Hello World (using .asciz)

```
.data
        hello_msg: .asciz "Hello World\n”

.text
main:
        la a0, hello_msg # load the addr of hello_msg into a0
        addi a7, zero, 4        # 4 is the print_string syscall.
        ecall                   # do the syscall.

        addi a7, zero,10        # 10 is the exit syscall.
        ecall                   # do the syscall.
```

Hello World (using .ascii)

Another way to declare the string "Hello World\n" and get the exact same output

```
.data
    hello_msg: .ascii "Hello"       # The word "Hello"
    .ascii " "                      # the space.
    .ascii "World"                  # The word "World"
    .ascii "\n"                     # A newline.
    .byte 0                         # a 0 byte.
```

Hello World (using Bytes)

Yet another way to declare the string "Hello World\n" and get the exact same output

```
.data
    hello_msg: .byte 0x48 # hex for ASCII "H"
    .byte 0x65 # hex for ASCII "e"
    .byte 0x6C # hex for ASCII "l"
    .byte 0x6C # hex for ASCII "l"
    .byte 0x6F # hex for ASCII "o"
    ... # and so on...
    .byte 0xA # hex for ASCII newline
    .byte 0x0 # hex for ASCII NUL
```

Live Coding Time

- **Download code from Canvas**
  - io.s:
    - Did you get an error message after execution? How to fix it?
    - Can you modify the code to display the number in hex?
  - prompt.s:
    - understand how to display prompts
  - HelloWorld.s
    - try the other two versions and compare the output
- **Run in RARS**
  - RARS help menu has lots of useful information

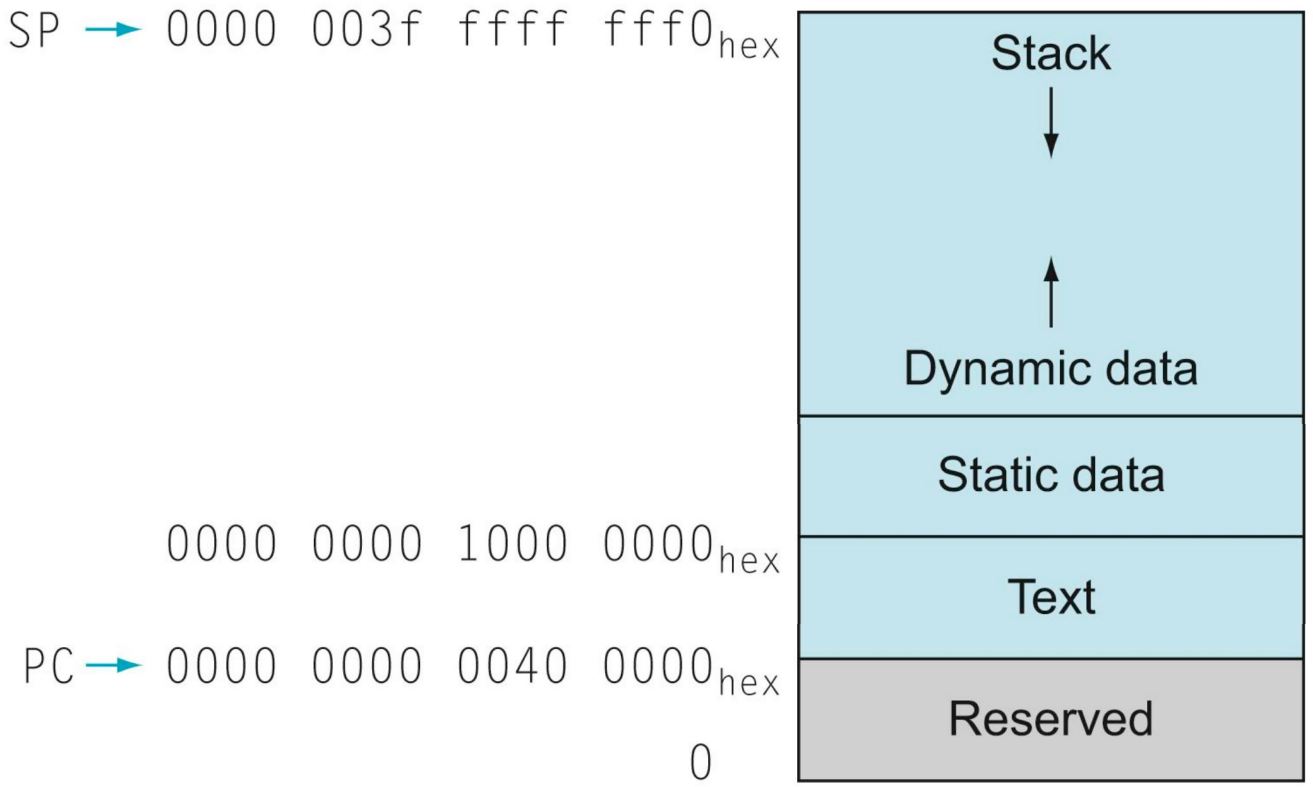
Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - ➔ Memory access instructions (Chapters 2.3, 2.9)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

Role of Registers vs. Memory

- **Which is faster to access?**
- **Which do we have more of?**
- **What to keep in register?**
- **What if more variables than registers?**
  - Compiler tries to keep most frequently used variable in registers
  - Less common in memory: spilling
- **Why not keep all variables in memory?**
  - Smaller is faster: registers are faster than memory

RISC-V Memory Allocation

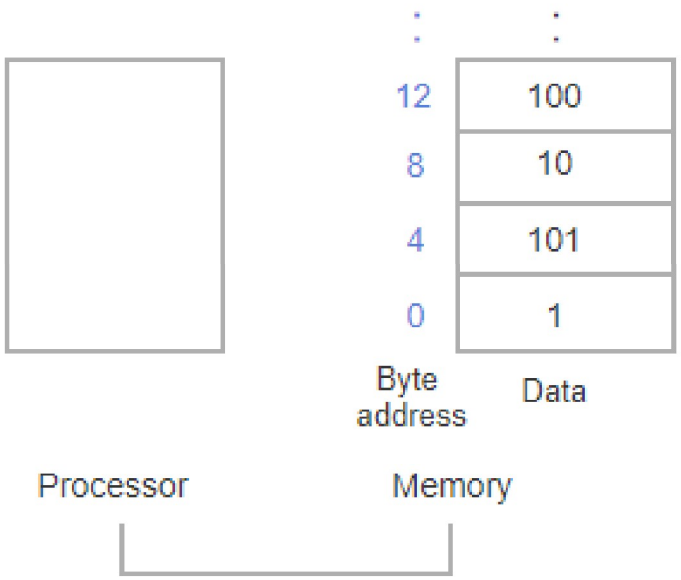


- **Static**: Variables declared once per program, cease to exist only after execution completes. e.g., C globals, arrays
- **Heap**: Variables declared dynamically (malloc)
- **Stack**: Space to be used by procedure during execution; this is where we can save register values

Memory Addressing

- **Every word in memory has an address, similar to an index in an array**
- **Each address is a 32-bit word => 4GB of memory**
- **Byte Addressed**
  - Computers need to access 8-bit bytes as well as words (4 bytes/word)
  - ⇒ address memory as bytes
  - ⇒32-bit (4 byte) word addresses differ by 4
  - ⇒ `Memory[0], Memory[4], Memory[8], ...`
- **sequential word addresses in machines with byte addressing differ by 4, not 1**

Memory Addressing Illustrated



15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

Showing all byte addresses

Memory for Large Data Structures

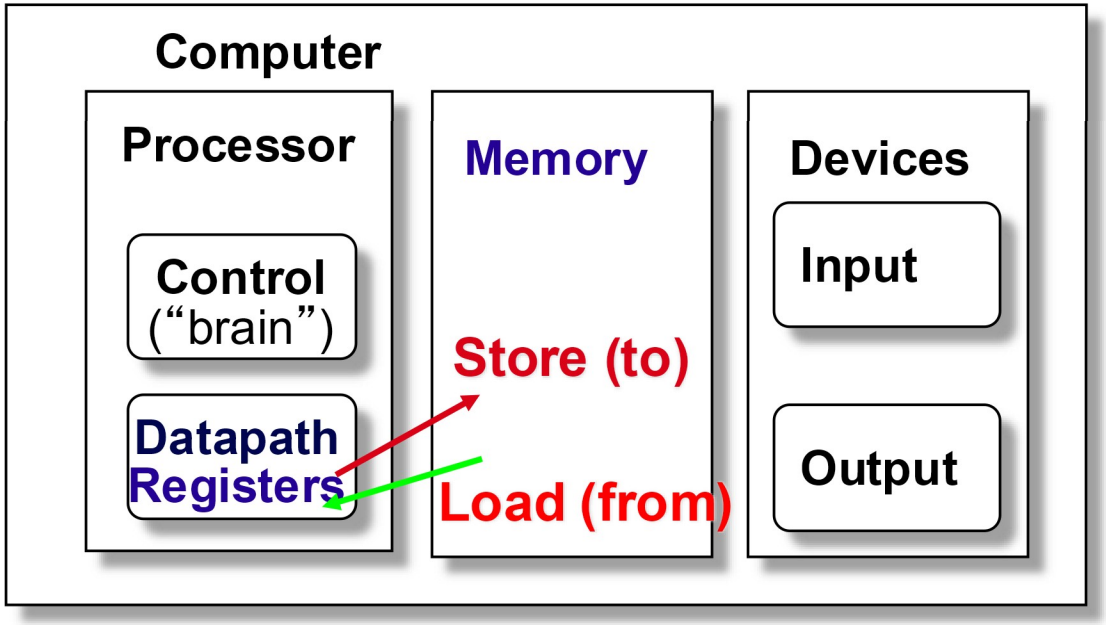
- **C variables map onto registers; what about large data structures like arrays?**
  - use memory
- **think of memory as a single one-dimensional array**

Array[index]	Address	Data or Content
A[5]	5020	32
A[4]	5016	23
A[3]	5012	14
A[2]	5008	72
A[1]	5004	56
A[0]	5000	44

Assuming A[ ] is an integer array

Data Transfer Instructions

- RISC-V arithmetic instructions only operate on registers, never directly on memory.
- Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.
- Data transfer instructions
  - Memory to register
  - Register to memory



Memory to Register Transfer

- **To transfer a word of data, we need to specify two things:**
  - Register: specify this by number (x0 - x31) or symbolic name (zero, a0, …)
  - Memory address:
    - sum of two values
      - A register containing a pointer to memory address
      - Numerical offset from this pointer (in bytes)
    - Example: `8(t0)`
      - specifies the memory address pointed to by the value in t0, plus 8 bytes
- **Load FROM memory**

Load Word

- **Load Instruction Syntax:**

  1 2,3(4)

  - where

    1) operation name

    2) register that will receive value

    3) numerical offset in bytes

    4) register containing pointer to memory

- **RISC-V Instruction Name:**

  - `lw` : meaning Load Word, so 32 bits (i.e., one word) are loaded at a time

Load Word: Example

Example: `lw t0,12(s0)`

This instruction will take the pointer in `s0`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `t0`

- **Notes:**
  - `s0`: base register
    - Often points to beginning of an array or structure
  - 12: offset
    - often used in accessing elements of array or structure

Register to Memory Transfer (sw)

- **Store from register into memory**
  - syntax is identical to lw
  - `sw` (meaning Store Word, so 32 bits or one word are loaded at a time)

- **Example:** `sw t0,12(s0)`

  This instruction will take the pointer in `s0`, add 12 bytes to it, and then store the value from register `t0` into that memory address

- **Store INTO memory**

Example 1

- **Compile using registers:**

  `g = h + A[5];`

  - `g`: `s1`, `h`: `s2`, `s3`: base address of integer array `A`

- **What offset in `lw` to select `A[5]`**
- **4x5=20 to select `A[5]`: byte vs. word**

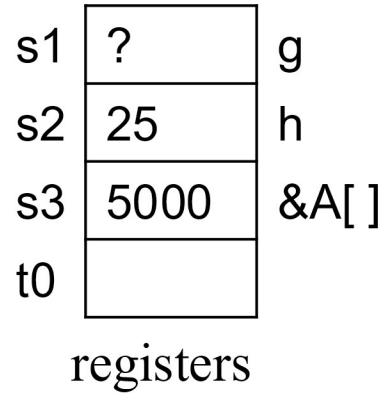
Array[index]	Memory Address	Data or Content
A[5]	5020	32
A[4]	5016	23
A[3]	5012	14
A[2]	5008	72
A[1]	5004	56
A[0]	5000	44

1. transfer from memory to register:

lw t0,20(s3) # *a*dd 20 to s3 to select A[5], #put into t0

2. **add it to `h` and place in `g`**

add s1,s2,t0 *# s1 = h+A[5]*



Example 2

- **What offset in `lw` to select `A[i]` (`integer array`)**
  - 4i
- **Compile using registers:**

```
g = h + A[i];
```

  - `g`: `s1`, `h`: `s2`, `s3`:base address of A; i: s4

```
add s4, s4, s4
add s4, s4, s4
add s4, s4, s3
lw  t0,0(s4)
add s1,s2,t0
```

Loading and Storing Bytes

- **byte data transfers:**
  - load byte: `lb`
  - store byte: `sb`
- **same syntax as `lw` and `sw`**
- **What to do with other 24 bits in the 32-bit register?**
  - `lb`: sign extends

  - `lbu`: zero extends
    - e.g., for chars

Working with Characters and Strings

- **ASCII**
  - American Standard Code for Information Interchange
  - 8-bit representation for English alphabet, numbers
  - RARS time: .word 0x65666768
- **Null-terminated series of characters are strings (same as in C)**
- **lb, sb used to read/write one char at a time**

C code:

```
char name [20];
name[3]
```

RISC-V code:

```
name: .space 20
la t0, name
lb t1, 3 (t0)
```

Other Load/Store Instructions

- **Half word:**
  - 2 bytes
  - Half word aligned
  - lh, lhu: load half word
  - sh: store halfword
- **For floating point numbers**
  - Use FP registers f0-f31
  - flw, fld – load float SP/DP
  - fsw, fsd – store float SP/DP