x86-64 Programming III & The Stack
CSE 351 Winter 2018

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan

http://xkcd.com/1652/
Administrative

❖ Homework 2 (x86) due Monday (1/29)
❖ Lab 2 due next Friday (2/2)

❖ Midterm: 2/5
  ▪ You will be provided a fresh reference sheet
  ▪ Must bring your UW Student ID to the exam!
  ▪ Topics are Lectures 1 – 12, Ch 1.0 – 3.7
  ▪ Review packet / suggested practice problems to be posted soon
x86 Control Flow

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

```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);  // cmp
    if (ntest) goto Else;  // jle
    result = x - y;
    goto Done;  // jmp
Else:
    result = y - x;
Done:
    return result;
}
```

- **C allows** `goto` **as means of transferring control** (`jump`)
  - Closer to assembly programming style
  - Generally considered bad coding style
Compiling Loops

C/Java code:

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

Assembly code:

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

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

What are the Goto versions of the following?

- Do...while: Test and Body
- For loop: Init, Test, Update, and Body

C/Java code:

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

Goto version:

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

- Do...while
  ```c
  Loop: Body
  if (Test) goto Loop;
  ```

- For loop
  ```c
  Init
  Loop: if (!Test) goto Exit;
    Body
    Update
    goto Loop;
  Exit:
  ```
Compiling Loops

**While Loop:**

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

x86-64:

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

loopDone:
```

**Do-while Loop:**

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

x86-64:

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

loopDone:
```

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

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

x86-64:

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

loopDone:
```
For Loop $\rightarrow$ While Loop

For Version

\begin{verbatim}
for (Init; Test; Update)
  Body
\end{verbatim}

While Version

\begin{verbatim}
Init;
while (Test) {
  Body
  Update;
}
\end{verbatim}

Caveat: C and Java have break and continue

\begin{itemize}
  \item Conversion works fine for break
    \begin{itemize}
      \item Jump to same label as loop exit condition
    \end{itemize}
  \item But not continue: would skip doing Update, which it should do with for-loops
    \begin{itemize}
      \item Introduce new label at Update
    \end{itemize}
\end{itemize}
x86 Control Flow

❖ Condition codes
❖ Conditional and unconditional branches
❖ Loops
❖ **Switches**
long switch_ex (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
Jump Table Structure

Switch Form

```c
switch (x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
        ...
    case val_{n-1}:
        Block n-1
}
```

Approximate Translation

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

Jump Table

<table>
<thead>
<tr>
<th>JTab</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>Targ_{n-1}</th>
</tr>
</thead>
</table>

Jump Targets

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targ2: Code Block 2
- Targ_{n-1}: Code Block n-1

Addresses (8-bytes wide)
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` 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 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
```

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

declaring data, not instructions

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
```

8-byte memory alignment

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;
    ...
}

.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)

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

.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

<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>
Code Blocks (rest)

```
switch (x) {
    . . .
    case 5:  // .L7
        w -= z;
        break;
    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
```

<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>
Roadmap

C:

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

Java:

```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:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:

OS:

- Windows 10
- OS X Yosemite

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
Procedures

❖ Stack Structure

❖ Calling Conventions
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data

❖ Register Saving Conventions

❖ Illustration of Recursion
Simplified Memory Layout

- **High Addresses**
  - Stack
  - Dynamic Data (Heap)
  - Static Data
  - Literals
  - Instructions

- **Low Addresses**
  - stk
  - Dynamic Data
  - Static Data
  - Literals
  - Instructions

- **Memory Addresses**
  - Local variables; procedure context
  - Variables allocated with `new` or `malloc`
  - `static` variables (including global variables (C))
  - Large constants (e.g. “example”)
  - Program code

- **Address Range**: $2^{N-1}$ to 0
Memory Permissions

- Stack: writable; not executable
  - Managed “automatically” (by compiler)

- Dynamic Data (Heap): writable; not executable
  - Managed by programmer

- Static Data: writable; not executable
  - Initialized when process starts

- Literals: read-only; not executable
  - Initialized when process starts

- Instructions: read-only; executable
  - Initialized when process starts

segmentation faults?
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
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` → 8

- Stack grows down
- High addresses
- Increasing addresses
- Stack grows down
- Low addresses
- `0x00...00`
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.