Administrivia

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

- 3 ways to set condition codes are:

- 2 ways to use condition code are:

- Does leaq set condition codes?
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!

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

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;
}
```
Compiling Loops

C/Java code:

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

Assembly code:

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

C/Java code:

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

Goto version

```c
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:

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

x86-64:

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

loopDone:
```

x86-64:

```assembly
loopTop:       testq  %rax, %rax
               je     loopDone

loopDone:
```

```assembly
loopTop:       <loop body code>
               testq  %rax, %rax
               jne    loopTop
```
For Loop → While Loop

For Version

\[
\text{for (} \text{Init;} \quad \text{Test;} \quad \text{Update}) \\
\text{Body}
\]

While Version

\[
\text{Init;} \\
\text{while (} \text{Test}) \{ \\
\text{Body} \\
\text{Update;} \\
\}
\]

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;
            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 Tables

- Compiles sometimes implement switch statements with:
  - Jump table
  - Uses the Indirect jump instruction
- Why? When?
Jump Table Structure

### Switch Form

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

### Jump Table

- `JTab`: Arrangement of targets
  - `Targ0`
  - `Targ1`
  - `Targ2`
  - `\ldots`
  - `Targ_{n-1}`

### Jump Targets

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

### Approximate Translation

```java
target = JTab[x];
goto target;
```
Jump Table Structure

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 \( x \leq 6 \):

```
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)

<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>

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

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

Indirect jump
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
Jump Table

Jump table

declaring data, not instructions

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;
}

Jump Table

.declaration .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

declaring data, not instructions

8-byte memory alignment

this data is 64-bits wide
Code Blocks \((x == 1)\)

```c
switch(x) {
    case 1:  // .L3
        w = y*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>

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

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

```c
case 3:
  w = 1;
merge:
  w += z;
```
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;
    . . .
}
```

### Register Use(s)

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

```
.L5:    # Case 2:
    movq  %rsi, %rax  # y in rax
    cqto  %rax        # 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)

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

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

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

**C:**
```c
car *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
```

**OS:**
- Windows 8
- Mac
- Linux

**Computer system:**
- Intel Core i5
- RAM
- SSD

**Memory & data:**
- Integers & floats
- x86 assembly
- 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

```c
int Q(int i) {
    int t = 3*i;
    int v[10];
    ...
    return v[t];
}

P(...) {
    ...
    y = Q(x);
    print(y);
    ...
}
```
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
- Literals
- Static Data
- Dynamic Data (Heap)
- Stack

- Program code
- Large constants (e.g. "example")
- Static variables (including global variables (C))
- Variables allocated with `new` or `malloc`
- Local variables; procedure context

Memory Addresses:
- High Addresses
- Low Addresses

Addressing:
- Low Addresses: 0
- High Addresses: $2^{N-1}$
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

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` -8
### 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`

---

**Stack Pointer:**  
- `%rsp`  
- `+8`

Stack Grows Down

High Addresses

Increasing Addresses

Low Addresses

0x00...00

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 args
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on 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

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

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

Compiler Explorer:
https://godbolt.org/g/52Sqxj
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*
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
  2) Jump to address
Procedure Call Example (step 1)

```
0000000000400540 <multstore>:
    •
    •
400544:  call  400550 <mult2>
400549:  movq  %rax,(%rbx)
    •
    •
```

```
0000000000400550 <mult2>:
    400550:  movq  %rdi,%rax
    •
    •
400557:  ret
```

```
%rip  0x400544
%rsp  0x120
0x120
0x128
0x130
  •
  •
```

```
Procedure Call Example (step 2)

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

00000000000400550 <mult2>:
  400550: movq %rdi,%rax
  •
  •
  400557: ret

%rsp 0x118
0x400549

%rip 0x400550
0x400550

0x118
0x120
0x128
0x130
Procedure Return Example (step 1)

0000000000400540 <multstore>:
  
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  
0000000000400550 <mult2>:
  
  400550: movq %rdi,%rax
  
  400557: ret

0x400557 0x118
0x118 0x120
0x120 0x130
%rip 0x400557
%rsp 0x118

0x400549

%rax (%%rbx)

Procedure Return Example (step 2)

0000000000400540 <multstore>:
  •
  •
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  •
  •

0000000000400550 <mult2>:
  •
  400550: movq %rdi,%rax
  •
  •
  400557: ret