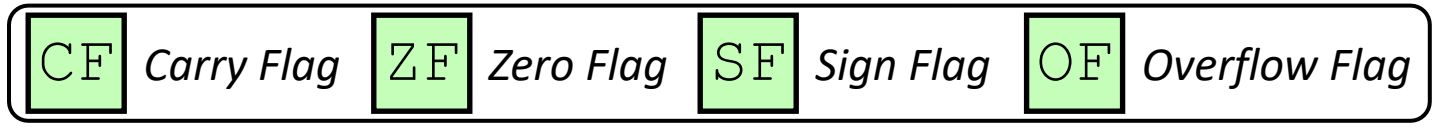
CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Condition Codes – Compare Instructions

- *Explicitly* set by **Compare** instruction
  - **`cmpq`** `src1, src2`
  - **`cmpq`** `a, b` sets flags based on `b-a`, but doesn't store

  - **`CF=1`** if carry out from MSB (used for unsigned comparison)
  - **`ZF=1`** if `a==b`
  - **`SF=1`** if `(b-a)<0` (signed)
  - **`OF=1`** if two's complement (signed) overflow

```
(a>0 && b<0 && (b-a)>0) ||
(a<0 && b>0 && (b-a)<0)
```



Condition Codes – Test Instruction

- *Explicitly* set by **Test** instruction
  - **`testq`** `src2, src1`
  - **`testq`** `a, b` sets flags based on `a&b`, but doesn't store
    - Useful to have one of the operands be a ***mask***
  - Can't have carry out (**`CF`**) or overflow (**`OF`**)
  - **`ZF=1`** if `a&b==0`
  - **`SF=1`** if `a&b<0` (signed)
  - Example: `testq %rax, %rax`
    - Tells you if (+), 0, or (−) based on ZF and SF

Using Condition Codes: Jumping

- j* Instructions
- Jumps to target (an address) based on condition codes

Instruction	Condition	Description
jmp *target*	1	Unconditional
je *target*	ZF	Equal / Zero
jne *target*	~ZF	Not Equal / Not Zero
js *target*	SF	Negative
jns *target*	~SF	Nonnegative
jg *target*	~(SF^OF)&~ZF	Greater (Signed)
jge *target*	~(SF^OF)	Greater or Equal (Signed)
jl *target*	(SF^OF)	Less (Signed)
jle *target*	(SF^OF)|ZF	Less or Equal (Signed)
ja *target*	~CF&~ZF	Above (unsigned “>”)
jb *target*	CF	Below (unsigned “<“)

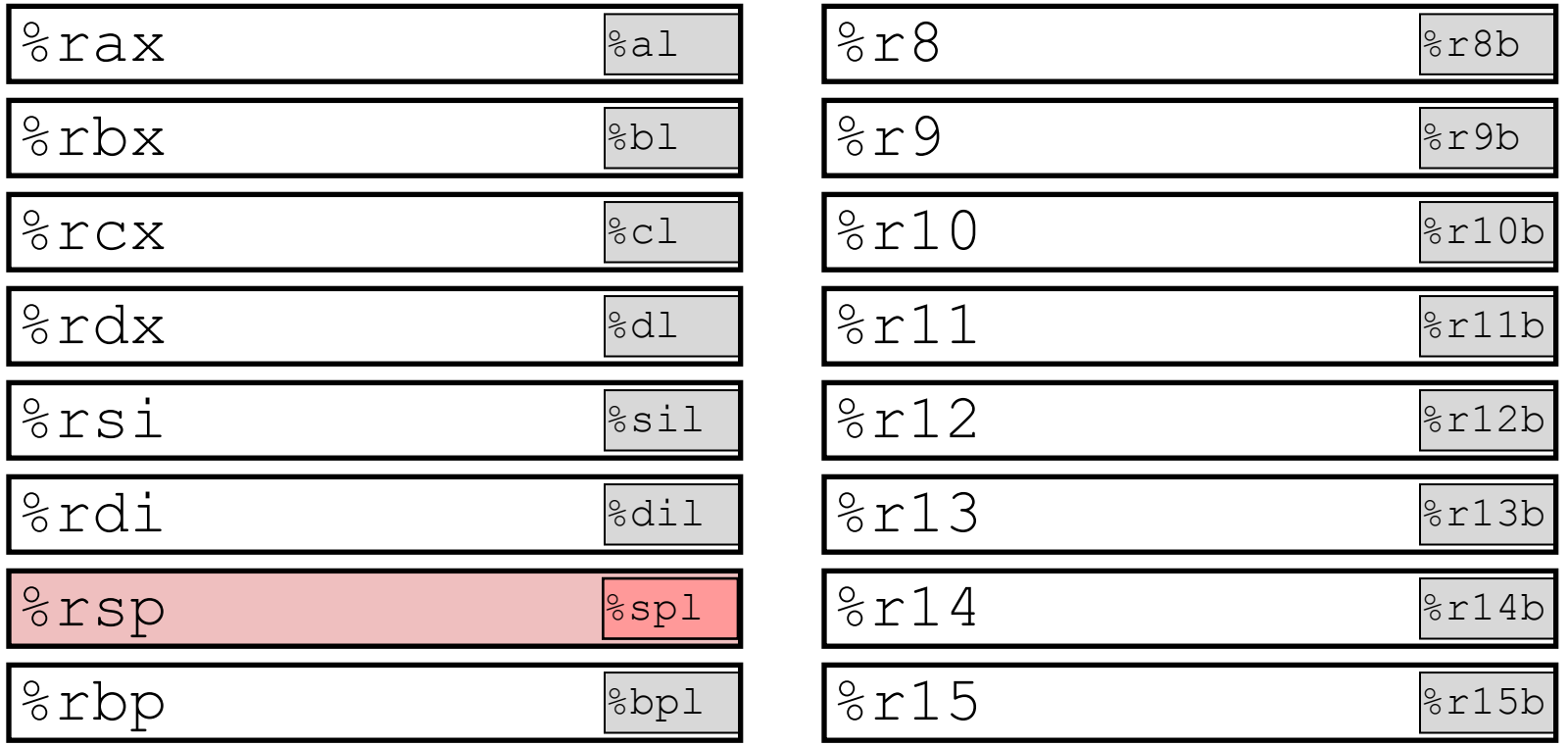
Using Condition Codes: Setting

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

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

Reminder: x86-64 Integer Registers

- Accessing the low-order byte:



Reading Condition Codes

- `set*` Instructions
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. `al`, `dl`) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

Register	Use(s)
`%rdi`	1st argument (`x`)
`%rsi`	2nd argument (`y`)
`%rax`	return value

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

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

Expressing with Goto Code

- Java allows **continue & break** with labels as means of transferring control
  - Closer to assembly programming style
  - Generally considered bad coding style

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

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

If

- Source

  if (cond) stmt

- x86-64

  <code evaluating cond>

  j_{false} skip

  <code for stmt>

skip:

  <code for stmt>

If-Else

- Source

  if (cond) $stmt_1$ else $stmt_2$

- x86-64

```
                <code evaluating cond>
                jfalse else
                <code for stmt1>
                jmp  done
else:           <code for stmt2>
done:
```

Jump Chaining

- Observation: naïve implementation can produce jumps to jumps (if-else if-…-else; or nested loops or conditionals, …)
- Optimization: if a jump has as its target an unconditional jump, change the target of the first jump to the target of the second
  - Repeat until no further changes
  - Often done in peephole optimization pass after initial code generation

Compiling Loops

- Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time
- Most important to consider:
  - When should conditionals be evaluated? (while vs. do-while)
  - How much jumping is involved?

C/Java code:

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

Assembly code:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

Compiling Loops

C/Java code:

```
while ( Test ) {
    Body
}
```

Goto version

```
Loop: if ( !Test )  goto Exit;
      Body
      goto Loop;
Exit:
```

- What are the Goto versions of the following?
  - Do…while: Test and Body
  - For loop: Init, Test, Update, and Body

Compiling Loops

While Loop:

JAVA:

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

x86-64:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

Do-while Loop:

JAVA:

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

x86-64:

```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```

While Loop (Optimized):

JAVA:

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

x86-64:

```
            testq %rax, %rax
            je    loopDone
loopTop:
            <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```

For Loop → While Loop

For Version

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

While Version

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

Caveat: C and Java have `break` *and* `continue`

- *Conversion works fine for* `break`
  - *Jump to same label as loop exit condition*
- *But not* `continue`*: would skip doing Update, which it should do with for-loops*
  - *Introduce new label at Update*

Other Control Flow: switch

- Naïve: generate a chain of nested if-else if statements
- Better: switch statement is intended to allow easier generation of O(1) selection, provided the set of switch values is reasonably compact
- Idea: create a 1-D array of jumps or labels and use the switch expression to select the right one
  - Need to generate the equivalent of an if statement to ensure that expression value is within bounds

Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

- Implemented with:
  - Jump table
  - Indirect jump instruction

```
public int switch_ex (int x, int y, int z) {
    int 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;
}
```

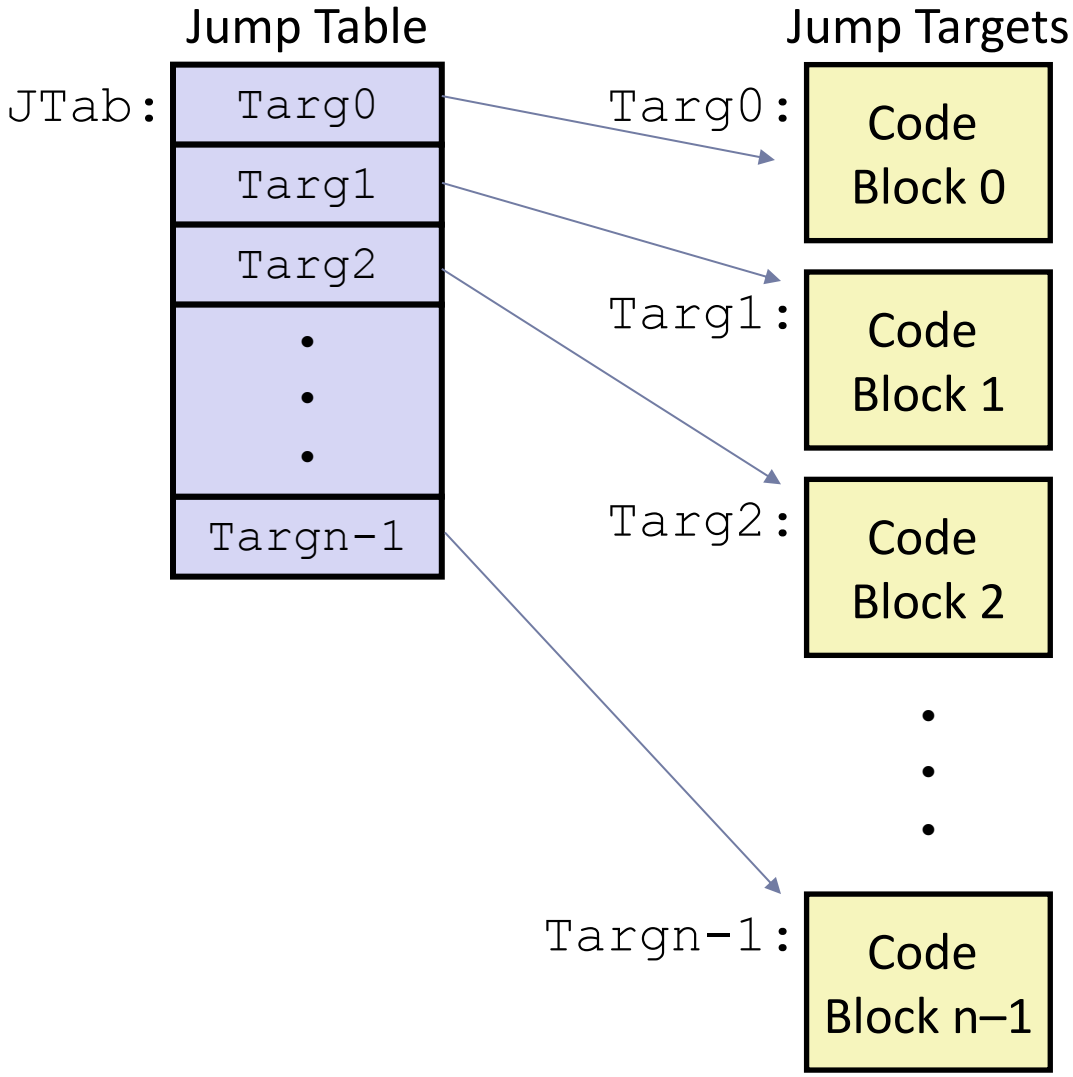
Jump Table Structure

Switch Form

```
switch (x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Approximate Translation

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



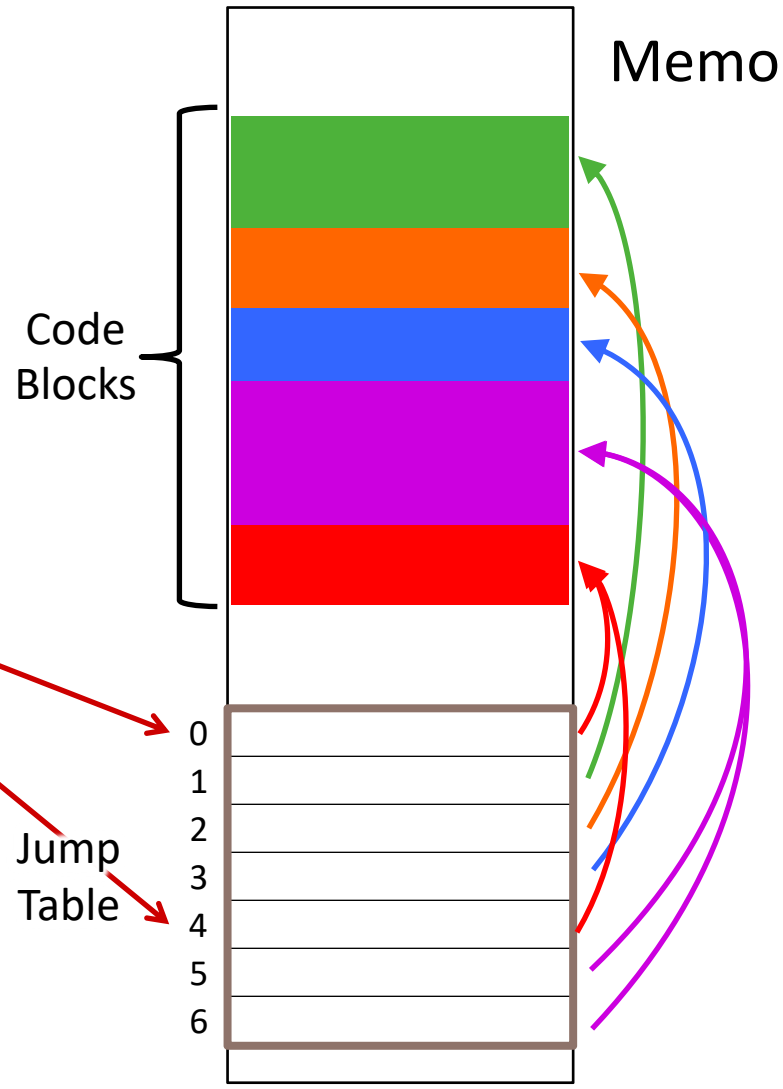
Jump Table Structure

Java / C code:

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

Use the jump table when x ≤ 6:

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



Switch Statement Example

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

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

Note compiler chose to not initialize w

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

jump **a**bove – unsigned > catches negative default cases

Switch Statement Example

```
public int switch_ex(int x, int y, int z)
{
    int w = 1;
    switch (x) {
      . . .
    }
    return w;
}
```

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

Indirect jump

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
```

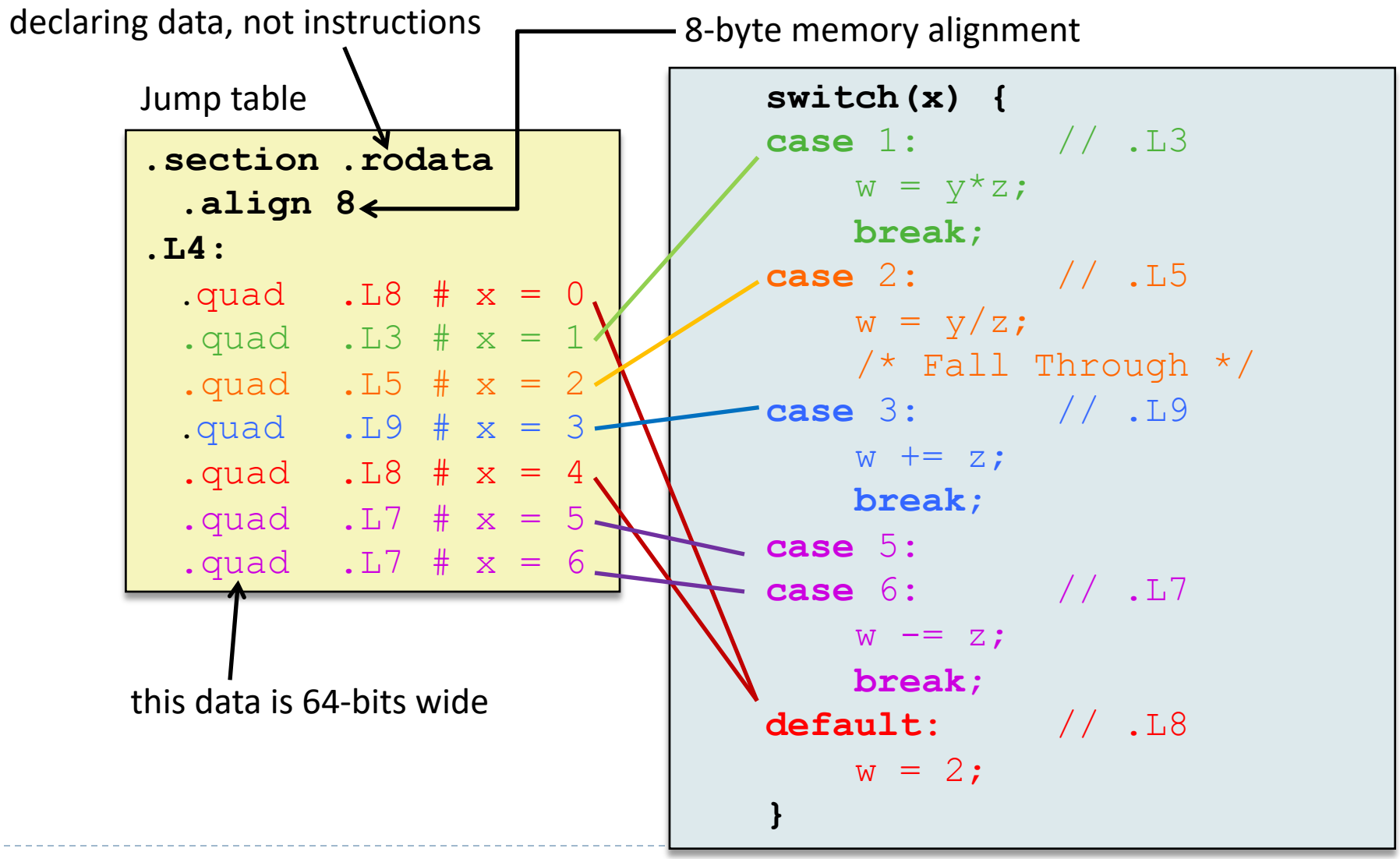
Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes (address)
  - Base address at .L4
- Direct jump: jmp .L8
  - Jump target is denoted by label .L8
- Indirect jump: 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 $0 \le x \le 6$

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
```

Jump Table


declaring data, not instructions
8-byte memory alignment
Jump table
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
this data is 64-bits wide
switch(x) {
case 1: // .L3
w = y*z;
break;
case 2: // .L5
w = y/z;
/* Fall Through */
case 3: // .L9
w += z;
break;
case 5:
case 6: // .L7
w -= z;
break;
default: // .L8
w = 2;
}

Code Blocks (x == 1)

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

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

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

Handling Fall-Through

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

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

```
case 3:
        w = 1;
merge:
        w += z;
```

More complicated choice than "just fall-through" forced by "migration" of `w = 1;`

- *Example compilation trade-off*

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

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

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

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

Code Blocks (rest)

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

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

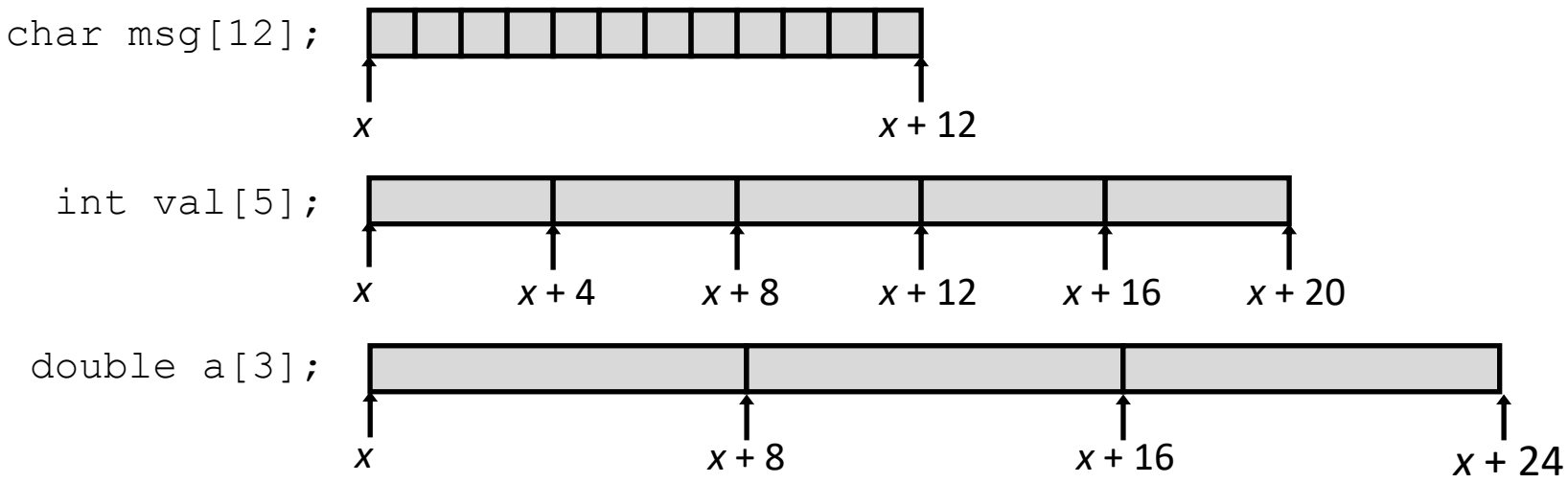
```
.L7:                    # Case 5,6:
  movl   $1, %eax       #   w = 1
  subq   %rdx, %rax     #   w -= z
  ret
.L8:                    # Default:
  movl   $2, %eax       #   2
  ret
```

Arrays

- Several variations
- C/C++/Java
  - 0-origin: an array with n elements contains variables a[0]…a[n-1]
  - 1 dimension (Java); 1 or more dimensions using row major order (C/C++)
- Key step is evaluate subscript expression, then calculate the location of the corresponding array element

Array Allocation

- Basic Principle
  - `T A[N];` → array of data type `T` and length `N`
  - *Contiguously* allocated region of `N*sizeof(T)` bytes



0-Origin 1-D Integer Arrays

- Source

  $exp_1[exp_2]$

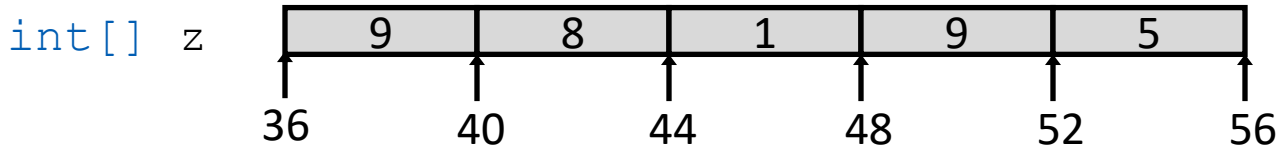
- x86-64

  <evaluate exp_1 (array address) in %rax>

  <evaluate exp_2 in %rdx>

  address is (%rax,%rdx,8) # if 8 byte elements

Array Accessing Example




```
public int get_digit(int[] z, int digit)
{
  return z[digit];
}
```

```
get_digit:
  movl (%rdi,%rsi,4), %eax   # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi+4*%rsi`, so use memory reference `(%rdi,%rsi,4)`

Array Loop Example

```
zi = 10*0 + 9 = 9
zi = 10*9 + 8 = 98
zi = 10*98 + 1 = 981
zi = 10*981 + 9 = 9819
zi = 10*9819 + 5 = 98195
```

```
int foo(int[] z, int len)
{
  int i;
  int zi = 0;
  for (i = 0; i < len; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

Array Loop Implementation

```
int foo(int[] z, int len)
{
  int i = 0;
  int zi = 0;
  while (i < len) {
    zi = 10 * zi + z[i];
    i++;
  }
  return zi;
}
```

```
        movl $0, -4(%rbp)       # i = 0
        movl $0, -8(%rbp)       # zi = 0
.L3:    movl -4(%rbp), %eax     # i
        cmpl -28(%rbp), %eax    # ( i < len )
        jge .L2
        movl -8(%rbp), %edx     # zi
        movl %edx, %eax
        sall $2, %eax           # zi = zi * 8
        addl %edx, %eax         # zi += zi
        addl %eax, %eax         # zi += zi
        movl %eax, %ecx
        movl -4(%rbp), %eax     # i
        cltq
        leaq 0(,%rax,4), %rdx
        movq -24(%rbp), %rax    # z
        addq %rdx, %rax
        movl (%rax), %eax       # z[i]
        addl %ecx, %eax         # zi += z[i]
        movl %eax, -8(%rbp)
        addl $1, -4(%rbp)       # i++
        jmp .L3
.L2:
```

Next…

- Functions / Methods
- Class structures
- Assignment 5: Code Translation