



Assembly Language: Part 1



1

1

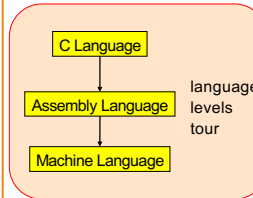
Context of this Lecture



First half of the semester: "Programming in the large"

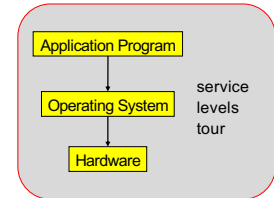
Second half: "Under the hood"

Starting Now



language
levels
tour

Later



service
levels
tour

2

2

Lectures vs. Precepts



Approach to studying assembly language:

Lectures	Precepts
Study partial pgms	Study complete pgms
Begin with simple constructs; proceed to complex ones	Begin with small pgms; proceed to large ones
Emphasis on reading code	Emphasis on writing code

3

3

Agenda



Language Levels

Architecture

Assembly Language: Performing Arithmetic

Assembly Language: Load/Store and Defining Global Data

4

4

High-Level Languages



Characteristics

- Portable
 - To varying degrees
- Complex
 - One statement can do much work – good ratio of functionality to code size
- Human readable
 - Structured – if(), for(), while(), etc.

```

count = 0;
while (n>1)
{
    count++;
    if (n&1)
        n = n*3+1;
    else
        n = n/2;
}
  
```

5

5

Machine Languages



Characteristics

- Not portable
 - Specific to hardware
- Simple
 - Each instruction does a simple task – poor ratio of functionality to code size
- Not human readable
 - Not structured
 - Requires lots of effort!
 - Requires tool support

```

0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
9222 9120 1121 A120 1121 A121 7211 0000
0000 0001 0002 0003 0004 0005 0006 0007
0008 0009 000A 000B 000C 000D 000E 000F
0000 0000 0000 FE10 FACE CAFE ACED CEDE
1234 5678 9ABC DEF0 0000 0000 F00D 0000
0000 0000 EEEE 1111 EEEE 1111 0000 0000
B1B2 F1F5 0000 0000 0000 0000 0000 0000
  
```

6

6

Assembly Languages

Characteristics

- Not portable
 - Each assembly lang instruction maps to one machine lang instruction
- Simple
 - Each instruction does a simple task
- **Human readable**
(In the same sense that Polish is human readable, if you know Polish.)

```

mov    w1, 0
loop:  cmp    w0, 1
        ble   endloop
        add   wzr, w0, #1
        beq   else
        add   w2, w0, w0
        add   w0, w0, w2
        add   w0, w0, 1
        b     endif
else:   asr    w0, w0, 1
endif:  b      loop
endloop:

```

7

7

Why Learn Assembly Language?

Q: Why learn assembly language?

A: Knowing assembly language helps you:

- Write faster code
 - In assembly language
 - In a high-level language!
- Write safer code
 - Understanding mechanism of potential security problems helps you avoid them – even in high-level languages
- Understand what's happening “under the hood”
 - Someone needs to develop future computer systems
 - Maybe that will be you!
- Become more comfortable with levels of abstraction
 - Become a better programmer!

8

8

Why Learn ARM Assembly Lang?

Why learn **ARMv8** (a.k.a. AARCH64) assembly language?

Pros

- ARM is the most widely used processor in the world (in your phone, in your Chromebook, in the internet-of-things, Armlab)
- ARM has a modern and (relatively) elegant instruction set, compared to the big and ugly x86-64 instruction set

Cons

- x86-64 dominates the desktop/laptop, for now (but there are rumors that Apple is going to shift Macs to ARM...)

9

9

Agenda

Language Levels

Architecture

Assembly Language: Performing Arithmetic

Assembly Language: Load/Store and Defining Global Data

10

10

John von Neumann (1903-1957)

In computing

- Stored program computers
- Cellular automata
- Self-replication

Other interests

- Mathematics and statistics
- Inventor of game theory
- Nuclear physics

Princeton connection

- Princeton Univ & IAS, 1930-1957
- <https://paw.princeton.edu/article/early-history-computing-princeton>

Known for “Von Neumann architecture (1950)”

- In which programs are just data in the memory
- Contrast to the now-obsolete “Harvard architecture”



11

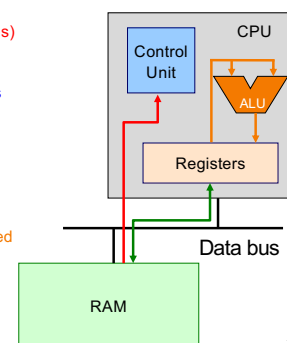
11

Von Neumann Architecture

Instructions (encoded within words) are fetched from RAM

Control unit interprets instructions

- to shuffle data between registers and RAM
- to move data from registers to ALU (arithmetic+logic unit) where operations are performed



12

12

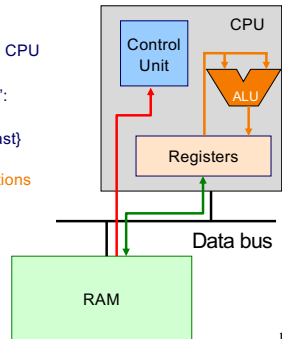
Von Neumann Architecture

Registers

Small amount of storage on the CPU

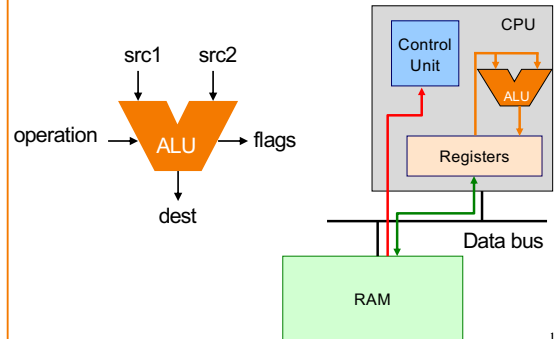
- Top of the "storage hierarchy": above RAM, disk, cloud, etc.
- i.e., *very* (small, expensive, fast)

ALU (arithmetic+logic unit) instructions operate on registers



13

ALU Arithmetic Example



14

13

14

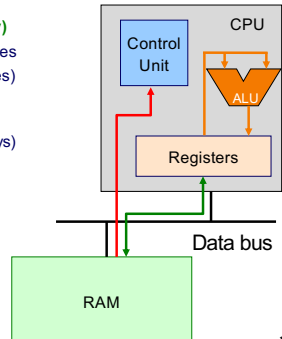
Von Neumann Architecture

RAM (Random Access Memory)

Conceptually: large array of bytes (gigabytes+ in modern machines)

- Contains data (program variables, structs, arrays)
- and the program!

Instructions are fetched from RAM



15

15

Time to reminisce about old TOYs

TOY REFERENCE CARD

INSTRUCTION FORMATS

Format	Rx	opcode	d	s	t	(0-4, A-B)
Format A1		opcode	d	s	addr	(1-9, C-F)

ARITHMETIC AND LOGICAL

- 1: add
- 2: subtract
- 3: and
- 4: xor
- 5: shift left
- 6: shift right

TRANSFER between registers

- 7: load address
- 8: load
- 9: store
- A: load indirect
- B: store indirect

CONTROL

- 0: halt
- C: branch zero
- D: branch positive
- E: jump register
- F: jump and link

Register 0 always reads 0.
Loads from M[FF] come from stdin.
Stores to M[FF] go to stdout.
16-bit registers (two's complement)
16-bit memory locations
8-bit program counter

Word size. The TOY machine has two types of storage: main memory and registers. Each entry stores one word of information. On the TOY machine, a word is a sequence of 16 bits. Typically, we interpret these 16 bits as a hexadecimal integer in the range 0000 through ffff. Using two's complement notation, we can also interpret it as a decimal integer in the range -32,768 to +32,767. See Section 5.1 for a refresher on number representations and two's complement integers.

Main memory. The TOY machine has 256 words of main memory. Each memory location is labeled with a unique memory address. By convention, we use the 256 hexadecimal integers in the range 00 through ff. Think of a memory location as a mailbox, and a memory address as a postal address. Main memory is used to store instructions and data.

Registers. The TOY machine has 16 registers, indexed from 0 through f. Registers are much like main memory: each register stores one 16-bit word. However, registers provide a faster form of storage than main memory. Registers are used as scratch space during computation and play the role of variables in the TOY language. Register 0 is a special register whose output value is always 0.

Program counter. The program counter or pc is an extra register that keeps track of the next instruction to be executed. It stores 8 bits, corresponding to a hexadecimal integer in the range 00 through ff. This integer stores the memory address of the next instruction to execute.

<https://introcs.cs.princeton.edu/java/62toy/>

16

16

Registers and RAM

Typical pattern:

- **Load** data from RAM to registers
- **Manipulate** data in registers
- **Store** data from registers to RAM

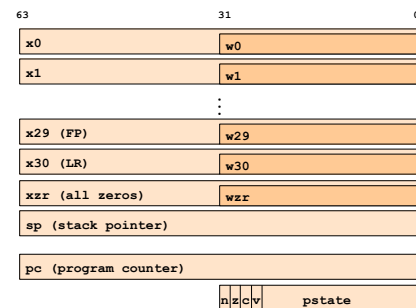
On AARCH64, this pattern is enforced

- "Manipulation" instructions can *only* access registers
- This is known as a **Load/store architecture**
- Characteristic of "RISC" (Reduced Instruction Set Computer) vs. "CISC" (Complex Instruction Set Computer) architectures, e.g. x86

17

17

Registers (ARM-64 architecture)



18

18

General-Purpose Registers

X0 .. X30

- 64-bit registers
- Scratch space for instructions, parameter passing to/from functions, return address for function calls, etc.
- Some have special roles defined *in hardware* (e.g. X30) or defined *by software convention* (e.g. X29)
- Also available as 32-bit versions: W0 .. W30

XZR

- On read: all zeros
- On write: data thrown away
- Also available as 32-bit version: WZR

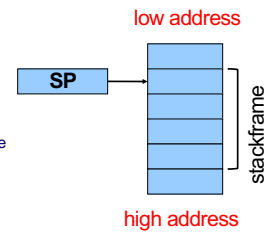
19

19

SP Register

Special-purpose register...

- **SP (Stack Pointer):**
Contains address of top (low memory address) of current function's stackframe



Allows use of the **STACK** section of memory

(See **Assembly Language: Function Calls** lecture later)

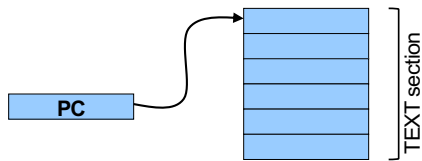
20

20

PC Register

Special-purpose register...

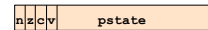
- Contains **PC (Program Counter)**
- Stores the location of the next instruction
 - Address (in TEXT section) of machine-language instructions to be executed next
- Value changed:
 - Automatically to implement sequential control flow
 - By branch instructions to implement selection, repetition



21

21

PSTATE Register



Special-purpose register...

- Contains **condition flags**:
n (Negative), **z** (Zero), **c** (Carry), **v** (oVerflow)
- Affected by compare (**cmp**) instruction
 - And many others, if requested
- Used by conditional branch instructions
 - **beq, bne, blo, bhi, ble, bge, ...**
- (See **Assembly Language: Part 2** lecture)

22

22

Agenda

Language Levels

Architecture

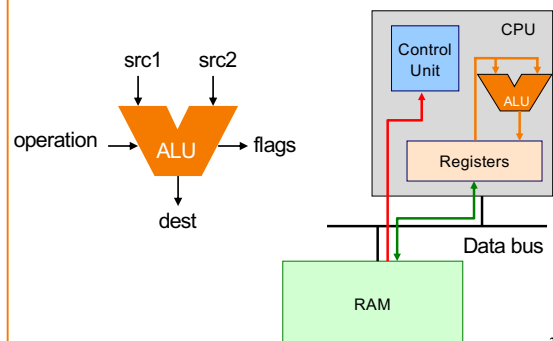
Assembly Language: Performing Arithmetic

Assembly Language: Load/Store and Defining Global Data

23

23

ALU Arithmetic Example



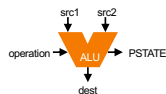
24

24

Instruction Format

Many instructions have this format:

```
name{,s} dest, src1, src2
name{,s} dest, src1, imm
```



- **name:** name of the instruction (add, sub, mul, and, etc.)
- **s:** if present, specifies that condition flags should be set
- dest and src1,src2 are **x** registers: 64-bit operation
- dest and src1,src2 are **w** registers: 32-bit operation
- src2 may be a constant ("immediate" value) instead of a register

25

25

64-bit Arithmetic

C code:

```
static long length;
static long width;
static long perim;
...
perim =
    (length + width) * 2;
```

Assume that...

- there's a good reason for having variables with file scope, process duration
- length stored in x1
- width stored in x2
- perim stored in x3

We'll see later how to make this happen

Assembly code:

```
add x3, x1, x2
lsl x3, x3, 1
```

Recall use of left shift by 1 bit to multiply by 2

26

26

More Arithmetic

```
static long x;
static long y;
static long z;
...
z = x - y;
z = x * y;
z = x / y;
z = x & y;
z = x | y;
z = x ^ y;
z = x >> y;
```

Assume that...

- x stored in x1
- y stored in x2
- z stored in x3

We'll see later how to make this happen

```
sub x3, x1, x2
mul x3, x1, x2
sdiv x3, x1, x2
and x3, x1, x2
orr x3, x1, x2
eor x3, x1, x2
asr x3, x1, x2
```

Note arithmetic shift! Logical right shift with lsr instruction

27

27

More Arithmetic: Shortcuts

```
static long x;
static long y;
static long z;
...
z = x;
z = -x;
```

Assume that...

- x stored in x1
- y stored in x2
- z stored in x3

We'll see later how to make this happen

```
mov x3, x1
neg x3, x1
```

```
orr x3, x3r, x1
sub x3, x3r, x1
```

These are actually assembler shortcuts for instructions with XZR!

28

28

Signed vs Unsigned?

```
static long x;
static unsigned long y;
...
x++;
y--;
```

Assume that...

- x stored in x1
- y stored in x2

```
add x1, x1, 1
sub x2, x2, 1
```

Mostly the same algorithms, same instructions!

- Can set different condition flags in PSTATE
- Exception is division: **sdiv** vs **udiv** instructions

29

29

32-bit Arithmetic

```
static int length;
static int width;
static int perim;
...
perim =
    (length + width) * 2;
```

Assume that...

- length stored in w1
- width stored in w2
- perim stored in w3

We'll see later how to make this happen

Assembly code using "w" registers:

```
add w3, w1, w2
lsl w3, w3, 1
```

30

30

8- and 16-bit Arithmetic?

```
static char x;
static short y;
...
x++;
y--;
```

No specialized instructions

- Use “w” registers
- Specialized “load” and “store” instructions for transfer of shorter data types from / to memory – we’ll see these later
- Corresponds to C language semantics: all arithmetic is implicitly done on (at least) ints

31

31

Agenda

Language Levels

Architecture

Assembly Language: Performing Arithmetic

Assembly Language: Load/Store and Defining Global Data

32

32

Loads and Stores

Most basic way to load (from RAM) and store (to RAM):

```
ldr dest, [src]
str src, [dest]
```

- dest and src are registers!
- Registers in [brackets] contain memory addresses
 - Every memory access is through a “pointer”!

33

33

Signed vs Unsigned, 8- and 16-bit

```
ldrb dest, [src]
ldrh dest, [src]
strb src, [dest]
strh src, [dest]
```

```
ldrsb dest, [src]
ldrsh dest, [src]
ldrsw dest, [src]
```

Special instructions for reading/writing bytes (8 bit), shorts (“half-words”: 16 bit)

- See appendix of these slides for information on ordering: little-endian vs. big-endian

Special instructions for signed reads

- “Sign-extend” byte, half-word, or word to 32 or 64 bits

34

34

Loads and Stores

Most basic way to load (from RAM) and store (to RAM):

```
ldr dest, [src]
str src, [dest]
```

- dest and src are registers!
- Registers in [brackets] contain memory addresses
 - Every memory access is through a “pointer”!
- How to get correct memory address into register?
 - Depends on whether data is on stack (local variables), heap (dynamically-allocated memory), or global / static
 - For today, we’ll look only at the global / static case

35

35

Our First Full Program*

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

* Sorry, I know by convention it should be “Hello, World!”.

36

36

Memory sections

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Sections

.data: read-write
 .rodata: read-only
 .bss: read-write, initialized to zero
 .text: read-only, program code
 Stack and heap work differently!

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

37

37

Variable definitions

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Declaring data

"Labels" for locations in memory
 .word: 32-bit int and initial value

See appendix for variables in other sections, with other types.

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

38

38

main()

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Global symbol

Declare "main" to be a globally-visible label

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

39

39

Make a "pointer"

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Generating addresses

adr puts address of a label in a register

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

40

40

Loads and Stores

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Load and store

Use "pointer" in x0 to load from and store to memory

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

41

41

Return

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Return a value

ret returns to the caller, with register 0 holding the return value

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

42

* or, in A5, not.

42

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
    (length + width) * 2;
    return 0;
}
```

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

Registers		Memory	
x0	—	length	1
w1	—	width	2
w2	—	perim	0

43

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
    (length + width) * 2;
    return 0;
}
```

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

Registers		Memory	
x0	—	length	1
w1	1	width	2
w2	—	perim	0

44

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
    (length + width) * 2;
    return 0;
}
```

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

Registers		Memory	
x0	—	length	1
w1	1	width	2
w2	—	perim	0

45

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
    (length + width) * 2;
    return 0;
}
```

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

Registers		Memory	
x0	—	length	1
w1	1	width	2
w2	2	perim	0

46

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
    (length + width) * 2;
    return 0;
}
```

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

Registers		Memory	
x0	—	length	1
w1	3	width	2
w2	2	perim	0

47

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
    (length + width) * 2;
    return 0;
}
```

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

Registers		Memory	
x0	—	length	1
w1	6	width	2
w2	2	perim	0

48

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Registers		Memory	
x0		length	1
w1	6	width	2
w2	2	perim	0

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

49

49

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Registers		Memory	
x0		length	1
w1	6	width	2
w2	2	perim	6

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

50

50

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Return value
Passed back in register w0

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

51

51

Trace

```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Return to caller
ret instruction

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```

52

52

Summary

Language levels

The basics of computer architecture

- Enough to understand AARCH64 assembly language

The basics of AARCH64 assembly language

- Instructions to perform arithmetic
- Instructions to define global data and perform data transfer

To learn more

- Study more assembly language examples
 - Chapters 2-5 of Pyeatt and Ughetta book
- Study compiler-generated assembly language code
 - `gcc217 -S somefile.c`

53

53

Appendix 1

Defining data: other sections and sizes

54

54

Defining Data: DATA Section 1

```
static char c = 'a';
static short s = 12;
static int i = 345;
static long l = 6789;
```

```
.section ".data"
c: .byte 'a'
s: .short 12
i: .word 345
l: .quad 6789
```

Notes:

`.section` instruction (to announce DATA section)
 label definition (marks a spot in RAM)
`.byte` instruction (1 byte)
`.short` instruction (2 bytes)
`.word` instruction (4 bytes)
`.quad` instruction (8 bytes)

55

55

Defining Data: DATA Section 2

```
char c = 'a';
short s = 12;
int i = 345;
long l = 6789;
```

```
.section ".data"
.global c
c: .byte 'a'
.global s
s: .short 12
.global i
i: .word 345
.global l
l: .quad 6789
```

Notes:

Can place label on same line as next instruction
`.global` instruction

56

56

Defining Data: BSS Section

```
static char c;
static short s;
static int i;
static long l;
```

```
.section ".bss"
c: .skip 1
s: .skip 2
i: .skip 4
l: .skip 8
```

Notes:

`.section` instruction (to announce BSS section)
`.skip` instruction

57

57

Defining Data: RODATA Section

```
...
... "hello\n"...;
...
```

```
.section ".rodata"
helloLabel:
.string "hello\n"
```

Notes:

`.section` instruction (to announce RODATA section)
`.string` instruction

58

58

Appendix 2

Big-endian vs little-endian byte order

59

59

Byte Order

AARCH64 is a **little endian** architecture

- Least significant byte of multi-byte entity is stored at lowest memory address
- "Little end goes first"

The int 5 at address 1000:

1000	00000101
1001	00000000
1002	00000000
1003	00000000

Some other systems use **big endian**

- Most significant byte of multi-byte entity is stored at lowest memory address
- "Big end goes first"

The int 5 at address 1000:

1000	00000000
1001	00000000
1002	00000000
1003	00000101

60

60

Byte Order Example 1

```
#include <stdio.h>
int main(void)
{
    unsigned int i = 0x003377ff;
    unsigned char *p;
    int j;
    p = (unsigned char *)&i;
    for (j = 0; j < 4; j++)
        printf("Byte %d: %2x\n", j, p[j]);
}
```

Output on a little-endian machine	Byte 0: ff	Output on a big-endian machine	Byte 0: 00
	Byte 1: 77		Byte 1: 33
	Byte 2: 33		Byte 2: 77
	Byte 3: 00		Byte 3: ff

61

61

Byte Order Example 2

Note:
Flawed code; uses "b"
instructions to load from
a four-byte memory area

```
.section ".data"
foo: .word 1
...
.section ".text"
...
adr    x0, foo
ldrb   w1, [x0]
```

AARCH64 is **little**
endian, so what will be
the value in x1?

What would be the value
in x1 if AARCH64 were
big endian?

62

62

Byte Order Example 3

Note:
Flawed code; uses word
instructions to manipulate
a one-byte memory area

```
.section ".data"
foo: .byte 1
...
.section ".text"
...
adr    x0, foo
ldr     w1, [x0]
```

What would happen?

63

63