CS107 Lecture 13
Assembly: Control Flow

reading:

B&O 3.6
Reflections: Share your CS107 midterm experience with your neighbor:

- What is one concept/skill you are comfortable with?
- What is one concept/skill you can work on for the final exam?
- Examples: logical operations, pointer arithmetic, generics, understanding the spec, studying an assignment, gdb, time management, ...

Warm-up: What’s the difference between `lea` and `mov`?

`movq -0x8(%rdi,%rsi,8), %rax`  
`leaq -0x8(%rdi,%rsi,8), %rax`
What’s the difference between lea and mov?

`movq -0x8(%rdi,%rsi,8), %rax`

`leaq -0x8(%rdi,%rsi,8), %rax`

lea makes use of indirect addressing to perform arithmetic with fewer instructions.
Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

Learning Assembly

- Moving data around
- Arithmetic and logical operations
- Control flow
- Function calls

Today

(get excited: learn how instructions execute in order)

2/10  2/17  Today  2/24
Register Responsibilities

Some registers take on special responsibilities during program execution.

- `%rax` stores the return value
- `%rdi` stores the first parameter to a function
- `%rsi` stores the second parameter to a function
- `%rdx` stores the third parameter to a function
- `%rip` stores the address of the next instruction to execute
- `%rsp` stores the address of the current top of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Conditional branches: If statements

• Announcements

• Loops
  • While loops
  • For loops
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Conditional branches: If statements

• Announcements

• Loops
  • While loops
  • For loops

• Optimizations
What does it mean for a program to execute?
Executing instructions

So far:
• Program values can be stored in memory or registers.
• Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
• Assembly instructions are also stored in memory.

Today:
• Who controls the instructions?
  How do we know what to do now or next?

Answer:
• The program counter (PC), %rip.
The program counter %rip

The program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.

0000000000004004ed <loop>:
4004ed: 55 push %rbp
4004ee: 48 89 e5 mov %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01 addl $0x1,-0x4(%rbp)
4004fc: eb fa jmp 4004f8 <loop+0xb>

%rip 0x4004ed
The program counter `%rip`

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction

<table>
<thead>
<tr>
<th>%rip</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4004ed</td>
<td>push 55</td>
</tr>
<tr>
<td>0x4004ee</td>
<td>mov 48 89 e5</td>
</tr>
<tr>
<td>0x4004f1</td>
<td>movl c7 45 fc 00 00 00 00</td>
</tr>
<tr>
<td>0x4004f8</td>
<td>addl 83 45 fc 01</td>
</tr>
<tr>
<td>0x4004fc</td>
<td>jmp eb fa</td>
</tr>
</tbody>
</table>
The program counter %rip

00000000004004ed <loop>:

4004ed: 55  push
4004ee: 48 89 e5  mov
4004f1: c7 45 fc 00 00 00 00  movl
4004f8: 83 45 fc 01  addl
4004fc: eb fa  jmp

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction

%rip  0x4004ee
The program counter %rip

00000000004004ed <loop>:
4004ed: 55 push
4004ee: 48 89 e5 mov
4004f1: c7 45 fc 00 00 00 00 movl
4004f8: 83 45 fc 01 addl
4004fc: eb fa jmp

Special hardware sets the program counter to the next instruction:
%rip += size of bytes of current instruction

%rip 0x4004f1
The program counter %rip

0000000000004004ed <loop>:

4004ed: 55 push
4004ee: 48 89 e5 mov
4004f1: c7 45 fc 00 00 00 00 movl
4004f8: 83 45 fc 01 addl
4004fc: eb fa jmp

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction

%rip = 0x4004f8
The program counter `%rip`

000000000004004ed <loop>:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4004ed</td>
<td>55</td>
<td>push</td>
</tr>
<tr>
<td>0x4004ee</td>
<td>48 89 e5</td>
<td>mov</td>
</tr>
<tr>
<td>0x4004f1</td>
<td>c7 45 fc 00 00 00 00</td>
<td>movl</td>
</tr>
<tr>
<td>0x4004f8</td>
<td>83 45 fc 01</td>
<td>addl</td>
</tr>
<tr>
<td>0x4004fc</td>
<td>eb fa</td>
<td>jmp</td>
</tr>
</tbody>
</table>

Special hardware sets the program counter to the next instruction:

`%rip += size of bytes of current instruction`

%rip = 0x4004fc
1. How do we repeat instructions in a loop?
The jmp instruction

```
000000000004004ed <loop>:
4004ed:  push %rbp
4004ee:  mov %rsp,%rbp
4004f1:  movl $0x0,-0x4(%rbp)
4004f8:  addl $0x1,-0x4(%rbp)
4004fc:  jmp 4004f8 <loop+0xb>
```

jmp is an **unconditional jump** that sets the program counter to the **jump target**.
The `jmp` instruction

```
00000000004004ed <loop>:
  4004ed:   push   %rbp
  4004ee:   mov    %rsp,%rbp
  4004f1:   movl   $0x0,-0x4(%rbp)
  4004f8:   addl   $0x1,-0x4(%rbp)
  4004fc:   jmp    4004f8 <loop+0xb>
```

`jmp` is an **unconditional jump** that sets the program counter to the **jump target**.
The `jmp` instruction

```
00000000004004ed <loop>:
  4004ed:  push  %rbp
  4004ee:  mov   %rsp,%rbp
  4004f1:  movl  $0x0,-0x4(%rbp)
  4004f8:  addl  $0x1,-0x4(%rbp)
  4004fc:  jmp   4004f8 <loop+0xb>
```

`jmp` is an **unconditional jump** that sets the program counter to the **jump target**.
The `jmp` instruction

00000000004004ed <loop>:

4004ed:   push   %rbp
4004ee:   mov    %rsp,%rbp
4004f1:   movl   $0x0,-0x4(%rbp)
4004f8:   addl   $0x1,-0x4(%rbp)
4004fc:   jmp    4004f8 <loop+0xb>

`jmp` is an **unconditional jump** that sets the program counter to the **jump target**.
“Interfering” with `%rip

1. How do we repeat instructions in a loop?

jmp [target]
• Gives us an infinite loop
• A 1-step unconditional jump (always jump when we execute this instruction)

What if we want a conditional jump?
“Interfering” with %rip

1. How do we repeat instructions in a loop?
2. How do we skip instructions in an if/if-else statement?
3. How do we loop while some condition is true?

Answer: condition codes + conditional jumps!
Typical 2-instruction control flow

1. Compare two values to **write** condition codes (implicit destination register)
   
   \[
   \text{cmp } S1, S2 \\
   \text{test } S1, S2
   \]

2. Conditionally jump based on **reading** condition codes (implicit source register)
   
   \[
   \text{je/jz, jne/jnz, jl, jg, ...}
   \]

**Condition codes** are special registers that auto-store the results of the most recent arith/logical operation.

in gdb:

`%eflags`
1. Compare two values to write condition codes (implicit destination register)

\[
\begin{align*}
\text{cmp } & \text{ S1, S2} \\
\text{test } & \text{ S1, S2}
\end{align*}
\]

S2 – S1 \quad \begin{cases} \text{Do not store result;} \\ \text{just set condition codes} \end{cases}

Condition codes are single-bit registers, packed into %eflags for convenience. If \( t \) is the result of cmp/test arithmetic operations:

- **ZF** = zero flag \((t = 0)\)
- **SF** = sign flag \((t < 0)\)
- **CF** = carry flag (there was a carry out of MSB*, i.e., unsigned overflow)
- **OF** = overflow flag (MSB* changed from 0 to 1, i.e., signed overflow)

*MSB: Most Significant Bit*
Exercise: Condition codes

1. After the cmp instruction, which of the below condition codes are set?
   - ZF = zero flag (\( t = 0 \))
   - SF = sign flag (\( t < 0 \))
   - CF = carry flag (unsigned overflow)
   - OF = overflow flag (signed overflow)

2. After the cmp instruction, what is %edi?
Exercise: Condition codes

000000000004004d6 <if_then>: 
4004d6:  83 ff 06   cmp   $0x6,%edi 
4004d9:  75 03   jne  4004de <if_then+0x8> 
400rdb:  83 c7 01   add   $0x1,%edi 
4004de:  8d 04 3f   lea   (%rdi,%rdi,1),%eax 
4004e1:  c3   retq 

1. After the cmp instruction, which of the below condition codes are set?
   • ZF = zero flag (t = 0)
   • SF = sign flag (t < 0)
   • CF = carry flag (unsigned overflow) (fixed since lecture)
   • OF = overflow flag (signed overflow)

2. After the cmp instruction, what is %edi? %edi is unchanged
Step 1, Control flow: \texttt{cmp, test}

1. Compare two values to \textbf{write} condition codes (implicit destination register)

\begin{align*}
\texttt{cmp} & \quad \text{S1, S2} \quad \text{S2} - \text{S1} \\
\texttt{test} & \quad \text{S1, S2} \quad \text{S2} \& \text{S1}
\end{align*}

\begin{itemize}
    \item Note the operand order!
    \item \texttt{cmp/test do not} store the result (unlike \texttt{sub/and})!
    \item They just set condition codes.
\end{itemize}

\textbf{Cool tip}: \texttt{testq %rax,%rax} checks if %rax is positive, negative, or zero.
Step 2, Control flow: Conditional jump

2. Conditionally jump based on reading condition codes (implicit source register)

    je target    jump if ZF is 1

- Target is a memory address—the address of instruction.
- We jump to target if specific condition codes are on (ZF, SF, CF, OF).
- Jumps are also known as branch instructions.
Exercise 1: Conditional jump

je target  

jump if ZF is 1

Let %edi store 0x10. Will we jump in the following cases?

1. cmp $0x10,%edi
   je  40056f
   add  $0x1,%edi

2. test $0x10,%edi
   je  40056f
   add  $0x1,%edi
Exercise 1: Conditional jump

je target  \hspace{1cm} \text{jump if ZF is 1}

Let %edi store 0x10. Will we jump in the following cases?

1. `cmp 0x10,%edi`
   `je 40056f`
   `add 0x1,%edi`

   S2 - S1 == 0, so jump

2. `test 0x10,%edi`
   `je 40056f`
   `add 0x1,%edi`

   S2 & S1 != 0, so don’t jump
### Step 2, Control flow: conditional jump

Conditionally jump based on **reading** condition codes (implicit source register)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>je Label</td>
<td>jz</td>
<td>Equal / zero (ZF = 1)</td>
</tr>
<tr>
<td>jne Label</td>
<td>jnz</td>
<td>Not equal / not zero (ZF = 0)</td>
</tr>
<tr>
<td>js Label</td>
<td></td>
<td>Negative (SF = 1)</td>
</tr>
<tr>
<td>jns Label</td>
<td></td>
<td>Nonnegative (SF = 0)</td>
</tr>
<tr>
<td>jg Label</td>
<td>jn1e</td>
<td>Greater (signed &gt;) (SF = 0 and SF = OF)</td>
</tr>
<tr>
<td>jge Label</td>
<td>jn1</td>
<td>Greater or equal (signed &gt;=) (SF = OF)</td>
</tr>
<tr>
<td>jl Label</td>
<td>jnge</td>
<td>Less (signed &lt;) (SF != OF)</td>
</tr>
<tr>
<td>jle Label</td>
<td>jng</td>
<td>Less or equal (signed &lt;=) (ZF = 1 or SF! = OF)</td>
</tr>
<tr>
<td>ja Label</td>
<td>jnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>jae Label</td>
<td>jnb</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>jb Label</td>
<td>jnae</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>jbe Label</td>
<td>jna</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
Exercise 2: Conditional jump

000000000004004d6 <if_then>:

4004d6:  83 ff 06  cmp  $0x6,%edi
4004d9:  75 03  jne  4004de <if_then+0x8>
4004db:  83 c7 01  add  $0x1,%edi
4004de:  8d 04 3f  lea  (%rdi,%rdi,1),%eax
4004e1:  c3  retq

%edi  0x5

1. What is the value of %rip after executing the jne instruction?
   A. 4004d9
   B. 4004db
   C. 4004de
   D. Other

2. What is the value of %eax when we hit the retq instruction?
   A. 4004e1
   B. 0x2
   C. 0xa
   D. 0xc
   E. Other
Condition code details and other conditional ops

- Condition codes are set for many operations other than `test` and `set`, and there are many details as to which instructions set what condition codes.
- There exist conditional operators other than jump: `setx` and `cmov`.

I want to cover more conceptually challenging material in today’s lecture, so the following slides are here for your reference.

Please read B&O 3.6 for more information if you find it useful.
Details about condition codes

• Different combinations of condition codes can indicate different things.
  • To check equality, we can cmp and look at the ZF flag (a = b means a – b = 0).
  • To check sign of %eax, we can test %eax,%eax and look at the SF or ZF flag.

• Previously-discussed arithmetic and logical instructions update these flags.
  lea does not (it was intended only for address computations).

• Logical operations (xor, etc.) set carry and overflow flags to zero.

• Shift operations set the carry flag to the last bit shifted out and set the overflow flag to zero.

• For more complicated reasons, inc and dec set the overflow and zero flags, but leave the carry flag unchanged.
**set: Read condition codes**

**set** instructions conditionally set a byte to 0 or 1.

- Reads current state of flags
- Destination is a single-byte sub-register (e.g., %al)
- Does not perturb other bytes of register
- Typically followed by movzbl to zero those bytes

```c
int small(int x) {
    return x < 16;
}
```

```asm
cmp $0xf,%edi
setle %al
movzbl %al, %eax
retq
```
### set: Read condition codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition (1 if true, 0 if false)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete D</td>
<td>setz</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>setne D</td>
<td>setnz</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>sets D</td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>sets D</td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg D</td>
<td>setnl</td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td>setge D</td>
<td>setn1</td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>setl D</td>
<td>setnge</td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td>setle D</td>
<td>setng</td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>seta D</td>
<td>setnbe</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>setae D</td>
<td>setnb</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>setb D</td>
<td>setnae</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>setbe D</td>
<td>setna</td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
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>cmovs S,R</td>
<td>cmovznz</td>
<td>Negative (SF = 1)</td>
</tr>
<tr>
<td>cmovns S,R</td>
<td>cmovnzs</td>
<td>Nonnegative (SF = 0)</td>
</tr>
<tr>
<td>cmovg S,R</td>
<td>cmovnle</td>
<td>Greater (signed &gt;) (SF = 0 and SF = OF)</td>
</tr>
<tr>
<td>cmovge S,R</td>
<td>cmovnle</td>
<td>Greater or equal (signed &gt;=) (SF = OF)</td>
</tr>
<tr>
<td>cmovl S,R</td>
<td>cmovnle</td>
<td>Less (signed &lt;) (SF != OF)</td>
</tr>
<tr>
<td>cmovle S,R</td>
<td>cmovnle</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>cmovnbe</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>cmovb S,R</td>
<td>cmovnbe</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>cmovbe S,R</td>
<td>cmovnbe</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
int signed_division(int x) {
    return x / 4;
}

signed_division:
    leal 3(%rdi), %eax          Put x + 3 into %eax
    testl %edi, %edi            Check the sign of x
    cmovns %edi, %eax           If x is positive, put x into %eax
    sarl $2, %eax               Divide %eax by 4
    ret
Condition code details and other conditional ops

(end of reference slides)

Please read B&O 3.6 for more information if you find it useful.
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Conditional branches: If statements

• Announcements

• Loops
  • While loops
  • For loops
“Interfering” with %rip

1. How do we repeat instructions in a loop?
2. How do we skip instructions in an if/if-else statement?
3. How do we loop while some condition is true?
The code we’ve been working with

This code can be translated into C function code containing a branch statement (if)!
int if_then(int param1) {
    if ( __________ ) {
        __________;
    }
    return __________;
}

Practice: Fill In The Blank

00000000004004d6 <if_then>:
4004d6: cmp $0x6,%edi
4004d9: jne 4004de
4004db: add $0x1,%edi
4004de: lea (%rdi,%rdi,1),%eax
4004e1: retq
Practice: Fill In The Blank

```c
int if_then(int param1) {
    if ( param1 == 6 ) {
        param1++;  
    }
    return param1 * 2;
}
```
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Conditional branches: If statements

• Announcements

• Loops
  • While loops
  • For loops
Hacking Pokemon Blue into Pong

- Instructions are bytes in memory!
- Find an exploit that lets you change the **program counter**:
  - “8F is an item executing machine code starting from $D163 (Number of Pokemon) upon use.”
- **Write** executable code by navigating the world and moving around items

https://www.youtube.com/watch?v=D3EvpRHL_vk
Announcements

• Midterm scores will be on website gradebook after regrade deadline closes on **Monday 2/24, 11:59pm**

• Note about makeup labs:
  • If you cannot attend your assigned lab, you may go to a different lab that same week *if space is available*.
  • We want to note that we may have to turn people away who are making up a lab if there is not enough space to accommodate them.
  • If you need to attend a makeup, you should plan ahead accordingly to ensure you can get lab credit for that week.

• Assignment 6 released today 😊
Preparing for assign 6

- atm.c Security and Robustness
- Binary Bomb

Binary Bomb is like an escape game:
- Secret codes (assembly instructions)
- Fun tools (gdb, objdump)
- Unlock each level to continue
- Catharsis with successful escape
- Time limit (due M 3/2, grace W 3/4)

- Please start early.
- Please use gdb.
- Read the textbook (B&O) if you find it useful.
• **Registers do not have addresses.** They are not located in memory.
• Draw pictures clearly depicting what is in memory and what is in registers.
• **Deadlisting** (reading assembly without executing anything) will be tedious.
• Use **gdb**.

• **lea does not** access memory; it performs arithmetic and stores the result.
• **test/cmp s1,s2** work like **and/sub s1,s2**. However, **test/cmp do not** store the result anywhere; they only update condition codes.
• **test/cmp + conditional jump** can often be translated into a single C control statement.
• **testq %rax, %rax** checks if %rax is positive, negative, or zero.
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Conditional branches: If statements

• Announcements

• Loops
  • While loops
  • For loops
Common If-Else Construction

If-Else In C
if (arg > 3) {
    ret = 10;
} else {
    ret = 0;
}
ret++;

If-Else In Assembly pseudocode
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
Practice: Fill in the Blank

If-Else In C
if ( __________ ) {
    _______; } else {
    _______; }
_____;

If-Else In Assembly pseudocode
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body

400552 <+0>:   cmp $0x3,%edi
400555 <+3>:   jle 0x40055e <if_else+12>
400557 <+5>:   mov $0xa,%eax
40055c <+10>:  jmp 0x400563 <if_else+17>
40055e <+12>:  mov $0x0,%eax
400563 <+17>:  add $0x1,%eax
Practice: Fill in the Blank

If-Else In C

if ( arg > 3 ) {
    ret = 10;
} else {
    ret = 0;
}
ret++;

If-Else In Assembly pseudocode

Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
“Interfering” with %rip

1. How do we repeat instructions in a loop?
2. How do we skip instructions in an if-else statement?
3. How do we loop while some condition is true?
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Conditional branches: If statements

• Announcements

• Loops
  • While loops
  • For loops
While loops and assembly

void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

0000000000000004004d6 <loop>:
    4004d6 <+0>: mov $0x0,%eax
    4004db <+5>: jmp 4004e0 <loop+0xa>
    4004dd <+7>: add $0x1,%eax
    4004e0 <+10>: cmp $0x63,%eax
    4004e3 <+13>: jle 4004dd <loop+0x7>
    4004e5 <+15>: repz retq

1. Which register is C code’s i?
2. What is the unconditional jmp instruction doing?
3. What are the cmp and jle instructions doing?
   (jle: jump less equal; signed <=)
while_loop
While loops and assembly: Recap

void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

1. Jumps to while loop conditional

2. If %eax – 0x63 <= 0 (i.e., %eax <= 99), then jump to execute loop body. Else, execute next instruction (i.e., exit loop)
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 (lab6)
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)
Common While Loop Construction

C pseudocode

```c
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

Assembly pseudocode

```asm
loop:
    mov $0x0,%eax
    jmp loop+0xa
    add $0x1,%eax
    cmp $0x63,%eax
    jle loop+0x7
    repz retq
```

Jump to test

Body

Test

Jump to body if success
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions
• Conditional branches: If statements
• Announcements
• Loops
  • While loops
  • For loops
Common For Loop Construction

C For loop
for (init; test; update) {
    body
}

C Equivalent While Loop
init
while(test) {
    body
    update
}

Assembly pseudocode
Init
Jump to test
Body
Update
Test
Jump to body if success

For loops and while loops are treated (essentially) the same when compiled down to assembly.
Back to Our First Assembly

```c
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

1. Which register is C code’s `sum`?
2. Which register is C code’s `i`?
3. Which assembly instruction is C code’s `sum += arr[i]`?
4. What are the `cmp` and `jl` instructions doing? (`jl`: jump less; signed `<`)
sum_array
gdb tips

\texttt{p/x$} \texttt{rdi} \quad \text{Print register value in hex}

\texttt{p/t$} \texttt{rsi} \quad \text{Print register value in binary}

\texttt{x$} \texttt{rdi} \quad \text{Examine the byte stored at this address}

\texttt{x/4bx$} \texttt{rdi} \quad \text{Examine 4 bytes starting at this address}

\texttt{x/4wx$} \texttt{rdi} \quad \text{Examine 4 ints starting at this address}
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Conditional branches: If statements

• Announcements

• Loops
  • While loops
  • For loops

• Optimizations
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.

**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
Loop optimization in GCC

C For loop
for (init; test; update) {
  body
}

GCC assembly pseudocode
Init
Jump to test
Body
Update
Test
Jump to body if success

Possible alternative?
Init
Test
Jump past loop is fails
Body
Update
Jump to Test
Loop optimization in GCC

C For loop
for (int i = 0; i < n; i++) {
    ;
}

GCC assembly pseudocode
Init
Jump to test
Body
Update
Test
Jump to body if success

Possible alternative?
Init
Test
Jump past loop is fails
Body
Update
Jump to Test

Are the number of instructions executed in the left greater, less than, or equal to those executed on the right if...
1. n = 0?
2. n = 1000?
Optimizing Instruction Counts

- 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 $n = 0$, right (possible alternative) is best b/c fewer instructions
  - If $n$ is large, left (gcc is best) 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?
- (take EE108, EE180, CS316 for more!)
Bonus assembly exercises
Practice: Parameters

00000000004005ac <sum_example1>:
  4005bd: 8b 45 e8  mov  %esi,%eax
  4005c3: 01 d0  add  %edi,%eax
  4005cc: c3  retq

Which of the following is most likely to have generated the above assembly?

// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}

// B)
int sum_example1(int x, int y) {
    return x + y;
}

// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
Which of the following is most likely to have generated the above assembly?

// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}

// B)
int sum_example1(int x, int y) {
    return x + y;
}

// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}

B
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}

1. What memory location, register, or immediate above represents the C code’s sum variable?

2. What memory location, register, or immediate in the assembly code above represents the C code’s 6 (as in arr[6])? 😳
Practice: Intermediates/registers

0000000000400578 <sum_example2>:
  400578:  8b 47 0c      mov 0xc(%rdi),%eax
  40057b:  03 07      add (%rdi),%eax
  40057d:  2b 47 18      sub 0x18(%rdi),%eax
  400580:  c3      retq

```c
int sum_example2(int arr[]) {
  int sum = 0;
  sum += arr[0];
  sum += arr[3];
  sum -= arr[6];
  return sum;
}
```

1. What memory location, register, or immediate above represents the C code’s `sum` variable?
   - `%eax`

2. What memory location, register, or immediate in the assembly code above represents the C code’s 6 (as in `arr[6]`)?
   - `0x18`
long loop(long a, long b) {
    long result = _______;  
    while (_________) {
        result = __________;
        a = __________;
    }
    return result;
}
C Code

long loop(long a, long b) {
    long result = _______;  
    while (_________)
        result = __________;
        a = __________;
    return result;
}

What does this assembly code translate to?

// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
    rep; ret
Practice: while loop

C Code

```c
long loop(long a, long b) {
    long result = 1;
    while (a < b) {
        result = result*(a+b);
        a = a + 1;
    }
    return result;
}
```

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
    .L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
    .L2
    cmpq %rsi, %rdi
    jl .L3
    rep; ret
```

Assembly pseudocode

Jump to test
Body
Test
Jump to body if success