x86 Assembly Language

CS 0449: Introduction to System Software

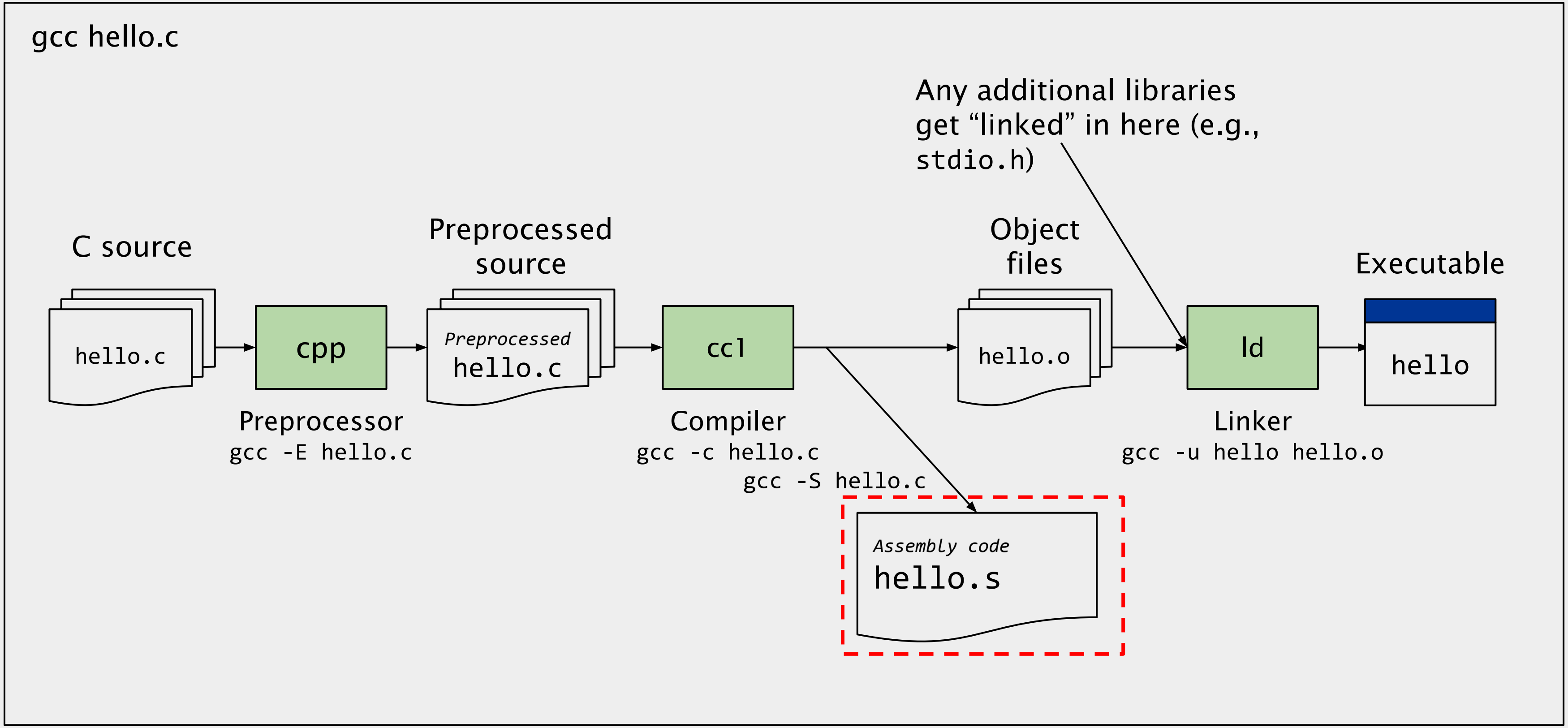
CS0449 Teaching Assistants

University of Pittsburgh | School of Computing and Information

Assembly Language

Because decoding 1s and 0s is hard

What we are building towards...



Moving down the ladder of abstractions

Very abstract

Level of abstraction

Not abstract

Application level (Word, Zoom, Firefox)

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

High-level language level (C, Java)

```
get_mpg:
        pushq     %rbp
        movq                %rsp,
%rbp
        ...
        popq                %rbp
        ret
```

Assembly language level (x86)

```
01110100000110001000110100000
10000000010100010011100001011
00000111111010000111111000100
11100001010001001110000101000
```

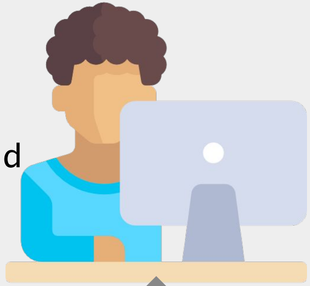
Machine language level

Operating system level (Linux, Windows, macOS)

Hardware level

What is assembly?

➔ **Assembly language** is a human-readable textual representation of machine language



High-level language (C, Java)

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

Relatively Easy for us to understand

```
get_mpg:
        pushq   %rbp
        movq            %rsp, %rbp
        ...
        popq            %rbp
        ret
```

Assembly acts as a translator between high-level code and machine code

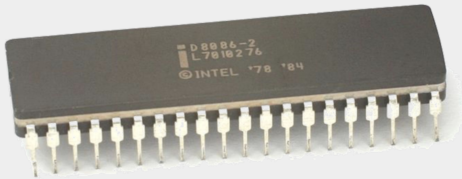
Machine language

```
0111010000011000100011010000
0010000000010100010011100000
1011000001111110100001111111
1000001111110100001111111100
```

Easy for computer to understand

Enter x86

- In CS447 Computer Organization & Assembly, you used **MIPS**
  - Which was based on a Reduced Instruction Set Computer (**RISC**) ISA
    - Small number of instructions
    - Simple instructions
- Now, we use **x86 asm**



Intel 8086
Released 1978

Intel i9-10900K
Released 2020

x86 assembly language

- Epitome of Complex Instruction Set Computer (**CISC**)
  - Lots of instructions and ways to use them
    - Hundreds of instructions
- Designed for humans to write
  - From way back when programmers used to program in assembly language
  - A time before compilers or high-level languages
- Complex (multi-step) instructions
  - Instruction to search a string for a character
  - `F2XM1` computes $2^x - 1$
    - Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range - 1.0 to +1.0. If the source value is outside this range, the result is undefined.
- Fewer instructions to write the same program
  - compared to RISC

But why use asm, if I can just code in C?

- Any C source can be compiled to assembly
  - `gcc -S <SOURCE>.c`
  - Not *really* helpful
- But what if we don't have the source code?
  - such as a `.exe` program you downloaded from the web
- You can **disassemble** any compiled program to emit the assembly
- What can you do with this?
  - Examine behavior of a program
  - Reverse engineering!

But why use asm, if I can just code in C?

Assembly is **good** for:

- ➔ Understanding the machine
  - ◆ You get to see what exactly the CPU is doing
- ➔ Better optimization of routines
  - ◆ Think you're better than a compiler?
- ➔ Programming hardware-dependent routines
  - ◆ E.g., compilers, operating systems,...
- ➔ Reverse-engineering and code obfuscation
  - ◆ malware/driver analysis...

Knowing assembly will enhance your code!

Assembly is **bad** for:

- ➔ Portability is lost
  - ◆ Code only works for a particular architecture, or processor
- ➔ Obfuscate the code
  - ◆ Not everyone can read assembly
    - • **But you can!**
- ➔ Debugging is hard
  - ◆ Most debuggers are lost when hitting assembly
    - • **But not GDB!**
- ➔ Optimizations is tedious
  - ◆ Tbh, you can't beat a modern compiler

Use it with caution and sparsity!

One code, two assembly

- Assembly language is simply a textual representation of machine language

  ⇒ Multiple representations for the same machine language

AT&T Syntax

- Developed by AT&T (duh)
- Used by GNU Assembler (`gas`)
- Opcode appended by type:
  - `b` – byte (8 bit)
  - `w` – word (16 bit)
  - `l` – long (32 bit)
  - `q` – quad (64 bit)
- First operand is **source**
- Second operand is **destination**
- Dereferences are denoted by `( )`

Intel Syntax

- Developed by Intel (duh)
- Used by Microsoft (MASM), intel, NASM
- Type sizes are spelled out:
  - `BYTE` – 1 byte
  - `WORD` – 2 bytes
  - `DWORD` – 4 bytes (double word)
  - `QWORD` – 8 bytes (quad word)
- First operand is **destination**
- Second operand is **source**
- Dereferences are denoted by `[ ]`

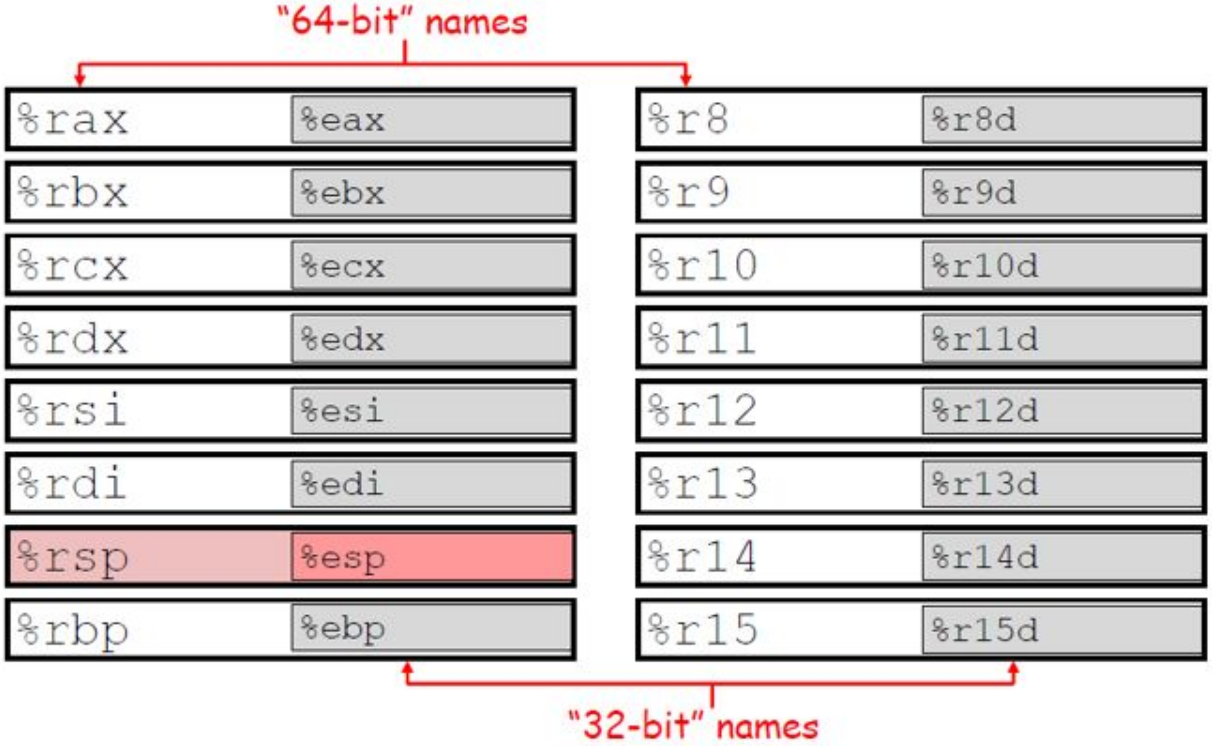
Keeping track of the registers

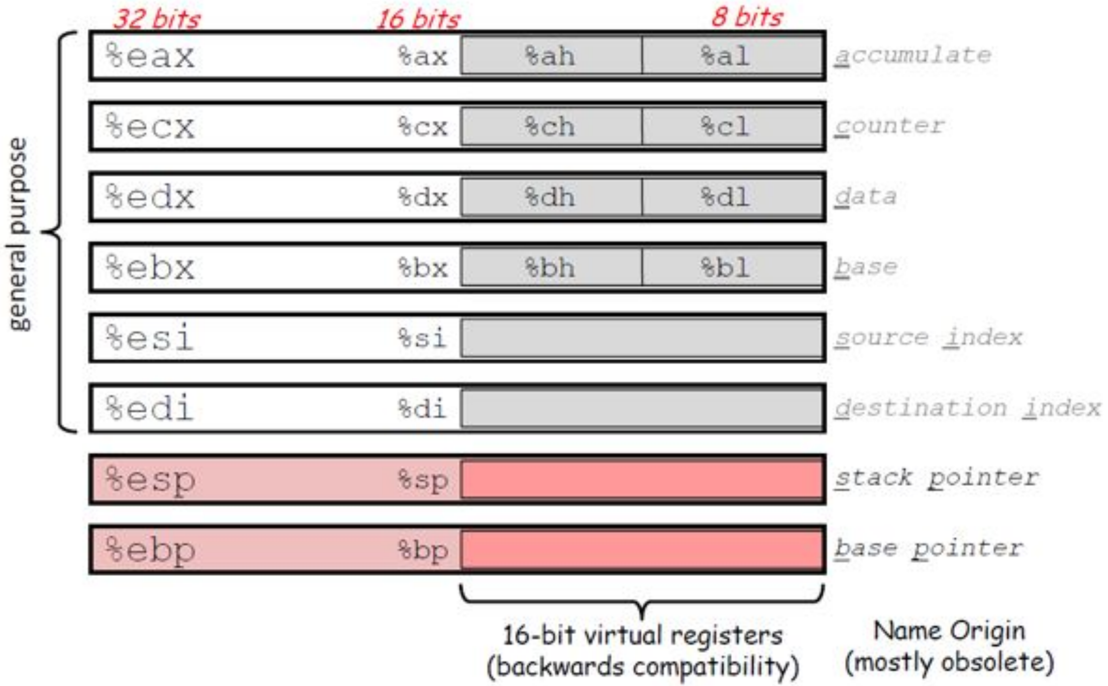
- Like in MIPS, x86 has calling conventions
  - The **C Application Binary Interface (ABI)**
  - Like MIPS, certain registers are typically used for returns values, args, etc
- The ABI is not defined by the language, but rather the OS
  - Windows and Linux (UNIX/System V) have a different C ABI
- In our x86-64 Linux C ABI,
  - `%rdi, %rsi, %rdx, %rcx, %r8, %r9` are used to pass arguments (like the `a` registers in MIPS)
    - Remaining arguments go on the stack
  - A function callee must preserve `%rbp, %rbx, %r12, %r13, %r14, %r15` (like the `s` registers in MIPS)
  - `%rax` (overflows into `%rdx` for 128-bits) stores the return value (like `v0, v1` in MIPS)
- Reference manual provides extra information

Registers

- A register is a location within the processor that is able to store data
  - Names, not addresses
  - Much faster than DRAM
  - Can hold any value: addresses, values from operations, characters etc.
  - Usually, register
    - `%rip` stores the address of the next instruction
    - `%rsp` is used as a stack pointer
    - `%rax` holds the return value from a function
  - A register in x86-64 is 64 bits wide
    - 'The lower 32-, 16- and 8-bit portions are selectable by a pseudo-register name'.

https://ctf101.org/binary-exploitation/what-are-registers/
https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/guide/x86-64.html


"64-bit" names
%rax
%eax
%rbx
%ebx
%rcx
%ecx
%rdx
%edx
%rsi
%esi
%rdi
%edi
%rsp
%esp
%rbp
%ebp
%r8
%r8d
%r9
%r9d
%r10
%r10d
%r11
%r11d
%r12
%r12d
%r13
%r13d
%r14
%r14d
%r15
%r15d
"32-bit" names



Dr Petrucci's slides - "Intro to x86-64"

mov

General form: `mov_ source, destination`

- `movb src, dst`
  Move 1-byte "byte"
- `movw src, dst`
  Move 2-byte "word"
- `movl src, dst`
  Move 4-byte "long word"
- `movq src, dst`
  Move 8-byte "quad word"

- movq src, dst # general form of instruction dst = src
- movl $0, %eax # %eax = 0
- movq %rax, $100 #Invalid!! destination cannot be an immediate value
- movsbl %al, %edx # copy 1-byte %al, sign-extend into 4-byte %edx
- movzbl %al, %edx # copy 1-byte %al, zero-extend into 4-byte %edx

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/guide/x86-64.html
Dr Petrucci's slides - "Intro to x86-64"

Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	`movq $0x4, %rax`	`var_a = 0x4;`
		Mem	`movq $-147, (%rax)`	`*p_a = -147;`
	Reg	Reg	`movq %rax, %rdx`	`var_d = var_a;`
		Mem	`movq %rax, (%rdx)`	`*p_d = var_a;`
	Mem	Reg	`movq (%rax), %rdx`	`var_d = *p_a;`

Addressing Modes - Example

- movq %rdi, 0x568892   # direct (address is constant value)
- movq %rdi, (%rax)   # indirect (address is in register %rax)
- mov (%rsi), %rdi   #%rdi = Mem[%rsi]
- movq %rdi,-24(%rbp) # indirect with displacement (address = %rbp -24)
- movq %rsi, 8(%rsp, %rdi, 4)

indirect with displacement and scaled-index (address = 8 + %rsp + %rdi*4)

- movq %rsi, 0x4(%rax, %rcx) #Mem[0x4 + %rax +%rcx*1] = %rsi
- movq %rsi, 0x8(, %rdx, 4) #Mem(0x8 + %rdx*4) = %rsi

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/guide/x86-64.html

lea

- `leaq src, dst`
  - "`lea`" stands for *load effective address*
  - `src` is address expression (any of the formats we've seen)
  - `dst` is a register
  - Sets `dst` to the *address* computed by the `src` expression (does not go to memory! - it just does math)
  - Example: `leaq (%rdx,%rcx,4), %rax`

lea

- lea or Load effective address
    - Does not dereference the source address, it simply calculates its location.
    - leaq 0x20(%rsp), %rdi   # %rdi = %rsp + 0x20 (no dereference!)
    - leaq (%rdi,%rdx,1), %rax   # %rax = %rdi + %rdx * 1

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/guide/x86-64.html

Will I have to write assembly code for this course?

- **No!** No matter how good you are at programming, you are no match for a modern compiler
  - Modern Compilers are just too good at optimization
    - There was a time when humans outperformed compilers
      - Those days are long gone now…
- However, you should be able to ***read*** assembly code
  - To figure out what your machine is doing
  - To *guess* the C code
- By the end of this lab, you should be able to freely translate assembly and C

Diving into the Code!

See code: https://github.com/shinwookim/asm-demo

Hello World! x86 edition

```c
#include <stdio.h>
int main(void)
{
    puts("Hello World!");
    return 0;
}
```

```
text (code) segment:
55 48 89 E5 BF 00 00 00 00 E8 00 00 00
00 B8 00 00 00 00 5D C3

data segment:
48 65 6C 6C 6F 2C 20 57 6F 72 6C

// Symbol table and other info omitted
```

```
.LC0:
   .string "Hello World!"
main:
   pushq   %rbp
   movq    %rsp, %rbp # rsp = stack pointer
   movl    $.LC0, %edi  # push func args
   call    puts # call a function
   movl    $0, %eax # eax = return register
   popq    %rbp # prepare to return
   ret  # return
```

Linker → Executable

Debugging Assembly

- Recall that **GDB** worked on *executables*
  - You ran `gdb mdriver` and not ~~`gdb mdriver.c`~~
- Having the source was nice
  - We used the `-g` flag when compiling
  - which allowed us to use `layout src` to view the code during execution
- ...but not necessary
- What if we don't have a source file ? (or the program was compiled without `-g` flag)
  - We can still run GDB!
  - Won't be able to see the source code ⇒ We need to inspect assembly code

```
Reading symbols from a.out...
(No debugging symbols found in a.out)
```

Displaying the assembly with `disas`

- Suppose we are in paused in a breakpoint
- We can view the assembly code around our current memory address using `disas`
  - Memory address that is held by the program counter
- But how do we set a breakpoint
  - if we don't have the code?
- Surely, we need a way to view ASM
  - Without first setting a breakpoint right?

```
Dump of assembler code for function __GI__IO_puts:
Address range 0x7ffff7e09ed0 to 0x7ffff7e0a069:
=> 0x00007ffff7e09ed0 <+0>:     endbr64
   0x00007ffff7e09ed4 <+4>:     push   %r14
   0x00007ffff7e09ed6 <+6>:     push   %r13
   0x00007ffff7e09ed8 <+8>:     push   %r12
   0x00007ffff7e09eda <+10>:    mov    %rdi,%r12
   0x00007ffff7e09edd <+13>:    push   %rbp
   0x00007ffff7e09ede <+14>:    push   %rbx
   0x00007ffff7e09edf <+15>:    sub    $0x10,%rsp
   0x00007ffff7e09ee3 <+19>:    call   0x7ffff7db1490 <*ABS*+0xa8720@plt>
   0x00007ffff7e09ee8 <+24>:    mov    0x197f49(%rip),%r13        # 0x7ffff7fa1e38
   0x00007ffff7e09eef <+31>:    mov    %rax,%rbx
   0x00007ffff7e09ef2 <+34>:    mov    0x0(%r13),%rbp
   0x00007ffff7e09ef6 <+38>:    mov    0x0(%rbp),%eax
   0x00007ffff7e09ef9 <+41>:    and    $0x8000,%eax
   0x00007ffff7e09efe <+46>:    jne    0x7ffff7e09f58 <__GI__IO_puts+136>
   0x00007ffff7e09f00 <+48>:    mov    %fs:0x10,%r14
   0x00007ffff7e09f09 <+57>:    mov    0x88(%rbp),%r8
   0x00007ffff7e09f10 <+64>:    cmp    %r14,0x8(%r8)
   0x00007ffff7e09f14 <+68>:    je     0x7ffff7e0a008 <__GI__IO_puts+312>
   0x00007ffff7e09f1a <+74>:    mov    $0x1,%edx
   0x00007ffff7e09f1f <+79>:    lock cmpxchg %edx,(%r8)
   0x00007ffff7e09f24 <+84>:    jne    0x7ffff7e0a050 <__GI__IO_puts+384>
   0x00007ffff7e09f2a <+90>:    mov    0x88(%rbp),%r8
   0x00007ffff7e09f31 <+97>:    mov    0x0(%r13),%rdi
   0x00007ffff7e09f35 <+101>:   mov    %r14,0x8(%r8)
   0x00007ffff7e09f39 <+105>:   mov    0xc0(%rdi),%eax
--Type <RET> for more, q to quit, c to continue without paging--
```

Displaying the assembly with `layout asm`

- The `layout asm` command displays the assembly of the entire program
    - You can scroll through the code and identify the memory addresses to set breakpoints
- But what if your program is *Huuuuge?*
    - That's gonna be a lot of scrolling

```
0x1119 <__do_global_dtors_aux+25>      je     0x1127 <__do_global_dtors_aux+39>
0x111b <__do_global_dtors_aux+27>      mov    0x2ee6(%rip),%rdi        # 0x4008
0x1122 <__do_global_dtors_aux+34>      call   0x1040 <__cxa_finalize@plt>
0x1127 <__do_global_dtors_aux+39>      call   0x1090 <deregister_tm_clones>
0x112c <__do_global_dtors_aux+44>      movb   $0x1,0x2edd(%rip)        # 0x4010 <completed.0>
0x1133 <__do_global_dtors_aux+51>      pop    %rbp
0x1134 <__do_global_dtors_aux+52>      ret
0x1135 <__do_global_dtors_aux+53>      nopl   (%rax)
0x1138 <__do_global_dtors_aux+56>      ret
0x1139 <__do_global_dtors_aux+57>      nopl   0x0(%rax)
0x1140 <frame_dummy>                   endbr64
0x1144 <frame_dummy+4>                 jmp    0x10c0 <register_tm_clones>
0x1149 <main>                          endbr64
0x114d <main+4>                        push   %rbp
0x114e <main+5>                        mov    %rsp,%rbp
0x1151 <main+8>                        lea    0xeac(%rip),%rax        # 0x2004

exec No process In:                                              L??   PC: ??
(gdb)
```

Let's put the asm in a file ⇒ Now we can `ctrl+f`

`objdump -d program > program.s`

- GNU provides a tool called object dump for unix-like systems
  - Let's you inspect information from object files
  - The `-d` flag disassembles the program and displays the `.code` section
  - The `>` flag redirects your standard I/O output to a file

```
USER@thoth:$ objdump -d a.out
a.out:     file format elf64-x86-64
Disassembly of section .init:
0000000000001000 <_init>:
    1000:    f3 0f 1e fa             endbr64
    1004:    48 83 ec 08             sub    $0x8,%rsp
    1008:    48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3fe8
    100f:    48 85 c0                test   %rax,%rax
    1012:    74 02                   je     1016 <_init+0x16>
    1014:    ff d0                   call   *%rax
    1016:    48 83 c4 08             add    $0x8,%rsp
    101a:    c3                      ret
…
```

GDB Assembly Edition

- Back to GDB…
- You can still set **breakpoints**
  - Not at specific lines of code…but at specific instructions (which are stored in memory)
  - `break *0x000055555555515b`
  - Why the `*`?
  - `*main+24`
    - You can set breakpoints at function offsets
    - Get this from GDB's `layout asm`
- You can still step through your code
  - Again, not stepping through lines of code, but through CPU instructions
  - Using `stepi` instead of `step`
    - `nexti` instead of `next`
    - `Continue`

GDB Assembly Edition

- Examining Memory
  - We can print values stored at memory address or at registers
  - `print/format expr`
    - Indicate registers with `$` (NOT `%`)
    - To print a value stored in a memory address use `*`
    - `format` tells us how to interpret values at that memory location
      - `d`: decimal
      - `x`:hex
      - `t`: binary
      - `f`: floating point
      - `i:` instruction
      - `c`: character
    - `p $rdi` displays the content at `%rdi` in a decimal format
  - `x MEM_ADDR` prints memory content
      - Just because you print it as decimal does not mean that the value is a decimal
      - Interpretation of values depends on the context (which you need to provide)
  - `info registers` lets you see all registers at once

Need help with GDB?

See (fmr) TA Gavin’s GDB videos on Canvas!

C Control Structures → Assembly

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 10; i++)
    {
        printf("%d", i);
    }
    return 0;
}
```

```
0x0000000000001155 <+12>:    movl   $0x0,-0x4(%rbp)
0x000000000000115c <+19>:    jmp    0x117b <main+50>
0x000000000000115e <+21>:    mov    -0x4(%rbp),%eax
0x0000000000001161 <+24>:    mov    %eax,%esi
0x0000000000001163 <+26>:    lea    0xe9a(%rip),%rax
0x000000000000116a <+33>:    mov    %rax,%rdi
0x000000000000116d <+36>:    mov    $0x0,%eax
0x0000000000001172 <+41>:    call   0x1050
<printf@plt>
0x0000000000001177 <+46>:    addl   $0x1,-0x4(%rbp)
0x000000000000117b <+50>:    cmpl   $0x9,-0x4(%rbp)
0x000000000000117f <+54>:    jle    0x115e <main+21>
```

C Control Structures → Assembly

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    while (i < 10)
    {
        printf("%d", i);
        i++;
    }
    return 0;
}
```

```
0x0000000000001155 <+12>:    movl   $0x0,-0x4(%rbp)
0x000000000000115c <+19>:    jmp    0x117b <main+50>
0x000000000000115e <+21>:    mov    -0x4(%rbp),%eax
0x0000000000001161 <+24>:    mov    %eax,%esi
0x0000000000001163 <+26>:    lea    0xe9a(%rip),%rax
0x000000000000116a <+33>:    mov    %rax,%rdi
0x000000000000116d <+36>:    mov    $0x0,%eax
0x0000000000001172 <+41>:    call   0x1050
<printf@plt>
0x0000000000001177 <+46>:    addl   $0x1,-0x4(%rbp)
0x000000000000117b <+50>:    cmpl   $0x9,-0x4(%rbp)
0x000000000000117f <+54>:    jle    0x115e <main+21>
```

C Control Structures → Assembly

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 10; i++)
    {
        printf("%d", i);
    }
    return 0;
}
```

```
0x0000000000001155 <+12>:    movl   $0x0,-0x4(%rbp)
0x000000000000115c <+19>:    jmp    0x117b <main+50>
0x000000000000115e <+21>:    mov    -0x4(%rbp),%eax
0x0000000000001161 <+24>:    mov    %eax,%esi
0x0000000000001163 <+26>:    lea    0xe9a(%rip),%rax
0x000000000000116a <+33>:    mov    %rax,%rdi
0x000000000000116d <+36>:    mov    $0x0,%eax
0x0000000000001172 <+41>:    call   0x1050
<printf@plt>
0x0000000000001177 <+46>:    addl   $0x1,-0x4(%rbp)
0x000000000000117b <+50>:    cmpl   $0x9,-0x4(%rbp)
0x000000000000117f <+54>:    jle    0x115e <main+21>
```

Wait….why is the assembly code the same?

for loops == while loops!

Your CPU treats them the same way!

* do-while loops also work the same way (Write a short program and inspect the assembly!)

C Control Structures → Assembly

```c
#include <stdio.h>
int main(void)
{
    int input;
    scanf("%d", &input);
    if (input > 10) printf("Big");
    else printf("Not Big");
    return 0;
}
```

```
11bf: 8b 45 f4                  mov     -0xc(%rbp),%eax
11c2: 83 f8 0a                  cmp     $0xa,%eax
11c5: 7e 16                     jle     11dd <main+0x54>
11c7: 48 8d 05 39 0e 00 00      lea     0xe39(%rip),%rax
11ce: 48 89 c7                  mov     %rax,%rdi
11d1: b8 00 00 00 00            mov     $0x0,%eax
11d6: e8 a5 fe ff ff            call    1080 <printf@plt>
11db: eb 14                     jmp     11f1 <main+0x68>
11dd: 48 8d 05 27 0e 00 00      lea     0xe27(%rip),%rax
11e4: 48 89 c7                  mov     %rax,%rdi
11e7: b8 00 00 00 00            mov     $0x0,%eax
11ec: e8 8f fe ff ff            call    1080 <printf@plt>
```

Conditional statements works as expected

Who knew that `if-else` executed different based on *conditions?*

Condition Codes

- `cmpq op2, op1` # computes result = op1 - op2, discards result, sets condition codes
- `testq op2, op1` # computes result = op1 & op2, discards result, sets condition codes

- Condition Codes - ***ZF*** *(zero flag),* ***SF*** *(sign flag),* ***OF*** *(overflow flag, signed), and* ***CF*** *(carry flag, unsigned)*

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/guide/x86-64.html

Our *real* first assembly code analysis

Looking through a real program!

Special thanks to Jake Kasper for providing slides & code

C Control Structures → Assembly

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int myNum = increment(5);
    printf("My num is %d\n", myNum);
    return 0;
}

int increment(int num)
{
    return ++num;
}
```

Prefix increment
Increments first, then returns

```
0000000000001149 <main>:
1149: f3 0f 1e fa          endbr64
114d: 55                   push   %rbp
114e: 48 89 e5             mov    %rsp,%rbp
1151: 48 83 ec 20          sub    $0x20,%rsp
1155: 89 7d ec             mov    %edi,-0x14(%rbp)
1158: 48 89 75 e0          mov    %rsi,-0x20(%rbp)
115c: bf 05 00 00 00       mov     $0x5,%edi
1161: e8 23 00 00 00       call    1189<increment>
1166: 89 45 fc             mov     %eax,-0x4(%rbp)
(…)
```

C Control Structures → Assembly

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int myNum = increment(5);
    printf("My num is %d\n", myNum);
    return 0;
}

int increment(int num)
{
    return ++num;
}
```

```
0000000000001189 <increment>:
1189: f3 0f 1e fa       endbr64
118d: 55                push %rbp
118e: 48 89 e5          mov  %rsp,%rbp
1191: 89 7d fc          mov  %edi,-0x4(%rbp)
1194: 83 45 fc 01       addl $0x1,-0x4(%rbp)
1198: 8b 45 fc          mov  -0x4(%rbp),%eax
119b: 5d                pop  %rbp
119c: c3                ret
```

C Control Structures → Assembly

%rbp needs maintains the current stack frame
- To preserve the previous stack frame
- it gets pushed onto the stack

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int myNum = increment(5);
    printf("My num is %d\n", myNum);
    return 0;
}

int increment(int num)
{
    return ++num;
}
```

```
0000000000001189 <increment>:
1189: f3 0f 1e fa          endbr64
118d: 55                   push %rbp
118e: 48 89 e5             mov  %rsp,%rbp
1191: 89 7d fc             mov  %edi,-0x4(%rbp)
1194: 83 45 fc 01          addl $0x1,-0x4(%rbp)
1198: 8b 45 fc             mov  -0x4(%rbp),%eax
119b: 5d                   pop  %rbp
119c: c3                   ret
```

C Control Structures → Assembly

%edi is our first argument register, so we're moving the value of our argument (num) into the current stack frame

Why -0x4?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int myNum = increment(5);
    printf("My num is %d\n", myNum);
    return 0;
}

int increment(int num)
{
    return ++num;
}
```

```
0000000000001189 <increment>:
1189: f3 0f 1e fa        endbr64
118d: 55                 push %rbp
118e: 48 89 e5           mov  %rsp,%rbp
1191: 89 7d fc           mov  %edi,-0x4(%rbp)
1194: 83 45 fc 01        addl $0x1,-0x4(%rbp)
1198: 8b 45 fc           mov  -0x4(%rbp),%eax
119b: 5d                 pop  %rbp
119c: c3                 ret
```

C Control Structures → Assembly

Increment the value of the argument we just stored in the stack

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int myNum = increment(5);
    printf("My num is %d\n", myNum);
    return 0;
}

int increment(int num)
{
    return ++num;
}
```

```
0000000000001189 <increment>:
1189: f3 0f 1e fa        endbr64
118d: 55                 push %rbp
118e: 48 89 e5           mov  %rsp,%rbp
1191: 89 7d fc           mov  %edi,-0x4(%rbp)
1194: 83 45 fc 01        addl $0x1,-0x4(%rbp)
1198: 8b 45 fc           mov  -0x4(%rbp),%eax
119b: 5d                 pop  %rbp
119c: c3                 ret
```

C Control Structures → Assembly

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int myNum = increment(5);
    printf("My num is %d\n", myNum);
    return 0;
}

int increment(int num)
{
    return ++num;
}
```

Move our data we've been editing in the stack, to our return register

```
0000000000001189 <increment>:
1189: f3 0f 1e fa       endbr64
118d: 55                push %rbp
118e: 48 89 e5          mov  %rsp,%rbp
1191: 89 7d fc          mov  %edi,-0x4(%rbp)
1194: 83 45 fc 01       addl $0x1,-0x4(%rbp)
1198: 8b 45 fc          mov  -0x4(%rbp),%eax
119b: 5d                pop  %rbp
119c: c3                ret
```

C Control Structures → Assembly

Pop the stack frame from the stack, as we're about to return from the current function scope, and this will load the previous stack frame back to `%rbp`

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int myNum = increment(5);
    printf("My num is %d\n", myNum);
    return 0;
}

int increment(int num)
{
    return ++num;
}
```

```
0000000000001189 <increment>:
1189: f3 0f 1e fa          endbr64
118d: 55                   push %rbp
118e: 48 89 e5             mov  %rsp,%rbp
1191: 89 7d fc             mov  %edi,-0x4(%rbp)
1194: 83 45 fc 01          addl $0x1,-0x4(%rbp)
1198: 8b 45 fc             mov  -0x4(%rbp),%eax
119b: 5d                   pop  %rbp
119c: c3                   ret
```

C Control Structures → Assembly

Return to caller
What about the return value?
It's already in the return register(%eax)

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int myNum = increment(5);
    printf("My num is %d\n", myNum);
    return 0;
}

int increment(int num)
{
    return ++num;
}
```

```
0000000000001189 <increment>:
1189: f3 0f 1e fa          endbr64
118d: 55                   push %rbp
118e: 48 89 e5             mov  %rsp,%rbp
1191: 89 7d fc             mov  %edi,-0x4(%rbp)
1194: 83 45 fc 01          addl $0x1,-0x4(%rbp)
1198: 8b 45 fc             mov  -0x4(%rbp),%eax
119b: 5d                   pop  %rbp
119c: c3                   ret
```

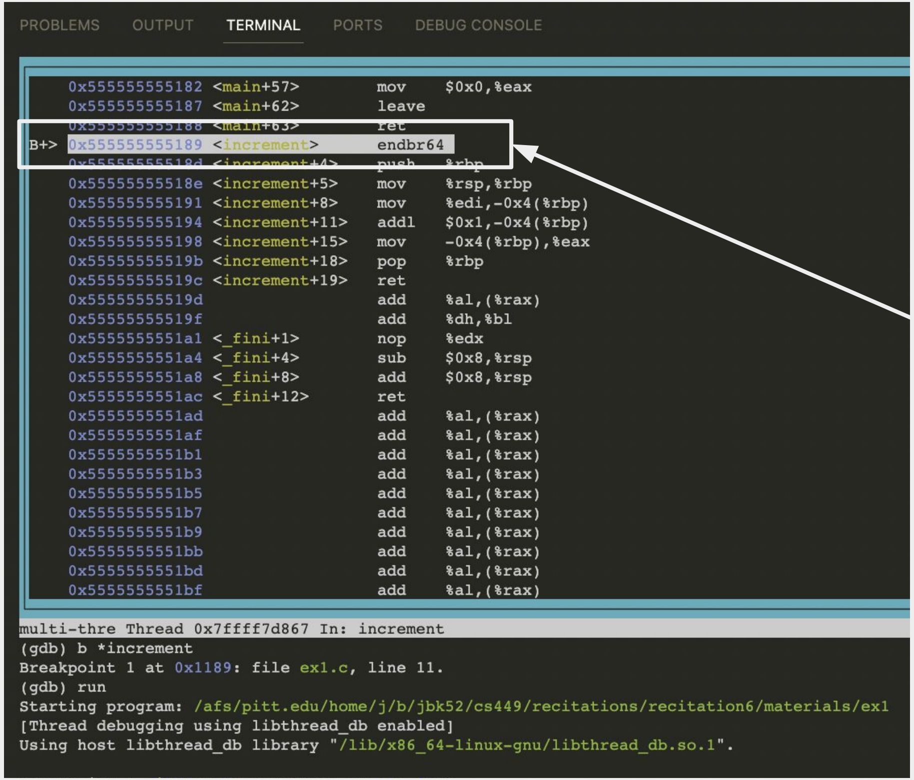
Let's inspect `increment()` with GDB

```
    0x1149 <main>            endbr64
    0x114d <main+4>          push   %rbp
    0x114e <main+5>          mov    %rsp,%rbp
    0x1151 <main+8>          sub    $0x20,%rsp
    0x1155 <main+12>         mov    %edi,-0x14(%rbp)
    0x1158 <main+15>         mov    %rsi,-0x20(%rbp)
    0x115c <main+19>         mov    $0x5,%edi
    0x1161 <main+24>         call   0x1189 <increment>
    0x1166 <main+29>         mov    %eax,-0x4(%rbp)
    0x1169 <main+32>         mov    -0x4(%rbp),%eax
    0x116c <main+35>         mov    %eax,%esi
    0x116e <main+37>         lea    0xe8f(%rip),%rax        # 0x2004
    0x1175 <main+44>         mov    %rax,%rdi
    0x1178 <main+47>         mov    $0x0,%eax
    0x117d <main+52>         call   0x1050 <printf@plt>
    0x1182 <main+57>         mov    $0x0,%eax
    0x1187 <main+62>         leave
    0x1188 <main+63>         ret
b+  0x1189 <increment>       endbr64
    0x118d <increment+4>     push   %rbp
    0x118e <increment+5>     mov    %rsp,%rbp
    0x1191 <increment+8>     mov    %edi,-0x4(%rbp)
    0x1194 <increment+11>    addl   $0x1,-0x4(%rbp)
    0x1198 <increment+15>    mov    -0x4(%rbp),%eax
    0x119b <increment+18>    pop    %rbp
    0x119c <increment+19>    ret

exec No process In:
(gdb) b *increment
Breakpoint 1 at 0x1189: file ex1.c, line 11.
(gdb)
```

Set a breakpoint at the start of the **assembly** for increment using the *

Tracing through the code w/ GDB



After running, we've hit the breakpoint at increment

Let's read the assembly line by line using **`ni`** (`next instruction`), though we can skip ahead a few lines until we get to the more important function details

Tracing through the code w/ GDB

```
   0x555555555182 <main+57>        mov    $0x0,%eax
   0x555555555187 <main+62>        leave
   0x555555555188 <main+63>        ret
B+ 0x555555555189 <increment>      endbr64
   0x55555555518d <increment+4>    push   %rbp
 > 0x55555555518e <increment+5>    mov    %rsp,%rbp
   0x555555555191 <increment+8>    mov    %edi,-0x4(%rbp)
   0x555555555194 <increment+11>   addl   $0x1,-0x4(%rbp)
   0x555555555198 <increment+15>   mov    -0x4(%rbp),%eax
   0x55555555519b <increment+18>   pop    %rbp
   0x55555555519c <increment+19>   ret
   0x55555555519d                  add    %al,(%rax)
   0x55555555519f                  add    %dh,%bl
   0x5555555551a1 <_fini+1>        nop    %edx
   0x5555555551a4 <_fini+4>        sub    $0x8,%rsp
   0x5555555551a8 <_fini+8>        add    $0x8,%rsp
   0x5555555551ac <_fini+12>       ret
   0x5555555551ad                  add    %al,(%rax)
   0x5555555551af                  add    %al,(%rax)
   0x5555555551b1                  add    %al,(%rax)
   0x5555555551b3                  add    %al,(%rax)
   0x5555555551b5                  add    %al,(%rax)
   0x5555555551b7                  add    %al,(%rax)
   0x5555555551b9                  add    %al,(%rax)
   0x5555555551bb                  add    %al,(%rax)
   0x5555555551bd                  add    %al,(%rax)
   0x5555555551bf                  add    %al,(%rax)
```

This is the line in which our stack frame pointer, `%rbp`, is being updated to contain the current stack address

Tracing through the code w/ GDB

```
   0x555555555182 <main+57>      mov    $0x0,%eax
   0x555555555187 <main+62>      leave
   0x555555555188 <main+63>      ret
B+ 0x555555555189 <increment>    endbr64
   0x55555555518d <increment+4>  push   %rbp
   0x55555555518e <increment+5>  mov    %rsp,%rbp
 > 0x555555555191 <increment+8>  mov    %edi,-0x4(%rbp)
   0x555555555194 <increment+11> addl   $0x1,-0x4(%rbp)
   0x555555555198 <increment+15> mov    -0x4(%rbp),%eax
   0x55555555519b <increment+18> pop    %rbp
   0x55555555519c <increment+19> ret
   0x55555555519d                add    %al,(%rax)
   0x55555555519f                add    %dh,%bl
   0x5555555551a1 <_fini+1>      nop    %edx
   0x5555555551a4 <_fini+4>      sub    $0x8,%rsp
   0x5555555551a8 <_fini+8>      add    $0x8,%rsp
   0x5555555551ac <_fini+12>     ret
   0x5555555551ad                add    %al,(%rax)
   0x5555555551af                add    %al,(%rax)
   0x5555555551b1                add    %al,(%rax)
   0x5555555551b3                add    %al,(%rax)
   0x5555555551b5                add    %al,(%rax)
   0x5555555551b7                add    %al,(%rax)
   0x5555555551b9                add    %al,(%rax)
   0x5555555551bb                add    %al,(%rax)
   0x5555555551bd                add    %al,(%rax)
   0x5555555551bf                add    %al,(%rax)
```

We've now executed the instruction to add the current stack pointer to %rbp

We are also about to execute the line to put the argument register's contents into the stack frame, so let's check the value of the argument register:

`p $rdi` →

```
(gdb) p $rdi
$1 = 5
```

This makes sense, as we passed 5 into our function in our C code

```
increment(5);
```

Tracing through the code w/ GDB

```
B+  0x555555555189 <increment>      endbr64
    0x55555555518d <increment+4>    push   %rbp
    0x55555555518e <increment+5>    mov    %rsp,%rbp
    0x555555555191 <increment+8>    mov    %edi,-0x4(%rbp)
  > 0x555555555194 <increment+11>   addl   $0x1,-0x4(%rbp)
    0x555555555198 <increment+15>   mov    -0x4(%rbp),%eax
    0x55555555519b <increment+18>   pop    %rbp
    0x55555555519c <increment+19>   ret
    0x55555555519d                  add    %al,(%rax)
    0x55555555519f                  add    %dh,%bl
    0x5555555551a1 <_fini+1>        nop    %edx
    0x5555555551a4 <_fini+4>        sub    $0x8,%rsp
    0x5555555551a8 <_fini+8>        add    $0x8,%rsp
    0x5555555551ac <_fini+12>       ret
    0x5555555551ad                  add    %al,(%rax)
    0x5555555551af                  add    %al,(%rax)
    0x5555555551b1                  add    %al,(%rax)
    0x5555555551b3                  add    %al,(%rax)
    0x5555555551b5                  add    %al,(%rax)
    0x5555555551b7                  add    %al,(%rax)
    0x5555555551b9                  add    %al,(%rax)
    0x5555555551bb                  add    %al,(%rax)
    0x5555555551bd                  add    %al,(%rax)
    0x5555555551bf                  add    %al,(%rax)
    0x5555555551c1                  add    %al,(%rax)
    0x5555555551c3                  add    %al,(%rax)
    0x5555555551c5                  add    %al,(%rax)
```

Now we stored the argument register value into our stack frame. To check that this update actually changed our stack frame, let's print the integer that lies below the stack pointer:

`x/-4bx $rbp` → Read the previous 4 **bytes**

```
(gdb) x/-4bx $rbp
0x7fffffffe18c: 0x05    0x00    0x00    0x00
```

`x/-1w $rbp` → Read the previous **word** (word is the size of an integer)

```
(gdb) x/-1w $rbp
0x7fffffffe18c: 5
```

We can see both of these led us to the value 5 being stored in the stack frame

Tracing through the code w/ GDB

```
    0x555555555182 <main+57>        mov    $0x0,%eax
    0x555555555187 <main+62>        leave
    0x555555555188 <main+63>        ret
B+  0x555555555189 <increment>      endbr64
    0x55555555518d <increment+4>    push   %rbp
    0x55555555518e <increment+5>    mov    %rsp,%rbp
    0x555555555191 <increment+8>    mov    %edi,-0x4(%rbp)
    0x555555555194 <increment+11>   addl   $0x1,-0x4(%rbp)
  > 0x555555555198 <increment+15>   mov    -0x4(%rbp),%eax
    0x55555555519b <increment+18>   pop    %rbp
    0x55555555519c <increment+19>   ret
    0x55555555519d                  add    %al,(%rax)
    0x55555555519f                  add    %dh,%bl
    0x5555555551a1 <_fini+1>        nop    %edx
    0x5555555551a4 <_fini+4>        sub    $0x8,%rsp
    0x5555555551a8 <_fini+8>        add    $0x8,%rsp
    0x5555555551ac <_fini+12>       ret
    0x5555555551ad                  add    %al,(%rax)
    0x5555555551af                  add    %al,(%rax)
    0x5555555551b1                  add    %al,(%rax)
    0x5555555551b3                  add    %al,(%rax)
    0x5555555551b5                  add    %al,(%rax)
    0x5555555551b7                  add    %al,(%rax)
    0x5555555551b9                  add    %al,(%rax)
    0x5555555551bb                  add    %al,(%rax)
    0x5555555551bd                  add    %al,(%rax)
    0x5555555551bf                  add    %al,(%rax)
```

At this point, we've run the line to increment the value in the stack frame, and are waiting to execute this line.

To see if this change was made, let's again print out the values:

`x/-4bx $rbp` → Read the previous 4 **bytes** as **hex**

```
(gdb) x/-4bx $rbp
0x7fffffffe18c: 0x06    0x00    0x00    0x00
```

`x/-1wx $rbp` → Read the previous **word** (word is the size of an integer) as **hex**

```
(gdb) x/-1wx $rbp
0x7fffffffe18c: 0x00000006
```

Since the value changed to 6, the increment was successful, and we can see where that change occurred.

Tracing through the code w/ GDB

```
   0x555555555182 <main+57>        mov    $0x0,%eax
   0x555555555187 <main+62>        leave
   0x555555555188 <main+63>        ret
B+ 0x555555555189 <increment>      endbr64
   0x55555555518d <increment+4>    push   %rbp
   0x55555555518e <increment+5>    mov    %rsp,%rbp
   0x555555555191 <increment+8>    mov    %edi,-0x4(%rbp)
   0x555555555194 <increment+11>   addl   $0x1,-0x4(%rbp)
   0x555555555198 <increment+15>   mov    -0x4(%rbp),%eax
 > 0x55555555519b <increment+18>   pop    %rbp
   0x55555555519c <increment+19>   ret
   0x55555555519d                  add    %al,(%rax)
   0x55555555519f                  add    %dh,%bl
   0x5555555551a1 <_fini+1>        nop    %edx
   0x5555555551a4 <_fini+4>        sub    $0x8,%rsp
   0x5555555551a8 <_fini+8>        add    $0x8,%rsp
   0x5555555551ac <_fini+12>       ret
   0x5555555551ad                  add    %al,(%rax)
   0x5555555551af                  add    %al,(%rax)
   0x5555555551b1                  add    %al,(%rax)
   0x5555555551b3                  add    %al,(%rax)
   0x5555555551b5                  add    %al,(%rax)
   0x5555555551b7                  add    %al,(%rax)
   0x5555555551b9                  add    %al,(%rax)
   0x5555555551bb                  add    %al,(%rax)
   0x5555555551bd                  add    %al,(%rax)
   0x5555555551bf                  add    %al,(%rax)
```

%eax, the return register, should contain the value 6 that we want to return to the user. Let’s see:

`p $rax` →

```
(gdb) p $rax
$3 = 6
```

`%eax` now contains the accurate return value from our function, so we can return to the previous caller after adjusting the stack.

Lab 4

Assembly Lab: ASM!

Now, it’s your turn!

- In lab 4, you will practice:
  - Reading assembly
  - Recognizing common patterns
  - Using **gdb** to *debug assembly code + inspect memory!*
- Part A: Investigating the code!
  - Reading simple functions
    - Similar to what we just did
    - https://godbolt.org/z/9c4Efqvoo
  - Deep dive into *control flow, raise operations, hidden arguments*
  - **The Test.**
    - Can you read assembly code tell me what it does?
      - **Gradescope submission**
- Part B: Inspecting memory
  - Can you debug an executable by looking at assembly code and using gdb?
    - **Gradescope submission**

References

Jonathan Misurda’s CS0449
Jake Kasper’s CS 0449 Recitation Slides (Spring 2023)
Gavin Heinrichs-Majetich’s CS 0449 Recitation Slides (Fall 2022)
Martha Dixon’s CS 0449 Recitation Slides (Fall 2020)
Randal Bryant & David R. O'Hallaron’s Computer Systems: A Programmer's Perspective
Carnegie Mellon University’s 15-213: *Introduction to Computer Systems* (Fall 2017)