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

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 allows goto as means of transferring control (jump)
  ▪ Closer to assembly programming style
  ▪ Generally considered bad coding style

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?
Compiling Loops

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

Do-while Loop:
```c
do {
    <loop body>
    while (sum != 0)
} while (sum != 0);
```

While Loop (ver. 2):
```c
while (sum != 0) {
    <loop body>
}
```

x86 Control Flow

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

Jump Table Structure

<table>
<thead>
<tr>
<th>Switch Form</th>
<th>Jump Table</th>
<th>Jump Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch (x)</td>
<td>JTab[0]</td>
<td>Code Block 0</td>
</tr>
<tr>
<td>case val_0: Block 0</td>
<td>Target 1</td>
<td>Code Block 1</td>
</tr>
<tr>
<td>case val_1: Block 1</td>
<td>Target 2</td>
<td></td>
</tr>
<tr>
<td>case val_n-1: Block n-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

For Loop → While Loop

Caveat: C and Java have break and continue
- Conversion works fine for break
- But not continue: would skip doing Update, which it should do with for-loops
- Introduce new label at Update
Switch Statement Example

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

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 ≤ x ≤ 6

Handling Fall-Through

```c
long w = 1;
switch (x) {  
  ...  
  case 2: // .L6  
    w = y/z;  
    break;  
    /* Fall Through */  
  case 3: // .L9  
    w += y;  
    break;  
  ...  
}  
```

- More complicated choice than "just fall-through" forced by "migration" of w = 1
- *Example compiler trade-off*
Code Blocks (x == 2, x == 3)

```
long x = 1;
    .
    .
switch (x) {
    .
    .
    case 2: // .L5
        w = y/z;  // Fall Through */
    case 3: // .L6
        w += z;
    break;
    .
    .
}
```

### Code Blocks (rest)

```java
switch (x) {
    .
    .
    case 2: // .L5
        w = y/z;
        break;  /* Fall Through */
    case 3: // .L9
        w += z;
        break;
    default: // .L8
        w = 2;
    }
```

### Roadmap

- **Assembly language:**
  - `get`: return v[t];
  - `set`: new Car();

- **Machine code:**
  - 01110100000011000
  - 100011010000010000000010
  - 1000100111000010
  - 110000011111101000011111

- **Computer system:**
  - Windows 10

### 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

### Simplified Memory Layout

- **High Addresses**
  - Program code
  - Stack
  - Variables allocated with new or malloc
  - Static variables (including global variables (C))
  - Large constants (e.g. "example")
  - Program code
- **Low Addresses**
  - Memory Addresses
  - Variables:
    - Stack: local variables; procedure context
    - Dynamic Data (Heap): variables allocated with new or malloc
    - Static Data: static variables (including global variables (C))
    - Literals: large constants (e.g. "example")
    - Instructions: program code
Memory Permissions

- Stack: Managed "automatically" (by compiler)
- Dynamic Data (Heap): Managed by programmer
- Static Data: Initialized when process starts
- Literals: Initialized when process starts
- Instructions: Initialized when process starts

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

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.