# Storage Device Speed vs. Size



Facts:

- **CPU** needs sub-nanosecond access to data to run instructions at full speed
- **Fast** storage (sub-nanosecond) is small (100-1000 bytes)
- **Big** storage (gigabytes) is slow (15 nanoseconds)
- **Huge** storage (terabytes) is *glacially* slow (milliseconds)

We need another illusion:

- Many, many gigabytes of memory
- And fast (sub-nanosecond) average access time

Solution: **caching**

- Most of the time, data accesses are satisfied in fast, small levels of memory hierarchy
- But why, if accesses are random or uniform?
- Answer: they are not. Most programs exhibit good **locality of access** to data
  - Such programs benefit from **caching**, which enables good **average** performance

# Locality



Two kinds of **locality**

- **Temporal** locality
  - If a program references item X now, then it probably will reference X again soon
  - So once X is accessed, keep X in the fast, small memory for a little while
- **Spatial** locality
  - If a program references item X now, then it probably will reference item at address X±1 soon
  - So fetch more than just X into the fast, small memory when X is brought in

Most programs exhibit very good temporal and spatial locality, on code and on data

# Locality Example



Locality example

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
```

Typical code
(good overall locality)

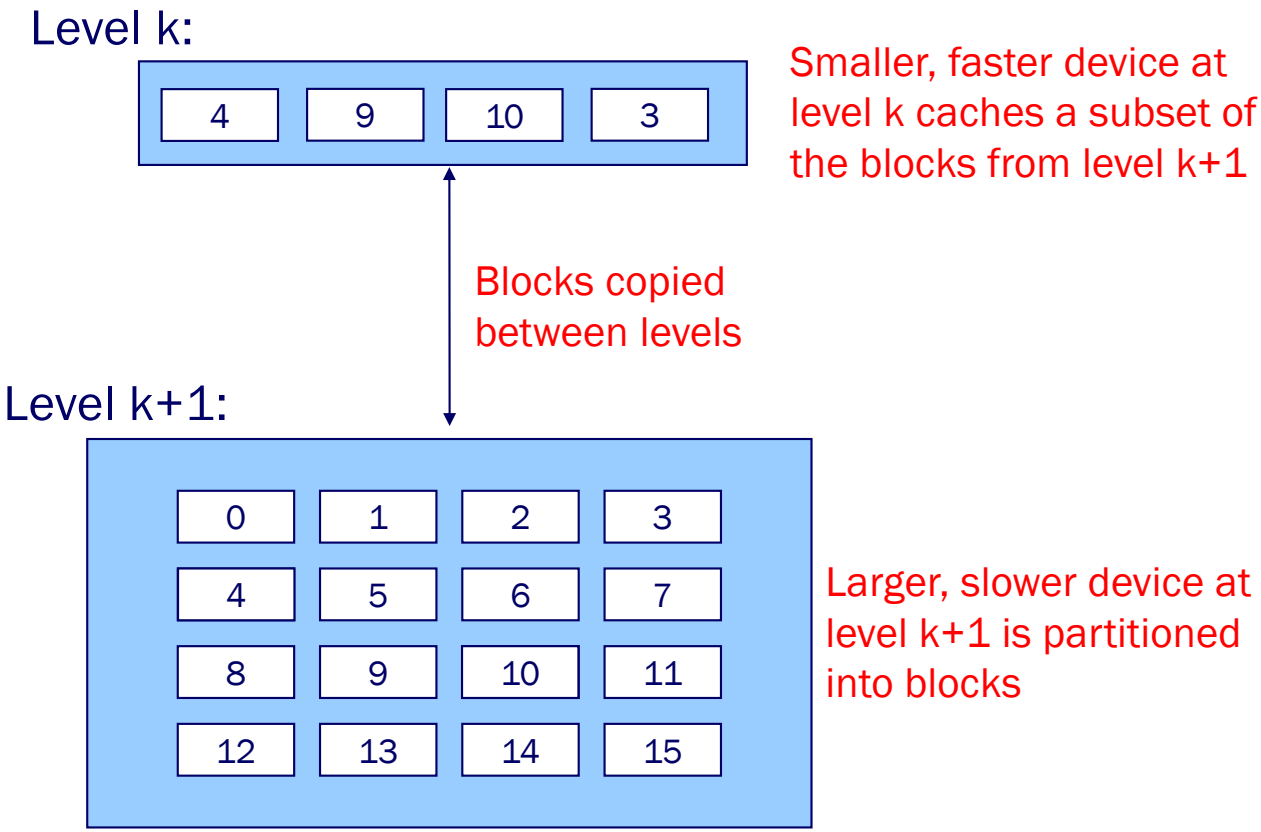
**Temporal locality**

- *Data:* When CPU accesses `sum`, it accesses `sum` again shortly thereafter
- *Instructions:* When CPU executes `sum += a[i]`, it does that again shortly thereafter

**Spatial locality**

- *Data:* When CPU accesses `a[i]`, it accesses `a[i+1]` shortly thereafter
- *Instructions:* When CPU executes `sum += a[i]`, it executes `i++` (the next machine language instructions) shortly thereafter

# Caching in a Storage Hierarchy



# Cache Hits and Misses



## Cache hit

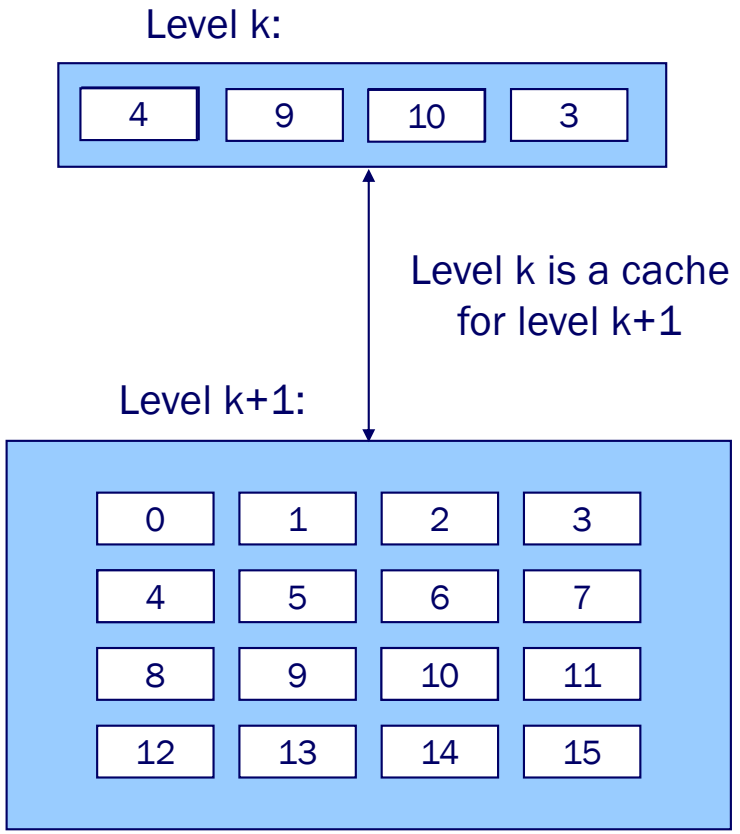
- E.g., request for block 10
- Access block 10 at level k
- Fast

## Cache miss

- E.g., request for block 8
- **Evict** some block from level k
- Load block 8 from level k+1 to level k
- Access block 8 at level k
- Slow
- But hopefully next time block 8 is accessed it will still be in cache

## Caching goal:

- Maximize cache hits
- Minimize cache misses



# Agenda



The process abstraction

The memory abstraction and memory management

- Virtual Memory
- The Storage Hierarchy, Locality and Caching
- Cache Management

# Cache Management



| Device | Managed by: |
|---|---|
| Registers<br>(cache of L1/L2/L3 cache and main memory) | Compiler, using complex code-analysis techniques<br>Assembly programmer |
| L1/L2/L3 cache<br>(cache of main memory) | Hardware, using simple algorithms |
| Main memory<br>(cache of local sec storage) | Hardware and OS, using virtual memory with complex algorithms (since accessing disk is expensive) |
| Local secondary storage (cache of remote sec storage) | Varies. System or end user, by deciding which files to download |

# Tradeoffs: Cache Block Size



**Large block size:**

+ do data transfer less often
+ take advantage of spatial locality
- longer time to complete data transfer
- can hurt temporal locality, since more data have to be evicted

**Small block size:** the opposite

**Typical:** further away memory ⇒ slower data transfer ⇒ larger block sizes

| Device | Block Size |
|---|---|
| Register | 8 bytes |
| L1/L2/L3 cache line | 128 bytes |
| Main memory page | 4KB or 64KB |
| Disk block | 512 bytes to 4KB |
| Disk transfer block | 4KB (4096 bytes) to 64MB (67108864 bytes) |

# Tradeoffs: Cache Eviction Policies



**Best** eviction policy: **"oracle"**

- Always evict a block that is *never* accessed again, or…
- Always evict the block accessed the *furthest in the future*
- Impossible to know in the general case

**Worst** eviction policy

- Always evict the block that will be accessed next
- Causes **thrashing**
- Also doesn't happen in general case

# Tradeoffs: Cache Eviction Policies



**Reasonable** eviction policy: **LRU policy**

- Evict the "Least Recently Used" (LRU) block
  - With the assumption that it will not be used again (soon)
  - Future behavior will look kind of like recent behavior
- Good for straight-line code
- (Can be) bad for (large) loops
- Expensive to implement
  - Often simpler approximations are used
  - See Wikipedia "Page replacement algorithm" topic

# Ordering Operations for Spatial Locality



Matrix multiplication example

- Matrix = two-dimensional array
- Multiply n-by-n matrices A and B
- Store product in matrix C

Performance depends upon

- Effective use of caching (as implemented by **system**)
- Good locality (as implemented by **you**)

# Spatial Locality Example: Matrix Multiplication



Two-dimensional arrays are stored in either **row-major** or **column-major** order

| a | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 18 | 19 | 20 |
| 1 | 21 | 22 | 23 |
| 2 | 24 | 25 | 26 |

| row-major | | col-major | |
|---|---|---|---|
| a[0][0] | 18 | a[0][0] | 18 |
| a[0][1] | 19 | a[1][0] | 21 |
| a[0][2] | 20 | a[2][0] | 24 |
| a[1][0] | 21 | a[0][1] | 19 |
| a[1][1] | 22 | a[1][1] | 22 |
| a[1][2] | 23 | a[2][1] | 25 |
| a[2][0] | 24 | a[0][2] | 20 |
| a[2][1] | 25 | a[1][2] | 23 |
| a[2][2] | 26 | a[2][2] | 26 |

C uses **row-major** order

- Access in row order ⇒ good spatial locality
- Access in column order ⇒ poor spatial locality

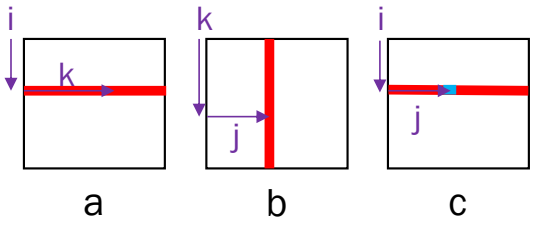
# Spatial Locality Example: Matrix Multiplication



```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      for (k=0; k<n; k++)
         c[i][j] += a[i][k] * b[k][j];
```

Reasonable cache effects

- Good spatial locality for A
- Bad spatial locality for B
- Good spatial (and temporal) locality for C



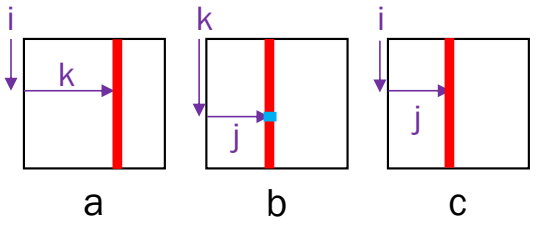
# Spatial Locality Example: Matrix Multiplication



```
for (j=0; j<n; j++)
   for (k=0; k<n; k++)
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * b[k][j];
```

Poor cache effects

- Bad spatial locality for A
- Bad spatial locality for B
- Bad spatial locality for C



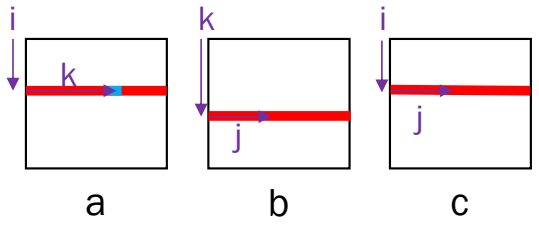
# Spatial Locality Example: Matrix Multiplication



```
for (i=0; i<n; i++)
   for (k=0; k<n; k++)
      for (j=0; j<n; j++)
         c[i][j] += a[i][k] * b[k][j];
```

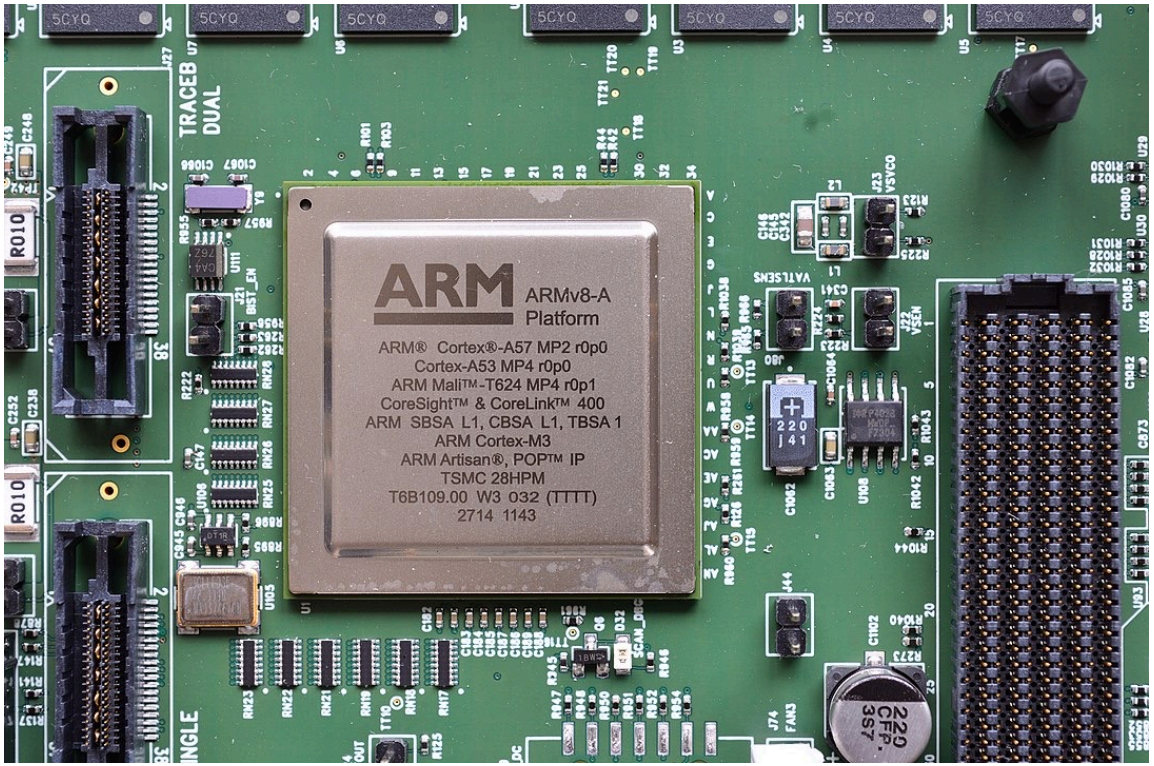
Good cache effects

- Good spatial locality for A
- Good spatial locality for B
- Good spatial locality for C



# Next time …

Getting started with ARM assembly language

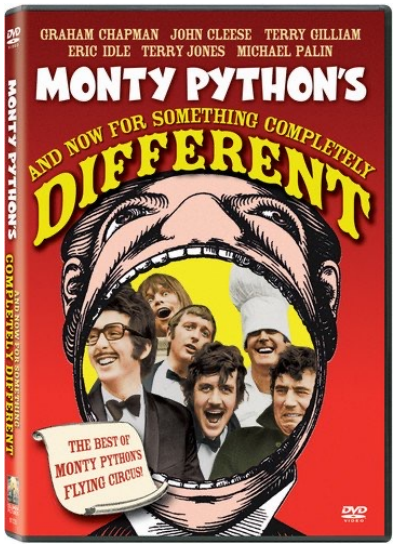



Lobsterthermidor , Raysonho

# COS 217: Introduction to Programming Systems

## Assembly Language

Part 1

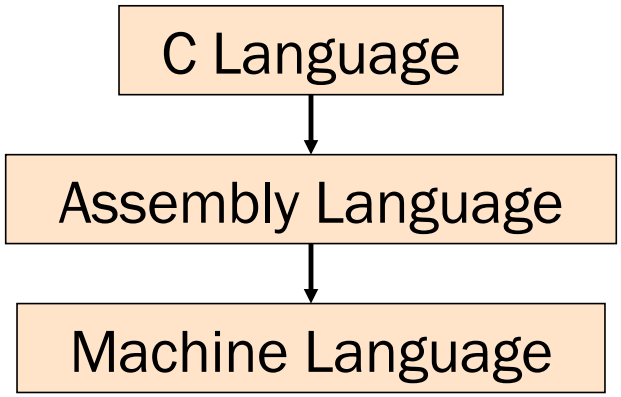



PRINCETON UNIVERSITY

# Context of this Lecture



"Under the hood"



# Agenda



**Language Levels: Abstraction**

Architecture

Assembly Language: Performing Arithmetic

Assembly Language: Load/Store and Defining Global Data

# High-Level Languages



## Characteristics

- Portable (to varying degrees)
- Complex/expressive
  - One statement can do a lot of work – good ratio of functionality to code size
- Human readable
  - Structured: if(), for(), while(), etc.
  - Variable names can hide details of where data is stored (stack, heap, etc.)
  - Push details of representation etc. to compiler, and focus on algorithm

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

# Machine Languages



## Characteristics

- Not portable (hardware-specific)
- Simple
  - Every instruction does a simple task – low expressivity (ratio of functionality to code size)
- Not human readable
  - Not structured
  - Requires a lot 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
```

# Assembly Languages



A human-readable abstraction of machine language

Characteristics

- Not portable
  - Every assembly language instruction maps to one machine instruction
- Simple
  - Every instruction does a simple task
- Human readable …

```
        mov     w1, 0
loop:
        cmp     w0, 1
        ble     endloop
        add     w1, w1, #1
        ands    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:
```

# Why Learn Assembly Language?



Knowing assembly language helps you:

- Write faster code
  - In assembly language (beat the compiler, as in Assignment 5)
  - Even in a high-level language, if you never code in assembly
- Write safer code
  - Understanding mechanism of potential security problems helps you avoid them – even in high-level languages
- Design better systems
  - RISC example
- Become more comfortable with levels of abstraction
  - Become a better programmer at all language levels

# Why Learn ARMv8 Assembly Language?



## Pros of learning ARMv8 (a.k.a. AARCH64 or A64) assembly

- ARM is the most widely used processor architecture in the world (in your phone, in your Mac, in your Chromebook, in Armlab, IoT devices)
- Especially in embedded systems, where optimizing performance/space is critical
- About 46% share by revenue today
- ARM has a modern and (relatively) elegant instruction set ("RISC" – Reduced Instruction Set Computer) with each instruction being the same size (4 bytes). C.f., the expansive but overwhelming x86-64 instruction set

## What's in a name: What does ARM stand for?

# Lectures and Precepts are Complementary



Similar approach we took to learning C along the way:

| Lectures | Precepts |
|---|---|
| Study partial programs (gloss over some things) | Study complete programs |
| Begin with simple constructs; proceed to complex ones | Begin with small programs; proceed to large ones |
| Emphasis on reading code | Emphasis on writing code |

# Agenda



Language Levels

**Architecture**

Assembly Language: Performing Arithmetic

Assembly Language: Load/Store and Defining Global Data

# Architecture



Fixed-program computers (1942-5)

- Do one thing (ABC), configurable with cables (Colossus), programmable with cables (ENIAC)

***Stored program computers***

- Alan Turning (1936): theory
- Konrad Zuse (1936): patent
- J.Presper Eckert and John Mauchly (1943-44): Design (ENIAC, buit in EDVAC)
- John von Neumann (1945): popularization
  - Became famous as von Neumann architecture: Made Eckert very upset
- F.C. Williams and Tom Kilburn (1948): first working machine (Manchester Baby)
- Maurice Wilkes and team (1949): first to run as computer service (EDSAC)

# John von Neumann (1903-1957)



I can not:

- Divide two 8 digit numbers in my head.  Converse in Ancient Greek.
- I am not a 6-year-old von Neumann
- Wigner: only one Hungarian genius; Bethe: ”superior species”

Princeton connection

- Princeton University & IAS, 1930-1957; von Neumann Hall
- https://paw.princeton.edu/article/early-history-computing-princeton
- “von Neumann architecture” also known as Princeton architecture, distinct from the less-used Harvard architecture (microcontrollers)

Mathematics, statistics, nuclear physics, computing

- Stored program computers
- Cellular automata, self-replication
- Game theory
- Mergesort (but with a bug, found by Don Knuth)

# “Von Neumann” Architecture



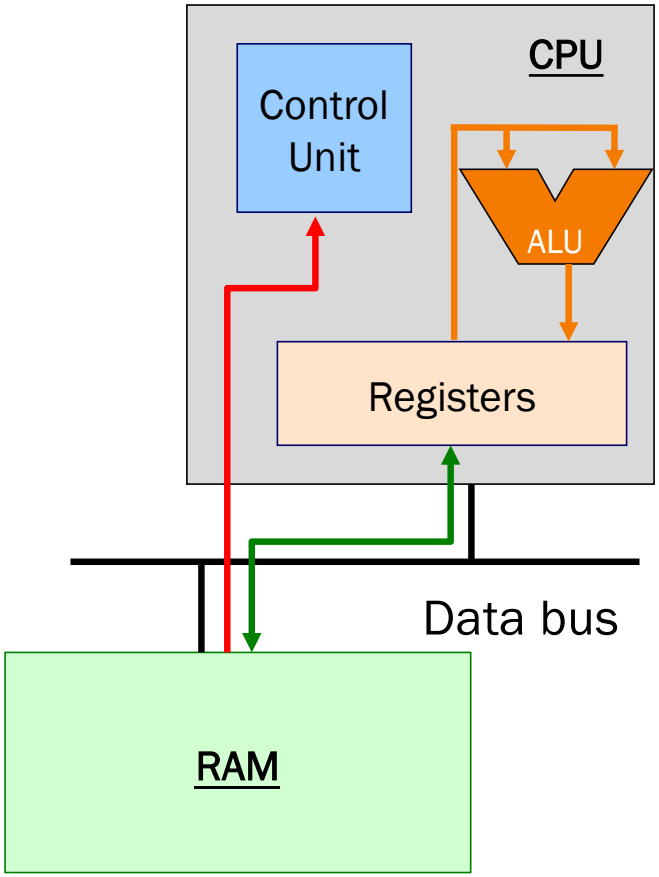
CPU(s) and RAM(s)

Within CPU:

Instructions (encoded within words) are fetched from RAM, just like data

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



# Von Neumann Architecture



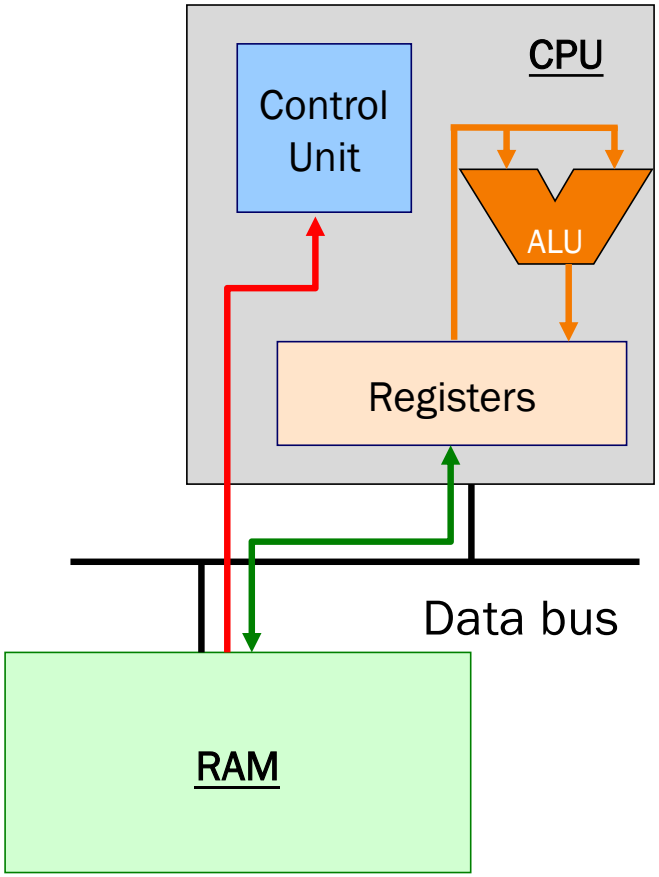
Within CPU:

Registers

Small amount of storage on the CPU

- Top of the “storage hierarchy”
- Very {small, expensive, fast}
- Temporarily store variables, addresses, scratch space

ALU instructions operate on registers



# Von Neumann Architecture



RAM (Random Access Memory)

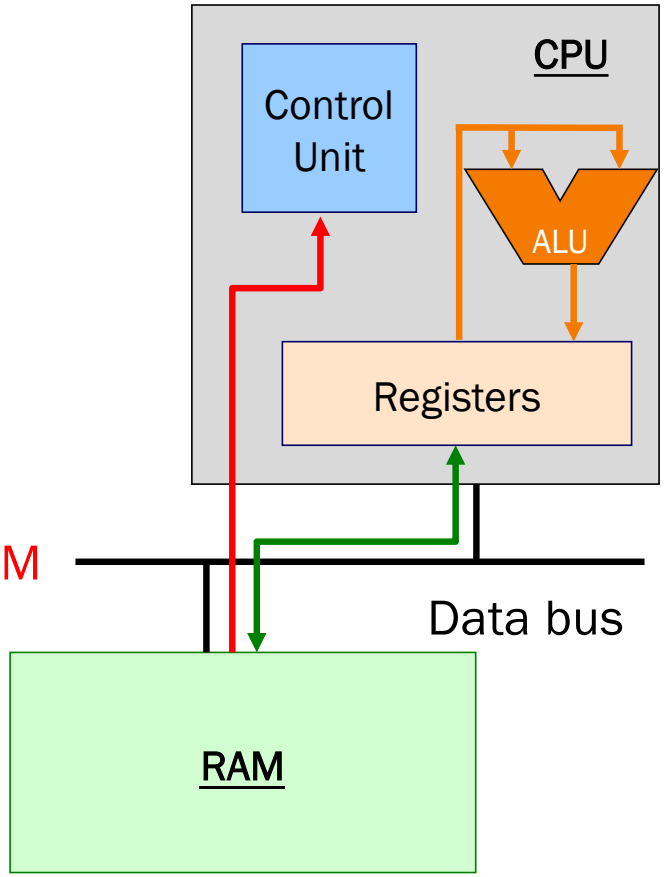
Conceptually: large array of bytes

(gigabytes+ in modern machines)

(registers are ~512 bytes)

- Contains data (variables, data structures)
- And the program itself, in machine code

Instructions and data are both fetched from RAM



# Time to reminisce about old TOYs

Thinking back to COS 126,
how did you feel about TOY?

A. Loved it

B. Wasn't a fan.

C. I took ECE115, so I have no idea what you're talking about

D. I placed out, so I have no idea what you're talking about



# Time to reminisce about old TOYs



```
                TOY REFERENCE CARD


INSTRUCTION FORMATS

              | . . . . | . . . . | . . . . | . . . .|
  Format RR:  | opcode  |    d    |    s    |    t    |  (0-6, A-B)
  Format A:   | opcode  |    d    |       addr        |  (7-9, C-F)


ARITHMETIC and LOGICAL o
    1: add
    2: subtract
    3: and
    4: xor
    5: shift left
    6: shift right

TRANSFER between registe
    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 entity 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.

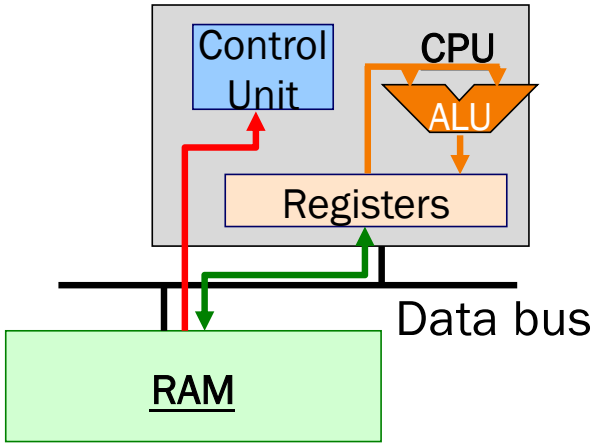
https://introcs.cs.princeton.edu/java/62toy/

# Registers and RAM



Typical TOY pattern:

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



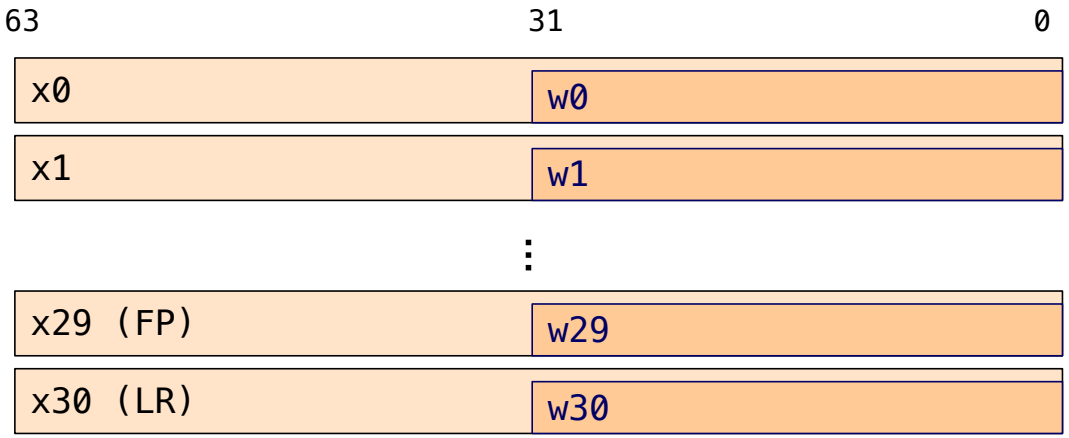

On AARCH64, this pattern is enforced

- "Manipulation" instructions can *only* access registers
- Called a **load-store architecture** (unlike "register-memory" architectures)
- Characteristic of RISC architectures (vs."CISC" or Complex Instruction Set Computer, e.g. x86)

Even RISC compilers don't use all their instructions

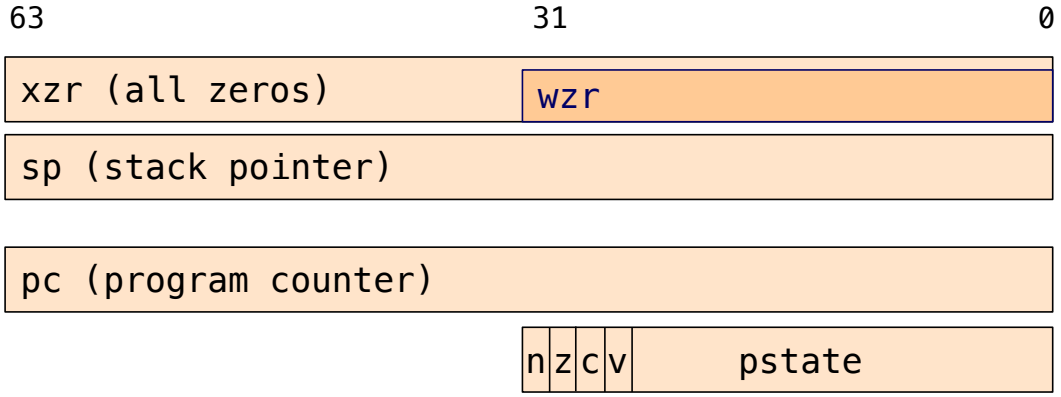
- We'll use this fact to "beat the compiler" in Assignment A5. CISC is much worse

# Registers (ARM-64 architecture)




General Purpose Registers

Special Registers



# General-Purpose 64-bit Registers



## X0 … X30

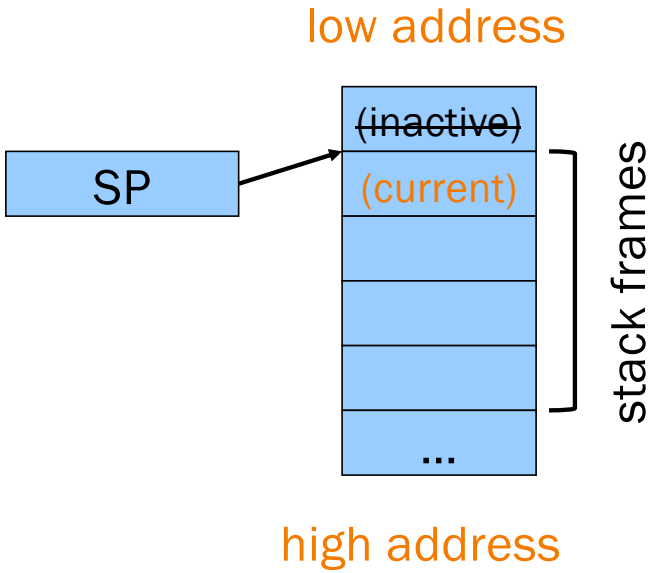
- Scratch space for instructions, parameter passing to/from functions, return address for function calls, etc.
- Some have special roles defined *in hardware* (e.g. X30 as return pointer) or defined *by software convention* (e.g. X29 as frame pointer)
- Also available as 32-bit versions: W0 … W30

# SP Register



Special-purpose register…

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




Allows use of the STACK section of memory

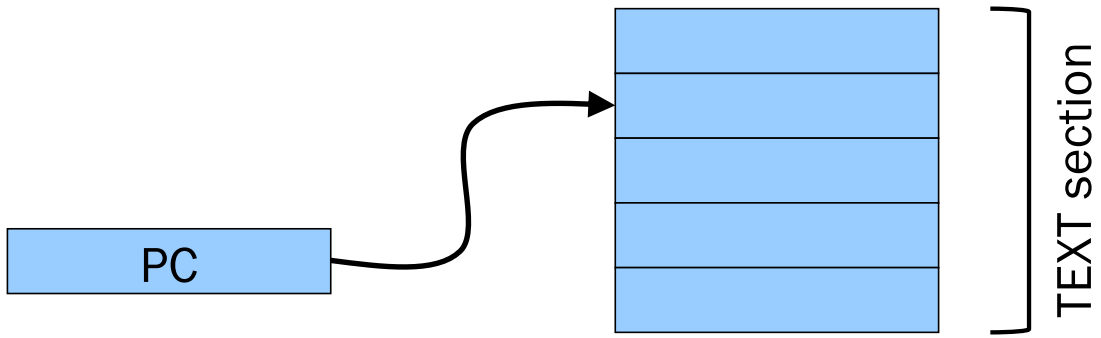
Hardware support for function calls (later lecture)

# PC Register

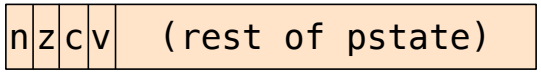


Special-purpose register…

- **PC (Program Counter)**
- Stores the location of the next instruction
  - Address (in TEXT section) of machine-language instruction to be executed next
- Its value is changed either:
  - Automatically to implement sequential control flow (increment by 4 bytes)
  - By branch instructions to implement selection, repetition



# PSTATE Register




Special-purpose register…

- Contains **condition flags:**

  n (**N**egative), z (**Z**ero), c (**C**arry), v (o**V**erflow)
- Affected by compare (`cmp`) instruction
  - And many others, if requested (e.g. with s suffix on arithmetic instructions)
- Used by conditional branch instructions
  - `beq`, `bne`, `blo`, `bhi`, `ble`, `bge`, …
  - (See **Assembly Language: Part 2** lecture)

XZR: Special register whose value is always 0, more to come

# Agenda



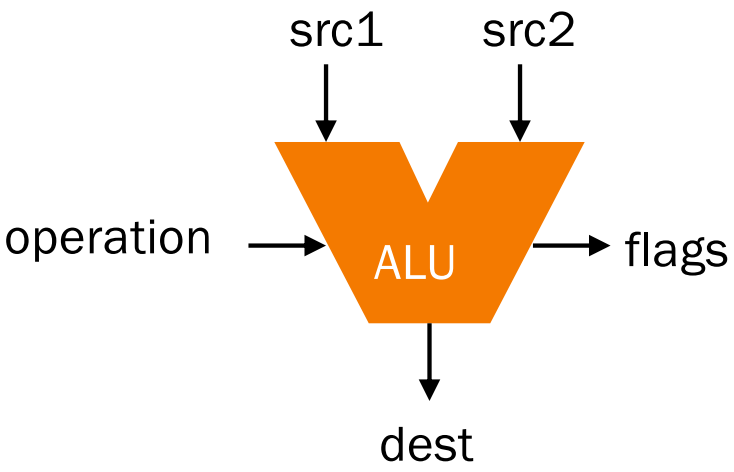
Language Levels

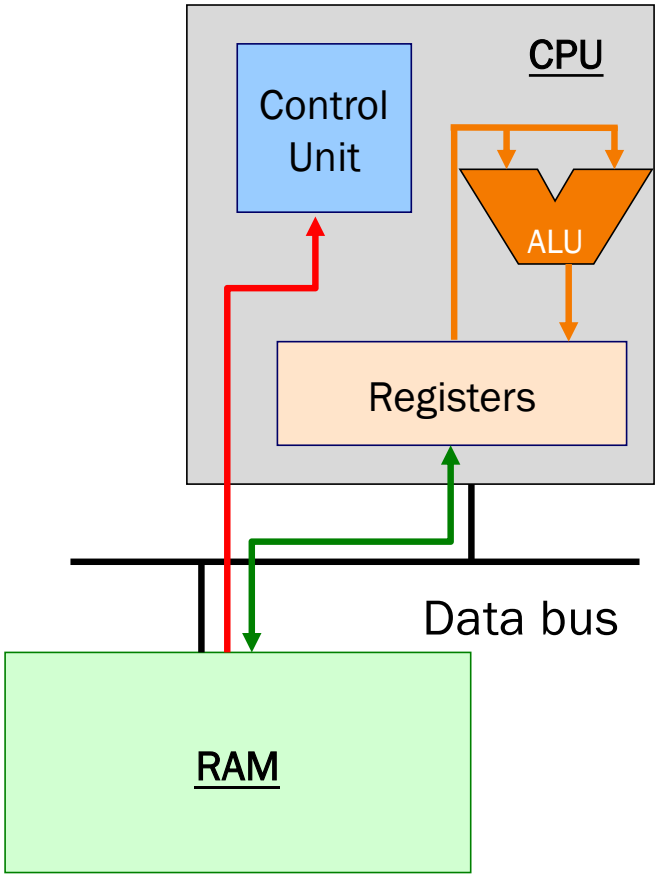
Architecture

**Assembly Language: Performing Arithmetic**

Assembly Language: Load/Store and Defining Global Data

# ALU Arithmetic Example



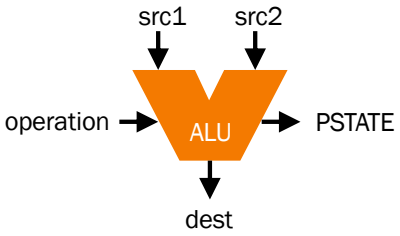

```
add dest, src1, src2
```

# Instruction Format



Many instructions have this format:

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




- **name:** mnemonic name of the instruction (`add`, `sub`, `mul`, `and`, etc.)
- **s:** if present, specifies that condition flags should be **S**et

- dest and src1,src2 are **x** registers: 64-bit operation
- dest and src1,src2 are **w** registers: 32-bit operation
  - No mixing and matching between x and w registers

- src2 may be a constant ("immediate" value) instead of a register
  - Limited in size (no. of bits): will see in Assignment 6

# 64-bit Arithmetic



C code:

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

Assume that…

- length is already held in x1
- width is already held in x2
- perim is already held in x3
- global variables (we'll see local later)

Assembly code:

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

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

You'll see "#1" instead of "1" in book. Also works, but not necessary.

# 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 held in x1
- y held in x2
- z held in x3

Assembly code:

```
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: unlike C, separate instructions for arithmetic and logical shifts

# XZR Register for Shortcuts



Special purpose register: always zero …

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

Enables several things with one register:

- zeroing registers
- copying values (OR with XZR)
- compare two registers to set condition codes but throw away result by making XZR destination

# Arithmetic Shortcuts with XZR



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

Assume that…

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

Assembly code:

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

```
orr  x3, xzr, x1
sub  x3, xzr, x1
```

*Abstraction*

These are actually assembler shortcuts for instructions that use XZR.



*Precepts will cover signed/unsigned arithmetic and handling smaller operand sizes*

# Signed vs Unsigned?



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

Assume that…

- x held in x1
- y held in x2

Assembly code:

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

Mostly the same algorithms, same instructions

- Can set different condition flags in PSTATE
- But some exceptions…

# Signed vs Unsigned: Exceptions



```
static long x;
static unsigned long y;
...
x /= 17;
y /= 42;
x >>= 1;
y >>= 2;
```

Assume that…

- x held in x1
- y held in x2

Assembly code:

```
sdiv x1, x1, 17
udiv x2, x2, 42
asr x1, x1, 1
lsr x2, x2, 2
```

Division needs different algorithms for signed and unsigned, and hence different instructions

"Arithmetic" right shift (shift in sign bit on left) vs. "logical" right shift (shift in zeros on left)

# 32-bit Arithmetic using “w” registers



C code:

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

Assume that…

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

Assembly code:

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

# 8- and 16-bit Arithmetic?



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

No specialized arithmetic instructions

- Just use the “w” registers
- Corresponds to C language semantics: all arithmetic is implicitly done on (at least) ints, values are implicitly promoted to 32 bit
- Specialized “load” and “store” instructions for transfer of shorter data types from / to memory – we’ll see these next

# Agenda



Language Levels

Architecture

Assembly Language: Performing Arithmetic

**Assembly Language: Load/Store and Defining Global Data**

# Loads and Stores



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

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

- `dest`, `src` are registers
- The memory operand (address register) always comes second
- **Memory addressing must be via registers (not literal addresses as in TOY)**
  - Every memory access effectively dereferences a 'pointer' (the register)
- Contents of registers in [brackets] must be memory addresses
  - Nicely represents memory as a C array with the addresses being its indices

- The addresses (`src` for `ldr`, `dest` for `str`) must be in x-flavored registers
- Other operands (`dest` for `ldr`, `src` for `str`) can be x-flavored or w-flavored

# 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) and shorts ("Half-words": 16 bit)

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

Special instructions for signed loads (tell how to fill in leftmost bits in register)

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

# A Note on Loads and Stores



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

- How to get correct memory address into register?
  - Depends on whether data are on stack (local variables), on heap (dynamically-allocated memory), or global / static
  - For today, we'll look only at the global / static data case (DATA or BSS)
  - There's a pseudo-instruction for that:
    - `adr reg, label` puts address of a labeled memory location in a register
    - Load/store can then use this register as a pointer/address (memory operand)
    - But how do we know or reference the label? Let's see with an example program

# Our First Full Program (Perimeter)*



```
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
```



* You'll see the more typical, "Hello, World!" program in precept …

# Memory Sections



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

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

Sections (Stack/heap are different!)

- .rodata: read-only
- .data: read-write
- .bss: read-write (uninit or init to 0)
- .text: read-only, program code

```
  .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
```

Will see stack later
Won't consider heap
- leave it to malloc

# Variable Definition and Usage with Labels



```
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
```

We have variables in assembly language

- Abstraction

Declaring data

"Labels" for locations in memory

.word: allocate 4 bytes and store this value there

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

# main()



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

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

Global visibility

.global: 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
```

In C external linkage is default

- Use static if want to keep it internal

In ARM, internal linkage is default

- Use .global if want external visibility

main function is called by _start, so must be globally visible

# Make a "Pointer" using `adr` and Labels



```
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
```

# Loads and Stores



```
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
```

Load and store

- Use x0 as a “pointer” to load from and store to memory
- Dereference using [x0]

# 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: return 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
```

We `mov` 0 to w0 before `ret` because 0 is our return value



* w0 for int, x0 for long

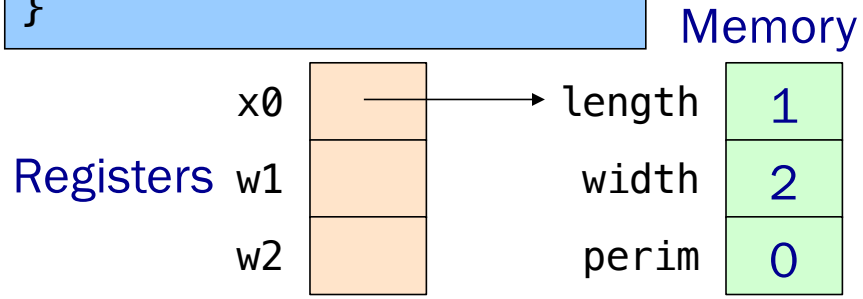
# Tracing Our Program



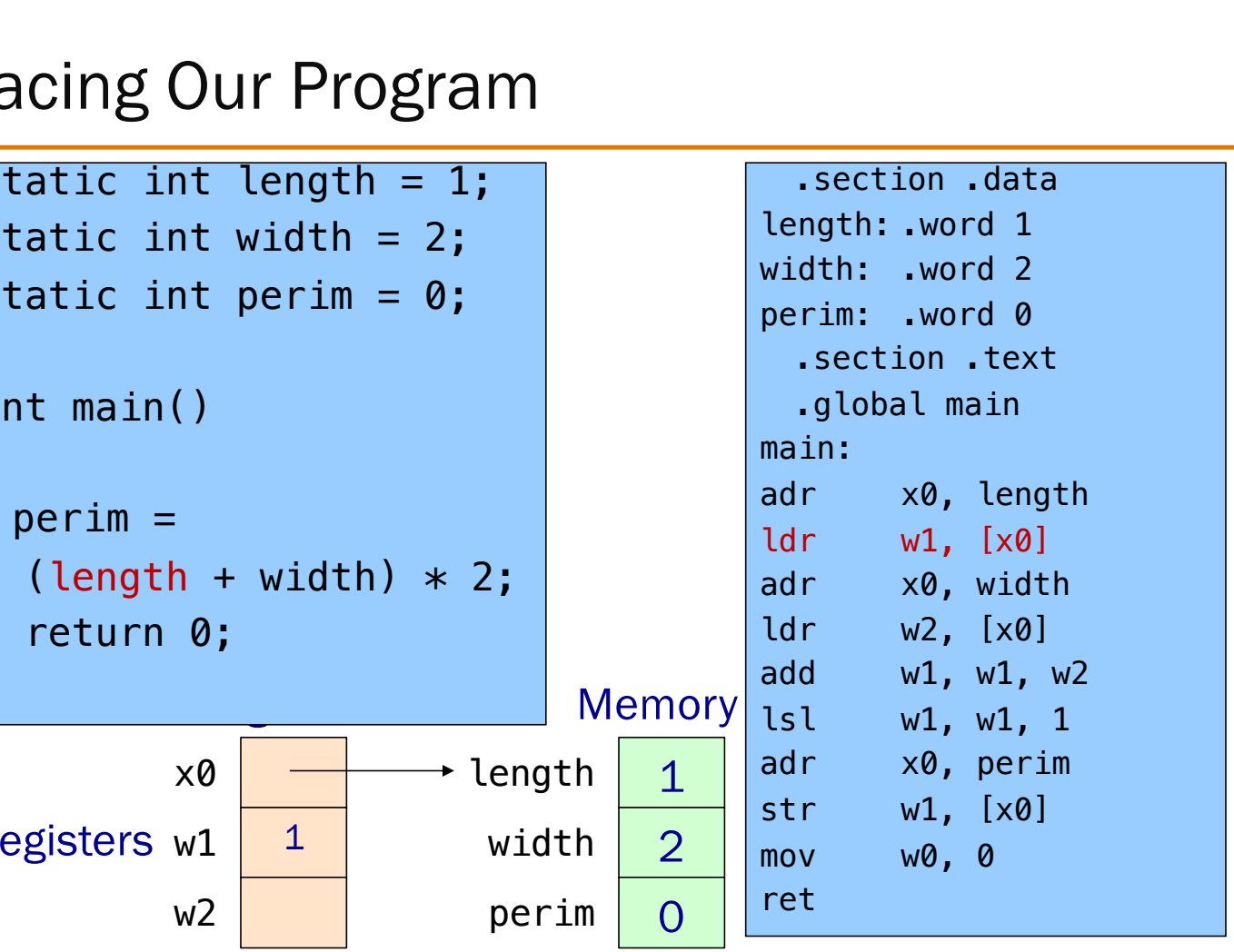
```
static int length = 1;
static int width = 2;
static int perim = 0;

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
```



# Tracing Our Program

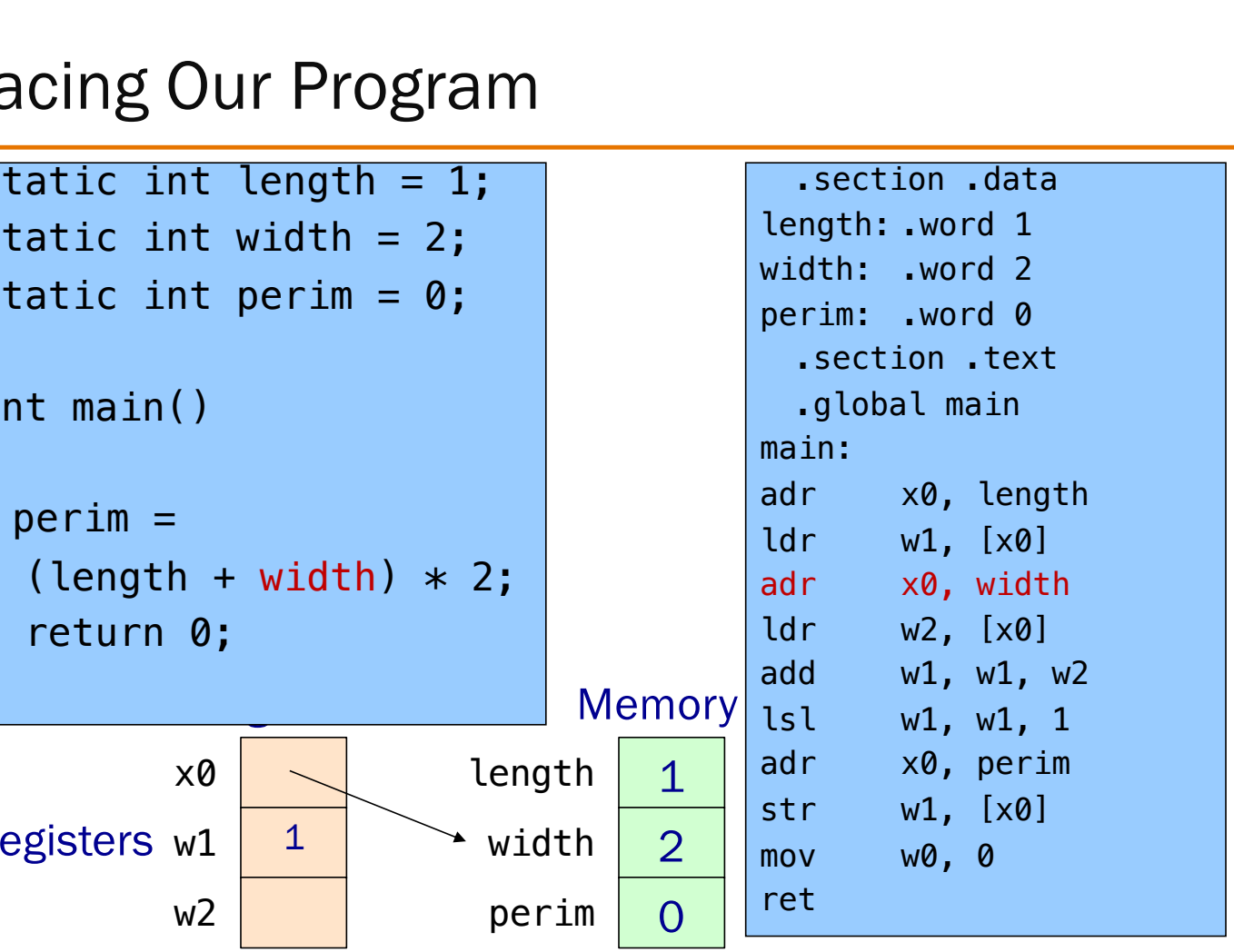


```
static int length = 1;
static int width = 2;
static int perim = 0;

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
```

# Tracing Our Program



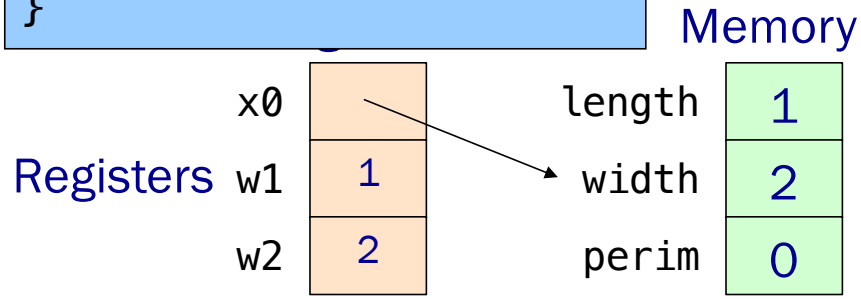
# Tracing Our Program



```
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
```



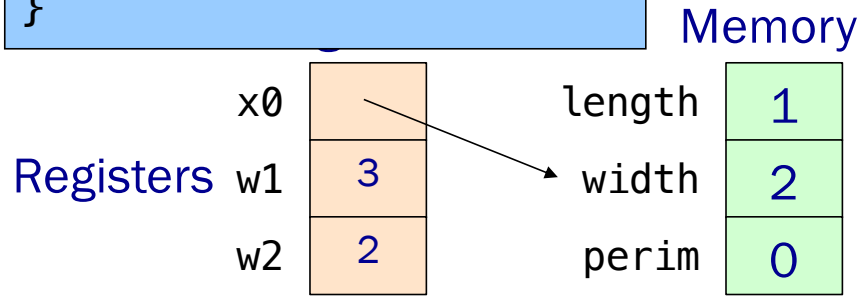
# Tracing Our Program



```
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
```



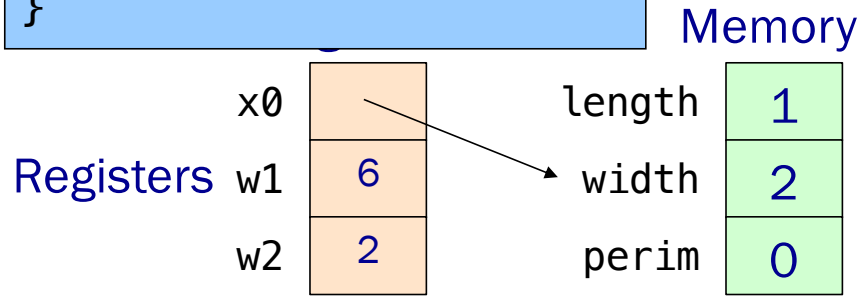
# Tracing Our Program



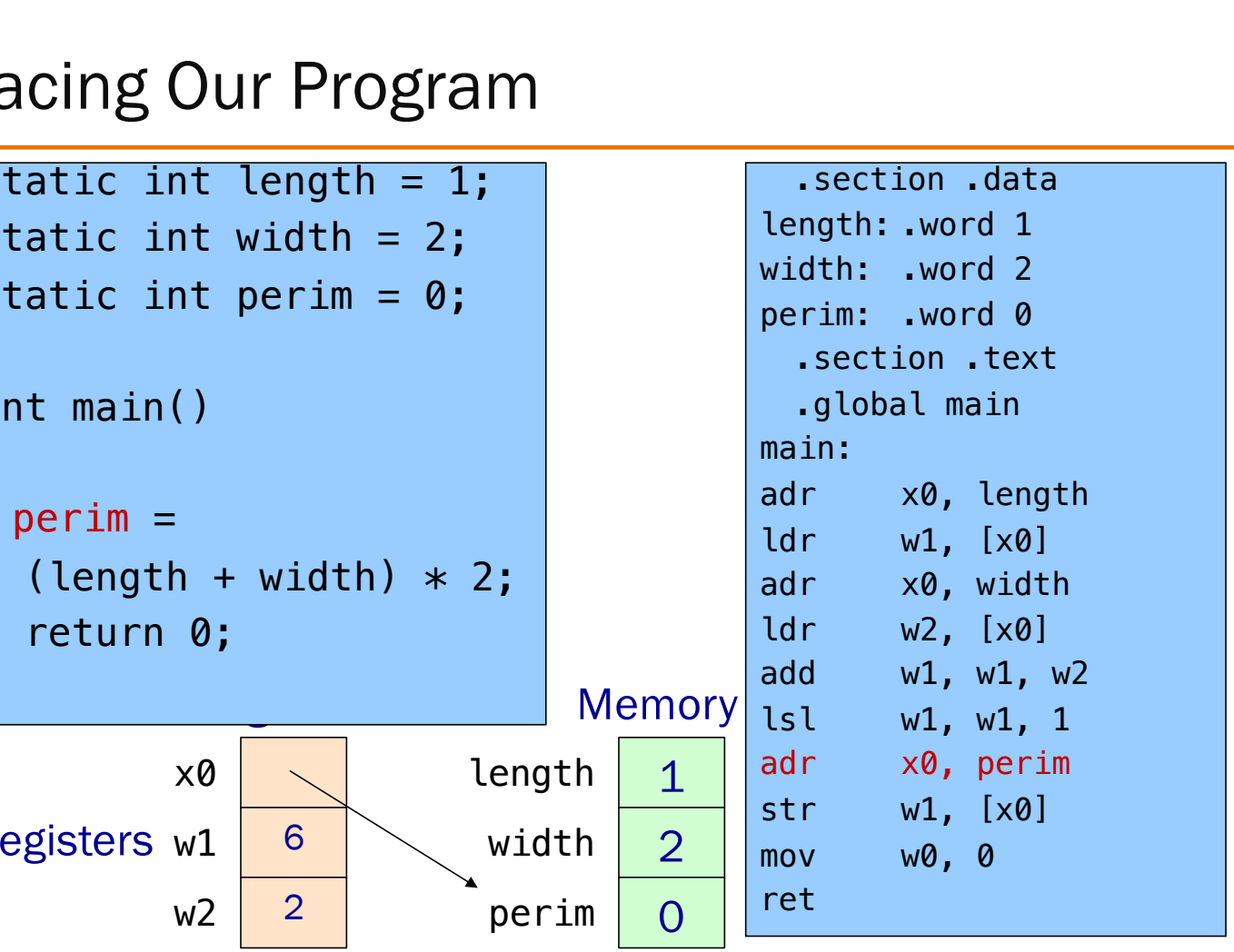
```
static int length = 1;
static int width = 2;
static int perim = 0;

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

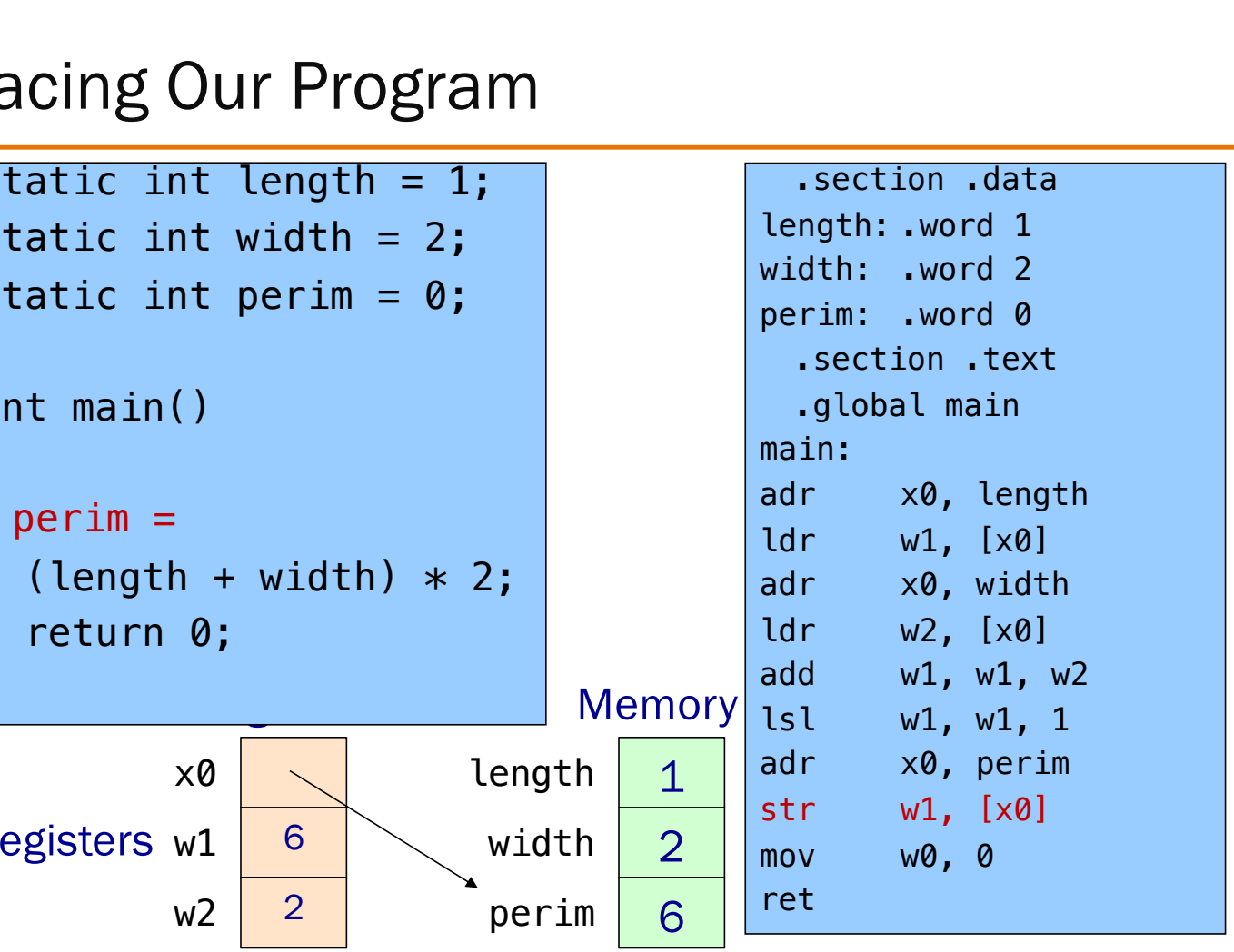
```
  .section .data
length: .word 1
width:  .word 2
perim:  .word 0
  .section .text
  .global main
main:
adr     x0, length
ldr     w1, [x0]
adr     x0, width
ldr     w2, [x0]
add     w1, w1, w2
lsl     w1, w1, 1
adr     x0, perim
str     w1, [x0]
mov     w0, 0
ret
```



# Tracing Our Program



# Tracing Our Program



# Tracing Our 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
```

Return value

Passed back in register w0

# Tracing Our Program



```
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
```

# Summary



Language levels: High-level language, assembly language, machine language

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 (though it will be challenging)
  - `gcc217 –S somefile.c`
- We will have three more lectures and four precepts on AARCH64



Appendix 1

# DEFINING DATA: OTHER SECTIONS AND SIZES

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

# 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 or directive

`.global` directive can also apply to variables, not just functions

# 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` directive
  (to announce BSS section)
- `.skip` directive
  (to specify number of bytes)

# Defining Data: RODATA Section



```
…
…"hello\n"…;
…
```

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

Notes:

- `.section` directive (to announce RODATA section)
- `.string` directive



Appendix 2

# BYTE ORDER: BIG-ENDIAN VS LITTLE-ENDIAN

# 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 (four bytes) 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 (four bytes) 5 at address 1000:

| | |
|---|---|
| 1000 | 00000000 |
| 1001 | 00000000 |
| 1002 | 00000000 |
| 1003 | 00000101 |

# 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
Byte 1: 77
Byte 2: 33
Byte 3: 00
```

Output on a big-endian machine

```
Byte 0: 00
Byte 1: 33
Byte 2: 77
Byte 3: ff
```

# Byte Order Example 2



Note:

Flawed code; uses “b” instructions to load from a four-byte memory area

```
    .section ".data"
foo: .word 7
    .section ".text"
    .global “main”
main:
adr   x0, foo
ldrb  w0, [x0]
ret
```

AARCH64 is little endian, so what will be the value returned from w0?

What would be the value returned from w0 if AARCH64 were big endian?