THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2 Lab 2
RISC-V Assembly Language

1. Introduction

Programming in higher-level languages such as C, Java, and Python is efficient for the programmer. These higher-level languages are translated into assembly language, which is a group of simple instructions. Sometimes performance- or timing-critical sections of code are written in assembly to guarantee specific timing or reduce computation time. This lab shows you how to create a RISC-V assembly language program that you can run on the RVfpgaEL2 System using Catapult. We first give a brief overview of RISC-V assembly and then show how to create and run an assembly program on RVfpgaEL2-NexysA7 (remember that you can also execute the programs on the different Verilator-based simulators and on the RVfpgaEL2-Whisper Instruction Set Simulator). Then we provide exercises for you to practice writing your own RISC-V assembly programs.

2. RISC-V Assembly Language Overview

RISC-V assembly language includes simple instructions that are used to implement higher-level code. For example, some common RISC-V instructions include the `add`, `sub`, and `mul` instructions that add, subtract or multiply two operands.

The basic types of RISC-V instructions are: computational (arithmetic, logical, and shift) instructions, memory operations, and branches/jumps. The most common RISC-V instructions are given in Table 1. Instructions use operands that are located in registers or memory or that are encoded as a constant (i.e., *immediate*). RISC-V includes 32 32-bit registers. Table 2 lists the names of the 32 RISC-V registers. They can be specified by either their name (for example, `zero`, `s0`, `t5`, etc.) or their register number (i.e., `x0`, `x8`, `x30`). Programmers typically use register names which retains some information about the typical purpose of the register. For example, the saved registers, `s0-s11`, are typically used for program variables, while the temporary registers, `t0-t6` are used for temporary calculations. The `zero` register (`x0`) always contains the value 0, as this is a value commonly needed in programs. The other registers have specific uses as well, as shown in Table 2, but in this lab, you need only use the `zero` register and the temporary and saved registers.

Table 1. Common RISC-V assembly instructions

	RISC-V Assembly	Description	Operation
Computational	`add s0, s1, s2`	Add	`s0 = s1 + s2`
	`sub s0, s1, s2`	Subtract	`s0 = s1 - s2`
	`addi t3, t1, -10`	Add immediate	`t3 = t1 - 10`
	`mul t0, t2, t3`	32-bit multiply	`t0 = t2 * t3`
	`div s9, t5, t6`	Division	`t9 = t5 / t6`
	`rem s4, s1, s2`	Remainder	`s4 = s1 % s2`
	`and t0, t1, t2`	Bit-wise AND	`t0 = t1 & t2`
	`or t0, t1, t5`	Bit-wise OR	`t0 = t1 | t5`
	`xor s3, s4, s5`	Bit-wise XOR	`s3 = s4 ^ s5`
	`andi t1, t2, 0xFFB`	Bit-wise AND immediate	`t1 = t2 & 0xFFFFFFFB`
	`ori t0, t1, 0x2C`	Bit-wise OR immediate	`t0 = t1 | 0x2C`
	`xori s3, s4, 0xABC`	Bit-wise XOR immediate	`s3 = s4 ^ 0xFFFFFABC`
	`sll t0, t1, t2`	Shift left logical	`t0 = t1 << t2`
	`srl t0, t1, t5`	Shift right logical	`t0 = t1 >> t5`
	`sra s3, s4, s5`	Shift right arithmetic	`s3 = s4 >>> s5`
	`slli t1, t2, 30`	Shift left logical immediate	`t1 = t2 << 30`

	`srli t0, t1, 5`	Shift right logical immediate	`t0 = t1 >> 5`
	`srai s3, s4, 31`	Shift right arithmetic immediate	`s3 = s4 >>> 31`
Memory	`lw s7, 0x2C(t1)`	Load word	`s7 = memory[t1+0x2C]`
	`lh s5, 0x5A(s3)`	Load half-word	`s5 = SignExt(memory[s3+0x5A]`$_{15:0}$`)`
	`lb s1, -3(t4)`	Load byte	`s1 = SignExt(memory[t4-3]`$_{7:0}$`)`
	`sw t2, 0x7C(t1)`	Store word	`memory[t1+0x7C] = t2`
	`sh t3, 22(s3)`	Store half-word	`memory[s3+22]`$_{15:0}$ `= t3`$_{15:0}$
	`sb t4, 5(s4)`	Store byte	`memory[s4+5]`$_{7:0}$ `= t4`$_{7:0}$
Branch	`beq s1, s2, L1`	Branch if equal	`if (s1==s2), PC = L1`
	`bne t3, t4, Loop`	Branch if not equal	`if (s1!=s2), PC = Loop`
	`blt t4, t5, L3`	Branch if less than	`if (t4 < t5), PC = L3`
	`bge s8, s9, Done`	Branch if greater than or equal	`if (s8>=s9), PC = Done`
Pseudoinstructions	`li s1, 0xABCDEF12`	Load immediate	`s1 = 0xABCDEF12`
	`la s1, A`	Load address	`s1 = Memory address where variable A is stored`
	`nop`	Nop	`no operation`
	`mv s3, s7`	Move	`s3 = s7`
	`not t1, t2`	Not (Invert)	`t1 = ~t2`
	`neg s1, s3`	Negate	`s1 = -s3`
	`j Label`	Jump	`PC = Label`
	`jal L7`	Jump and link	`PC = L7; ra = PC + 4`
	`jr s1`	Jump register	`PC = s1`

In addition to actual RISC-V instructions, RISC-V includes pseudoinstructions (as shown in the bottom of Table 1), instructions that are not really RISC-V instructions but that are commonly used by programmers. Pseudoinstructions are implemented using one or more real RISC-V instruction. For example, the move pseudoinstruction (`mv s1, s2`) copies the contents of `s2` and puts it in `s1`. It is implemented using the real RISC-V instruction: `addi s1, s2, 0`.

Table 2. RISC-V registers

Name	Register Number	Use
zero	**x0**	Constant value 0
ra	**x1**	Return address
sp	**x2**	Stack pointer
gp	**x3**	Global pointer
tp	**x4**	Thread pointer
t0-2	**x5-7**	Temporary variables
s0/fp	**x8**	Saved register / Frame pointer
s1	**x9**	Saved register
a0-1	**x10-11**	Function arguments / Return values
a2-7	**x12-17**	Function arguments
s2-11	**x18-27**	Saved registers
t3-6	**x28-31**	Temporary variables

The commands that start with a period are assembler directives. They are commands to the assembler rather than code to be translated by it. They tell the assembler where to place code and data, specify text and data constants for use in the program, and so forth. Table 3 shows the main assembler directives of RISC-V (*The RISC-V Reader: An Open Architecture Atlas, Patterson & Waterman*, © 2017).

Table 3. RISC-V main directives

Directive	Description
.text	Subsequent items are stored in the `text` section (machine code).
.data	Subsequent items are stored in the `data` section (global variables).
.bss	Subsequent items are stored in the `bss` section (global variables initialized to 0).
.section .foo	Subsequent items are stored in the section named `.foo`.
.align n	Align the next datum on a 2^n-byte boundary. For example, `.align 2` aligns the next value on a 4-byte (word) boundary.
.balign n	Align the next datum on an n-byte boundary. For example, `.balign 4` aligns the next value on a 4-byte (word) boundary.
.globl sym	Declare that label `sym` is global and may be referenced from other files
.string "str"	Store the string `str` in memory and null-terminate it.
.word w1,…,wn	Store the `n` 32-bit quantities in successive memory words.
.byte b1,…,bn	Store the `n` 8-bit quantities in successive bytes of memory.
.space	Reserve memory space to store variables without an initial value. It is commonly used to declare the output variables, when they are not also serving as input variables. The space we want to reserve must always be expressed as a number of bytes. For example, the directive `RES: .space 4` reserves four bytes (i.e. one word) that are not initialized.
.equ name,constant	Define symbol `name` with value `constant`. For example, `.equ N,12`, defines symbol N with the value 12.
.end	The assembler will conclude its work when it reaches the directive `.end`. Any text located after this directive will be ignored.

The examples below (see Table 4 - Table 5) show how to code some common high-level constructs in RISC-V assembly. Notice that branch instructions (`beq`, `bne`, `blt`, and `bge`) conditionally jump to a label; whereas the jump instruction (`j`) unconditionally jumps to a label. Single-line comments are indicated by `//` in C and `#` in RISC-V assembly.

In the first example (that implements an if/else statement, see Table 4), notice that the C code and RISC-V assembly code check for the opposite cases: the C code checks for less than (<) and the assembly equivalent checks for greater than or equal (>=).

Table 4. RISC-V Assembly Example 1: if/else statement

// C Code	**# RISC-V Assembly**
`int a, b, c;`	`# s0 = a, s1 = b, s2 = c`

```
if (a < b)
  c = 5;
else
  c = a + b;
```

```
    bge  s0, s1, L1   # if (a >= b) goto L1
    addi s2, zero, 5  # c = 5
    j    L2           # jump over else block
L1: add  s2, s0, s1   # c = a + b
L2:
```

In the second example (manipulating an array of integers, see Table 5), the RISC-V assembly code uses temporary registers (`t0-t3`) to hold temporary values, such as the constant 100 and the base address of the data array. After initializing the registers in the first three instructions, the RISC-V assembly code checks for `i` >= 100 using the `bge` (branch if greater than or equal to) instruction; again, this is the opposite case from the C code. If that condition is met, the for loop is done. If the branch is **not** taken, `i` is less than 100 and the remaining code is executed. Notice that the index `i` is multiplied by 4 (using the `slli t2, s0, 2` instruction) before it is added to the base address because integers (32-bit two's complement numbers) occupy 4 bytes of memory. In RISC-V, memory is byte-addressable (i.e., each byte has its own address). If the array had been an array of characters (i.e., `char data[100];`), then each array element would only occupy a byte and `i` could be added directly to the base address to form the address of array index `i`, i.e., `array[i]`. After the array element is read, decremented by ten, and written (via the `lw`, `addi`, and `sw` instructions, respectively), the array index `i` (i.e., `s0`) is incremented and the program jumps back to the beginning of the for loop (using the `j L5` instruction).

Table 5. RISC-V Assembly Example 2: manipulating an array of integers

```
// C Code
int i;
int data[100];

for (i=0; i<100; i++)

  array[i] = array[i]-10;
```

```
# RISC-V Assembly
# s0 = i, t1 = base address of data (assumed
# to be at 0x300)

    addi s0, zero, 0   # i = 0
    addi t0, zero, 100 # t0 = 100
    li   t1, 0x300     # base address of array
L5: bge  s0, t0, L7    # if (i>=100) exit loop
    slli t2, s0, 2     # t2 = i*4
    add  t2, t1, t2    # address of data[i]
    lw   t3, 0(t2)     # t3 = array[i]
    addi t3, t3, -10   # t3 = array[i]-10
    sw   t3, 0(t2)     # array[i] = array[i]-10
    addi s0, s0, 1     # i++
    j    L5            # loop
L7:
```

For more details about the RISC-V assembly language, refer to the RISC-V Instruction Set Manual (available here: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf) or a textbook such as *Digital Design and Computer Architecture: RISC-V Edition*, Harris & Harris, © Morgan Kaufmann 2021 or *The RISC-V Reader: An Open Architecture Atlas*, Patterson & Waterman, © 2017.

3. Writing a RISC-V Assembly Program for RVfpgaEL2

Now you are ready to explore and practice writing RISC-V assembly programs on your own. Before you write your own programs, follow these steps to setup a Catapult project and create and run an assembly program on RVfpgaEL2-NexysA7 (remember that you can also run these programs in simulation, using Verilator or Whisper):

1. Create an RVfpgaEL2 project
2. Write a RISC-V assembly language program
3. Download RVfpgaEL2-NexysA7 onto the Nexys A7 Board and compile, download, and run an assembly program on RVfpgaEL2-NexysA7
4. Compile, download, and run an assembly program on RVfpgaEL2-ViDBo

Step 1. Create an RVfpgaEL2 project

In folder *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab02*, create a directory called *Project2*.

As in Lab 1, at *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab02/ProjectSources* you will find the sources for the project. The only difference with respect to the sources used in the first lab is that file *CMakeLists.txt* defines an Assembly project instead of a C project and that the source for the project has a ***.S*** extension (instead of ***.c***):

```
project(Test ASM)

${CMAKE_CURRENT_SOURCE_DIR}/src/Test.S
```

You can follow the same steps described in Step 1 of Lab 1 for creating the project.

Step 2. Write a RISC-V assembly language program

Now you will write a RISC-V assembly program. Start by creating directory **src** in folder *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab02/Project2*.

Then, click on File → New File (see Figure 1).

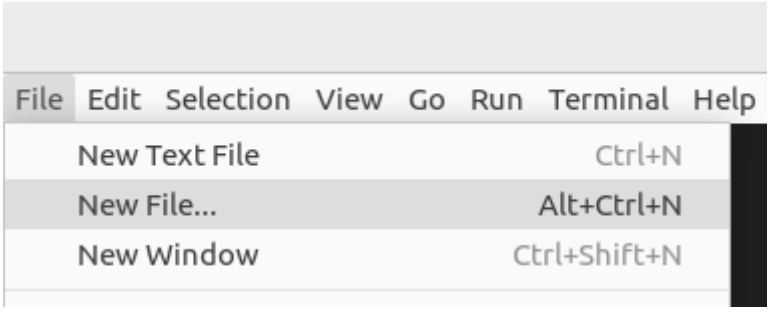

Figure 1. Add file to project

A menu window will open (see Figure 2). Select “Text File”.

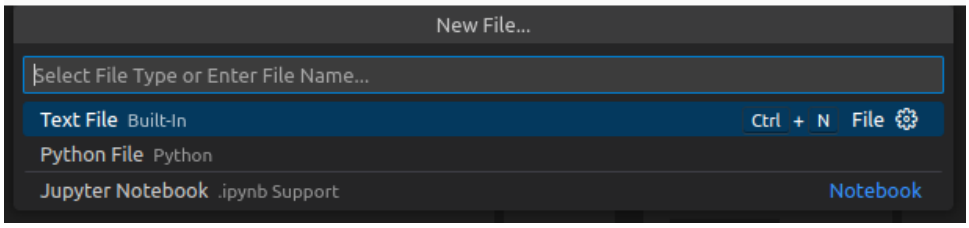

Figure 2. Select "Text File"

A blank file will open where you will be asked some questions (see Figure 3).

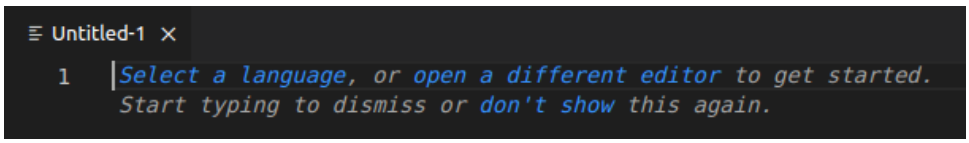

Figure 3. Blank file

Click on "Select a language" and, in this case, choose Plain Text (see Figure 4).

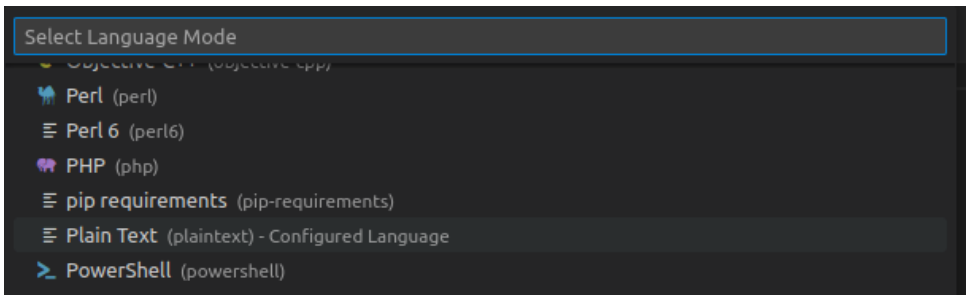

Figure 4. Select a language – Plain Text

Type (or copy/paste) the following RISC-V assembly program into that window (see Figure 5).

```
// memory-mapped I/O addresses
# GPIO_SWs   = 0x80001400
# GPIO_LEDs  = 0x80001404
# GPIO_INOUT = 0x80001408

.globl main

main:
  li t0, 0x80001400   # base address of GPIO memory-mapped registers
  li t1, 0xFFFF       # set direction of GPIOs
                      # upper half = switches (inputs)  (=0)
                      # lower half = outputs (LEDs)     (=1)
  sw t1, 8(t0)        # GPIO_INOUT = 0xFFFF
repeat:
  lw   t1, 0(t0)      # read switches: t1 = GPIO_SWs
  srli t1, t1, 16     # shift val to the right by 16 bits
  sw   t1, 4(t0)      # write value to LEDs: GPIO_LEDs = t1
  j    repeat         # repeat loop
```

This program is also available in the following file for your convenience:

[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab02/ProjectSources/ReadSwitches.S

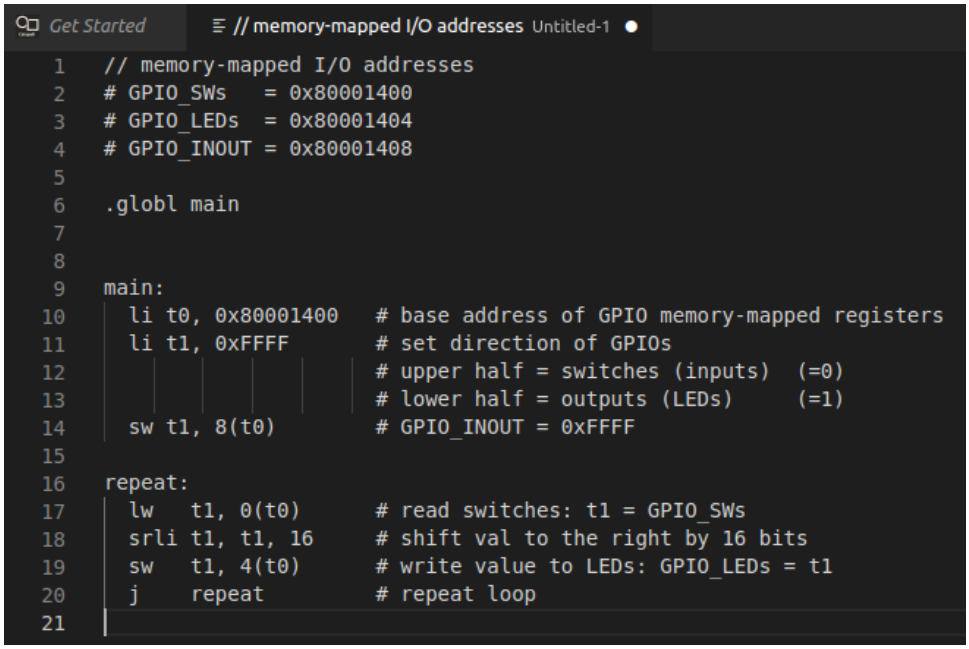

Figure 5. Enter RISC-V assembly program

The assembly code must contain the following lines at the beginning of the code:

```
.globl main
main:
```

The `.globl` assembler directive makes the label visible in all linked files. The boot code will configure the system and jump to this label (*main*). The debugger will set a temporary breakpoint there when it begins.

This RISC-V assembly program is the same example program as in Lab 1, but this time written in RISC-V assembly. It sets the direction of the inputs and outputs of the general-purpose I/O (GPIO) and then repeatedly reads the value of the switches and writes that value to the LEDs.

After entering the program into the pane, press Ctrl-s to save the file. Name it Test.S and save it to the **src** folder of the Project2 directory (see Figure 6).

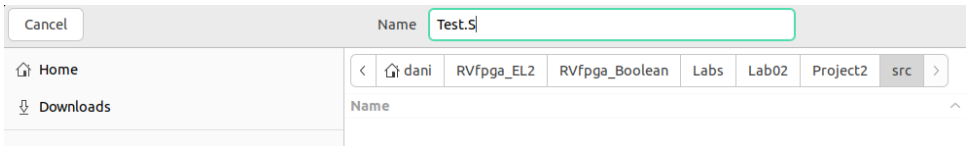

Figure 6. Save file as ReadSwitches.S

Step 3. Download RVfpgaEL2-NexysA7 onto the Nexys A7 Board and compile, download, and run a C program on RVfpgaEL2-NexysA7

You will now run the example program on RVfpgaEL2-NexysA7. For that purpose, download the bitstream onto the board, compile the program and launch it as explained in detail in the Getting Started Guide. You can do as many tests as you wish: run step-by-step, inspect memory, the peripherals, and the registers, view the disassembly code, etc.

Step 4. Compile, download, and run a C program on RVfpgaEL2-ViDBo

You can also test execution on RVfpgaEL2-ViDBo. For that purpose, compile the program and launch it as explained in detail in the GSG. Confirm that when a switch changes its state, the corresponding LED also changes its state.

4. Exercises

Now create your own RISC-V assembly programs by completing the same exercises as in Lab 1, but this time in RISC-V assembly instead of C. The exercise descriptions are repeated below for your convenience.

Remember that if you leave the Nexys A7 board connected to your computer and powered on, you do not need to reload RVfpgaEL2-NexysA7 onto the board between running different programs. However, if you turn off the Nexys A7 board, you will need to reload RVfpgaEL2-NexysA7 onto the board using Catapult.

Remember as well that you can run these programs in simulation, using the different Verilator-based simulators and the RVfpgaEL2-Whisper Instruction Set Simulator.

Exercise 1. Write a RISC-V assembly program that flashes the value of the switches onto the LEDs. The value should pulse on and off slow enough that a person can view the flashing. Name the program **FlashSwitchesToLEDs.S**.

Exercise 2. Write a RISC-V assembly program that displays the inverse value of the switches on the LEDs. For example, if the switches are (in binary): 0101010101010101, then the LEDs should display: 1010101010101010; if the switches are: 1111000011110000, then the LEDs should display: 0000111100001111; and so on. Name the program **DisplayInverse.S**.

Exercise 3. Write a RISC-V assembly program that scrolls increasing numbers of lit LEDs back and forth until all of the LEDs are lit. Then the pattern should repeat. Name the program **ScrollLEDs.S**.

The program should cause the following to occur:

1. First, one lit LED should scroll from right to left.
2. Once it reaches the left-most LED, two lit LEDs should scroll from left to right and then right to left.
3. Once those two LEDs reach the left-most LED, three lit LEDs should scroll from left to right then right to left.
4. Then four lit LEDs should scroll.

5. And so on, until all the LEDs are lit.
6. Then the pattern should repeat.

Exercise 4. Write a RISC-V assembly program that displays the unsigned 4-bit sum of the 4 least significant bits of the switches and the 4 most significant bits of the switches. Display the result on the 4 least significant (right-most) bits of the LEDs. Name the program **4bitAdd.S**. The fifth bit of the LEDs should light up when unsigned overflow occurs (that is when the carry out is 1).

Exercise 5. Write a RISC-V assembly program that finds the *greatest common divisor* of two numbers, *a* and *b*, according to the Euclidean algorithm. The values *a* and *b* should be statically defined variables in the program. Name the program **GCD.S**. Here is some additional information about the Euclidean algorithm: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm. You can also simply google "Euclidean algorithm".

Exercise 6. Write a RISC-V assembly program that computes the first 12 numbers in the Fibonacci sequence, and stores the result in a finite vector (i.e. array), *V*, of length 12. This infinite sequence of Fibonacci numbers is defined as:

`V(0)=0, V(1)=1, V(i)=V(i-1)+V(i-2)` (where `i=0,1,2...`)

In words, the Fibonacci number corresponding to element i is the sum of the two previous Fibonacci numbers in the series. Table 6 shows the Fibonacci numbers for i = 0 to 8.

Table 6. Fibonacci series

i	0	1	2	3	4	5	6	7	8
V	0	1	1	2	3	5	8	13	21

The dimension of the vector, *N*, must be defined in the program as a constant. Name the program **Fibonacci.S**.

Exercise 7. Given an *N*-element vector (i.e., array), *A*, generate another vector, *B*, such that *B* only contains those elements of *A* that are even numbers greater than 0. For example: suppose *N* = 12 and *A* = [0,1,2,7,-8,4,5,12,11,-2,6,3], then B would be: *B* = [2,4,12,6]. Name the program **EvenPositiveNumbers.S**.

Exercise 8. Given two *N*-element vectors (i.e., arrays), *A* and *B*, create another vector, *C*, defined as:

`C(i) = |A[i] + B[N-i-1]|, i = 0,..,N-1.`

Write a program in RISC-V assembly that computes the new vector. Use 12-element arrays in your program. Name the program **AddVectors.S**.

Exercise 9. Implement the bubble sort algorithm in RISC-V assembly. This algorithm sorts the components of a vector in ascending order by means of the following procedure:

1. Traverse the vector repeatedly until done.
2. Interchanging any pair of adjacent components if `V(i) > V(i+1)`.
3. The algorithm stops when every pair of consecutive components is in order.

Use 12-element arrays to test your program. Name the program **BubbleSort.S**.

Exercise 10. Write a program in RISC-V assembly that computes the factorial of a given non-negative number, *n*, by means of iterative multiplications. While you should test your program for multiple values of n, your final submission should be for *n* = 7. *n* should be a variable that is statically defined within the program. Name the program **Factorial.S**.