Lab 01
Custom Processor Design, Program Compilation and Assembly Language

Module ID: CX-204

Modern Computer Architecture

Instructor: Dr. Muhammad Imran

Version 1.1

Document History

The changes and versions of the document are outlined below:

Version	State / Changes	Date	Author (s)
1.0	Initial Draft	May, 2024	Ali Aqdas
1.1	Modified	December, 2024	Dr. Muhammad Imran

Table of Contents

Objectives

The objective of this lab is to enable students to answer following questions:

- How does a simple microprocessor work?
- What is the program counter, instruction memory and register file?
- How does software interact with the hardware?
- How to use **gcc** compiler to convert a C code to Assembly Language?
- How to use the RISC-V GNU Toolchain?
- What is Assembly Language?

Tools

- SystemVerilog
- Xilinx Vivado
- Synopsys VCS (if available)
- gcc
- RISC-V gcc toolchain

Deliverables

- RTL design (source code files of simple microprocessor)
- Compiled code

1. Building a Custom Microprocessor

1.1. Instructions for Lab Tasks

The submitted tasks (on github repo) 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 directory names without spaces and each file name must be exactly as listed in the table below. For each signal, use exactly the names given in figures used in this document (See Figure 1.4).

./cx-204/lab1/	register_file.sv
	alu.sv
	instruction_memory.sv
	program_counter.sv
	top.sv
	support_files/ 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 (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

1.2. Introduction to Processor Design

In order 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.

1.3. 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.

instruction [7:0]			
7 6	5 4	3 2	1 0
rd	rs2	rs1	opcode
2	2	2	2

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

Lab Task 1.1. 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.

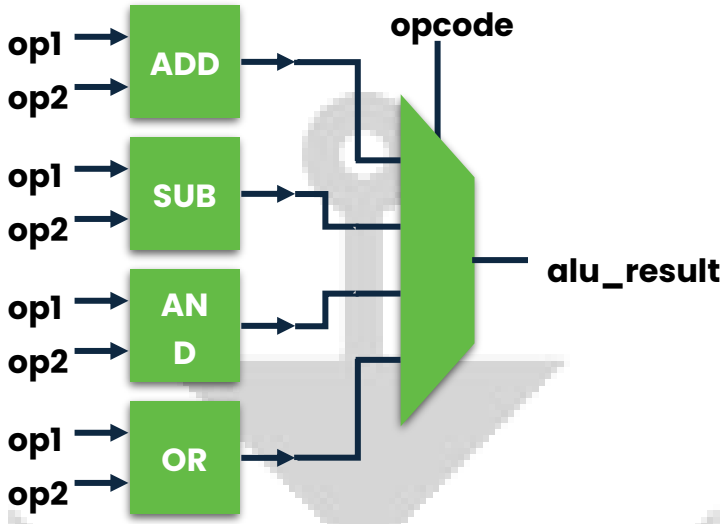

Figure 1.1: Arithmetic Logic Unit

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 1.2**.

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, namely.

$$x0, x1, x2, x3$$

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.

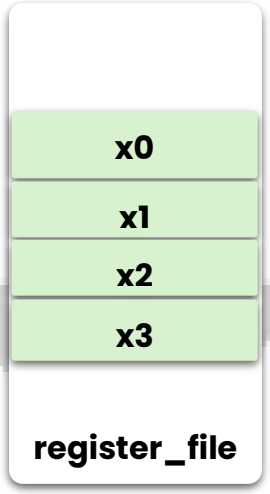

Figure 1.2: Register File

Lab Task 1.2: 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 1.3: 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 1.4: 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.

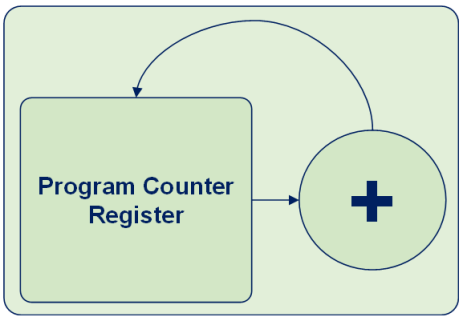

Figure 1.3: Program Counter

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 1.5: 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
./support_files/fib_rf.mem
```

and register file with the following memory initialization file.
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 name and register file variable name.

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

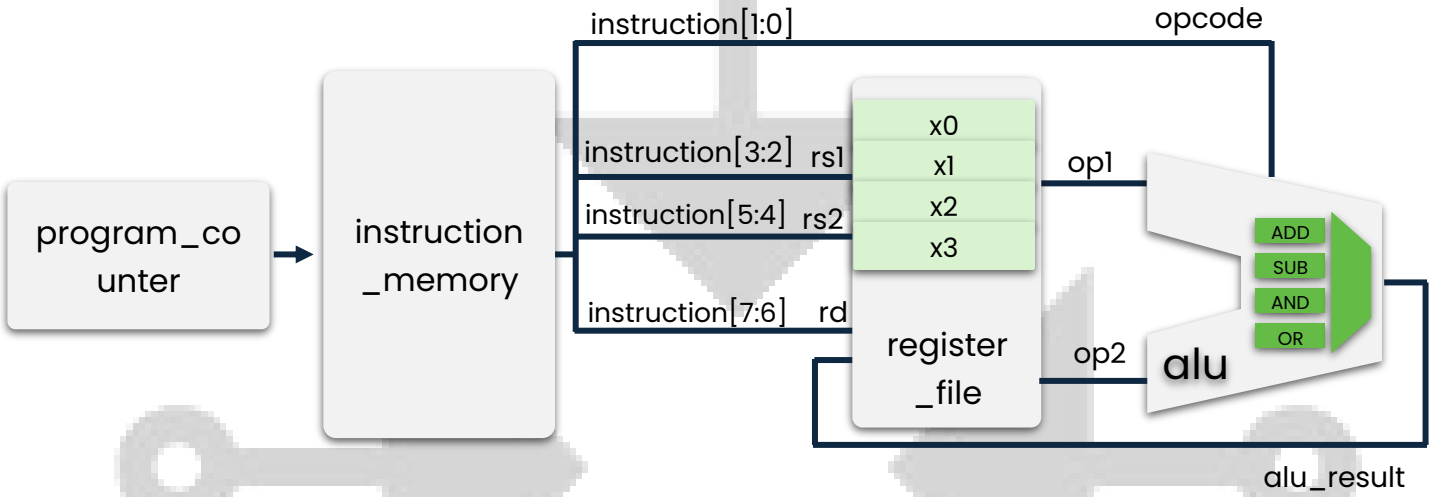

Figure 1.4: Complete Microprocessor

2. Introduction to Program Compilation and Assembly Language

2.1. 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.

2.2. Introduction to Assembly Language

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 2.1**.

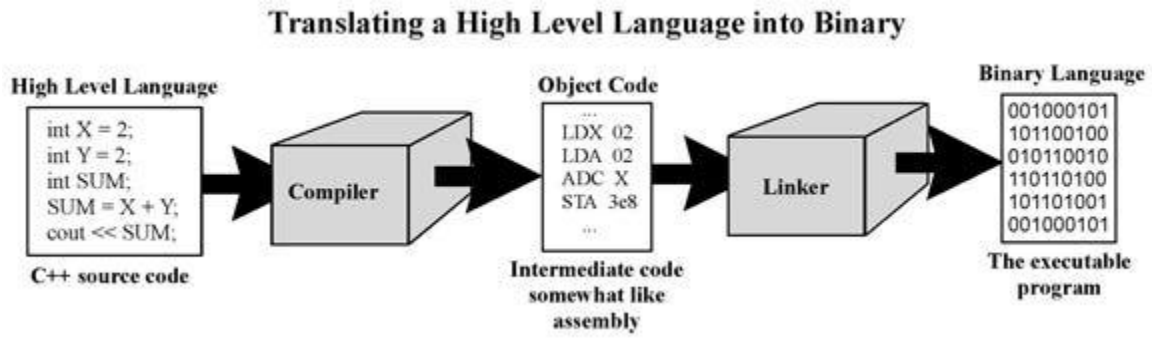

Figure 2.1: Translating a High Level Language to Binary (Medium [1])

2.3. 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 2.1: 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, we will compile the same program with RISC-V GNU Toolchain to generate RISC-V Assembly and save it in hello.rv32.asm and

hello.rv64.asm. If you have not setup the RISC-V GNU Toolchain, you can set it up using following command (the process takes 2-3 hours).

```
$ cd /home/$USER && sudo mkdir risc-v && cd risc-v && sudo apt
install git-all -y && sudo apt-get install autoconf automake
autotools-dev curl python3 python3-pip libmpc-dev libmpfr-dev
libgmp-dev gawk build-essential bison flex texinfo gperf
libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git
cmake libglib2.0-dev libslirp-dev -y && sudo apt-get install
libc6-dev-i386 -y && sudo git clone
https://github.com/riscv/riscv-gnu-toolchain && cd /opt && sudo
mkdir /opt/riscv && cd /home/$USER/risc-v/riscv-gnu-toolchain
&& sudo ./configure --prefix=/opt/riscv --enable-multilib &&
sudo make linux && echo 'export PATH="$PATH:/opt/riscv/bin"' >>
~/.bashrc && source ~/.bashrc
```

The RISC-V GCC Compiler has the following binary riscv64-unknown-linux-gnu-gcc for compilation. RV64 refers to the 64-bit variant of RISC-V while RV32 refers to the 32-bit variant of RISC-V. For compiling to 32-bit integer (RV32i) variant use the following command.

```
$ riscv64-unknown-linux-gnu-gcc -mabi=ilp32 -march=rv32i
hello.c -o hello.rv32.asm -S
```

mabi is used to specify the integer and floating-point calling convention (we will discuss about it in lecture), while march is used to specify the architectural specifications. Through these two arguments, you can vary the risc-v variant and extensions that you want to use during compilation. The default (without mabi / march arguments) targets the RISC-V 64 bit variant. Now compile for the 64-bit variant and store the output in **`hello.rv64.asm`**

Compare the 32-bit and 64-bit files and note the differences. Submit all the assembly files for lab assessment.

2.4. 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

Table 1: Assembler Directives (Courtesy of RVFPGA)

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.

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 pseudo-instructions 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 2.2: Identifying Components of Assembly Language

List the following in the RISC-V (RV32) Assembly of "hello.c" generated in Task 2.1.

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 2.3:

Now, fully compile "hello.c" program to generate a binary stream. This can be done by removing the -S flag from the commands given in Task 2.1. Add -static option when compiling for RV32i and save the binary file as hello.rv32.bin.

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"

```
$ riscv64-unknown-linux-gnu-objdump -mabi=ilp32 -march=rv32i -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.

```
hello.rv32.bin:     file format elf32-littleriscv

Contents of section .note.ABI-tag:
 10114 04000000 10000000 01000000 474e5500  ............GNU.
 10124 00000000 05000000 04000000 00000000  ................
Contents of section .rela.plt:
 10134 00a00700 3a000000 a2950100           ....:.......
Contents of section .plt:
 10140 17ae0600 032e0eec 67030e00 13000000  ........g.......
Contents of section .text:
 10150 411106c6 09203571 22cd17d4 06001304  A.... 5q".......
 10160 a4501c44 26cb4ac9 06cf1309 02b2a284  .P.D&.J.........
 10170 638f2701 0547af27 041481e7 af26e418  c.'..G.'.....&..
 10180 fdfa81c7 2285ef40 10272324 24015c40  ....".."@.'#$$.\@
 10190 58448507 5cc011ef 05459307 000202c4  XD..\....E......
 101a0 48c43ec2 93087008 4c000146 a1467300  H.>...p.L..F.Fs.
 101b0 00005844 85476314 f7065c40 97d60600  ..XD.Gc...\@....
 101c0 23aa064a fd175cc0 99ef97d6 060023a1  #..J..\.......#.
 101d0 064aafa7 f40a6358 f70017d5 06001305  .J....cX........
 101e0 a548ef40 702a1945 efb0f208 1c441309  .H.@p*.E.....D..
 101f0 02b26382 27030547 afa70414 81e7afa6  ..c.'..G........
 10200 e418fdfa 99c717d5 06001305 e545ef40  .............E.@
 10210 901e2324 24015c40 85075cc0 21a08947  ..#$$.\@..\.!..G
 10220 6313f702 5c003e85 0d471306 c0088145  c...\.>..G.....E
 10230 58c4ef90 804efd57 aa850146 19453ec6  X....N.W...F.E>.
```

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.

References

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

Good Luck