CS107, Lecture 13
Control Flow: When in doubt just JMP!

Reading: B&O 3.1-3.4
Attendance

https://forms.gle/Ee4rksrsHaBR3N4A8
Learning Goals

• Learn about how assembly stores comparison and operation results in condition codes
• Understand how assembly implements loops and control flow
**cmov: Conditional move**

**cmovx src, dst** conditionally moves data in src to data in dst.

- Mov src to dst if condition x holds; no change otherwise
- src is memory address/register, dst is register
- May be more efficient than branch (i.e., jump)
- Often seen with C ternary operator: `result = test ? then : else;`

```c
int max(int x, int y) {
    return x > y ? x : y;
}
```

```assembly
cmp     %edi,%esi
mov     %edi, %eax
cmovge  %esi, %eax
retq
```
### cmov: Conditional move

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Move Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmov S,R</td>
<td>cmovz</td>
<td>Equal / zero (ZF = 1)</td>
</tr>
<tr>
<td>cmovne S,R</td>
<td>cmovnzs</td>
<td>Not equal / not zero (ZF = 0)</td>
</tr>
<tr>
<td>cmovs S,R</td>
<td>cmovns</td>
<td>Negative (SF = 1)</td>
</tr>
<tr>
<td>cmovns S,R</td>
<td>cmovns</td>
<td>Non-negative (SF = 0)</td>
</tr>
<tr>
<td>cmovg S,R</td>
<td>cmovgne</td>
<td>Greater (signed &gt;) (SF = 0 and SF = OF)</td>
</tr>
<tr>
<td>cmovge S,R</td>
<td>cmovgel</td>
<td>Greater or equal (signed &gt;=) (SF = OF)</td>
</tr>
<tr>
<td>cmovl S,R</td>
<td>cmovleng</td>
<td>Less (signed &lt;) (SF != OF)</td>
</tr>
<tr>
<td>cmovle S,R</td>
<td>cmovleng</td>
<td>Less or equal (signed &lt;=) (ZF = 1 or SF! = OF)</td>
</tr>
<tr>
<td>cmova S,R</td>
<td>cmovnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>cmovae S,R</td>
<td>cmovnb</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>cmovb S,R</td>
<td>cmovnae</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>cmovbe S,R</td>
<td>cmovna</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
How to remember cmp/jmp

- CMP S1, S2 is S2 – S1 (just sets condition codes). **But generally:**

```
    cmp S1, S2
    jg ...
```

- Much less important to remember exact condition codes
  - Yes, they fully explain conditional jmp...
  - ...but more important to know how to translate assembly back into C
  - If you’re interested, B&O p. 206 has details
• TEST S1, S2 is S2 & S1

```
test %edi, %edi
```

```c
jns ...
```

%edi & %edi is nonnegative

%edi is nonnegative
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
        result = ___(3)___;
        a = ___(4)___;
    }
    return result;
}

GCC common while loop construction:
Test
Jump past loop if fails
Body
Jump to test

https://godbolt.org/z/zrW6c5MGa
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
        result = ___(3)___;
        a = ___(4)___;
    }
    return result;
}

GCC common while loop construction:
Test
Jump past loop if fails
Body
Jump to test
long loop(long a, long b) {
    long result = __________;
    while (__________) {
        result = __________;
        a = __________;
    }
    return result;
}
long loop(long a, long b) {
    long result = 1;
    while (a < b) {
        result = result*(a+b);
        a = a + 1;
    }
    return result;
}
int elem_arithmetic(int nums[], int y) {
    int z = nums[________] * ________;

    z -= ________;

    return ________;
}

----------
// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax
    imull 4(%rdi), %eax
    movslq %esi, %rsi
    subl (%rdi,%rsi,4), %eax
    lea 2(%rax, %rax), %eax
    ret
Warm-up: Reverse Engineering

```c
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;

    z -= ________;

    return ________;
}
```

----------
// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax       // copy y into %eax
    imull 4(%rdi), %eax   // multiply %eax by nums[1]
    movslq %esi, %rsi     // sign-extend %esi to %rsi
    subl (%rdi,%rsi,4), %eax
    lea 2(%rax, %rax), %eax
    ret
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;
    
    z -= nums[y];
    
    return 2 * z + 2;
}

// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax          // copy y into %eax
    imull 4(%rdi), %eax      // multiply %eax by nums[1]
    movslq %esi, %rsi        // sign-extend %esi to %rsi
    subl (%rdi,%rsi,4), %eax // subtract nums[y] from %eax
    lea 2(%rax, %rax), %eax  // multiply %rax by 2, and add 2
    ret
test  practice: What’s the C code?

0x400546 <test_func>  test  %edi,%edi
0x400548 <test_func+2>  jns  0x400550 <test_func+10>
0x40054a <test_func+4>  mov  $0xfeed,%eax
0x40054f <test_func+9>  retq
0x400550 <test_func+10>  mov  $0xaabbccddd,%eax
0x400555 <test_func+15>  retq
test practice: What’s the C code?

int test_func(int x) {
    if (x < 0) {
        return 0xfeed;
    }
    return 0xaabbccdd;  // (or anything like this)
}
Practice: “Escape Room”

What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

You don’t have to reverse-engineer C code exactly!
What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

First param > 2 or == 1.
%rip

• %rip is a special register that points to the next instruction to execute.

• Let’s dive deeper into how %rip works, and how jumps modify it.
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

0x40113f <+0>:  b8 00 00 00 00  mov  $0x0,%eax
0x401144 <+5>:  83 f8 63  cmp  $0x63,%eax
0x401147 <+8>:  7f 05  jg  40114e <loop2+15>
0x401149 <+10>: 83 c0 01  add  $0x1,%eax
0x40114c <+13>: eb f6  jmp  401144 <loop2+5>
0x40114e <+15>: c3  retq
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

These are 0-based offsets in bytes (hex) for each instruction relative to the start of this function.
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

These are bytes for the machine code instructions. Instructions are variable length.
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
0x40113f <+0>: b8 00 00 00 00 00 mov $0x0,%eax
0x401144 <+5>: 83 f8 63 cmp $0x63,%eax
0x401147 <+8>: 7f 05 jg 40114e <loop2+15>
0x401149 <+10>: 83 c0 01 add $0x1,%eax
0x40114c <+13>: eb f6 jmp 401144 <loop2+5>
0x40114e <+15>: c3 retq
%rip

0x40113f <+0>:  b8 00 00 00 00 00  mov  $0x0,%eax
0x401144 <+5>:  83 f8 63  cmp  $0x63,%eax
0x401147 <+8>:  7f 05  jg   40114e <loop2+15>
0x401149 <+10>:  83 c0 01  add  $0x1,%eax
0x40114c <+13>:  eb f6  jmp  401144 <loop2+5>
0x40114e <+15>:  c3  retq

0x7f means jg.
%rip

0x40113f <+0>: b8 00 00 00 00 00 mov $0x0,%eax
0x401144 <+5>: 83 f8 63 cmp $0x63,%eax
0x401147 <+8>: 7f 05 jg 40114e <loop2+15>
0x401149 <+10>: 83 c0 01 add $0x1,%eax
0x40114c <+13>: eb f6 jmp 401144 <loop2+5>
0x40114e <+15>: c3 retq

0x05 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This jg says to then go 5 bytes further!
0x40113f <+0>:  b8 00 00 00 00 00  mov  $0x0,%eax
0x401144 <+5>:  83  f8  63          cmp  $0x63,%eax
0x401147 <+8>:  7f  05          jg   40114e <loop2+15>
0x401149 <+10>:  83  c0  01          add  $0x1,%eax
0x40114c <+13>:  eb  f6          jmp  401144 <loop2+5>
0x40114e <+15>:  c3          retq

0x05 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This **jg** says to **then** go 5 bytes further!
0x40113f <+0>: b8 00 00 00 00 00    mov $0x0,%eax
0x401144 <+5>: 83 f8 63          cmp $0x63,%eax
0x401147 <+8>: 7f 05             jg   40114e <loop2+15>
0x401149 <+10>: 83 c0 01         add $0x1,%eax
0x40114c <+13>: eb f6             jmp  401144 <loop2+5>
0x40114e <+15>: c3              retq

%rip

0xeb means jmp.
$0x0$, %eax

mov $0x0, %eax

cmp $0x63, %eax

jg 40114e <loop2+15>

add $0x1, %eax

jmp 401144 <loop2+5>

retq

$0xf6$ is the number of instruction bytes to jump relative to %rip. This is -10 (in two’s complement!).

With no jump, %rip would advance to the next line. This jmp says to then go 10 bytes back!
%rip

0x40113f <+0>:  b8 00 00 00 00 00  mov  $0x0,%eax
0x401144 <+5>:  83 f8 63  cmp  $0x63,%eax
0x401147 <+8>:  7f 05  jg  40114e <loop2+15>
0x401149 <+10>: 83 c0 01  add  $0x1,%eax
0x40114c <+13>:  eb f6  jmp  401144 <loop2+5>
0x40114e <+15>:  c3  retq

0xf6 is the number of instruction bytes to jump relative to %rip. This is -10 (in two’s complement!).

With no jump, %rip would advance to the next line. This jmp says to then go 10 bytes back!
Machine code instructions live in main memory, just like stack and heap data.

%rip is a register that stores a number (an address) of the next instruction to execute. It marks our place in the program’s instructions.

To advance to the next instruction, special hardware adds the size of the current instruction in bytes.

jmp instructions work by adjusting %rip by a specified amount.
How do we call functions in assembly?
To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the callee’s instructions, and then resume the caller’s instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
• `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
• \%rsp is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
- `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
• `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
• `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).

**Key idea:** `%rsp` must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.
• The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting %rsp accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
</table>
| `pushq S`   | `R[%rsp] ← R[%rsp] - 8;`  
             | `M[R[%rsp]] ← S`         |
- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
</table>
| `pushq S`   | `R[%rsp] ← R[%rsp] - 8;`  
              |   `M[R[%rsp]] ← S`         |
• The `push` instruction pushes the data at the specified source onto the top of the stack, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq S</td>
<td>R[%rsp] ← R[%rsp] − 8; M[R[%rsp]] ← S</td>
</tr>
</tbody>
</table>
- The `push` instruction pushes the data at the specified source onto the top of the stack, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pushq S</code></td>
<td><code>R[%rsp] ← R[%rsp] - 8; M[R[%rsp]] ← S</code></td>
</tr>
</tbody>
</table>

- This behavior is equivalent to the following, but `pushq` is a shorter instruction:
  - `subq $8, %rsp`
  - `movq S, (%rsp)`

- Sometimes, you’ll see instructions just explicitly decrement the stack pointer to make room for future data.
• The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting %rsp accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
</table>
| popq D      | D ← M[R[%rsp]]  
              R[%rsp] ← R[%rsp] + 8; |

• **Note**: this *does not* remove/clear out the data! It just increments %rsp to indicate the next push can overwrite that location.
The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>popq D</td>
<td>D ← M[R[%rsp]]&lt;br&gt;R[%rsp] ← R[%rsp] + 8;</td>
</tr>
</tbody>
</table>

This behavior is equivalent to the following, but `popq` is a shorter instruction:

```
movq (%rsp), D
addq $8, %rsp
```

Sometimes, you’ll see instructions just explicitly increment the stack pointer to pop data.
Stack Example

Initially

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

pushq %rax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x100</td>
</tr>
</tbody>
</table>

popq %rdx

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x123</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>
To call a function in assembly, we must do a few things:

• **Pass Control** – %rip must be adjusted to execute the callee’s instructions, and then resume the caller’s instructions afterwards.

• **Pass Data** – we must pass any parameters and receive any return value.

• **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
**Problem:** %rip points to the next instruction to execute. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

\[\text{E.g. main() calls foo():}\]

\[\begin{array}{c}
\text{Stack} \\
\vdots \\
\text{main()} \\
\vdots \\
\text{...} \\
\end{array}\]

\[\begin{array}{c}
%rip \\
\text{0x3021} \\
%rsp \\
\text{0xff20} \\
\end{array}\]
**Problem:** %rip points to the next instruction to execute. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

*E.g. main() calls foo:*
**Problem:** %rip points to the next instruction to execute. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

*E.g. main() calls foo:*
**Remembering Where We Left Off**

**Problem:** `%rip` points to the next instruction to execute. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. `main()` calls `foo()`:

```
Stack
...
|
|
|
|
|
|
|
|
0x3026
```

```
main()
```

```
0xff18
%rsp
```

```
0x4058
%rip
```

E.g. `main()` calls `foo()`:
**Problem:** \%rip points to the next instruction to execute. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the next value of \%rip onto the stack. Then call the function. When it is finished, put this value back into \%rip and continue executing.
The **call** instruction pushes the address of the instruction immediately following the **call** instruction onto the stack and sets %rip to point to the beginning of the specified function’s instructions.

```
call Label

call *Operand
```

The **ret** instruction pops this instruction address from the stack and stores it in %rip.

```
ret
```

The stored %rip value for a function is called its **return address**. It is the address of the instruction at which to resume the function’s execution. (not to be confused with **return value**, which is the value returned from a function).
To call a function in assembly, we must do a few things:

• **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.

• **Pass Data** – we must pass any parameters and receive any return value.

• **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
Register Restrictions

There is only one copy of registers for all programs and functions.

• **Problem:** what if `funcA` is building up a value in register `%r10`, and calls `funcB` in the middle, which also has instructions that modify `%r10`? `funcA`’s value will be overwritten!

• **Solution:** make some “rules of the road” that callers and callees must follow when using registers so they do not interfere with one another.

• These rules define two types of registers: **caller-owned** and **callee-owned**
**Caller/callee** is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. function1 at right).

- **main** calls **function1**
- **function1** calls **function2**

- main is the caller, and function1 is the callee.
- function1 is the caller, and function2 is the callee.
Register Restrictions

**Caller-Owned**
- Callee must *save* the existing value and *restore* it when done.
- Caller can store values and assume they will be preserved across function calls.

**Callee-Owned**
- Callee does not need to save the existing value.
- Caller’s values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.
Caller-Owned Registers

main can use caller-owned registers and know that function1 will not permanently modify their values.

If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.
Caller-Owned Registers

main

function1

calls

function1:
  push %rbp
  push %rbx
  ...
  pop %rbx
  pop %rbp
  retq
Callee-Owned Registers

main can use callee-owned registers but calling function1 may permanently modify their values.

If function1 wants to use any callee-owned registers, it can do so without saving the existing values.
Callee-Owned Registers

main:
...
push %r10
push %r11
callq function1
pop %r11
pop %r10
...

function1
A Day In the Life of function1

Caller-owned registers:
- function1 must save/restore existing values of any it wants to use.
- function1 can assume that calling function2 will not permanently change their values.

Callee-owned registers:
- function1 does not need to save/restore existing values of any it wants to use.
- calling function2 may permanently change their values.
Parameters and Return

• There are special registers that store parameters and the return value.
• To call a function, we must put any parameters we are passing into the correct registers. (%rdi, %rsi, %rdx, %rcx, %r8, %r9, in that order)
• Parameters beyond the first 6 are put on the stack.
• If the caller expects a return value, it looks in %rax after the callee completes.
Calling Functions In Assembly

To call a function in assembly, we must do a few things:

• **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.

• **Pass Data** – we must pass any parameters and receive any return value.

• **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                       i1, i2, i3, i4);

    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}

0x40054f <+0>:  sub    $0x18,%rsp
0x400553 <+4>:  movl   $0x1,0xc(%rsp)
0x40055b <+12>: movl   $0x2,0x8(%rsp)
0x400563 <+20>: movl   $0x3,0x4(%rsp)
0x40056b <+28>: movl   $0x4,(%rsp)
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    ...
}

0x40056f <+0>:    sub $0x18,%rsp
0x400553 <+4>:    movl $0x1,0xc(%rsp)
0x40055b <+12>:   movl $0x2,0x8(%rsp)
0x400563 <+20>:   movl $0x3,0x4(%rsp)
0x40056b <+28>:   movl $0x4,(%rsp)
```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x400553 <+4>:    movl   $0x1, 0xc(%rsp)
0x40055b <+12>:   movl   $0x2, 0x8(%rsp)
0x400563 <+20>:   movl   $0x3, 0x4(%rsp)
0x40056b <+28>:   movl   $0x4, (%rsp)
0x400572 <+35>:   pushq  $0x4
```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
```

```
$0x40056b <+28>:  movl  $0x4, (%rsp)
$0x400572 <+35>:  pushq  $0x4
$0x400574 <+37>:  pushq  $0x3
$0x400576 <+39>:  mov   $0x2, %r9d
$0x40057c <+45>:  mov   $0x1, %r8d
```

```c
... 1 2 3 4
0xffe9fc
0xffe9f8
0xffe9f4
0xffe9f0
4
4
3
0xffe9e0
... 0x400576
%rsp
0x400576
%rip
```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
        i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
    int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...}
```

```
0x400582 <+51>: lea 0x10(%rsp),%rcx
0x400587 <+56>: lea 0x14(%rsp),%rdx
0x40058c <+61>: lea 0x18(%rsp),%rsi
0x400591 <+66>: lea 0x1c(%rsp),%rdi
0x400596 <+71>: callq 0x400546 <func>
```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
        i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
    int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
Local Storage

- So far, we’ve often seen local variables stored directly in registers, rather than on the stack as we’d expect. This is for optimization reasons.

- There are **three** common reasons that local data must be in memory:
  - We’ve run out of registers
  - The ‘&’ operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}

We’re done with all our assembly lectures! Now we can fully understand what’s going on in the assembly below, including how someone would call sum_array in assembly and what the ret instruction does.
Optimizations you’ll see

**nop**
- **nop/nopl** are “no-op” instructions – they do nothing!
- Intent: Make functions align on address boundaries that are nice multiples of 8.
- “Sometimes, doing nothing is how to be most productive” – Philosopher Nick

**mov %ebx,%ebx**
- Zeros out the top 32 register bits (because a mov on an e-register zeros out rest of 64 bits).

**xor %ebx,%ebx**
- Optimizes for performance as well as code size (read more [here](#)):
  - b8 00 00 00 00 00
  - 31 c0
  - mov $0x0,%eax
  - xor %eax,%eax
GCC For Loop Output

GCC Common For Loop Output

Initialization
Test
Jump past loop if success
Body
Update
Jump to test

Possible Alternative

Initialization
Jump to test
Body
Update
Test
Jump to body if success
GCC For Loop Output

GCC Common For Loop Output

Initialization
Test
Jump past loop if success
Body
Update
Jump to test

for (int i = 0; i < n; i++)  // n = 100
GCC Common For Loop Output

Initialization
Test
Jump past loop if success
Body
Update
Jump to test

Jump to test

 GCC For Loop Output

for (int i = 0; i < n; i++) // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...

for (int i = 0; i < n; i++) // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...

...
GCC Common For Loop Output

Initialization
Test
Jump past loop if success
Body
Update
Jump to test

GCC For Loop Output

for (int i = 0; i < n; i++)  // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...

// n = 100
for (int i = 0; i < n; i++) // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...

Possible Alternative

Initialization
Jump to test
Body
Update
Test
Jump to body if success
for (int i = 0; i < n; i++)  // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...

Possible Alternative

Initialization
Jump to test
Body
Update
Test
Jump to body if success
GCC For Loop Output

GCC Common For Loop Output

- Initialization
- Test
- Jump past loop if passes
- Body
- Update
- Jump to test

Possible Alternative

- Initialization
- Jump to test
- Body
- Update
- Test
- Jump to body if success

Which instructions are better when n = 0? n = 1000?

```
for (int i = 0; i < n; i++)
```
Both versions have the same **static instruction count** (# of written instructions).

But they have different **dynamic instruction counts** (# of executed instructions when program is run).

- If \( n = 0 \), left (GCC common output) is best b/c fewer instructions
- If \( n \) is large, right (alternative) is best b/c fewer instructions

The compiler may emit a static instruction count that is several times longer than an alternative, but it may be more efficient if loop executes many times.

Does the compiler *know* that a loop will execute many times? (in general, no)

So what if our code had loops that always execute a small number of times? How do we know when gcc makes a bad decision?

(\text{take EE108, EE180, CS316 for more!})
Optimizations

• **Conditional Moves** can sometimes eliminate “branches” (jumps), which are particularly inefficient on modern computer hardware.

• Processors try to *predict* the future execution of instructions for maximum performance. This is difficult to do with jumps.
Data Alignment

• Computer systems often put restrictions on the allowable addresses for primitive data types, requiring that the address for some objects must be a multiple of some value $K$ (normally 2, 4, or 8).
• These alignment restrictions simplify the design of the hardware.
• For example, suppose that a processor always fetches 8 bytes from the memory system, and an address must be a multiple of 8. If we can guarantee that any double will be aligned to have its address as a multiple of 8, then we can read or write the values with a single memory access.
• For x86-64, Intel recommends the following alignments for best performance:

<table>
<thead>
<tr>
<th>$K$</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
</tr>
<tr>
<td>8</td>
<td>long, double, char *</td>
</tr>
</tbody>
</table>
The compiler enforces alignment by making sure that every data type is organized in such a way that every field within the struct satisfies the alignment restrictions.

For example, let's look at the following struct:

```c
struct S1 {
    int i;
    char c;
    int j;
};
```

If the compiler used a minimal allocation:

- This would make it impossible to align fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap between fields `c` and `j`:

So, don't be surprised if your structs have a `sizeof()` that is larger than you expect!
Some Extra Reading
Key GDB Tips For Assembly

• Examine 4 giant words (8 bytes) on the stack:
  (gdb) x/4g $rsp
  0x7fffffffffe870: 0x0000000000000005 0x0000000000400559
  0x7fffffffffe880: 0x0000000000000000 0x0000000000400575

• display/undisplay (prints out things every time you step/next)
  (gdb) display/4w $rsp
  1: x/4xw $rsp
  0x7fffffffffe8a8:
  0xf7a2d830 0x00007fff 0x00000000 0x00000000
Key GDB Tips For Assembly

• `stepi/finish`: step into current function call/return to caller:
  
  (gdb) finish

• Set register values during the run
  
  (gdb) p $rdi = $rdi + 1

(Might be useful to write down the original value of $rdi somewhere)

• Tui things
  
  • refresh
  
  • focus cmd – use up/down arrows on gdb command line (vs `focus asm, focus regs`)
  
  • layout regs, layout asm
gdb tips

layout split  (ctrl-x a: exit, ctrl-l: resize)
info reg

p $eax
p $eflags

b *0x400546
b *0x400550 if $eax > 98

ni
si

View C, assembly, and gdb (lab5)
Print all registers
Print register value
Print all condition codes currently set
Set breakpoint at assembly instruction
Set conditional breakpoint
Next assembly instruction
Step into assembly instruction (will step into function calls)
gdb tips

p/x $rdi  Print register value in hex
p/t $rsi  Print register value in binary
x $rdi   Examine the byte stored at this address
x/4bx $rdi Examine 4 bytes starting at this address
x/4wx $rdi Examine 4 ints starting at this address
Array Allocation and Access

- Arrays in C map in a fairly straightforward way to X86 assembly code, thanks to the addressing modes available in instructions.
- When we perform pointer arithmetic, the assembly code that is produced will have address computations built into them.
- Optimizing compilers are very good at simplifying the address computations (in lab you will see another optimizing compiler benefit in the form of division — if the compiler can avoid dividing, it will!). Because of the transformations, compiler-generated assembly for arrays often doesn't look like what you are expecting.
- Consider the following form of a data type $T$ and integer constant $N$:

$$T \ A[N]$$

- The starting location is designated as $x_A$
- The declaration allocates $N \times \text{sizeof}(T)$ bytes, and gives us an identifier that we can use as a pointer (but it isn't a pointer!), with a value of $x_A$. 
Array Allocation and Access

• Example:

<table>
<thead>
<tr>
<th>Array</th>
<th>Element Size</th>
<th>Total Size</th>
<th>Start address</th>
<th>Element i</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>12</td>
<td>$x_A$</td>
<td>$x_A + i$</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>64</td>
<td>$x_B$</td>
<td>$x_B + 8i$</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>24</td>
<td>$x_C$</td>
<td>$x_C + 4i$</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
<td>40</td>
<td>$x_D$</td>
<td>$x_D + 8i$</td>
</tr>
</tbody>
</table>

• The memory referencing operations in x86-64 are designed to simplify array access. Suppose we wanted to access $C[3]$ above. If the address of $C$ is in register `%rdx`, and $3$ is in register `%rcx`

• The following copies $C[3]$ into `%eax`,

```
movl (%rdx,%rcx,4), %eax
```
### Pointer Arithmetic

- C allows arithmetic on pointers, where the computed value is calculated according to the size of the data type referenced by the pointer.
- The array reference $A[i]$ is identical to $*(A+i)$
- Example: if the address of array $E$ is in `%rdx`, and the integer index, $i$, is in `%rcx`, the following are some expressions involving $E$:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>int *</td>
<td>$x_E$</td>
<td>movq %rdx, %rax</td>
</tr>
<tr>
<td>$E[0]$</td>
<td>int</td>
<td>$M[x_E]$</td>
<td>movl (%rdx), %eax</td>
</tr>
<tr>
<td>$E[i]$</td>
<td>int</td>
<td>$M[x_E+4i]$</td>
<td>movl (%rdx,%rcx,4) %eax</td>
</tr>
<tr>
<td>$&amp;E[2]$</td>
<td>int *</td>
<td>$x_E+8$</td>
<td>leaq 8(%rdx), %rax</td>
</tr>
<tr>
<td>$E+i-1$</td>
<td>int *</td>
<td>$x_E+4i-4$</td>
<td>leaq -4(%rdx,%rcx,4), %rax</td>
</tr>
<tr>
<td>*(E+i-3)</td>
<td>int</td>
<td>$M[x_E+4i-12]$</td>
<td>movl -12(%rdx,%rcx,4) %eax</td>
</tr>
<tr>
<td>$&amp;E[i]-E$</td>
<td>long</td>
<td>$i$</td>
<td>movq %rcx,%rax</td>
</tr>
</tbody>
</table>
Pointer Arithmetic

- Practice: $x_S$ is the address of a `short` integer array, $S$, stored in `%rdx`, and a long integer index, $i$, is stored in register `%rcx`.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in `%rax` if it is a pointer, and the result should be in register `%ax` if it has a data type `short`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S+1$</td>
<td><code>short *</code></td>
<td>$x_S + 2$</td>
<td><code>leaq 2(%rdx),%rax</code></td>
</tr>
<tr>
<td>$S[3]$</td>
<td><code>short</code></td>
<td>$M[x_S + 6]$</td>
<td><code>movw 6(%rdx),%ax</code></td>
</tr>
<tr>
<td><code>&amp;S[i]</code></td>
<td><code>short *</code></td>
<td>$x_S + 2i$</td>
<td><code>leaq (%rdx,%rcx,2),%rax</code></td>
</tr>
<tr>
<td>$S[4*i+1]$</td>
<td><code>short</code></td>
<td>$M[x_S + 8i + 2]$</td>
<td><code>movw 2(%rdx,%rcx,8),%ax</code></td>
</tr>
<tr>
<td>$S+i-5$</td>
<td><code>short *</code></td>
<td>$x_S + 2i - 10$</td>
<td><code>leaq -10(%rdx,%rcx,2),%rax</code></td>
</tr>
</tbody>
</table>
References and Advanced

• References:
  • Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  • CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  • gdbtui: https://beej.us/guide/bggdb/
  • More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  • Compiler explorer: https://gcc.godbolt.org

• Advanced Reading:
  • Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
  • history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  • x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64