Assembly Programming IV
CSE 351 Spring 2017

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis
Administrivia

- Homework 2 due this Wednesday (4/19)
- Lab 2 (x86-64) due next Wednesday (4/26)
  - Learn to read x86-64 assembly and use GDB
Review

- 3 ways to set condition codes are: `cmp`, `test`, arithmetic instructions
- 2 ways to use condition code are: `jmp`, `set`
- Does `leaq` set condition codes? no
The `leaq` Instruction

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

Does the `leaq` instruction go to memory?

“`leaq` – it just does math”
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- **Loops**
- Switches
Expressing with Goto Code

- C allows goto as means of transferring control (jump)
  - Closer to assembly programming style
  - Generally considered bad coding style
  - This is just to help you understand assembly code generated by the compiler. Do NOT use goto in your C code!
Compiling Loops

C/Java code:
```java
while ( sum != 0 ) {
    <loop body>
}
```

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

- 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?
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:**

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

**Do-while Loop:**

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

**While Loop (ver. 2):**

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

x86-64:
```
loopTop:  
    testq %rax, %rax  
    je    loopDone
    <loop body code>
    jmp   loopTop

loopDone:  
```

All jump instructions update the program counter (pc).
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
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4
- How to implement this?

```c
long switch_ex (long x, long y, long z) {
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;<br>
            break;
        case 2:
            w = y/z;<<br>  /* Fall Through */
        case 3:
            w += z;<
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```
Jump Tables

- Compiles sometimes Implement switch statements with:
  - Jump table
  - Uses the Indirect jump instruction

- Why? When?
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

<table>
<thead>
<tr>
<th>Target</th>
<th>Code Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ0</td>
<td>Block 0</td>
</tr>
<tr>
<td>Targ1</td>
<td>Block 1</td>
</tr>
<tr>
<td>Targ2</td>
<td>...</td>
</tr>
<tr>
<td>Targn-1</td>
<td>Block n-1</td>
</tr>
</tbody>
</table>

Jump Targets

- Target 0: Code Block 0
- Target 1: Code Block 1
- Target 2: Code Block 2
- Target n-1: Code Block n-1

Jump Table:

- Target = JTab[x];
- goto target;

Addresses (8 bytes wide)

Like an array of pointers
Jump Table Structure

C code:

```c
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 \( x \leq 6 \):

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

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

### Register Use(s)

- `%rdi` 1\(^{st}\) argument (`x`)
- `%rsi` 2\(^{nd}\) argument (`y`)
- `%rdx` 3\(^{rd}\) argument (`z`)
- `%rax` Return value

Note: compiler chose to not initialize `w`

### Assembly Code Example

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

Jump to default case if `x` ≥ 6 (unsigned)

Jump above – unsigned > catches negative default cases

Take a look!

[https://godbolt.org/g/DnOmXb](https://godbolt.org/g/DnOmXb)
Switch Statement Example

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

Jump table

```c
.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
```

Indirect jump

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

de-reference Mem operator and store that in %rip
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 $0 \leq x \leq 6$

**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

Jump table

```
.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;
}
```

this data is 64-bits wide

declaring data, not instructions

8-byte memory alignment
Code Blocks (x == 1)

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Handling Fall-Through

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

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

- Example compilation trade-off
Code Blocks \((x == 2, x == 3)\)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1\textsuperscript{st} argument ((x))</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} argument ((y))</td>
</tr>
<tr>
<td>%rdx</td>
<td>3\textsuperscript{rd} argument ((z))</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

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

.L8:  # Default:
    movl $2, %eax  # 2
    ret
Roadmap

C:
```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:
```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

Assembly language:
```
get_mpg:       pushq  %rbp
               movq  %rsp, %rbp
              ...     popq   %rbp
               ret
```

Machine code:
```
011110100000110000
100011010000010000000010
10001001110000010
1100000111111010000011111
```

OS:
```
Windows 8

Mac
```

Memory & data
Integers & floats
x86 assembly

Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Mechanisms required for procedures

1) Passing control
   - To beginning of procedure code
   - Back to return point

2) Passing data
   - Procedure arguments
   - Return value

3) Memory management
   - Allocate during procedure execution
   - Deallocate upon return
   ❖ All implemented with machine instructions!
   - An x86-64 procedure uses only those mechanisms required for that procedure
Questions to answer about Procedures

- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return?

- To answer some of these questions, we need a call stack...
Procedures

- **Stack Structure**

- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data

- **Register Saving Conventions**

- **Illustration of Recursion**
Memory Layout

- **Instructions**
  - Large constants (e.g., "example")
  - Program code

- **Static Data**
  - Static variables (including global variables (C))

- **Dynamic Data (Heap)**
  - Variables allocated with `new` or `malloc`

- **Stack**
  - Local variables; procedure context

- **Literals**

**Memory Addresses**

- **Low Addresses**
  - 0
  - \( 0 \times 004 \ldots 0d \)

- **High Addresses**
  - \( 2^{N-1} \)
## Memory Permissions

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
<th>Managed/Initialized</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stack</td>
<td>Managed “automatically”</td>
<td>Writable; Not Executable</td>
</tr>
<tr>
<td></td>
<td>Dynamic Data (Heap)</td>
<td>Managed by programmer</td>
<td>Writable; Not Executable</td>
</tr>
<tr>
<td></td>
<td>Static Data</td>
<td>Initialized when process starts</td>
<td>Writable; Not Executable</td>
</tr>
<tr>
<td></td>
<td>Literals</td>
<td>Initialized when process starts</td>
<td>Read-Only; Not Executable</td>
</tr>
<tr>
<td></td>
<td>Instructions</td>
<td>Initialized when process starts</td>
<td>Read-Only; Executable</td>
</tr>
</tbody>
</table>

**segmentation faults?**

Accessing memory in a way that you are not allowed to.

Grow towards each other to maximize use of space.
x86-64 Stack

- Region of memory managed with stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”

- Register `%rsp` contains lowest stack address
  - `%rsp` = address of top element, the most-recently-pushed item that is not-yet-popped

**Stack Pointer:** `%rsp`
x86-64 Stack: Push

- `pushq src`
  - Fetch operand at `src`
    - `Src` can be `reg`, `memory`, `immediate`
  - **Decrement** `%rsp` by 8
  - Store value at address given by `%rsp`

- **Example:**
  - `pushq %rcx`
  - Adjust `%rsp` and store contents of `%rcx` on the stack

Stack Pointer: `%rsp`
x86-64 Stack: Pop

- `popq dst`
  - Load value at address given by `%rsp`
  - Store value at `dst` (must be register)
  - **Increment** `%rsp` by 8

- **Example:**
  - `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`

Those bits are still there; we’re just not using them.
Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Procedure Call Overview

- **Callee** must know where to find the arguments.
- **Callee** must know where to find the return address.
- **Caller** must know where to find the return value.
- **Caller** and **Callee** run on the same CPU, so use the same registers.
  - How do we deal with register reuse?
- Unneeded steps can be skipped (e.g., no arguments).
Procedure Call Overview

- The convention of where to leave/find things is called the calling convention (or procedure call linkage)
  - Details vary between systems
  - We will see the convention for x86-64/Linux in detail
  - What could happen if our program didn’t follow these conventions?
Code Examples

```c
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

Compiler Explorer: https://godbolt.org/g/52Sqxj

```assembly
0000000000400540 <multstore>:
    400540: push %rbx           # Save %rbx
    400541: movq %rdx,%rbx      # Save dest
    400544: call 400550 <mult2> # mult2(x,y)
    400549: movq %rax,(%rbx)    # Save at dest
    40054c: pop %rbx            # Restore %rbx
    40054d: ret                 # Return

long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```assembly
0000000000400550 <mult2>:
    400550: movq %rdi,%rax      # a
    400553: imulq %rsi,%rax     # a * b
    400557: ret                 # Return
```
Procedure Control Flow

- Use stack to support procedure call and return

Procedure call: \texttt{call label}

1) Push return address on stack (why? which address?)
2) Jump to \texttt{label}
Procedure Control Flow

- Use stack to support procedure call and return

- **Procedure call:** `call label`
  1) Push return address on stack *(why? which address?)*
  2) Jump to `label`

- **Return address:**
  - Address of instruction immediately after `call` instruction
  - Example from disassembly:

```
400544: call 400550 <mult2>
400549: movq %rax,(%rbx)
```

  Return address = `0x400549`

- **Procedure return:** `ret`
  1) Pop return address from stack *(why?)*
  2) Jump to address

  next instruction happens to be a move, but could be anything
Procedure Call Example (step 1)

00000000000400540 <multstore>:
  ...
  400544: call 400550 <mult2>
  400549: movq %rax, (%rbx)
  ...

00000000000400550 <mult2>:
  400550: movq %rdi, %rax
  ...
  400557: ret
Procedure **Call Example** (step 2)

```
00000000000400540 <multstore>:
    .
    .
    400544: call 400550 <mult2>
    400549: movq %rax, (%rbx)
    .

00000000000400550 <mult2>:
    400550: movq %rdi, %rax
    .
    .
    400557: ret
```
Procedure **Return** Example (step 1)

```
0000000000400540 <multstore>:
   
400544: call 400550 <mult2>
400549: movq %rax, (%rbx)
   
0000000000400550 <mult2>:
   
400550: movq %rdi, %rax
   
400557: ret
```
Procedure **Return Example** (step 2)

```
000000000000400540 <multstore>:
    .
    .
400544: call 400550 <mult2>
400549: movq %rax,(%rbx)
    .
    .
```

```
000000000000400550 <mult2>:
    400550: movq %rdi,%rax
    .
    .
400557: ret
```