Digital Design Verification

Lab Manual # 12 – Introduction to RISC-V Assembly & Custom Processor Design

Release: 2.0

Date: 8-Aug-2025

NUST Chip Design Centre (NCDC), Islamabad, Pakistan

Revision History

Revision Number	Revision Date	Revision By	Nature of Revision	Approved By
1.0	21/05/2024	Ali Aqdas	Complete Manual	-
2.0	8/12/2025	Umer Farooq	Content Consolidation	-

Contents

Objective

The objective of this lab is to

- Understand the fundamentals of Assembly Language (RISC-V) and its components.
- Learn how to compile high-level programs to Assembly and identify assembly components.
- Implement and verify the building blocks of a custom microprocessor in SystemVerilog.
- Integrate the processor design with RISC-V assembly programs for testing.

Tools and Technologies

- GNU Toolchain **(RISC-V & x86)**
- Visual Studio Code
- Linux Environment
- SystemVerilog
- Simulator **(e.g., Vivado)**

PART 1 -- Introduction to Assembly Language

Instructions for Lab Tasks

Make sure that the lab directory has no spaces in the path. In this lab, the students are required to submit three files one for each task.

Introduction

In today's age, modern computing applications like the one you are using to read this text are designed in high-level languages such as C/C++, Python and Java. High-Level Languages offer an abstract layer hiding registers and individual instructions by introducing high level constructs which offer ease and flexibility to the programmer. Although the languages offer a high amount of flexibility, a computer does not understand them.

In order for computer to use the program, it needs to be converted first to **assembly** which can access individual registers within the processor. Each assembly instruction is a single instruction of the processor and takes a fixed number of cycles. Although this fine-grained control of the processor is highly effective for time and space critical applications, the programmer must understand the design of processor architecture.

After an instruction is converted to the assembly language, each assembly line directly translates to machine code, which is a stream of 0 s and 1 s that directly controls the hardware. The entire flow is known as the compilation and assembling. An overview of the entire flow is presented in **Figure 1**.

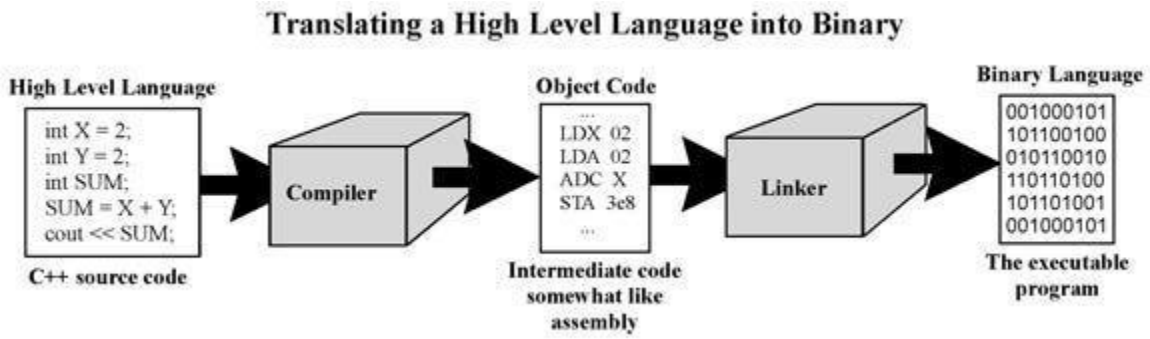


Figure 1: Translating a High Level Language to Binary (Medium [1])

Compiling an Application

Today, compilers are available for several processor architectures such as

- ARM
- X86
- RISC-V
- Espressif (ESP)

GNU Toolchain is one of the most popular compiler toolchains, which comes pre-installed in the Linux Operating System for X86 Architecture. It can be downloaded for several other processor architectures, both in prebuilt form and source files which can be build on your computers.

Lab Task 1: Compiling a C Program

In the first lab task, we are required to compile the `"hello.c"` program to assembly language. In the program, replace `<Your-Name>` with your name and compile the program to assembly language.

a) First compile the application into x86 assembly language using the following command in `src` directory

```
$ gcc ./hello.c -o ./hello.x86.asm -S
```

The assembly program will be written into hello.x86.asm file.

In the above command, `gcc` is the compiler, `-o` flag specifies the output file and `-S` flag specifies that compiler should only compile and not assemble.

b) Now, compile the same program with RISC-V GNU Toolchain to generate RISC-V Assembly and save it in `hello.rv32.asm`. The RISC-V GCC Compiler has the following binary `riscv32-unknown-elf-gcc.`

Submit both the assembly files for lab assessment.

Components of RISC-V Assembly Language

The assembly language comprises of various components such as assembler directives, labels and pseudo-instructions. A complete detail of the RISC-V Assembly Language can be found here. The document lists key components, a brief description of which is presented in the following subsections.

Assembler Directives

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. A list of assembler directives is available in the The RISC-V Reader. The most common assembler directives which we will be frequently using in the next labs are

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).
`.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.

Table 1: Assembler Directives (Courtesy of RVFPGA)

Labels
Labels are used to identify different portions of assembly code, to which the user can jump and different variables which the user can access in assembly language. The below textbox shows the example of labels in `.text` section. The labels are highlighted.

Instructions and Pseudoinstructions
The basic types of RISC-V instructions are: computational (arithmetic, logical, and shift) instructions, memory operations, and branches/jumps. Instructions use operands that are located in registers or memory or that are encoded as a constant (i.e., *immediate*).

Pseudoinstructions, on the other hand are implemented using one or more real RISC-V instructions. For example, the move pseudoinstruction (`mv s1, s2`) copies the contents of `s2` and puts it in `s1`. Figure 3.3 in The RISC-V Reader presents a list of pseudoinstructions and the corresponding base instructions.

```
_start:

andi t0, t0, 0          # clear register t0
andi t1, t1, 0          # clear register t1
andi t2, t2, 0          # clear register t2
andi t3, t3, 0          # clear register t3
andi t4, t4, 0          # clear register t4
andi t5, t5, 0          # clear register t5
li t0, 2                # t0 = 2
li t3, -2               # t3 = -2
slt t1, t0, zero        # t1 = t0 < 0 ? 1 : 0
beq t1, zero, ElseIf    # go to ElseIf if t1 = 0
j EndIf                 # end If statement
ElseIf:
sgt t4, t3, zero        # t4 = t3 > 0 ? 1 : 0
beq t4, zero, Else      # go to Else if t4 = 0
j EndIf                 # end Else statement
Else:
seqz t5, t4, zero       # t5 = t4 == 0 ? 1 : 0
EndIf:
j EndIf           # end If-ElseIf-Else statement
```

Lab Task 2: Identifying Components of Assembly Language

List the following in the RISC-V Assembly of `"hello.c"` generated in Task 1.

1. Unique Assembler Directives
2. Unique Base Instructions
3. Unique Labels
4. Unique Pseudoinstruction and their corresponding base instructions.

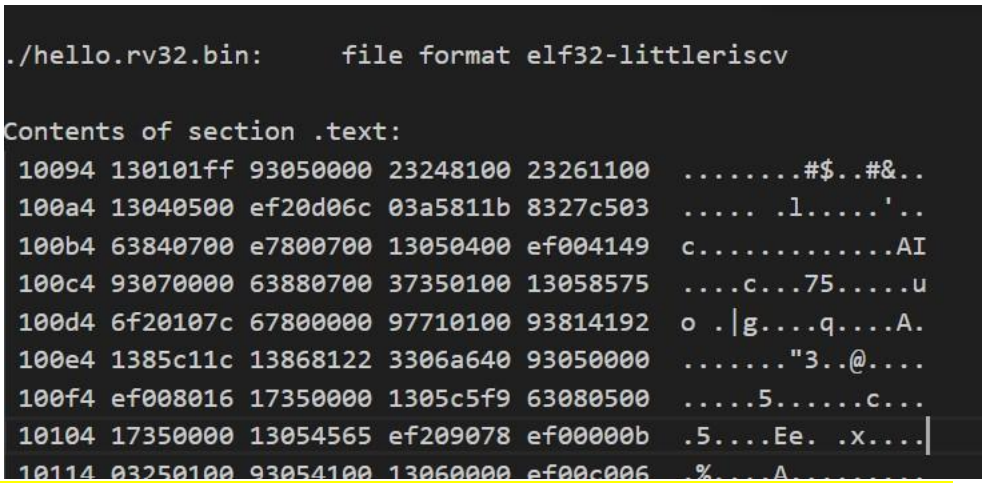
   You are required to identify unique items, for example if there are multiple **add** instructions you may list them only once in your answers.

Lab Task 3:

Now, fully compile `"hello.c"` program to generate a binary stream. This can be done by removing the `-s` flag from the command given in Task 1. The stream is made of binary characters (displayed as ASCII in VS Code). To make the characters legible run the following command to display a hexadecimal dump of the file in the file `"hello.rv32.hex"`

```
$ riscv32-unknown-elf-objdump -s ./hello.rv32.bin > hello.rv32.hex
```

In the new file, different sections are visible such as the one given below. The first column displays the addresses in the memory locations and the next four columns show sixteen bytes of data stored in these addresses. The final column displays the data in ASCII Format.



Identify your name (which was added in the first task) in the dump file and attach a screenshot of the window displaying all the columns of the data in that line.

PART 2 – Introduction to Processor Design

Instructions for Lab Tasks

The submitted tasks must have modular design i.e. each lab task must be written in a separate module which is instantiated in the top level. It must have the following hierarchy with folder name as the name of student without spaces and each file name must be exactly as listed in the table below.

./student_name/
- register_file.sv
- alu.sv
- instruction_memory.sv
- program_counter.sv
- top.sv

support_file s/
- fib_ im.mem
- fib_rf.mem

The design must be parameterized (parameters are listed in the following sections) and the parameters must be passed from the top level of the design i.e. **top.sv**. The names of each of these parameters must be as follows.

Parameter	Name (System Verilog)	Default Value
Instruction Memory Depth	IMEM_DEPTH	4 Words
Register File Width	REGF_WIDTH	16-Bit
ALU Width	ALU_WIDTH	16-Bit
Program Counter Maximum Value	PROG_VALUE	3

Introduction.

To design a simple microprocessor, the fundamental building block is the instruction set architecture (ISA). An ISA essentially defines a language or set of instructions that the processor understands and can execute. It's like a contract between the software (programs you run) and the hardware (the processor itself).

Some of the key components of any processor are

Register Files

They hold a limited amount of data but offer incredibly fast access times. The processor utilizes registers to store temporary data crucial for its current operations.

Control Unit

Consider the CU the conductor of the processor's orchestra. It retrieves instructions from memory, decodes them to understand the task at hand, and then directs other processor components (like the ALU) to execute those instructions.

Arithmetic Logic Unit

The ALU is the engine powering the processor's calculations. It performs all the mathematical (arithmetic) and logical operations based on the instructions received from the control unit. This encompasses operations like addition, subtraction, comparisons (greater than, less than), and bitwise operations (AND, OR, NOT).

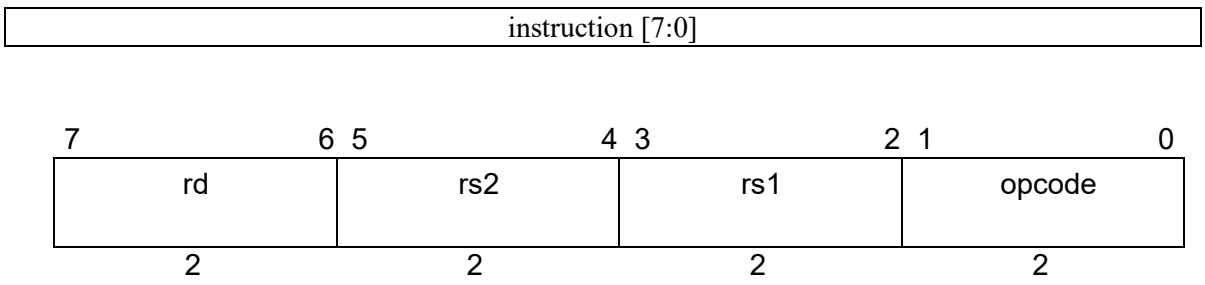
Memory

Although not directly part of the processor, memory (like RAM or storage) plays a critical role. The processor retrieves instructions and data from memory, performs operations on them in the ALU and registers, and then may write the results back to memory.

The Foundation of Our Processor: The Instruction Set Architecture (ISA)

Without getting any further, lets step towards the design of our processor which is based on a custom instruction set architecture.

The core features a very basic instruction set architecture. Each instruction is 6 − bit wide and features an opcode and two operands.



Our processor feature four main arithmetic operations

$$ADD, SUB, AND, OR$$

opcode field allows the user to select the outputs of one of the four operations supported by the processor as follows

opcode	Operation
00	ADD
01	SUB
10	AND
11	OR

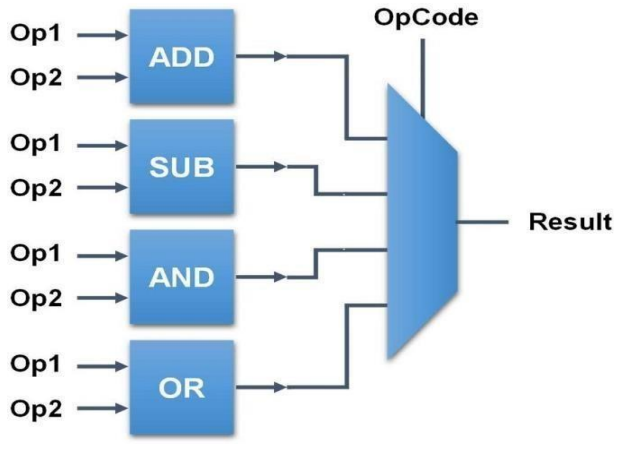


Figure 1: Arithmetic Logic Unit

Lab Task 1: Design an Arithmetic Logic Unit

In accordance with the above instruction set architecture, design a $n - bit$ wide arithmetic logic unit (must be parameterized) supporting instructions listed in the instruction set architecture, and verify the functionality with a testbench.

Now that we have designed a fully functional arithmetic logic unit, our next goal is to design a data storage unit to store the data, namely a register file, which can supply operands to the ALU.

Our instruction set architecture features four registers, each of which can be one of the two operands of our arithmetic logic unit as shown in **Figure 2**.

With reference to the instruction set architecture, the user can access two operands as a source to supply data to the arithmetic logic unit

$$rs1, rs2$$

and one operand as a destination to store the result of our operation

$$rd$$

These encodings are designed to access the data elements stored in the register file,

$$x0,\ x1,\ x2,\ x3$$

The encoding of $rs1$, $rs2$ and rd return the data stored in the register file according to the following encodings.

Encoding $(rs1, rs2, rd)$	**Accessed Element**
00	$x0$
01	$x1$
10	$x2$
11	$x3$

The element $x0$ of the register file is a read-only register which supplies the value 0.

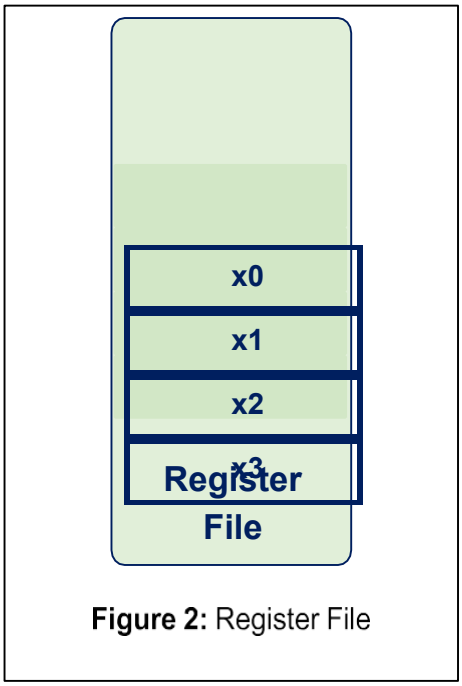


Figure 2: Register File

Lab Task 2: Design a Register File

In accordance with the above instruction set architecture, design a parametrized register file which can supply two operands and write back one data element, $n - bit$ each simultaneously. The write-back data is written on positive edge of the clock cycle.
To initialize the register file with user data, one of the following commands can be utilized.

```
$readmemh("rom_image_hex.mem", test_memory); // For Hexadecimal File
$readmemb("rom_image_bin.mem", test_memory); // For Binary File
```

It is highly recommended to read a binary file to enhance the understanding of microprocessor working.
Ensure that your design is parameterized and verify the functionality with a testbench.

Lab Task 3: Design an Instruction Memory

In order to perform user operations, the instructions need to be stored in a memory inside the processor, in addition to the data on which operations are to be performed. Design an addressable memory with **eight-bit word length** and a parameterized depth (number of instructions). Ensure that your instruction memory can be initialized with binary file.

Lab Task 4: Design of A Program Counter

A program counter is an essential component of any microprocessor. It addresses the instruction memory and allows the processor to locate instructions on the memory. The program counter is a simple accumulator comprising of a single memory element (register) and an adder. It shall be capable of addressing each and every instruction located in the instruction memory.

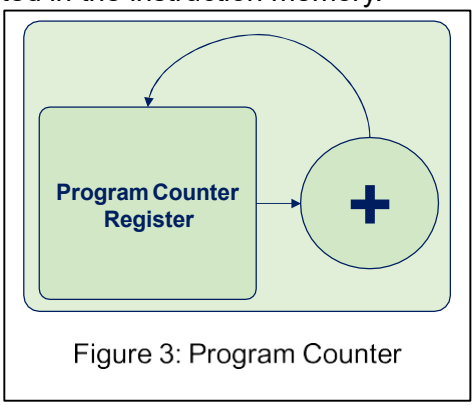


Figure 3: Program Counter

Create a program counter to address instructions from the instruction memory and verify its functionality with a testbench. The program counter shall be initialized with zero and increment on each clock cycle.

Lab Task 5: Integration of Components

Now that each and every component of our desired microprocessor has been developed, interface the components based on the following system level diagram. Finally, adjust the parameters as follows,

Parameter	Name (SystemVerilog)	Default Value
Instruction Memory Depth	IMEM_DEPTH	4 Words
Register File Width	REGF_WIDTH	16-Bit
ALU Width	ALU_WIDTH	16-Bit
Program Counter Maximum Value	PROG_VALUE	3

and initialize the instruction memory with user instructions located in the file `./support_files/fib_im.mem` and register file with the following

memory initialization file.

`./support_files/fib_rf.mem`

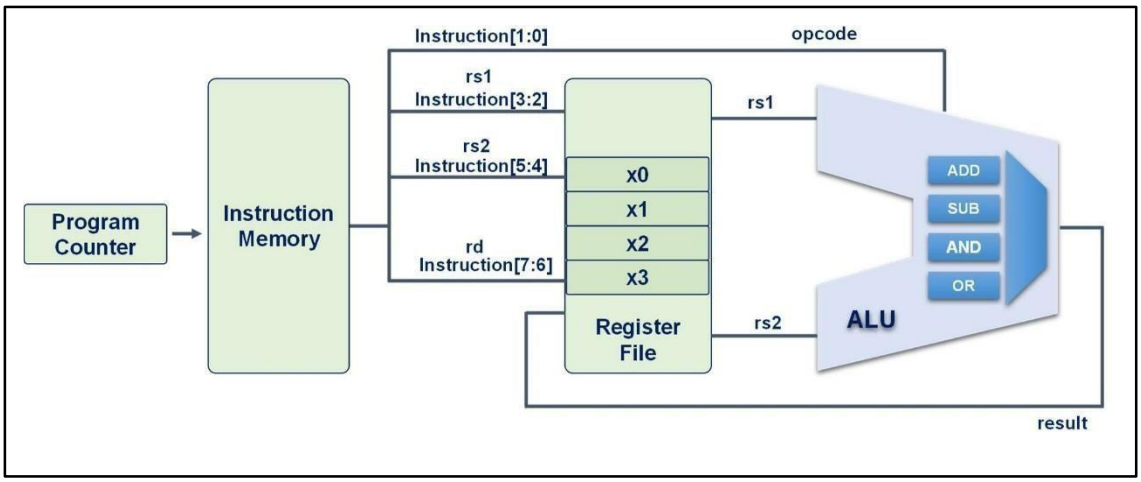
Note that these are binary files and must be loaded into the respective memory with correct functions described above.
It is essential to include the following lines in the register file module before handing over the completed labs.
The following block needs to be inserted at the top of the module before any writing any logic.

```
integer     fd;
integer     i;
initial begin
        // Create a new file
        fd = $fopen("regfile.dump", "w");
        #100;
        $fclose(fd);
end
```

The following block needs to be inserted in the same module just before endmodule. It is important that <clk-signal> and <reg-file-name> be replaced with the clock signal nameand register file variable name.

```
always @ (posedge <clk-signal>) begin
   for (i = 0; i < 4; i=i+1) begin
      $fdisplay(fd,<reg-filename>[i]);
end
```

```
end
```



References

1. What is High-Level-Programming Language? | by Saikrishna Reddem | Medium
2. RISC-V Assembly Manual
3. The RISC-V Reader