Lab 1: Using RISC-V Tools

Instructor: Shahriar Shahabuddin
TAs: Hridi Prova Debnath, MD Salman Sakib, KM Faisal

1. Introduction

The main objectives of this lab are to:

- Assemble and disassemble programs with the RISC-V GNU compiler
- Simulate RISC-V programs with Spike (a functional RISC-V simulator)
- Simulate and debug the `riscv-wally` SystemVerilog model with Verilator and Questa

RISC-V is an open-source Instruction Set Architecture (ISA). To help get more interaction with the RISC-V ISA, many developers use many open-source tools. Many of these tools rely on the GNU gcc and binutils compilers. GNU GCC is a free and open-source compiler system produced by the GNU Project. It is a cornerstone of free software development and a key part of many Linux distributions and open-source projects. GNU Binutils is a collection of binary tools that work closely with GCC to produce executable programs and manage binary files. It is often bundled with GCC to provide a complete toolchain for software development. These tools are just a few of them that encompass the RISC-V toolsets and are actively and constantly upddated through online repositories, such as git. You will learn how to use many of them for this lab and see them in action.

The GNU Project was founded by Richard Stallman in 1983 with the goal of creating a free, Unix-like operating system that respects users' freedom to study, modify, and share software. Announced on September 27, 1983, GNU stands for "GNU's Not Unix," highlighting its compatibility with Unix while being entirely free of proprietary code. Stallman also introduced the philosophy of free software, formalized in the Four Freedoms, which emphasize user freedom over software ownership. As part of the GNU Project, Stallman developed essential tools like GNU Emacs, the GNU Compiler Collection (GCC), and the GNU Debugger (GDB), providing the foundational utilities needed for the system. While the GNU operating system itself was incomplete for years, the combination of GNU tools with the Linux kernel formed the basis of modern GNU/Linux systems, making the GNU Project a cornerstone of open-source software and digital freedom movements worldwide.

Git, created by Linus Torvalds in 2005, is a distributed version control system that has revolutionized software development by enabling efficient tracking and collaboration on code. Torvalds developed Git to support the development of the Linux kernel after a licensing dispute with the proprietary version control system previously in use. Git was designed with speed, efficiency, and support for distributed, non-linear workflows in mind, making it ideal for large, collaborative projects. Its decentralized nature allows every user to have a full copy of the repository, enabling offline work and reducing dependency on a central server. Git introduced powerful features like branching and merging, which facilitate experimentation and parallel development without disrupting the main codebase. Over time, Git has become the industry standard for version control, powering major platforms like GitHub, GitLab, and Bitbucket, and is essential in modern DevOps, open-source development, and team collaboration worldwide.

If you are not already familiar with `git`, review our textbook's `git` tutorial in the Appendix of the textbook [1]. Follow the directions below to clone the `https://github.com/openhwgroup/cvw.git` repository into your home directory. Remember to clone the repository *recursively* (`--recursive`) because there are submodules the repository needs. Cloning a repository with submodules is important in computer architecture projects so that dependent libraries and modules are always consistent with the main codebase, preventing version mismatches when compiling hardware or simulation tools.

2. Cloning and Setting Up Wally Locally

To get started with Wally for this lab, it is essential to clone the repository correctly and set up the environment. Ensuring a proper setup is critical in computer architecture because projects often involve

multiple simulation and verification tools that must work seamlessly together.

This next couple of subsections walk you through a typical compilation process. It is meant to take you step by step through each item. The output is important for those in both software and hardware to see what the program is doing and generating. It is also useful in helping debug how code maps the program to the hardware. Therefore, pay close attention to what is being produced by being inquisitive about the output and asking questions what something is confusing.

2.1 Cloning the Repository

Wally's repository includes several submodules, which are integral to its functionality. To ensure all submodules are cloned along with the main repository, you must use the `--recursive` flag. Submodules often contain dependencies or additional resources that the main repository references, making them crucial for proper setup. Properly managing these dependencies prevents broken integrations, which are a major concern when building complex hardware/software co-designs.

To clone the `cvw` repository:

- Fork the openhardware cvw repository: https://github.com/openhwgroup/cvw/tree/main
- Use the command: `git clone --recurse-submodules https://github.com/<yourgithubid>/cvw`.
- Navigate into the directory: `cd cvw`.

2.2 Sourcing the Setup Script

After cloning the repository, it is necessary and critical to source the `setup.sh` script to configure your environment. This script sets up the required environment variables, updates your `PATH`, and activates the Python virtual environment. Having a correctly configured environment is vital in computer architecture labs, as mismatched paths or variables can cause inconsistencies between simulation tools and compiled binaries.

Setup scripts play a critical role in operating systems by automating tasks, configuring environments, and ensuring consistency across system configurations. These scripts, typically written in scripting languages like Bash, Python, or PowerShell, simplify complex and repetitive processes, making them vital for system administration, deployment, and maintenance. You can check if your script is not set up correctly by typing: `echo $WALLY` at the terminal. If you do not get a response fronm this command, you did not set up the environment correctly and must run the setup command again.

The following steps are utilized to set up the enviornment inside the `cvw` directory:

- Run the command to setup the Wally environment: `source ./setup.sh`.

2.3 Next Steps

After setting up the repository and sourcing the environment, you can test your setup by compiling and running the `HelloWally` program. This quick check ensures that your compiler toolchain, simulators, and environment are aligned before delving into more complex design

- Navigate to the example program: `cd examples/C/hello`.
- Compile the program: `make`.
- Simulate it with Questa: `wsim --sim questa rv64gc --elf hello`.

By following these steps, you ensure that Wally is set up correctly and ready for use in this lab, minimizing the risk of environment-related failures, which are common in hardware/software co-development.

3. Compiling and Disassembling Programs with the GCC Toolchain

We will use the `riscv64-unknown-elf-gcc` cross-compiler to build RISC-V assembly and C programs into ELF binaries, which can then be simulated with Spike or run on hardware. In computer architecture, the compilation step is crucial to transform high-level or assembly code into machine code that can be executed on a target processor model.

3.1 Using and Examining Output from GCC

GCC (GNU Compiler Collection) is a cornerstone in computer engineering due to its versatility, cross-platform portability, and support for multiple programming languages like C, C++, and Fortran. Its powerful optimization features enhance performance, especially in resource-constrained environments like embedded systems. As a free and open-source tool, GCC fosters innovation and collaboration while adhering to industry standards for language compliance. Widely used for systems programming, cross-compilation, and embedded development, GCC integrates seamlessly with tools like Make and GDB, providing a complete development environment.

In summary, GCC plays a pivotal role in computer engineering by providing a reliable, versatile, and high-performance compiler that supports a wide range of applications, from academic research to industrial software development. Its free and open-source nature, combined with extensive features, makes it indispensable for modern engineering practices.

3.1.1 Example: `example.S`

As a simple illustration, we can use a short assembly file, `example.S`:

1. **Navigate to the example directory:**

   ```
   cd $WALLY/examples/asm/example
   ```

   Knowing how to locate and organize source files is essential for clarity and collaboration in computer architecture projects, where multiple modules often reside in different directories.

2. **Assemble and link the program (targeting RV32I and the ilp32 ABI):**

   ```
   riscv64-unknown-elf-gcc -o example -march=rv32i -mabi=ilp32 -mcmodel=medany \
       -nostartfiles -T../../link/link.ld  example.S
   ```

   Choosing the correct architecture (`rv32i`) and ABI (`ilp32`) ensures the generated machine code aligns with the processor's word size and calling conventions, both crucial to functional correctness in hardware simulation.

3. **Disassemble the resulting binary to see machine code and assembly:**

   ```
   riscv64-unknown-elf-objdump -D example > example.objdump
   ```

   Viewing the disassembled output is key to verifying that the compiler generated the expected instructions, which is vital in understanding pipeline behavior and performance trade-offs in computer architecture.

4. **(Optional) Compare the original `example.S` and the disassembly:** By comparing the original source with the disassembly, you can see how the assembler handles pseudoinstructions and immediates, giving insight into how certain high-level constructs map to hardware instructions.

You can also compile and generate the objdump using a Makefile by typing `make`. A Makefile is a file used by the make build automation tool to specify how to compile and link a program, as well as other build-related tasks. It provides a set of rules, typically written in the form of targets, dependencies, and commands, to define the steps needed to transform source files into executables or other output. The make tool reads the Makefile and executes the necessary commands, only rebuilding parts of the project that have changed, which saves time and ensures efficiency in the build process.

3.2 Automating Builds with Makefiles

Many examples include a `Makefile` that automates:

- Building (`make`)
- Cleaning (`make clean`)
- Optional simulation or signature checks (`make sim`)

Automated build systems are crucial in computer architecture for ensuring that complex multi-file projects are compiled with consistent flags, especially when integrating multiple IP blocks or third-party libraries.

For instance, in the `example` directory:

1. Inspect the `Makefile`:

   ```
   cat Makefile
   ```

2. Build the program:

   ```
   make
   ```

3. Clean intermediate and output files:

   ```
   make clean
   ```

3.3 Simulating RISC-V Programs with Spike

Spike is a functional simulator for RISC-V. After compiling a program to an ELF binary, you can run:

```
spike --isa=rv32i *programname*
```

Optionally, you can debug step-by-step (`-d`), inspect registers, or capture a signature. Early-stage functional simulators are crucial in computer architecture because they allow rapid testing of instruction correctness before running time-consuming cycle-accurate simulations.

3.3.1 Example: `sumtest.S`

A more complex assembly program is `sumtest.S`, which computes a sum and demonstrates performance measurement. Such performance checks help verify architectural decisions, such as branching strategies or pipeline depth.

1. **Navigate:**

   ```
   cd $WALLY/examples/asm/sumtest
   ```

2. **Build:**

   ```
   make
   ```

3. **Simulate with Spike and generate a signature:**

   ```
   spike +signature=sumtest.signature.output sumtest
   ```

   Signatures are commonly used to compare actual results against expected outcomes in large-scale verification frameworks.

4. **Compare with the reference:**

   ```
   diff sumtest.signature.output sumtest.reference_output
   ```

   A successful match indicates your program ran as expected, which is essential to validate the correctness of the instruction execution flow.

3.3.2 Reviewing the Sum Example: `sum.S`

You can examine `sum.S` to see how it handles function calls, stack usage, and the RISC-V calling convention. Studying these details helps architects understand how calling conventions affect pipeline hazards, register usage, and memory bandwidth.

3.3.3 Makefile and ELF Analysis

1. The `sumtest/Makefile` automates building and simulation:

   ```
   make clean
   make
   make sim
   ```

2. The `sumtest.objdump` file shows a disassembly of the final ELF.

3. Use `riscv64-unknown-elf-readelf -a sumtest` to inspect ELF metadata, including sections, symbol tables, and relocation info. This reveals how the compiler and linker organize code and data, which can impact both performance and memory footprint.

3.3.4 C Program Example: `sum.c`

You can also compile C code:

1. Navigate to:

   ```
   cd $WALLY/examples/C/sum
   ```

2. Build and simulate:

   ```
   make
   spike sum
   ```

   Testing higher-level languages helps you verify that the entire compilation flow—from C source to final RISC-V binary—functions correctly and adheres to architectural constraints.

3.4 Simulation with Siemens Questa

We utilize Questa to perform HDL simulations on the hardware and programs we develop. For this project, we will primarily use the rv64gc hardware, with the rv32 architecture being introduced in a subsequent lab. Both architectures share a similar ISA, allowing for seamless transition and comparison during development and testing.

Cycle-accurate simulation is a critical technique in hardware development and computer architecture design, offering detailed modeling of a hardware system's behavior at the level of individual clock cycles. This approach provides an accurate representation of a design's performance, including timing, execution, and interactions between components. By using cycle-accurate or event-based simulation, we can identify and address key issues such as pipeline stalls, hazards, and incorrect signal timings that may not be evident in a functional-only model. This level of precision is essential for validating complex hardware designs and ensuring they meet performance and reliability requirements.

To simulate a design, let's try simulating the sum design. You want to compile the sum.c program in the `$WALLY/examples/C/sum` directory using a Makefile. We now want to simulate our design and see what the impact of our program is in real hardware. [1].

In class, we learned about the importance of cycle time, cycles per instruction, and clock speed using the important equation found in [2]:

$$\text{CPU time} = \text{Instruction Count} \times CPI \times \text{Clock cycle time}$$

Optimization Level	mcycle	minstret	True Cycles = True # Instructions
None	115	132	62
-O	31	38	21
-O2 or -O3	11	16	1

Table 1: Cycles to execute sum(4)

This equation really indicates that performance is a function of many parameters from what the programmer does to how the hardware behaves.

If you simulate your compiled program after make, you can simulate with spike. The program adds the numbers from 1 to 4 and successfully prints the sum of 10. It also prints the number of cycles and instructions retired during the sum function call, between the `setStats(1)` and `setStats(0)` calls. These two function calls use something that both programmers and hardware designers utilize called performance counters.

Performance counters are specialized hardware registers or mechanisms used in computer architecture to monitor, measure, and analyze the performance of a system. They provide valuable insights into how hardware components operate under different workloads, enabling engineers to optimize both hardware and software for better performance and efficiency.

RISC-V has several CSRs that are used for performance measurement. As we will learn in class, RISC-V uses control and status registers (CSRs) that hold information such as flags from a floating-point operation, which RISC-V features are enabled, and where to go when a trap occurs. The CSRs also include performance counters to track things like the number of cycles the processor has run, the number of instructions that have retired (i.e., finished successfully), and other implementation-dependent data.

Table 1 shows the number of cycles and instructions retired to run `sum`, as reported by the program at various optimization levels. Spike simulates one instruction per cycle, so the two columns ideally would be the same, but the program has some overhead to call `setStats` and read each performance counter, and the overhead differs slightly between counters because of the order in which the counters are read. Table 1 also shows the true number of instructions used by `sum`, measured by manually counting during the Spike simulation. These *performance counters* are more useful for longer programs where the overhead is negligible.

In this `sum` program, `instret` (minstret in the spike simulation) indicates how many instructions have been retired (i.e., completed) and `cycle` (mcycle in the spike simulation) indicates how many clock cycles have passed. The `csrr reg, CSR` pseudoinstruction reads a CSR into reg, an integer register. The example in the next section illustrates using `instret` to count the number of instructions executed by a function call. In Spike, `cycle` and `instret` normally return the same value because one instruction is executed in each cycle.

Now, let's run the same program against the hardware. To run the program on the hardware, type the following: `wsim --sim questa rv64gc --elf sum` and compare the CPI you get with what you got with spike.

4. Laboratory: Using RISC-V for FIR filtering and Examining its Performance

In Signals and Systems class, you learned that looking at signals in the time and frequency domain is helpful in processing data. One particular important use of this is called Finite Impulse Response filtering. FIR filtering is very important in a field called Digital Signal Processing (DSP) and is often used in the audio engineering area among other fields. FIR (Finite Impulse Response) filtering refers to a method of digital signal processing where a signal is filtered using a linear filter whose response to an impulse (a signal with value 1 at a single point and 0 elsewhere) is of finite duration, meaning that it settles to zero in a finite number of steps. More information on FIR filters can be found in the white paper in your repository and Wikipedia `https://en.wikipedia.org/wiki/Finite_impulse_response`.

FIR (Finite Impulse Response) filtering is a digital signal processing technique used to manipulate or analyze signals by filtering out certain frequencies or enhancing others. A FIR filter operates by applying a weighted sum of a finite number of past input samples to produce each output sample. Unlike its counterpart,

the IIR (Infinite Impulse Response) filter, FIR filters have a finite response, meaning the output depends only on a limited number of input samples, making them inherently stable. FIR filters are widely used in applications such as audio processing, communications, and image enhancement because they can achieve linear phase, which preserves the shape of signals by ensuring all frequency components are delayed equally. In essence, FIR filtering aims to isolate desired frequency components or suppress unwanted noise, providing clean and precise signal manipulation in various engineering and scientific fields.

An M-tap Finite Impulse Response (FIR) filter computes an $N - M + 1 - element$ output sequence $Y[0], Y[1], \ldots, Y[N - M + 1]$ as a weighted sum based on an N-element input sequence $X[0], X[1], \ldots, X[N - 1]$ and M filter coefficients $c[0], c[1], \ldots, c[M - 1]$ according to:

$$Y[j] \quad = \quad \sum_{i=1}^{M-1} c[i] \cdot X[j - i + (M - 1)]$$

The filter coefficients are sometimes called the "taps" of a FIR filter and the number of taps and subsequently the value of M increases the order and robustness of the filter. However, higher-order filters can be hardware intensive and possibly require a lot of energy to perform.

Digital signal processing often uses fixed-point arithmetic to represent fractional numbers with integers. For example, `Q1.31` (sometimes called Q31 as an abbreviation) fixed-point uses 32-bit 2's complement integers to represent a number between $[-1, 1)$ with 31 fractional bits. Fractional notation in binary numbers is easily visualized by assuming that the weights are negative after the radix point. The most significant bit has a weight of -1 for a `Q1.31` two's complement notation, so:

- $0.625 = 1/2 + 1/8 = 0.101_2$ is written as `0101000...00 = 0x5000_0000`
- $-0.75 = -1 + 1/4 = 1.010_2$ is written as `1010000...000 = 0xA000_0000`

Addition is exactly the same as with ordinary integers:

$$\begin{aligned} 0.625 + -0.75 &= \texttt{0x5000\_0000 + 0xA000\_00000 = 0xF000\_0000} \\ &= 1.111_2 = -1 + 1/2 + 1/4 + 1/8 = -0.125 \end{aligned}$$

Applying standard signed (2's complement) integer multiplication to `Q1.31` numbers produces a 64-bit product in `Q2.62` format. (Note that signed integer multiplication differs from unsigned integer multiplication because the most significant bit has a negative weight). To get the result back into `Q1.31` format, left shift by 1 to discard the most significant bit, and then drop the bottom 32 bits. For example:

`0.625 * -0.75 = 0x5000_0000 * 0xA000_0000 = 0xE200_0000_0000_000a = 11.10001` in `Q2.64`.

Shifting left and discarding the bottom bits gives:

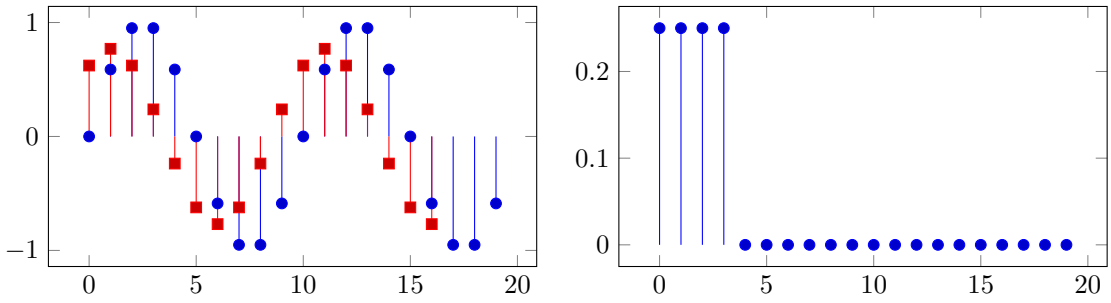
$$\texttt{0xC400\_0000} = 1.10001_2 = -1 + 1/2 + 1/32 = -15/32$$

which is the correct answer.

We will now use this information and apply what we have learned about compiling and simulating to complete and test the provided program `fir1.c`. This program is incomplete and some empty function definitions for `add_q31()`. The file also contains arrays representing our input signal and FIR filter in `Q1.31` format. The input signal is $\sin(\frac{x*2\pi}{10})$ with $0 \leq x \leq 19$ and the filter coefficients are all 1/4. Figure 1 contains graphs of the input, output, and filter values. Notice that each red output is the average of the four input values ahead of that output in Figure 1. There is also C and Python programs available in Canvas that allow you to see the representation in `Q1.31` form too.

Complete the program `fir1.c` in the Canvas to perform FIR filtering of Q1.31 fixed-point numbers in RV64GC. The filter coefficients are each slightly different than 1/4 so the compiler can't optimize them into a single coefficient. Write a `Makefile` with `-O` optimization. Compile the example with make and run the example with `spike fir`. How many cycles did the FIR function take? Modify the Makefile to enable -O2 optimization. Recompile and rerun. How many cycles did the function take? Compare running it with Siemens Questa?

In summary:



(a) Input/Output Signals (b) Filter Coefficients

Figure 1: Graphs of the expected result of fir1.c (Note: inputs shown with blue dots and outputs with squares in Figure (a))

1. Copy `fir1.c` into a new directory in your folder in your machine at `cvw/examples/C/fir1`
2. Implement the functions `add_q31()`. Inspect `mul_q31()` to understand how multiplication is done. Also look at the complexity of the fir function and see why it can be quite computationally intensive.
3. Using those functions for add, implement `fir()`
4. Write a Makefile to compile this code for RV64GC with -O optimization. (It is important that you use 64 bit architecture, otherwise you won't have enough bits for Q31 multiplication). The $XLEN = 64$ rv64gc architecture still stores the instruction in a 32-bit format but occupies a 64-bit segment of memory. Change the Makefile when needed to compile and test with both -O and -O2 optimization.
5. Record the output of each simulation under each optimization flag in a file called `results.md`.

4.1 Using Assembly to Optimize Designs

Assembly routines are sometimes crucial to develop over higher-level languages like C or Python when performance, control, or hardware-specific features are critical. As discussed in class, understanding assembly is critical to understanding how to write programs that interact with hardware.

Assembly provides direct access to the processor's instructions, enabling developers to write highly optimized code that can outperform higher-level implementations, which is essential in areas like real-time systems, cryptography, or signal processing. It allows precise control over hardware, such as registers, I/O ports, and custom processor instructions, which higher-level languages may not expose, making it ideal for device drivers, firmware, and embedded systems. Additionally, assembly eliminates the overhead introduced by abstraction layers in high-level languages, resulting in lean and efficient code, particularly for resource-constrained environments. It also enables the use of specialized processor instructions, such as SIMD or AVX, for performance optimizations that compilers might miss [2].

In critical real-time applications, where predictable execution and tight control over instruction sequencing are essential, assembly is often indispensable. Furthermore, it is valuable for working with legacy systems, debugging performance bottlenecks, and gaining a deeper understanding of processor and memory operations. While assembly offers unparalleled control, its complexity and reduced portability often mean it is reserved for performance-critical components, with the rest of an application written in higher-level languages for ease of development and maintenance.

For this part of the lab you will need to:

1. Make a separate folder called `fir2` and copy your Makefile and code from `fir1` into this directory. The following items are done in the context of being inside this folder.
2. Without copying the objdump exactly, rewrite the fir() function in assembly language in a separate file called `fir1.S`. There is a template in the repository to help you get started. Optimize this implementation to use as few instructions as possible. Be sure to identify that the assembly label `fir:` is a global symbol and not just a local one.

3. Make a header file with the function prototype of the `fir` function in it. This header file will be used to let `fir1.c` be aware of the assembly `fir` implementation. Include this header file in `fir1.c`.

4. Now comment out or remove the C implemented fir function in `fir1.c` Now there will be no name conflicts and `fir1.c` will use your assembly function when it calls `fir()`.

5. Now again, test this function under different optimization methods and place the results in a new `results.md`

5. What to Turn In

- For Section 2: provide a log or description of how you set up your environment, including any issues encountered. Use the history command to give us your logs.
- For Section 3: show how you compiled, disassembled, and ran each example (`example.S`, `sumtest.S`, `sum.c`, etc.). Again use the history command to document your steps.
- Include the outputs from your Spike and wsim simulations and organize the information into a table. Use the diff command to show your output is correct.
- For Section 4:
  - Submit your `fir1` and `fir2` folders. Make sure all relevant files are inside these directories and that the `results.md` files are included.
- Submit a README.md file describing how to execute each section as mentioned above.

References

[1] D. Harris, J. Stine, R. Thompson, and S. Harris, *RISC-V System-On-Chip Design*. Elsevier Science, 2025.

[2] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Elsevier Science, 2020.