Module 06: Instruction Set Architecture, RISC-V Assembly Programming, and Assembly Format of a C Program

Unit 2: Memory Operands and Memory Access Instructions

ITSC 2181 Introduction to Computer Systems
College of Computing and Informatics
Department of Computer Science

Module 06: Instruction Set Architecture, RISC-V Assembly Programming, and Assembly Program of a C Program

- **Unit 1: Module overview, Instruction Set Architecture (ISA) and assembly programs, registers, instruction operations and operands, register and immediate operands, arithmetic and logic instructions**
- ☛ **Unit 2: Memory Operands and Memory Access Instructions**
- **Unit 3: Conditional control instructions for making decisions (if-else) and loops**
- **Unit 4: Supporting Functions and procedures**
- **Unit 5: Sort examples and comparison with other ISAs**

- **Materials are developed based on textbook:**
  - Computer Organization and Design RISC-V Edition: The Hardware/Software Interface, Amazon
  - RISC-V Specification: https://riscv.org/technical/specifications/
  - ITSC 3181: https://passlab.github.io/ITSC3181/

Three Kinds of Operands and Three Classes of Instructions

- General form:
  - <op word> <dest operand> <src operand 1> <src operand 2>
  - E.g.: add x5, x3, x4, which performs [x5] = [x3] + [x4]

Three Kinds of Operands

1. Register operands, e.g., x0 – x31
2. Immediate operands, e.g., 0, -10, etc
3. Memory operands, e.g. 16(x4)

Module 06: Unit 1

Module 06: Unit 2

Module 06: Unit 3

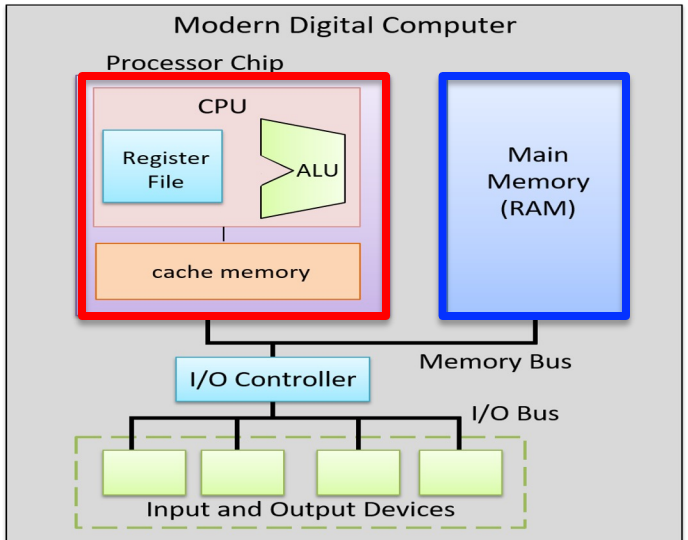
Three Classes of Instructions

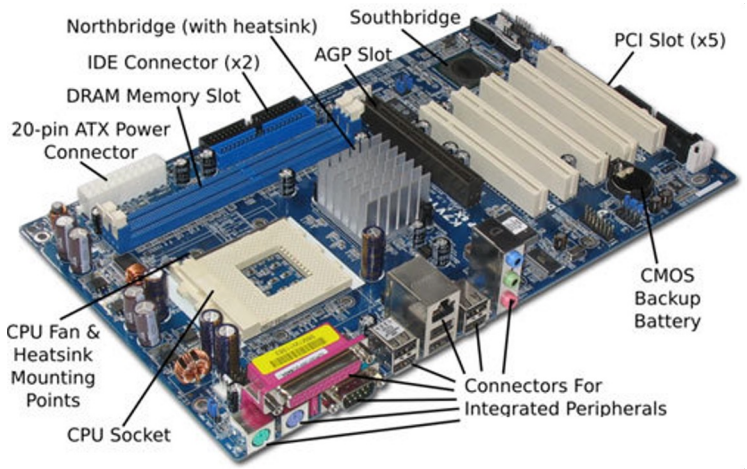
1. Arithmetic-logic instructions
   - add, sub, addi, and, or, shift left|right, etc
2. Memory load and store instructions
   - lw and sw: Load/store word
   - ld and sd: Load/store doubleword
3. Control transfer instructions (changing sequence of instruction execution)
   - Conditional branch: bne, beq
   - Unconditional jump: j (
   - Procedure call and return: jal and jr

Instructions Used So Far: add, addi, sub and slli

```
add  x10, x5, x6        // [x10] = [x5] + [x6]
addi x10, x5, 100       // [x10] = [x5] + 100
sub  x11, x5, x6        // [x11] = [x5] – [x6]
slli x12, x5, 5         // [x12] = [x5] * 2^^5
```

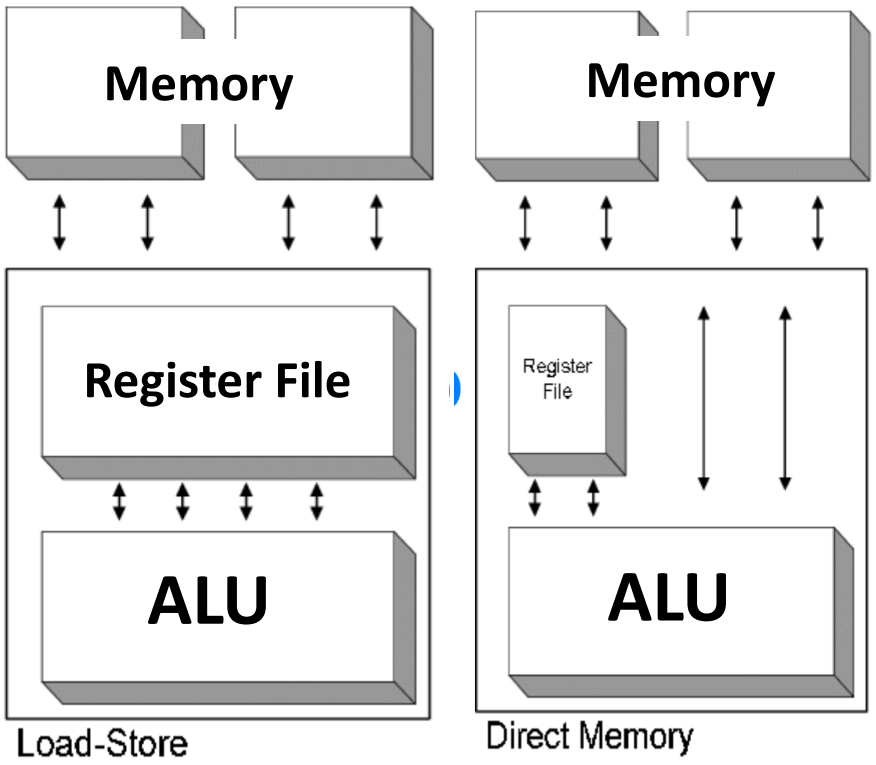
- All within CPU, i.e. ALU (Arithmetic Logic Unit) and register
  - How to access memory?





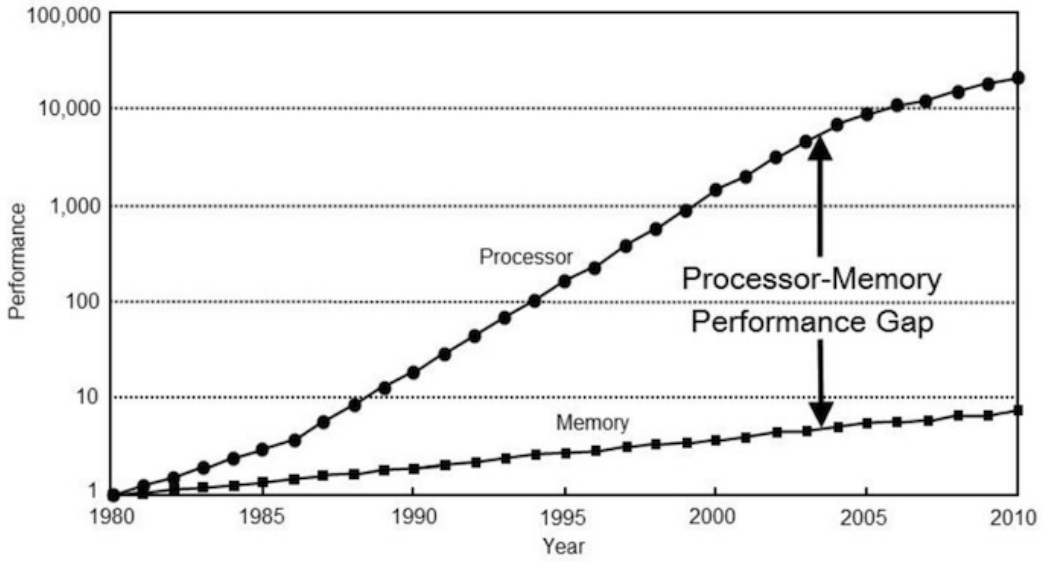
Module 6 Unit 2 Assigned Reading and Pre-knowledge Check

- Load-store architecture
  - Memory access and ALU operations are separated
  - Data must be loaded or stored between memory and register first
  - ALU can only operates on register/immediate operands
  - E.g. RISC-V, ARM, MIPS, PowerPC, SPARC
- Register-memory architecture
  - ALU operation can be performed on/from memory,
    - E.g. add, x6, x6, 0(x22)
  - Example: X86
- Why?: Memory wall
  - Large speed gap between CPU and DRAM



Module 6 Unit 2 Assigned Reading and Pre-knowledge Check

- Load-store architecture
- Register-memory architecture
- Memory wall problem and DRAM speed
  - The memory wall problem refers to a phenomenon that occurs in computer architecture when the processor's speed outpaces the rate at which data can be transferred to and from the memory system. As a result, the processor must wait for the data to be fetched from memory, which slows down its performance and limits its speed.

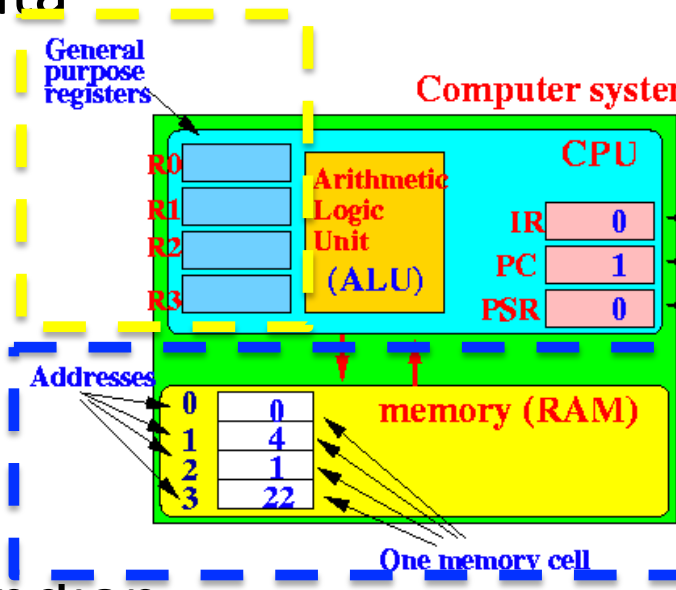


Computer Architecture: A Quantitative Approach by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau

DRAM Generation	Frequency Range
DDR1	200 MHz to 400 MHz
DDR2	400 MHz to 1600 MHz
DDR3	800 MHz to 2133 MHz
DDR4	1600 MHz to 5333 MHz
DDR5	3200 MHz to 6400 MHz

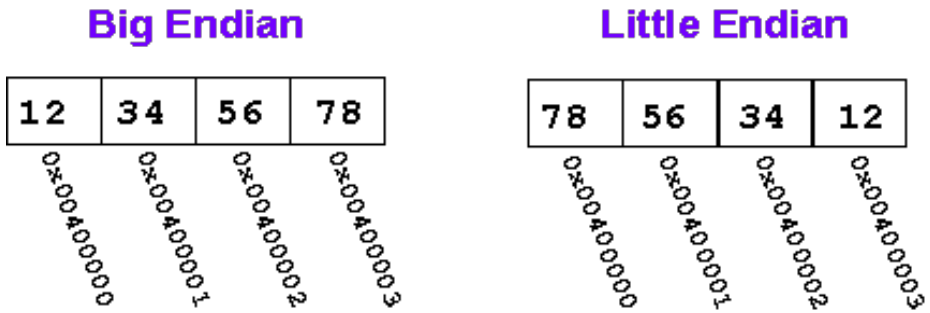
Memory Operands and Memory Access Instructions

- Main memory used for composite data
  - Arrays, structures, dynamic data
- **To apply arithmetic operations**
  - **Load data from memory into registers**
  - **Store result from register to memory**



- Memory is byte addressed
  - Each address identifies an 8-bit byte
- For multi-byte data, RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: Most-significant byte at least address

To store number 12345678



For Register/Immediate Operands and Arithmetic/Logic Instructions

Using ONLY the add, addi, sub and slli instruction to convert the following C statement to the corresponding RISC-V assembly. Assume that the variables `a` and `i` are long integers whose values are already in registers `x1` and `x2`, respectively. You can use other temporary registers such as x10, x11, x12, etc. Register `x0` always contains 0 and cannot be changed.

`a = i + 1;` ➔

```
addi x1, x2, 1;
```

`a = i++;` ➔

```
add  x1, x2, 0   # a = i
addi x2, x2, 1;  # i++
```

`a = ++i;` ➔

```
addi x2, x2, 1;  # i++
add  x1, x2, 0   # a = i
```

We already loaded data into registers!

Memory Access Example #1

- **Using ONLY the add, addi, sub, slli, ld and sd instruction to** convert the following C statements to the corresponding RISC-V assembly. Assume that the values for variables **a** and **i** are in memory. The addresses of the memory location for **a** and **i** are in register **x6** and **x7**. The result should be written back to memory location for **a** and **i**.

```
a = i + 1; //a and i are long int
```

- **To apply arithmetic/logic operations: data need to be loaded from memory into registers**
- **After finishing computation, data in registers can be stored to memory**

load i
add 1
store a

Memory Access Instructions and Memory Operands

- **Using ONLY the add, addi, sub, slli, ld and sd instruction to** convert the following C statements to the corresponding RISC-V assembly. Assume that the values for variables **a** and **i** are in memory. The addresses of the memory location for **a** and **i** are in register **x6** and **x7**. The result should be written back to memory location for **a** and **i**.

```
a = i + 1; //a and i are long int
```

```
ld x10, 0(x7)       # load data of i from memory address 0+[x7] to register x10
addi x11, x10, 1    # increment i (in register x10) by 1 and store the result to register x11
                    # which is the value for a
sd x11, 0(x6)       # store the data of [x11] to memory address 0+[x6],
                    # which is for memory location of variable a
```

- **ld: load instruction, load a double word (8 bytes) from mem to registe**
- **sd: store instruction, store a double word (8 bytes) from register to memory**
- **0(x6), 0(x7) are memory operands: for memory address 0+[x6] or 0+[x**

Load and Store Instructions

```
Format:  ld  rd, offset(rs1)
Example: ld  x9, 64(x22)  // load doubleword to x9
```

- ld: load a doubleword from a memory location whose address is specified as rs1+offset (base+offset, [x22]+64) into register rd (x9)
  - Base should be stored in a register, **offset MUST be a constant number**
  - Address is specified similar to array element, e.g. A[8], for ld, the address is offset(base), e.g. 64(x22)

```
Format:  sd  rs2, offset(rs1)
Example: sd  x9, 96(x22)    // store a doubleword
```

- sd: store a doubleword from register rs2 (x9 in the example) to a memory location whose address is specified as rs1+offset(base+offset, [x22]+96). **Offset MUST be a constant number.**

Memory Access Example #2

- **Using ONLY the add, sub, slli, ld and sd instruction to** convert the following C statements to the corresponding RISC-V assembly. Assume that the values for variables **a** and **i** are in memory. The addresses of the memory location for **a** and **i** are in register **x6** and **x7**. The result should be written back to memory location for **a** and **i**.

```
i++; a = i; //a and i are long int
```

Program with pseudo code first

i = i+1;	**load i**
	add
	store i
a = i;	**load i**
	store a

Memory Access Example #2

- **Using ONLY the add, sub, slli, ld and sd instruction to** convert the following C statements to the corresponding RISC-V assembly. Assume that the values for variables **a** and **i** are in memory. The addresses of the memory location for **a** and **i** are in register **x6** and **x7**. The result should be written back to memory location for **a** and **i**.

```
i++; a = i; //a and i are long int
```

```
load i      ld x10, 0(x7)       # load data of i from memory address 0+[x7] to x10
add         addi x11, x10, 1    # increment i by 1 and store the result to x11,
                                # which is the value for a
store i     sd x11, 0(x7)       # store the new value of i to its memory location (0+[x6])

load i      ld x10, 0(x7)       # load the data of i from memory address 0+[x7] to x10
store a     sd x10, 0(x6)       # store [x10] to memory address 0+[x6],
                                # which is for memory location of variable a
```

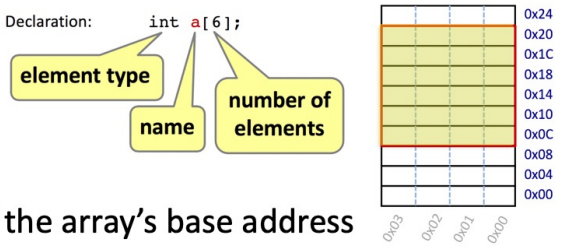
Memory Access Example #3

- C code:

```
double A[N]; //double size is 8 bytes
double h = A[8];
```

  - **h in x21, base address of A in x22**
- Compiled RISC-V code:
  - Element A[8] is 64 bytes offset from A[0]
  - **A[8] right-val ➔ load**
- **Pseudo code:**
  - **load A[8]**
  - **Assign to h (x21)**

- `int a[6];`



- a is the name of the array's base address
  - 0x0C

&a[i]: (char*)a + i * sizeof(int)

```
ld  x9, 64(x22)  // load doubleword A[8]
add x21, x9, x0  // copy from x9 to x21 (h)
```

or just simple as:

```
ld x21, 64(x22)
```

64(x22) is a **memory operand**, in contrast to register operands (x9)

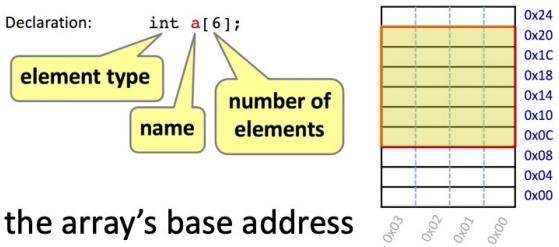
Memory Access Example #4

- C code:

```
double A[N]; //double size is 8 bytes
A[12] = h;
```

  - **h in x21, base address of A in x22**

- int a[6];



- a is the name of the array's base address
  - 0x0C

&a[i]: (char*)a + i * sizeof(int)

- Compiled RISC-V code:
  - Element A[12] is 12*8 bytes offset from A[0]
  - **A[12]: left-val ➔ store instruction**

```
sd  x21, 96(x22)  //store doubleword A[12]
```

96(x22) is a **memory operand**, in contrast to register operands (x9)

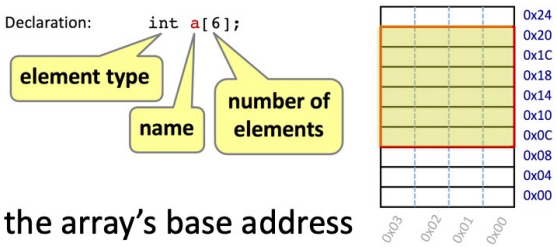
Memory Access Example #5

- C code:

```
double A[N]; //double size is 8 bytes
A[12] = h + A[8];
```

  - h in x21, base address of A in x22

- `int a[6];`



- a is the name of the array's base address
  - 0x0C

&a[i]: (char*)a + i * sizeof(int)

```
ld  x9, 64(x22)     // load A[8]
add x9, x21, x9
sd  x9, 96(x22)     // store A[12]
```

64(x22) and 96(x22) are **memory operands**, in contrast to register operands (x9)

Memory Access Example #6

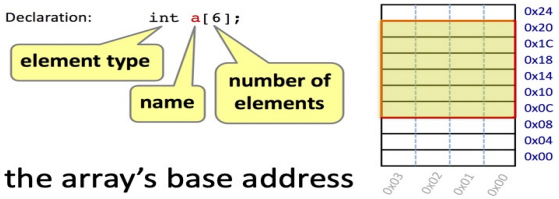
- C code:

```
double A[N]; //double size is 8 bytes
A[12] = h + A[8];
```

  - Memory address for h is in x 11, base address of A in x22

- Compiled RISC-V code:
  - A[8] right-val, A[12]: left-val

- `int a[6];`



- a is the name of the array's base address
  - 0x0C

&a[i]: (char*)a + i * sizeof(int)

```
ld  x21, 0(x11)      //load h
ld  x9, 64(x22)      // load A[8]
add x9, x21, x9
sd  x9, 96(x22)      // store A[12]
```

Load and Store Instructions for Other Data Types

- Load and store are the ONLY two instructions that access memory
- ld/sd: load/store a double word (8 bytes)
  - E.g. double, long int, pointer variable in a 64-bit systems
- lw/sw and lwu: load/store a word (4 bytes)
  - E.g. int, float
  - E.g. unsigned int
- lh/sh and lhu: load/store a half-word (2 bytes)
  - E.g. short
  - E.g. unsigned short
- lbu/sbu: load/store a byte (1 byte)
  - E.g. char
- Load a smaller number to 64-bit register: sign or logic extension

More Load/Store Examples for int A[100]:

- **int A[100]; base address (A, or &A[0]) is in x22, int is 4 bytes**
  - Need to use lw/sw since we are dealing with 4-byte (word) elements
  - lw/sw A[0]: address can be specified as 0(x22).

- A scalar variable (e.g. int f;) can be considered as one-element array (e.g. int f[1]) for load/store
  - lw/sw a variable's (e.g. int f) 32-bit value stored in a specific memory address which is stored in register x6 to register x8
    - lw x8, 0(x6) //offset is 0
    - sw x8, 0(x6)

#7: A[8] = A[10], base is in x22, each element 4 bytes

- lw x6, 40(x22)
- sw x6, 32(x22)

- The context of the terms we use: **base and offset**
  - For array/variable: **base**: &A[0], **offset**: bytes between A[0] and A[i];
  - For lw/sw: **base**: base register, **offset**: the constant in the instr

  - If you have address of A[4] in x9, [x9] can be used for lw/sw as base address
    - lw x5, 0(x9): load A[4]
    - sw x5, 8(x9): store to A[6]
    - sw x5, -8(x9): store to A[2]

#8: a += A[10], base is in x22, each element 4 bytes

- The memory address of variable a is in register x10

+=: ➔ a = a + A[10];

```
lw x5, 0(x10)          //load a to x5
lw x6, 40(x22)         //load A[10] to x6
add x5, x5, x6         // addition
sw x5, 0(x10)          //store back to a
```

#9: A[8] += a, base is in x22, each element 4 bytes

- The memory address of variable a is in register x10

+=: A[8] += a ➔ A[8] = A[8] + a;

```
lw x5, 0(x10)          //load a to x5
lw x6, 32(x22)         //load A[8] to x6
add x6, x5, x6         //addition
sw x6, 32(x22)         //store to A[8]
```

#10: Load/Store Example: Accessing Memory A[i]

int B[N]; // int type, 4 bytes

a = B[i]; //i is a variable reference, not a constant

- Base address for B[] is in x23. i is already loaded in register x5.
  - To load B[i] to a register, e.g. x9, needs to find the address for B[i] in load and store in the form of:
    - base+offset: B+i*4
  - But i*4 is not constant, cannot be the offset for load or store instructions
- **Solution: Calculate the address of B[i] and store in registers as base for LW/SW, and then use 0 as offset in L/S**

```
slliw x6, x5, 2     // x6 now has i*4, slliw is i<<2 (shift left logic)
add x7, x23, x6     // x7 now has the address of B[i].
lw x10, 0(x7)       // load a word from memory location 0+[x7],
                    //which is B[i], into reg x10 which is for a
```

#11: Load/Store Example: Accessing Memory A[i]

int A[N], B[N]; // int type, 4 bytes

A[i] = B[i]; //i is a variable reference, not a constant

- Base address for A and B are in x22 and x23. i is stored in x5
  - **Calculate the address of A[i] and B[i] and store in registers as base for LW/SW, and then use 0 as offset in L/S**

```
slliw x6, x5, 2      // x6 now has i*4, slliw is i<<2 (shift left logic)
add x7, x23, x6   // x7 now has the address of B[i].
lw x9, 0(x7)       // load a word from mem 0+[x7], which is B[i] to x9

add x8, x22, x6  // x8 now has the address of A[i]
sw x9, 0(x8)       // store a word from register x9 to memory 0+[x8]
                        // which is A[i]
```

#12: Load/Store Example: Accessing Memory A[i], A[i-1] and A[i+1]

int A[N], B[N]; // int type, 4 bytes

A[i] = B[i-1] + B[i] + B[i+1]; //i is a variable reference, not a constant

- Base address for A and B are in x22 and x23. i is already loaded in register x5
  - **Calculate the address of A[i] and B[i] and store in registers as base for LW/SW, and then use 0, 4 and -4 as offset in load/store**

```
slliw x6, x5, 2      // x6 now has i*4, slliw is i<<2 (shift left logic)
add x7, x23, x6   // x7 now has the address of B[i].

lw x9, 0(x7)        // load a word from memory 0+[x7], which is B[i], into reg x9
lw x10, 4(x7)     //load a word from memory 4+[x7], which is B[i+1] into x10
lw x11, -4(x7)    //load a word from memory -4+[x7], which is B[i-1] into x11

add x9, x9, x10
add x9, x9, x11

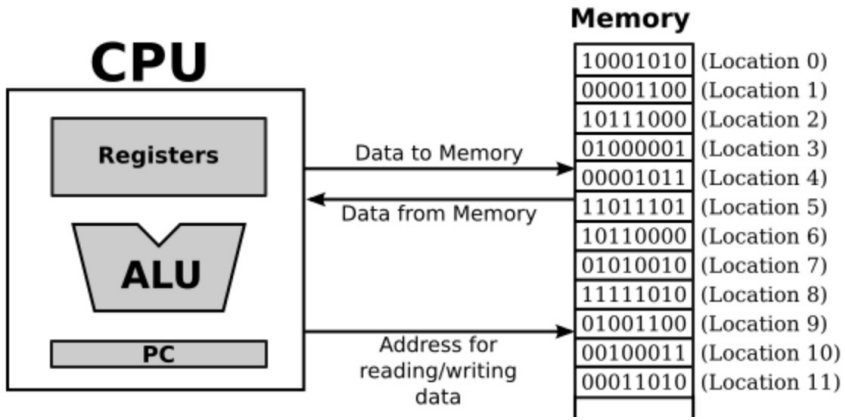
add x8, x22, x6  // x8 now has the address of A[i]
sw x9, 0(x8)       // store a word from register x9 to memory location 0+[x8]
                    // which is A[i]
```

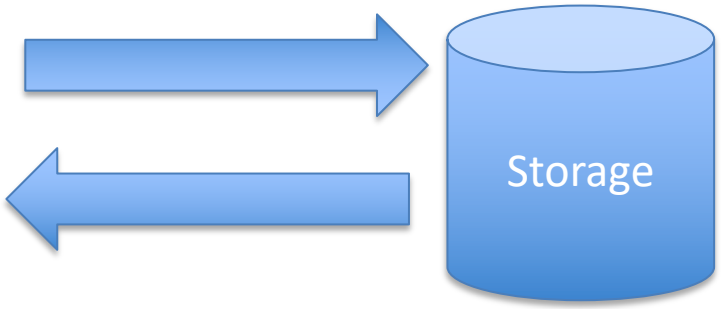
Two Classes of Instructions so Far

- Arithmetic instructions
  - Three operands, could be either register or immediate. Immediate can only be the second source operand.
    - add x10, x5, x6; sub x5, x4, x7
    - addi x10, x5, 10;
- Load and store instructions: Load data from memory to register and store data from register to memory
  - Remember the way of specifying memory address (base+offset)
  - ```
    ld  x9, 64(x22)    // load doubleword
    sd  x9, 96(x22)    // store doubleword
    ```

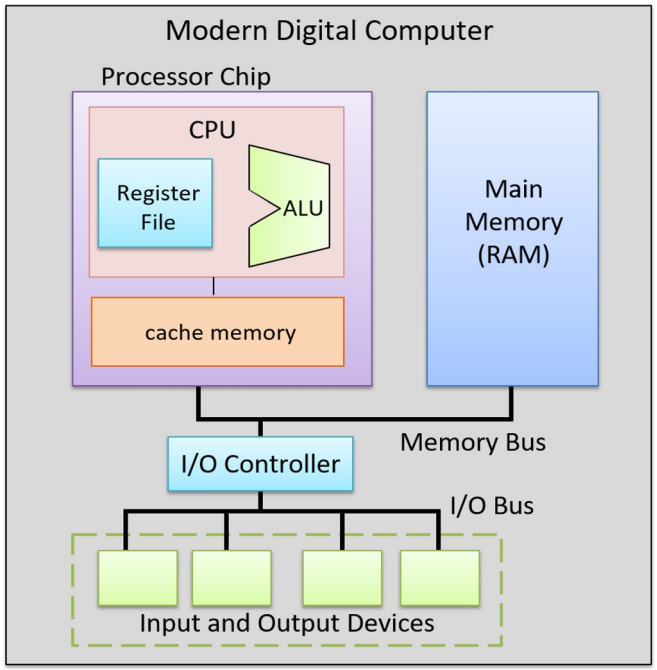
- With these two classes instructions, you can implement the following high-level code, and different ways of combining them
  - `f = (g + h) - (i + j);`
  - `A[12] = h + A[8];`
    ```

Clarifying the Terms





- For ALU to access register
  - Fetch and set, e.g. add x5, x6, x7
    - ALU fetches data from register x6 and x7, performs add, then set x5 with the result
- For move data between mem and register
  - Load and store
- For move data between storage and mem
  - Read and write



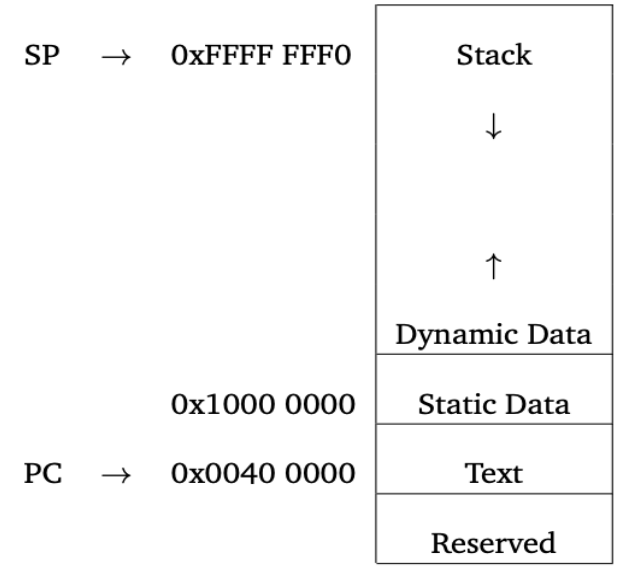
RISC-V Base Integer Instructions

Inst	Name	Description	Note
add	ADD	R[rd] = R[rs1] + R[rs2]	
sub	SUB	R[rd] = R[rs1] - R[rs2]	
xor	XOR	R[rd] = R[rs1] ^ R[rs2]	
or	OR	R[rd] = R[rs1] | R[rs2]	
and	AND	R[rd] = R[rs1] & R[rs2]	
sll	Shift Left Logical	R[rd] = R[rs1] << R[rs2]	
srl	Shift Right Logical	R[rd] = R[rs1] >> R[rs2]	
sra	Shift Right Arith*	R[rd] = R[rs1] >> R[rs2]	sign-extends
slt	Set Less Than	R[rd] = (rs1 < rs2)?1:0	
addi	ADD Immediate	R[rd] = R[rs1] + SE(imm)	
xori	XOR Immediate	R[rd] = R[rs1] ^ SE(imm)	
ori	OR Immediate	R[rd] = R[rs1] | SE(imm)	
andi	AND Immediate	R[rd] = R[rs1] & SE(imm)	
slli	Shift Left Logical Imm	R[rd] = R[rs1] << imm[4:0]	
srli	Shift Right Logical Imm	R[rd] = R[rs1] >> imm[4:0]	
srai	Shift Right Arith Imm	R[rd] = R[rs1] >> imm[4:0]	sign-extends
lw	Load Word	R[rd] = M[R[rs1]+SE(imm)]	
sw	Store Word	M[R[rs1]+SE(imm)] = R[rs2]	
beq	Branch ==	if(rs1 == rs2) PC += SE(imm) << 1	
bne	Branch !=	if(rs1 != rs2) PC += SE(imm) << 1	
blt	Branch <	if(rs1 < rs2) PC += SE(imm) <<1	
bge	Branch >=	if(rs1 >= rs2) PC += SE(imm) <<1	
jal	Jump And Link	R[rd] = PC+4; PC += SE(imm) <<1	
jalr	Jump And Link Reg	R[rd] = PC+4; PC = R[rs1]+ SE(imm)	
lui	Load Upper Imm	R[rd] = SE(imm) << 12	
auipc	Add Upper Imm to PC	R[rd] = PC + (SE(imm) << 12)	
csrrw	CSR read & write	R[rd] = CSRs[csr]; CSRs[csr] = R[rs1]	
csrrs	CSR read & set	R[rd] = CSRs[csr]; CSRs[csr] = CSRs[csr] | R[rs1]	
csrrc	CSR read & clear	R[rd] = CSRs[csr]; CSRs[csr] = CSRs[csr] & ~R[rs1]	
ecall	Environment Call	Transfer control to OS	
ebreak	Environment Break	Transfer control to debugger	

Registers

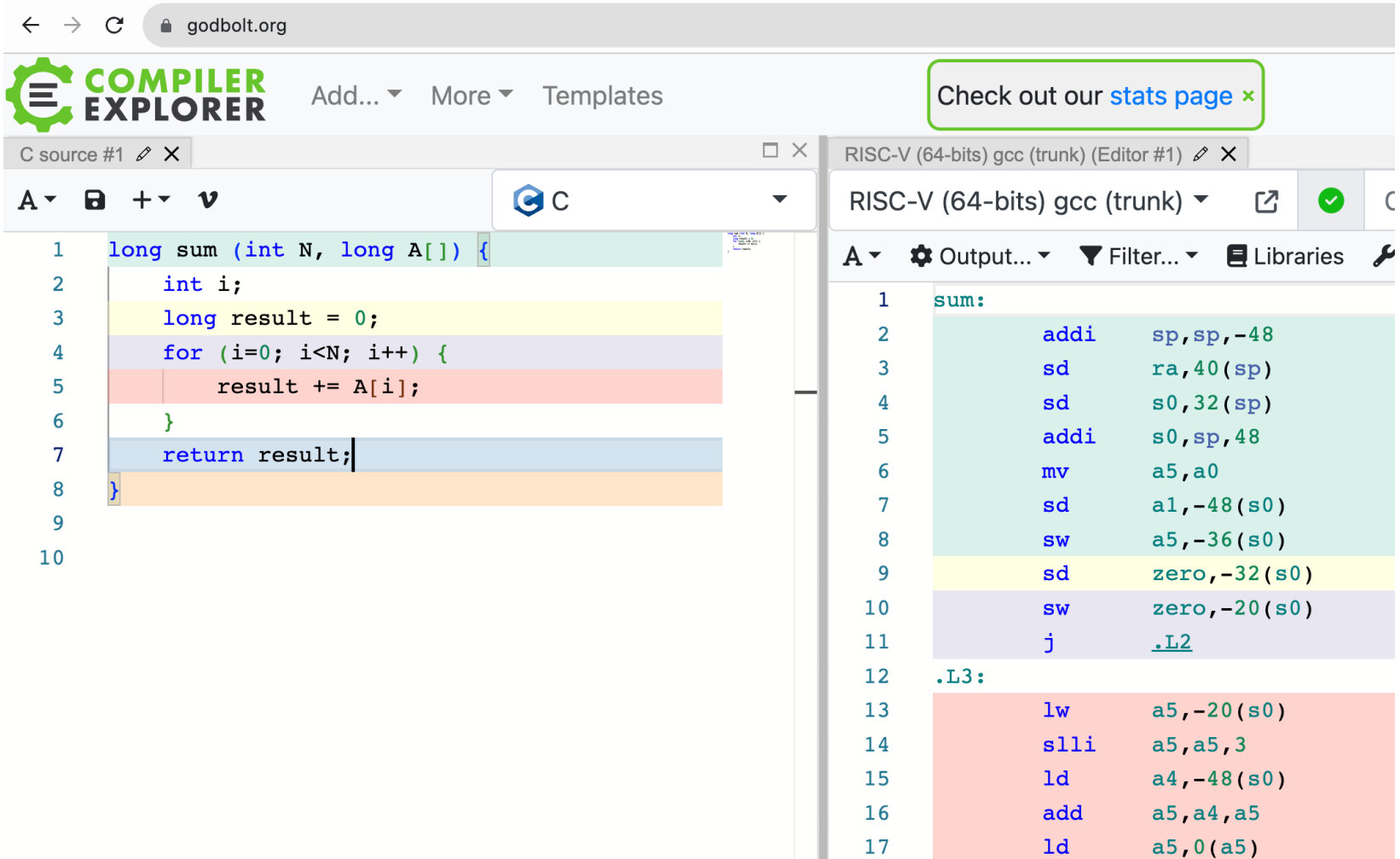
Register	Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x30	t3-t5	Temporaries	Caller
x31			

Memory Allocation



Module 6 Unit 2: Exercise 1

- Recognizing load/store instructions in an assembly program
- From [compiler explorer](), count load/store instructions



Module 6 Unit 2: Exercise 2

Module 06 - Unit 2 Exercise 2: Translate C statements to RISC-V assembly using load/store instructions

ⓘ This is a preview of the draft version of the quiz

Started: Oct 24 at 10:13pm

Quiz Instructions

For all the questions in this quiz, you use ONLY the add, addi, sub, slli, load/store (ld/sd, lw/sw, lhw/shw, lb/sb) instructions to convert the given C statements to the corresponding RISC-V assembly. The use of registers for their values and memory address are pre-assigned as in the following table. You can use the temporary register x20-x31. x0 always contains 0 and cannot be changed. At the beginning of the program, the data for all variables and arrays are in memory. The value of a variable or an array element must be stored back to memory EACH time it is modified. The int type has 4 bytes and the long int type has 8 bytes.

Variables and arrays	int a	int b	long int la	long int lb	int i	int A[]	int B[]
Assigned register for data	x1	x2	x3	x4	x5		
Assigned register for memory addresses	x11	x12	x13	x14	x15	Base address: x16	Base address: x17

Question 1 2 pts

a = b * 3 + i * 4;

Module 6 Unit 2: Lab #1 Write two C programs for Use with RISC-V Assembly Programming

1. Write two C programs for Use with RISC-V Assembly Programming
   - A program to accumulate integers from 1 to 100
   - A program to find the average of 100 integers that are randomly generated

Module 6 Unit 2: Lab #2 Declare and access an array in RISC-V RARS

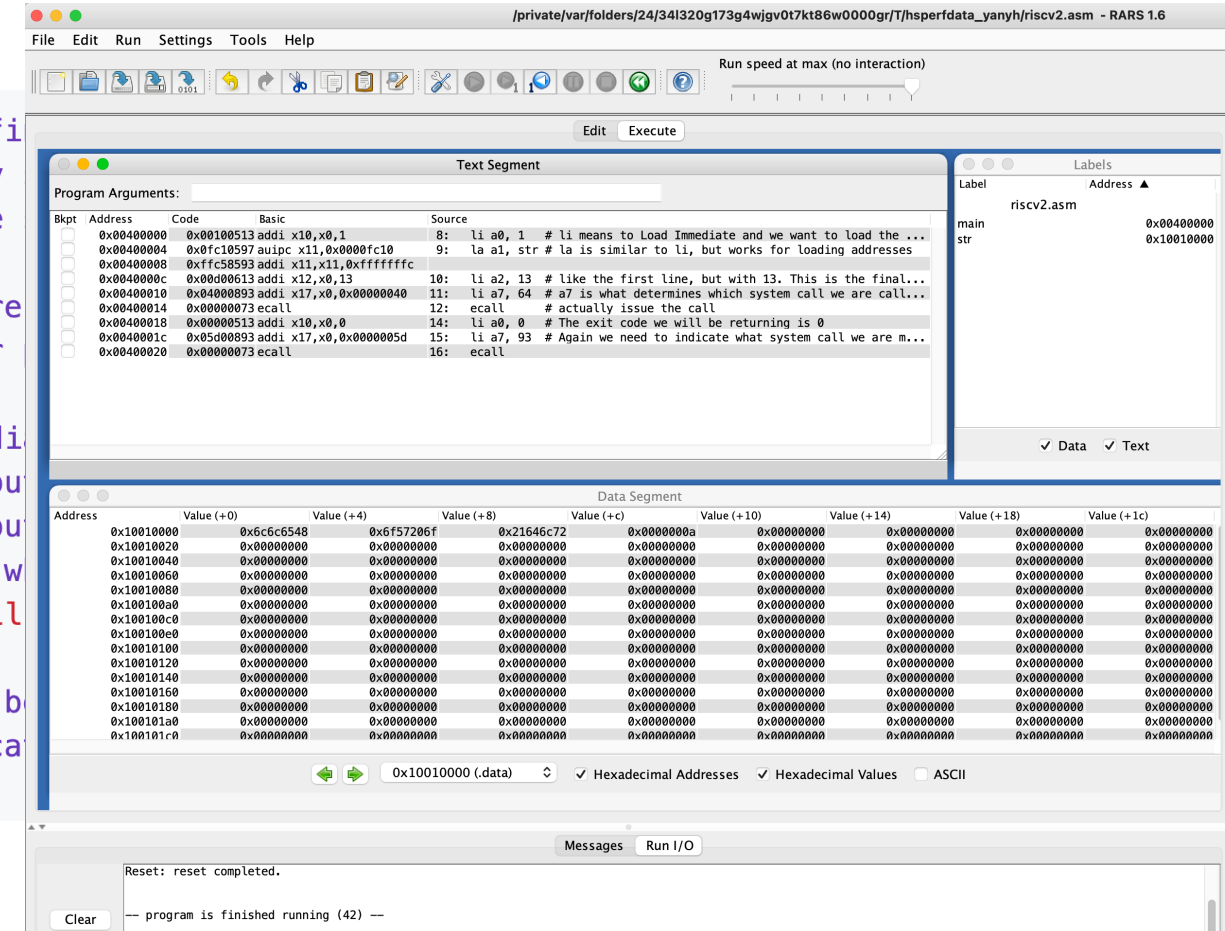
1. Understanding the code structure of an assembly program,
   - Read the document Fundamental of RISC-V Assembly
   - https://github.com/TheThirdOne/rars/wiki/Creating-Hello-World

```
.data # Tell the assembler we are defi
str:  # Label this position in memory
  .string "Hello World!\n" # Copy the

.text # Tell the assembler that we are
main: # Make a label to say where our

  li a0, 1   # li means to Load Immedi
  la a1, str # la is similar to li, bu
  li a2, 13  # like the first line, bu
  li a7, 64  # a7 is what determines w
  ecall      # actually issue the call

  li a0, 0   # The exit code we will b
  li a7, 93  # Again we need to indica
  ecall
```



Code Structure of A Program

```
.globl main    #declare main function
.data          # The .data section of the program is used to
               # reserve memory to use for the variables/arrays
.text          #The .text section is the actual code
main:          #definition of main funct
```

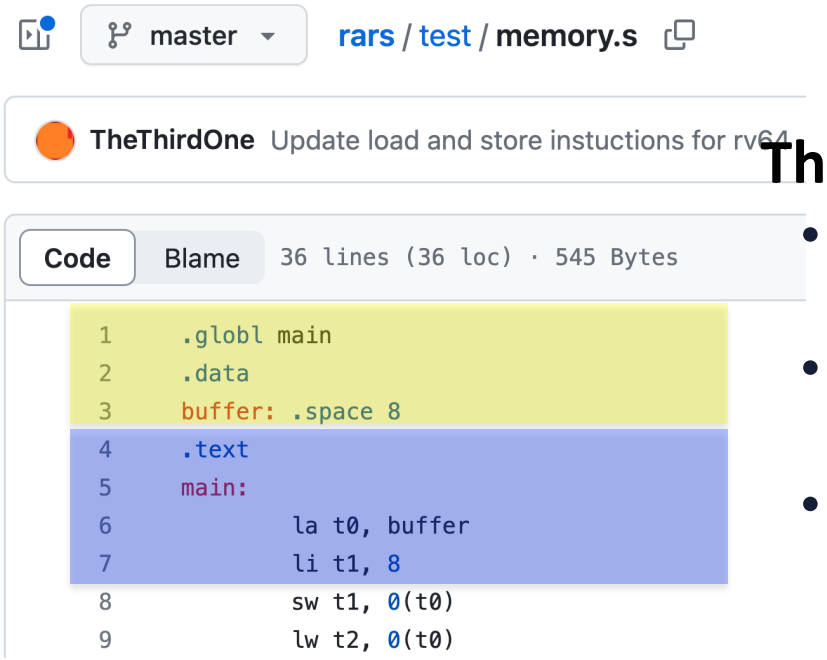
```
.data # Tell the assembler we are defining data not code
str:  # Label this position in memory so it can be referred to in our code
  .string "Hello World!\n" # Copy the string "Hello World!\n" into memory

.text # Tell the assembler that we are writing code (text) now
main: # Make a label to say where our program should start from

  li a0, 1   # li means to Load Immediate and we want to load the value 1 into register
  la a1, str # la is similar to li, but works for loading addresses
  li a2, 13  # like the first line, but with 13. This is the final argument to the syste
  li a7, 64  # a7 is what determines which system call we are calling and we what to cal
  ecall      # actually issue the call
```

Module 6 Unit 2: Lab #2 Declare and access an array in RISC-V RARS

1. Understanding the code structure of an assembly program
   - Read the document Fundamental of RISC-V Assembly
   - https://github.com/TheThirdOne/rars/wiki/Creating-Hello-World
2. Declare and access an array in RISC-V RARS
   - https://github.com/TheThirdOne/rars/blob/master/test/memory.s



Three commonly used pseudo instructions

- **mv x6, x7 //move/copy value from x7 to x6**
- **li x8, 100 //set the value of a register to be an immediate**
- **la x10, label //load address of label to register**

Declare An Array

```
.globl main          #declare main function
.data                #The .data section, for the variables/arrays
    buffer: .space 8   #declare a symbol named "buffer" for
                       # 8 bytes of memory.
        # For a word element, this correspond to "int buffer[2]"
        #If you need to declare an array of 100 elements of int,
        # use "myArray: .space 400
.text                #The .text section of the program is the actual code
main:                #definition of main function
    la t0, buffer    # set register t0 to have the address of the buffe[0]
    li t1, 8           # Set register t1 to have immediate number 8
```

Memory.s file

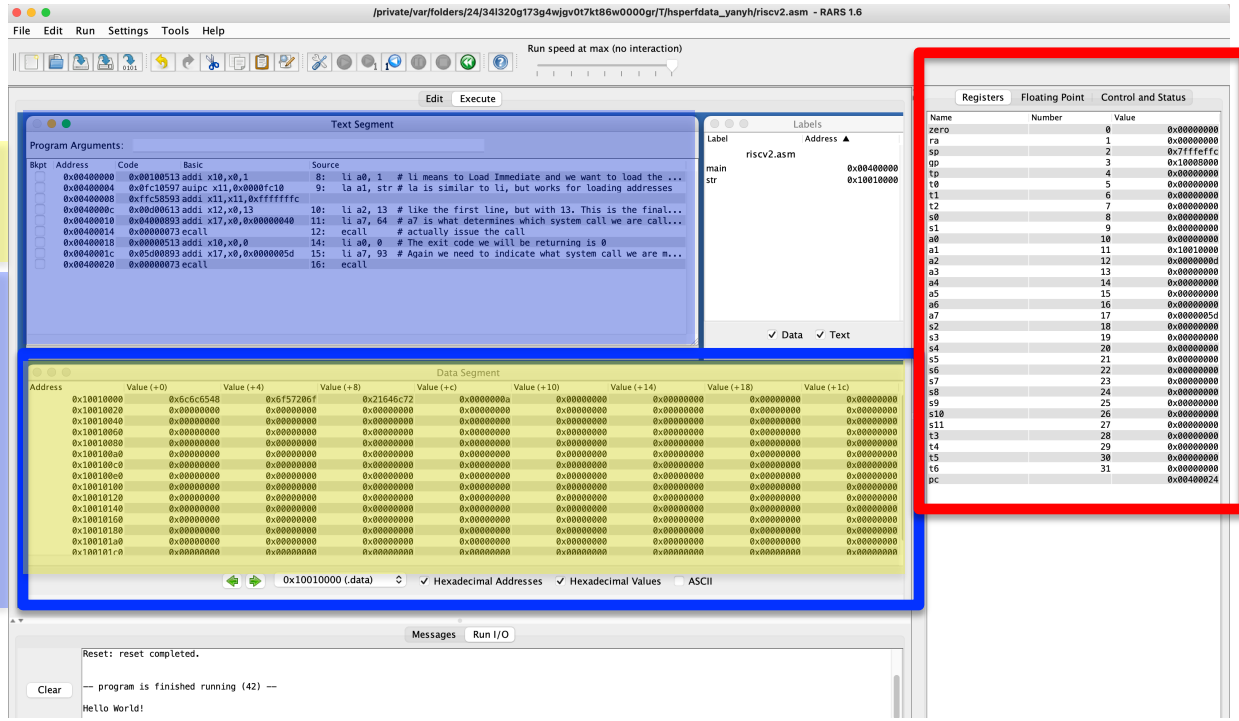
```
.globl main    #declare main function
.data          #The .data section of the program is used to claim memory to use for the variables/arrays of the program
    buffer: .space 8   #declare a symbol named "buffer" for 8 bytes of memory.
                       # For a word element, this corresponds to "int buffer[2]"
                       #This declaration claims 8 bytes of memory.
                      # If you need to declare an array of 100 elements of word, use "myArray: .space 400
.text                 #The .text section of the program is the actual code
main:                 #definition of main function
    la t0, buffer      # set register t0 to have the address of the buffer variable
    li t1, 8           # Set register t1 to have immediate number 8
    sw t1, 0(t0)       # store a word (4 bytes) of what register t1 contains (8) to memory address 0(t0), which is buffer[0]
    lw t2, 0(t0)       # load a word from memory address 0(t0) to register t2, i.e. buffer[0] -> t2
    bne t1, t2, failure  # check whether register t1 and t2 contain the same value or not. If not, branch to failure, else continue the next instruction
    li t3, 56          # set register t3 to have immediate 56
    sw t3, 4(t0)       # store a word of what register t3 contains (56) to memory address 4(t0), which is buffer[1]
    addi t0, t0, 4     # increment register t0 (&buffer) by 4, t0 now contains buffer+4, which is &buffer[1]
    lw t4, 0(t0)       # load a word from memory 0(t0) (&buffer[1]) to register t4
    bne t3, t4, failure  # check whether register t3 and t4 contain the same value or not. If not, branch to failure, else continue.
    lw t5, -4(t0)      # load a word from memory -4(t0) to register t5. -4(t0) address is actually &buffer[0] since register t0 now contains the address of buffer[1]
    bne t5,t1, failure     # check whether register t5 and t1 contain the same value or not. They should both contain 8
    li t1, 0xFF00F007      # set register t1 to have value 0xFF00F007
    sw t1, 0(t0)       # store a word of what register t1 contains to memory address 0(t0) (&buffer[1])
    lb t2, 0(t0)
```

Module 6 Unit 2: Lab #2 Declare and access an array in RISC-V RARS

- Create a main program to declare, initialize and use an array. We will go through this example during the class or lab. The example in the lab is different, but similar.
  - Use the data segment and registers to check the value in memory and register while you debug your program

```
void main () {
    int A[2];
    int a;
    A[0] = 1;
    A[1] = 2;

    a = A[0] + A[1];
    A[0] = a;
    A[1] = a;
}
```



Random Number Generator

```
li a0, 0  # for random number seed
li a1, 100 # range of random number
li a7, 42 # rand code
ecall # call random number generator to
generate a random number stored in a0
```

- Check: https://github.com/TheThirdOne/rars/wiki/Environment-Calls

Module 6 Unit 2: Review Quiz

- Similar questions as in Exercise 2