ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

DESIGN AND TESTING OF A HIGHLY-ADAPTABLE RISC-V VERIFICATION ENVIRONMENT

SENIOR DESIGN PROJECT

Mete Kaan ÖZDEN
Deniz Zakir EROĞLU

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

JUNE 2025

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

DESIGN AND TESTING OF A HIGHLY-ADAPTABLE RISC-V VERIFICATION ENVIRONMENT

SENIOR DESIGN PROJECT

Mete Kaan ÖZDEN
040200215

Deniz Zakir EROĞLU
040200249

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

JUNE 2025

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

YÜKSEK DERECEDE UYARLANABİLİR BİR RISC-V DOĞRULAMA ORTAMININ TASARIMI VE TEST EDİLMESİ

LİSANS BİTİRME TASARIM PROJESİ

Mete Kaan ÖZDEN
040200215

Deniz Zakir EROĞLU
040200249

Proje Danışmanı: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

HAZİRAN, 2025

We are submitting the Senior Design Project Report entitled as “DESIGN AND TESTING OF A HIGHLY-ADAPTABLE RISC-V VERIFICATION ENVIRONMENT”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity .

Mete Kaan ÖZDEN .....
040200215

Deniz Zakir EROĞLU .....
040200249

FOREWORD

We would like to express our gratitude to our advisor Prof. Dr. Sıddıka Berna Örs Yalçın, who dedicated her time for us and provided all the help she could. We also acknowledge the support our families gave us; without which we would not succeed

June 2025

Mete Kaan ÖZDEN
Deniz Zakir EROĞLU

TABLE OF CONTENTS

Page

ABBREVIATIONS

AI	**:** Artificial Intelligence
ISA	**:** Instruction Set Architecture
ReLU	**:** Rectified Linear Unit
IEEE	**:** Institute of Electrical and Electronics Engineers
NaN	**:** Not a Number
MLP	**:** Multi Layer Perceptron
UVM	**:** Universal Verification Methodology
OOP	**:** Object Oriented Programming
DUT	**:** Design Under Test
MLP	**:** Multi Layer Perceptron
GUI	**:** Graphical User Interface
CSR	**:** Control and Status Register
FPU	**:** Floating Point Unit
SQRT	**:** Square Root
MDS	**:** Multiply-Divide-Square Root
RAM	**:** Random Access Memory
ROM	**:** Read Only Memory
RTL	**:** Register Transfer Level
WSL	**:** Windows Subsystem for Linux
RNE	**:** Round to Nearest, ties to Even
RTZ	**:** Round Towards Zero
RDN	**:** Round Down
RUP	**:** Round Up
RMM	**:** Round to Nearest, Ties to Maximum Magnitude
DYN	**:** Dynamic Mode
G Bit	**:** Guard Bit
R Bit	**:** Rounding Bit
S Bit	**:** Sticky Bit
MNIST	**:** Modified National Institute of Standards and Technology

LIST OF TABLES

Page

LIST OF FIGURES

Page

SUMMARY

In this project, the main goal was to create a reusable and scalable verification environment for RISC-V processors. The Hornet core, designed at Istanbul Technical University (ITU), was selected as the main processor for verification. Hornet mainly supports integer arithmetic and multiplication/division operations. It was later extended to support single precision floating point operations. Since RISC-V is an open-source instruction set architecture, it enables universities and research groups to design their own cores. However, the lack of open-source and general-purpose verification systems for floating point RISC-V cores limits this potential. This project focused on solving this problem by building an adaptable, open-source verification system. Spike was used as a reference software simulator to execute RISC-V instructions and check behavior. RISCV-DV was used to generate random test cases to evaluate how the processor behaves under many different input conditions.

The Ibex verification environment was also studied. Ibex is another open-source RISC-V core with an existing UVM-based testbench. To use this environment, multiple setup issues were resolved. The required simulation tools and RISC-V toolchain did not exist in ITU's laboratories. Therefore, a new simulator program named QuestaSim was used via a company named Electra IC's servers. To do this, several configuration changes needed to be made. After that, floating point test support was added by modifying instruction generation settings and writing a custom test configuration. Unfortunately, server access was later shut down due to technical issues. This part of the work helped in understanding RISCV-DV in general.

Before searching for a new environment, several manual fixes were applied to Hornet to improve memory usage and address floating point unit problems. Later, a new custom verification environment was created for Hornet. A tracing system was added to monitor instruction-level behavior during execution to be used in the new environment. This environment was designed to use Python to generate random instructions and to compare RTL-level results with Spike results. Edge behaviors, such as rounding, subnormal numbers, and NaN cases, were evaluated. To automate the process, scripts were created to generate programs, run simulations, parse logs, and compare outputs. These scripts made it possible to automatically detect mismatches which means that there is an error. This environment was built to work with widely used open-source tools such as RISCV-DV, Spike, and Python. A main script was added to manage the entire process, from code generation to result checking. This made the verification process much easier to repeat and reuse for future designs. Using this environment, many changes were made both in the FPU and in Hornet itself.

After fixing errors, it was desired to show that the verification system could support realistic applications such as a simple artificial neural network. A multilayer

perceptron model was trained using TensorFlow to recognize handwritten digits. This model was later adapted and executed on Hornet. Environment results showed that the processor was able to run the neural network correctly. This confirmed that Hornet can now be considered a fully functional open-source RV32IMF core.

In conclusion, an open-source, flexible, and automated verification environment was successfully built for testing Hornet. It enabled efficient error detection and supported real-world applications, making it a useful framework for both academic and industrial research.

ÖZET

Bu projede temel amaç, RISC-V işlemcileri için tekrar kullanılabilir ve ölçeklenebilir bir doğrulama ortamı oluşturmaktı. İstanbul Teknik Üniversitesi (İTÜ) tarafından tasarlanan Hornet çekirdeği, doğrulama işlemi için ana işlemci olarak seçildi. Hornet, temel olarak tam sayı aritmetiği ve çarpma/bölme işlemlerini desteklemektedir. Daha sonra, tek duyarlıklı kayan nokta aritmetiğini destekleyecek şekilde genişletilmiştir. RISC-V açık kaynaklı bir komut seti mimarisi olduğu için, üniversiteler ve araştırma grupları kendi çekirdeklerini tasarlayabilmektedir. Ancak kayan nokta aritmetiği destekli RISC-V çekirdekleri için genel amaçlı ve açık kaynaklı doğrulama sistemlerinin eksikliği, bu potansiyeli sınırlamaktadır. Bu proje, uyarlanabilir ve açık kaynaklı bir doğrulama sistemi oluşturarak bu sorunu çözmeyi hedeflemiştir. Spike, RISC-V komutlarını çalıştırmak ve davranışı kontrol etmek için referans yazılım simülatörü olarak kullanılmıştır. RISCV-DV ise işlemcinin farklı giriş koşullarındaki davranışını değerlendirmek için rastgele test senaryoları üretmekte kullanılmıştır.

Ibex doğrulama ortamı da incelenmiştir. Ibex, halihazırda UVM tabanlı bir testbench'e sahip olan başka bir açık kaynaklı RISC-V çekirdeğidir. Bu ortamı kullanmak için çeşitli kurulum sorunları çözülmüştür. İTÜ laboratuvarlarında gerekli simülasyon araçları ve RISC-V toolchain mevcut değildi. Bu nedenle, Electra IC adlı bir şirketin sunucuları üzerinden QuestaSim isimli bir simülatör programı kullanılmıştır. Bu işlemi gerçekleştirmek için bazı yapılandırma dosyalarında değişiklik yapılması gerekmiştir. Sonrasında, kayan nokta aritmetiği testi desteği sağlamak amacıyla talimat üretim ayarları değiştirilmiş ve özel bir test konfigürasyonu yazılmıştır. Ancak daha sonra, teknik sorunlar nedeniyle sunucu erişimi kapatılmıştır. Bu süreç, genel olarak RISCV-DV'nin nasıl çalıştığını anlamaya katkı sağlamıştır.

Yeni bir ortam arayışına başlamadan önce, Hornet üzerinde çeşitli manuel düzeltmeler yapılmıştır. Bu düzeltmeler, bellek kullanımıyla ilgili iyileştirmeler ve kayan nokta birimiyle ilgili sorunların giderilmesi amacıyla gerçekleştirilmiştir. Daha sonra, Hornet için özel bir doğrulama ortamı oluşturulmuştur. Bu yeni ortamda kullanılmak üzere, komut düzeyindeki işlem davranışını izleyebilen bir tracer sistemi eklenmiştir. Ortam, rastgele komut üretmek için Python kullanacak ve RTL düzeyindeki sonuçları Spike çıktılarıyla karşılaştıracak şekilde tasarlanmıştır. Yuvarlama, subnormal sayılar ve NaN durumları gibi uç durumlar değerlendirilmiştir. Doğrulama sürecini otomatikleştirmek için program üretimi, simülasyon çalıştırma, log dosyalarını ayrıştırma ve sonuç karşılaştırma işlemlerini yöneten scriptler yazılmıştır. Bu scriptler sayesinde hataya işaret eden uyuşmazlıklar otomatik olarak tespit edilebilmiştir. Ortam, RISCV-DV, Spike ve Python gibi yaygın açık kaynak araçlarla çalışacak şekilde inşa edilmiştir. Sürecin tamamını kod üretiminden sonuç kontrolüne kadar yöneten bir script eklenmiştir. Bu sayede doğrulama işlemi tekrar edilebilir ve gelecek

tasarımlar için yeniden kullanılabilir hale gelmiştir. Bu ortam kullanılarak hem FPU'da hem de Hornet üzerinde birçok değişiklik yapılmıştır.

Hatalar giderildikten sonra, doğrulama sisteminin gerçekçi uygulamaları destekleyip desteklemediğini göstermek amaçlanmıştır. Bu doğrultuda, basit bir yapay sinir ağı örneği test edilmiştir. El yazısı rakamları tanımak üzere TensorFlow kullanılarak bir multilayer perceptron modeli eğitilmiştir. Bu model, Hornet üzerinde çalıştırılmak üzere uyarlanmış ve simülasyon ortamında test edilmiştir. Ortamdan elde edilen sonuçlar, işlemcinin sinir ağını doğru şekilde çalıştırabildiğini göstermiştir. Böylece Hornet'in artık tam işlevsel, açık kaynaklı bir RV32IMF çekirdeği olarak değerlendirilebileceği doğrulanmıştır.

Sonuç olarak, Hornet'in doğrulanması için açık kaynaklı, esnek ve otomatik bir doğrulama ortamı başarıyla oluşturulmuştur. Bu ortam, hataların etkin bir şekilde tespit edilmesini sağlamış ve gerçek dünya uygulamalarını destekleyerek hem akademik hem de endüstriyel araştırmalar için kullanışlı bir çerçeve sunmuştur.

1. INTRODUCTION

1.1 Purpose of the Project

The verification of RISC-V processors faces challenges due to bespoke solutions tailored for specific cores, limiting scalability and adaptability. This creates a bottleneck for developers, particularly in validating processors with advanced instruction sets like RV32IMF for floating-point operations, which are critical for AI applications. Current methods are inefficient for diverse core designs, hindering broader adoption. This project addresses these issues by developing a modular, scalable, and reusable verification environment to ensure compliance with RISC-V standards, detect errors, and enable seamless integration into computationally demanding tasks, fulfilling a need for researchers and companies.

1.2 Literature Review

Current solutions for RISC-V core verification include tools and frameworks designed for specific needs but with notable limitations. The Ibex verification environment [3] is tailored for the Ibex core, offering precise validation but requiring extensive effort to adapt to other cores. Similarly, RISCV-DV [4] provides a strong foundation with its random instruction generation capabilities, yet it requires additional integration effort for hardware validation in diverse core designs. ITU's Hornet [5], supporting the RV32IM instruction set, demonstrates academic advancements in RISC-V core design. Extensions to Hornet, such as the F extension for floating point operations [6], emphasize the need for robust validation in floating point computation, a critical area for AI workloads. The RISC-V Instruction Set Architecture (ISA) specification [1] provides the foundational standards for compliance in all RISC-V cores. Tools like Spike, the RISC-V ISA Simulator [7], are used as reference models to verify compliance. However, Spike's focus on simulation limits its capability to address real-time validation needs in hardware. The proposed solution builds upon the

strengths of these existing tools and standards while addressing their limitations. By creating a scalable and modular verification framework, this project supports compliance across multiple cores, enables advanced feature validation, and facilitates practical applications such as AI workloads.

2. BACKGROUND INFORMATION

2.1 About RISC-V and Hornet

RISC-V is an open-source instruction set architecture created by researchers at the University of California, Berkeley, for teaching and research use [1]. Later, it became more popular and is now used around the world. Its open design also helps universities and companies create their own custom processors. At the same time, it is flexible and can grow to support many types of applications. In addition, Hornet is a 32-bit processor that supports the I and M instruction sets meaning that it can process basic integer arithmetic and multiplication and division. It was designed by students from Istanbul Technical University [5]. Later, it was updated to have the F extension, meaning that it has support to perform operations using floating point numbers [6].

2.2 About the IEEE-754 Standard

IEEE-754 is a standard for floating point numbers made by the IEEE (Institute of Electrical and Electronics Engineers) [8]. It is used in many processors. The standard helps to store and calculate decimal numbers in binary form. It is useful for science, engineering, and graphics. In this standard, there are different formats, such as single precision and double precision. However, Hornet only supports single precision floating point numbers.

2.2.1 Single precision format

In single precision, a number has 32 bits. The first bit is the sign bit. It shows if the number is positive or negative. The next 8 bits are the exponent. The last 23 bits are the mantissa. The real value of the number is calculated with this formula $(-1)^{\text{sign}} \times 1.\text{mantissa} \times 2^{(\text{exponent}-127)}$. This formula is used when the number is a normal value.

2.2.2 Subnormal numbers

Subnormal numbers are very small numbers close to zero. They are used when the exponent is zero and the mantissa is not zero. In this case, the formula becomes $(-1)^{\text{sign}} \times 0.\text{mantissa} \times 2^{-126}$. Subnormal numbers help avoid sudden jumps from zero to normal values. They give smooth results near zero.

2.2.3 NaNs (not a numbers)

NaN means “Not a Number.” It shows up when a calculation has no real answer, like dividing 0 by 0. There are two types of NaN:

- **Quiet NaN:** This type does not stop the program. It gives NaN as a result. The program continues to run.
- **Signaling NaN:** This type causes an error or warning. It tells the program that something is wrong.

The difference between them is in one special bit in the mantissa.

2.3 About RISC-V’s "F" Single Precision Float Extension

The F extension in RISC-V [9] adds support for floating-point operations. These operations work with decimal numbers using special registers. The extension follows the IEEE 754 standard. It allows the processor to do math like addition, subtraction, multiplication, and division with floating-point numbers. There are also instructions to move data between integer and floating-point registers. Some instructions check if two values are equal or compare their sizes. Other instructions help convert between integer and floating-point formats. All instructions in the F extension are listed and explained in the Table 2.1. One important thing to mention is that the Hornet Core does not have fused instructions FMADD.S, FMSUB.S, FNMADD.S and FNMSUB.S implemented as mentioned in [6]. Therefore, `-ffp-contract=off` is used while the compiling to tell the compiler to not use them.

Table 2.1 : Instructions of 'F' standard extension.

Instructions	Executing Operations
FLW, FSW	Loads/Stores Floating-point data to/from destination.
FMADD.S, FMSUB.S	Multiplies source1, source2, adds source3, stores in destination.
FNMADD.S, FNMSUB.S	Multiplies source1, source2, negates, adds source3, stores in destination.
FADD.S, FSUB.S, FMUL.S, FDIV.S	Adds/Subtracts/Multiplies/Divides source1, source2, stores in destination.
FSQRT.S	Computes square root of source1, stores in destination.
FSGNJ.S, FSGNJN.S, FSGNJX.S	Takes all bits from source1 except sign bit, which is determined by the sign of source2, the opposite sign of source2, or XOR of signs of source1 and source2, stores in destination.
FMIN.S, FMAX.S	Takes min/max of source1 and source2, stores in destination.
FCVT.W.S, FCVT.WU.S	Converts floating-point source1 value to signed/unsigned integer value, stores in destination.
FMV.X.W, FMV.W.X	Moves floating-point value from source1 to lower 32 bits of integer register destination, or vice versa.
FEQ.S, FLT.S, FLE.S	Equality/Less than/Less than or equal to of source1, source2, stores in destination.
FCLASS.S	Examines value in source1, stores 10-bit mask in destination that indicates class of floating-point number.
FCVT.S.W, FCVT.S.WU	Converts signed/unsigned source1 value to floating-point value, stores in destination.

2.4 About Spike and RISCV-DV

The main environment uses the Spike and RISCV-DV as core programs to compare the results to see if they works as expected and to generate random tests.

2.4.1 Spike

Spike is a software tool that simulates a RISC-V processor. It is called instruction set simulator. It runs RISC-V programs and shows how the processor works step by step. It is very useful for testing and debugging software before running it on real hardware.

Spike is often used by researchers and developers to check if their cores are working correctly. It supports different instruction sets. In this project, we use the RV32IMF instruction set since the Hornet was designed to be.

A sample command to run Spike is shown below:

```
spike --log-commits --isa=rv32imf --priv=M \
  -m0xf000:1,0x10000:0x8000,0x8010:1 -l \
  --log=spike.log ${TEST}.elf
```

This command starts Spike with the following settings:

- `-log-commits` records each instruction.
- `-isa=rv32imf` sets the instruction set to RV32IMF.
- `-priv=M` means machine mode is used.
- `-m...` defines memory regions for the simulation.
- `-l` enables logging to the terminal.
- `-log=spike.log` saves the log in a file.
- `$TEST.elf` is the test program to run.

2.4.2 RISCV-DV

RISCV-DV is a test generation tool for RISC-V processors. It creates random instruction tests to check if the processor behaves correctly. It is often used during processor design and verification. The tool specifically uses some corner cases that are hard to catch with normal tests.

RISCV-DV supports many RISC-V extensions, including the floating point F extension. It works with simulation tools like Spike. It is written in Python and can be run using the command line.

A basic command to run a test with RISCV-DV is:

```
run.py --verbose --test ${TEST} --simulator pyflow \
  --isa rv32imf --mabi ilp32f --sim_opts=""
```

This command does the following:

- `-verbose` prints more details while running.

- `-test $TEST` selects the test name.
- `-simulator pyflow` sets the simulator.
- `-isa rv32imf` sets the instruction set.
- `-mabi ilp32f` sets the memory ABI for floating point.
- `-sim_opts=""` sets simulator options (empty here).

After running this command, it generates the assembly code, compiles it, and runs the simulation. The results can then be checked for errors or unexpected behavior.

2.5 About Basics of Artificial Neural Networks

Neural networks are not taking a big part in this project. However, it was used as an example application to show that the project works as expected in Chapter 7.

2.5.1 Neural networks

Artificial Neural Networks are computational models that are designed to behave like the biological neural systems. They create a network with connected units called neurons. When neurons receives inputs, it applies a weighted matrix to it, and the result goes through a function called activation function. The result of the activation functions are given to the next layer.

Neural networks are widely used across the scientific community for the purposes involving pattern recognition, classification, regression and approximation of functions that are incapable to solve analytically. The neural networks can be as basic as just a few neurons to big complex multilayer networks.

Training of a neural network is done by calculating its weights using a dataset and a learning algorithm. Neural networks are powerful because they are able to learn from data and generalize these data without almost no problem. This is the reason why it is used widely.

2.5.2 Mathematical foundations

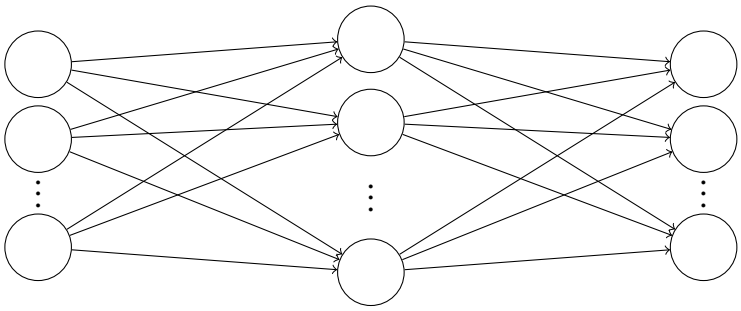


Figure 2.1 : Illustration of a basic neural network.

Assume a neural network with one hidden layer, which means that it has one layer between output and input layer. Such a neural network is called multilayer perceptron or MLP as it has more than 1 layers. It can be seen in Figure 2.1. Let:

- $\mathbf{x} \in \mathbb{R}^n$: input vector
- $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times n}$: weight matrix from input to hidden layer
- $\mathbf{b}^{(1)} \in \mathbb{R}^h$: bias vector for hidden layer
- $\phi(\cdot)$: activation function for hidden layer
- $\mathbf{W}^{(2)} \in \mathbb{R}^{m \times h}$: weight matrix from hidden to output layer
- $\mathbf{b}^{(2)} \in \mathbb{R}^m$: bias vector for output layer
- $f(\cdot)$: output activation function

The computations of the MLP are as follows:

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

$$\mathbf{a}^{(1)} = \phi(\mathbf{z}^{(1)})$$

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$$

$$\mathbf{y} = f(\mathbf{z}^{(2)})$$

The model is trained using gradient descent and backpropagation. The loss function $\mathscr{L}(\hat{\mathbf{y}}, \mathbf{y})$ measures the error between prediction and truth. The weights are updated using:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathscr{L}$$

where η is the learning rate.

2.5.3 Activation functions

There are a lot of activation functions used in the context of artificial networks. However, only two are used in this project named ReLU (rectified linear unit) and softmax function.

2.5.3.1 ReLU (rectified linear unit) function

The ReLU activation function is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

Its derivative is:

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

2.5.3.2 Softmax function

The Softmax function for a vector $\mathbf{z} = [z_1, z_2, \ldots, z_K]$ is defined as:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad \text{for } i = 1, 2, \ldots, K$$

Its derivative is:

$$\frac{\partial \text{Softmax}(z_i)}{\partial z_j} = \text{Softmax}(z_i)(\delta_{ij} - \text{Softmax}(z_j))$$

3. A UNIVERSAL VERIFICATION METHODOLOGY ENVIRONMENT

3.1 What is Universal Verification Methodology

In modern digital design, verifying a circuit before manufacturing is essential. The Universal Verification Methodology (UVM) is a widely used framework that helps engineers test complex hardware designs in a structured and reusable way. To understand how UVM works an 8-bit adder is verified.

UVM uses object-oriented programming (OOP) principles to create a testbench that interacts with the Device Under Test (DUT). Instead of manually writing multiple test cases, UVM provides an automated approach to generate inputs, monitor outputs, and compare results.

3.2 Basic UVM Testbench Structure

A UVM testbench consists of several key components, each responsible for a specific part of the verification process. The main modules and their functions are described below. A general architecture can be seen in Figure 3.1.

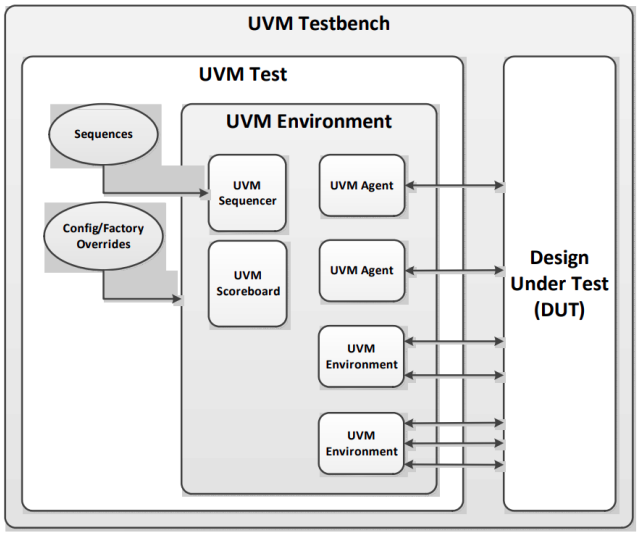


Figure 3.1 : General UVM architecture [2].

3.2.1 Sequence item and interface

In UVM-based verification, the interface (`adder_if`) and sequence item (`adder_tx`) are essential components that facilitate structured and efficient communication between the testbench and the DUT.

The interface (`adder_if`) acts as a connection between the DUT and the UVM testbench. It groups the DUT's input and output signals in a single construct, making it easier for multiple testbench components, such as the driver and monitor, to interact with the DUT without directly accessing its ports. The `adder_if` includes a clock signal (clk), a reset signal (`rst`), two 8-bit inputs (`x` and `y`), and a 9-bit output (`out`). Additionally, it contains a clocking block (`cb`) that ensures synchronized signal interaction. The clocking block specifies which signals are read and which signals are driven, ensuring that the testbench applies input data and captures output data at the correct time in the simulation.

```
import uvm_pkg::*;
`include "uvm_macros.svh"

interface adder_if(input bit clk);
    logic rst;
    logic [7:0] x;
    logic [7:0] y;
    logic [8:0] out;

    clocking cb @(posedge clk);
        input rst;
        input out;
        output x;
        output y;
    endclocking

endinterface
```

The sequence item (`adder_tx`) represents a single transaction that is sent to the DUT. In UVM, a transaction is an object that contains the necessary data to stimulate the DUT. The `adder_tx class extends uvm_sequence_item`, meaning it serves as the basic unit of communication between the sequencer and the driver. It defines two randomized 8-bit input values and a 9-bit output. The constraint block ensures that the inputs stay within the range of 0 to 255, preventing unexpected

behavior. Additionally, the `convert2str()` function formats the transaction data into a string.

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import tb_pkg::*;

class adder_tx extends uvm_sequence_item;
    `uvm_object_utils(adder_tx)

    rand bit [7:0] x, y;  //Inputs
    bit [8:0] out;        //Output

    //Constraining input values
    constraint valid_range {
        x inside {[0:255]};
        y inside {[0:255]};
    }

    function new(string name = "adder_tx");
        super.new(name);
    endfunction

    virtual function string convert2str();
        return $sformatf("x=%0d, y=%0d, sum=%0d", x, y, out);
    endfunction

endclass
```

The interface and sequence item work together in the UVM testbench to generate test scenarios and apply them to the DUT. The sequencer creates multiple `adder_tx` transactions, each containing random values of `x` and `y`. The driver retrieves these transactions from the sequencer and assigns them to the DUT's input signals using `adder_if`. After the DUT processes the inputs, the monitor captures the output using the interface and sends the transaction data to the scoreboard for verification.

By using an interface for signal communication and a sequence item for structured transactions, UVM enables efficient and reusable verification. This approach ensures that testbench components remain modular and can interact with the DUT without direct dependencies, leading to a more scalable and maintainable verification environment.

3.2.2 Sequencer

This UVM sequence (`gen_item_seq`) class is responsible for generating and sending transactions to the sequencer, which then forwards them to the driver for execution on the DUT. The class defines a random variable (`num`) that determines how many transactions will be generated, constrained between 10 and 50. In the `body()` task, the sequence creates transaction objects (`adder_tx`), randomizes their values, and sends them to the driver using `start_item()` and `finish_item()`. If randomization fails, an error message is logged. This kind of error checking is important in UVM since it makes the project easy to debug. Sequences are responsible for generating transaction-level stimuli, while the sequencer acts as a middleman, managing the flow of these transactions to the driver. If multiple sequences are used, the sequencer will schedule them based on arbitration policies, meaning transactions can be interleaved or prioritized depending on configuration. This allows for parallel testing scenarios, improving verification coverage.

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import tb_pkg::*;

class gen_item_seq extends uvm_sequence;
    `uvm_object_utils(gen_item_seq)

    function new(string name = "gen_item_seq");
        super.new(name);
    endfunction

    rand int num;

    constraint c1 {soft num inside {[10:50]};}

    //Body: Generates and randomizes multiple transactions
    virtual task body();
        for (int i = 0; i < num; i++) begin
            adder_tx m_item = adder_tx::type_id::create("m_item");
            start_item(m_item);

            if (!m_item.randomize())
             `uvm_error("SEQ", "Randomization failed for m_item!");

            `uvm_info("SEQ", $sformatf("Generate new item: %s",
            m_item.convert2str()), UVM_HIGH);
```

```
            finish_item(m_item);
        end
        `uvm_info("SEQ", $sformatf("Done generation of %0d items",
        num), UVM_LOW);
    endtask
endclass
```

3.2.3 Driver

The UVM driver class is responsible for receiving transactions from the sequencer and applying them to the DUT through the virtual interface (`vif`). The `build_phase()` retrieves the virtual interface from the UVM configuration database and reports a fatal error if it is not found. The `run_phase()` runs indefinitely, waiting for transactions from the sequencer using `get_next_item()`, then calls `drive_item()` to apply the input values (`x` and `y`) to the DUT. The `drive_item()` task synchronizes with the DUT's interface using the clocking block (`vif.cb`) and assigns the transaction values to the interface signals. This setup ensures that the generated test data flows correctly from the sequencer to the DUT, enabling effective stimulus application.

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import tb_pkg::*;

class driver extends uvm_driver #(adder_tx);
    `uvm_component_utils(driver)

    function new(string name = "driver", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    virtual adder_if vif;

    /*Build phase: Retrieves the virtual interface from
    the UVM configuration database*/
    virtual function void build_phase(uvm_phase phase);
        if (!uvm_config_db#(virtual adder_if)::get(this, "",
        "adder_vif", vif))
            `uvm_fatal("DRV", "Could not get vif");
    endfunction

    //Run phase: Waits for transactions and sends them to the DUT
    virtual task run_phase(uvm_phase phase);
```

```
        super.run_phase(phase);
        forever begin
            adder_tx m_item;
            `uvm_info("DRV",
            $sformatf("Wait for item from sequencer"), "UVM_HIGH")
            seq_item_port.get_next_item(m_item);
            drive_item(m_item);
            seq_item_port.item_done();
        end
    endtask

    /*Drive item: Applies transaction values to the DUT
    through the virtual interface*/
    virtual task drive_item(adder_tx m_item);
        @(vif.cb);
        vif.cb.x <= m_item.x;
        vif.cb.y <= m_item.y;
    endtask
endclass
```

3.2.4 Monitor

The UVM monitor class is responsible for observing the DUT signals and forwarding them to other components, such as the scoreboard, for verification. The `build_phase()` retrieves the virtual interface (`vif`) from the UVM configuration database and initializes the analysis port (`mon_analysis_port`), which is used to send collected transactions to other components. The `run_phase()` runs continuously, waiting for DUT signal changes using the clocking block (`vif.cb`). It captures the input values and the output from the DUT, creates a new transaction object (`adder_tx`), and sends it through the analysis port. If reset is active, it skips capturing data. This ensures that every transaction processed by the DUT is observed and logged, enabling the scoreboard to check for correctness.

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import tb_pkg::*;

class monitor extends uvm_monitor;
    `uvm_component_utils(monitor)

    function new(string name="monitor", uvm_component parent=null);
```

```
        super.new(name, parent);
    endfunction

    uvm_analysis_port #(adder_tx) mon_analysis_port;
    virtual adder_if vif;

    /*Build phase: Retrieves the virtual interface
    and initializes the analysis port*/
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if (!uvm_config_db#(virtual adder_if)::get(this, "",
        "adder_vif", vif))
            `uvm_fatal("MON", "Could not get vif")
        mon_analysis_port = new ("mon_analysis_port", this);
    endfunction

    /*Run phase: Monitors DUT signals and sends
    transactions through the analysis port*/
    virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);
        forever begin
            @ (vif.cb);
                if(!vif.rst) begin
                    adder_tx item=adder_tx::type_id::create("item");
                    item.x = vif.x;
                    item.y = vif.y;
                    item.out = vif.cb.out;
                    mon_analysis_port.write(item);
                    `uvm_info("MON", $sformatf("Saw item %s",
                    item.convert2str()), UVM_HIGH)
                end
        end
    endtask
endclass
```

3.2.5 Agent

The UVM agent class in this code serves as a container for the driver (`d0`), monitor (`m0`), and sequencer (`s0`). The agent is responsible for handling the transaction flow between the sequencer and the DUT. The `build_phase()` is responsible for creating instances of the sequencer, monitor, and driver using the UVM factory. The `connect_phase()` ensures that the driver is connected to the sequencer, allowing the driver to receive transaction items generated by sequences. This setup ensures that

input data flows correctly from the sequencer to the driver, while the monitor observes the DUT behavior and sends transaction data to other testbench components.

Multiple agents are used when the DUT has multiple interfaces or communication channels that need independent verification. Each agent is responsible for managing transactions on a specific interface, ensuring modularity and scalability in the testbench. Agents contain a sequencer, driver, and monitor, which work together to generate, apply, and observe transactions.

When multiple agents are present, each agent operates independently, processing its own set of transactions. The sequencer in each agent generates stimulus, and the driver applies these stimuli to different parts of the DUT. Meanwhile, the monitors capture DUT responses and send them to the scoreboard for validation. If the agents interact with shared DUT resources, arbitration mechanisms may be required to handle potential conflicts.

Having multiple agents allows the testbench to verify parallel data paths, master-slave architectures, or multi-protocol communication. For example, in a bus-based design, one agent may act as a master generating read/write requests, while another agent acts as a slave responding to those transactions. This enables a realistic verification of complex DUT behaviors, ensuring that different functional blocks operate correctly together.

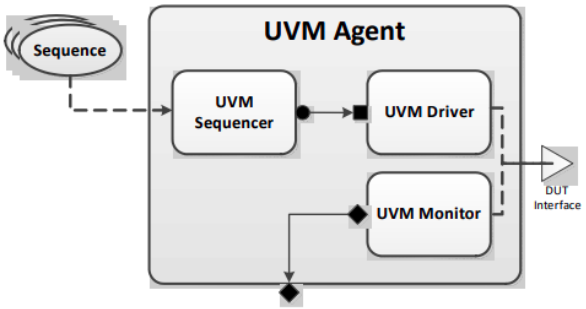


Figure 3.2 : Agent structure [2].

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import tb_pkg::*;

class agent extends uvm_agent;
    `uvm_component_utils(agent)
```

```
    function new(string name = "agent", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    driver d0;
    monitor m0;
    uvm_sequencer #(adder_tx) s0;

    //Build phase: Creates instances of the sequencer,
    monitor, and driver*/
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        s0 = uvm_sequencer#(adder_tx)::type_id::create("s0", this);
        m0 = monitor::type_id::create("m0", this);
        d0 = driver::type_id::create("d0", this);
    endfunction

    //Connect phase: Connects the driver to the sequencer
    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        d0.seq_item_port.connect(s0.seq_item_export);
    endfunction
endclass
```

3.2.6 Scoreboard

The UVM scoreboard class is responsible for checking the correctness of DUT outputs by comparing them with expected values. The `build_phase()` initializes the analysis import port (`m_analysis_imp`), which receives transaction data from the monitor. The write() function computes the expected output (expected_value) by adding `x` and `y` from the transaction and stores it in a queue (`exp_out_queue`). Since the DUT has a 1-clock delay, the scoreboard waits until at least one previous expected value is available before comparing the actual output (`item.out`) with the delayed expected output (`delayed_exp_out`). If the values match, a PASS message is logged; otherwise, an ERROR message is generated. This ensures that verification accounts for timing delays in the DUT, preventing incorrect failures due to mismatched comparisons.

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import tb_pkg::*;

class scoreboard extends uvm_scoreboard;
  `uvm_component_utils(scoreboard)

  function new(string name="scoreboard", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    bit [8:0] exp_out_queue[$];
    bit [8:0] delayed_exp_out;

    uvm_analysis_imp #(adder_tx, scoreboard) m_analysis_imp;

    //Build phase: Initializes the analysis import port
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        m_analysis_imp = new("m_analysis_imp", this);
    endfunction

    /*write: Receives transactions, computes expected values,
    and checks correctness*/
    virtual function write(adder_tx item);
        bit [8:0] expected_value = item.x + item.y;

        exp_out_queue.push_back(expected_value);

        if (exp_out_queue.size() > 1) begin
            delayed_exp_out = exp_out_queue.pop_front();

            `uvm_info("SCBD", $sformatf("x=%d y=%d out=%d
            exp_out=%d", item.x, item.y, item.out, delayed_exp_out),
            UVM_LOW)

            if (item.out == delayed_exp_out) begin
                `uvm_info("SCBD", $sformatf("PASS ! out=%0d,
                exp=%0d", item.out, delayed_exp_out), UVM_LOW)
            end else begin
                `uvm_error("SCBD", $sformatf("ERROR ! out=%0d
                exp=%0d", item.out, delayed_exp_out))
            end
        end
    endfunction
endclass
```

3.2.7 Environment

This UVM environment (`env`) class is a key part of the testbench, serving as a container for the agent (`a0`) and scoreboard (`sb0`). The agent is responsible for driving and monitoring transactions, while the scoreboard checks if the DUT outputs are correct. The `build_phase()` creates instances of the agent and scoreboard using the UVM factory. The `connect_phase()` ensures that the monitor inside the agent (`a0.m0`) is connected to the scoreboard, allowing it to receive and compare transactions. This setup ensures that the verification environment collects data, compares results, and reports errors if the DUT's behavior does not match expectations.

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import tb_pkg::*;

class env extends uvm_env;
    //Register this class with the UVM factory
    `uvm_component_utils(env)

    function new(string name = "env", uvm_component parent=null);
        super.new(name, parent);
    endfunction

    agent a0;
    scoreboard sb0;

    //Build phase: Create instances of the agent and sb
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        a0 = agent::type_id::create("a0", this);
        sb0 = scoreboard::type_id::create("sb0", this);
    endfunction

    //Connect phase: Connect the monitor's analysis port to the sb
    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        a0.m0.mon_analysis_port.connect(sb0.m_analysis_imp);
    endfunction
endclass
```

3.2.8 Test

This UVM testbench code defines a base test (`base_test`) and a derived test (`adder_test`) to verify an adder module. The `base_test class extends uvm_test` and serves as the foundation for all tests. It includes an environment (`env e0`), a sequence object (`seq`), and a virtual interface (`vif`) for interacting with the DUT.

The `build_phase()` creates the test environment (`e0`), retrieves the adder interface (`adder_if`) from the UVM configuration database, and randomizes the sequence (`seq`). If any of these steps fail, a fatal error is reported.

The `run_phase()` raises an objection (preventing the test from ending immediately), calls `apply_reset()` to reset the DUT, starts the sequence on the sequencer (`e0.a0.s0`), waits 200 time units, and then drops the objection, allowing the test to end.

The `apply_reset()` task asserts reset, initializes input signals (x and y to 0), waits for five clock cycles, then deasserts reset, and waits for ten more clock cycles.

The `adder_test class extends base_test` and modifies the `build_phase()` to ensure the sequence is randomized within a specific range (num between 300 and 500). If this randomization fails, the test reports a fatal error.

This testbench structure allows for reusable and automated verification of the adder module. The code can be seen in the following:

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import tb_pkg::*;

//Base test class for verifying the adder module
class base_test extends uvm_test;
    //Register the test class with the UVM factory
    `uvm_component_utils(base_test)

    function new(string name="base_test",uvm_component parent=null);
        super.new(name, parent);
    endfunction

    //Environment that contains all UVM components
    env e0;
    //Sequence object to generate test inputs
    gen_item_seq seq;
```

```
    //Virtual interface to interact with the DUT
    virtual adder_if vif;

    /*Build phase: Initializes the environment,
    gets the interface, and creates the sequence*/

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        e0 = env::type_id::create("e0", this);

        if (!uvm_config_db#(virtual adder_if)::get(this, "",
        "adder_vif", vif))
            `uvm_fatal("TEST", "Did not get vif");

        uvm_config_db#(virtual adder_if)::set(this, "e0.a0.*",
        "adder_vif", vif);

        seq = gen_item_seq::type_id::create("seq");
        if (seq == null)
            `uvm_fatal("TEST", "Sequence creation failed");

        if (!seq.randomize())
            `uvm_fatal("TEST", "Randomization failed for seq");
    endfunction

    //Run phase: Applies reset and runs the sequence
    virtual task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        apply_reset();
        seq.start(e0.a0.s0);
        #200;
        phase.drop_objection(this);
    endtask

    //Apply reset sequence to the DUT
    virtual task apply_reset();
        vif.rst <= 1;
        vif.x <= 0;
        vif.y <= 0;
        repeat(5) @ (posedge vif.clk);
        vif.rst <= 0;
        repeat(10) @ (posedge vif.clk);
    endtask
endclass

//Derived test class that extends base_test
class adder_test extends base_test;
    `uvm_component_utils(adder_test)
```

```
    function new(string name="adder_test",uvm_component parent=null);
        super.new(name, parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        if (!seq.randomize() with { num inside {[300:500]}; })
            `uvm_fatal("TEST", "Randomization of seq.num failed!");
    endfunction
endclass
```

3.2.9 Testbench

This UVM testbench (`tb`) is designed to verify the functionality of an adder module using UVM methodology. The testbench first defines a clock signal and toggles it every 10 ns using an always block. The interface (`adder_if`) is instantiated and connected to the DUT, which takes `clk`, `rst`, `X`, and `Y` as inputs and produces OUT as an output.

In the initial block, the clock is initialized to 0, and the virtual interface (_if) is registered in the UVM configuration database using `uvm_config_db::set()`, allowing UVM testbench components (like the driver and monitor) to access it. The simulation starts by calling `run_test("adder_test")`, which triggers the execution of the UVM test. This setup ensures that the DUT operates in a controlled and reusable test environment, where the UVM framework handles stimulus generation, signal monitoring, and result checking.

```
`timescale 1ns/1ps

module tb;

    import uvm_pkg::*;
    `include "uvm_macros.svh"
    import tb_pkg::*;

    reg clk;

    always #10 clk = ~clk;
    adder_if _if(clk);
```

```
    adder u0 (.clk(clk),
    .rst(_if.rst),
    .X(_if.x),
    .Y(_if.y),
    .OUT(_if.out));

    initial begin
        clk <= 0;
        uvm_config_db#(virtual adder_if)::set(null, "uvm_test_top",
        "adder_vif", _if);
        run_test("adder_test");
    end
endmodule
```

3.2.10 Adder module

Basic Ripple Carry Adder used for test:

```
module adder(
    input [7:0] X,
    input [7:0] Y,
    input clk,
    input rst,
    output reg [8:0] OUT
);

    wire [7:0] w;
    wire [8:0] OUT_tmp;

    full_adder FA0(
        .A(X[0]),
        .B(Y[0]),
        .Cin(1'b0),
        .S(OUT_tmp[0]),
        .Cout(w[0])
    );

    full_adder FA1(
        .A(X[1]),
        .B(Y[1]),
        .Cin(w[0]),
        .S(OUT_tmp[1]),
        .Cout(w[1])
    );
```

```
    full_adder FA2(
        .A(X[2]),
        .B(Y[2]),
        .Cin(w[1]),
        .S(OUT_tmp[2]),
        .Cout(w[2])
    );

    full_adder FA3(
        .A(X[3]),
        .B(Y[3]),
        .Cin(w[2]),
        .S(OUT_tmp[3]),
        .Cout(w[3])
    );

    full_adder FA4(
        .A(X[4]),
        .B(Y[4]),
        .Cin(w[3]),
        .S(OUT_tmp[4]),
        .Cout(w[4])
    );

    full_adder FA5(
        .A(X[5]),
        .B(Y[5]),
        .Cin(w[4]),
        .S(OUT_tmp[5]),
        .Cout(w[5])
    );

    full_adder FA6(
        .A(X[6]),
        .B(Y[6]),
        .Cin(w[5]),
        .S(OUT_tmp[6]),
        .Cout(w[6])
    );

    full_adder FA7(
        .A(X[7]),
        .B(Y[7]),
        .Cin(w[6]),
        .S(OUT_tmp[7]),
        .Cout(w[7])
    );

    assign OUT_tmp[8] = w[7];
```

```
    always @(posedge clk) begin
        if(rst) OUT <= 8'b00000000;
        else OUT <= OUT_tmp;
    end

endmodule
```

Error putted in the last stage of full adder chain:

```
module adder(
    input [7:0] X,
    input [7:0] Y,
    input clk,
    input rst,
    output reg [8:0] OUT
);

    wire [7:0] w;
    wire [8:0] OUT_tmp;

    full_adder FA0(
        .A(X[0]),
        .B(Y[0]),
        .Cin(1'b0),
        .S(OUT_tmp[0]),
        .Cout(w[0])
    );

    full_adder FA1(
        .A(X[1]),
        .B(Y[1]),
        .Cin(w[0]),
        .S(OUT_tmp[1]),
        .Cout(w[1])
    );

    full_adder FA2(
        .A(X[2]),
        .B(Y[2]),
        .Cin(w[1]),
        .S(OUT_tmp[2]),
        .Cout(w[2])
    );

    full_adder FA3(
        .A(X[3]),
```

```
        .B(Y[3]),
        .Cin(w[2]),
        .S(OUT_tmp[3]),
        .Cout(w[3])
    );

    full_adder FA4(
        .A(X[4]),
        .B(Y[4]),
        .Cin(w[3]),
        .S(OUT_tmp[4]),
        .Cout(w[4])
    );

    full_adder FA5(
        .A(X[5]),
        .B(Y[5]),
        .Cin(w[4]),
        .S(OUT_tmp[5]),
        .Cout(w[5])
    );

    full_adder FA6(
        .A(X[6]),
        .B(Y[6]),
        .Cin(w[5]),
        .S(OUT_tmp[6]),
        .Cout(w[6])
    );

    full_adder FA7(
        .A(X[7]),
        .B(Y[7]),
        .Cin(1'b0),
        .S(OUT_tmp[7]),
        .Cout(w[7])
    );

    assign OUT_tmp[8] = w[7];

    always @(posedge clk) begin
        if(rst) OUT <= 8'b00000000;
        else OUT <= OUT_tmp;
    end

endmodule
```

3.3 UVM Factory

The UVM Factory is a centralized object creation mechanism in UVM. It allows testbench components such as drivers, monitors, agents, environments, and sequences to be dynamically created at runtime instead of being instantiated manually in the code. This makes the testbench more flexible and reusable, as different configurations can be applied without modifying the source code.

In the adder verification testbench, the factory is used to create various UVM components dynamically. For example, an agent, monitor, and driver are instantiated using the `type_id::create("instance_name", parent)` method. This ensures that these components are correctly registered with the UVM framework and can be overridden if needed.

Example usage in the adder testbench:

```
env e0;
e0 = env::type_id::create("e0", this);
```

This method ensures that the environment (**env**) can be replaced with different implementations if required, making the verification environment scalable and configurable.

3.4 Basic UVM Macros Used in Adder Verification

UVM provides several macros to simplify testbench development. Below are some of the most basic macros used in the adder verification:

3.4.1 uvm_component_utils

This macro registers a UVM component with the factory so it can be created dynamically. It is required in every UVM component class.

Example from the agent:

```
class agent extends uvm_agent;
    `uvm_component_utils(agent)
```

Without this macro, the factory cannot create instances of the agent class.

3.4.2 uvm_object_utils

Similar to `uvm_component_utils`, this macro registers UVM objects (such as sequence items) with the factory. Unlike components, sequence items (`uvm_sequence_item`) do not have a hierarchy.

Example from the sequence item:

```
class adder_tx extends uvm_sequence_item;
    `uvm_object_utils(adder_tx)
```

This enables randomization and dynamic creation of transactions.

3.4.3 uvm_fatal

This macro reports a fatal error and stops the simulation immediately if a critical issue occurs. It is used in cases where the testbench cannot proceed.

Example from the driver:

```
if (!uvm_config_db#(virtual adder_if)::get(this, "",
"adder_vif", vif))
    `uvm_fatal("DRV", "Could not get vif");
```

If the virtual interface (vif) is not set up correctly, the testbench will halt with an error.

3.4.4 uvm_info

This macro is used for logging informational messages during simulation. It helps in debugging by providing insights into the testbench operation.

Example from the monitor:

```
`uvm_info("MON", $sformatf("Saw item %s",
item.convert2str()), UVM_HIGH)
```

This logs transactions observed by the monitor, which helps in tracking the values sent to the DUT.

3.4.5 uvm_error

This macro is used to report an error in the testbench but does not stop the simulation immediately like `uvm_fatal`. It is useful for detecting incorrect DUT outputs while allowing further execution.

Example from the scoreboard:

```
if (item.out != delayed_exp_out)
    `uvm_error("SCBD", $sformatf("ERROR ! out=%0d exp=%0d",
    item.out, delayed_exp_out))
```

If the DUT output does not match the expected value, this error will be reported, but the test will continue running.

3.5 UVM Testbench Results

The UVM testbench starts with a text in the Figure 3.3.

```
VSIM 2> run -all
# ----------------------------------------------------------------
# UVM-1.1d
# (C) 2007-2013 Mentor Graphics Corporation
# (C) 2007-2013 Cadence Design Systems, Inc.
# (C) 2006-2013 Synopsys, Inc.
# (C) 2011-2013 Cypress Semiconductor Corp.
# ----------------------------------------------------------------
#
#   **********       IMPORTANT RELEASE NOTES         ************
#
#   You are using a version of the UVM library that has been compiled
#   with `UVM_NO_DEPRECATED undefined.
#   See http://www.eda.org/svdb/view.php?id=3313 for more details.
#
#   You are using a version of the UVM library that has been compiled
#   with `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR undefined.
#   See http://www.eda.org/svdb/view.php?id=3770 for more details.
#
#       (Specify +UVM_NO_RELNOTES to turn off this notice)
#
```

Figure 3.3 : UVM testbench start.

For the working adder module the `PASS` messages are in the Figure 3.4.

```
# UVM_INFO adder_scoreboard.sv(31) @ 330000: uvm_test_top.e0.sb0 [SCBD] x= 44 y=135 out= 94 exp_out= 94
# UVM_INFO adder_scoreboard.sv(34) @ 330000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=94, exp=94
# UVM_INFO adder_scoreboard.sv(31) @ 350000: uvm_test_top.e0.sb0 [SCBD] x=131 y=165 out=179 exp_out=179
# UVM_INFO adder_scoreboard.sv(34) @ 350000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=179, exp=179
# UVM_INFO adder_scoreboard.sv(31) @ 370000: uvm_test_top.e0.sb0 [SCBD] x=243 y=163 out=296 exp_out=296
# UVM_INFO adder_scoreboard.sv(34) @ 370000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=296, exp=296
# UVM_INFO adder_scoreboard.sv(31) @ 390000: uvm_test_top.e0.sb0 [SCBD] x=252 y=203 out=406 exp_out=406
# UVM_INFO adder_scoreboard.sv(34) @ 390000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=406, exp=406
# UVM_INFO adder_scoreboard.sv(31) @ 410000: uvm_test_top.e0.sb0 [SCBD] x= 74 y=180 out=455 exp_out=455
# UVM_INFO adder_scoreboard.sv(34) @ 410000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=455, exp=455
# UVM_INFO adder_scoreboard.sv(31) @ 430000: uvm_test_top.e0.sb0 [SCBD] x=108 y=123 out=254 exp_out=254
# UVM_INFO adder_scoreboard.sv(34) @ 430000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=254, exp=254
# UVM_INFO adder_scoreboard.sv(31) @ 450000: uvm_test_top.e0.sb0 [SCBD] x= 43 y= 15 out=231 exp_out=231
# UVM_INFO adder_scoreboard.sv(34) @ 450000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=231, exp=231
# UVM_INFO adder_scoreboard.sv(31) @ 470000: uvm_test_top.e0.sb0 [SCBD] x=155 y=190 out= 58 exp_out= 58
# UVM_INFO adder_scoreboard.sv(34) @ 470000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=58, exp=58
```

Figure 3.4 : `PASS` messages coming from the scoreboard.

When the working adder module is given the results are in the Figure 3.5.

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  881
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [SCBD]   876
# [SEQ]     1
# [TEST_DONE]     1
# ** Note: $finish    : /sw/mentor/questasim/2019.4_1/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 8850200 ps  Iteration: 54  Instance: /tb
```

Figure 3.5 : UVM report summary for the correct adder module.

When the incorrect adder module is given to the testbench the scoreboard shows errors like in the Figure 3.6.

```
# UVM_INFO adder_scoreboard.sv(31) @ 8570000: uvm_test_top.e0.sb0 [SCBD] x=249 y= 63 out=362 exp_out=362
# UVM_INFO adder_scoreboard.sv(34) @ 8570000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=362, exp=362
# UVM_INFO adder_scoreboard.sv(31) @ 8590000: uvm_test_top.e0.sb0 [SCBD] x= 12 y=170 out=184 exp_out=312
# UVM_ERROR adder_scoreboard.sv(36) @ 8590000: uvm_test_top.e0.sb0 [SCBD] ERROR ! out=184 exp=312
# UVM_INFO adder_scoreboard.sv(31) @ 8610000: uvm_test_top.e0.sb0 [SCBD] x= 88 y=192 out=182 exp_out=182
# UVM_INFO adder_scoreboard.sv(34) @ 8610000: uvm_test_top.e0.sb0 [SCBD] PASS ! out=182, exp=182
```

Figure 3.6 : `ERROR` messages coming from the scoreboard.

The incorrect adder module's UVM report summary is in the Figure 3.7.

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  665
# UVM_WARNING :    0
# UVM_ERROR :  216
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [SCBD]   876
# [SEQ]     1
# [TEST_DONE]     1
# ** Note: $finish    : /sw/mentor/questasim/2019.4_1/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 8850200 ps  Iteration: 54  Instance: /tb
```

Figure 3.7 : UVM report summary for the incorrect adder module.

4. IBEX VERIFICATION ENVIRONMENT

4.1 Getting the Ibex Environment Running

4.1.1 Initial attempt

We start by setting up the verification environment according to Ibex's documentation [3]:

As described in the documentation, the following lines are added to the `.bashrc` file, to have them set automatically at system startup:

```
export PATH=$PATH:/opt/riscv/bin
export RISCV=/opt/riscv/
export RISCV_GCC=/opt/riscv/bin/riscv32-unknown-elf-gcc
export RISCV_TOOLCHAIN=$RISCV
export RISCV_OBJCOPY=/opt/riscv/bin/riscv32-unknown-elf-objcopy
export SPIKE_PATH_=/opt/riscv/bin/spike
export PKG_CONFIG_PATH=/opt/riscv/lib/pkgconfig
```

Then, as stated in the documentation, the command `make` was run from the directory `./ibex/dv/uvm/core_ibex/`. This makefile calls a Python script found in the directory `./ibex/dv/uvm/core_ibex/scripts/`.

Error messages were encountered, as the libraries used in this script weren't installed. Since the documentation did not provide a list of requirements, the missing libraries were installed one by one, until the error messages went away.

After installing all the required packages, the following error message appeared:

```
RuntimeError: Failed to find ['riscv-riscv', \\
'riscv-disasm', 'riscv-fdt', 'riscv-fesvr'] \\
pkg-config packages.
Did you set the PKG_CONFIG_PATH correctly?
```

After checking, it was found that this package wasn't installed in our system. Assuming this was a possible misconfiguration during the compilation of thee Spike ISA Simulator, it was recompiled with the commands:

```
./configure --prefix=$RISCV
make -j 12
```

This didn't alleviate the problem, as the required `riscv-fdt.pc` file was still missing. As a temporary fix, the requirement of this file was removed from the `compile_tb.py` python script.

This led to the next error: `No such file 'xrun'`. This is the command to run Cadence's Xcelium simulator. As was described in the documentation, this environment requires a SystemVerilog simulator that supports UVM. These were listed in the Ibex documentation as Synopsys VCS, Cadence Xcelium and Questa. This requirement is the source of our technical problems.

4.1.2 Trying to obtain a working environment

Cadence Xcelium was available to use, via the Embedded Systems Design Lab's server. But, since the license was borrowed from the VLSI Lab, there were issues accessing the software. While the system administrators were trying to solve the problem, we decided to set up the other requirements on the server, namely the RISC-V Toolchain and Spike ISA Simulator.

The RISC-V Toolchain source code [10] was downloaded using `git clone`, and compiled with:

```
    ./configure --prefix=/opt/riscv --with-abi=ilp32f \\
    --with-arch=rv32imf_zicsr_zicntr
    make -j 12
```

This resulted in an error, since the user account assigned to us in the server did not have access to the folder `/opt/riscv`. To bypass this issue, we decided to install the toolchain to the user directory instead, by modifying the `--prefix` parameter. Compiling with these parameters resulted in several errors, caused by git commands. After checking, it was realized that the version of git used in the server, which is

running CentOS 7, is 11 years out of date. This version lacked the `git submodule` command the makefile used. This command allows the compilation to download the additional modules as needed. To not use this command, the repository can be cloned with `git clone --recursive`, which downloads all the required modules.

Even this isn't enough to solve the problem though, since the other tools in the server, mainly gcc, are also 11 years out of date. The RISC-V Toolchain is not compatible with build environments that are this old, so we were unable to obtain these tools on the server. Other methods, such as compiling the tools on our personal computers and copying them, or downloading other pre-compiled binaries were tested. Since these both depend on the host system to have up-to-date libraries, they too were unable to be run on the server.

Obtaining up-to-date compilation tools on the server might require a whole system upgrade, so it wasn't possible during the academic term. Another option is obtaining a Questa license from ElectraIC, the company aiding us with the project. That has also been delayed due to schedule issues, therefore we decided to focus on fixing Hornet instead.

4.2 Setting Up and Analyzing the Environment

By utilizing a server equipped with Questa Sim and RISC-V tools, access to which was provided by Electra IC, the Ibex Verification Environment was successfully run. Before doing so, however, making several changes to the configuration files were required. Before discussing the changes, it is beneficial to briefly mention the files most utilized within this chapter with Table 4.1.

4.2.1 Setup of the environment

In the current versions of the Ibex Verification Environment [3], the verification flow utilizes co-simulation, in which the ISS and RTL simulations are run synchronized with each other. Because of this reason, the core trace log based verification method, which we aim to utilize, is absent. For this reason, an older branch of the environment, modified to accept the Pulpissimo RISC-V core [11], will be utilized. This branch still contains all the trace log comparison structure required.

Table 4.1 : File descriptions and their purposes.

File	Purpose
`Makefile`	Defines ISA, ABI, simulation flow, and toolchain parameters
`rtl_simulation.yaml`	Command line parameters for various RTL simulators used to simulate the core under test. These parameters control simulation behavior and debugging options.
`simulator.yaml`	Configuration used by RISCV-DV [4] to compile the instruction generator and generate random instructions. Similar to but separate from `rtl_simulation.yaml`.
`testlist.yaml`	Comprehensive list of tests and their generation parameters, including test weights and constraints.
`riscv_core_setting.sv`	SystemVerilog file defining core capabilities including supported ISA extensions and register count.
`ibex_asm_program_gen.sv`	Generates assembly program headers including various directives like `.option` flags and memory initialization sequences.
`core_ibex_tb_top.sv`	Top-level testbench module that instantiates and connects the DUT (Design Under Test) with all test interfaces and verification components.
`ibex_dv.f`	Simulation file list for Questa simulator, containing all required design and verification files.

To get this environment working with the Questa sim, several fixes to the aforementioned files are required. These are:

- In `ibex/dv/uvm/core_ibex/Makefile`, changing the ISA parameter from `rv32imcb` to `rv32imc`
- In `ibex/dv/uvm/core_ibex/yaml/rtl_simulation.yaml`, adding several parameters to Questa launch options, to point to the UVM source files. To be specific, changing line 76 from

```
vsim -64 -c <cov_opts> -do "run -a; quit -f" +designfile
-f <out>/top.list <sim_opts>  -sv_seed <seed> +access +r+w
+UVM_TESTNAME=<rtl_test> +bin=<binary>
```

```
    +ibex_tracer_file_base="<sim_dir>/trace_core"
    -l <sim_dir>/sim.log
```

to

```
    vsim -64 -c <cov_opts> -t 1ps -sv_lib
    $UVM_HOME/linux_x86_64/uvm_dpi -L \$UVM_HOME
    -dpicpppath /usr/bin/gcc -do "run -a; quit -f"
    +designfile -f <out>/top.list <sim_opts>  -sv_seed <seed>
    +access +r+w +UVM_TESTNAME=<rtl_test> +bin=<binary>
    +ibex_tracer_file\base="<sim_dir>/trace_core"
    -l <sim_dir>/sim.log
```

- Similar additions to above were also done in `ibex/vendor/google_riscv-dv/yaml/simulator.yaml`.
- In `ibex/rtl/ibex_core.sv`, instantiating the `prim_clock_gating` module instead of the `tc_clk_gating`.
- Finally, in

  ```
  ibex/vendor/google_riscv-dv/scripts/spike_log_to_trace_csv.py
  ```

  , the regex for parsing the log into a csv file was malformed, requiring a fix. Doing this by hand would be complicated, but the fix was already available on the master branch of the github repo, so it was pulled from there.

After applying these fixes, and running the environment with

```
make TEST=riscv_rand_instr_test ITERATIONS=1
```

, the test is successfully completed as it can be seen in 4.1

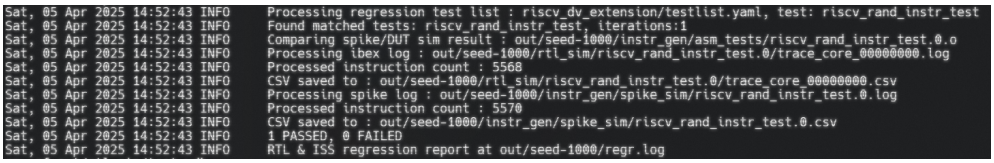


Figure 4.1 : Terminal output for successful Ibex Verification.

As a default, the ISA is defined as `rv32imc` in the Makefile. But, in Hornet, there's also the F extension, and there's also no support for the C extension (compressed instructions). The configuration files should thus be changed accordingly.

The addition of the F extension also necessitates changing the RISC-V ABI from `ilp32` to `ilp32f`. There was no parameter in the Makefile for the ABI, so it was added and appended as a parameter to the necessary places. It's kept as `ilp32` for now.

To try to remove the C extension, the ISA is changed to `rv32im` in the Makefile, and the environment is run again. The result is a failure, which is unexpected since the existing Ibex core should support this ISA. When the generated assembly files are analyzed, it's quickly seen that despite setting the ISA to `rv32im`, compressed instructions are still being generated. But as Spike [7] correctly runs with `rv32im`, it doesn't recognize these instructions and sends an `illegal_instruction` CSR trap. This causes the logs to differ, and results in a failure.

To fix this issue, the `riscv_dv_extension/riscv_core_setting.sv` file should be modified. This file defines the instruction extensions supported by the core. When `RV32C` is removed from this list, RISCV-DV stops putting compressed instructions in assembly files.

Despite this, when inspecting the binary file compiled from these assembly files, the compressed instructions can still be seen. To alleviate this, the ISA parameter is also added to places in the Makefile where RISCV-DV calls the gcc compiler. This does not fix the issue. As a sanity check, the ISA parameter is kept as `rv32im`, but gcc is called with `rv32i`. This causes gcc to give an error, stating it doesn't recognize some of the functions. This shows that gcc correctly uses the parameters given to it, so the issue must be elsewhere.

In `riscv_dv_extension/ibex_asm_program_gen.sv`, by utilizing SystemVerilog classes defined elsewhere in the environment, the header part of the assembly code is generated. Within this file, it's seen that the line `.option rvc` is added to the header. This likely instructs the compiler to use compressed instructions wherever possible. Removing this line finally solves the problem, and the environment runs successfully when ISA is set to `rv32im`.

If at any point, compressed instructions are added, and it's desired to re-enable this feature, simply adding `RV32C` back in `riscv_core_setting.sv` causes them to re-appear, regardless of the ISA parameter. It can then be noted that the ISA parameter mostly affects Spike.

Next, floating point instruction generation is tested. After setting ISA to `rv32imf` and ABI to `ilp32f`, the environment is ran again. The result is a success, which is unexpected, because the core under test (Ibex) does not support floating point instructions. Indeed, when the generated assembly is checked, there aren't any floating point instructions. Adding `RV32F` to `riscv_core_setting.sv` also doesn't cause any difference.

It's clear that a new test for floating point instructions must be written. Like all other tests, the new test should be included within the `testlist.yaml` file. When inspecting the environment files, it's found out that RISCV-DV includes some example tests that can be utilized. These files reside in `ibex/vendor/google_riscv-dv/target`. One that is of particular note is a test under `target/rm32imfdc`:

```
- test: riscv_floating_point_arithmetic_test
  description: >
    Enable floating point instructions
  gen_opts: >
    +instr_cnt=10000
    +num_of_sub_program=0
    +no_fence=1
    +no_data_page=1
    +no_branch_jump=1
    +enable_floating_point=1
    +boot_mode=m
  iterations: 1
  gen_test: riscv_instr_base_test
  rtl_test: core_base_test
```

This can directly be copied into the `testlist.yaml`, with some minor changes. Like all other tests in the file, the `rtl_test` parameter should be changed to `core_ibex_base_test`. The critical difference in this test is simply the addition of `+enable_floating_point=1` under `gen_opts`.

When the environment is run with

```
make TEST=riscv_floating_point_arithmetic_test ITERATIONS=1
```

, the test results in failure, as expected. The generated assembly code also includes floating point operations. One thing to note is that these also include the fused multiply/add instructions, which Hornet currently doesn't support. This can be handled by utilizing methods previously mentioned in chapter 2.

4.2.2 Analyzing the environment

Next, it would be beneficial to understand how the environment works step by step, by analyzing the contents of the Makefile.

- Each time the environment is run, a random seed is generated, and an output folder associated with this seed is created. This is controlled by the `SEED` parameter.
- Next, the RISCV-DV Instruction Generator is called with the given arguments. RISCV-DV is utilized via its `run.py` python script located in the directory `ibex/vendor/google_riscv-dv/`.
  - Most importantly, one of the given arguments is

    ```
    --custom_target=riscv_dv_extension
    ```

    , which allows the generator to utilize the files within that directory, which are the `testlist.yaml` and the `riscv_core_setting.sv` previously mentioned.
  - Along with this, the `ISA` and `ABI` parameters are also given.
  - Then, all tests are read from the `testlist.yaml` file. If `TEST` parameter is given, it's searched for within that file.
  - After finding the test(s) to be generated, the aforementioned `simulator.yaml` file is processed, and the parameters to be used when calling the simulator are obtained. The simulator of choice is also defined with the `SIMULATOR` parameter, which is Questa in this case.

  - The instruction generator code is written with SystemVerilog, with the source files located in the directories `ibex/vendor/google_riscv-dv/src` and `ibex/vendor/google_riscv-dv/test`. These files are imported into Questa and run to generate the instructions. The source files also import the `riscv_core_setting.sv` and from there, determine the instructions to be utilized.
  - One of these source files, `src/riscv_instr_pkg.sv` includes the file `user_extension.svh` from the `riscv_dv_extension` directory, in which the previously mentioned assembly header generator file, `ibex_asm_program_gen.sv` is contained.
  - The generated assembly code is then saved to the directory `out/seed-<SEED>/instr_gen/asm_tests/`.

- In the next step, the assembly code is compiled, again by calling RISCV-DV.
  - In this case, the RISCV-DV script is merely used as a wrapper to call gcc with the given compiler parameters. It doesn't have any special purpose in compiling the code.

- After the code is compiled, the Spike Instruction Set Simulator is called, again using RISCV-DV's `run.py` as a wrapper. The same parameters given to RISCV-DV are also passed along to Spike. Spike's log output is stored inside the directory `out/seed-<SEED>/instr_gen/spike_sim/`.

- The second part of the test flow involves the RTL simulation. The steps involving this part are managed via the `sim.py` python script located in the directory `ibex/dv/uvm/core_ibex/`.

- First, the RTL testbench is compiled. This is accomplished by running the correct command according to the simulator used. For Questa, this is defined in the `rtl_simulation.yaml` file as follows:

```
- tool: questa
  compile:
    cmd:
      - "vmap mtiUvm $QUESTA_HOME/questasim/uvm-1.2"
```

```
    - "vlog -64
            -access=rwc
            -f ibex_dv.f
            -sv
            -mfcu -cuname design_cuname
            +define+UVM_REGEX_NO_DPI
            +define+UVM
            -writetoplevels <out>/top.list
            -l <out>/compile.log <cmp_opts>"
```

- It can be seen that, using the aforementioned `ibex_dv.f` file, the list of all the required RTL components is obtained, and compiled. The compilation log generated by Questa is located at `out/seed-<SEED>/rtl_sim/compile.log`.
- In the next step, the RTL simulation is run, again reading the Questa parameters from `rtl_simulation.yaml`. It is helpful to understand the structure of the testbench and the roles of the various files referenced in the `ibex_dv.f` compilation list. These files collectively define the Ibex verification environment and dictate how the DUT and the interfaces are connected and configured.
- Here is an overview of these components and their responsibilities within the testbench hierarchy:
  - `tb/core_ibex_tb_top.sv`:
    - Instantiates the DUT, `ibex_core_tracer.sv`, which is the tracer included version of the core.
    - Declares and connects the UVM interfaces for various components like instruction memory, data memory, core interrupts, CSR unit interface and RVFI (RISC-V Formal Verification Interface) [12].
  - `tests/core_ibex_test_pkg.sv`: The SystemVerilog package file for the test, that imports other various interface components and the UVM library.
  - `tests/core_ibex_base_test.sv`: The "control unit" of the test environment, which is responsible from various tasks like loading the test binaries to memory, checking for test timeouts and stopping the test on errors.
  - `tests/core_ibex_test_lib.sv`: Defines various test types, as utilized in `testlist.yaml`, like CSR tests, interrupt tests etc.

- Items under `common/ibex_mem_intf_agent/` are responsible for the definition of the various UVM components for the memory interface agent. This agent is used for both the data and instruction memory agents. In fact, there are two agents within this directory, for both request and response, but only the response agent seems to be used. The important files within this directory are:
  - `ibex_mem_intf.sv`: The SV interface itself.
  - `ibex_mem_intf_monitor.sv`: The monitor for the interface, responsible from reading data written to memory.
  - `ibex_mem_intf_response_driver.sv`: Responsible from driving the data from the sequencer into the core."
  - `ibex_mem_intf_response_sequencer.sv`: The sequencer for the memory interface, almost unchanged from a default uvm class.
  - `ibex_mem_intf_response_seq_lib.sv`: The sequence used in the interface, responsible from handling various byte size read/writes.
  - `ibex_mem_intf_response_agent.sv`: The combination of the above components
  - Request equivalents for these also exist, but are unused as stated above.
- Items under `common/irq_agent/` are similar, but for the interrupt interface.
- The UVM environment, which contains all the various agents, is defined in the files under `env/`. The notable files here are:
  - `core_ibex_csr_if.sv`: The CSR interface. Is only utilized within the CSR test found in `core_ibex_test_lib.sv`
  - `core_ibex_dut_probe_if.sv`: Interface for the internal signals of the DUT, like `illegal_instruction`, `ecall` or `ebreak`.
  - `core_instr_monitor_if.sv`: The interface to monitor the instructions decoded by the DUT. Used in some specific tests.
  - `core_ibex_rvfi_if.sv`: The interface utilized to probe the RVFI interface between the core and the tracer (will be discussed in more detail ahead). Doesn't seem to have a purpose apart from debugging.

    * `core_ibex_env.sv`: The top environment module. Data and instruction memory agents, and the interrupt agent are called and configured here.

- One very important component, that's stored apart from the UVM environment files, is the tracer module, found in `ibex/rtl/ibex_tracer.sv`. This module is responsible from getting the RVFI signals generated within the core, and converting them into a logfile.
  - RVFI, short for RISC-V Formal Interface [12], is a standard set of RISC-V signals that describes the instructions executed on the core, and the effects of the instruction on the state of the core, like the operands and the result values. It's created to create a processor-agnostic definition of the ISA, to aid in verification.
  - The part of the core responsible for generating the RVFI signals is only used for verification, and is defined within the core itself, `ibex/rtl/ibex_core.sv`. It must be modified according to the core's internal signals, and its pipeline size, when porting over to other cores like Hornet.

4.3 Connecting Hornet to the Environment

As preparation for adapting the Hornet core to the environment, everything relating to RVFI inside `ibex_core.sv` was copied inside Hornet's `core.v`, and the file was renamed accordingly to `core.sv`. To remove the syntax errors, the internal signals on the Hornet that seemed equivalent to the Ibex signals were written in place within the RVFI logic, very roughly, as a preliminary fix. Any errors due to the connections made here will be fixed in the future.

Next, the Hornet source files were stored inside `ibex/rtl_hornet/`, and the files were added into `ibex_dv.f`. After this, Ibex's tracer module was added to `core_wb.v`, which was also changed to `core_wb.sv`, and the necessary connections to the RVFI signals generated within the core were made. Then, the `ibex_core_tracing` module on the testbench was replaced with `core_wb`. Tentative connections between the core and the various interfaces were made.

When running the environment with these changes, the test runs until the timeout period is reached, and returns an automatic FAILED result. When looking at the logs, it's seen that there's no trace log produced. At this point, it's necessary to debug the connections by looking at the waveforms.

To be able to see the simulator window, the launch parameters in `rtl_simulation.yaml` must be changed. In particular, the `-c` parameter given to vsim runs the simulator without a GUI. When this is removed, the simulation window opens, but since the test is automated, it very rapidly shuts down. To prevent the test from starting automatically, one more modification is needed. The `-do "run -a; quit-f"` command given to vsim is removed, which caused the run and quit commands to be automatically inputted. In addition, `-voptargs=+acc` was aded to the `SIM_OPTS` field in the Makefile, to make all the internal signals visible in the waveform. With this, the simulator window is ready.

Additionally, the following commands can be used to save the waveforms and inspect later:

```
vcd file my_vcdfile.vcd
vcd add -r <top-module>/*
```

Using these commands, both the working Ibex testbench, and the Hornet connected testbench are simulated, and their waveforms dumped. There are several discrepancies that stand out:

- First of all, neither the data nor the instruction memory sends any data, and thus the core is unable to continue the test. This is likely caused by the core not sending the correct control signals to the memory interfaces.
- When the various control signals are investigated, It's seen that the `request` signal sent to the instruction memory interface isn't set to 1. In fact, Hornet completely lacks such a signal. Temporarily, this signal is set to be always 1. With this change, Hornet starts to receive instructions.
- Hornet's write enable signal is always 1, since it's active low, but in Ibex it's active high, so only 1 when needed. The write enable signal coming from Hornet should

thus be negated when passed along to the interface. Upon further inspection, this negation is already done when connecting the core to the `core_wb` module, so no additional change is necessary.

- When inspecting the waveforms, it's seen that the instruction memory address value inputted into Hornet is `8000_000X`. Looking at the waveforms, the signals `mux2_o_IF`, `mux3_o_IF` and `mux4_o_IF` also share this value. Their definition in the source code is:

```
//this mux is responsible for stalling the IF stage.
assign mux2_o_IF = mux2_ctrl_IF ? pc_o : pc_o + 32'd4;
//branch mux
assign mux3_o_IF = mux3_ctrl_IF ? branch_target_addr : mux2_o_IF;
assign mux4_o_IF = mux4_ctrl_IF ? mux3_o_IF : mux1_o_IF;
```

When the corresponding control signals are also checked in the waveform, the problem can be pinned down to the signal `mux2_ctrl_IF`, which is also X. This creates a mux choosing between `pc_o` and `pc_o + 4`. Since the `pc_o signal` only has an X in the bit corresponding to the value 4, this theory is confirmed.

This control signal is assigned to `stall_IF`, which is defined as:

```
assign stall_IF = hazard_stall | muldiv_stall_EX | fpu_stall_EX |
                  misaligned_access | data_stall_i | csr_stall;
```

Among these signals, the only problematic one is `data_stall_i`, which has a constant Z value, indicating that it's not connected. This signal is utilized to stall the pipeline when the data memory access request is answered with a stall. This is one of the core's inputs, so ideally it should have some sort of control signal from the data memory interface.

Among the data memory interface's signals, there's not an exact equivalent to the data stall signal, so one has to be created using the available signals. The temporary signal generated for this purpose is:

```
assign data_stall = data_mem_vif.request & ~data_mem_vif.rvalid;
```

When the data from the memory is valid, the `data_mem_vif.rvalid` signal is pulled high. The idea is, when the data is not valid, the pipeline should stall, so the signal is inverted. To avoid stalls when there's no memory accesses, this signal is also AND'ed with `data_mem_vif.request`.

- Even after this fix, the simulation still has issues and can't create a trace log. Inspecting the waveform, it's seen that the instruction address is filled with X again, and the program is halted. The problem is caused by the signal `mux4_ctrl_IF`, which is connected to the CSR unit's `mux2_ctrl_o` output.

  CSR unit's source code shows that this value can be traced to the `mret_id_i` signal, which is defined in the control unit source code as:

```
assign mret    = instr_i == 32'h3020_0073;
```

  So the issue is due to the `instr_i` value entering the core being X. The instruction memory interface only sends valid data when it sends the `rvalid signal`. Due to the way Hornet is designed, it always assumes the instruction memory will send valid data, with no gaps.

  To fix this, a new `instr_stall_i` input is created, and added everywhere that also has a `data_stall_i` signal, mainly the pipeline stall signals. To this input, the inverse of the `instr_mem_vif.rvalid` is connected.

- Ibex seems to start executing the code from address 0x80000080, whereas Hornet starts from 0x80000000. The address 0x80000000 is defined as `` `BOOT_ADDR `` and is given to both cores, but Ibex seems to have some internal logic to skip to 0x80000080 at boot, skipping over the initial debug jumps in the header. This can be seen in the `rtl/ibex_if_stage.sv` file:

```
// fetch address selection mux
  always_comb begin : fetch_addr_mux
    unique case (pc_mux_internal)
      PC_BOOT: fetch_addr_n = { boot_addr_i[31:8], 8'h80 };
      PC_JUMP: fetch_addr_n = branch_target_ex_i;
       // set PC to exception handler
      PC_EXC:  fetch_addr_n = exc_pc;
       // restore PC when returning from EXC
```

```
      PC_ERET: fetch_addr_n = csr_mepc_i;
      PC_DRET: fetch_addr_n = csr_depc_i;
      /* Without branch predictor will never get
      pc_mux_internal == PC_BP. We still handle no branch
       predictor case here to ensure redundant mux logic
       isn't synthesised.*/
      PC_BP:   fetch_addr_n = BranchPredictor ? predict_branch_pc
      : { boot_addr_i[31:8], 8'h80 };
      default: fetch_addr_n = { boot_addr_i[31:8], 8'h80 };
    endcase
  end
```

The code that is skipped over is as follows:

```
80000000 <_start-0x80>:
80000000:    2580c06f    j    8000c258 <debug_rom>
80000004:    2580c06f    j    8000c25c <debug_exception>
80000008:    00000013    nop
8000000c:    00000013    nop
80000010:    00000013    nop
...

80000080 <_start>:
80000080:    40001fb7    lui        t6,0x40001
```

So the two cores aren't executing the same code. To alleviate this, 0x80 is added to the `BOOT_ADDR passed to Hornet.

With all the fixes in place, when the environment is run again, it's seen that the test fails early with the following error:

```
# UVM_ERROR /home/intern_mozden/ibex/vendor/lowrisc_ip/ \
dv/sv/mem_model/mem_model.sv(30) @ 53942966: reporter   \
[mem] read to uninitialzed addr 0x800d7763
```

The core tries to access the instruction memory address 0x800d7762, in which the second byte is uninitialized as per the error message. The root cause of this will be analyzed.

4.4 Technical Difficulties

Before the work on integrating Hornet into the Ibex Verification Environment could be completed, we ran into some technical issues that removed our access to QuestaSim. With no UVM supporting simulator available to use for the project, creating a brand new verification environment without any UVM components became necessary.

5.1 Manual Fixes on Hornet

Before a verification environment could be used, there are some fixes that had to be done to prepare Hornet for it. In addition, since there were technical issues that prevented us from getting a working verification environment for some time, some manual tests and fixes for the FPU were also performed.

5.1.1 Increasing the memory

Initially, the memory map of Hornet was defined as follows
in the `barebones_wb_top.v` file:

```
assign slave_adr_begin[0] = 32'h0000_0000;
assign slave_adr_end[0] = 32'h0000_1DFF;
assign slave_adr_begin[1] = 32'h0000_0000;
assign slave_adr_end[1] = 32'h0000_1FFF;
assign slave_adr_begin[2] = 32'h0000_2000;
assign slave_adr_end[2] = 32'h0000_200F;
assign slave_adr_begin[3] = 32'h0000_2010;
assign slave_adr_end[3] = 32'h0000_2010;
```

The four sections of the memory space are for instruction memory, data memory, privileged "mtime" registers and the debug address, respectively. From here, we see that Hornet has 8KiB of memory, with 7.5KiB reserved for instructions.

Most testing and benchmarking code that can be found on the internet, doesn't fit in this 7.5KiB of memory. As such, the address space of Hornet was modified as such:

```
assign slave_adr_begin[0] = 32'h0000_0000;
assign slave_adr_end[0] = 32'h0000_6FFF;
assign slave_adr_begin[1] = 32'h0000_0000;
assign slave_adr_end[1] = 32'h0000_7FFF;
assign slave_adr_begin[2] = 32'h0000_8000;
assign slave_adr_end[2] = 32'h0000_800F;
```

```
assign slave_adr_begin[3] = 32'h0000_8010;
assign slave_adr_end[3] = 32'h0000_8010;
```

With this change, Hornet now has 32KiB of memory, of which 28KiB is reserved for instructions, and 4KiB for data. This has proven to be enough for our uses.

Another fix that has been done is to initialize the memory to 0 before loading the code. Most compilers assume that, on run-time, the unwritten data addresses will have all 0 written in them. As it stands, Hornet simply leaves them as X (undefined). On physical hardware, this would mean that the values in those addresses are completely random. This could cause issues in the code.

```
reset_i = 1'b0; fast_irq_i = 16'b0; meip_i = 1'b0;
for (i = 0; i < uut.memory.RAM_DEPTH; i = i + 1) begin
    uut.memory.mem[i] = {uut.memory.DATA_WIDTH{1'b0}};
end
#200;
reset_i = 1'b1;
$readmemh("fpu_test.data",uut.memory.mem);
```

To fix this problem in simulation, the testbench was modified to write all 0s to the memory, then load the code. In this way, the unwritten values simply remain as 0, instead of X.

5.1.2 Fixes relating to the verification environment

In Spike ISA Simulator [7], roughly a few thousand bytes of memory address space, starting from 0, is dedicated to the simulator's internal functions (like debugging). As such, the code compiled for Hornet, which start from address 0, don't directly work on Spike.

A different address range can be defined via a linker script. In that case though, the different instruction addresses and the contents of instructions that directly reference these addresses can be different between Spike and Hornet. These differences make both the automated and manual comparisons between the two difficult. Since the verification environment also utilizes Spike, it's important that the tested cores can execute code starting from an arbitrary address.

In Hornet, a parameter called `reset_vector` was defined, to allow it to start from any address. But this parameter doesn't work as expected, as it seems like it's not used where it's needed. For example, when the reset signal is sent, which is always done at the start, the value in `reset_vector` is correctly sent to the program counter, but not the CSR unit.

As a result of this, the "mtvec" register in the CSR unit, which holds the reset vector value according to the Privileged ISA documentation [9], gets a 0 input value. At simulation start, since the input signals are reset to 0, the first instruction read by the core is 0x00000000, which is invalid. Due to this, the CSR unit raises illegal instruction trap error, which resets the core to the address stored in "mtvec". Since "mtvec" didn't receive the correct reset vector, the core instead returns to the address 0. We see that, in the `csr_unit.v` file, "mtvec" is assigned a 0 value, instead of `reset_vector`. After this is fixed, the core works as expected, as can be seen in Figure 5.1 and Figure 5.2.

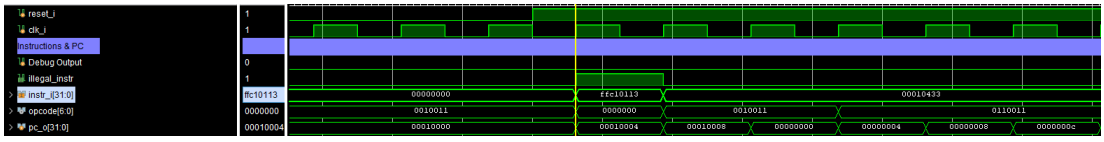


Figure 5.1 : The operation of the core before the fix.

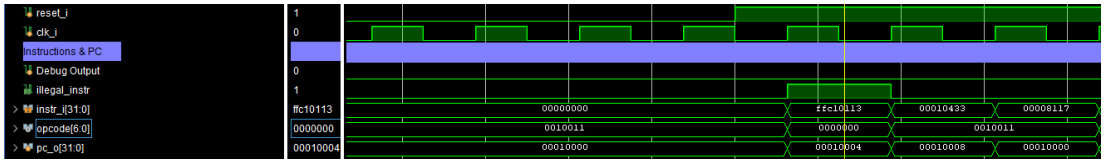


Figure 5.2 : The operation of the core after the fix.

5.1.3 Preliminary FPU fixes

5.1.3.1 Some basic tests for edge cases

The issues found in the FPU were mostly edge cases, especially concerning special values like 0, infinity and NaN. But there were also some that were more significant. Perhaps one of the most important issues was the complete lack of fused multiply-add instructions. Not only is there no specific hardware path to execute these instructions, there's no control logic to handle it either. As such, whenever these instructions are decoded, the core raises an illegal instruction trap, and returns to the beginning. These instructions are part of the RISC-V ISA F extension [1], so it can be said

that the FPU implementation in Hornet is an incomplete one. The parameter `-ffp-contract=off` is used to force the compiler to not use these instructions.

The edge case issues, and their fixes, are as follows:

- The fcvt.w.s instruction, which converts a floating point value into an integer one, returns 0x7fffffff when the input is 0. The expected output is 0x00000000. In the process of fixing this issue, three distinct errors in Hornet were discovered. These are:
  - The FPU decoder unit, which converts the floating point values into sign, exponent and mantissa components for further processing, returns an isZero flag when the input of the decoder is 0. However, it also decodes the mantissa as 0x800000. For normalized numbers (not subnormal), the top bit of the mantissa, known as the hidden bit, is 1. However, in Hornet's implementation, this bit is 1 when the input is 0, which doesn't follow the IEEE754 standard. When the input is 0, all components of the floating point number must be 0. The code that controls the assignment of this bit can be found in line 37 of `fpu_decoder.v`, and if it's changed from

    ```
    assign sig_o = {!isSubnormal, fract};
    ```

    to

    ```
    assign sig_o = {!isSubnormal && !isZero, fract};
    ```

    this issue is fixed.
  - This alone isn't enough to solve the problem. We also note that, the overflow flag set in the module fpu_cvt_to_int is set to 1 even if the number is 0. The root cause is that, the overflow flag is set according to the decoded exponent of the number. From that decoded exponent, 127 is subtracted to obtain the actual exponent of the number. If that value is greater than 31, it won't fit in an integer, so overflow flag is raised. The problem with this, is that the subtracted value is held in an unsigned wire, which underflows for negative values. As such, when the actual exponent value is negative, the comparison assumes it's

greater than 31, and mistakenly raises the overflow flag. When this wire is set to signed, this issue is fixed.

- The last problem is rather simple. In the fpu_arithmetic_top module, the value is_exp_neg, which denotes whether the exponent is negative, returns the exact opposite value it's supposed to, due to a simple typo. This value is used as an input in the integer conversion unit, and if it's 1, the integer conversion automatically outputs 0. Changing the relevant line (line 117 in `fpu_arithmetic_top.v`) from

```
    assign is_exp_neg = exp_A[7] ? 1'b0 :
                        (&exp_A[6:0] ? 1'b1 : 1'b0);
```

to

```
    assign is_exp_neg = exp_A[7] ? 1'b0 :
                        (&exp_A[6:0] ? 1'b0 : 1'b1);
```

fixes this issue.

- A similar issue can be found in the opposite fcvt.s.w instruction, which converts an integer to a floating point. When it's input is 0, it returns 1.0, instead of 0. This is fixed by simply adding an exception for 0 input, in the `fpu_cvt_to_float.v` file, as the operation for all other values seems to be correct. The applied fix, on the 41st line, is as follows:

```
assign cvrt_to_float_out = |A[31:0] ?
       {final_sign, final_exp, final_sig} : 32'b0;
```

Next, the operations were checked using special input values, like 0, infinity and NaN. A test code was written, in which the special values were used with different operations. The results for most of these operations are as one would expect, but there are a few that may have been missed during implementation. Some examples of these are as follows:

- 1.0/0.0 = +inf

- -1.0/0.0 = -inf
- sqrt(-0) = -0
- -0.0 = +0.0
- -0.0 - (-0.0) = +0.0

All of the test cases were confirmed to be correct in Spike. But on Hornet, some of these cases weren't considered. The two that we found, and their fixes are as follows:

- The division 1.0/0.0 = +inf is correct, but -1.0/0.0 also returns +inf, instead of -inf. For these edge cases, Hornet returns an output without performing an actual division operation (which would fail). The outputs for such cases are defined in the `fpu_mds_ctrl.v` file. The relevant code segment is as follows:

```
// A = (SUB)NORMAL & B = 0
if(isZeroB) begin
    fast_res = {1'b0, 8'd255, 23'd0};
    overflow_fast = 1;
    invalid_fast = 0;
    divByZero_fast = 1;
    mux_fastres_sel_temp = 1'b1;
end
```

  For edge cases, the output is written into the `fast_res` register, from which it can be read by the rest of the core. It can be seen that, when A is a normal or subnormal number and B is 0, the output is set to +inf. If the output sign is set to the input A's sign instead, the problem will be solved. The fixed line is:

```
    fast_res = {sign_A, 8'd255, 23'd0};
```

- The other mismatch with Spike was the comparison between -0.0 and +0.0. This special case wasn't considered in the implementation, so when these inputs are given, the FPU uses the regular input path, and gives results that indicate -0.0 < +0.0, due to the sign difference. To fix it, an additional special case was added to the relevant file, `fpu_compare.v`. IsZeroA and IsZeroB signals were given to this module, and the equal/less outputs were modified from

```
assign is_equal = is_sign_equal & is_exp_equal & is_sig_equal;
assign is_less  = !is_sign_equal ? sign_A & !sign_B :
                  !is_exp_equal  ? exp_A  < exp_B   :
                  !is_sig_equal  ? sig_A  < sig_B   :
                  1'b0;
```

to

```
assign is_equal = (isZeroA & isZeroB) | \
(is_sign_equal & is_exp_equal & is_sig_equal);
assign is_less  = (isZeroA & isZeroB) ? 1'b0        :
                  !is_sign_equal ? sign_A & !sign_B :
                  !is_exp_equal  ? exp_A  < exp_B   :
                  !is_sig_equal  ? sig_A  < sig_B   :
                  1'b0;
```

5.1.3.2 Test on digital filter

A code of a digital biquad filter 5.1 is written to check for errors and validate how the FPU works.

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n] - a_2y[n-1] \tag{5.1}$$

The test has helped us to find two errors regarding the multi-cycle instructions:

- When multi-cycle instructions, such as division and multiplication are to be done, the FPU stores the input values in the registers, and uses those for the operation. This is done to ensure that the inputs do not change during the operation. The utilization of these registers are controlled by a state machine, found in the file `fpu_top_ctrl.v`:

```
always @* begin
    case(current_state)
        FIRST: begin
            in_sel = 1'b1;
            reg_AB_en = 1'b1;
            casez(op)
                5'b0001?, 5'b01011 : next_state = SECOND;
                default : next_state = FIRST;
            endcase
```

```
        end
        SECOND: begin
            in_sel = 1'b0;
            reg_AB_en = 1'b0;
            if(start)
                next_state = SECOND;
            else
                next_state = FIRST;
        end

    endcase
end
```

But, when two multi-cycle instructions are sequentially fed, the `start` signal stays at a constant 1, therefore the control unit doesn't realize the first instruction has ended. Therefore it feeds the second instruction with the inputs of the first one. To solve it, the `done` signal coming from the `fpu_arithmetic_top.v` module can be utilized as such:

```
if(start)
    if(done)
        next_state = FIRST;
    else
        next_state = SECOND;
else
    next_state = FIRST;
```

- A similar issue happens when when a multi-cycle operation is done shortly (for example, when one of the inputs is 0.0f). The problem is solved by checking if start and done are high at the same time which means `fpu_ctrl_top.v` checks that if a multi-cycle operation is fed with a shortcut resulting in a single-cycle operation, in which case it makes the `next_state` variable `FIRST` so that the input will be correct. For this purpose the code is altered as such:

```
if(start && done)
    next_state = FIRST;
else
    casez(op)
        5'b0001?, 5'b01011 : next_state = SECOND;
```

```
        default : next_state = FIRST;
    endcase
```

The digital filter test passed all the cases after the mentioned changes. However, some edge cases might still be not solved. For this purpose more tests are required to be more precise about the sufficiency of the processor.

5.1.3.3 Test on "Paranoia"

Not content with the extent of our test code, we have also looked for more comprehensive tests. One of particular note was Paranoia [13], written by Professor William Kahan, who's known as the main creator of the IEEE-754 floating point standard.

When we naively try to compile this code for Hornet, we are faced with an error, that states the provided memory is roughly 80 KB too small. The reason for this is the use of several C libraries like `stdio` and `stdlib` within the code. To be able to run this on Hornet, we need to make some adjustments. Some of these are:

- Since the code is running on bare metal, without an operating system, `printf` cannot be properly utilized. Instead of it, a simple string printing function is written, which outputs one character at a time to Hornet's debug interface, which can print the characters to the simulator terminal. This function doesn't support the printing of variables, so it's not an exact replacement of `printf`. The code is:

```
void write_char(char c) {
    volatile char *address = (volatile char *)0x00008010;
    *address = c;
}

void write_string(const char *str) {
    while (*str) {
        write_char(*str++);
    }
}
```

- The single precision floating point power and logarithm operations used in the code rely on fused multiply-add instructions, and as such aren't currently compatible

with Hornet. It's possible to remove fused instructions with the compiler parameter `-ffp-contract=off`, but this doesn't apply to the pre-built functions that are called when power and logarithm functions are compiled. This is due to the fact that these functions aren't implemented in hardware, as they aren't part of the RISC-V ISA Specification [1]. The tests that utilized these functions are thus commented off.

After the changes that are done, Paranoia is used to see if the errors still exist in the processor. The tests are done on the Spike and the Hornet processor. Then they are compered and checked if any errors exist. Errors found are in the following:

- If both inputs are `1.0f` or `0x3f800000` in hexadecimal then the output comes as `0xb2000000`. Then it is checked and it is realized that whenever same numbers are subtracted Hornet gives a wrong output. Mantissa's leading zeros are subtracted from exponent during the normalization process. However, if mantissas are equal then result's mantissa will be 0, which makes FPU subtract 27 from the exponent. To fix this, an edge case is added, where if the mantissa is 0, the output exponent will be 0. This should only occur in this specific instance, though it's not 100% certain, so this fix should be considered temporary. The fix applied to the file `fpu_add_sub.v` can be seen in the following:

```
else if (zeroCount == 27)
begin
    outExp = 8'b0;
    uf = 1'b0;
    temp = inSig[25:1] << zeroCount - 2;
end
```

- An error occurs when `fsub.s` instruction comes right after `fmul.s`, if one of `fsub.s`'s inputs depend on `fmul.s`'s output. The problem probably will also occur if `fadd.s` is used instead of `fsub.s`. The problem occurs because of the `forwarding_unit.v` module. 0th register in integer registers is out of the forwarding process because it is constant zero register. However, the 0th float register doesn't have such a purpose, so it shouldn't be exempt from forwarding. But as it stands, it is out of forwarding process too. The error occurs because the

0th float register was used. The forwarding unit's check is normally done as the following:

```
rs1 != 00000
```

However, this causes the problem and it is changed as the following:

```
(rs1 != 00000) OR (rs1_bank_sel == 1)
```

As a result, the integer 0th register still doesn't get included in forwarding, but the 0th float register does.

- The multiplication $3.0 * 9.0 = 27.0$ fails. Such a simple multiplication failing would indicate a significant oversight, so to be certain, the operation was re-tested using a separate test code. The operation was successful in that case, which points out that the issue is again somewhere in the control units.

  Looking at the waveforms in Figure 5.3, it can be seen that the `done` signal is sent one cycle early. Where a multiply operation normally takes 3 cycles, it has now taken only 2. For this reason, the wrong output is sent along the pipeline.

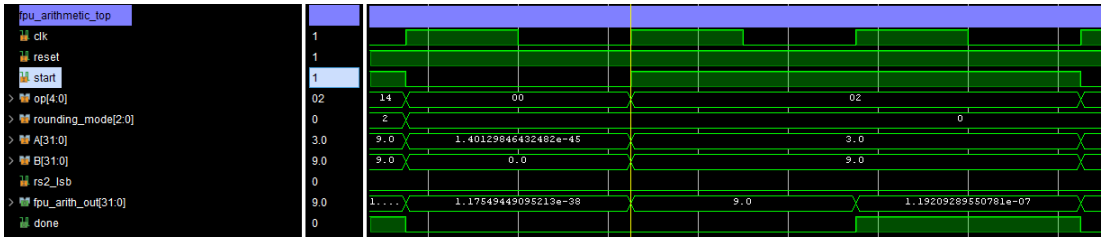


Figure 5.3 : The input and outputs of the FPU.

It's found out that this problem arises from the state machine that is found in `fpu_mds_ctrl.v`. In normal operation, when a multiplication instruction is received, the state machine follows the states in the order IDLE -> MUL1 -> MUL2 -> MUL_OUT. As observed, a multiplication operation takes 3 cycles. But, when the state machine is inspected during the faulty operation mentioned above, it's observed that the state machine is already in the MUL2 state when the start signal is sent. This can be seen in Figure 5.4, where the yellow cursor denotes the point where multiplication starts.

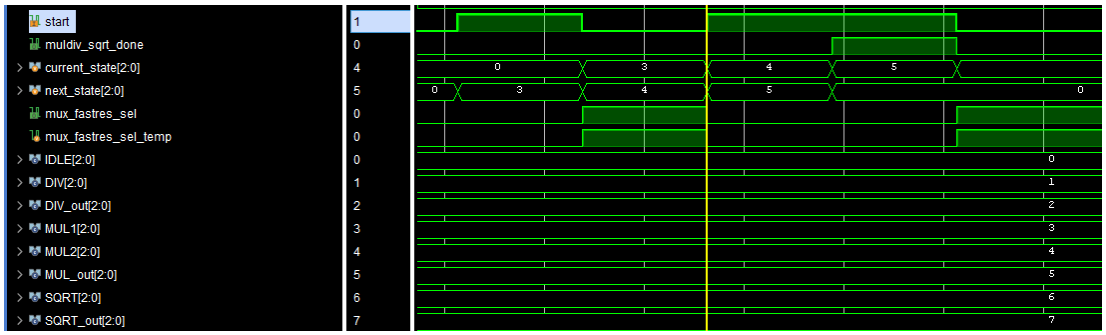


Figure 5.4 : The states that govern the Multiply-Divide-SQRT unit operation.

From Figure 5.4, it can clearly be observed that an unrelated `start` signal, sent due to a previous instruction, affects the state machine in the Multiply-Divide-SQRT (MDS) unit. It's found out that no matter the instruction, the state machine in this unit always runs, but simply discards the output if the instructions aren't targeting it. In some cases, when an unrelated instruction is quickly followed by a multiplication or division operation, this can cause malfunctions in the operation, as was observed.

To fix this issue, a separate start signal, that is only activated when the opcodes corresponding to the MDS unit are detected. This new start signal is fed into the MDS unit. Now, the other instructions' start signals do not pollute the MDS unit, and proper operation is restored.

```
assign mds_start = start & (op == 5'b00010 | op == 5'b00011 \\
                            | op == 5'b01011);
fpu_mds_top fpu_mds_top(clk, mds_start, ...);
```

5.2 Creating a New Verification Environment

Even before losing access to QuestaSim, there were significant hurdles in adapting Hornet into the Ibex Verification Environment, particularly concerning the memory subsystem used. Ibex's verification environment utilized UVM drivers for simulating the instruction and data memory. These had control signals that were tailored for the Ibex core, which made adapting it to other cores like Hornet difficult.

By creating a new environment from scratch, Hornet's existing memory could be easily utilized. Within this environment, logs obtained from a tracer attached to Hornet will be compared with the logs outputted from Spike ISS. This will be accomplished by converting the logs to structured .csv files and comparing them instruction by

instruction, with the aid of Python scripts. The results will be logged to both a final comparison .csv and also printed in the terminal for quickly viewing the results. The test code will be generated by RISCV-DV, with the option of using custom hand-written tests for inspecting specific edge cases.

5.2.1 Hornet tracer

The created trace log will be compared against Spike's output log, so the tracer should also be able to obtain the values found in that log from the core. These values are:

- Program Counter (PC) value of the instruction
- The instruction itself
- Register written to (also considering writing to x0 results in no actual write)
- The value written to the register
- Memory access address and written/read data, taking word size into consideration

In addition, the distinction between integer and floating point registers should also be made.

The tracer implementation is made up of two components:

- The tracer module itself, which takes the relevant data input and prints it out to the log in a formatted manner
- Trace outputs on the top core module, which collect the required values, forward them to the final WB (Writeback) pipeline stage if required, and sending the data to the tracer module. This part should also filter out the NOP instructions injected into the pipeline for hazard handling purposes.

The following are the internal signals generated to pass along to the tracer module, and how they are derived:

- **tr_pc**: The program counter (PC). Obtained by pipelining `pc_MEM` from the MEM (Memory) stage to the WB stage.

- **tr_instr**: The executed instruction. Normally, the undecoded instruction isn't needed after the ID (Instruction Decode) stage, so the signal was pipelined from the ID stage to the WB stage, obtaining `instr_WB`.
- **tr_is_float**: The signal that tells whether the instruction writes to the float register bank or not. The existing `mux_ctrl_rb_WB` signal was used.
- **tr_reg_addr**: The index of the target register. The signal `rd_WB` is used. This signal is then ANDed with the inverse (since it's active low) of the register file write enable signal `rf_wen_WB`. Whether the register is an integer or a float register is determined by `tr_is_float`.
- **tr_reg_data**: The data written to the target register. The signal `mux_o_WB` is used, which is the output of the mux that selects whether the data written to the register comes from the memory, ALU or the FPU. This signal is also ANDed with the inverse of `rf_wen_RB`.
- **tr_load** and **tr_store**: Generated by the last 7 bits (opcode) of the `instr_WB` signal. Depending on whether the operation is a load or store (also taking float load and stores into account), these signals are raised high or low.
- **tr_mem_addr**: Data Memory address. Since all address calculations are done in the ALU, the `aluout_WB` signal was used. If neither `tr_load` or `tr_store` are high, this signal is 0 instead.
- **tr_mem_data**: Data written to, or read from the Data Memory. For the generation of this signal, two separate inputs are used:
  - The `data_o` signal, which is the data outputted to the memory, pipelined from the MEM stage to the WB stage
  - The `memout_WB` signal, which is the data coming from the memory
  - Depending on the values of `tr_load` and `tr_store`, the output is either one of these two signals, or 0.
- The aforementioned assignment based on the `tr_load` and `tr_store` signals is done as follows:

```
always @(*) begin
    if (instr_WB[6:0] == 7'b0000011) begin //Load
        tr_mem_data = memout_WB; //Data to be read from memory
        tr_mem_addr = aluout_WB; //Address to be read from ALU
        is_load = 1'b1;
        is_store = 1'b0;
    end
    else if (instr_WB[6:0] == 7'b0100011) begin //Store
        tr_mem_data = memin_WB; //Data to be written to memory
        tr_mem_addr = aluout_WB; //Address to be written to ALU
        is_load = 1'b0;
        is_store = 1'b1;
    end
    else if (instr_WB[6:0] == 7'b0000111) begin //Float Load
        tr_mem_data = memout_WB; //Data to be read from memory
        tr_mem_addr = aluout_WB; //Address to be read from ALU
        is_load = 1'b1;
        is_store = 1'b0;
    end
    else if (instr_WB[6:0] == 7'b0100111) begin //Float Store
        tr_mem_data = memin_WB; //Data to be written to memory
        tr_mem_addr = aluout_WB; //Address to be written to ALU
        is_load = 1'b0;
        is_store = 1'b1;
    end
    else begin //Branch
        tr_mem_data = 32'b0; //No data to be written to memory
        tr_mem_addr = 32'b0; //No address to be written to ALU
        is_load = 1'b0;
        is_store = 1'b0;
    end
end
```

- **tr_mem_len**: Signal which determines the size of the length of the data read from/written to the Data Memory. The `mem_length_WB` signal was used, which has the following mapping:
  - 00: Byte
  - 01: Halfword
  - 10: Word
- **tr_valid**: Signal that indicates whether the other outputs are valid or not. This is determined by the use of dummy signals, which are raised high in each pipeline

register when it's flushed, or a NOP is inserted. These signals also propagate along the pipeline, which ensures that these instructions are accurately marked. To generate the valid signal, the inverse of `MEMWB_preg_dummy` was used.

The tracer module is designed such that it writes the values to the log file on `tr\_valid`'s rising edge. As such, `MEMWB_preg_dummy` is also ANDed with the inverse of the clock, so that it has a rising edge at the middle of the clock cycle. With this any possible wrong readings at the clock edge is prevented.

- **fflags**: The FPU status flags. These are outputted from the FPU, and are sent to the CSR unit. As such, they were pipelined from the EX (Execute) stage to the WB stage, to also be sent to the tracer. While not necessarily critical to achieve correct operation, these values can be helpful in debugging. According to the RISC-V Unprivileged spec [1], these are:
  - **NV**: Invalid Operation
  - **DZ**: Divide by Zero
  - **OF**: Overflow
  - **UF**: Underflow
  - **NX**: Inexact

To convert these signals into a log file, the following tracer module is written:

```
module tracer(input clk_i,
                input valid,
                input [31:0] pc,
                input [31:0] instr,
                input [4:0] reg_addr,
                input [31:0] reg_data,
                input is_load, is_store, is_float,
                input [1:0] mem_size,
                input [31:0] mem_addr,
                input [31:0] mem_data,
                input [31:0] fpu_flags);

integer file_pointer;
initial begin
file_pointer = $fopen("../../../../../trace.log", "w");
//The file is normally located in \
//<vivado-dir>\HornetRISCV-vivado.sim\sim_1\behav\xsim, \
```

```
//so let's use relative path to move it to the main folder
  forever begin
    @(posedge valid);
    $fwrite(file_pointer, "0x%8h (0x%8h)", pc, instr);
    if (is_store) begin
      if(mem_size == 2'b00) begin
        $fwrite(file_pointer, " mem 0x%8h 0x%2h",mem_addr,mem_data);
      end
      else if(mem_size == 2'b01) begin
        $fwrite(file_pointer, " mem 0x%8h 0x%4h",mem_addr,mem_data);
      end
      else begin
        $fwrite(file_pointer, " mem 0x%8h 0x%8h",mem_addr,mem_data);
      end
    end
    else begin
      if (!is_float) begin
        if (reg_addr == 0) begin
          //Do nothing
        end else begin
          if (reg_addr > 9) begin
            $fwrite(file_pointer, " x%0d 0x%8h",reg_addr,reg_data);
            if (is_load) begin
              $fwrite(file_pointer, " mem 0x%8h", mem_addr);
            end
          end else begin
            $fwrite(file_pointer, " x%0d  0x%8h",reg_addr,reg_data);
            if (is_load) begin
              $fwrite(file_pointer, " mem 0x%8h", mem_addr);
            end
          end
        end
      end
      else begin
        if (fpu_flags != 0)
        $fwrite(file_pointer, " c1_fflags 0x%8h", fpu_flags);

        if (reg_addr > 9) begin
          $fwrite(file_pointer, " f%0d 0x%8h",reg_addr,reg_data);
          if (is_load) begin
            $fwrite(file_pointer, " mem 0x%8h", mem_addr);
          end
        end else begin
          $fwrite(file_pointer, " f%0d  0x%8h",reg_addr,reg_data);
          if (is_load) begin
            $fwrite(file_pointer, " mem 0x%8h", mem_addr);
          end
        end
      end
```

```
    end
    $fwrite(file_pointer, "\n");
  end
end
endmodule
```

The format of the log is largely identical to Spike's logs, mainly the part outputted when the `--log-commits` option is used. This is mainly to aid in testing the tracer by manually comparing the logs, before implementing the comparison. Since both logs will eventually be converted to csv files before comparing, the trace log doesn't necessarily have to be similar to Spike's log. In the end, the log takes the following form:

```
...
0x000100f8 (0xffff0f13) x30 0xff7fffff
0x000100fc (0xf00f0753) f14 0xff7fffff
0x00010100 (0x80000f37) x30 0x80000000
0x00010104 (0xf00f07d3) f15 0x80000000
0x00010108 (0x7f800f37) x30 0x7f800000
0x0001010c (0xf00f0853) f16 0x7f800000
0x00010110 (0xff800f37) x30 0xff800000
0x00010114 (0xffff0f13) x30 0xff7fffff
0x00010118 (0xf00f08d3) f17 0xff7fffff
0x0001011c (0xff800f37) x30 0xff800000
...
0x00013610 (0x340d9df3) x27 0x00000000
0x00013614 (0x000e8db3) x27 0x00017fe4
0x00013618 (0xf84d8d93) x27 0x00017f68
0x0001361c (0x001da223) mem 0x00017f6c 0x00000001
0x00013620 (0x002da423) mem 0x00017f70 0x00000000
0x00013624 (0x003da623) mem 0x00017f74 0x00000001
0x00013628 (0x004da823) mem 0x00017f78 0x00000000
0x0001362c (0x005daa23) mem 0x00017f7c 0xd635a000
...
```

5.2.2 Automating the environment

5.2.2.1 Automatically exiting the program

Both Spike and our RTL simulator (Which is Xilinx Vivado) need a signal to tell whether the executed program is over or not, and terminate the simulation. To accomplish this, a different solution for each needs to be implemented.

In case of Spike, the sections `tohost` and `fromhost` in the compiled .elf file are used to facilitate communication between the simulator and the simulated core. For this use case, only `tohost` is needed. To utilize this, the entry function used when compiling for Hornet, `crt0.s`, must be modified. The modifications are as follows:

- By using assembler directives, the `tohost` section is defined as a writable section stored in the executable file. Then, the `tohost` label is defined globally, so it can be directly accessed by other functions. Finally, 8 bytes of memory space is reserved for this section, and it's aligned to a 16 byte boundary. In code, these definitions look like:

```
.section .tohost, "aw", @progbits
.globl tohost
.align 4
tohost: .dword 0
```

- When 1 is written to the lowest significant bit (LSB) of the `tohost` section, Spike terminates the simulation. To accomplish this, an exit function is used, which the program jumps to at the end of the start function. Since the program only returns to the end of the start function after it completes the main function, this solution works quite well. The exit function is as follows:

```
    li a0, 0
    j tohost_exit # just terminate with exit code 0
    .cfi_endproc

tohost_exit:
        slli a0, a0, 1
        ori a0, a0, 1

        la t0, tohost
```

```
        sw a0, 0(t0)

        1: j 1b # wait for termination
.end
```

The bits other than the LSB define the exit code, so the above function preserves this code. For our purposes, this is not utilized. Also of note is the final loop at the end. Spike's termination after receiving the signal is not instant, so the code waits here until then. This behavior can also be observed in the RTL simulation.

To give the RTL simulation a way to detect program end as well, the debug interface of Hornet can be used. A single byte can be sent to the defined address of this interface. By sending a specific value, which would call the Verilog `$finish` function on arrival, the simulation can be terminated.

This interface was previously used to print the program output to the terminal, by sending an ASCII character to the debug interface address. Normally, ASCII characters include all uppercase and lowercase Latin characters, along with punctuation marks, numbers and several control codes. To preserve the printing function of the debug interface, none of these characters can be used.

Luckily, the ASCII characters only use the values between 0-127, out of the 256 available. The remaining section is referred to as "Extended ASCII", which is used for characters from other languages, like Turkish for example. Moreover, not all of these values are used. By utilizing this fact, the value 144 (0x90) is used for the exit function. With this change, the main logic of `debug_interface_wb.v` is as follows:

```
begin
        if(wb_rst_i) begin end

        else
        begin
                if(wb_cyc_i && wb_stb_i && wb_we_i)
                begin
                        if(wb_dat_i == 32'b1)
                        begin
                                $display("Success!");
                                //$finish;
                        end
```

```
                        else if(wb_dat_i == 32'b0)
                        begin
                                $display("Failure!");
                                //$finish;
                        end
                        else if(wb_dat_i == 32'h90)
                                $finish;
                        else
                                $write("%c",wb_dat_i[7:0]);
                end
        end
end
```

The `crt0.s` entry function is also modified to send this value to the debug interface:

```
        li t1, 0x8010 # Our debug address
        li a1, 0x90

        la t0, tohost
        sw a0, 0(t0) #Terminate spike
        sw a1, 0(t1) #Terminate RTL
        1: j 1b # wait for termination
```

5.2.2.2 RISCV-DV

For details on various components of RISCV-DV, refer to Chapter 4. This section will outline the various problems faced when trying to implement RISCV-DV in our verification environment, and their fixes. Since RISCV-DV's code is written using SystemVerilog, and utilizes UVM, a UVM supporting simulator is required. Luckily, RISCV-DV also has a Python port of its SystemVerilog components, which has most (but not all) of the original functionality. As such, it can still be integrated into our environment.

One of the most important things to make sure in a verification environment is that both the RTL simulator and ISS run the exact same code. In this instance, Hornet has a limited amount of memory with a specific memory map (refer to 5.1.1). As such, it's the limiting factor here and Spike should also work within this constraint.

To make sure both get the same code, it's beneficial to start by using the same linker script to compile for both simulators. A linker script is a set of directives that tell the

compiler (more accurately, the linker) where to put the instructions and the program data in the address space. To change the linker script RISCV-DV uses, the file at `scripts/link.ld` is replaced with Hornet's linker script. A small segment of this file (since it's too long) is shown below:

```
OUTPUT_FORMAT("elf32-littleriscv", "elf32-littleriscv",
              "elf32-littleriscv")
OUTPUT_ARCH(riscv)
MEMORY
{
    ROM(RX)   : ORIGIN = 0x00010000, LENGTH = 0x00007000
    RAM(WAIL) : ORIGIN = 0x00017000, LENGTH = 0x00000FFC
}

ENTRY(_start)
SEARCH_DIR("=/opt/riscv/riscv32-unknown-linux-gnu/lib32/ilp32d");
SEARCH_DIR("=/opt/riscv/riscv32-unknown-linux-gnu/lib32");
...

SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x10000));
  . = SEGMENT_START("text-segment", 0x10000) + SIZEOF_HEADERS;
  PROVIDE(__stack_top = ORIGIN(RAM) + LENGTH(RAM));

...
```

If RISCV-DV is now run as is, the linker gives out an error, stating that the RAM and ROM regions overflow by many bytes. This is expected, since the default values of RISCV-DV are determined with an unconstrained memory in mind. Both the RAM and ROM overflow is fixed with these following changes:

- In the file `riscv_instr_gen_config.py` located at `pygen/pygen_src/`, at line 144, the variable `stack_len` is set to 5000. This means that the stack size in the generated assembly file is determined to be 5000 words, which is around 20 KBytes. Since Hornet only has 4 KBytes of data memory, this value is reduced to 500.
- In the file `testlist.yaml`, in which the floating point arithmetic test was added the same way as in 4.2.1, it's noted that the `instr_cnt` parameter is set to 10000.

This tells RISCV-DV that it's allowed to generate up to 10000 instructions per test. Since each instruction is 32-bits long, this puts the instruction size at a maximum of 40 KBytes. Since Hornet only has 28 KBytes of instruction memory, this value should be at most 7000. To have some margin of error, this value is set to 6000.

After these fixes, RISCV-DV successfully generates and compiles the test code, and calls Spike to simulate it. To get Spike to work with this code, it should be informed of Hornet's memory address space. To accomplish this, RISCV-DV's `iss.yaml` file is modified:

```
- iss: spike
  path_var: SPIKE_PATH
  cmd: >
    <path_var>/spike --log-commits --isa=<variant> --priv=<priv> \
    -m0xf000:1,0x10000:0x8000,0x8010:1 -l <elf>
```

The `-m0xf000:1,0x10000:0x8000,0x8010:1` segment tells Spike that Hornet has 0x8000 bytes (32KB) of memory, starting from the address 0x10000. Since the compiled programs have their starting address set at 0xf000, even though the instructions start at 0x10000, that address is also included. The debug interface address of 0x8010 is also added to prevent Spike from giving an error when accessing it.

5.2.2.3 Log parsing and comparison scripts

The `spike_log_to_trace_csv.py` script takes a log file created by the Spike RISC-V simulator and converts it into a structured CSV file. It reads every instruction that was executed, along with its program counter, binary encoding, and any updates to general-purpose or control registers. The script uses regular expressions to extract the needed data from the log. It can also include full instruction details if the `-full_trace` option is enabled. It ignores the startup section and stops parsing after the `ecall` instruction. The CSV file it produces is useful for understanding the processor's behavior and is ready for comparison with other outputs.

The `trace_to_csv.py` script processes a simpler type of trace log and writes the information into a CSV format. It uses pattern matching to extract data like instruction

addresses, binary instructions, flags, and register values. The script skips empty lines and starts processing from the second line of the file. For each instruction, it stores one row in the CSV file that includes all the extracted fields. This script is useful when working with custom logs or logs that don't follow the Spike format. It makes the data easy to read and suitable for later comparison or review.

The `compare.py` script compares two CSV files that were generated from different simulation sources, one from RTL simulation and one from a reference simulator like Spike. It checks if both files show the same updates to registers and control/status flags at the same instruction addresses. To do this properly, the script standardizes register names (for example, changing `a0` to `x10`) and formats all hexadecimal values the same way. If there are mismatches in addresses, registers, or values, the script prints error or warning messages. At the end, it gives a summary of the result: either PASS, PASS with warnings, or FAIL. This helps verify that the processor under test behaves correctly.

Last two files are made for this project. However, `spike_log_to_trace_csv.py` file already exists in the RISCV-DV repository [4]. Unfortunately, an error was seen in the script that affects the CSR parsing. The problem was solved and pull request #1012 was created.

5.2.2.4 Main script

All the steps of the verification process, starting from RISCV-DV and Spike (which run together), then the RTL simulation, and finally converting and comparing the logs, are handled within one main script, written in the Bash scripting language. This script also contains several quality of life features, like running custom tests and indefinitely running RISCV-DV with new tests until an error is found (this one in particular was heavily used to find some rarer issues).

The script utilizes Xilinx Vivado for the RTL Simulation, and as such it needs to be installed on the system. A Vivado project, which contains the source files for Hornet, must be supplied to the script. Then, by generating a .tcl script with the appropriate parameters, Vivado can be called. This is accomplished as follows:

```
# Generate TCL script
cat > run_sim.tcl << EOF
# Open project
if {
[catch {open_project ${PROJECT_DIR}/${PROJECT_NAME}.xpr} result]
} {
    puts stderr "Error opening project: \$result"
    exit 1
}

# Optional: Load waveform configuration
if {[file exists ${WAVE_CONFIG}]} {
    open_wave_config ${WAVE_CONFIG}
}

# Launch simulation
launch_simulation

# Run simulation (adjust time as needed)
run ${VIVADO_DURATION}

# Close project and exit
close_project
exit
EOF
```

As bash scripts are usually run under Linux, the script expects the Linux version of Vivado. But for Windows users, who might be using Windows Subsystem for Linux (WSL) to run this script, the script automatically detects if it's running under WSL and calls the Windows version of Vivado in that case. This is done in the following way:

```
# Detect if running under WSL
if uname -r | grep -qi "microsoft"; then
    WSL=1
else
    WSL=0
fi

# Run simulation
echo "Starting RTL simulation at $(date)" | tee ${LOG_FILE}
if [ "$WSL" -eq 0 ]; then
    vivado -mode batch -source run_sim.tcl \
                -notrace | tee -a ${LOG_FILE}
else
    cmd.exe /C vivado -mode batch -source run_sim.tcl \
```

```
                        -notrace | tee -a ${LOG_FILE}
fi
```

Giving the same compiled code to Hornet and Spike is accomplished by converting the binary file sent to Spike to the appropriate format and giving it to Vivado. This is mainly based on the makefiles used by the original developers of Hornet [5], with only minor changes:

```
${CC32}-objcopy -O binary -j .init -j .text -j .rodata \
                -j .sdata asm_test/${TEST}_0.o final.bin
../../rom_generator final.bin
cp final.data ../../memory_contents/instruction.data
```

This simply takes the instruction and data portions of the binary, converts them into a plain text file of the instructions in 32-bit hexadecimal format, and saves it into the file `instruction.data`. By always overwriting the same data file, no new files need to be introduced into the Vivado project.

Finally, the generated log files are input into the csv conversion Python scripts, and then compared with the comparison Python script. All file name references use relative paths, so the project can be moved anywhere without breaking. The only criteria that needs to be followed is to put the Vivado project directory on the top directory of Hornet.

6. FIXING HORNET USING THE ENVIRONMENT

Initially, our environment returns hundreds of mismatched results. The vast majority of these are on floating point instructions, but there are some that are standard RV32I instructions as well. It should be noted that most of these are false positives, since the false result of a float operation can "pollute" the instructions that use its result, bloating the mismatch count.

6.1 Non-FPU Fixes

Even after taking these false positives to account, there were a few mismatches outside the Floating Point Unit (FPU), mainly in some edge cases of the integer Multiply/Divide unit. Almost all of the mismatches in this unit are due to missing edge case definitions in the fast result section of the unit, in `MULDIV_ctrl.v`. Let's go over these:

- According to the RISC-V Unprivileged Specification [1], the result of $0/0$ should be -1 (0xFFFFFFFF), which was missing in the fast result cases.
- Similar to this, the remainder instruction REM should return $x/0 = x$. This is also added to the fast result cases.
- For the instruction MULH, which returns the upper 32-bits of the multiplication result, the multiplication of a number with -1 or 1 weren't being considered. A negative result should give 0xFFFFFFFF for the upper 32-bits, whereas a positive result should give 0. For multiplying with 1, the sign bit of the input is used, whereas for -1 the sign bit of the input's 2's complement is used.
- For the instruction MULHU, which returns the upper 32-bits of unsigned multiplication, multiplying with -1 can't be considered a fast case, as it's unsigned equivalent is $2^{32} - 1$.
- Similar fixes were also done for the unsigned division and remainder instructions.

6.2 FPU Fixes

The remaining fixes all relate to the FPU. Let's start with smaller, edge cases:

6.2.1 Edge cases

- When doing comparison operations, the FPU doesn't take sign into account. This results in cases where -0.5 is considered a smaller number than -1. It only compares the exponent and significands' absolute values. To fix this issue, the relevant portion of the code in `fpu_compare.v` was changed as follows:

```
assign is_less  = (isZeroA & isZeroB) ? 1'b0        :
                  !is_sign_equal ? sign_A & !sign_B :
                  !is_exp_equal  ? (sign_A ? exp_A > exp_B :
                                             exp_A  < exp_B)   :
                  !is_sig_equal  ? (sign_A ? sig_A > sig_B :
                                             sig_A < sig_B)   :
                  1'b0;
```

- A similar issue to above was also seen in the comparison logic of `fpu_min_max.v`, and was corrected.

- There are several edge cases regarding the handling of Inf and NaN values. Some of these relate to the fact that non-standard NaN values are used. The RISC-V Specification defines the only valid NaN as the positive signed quiet NaN, which is 0x7fc00000. The edge cases are:

  - The result of the operation sqrt(-Inf) should return NaN, but it returns -Inf instead. In fact, sqrt should return NaN for all negative inputs. This was previously checked, but probably the logic for -Inf overrides this. The relevant part in `fpu_mds_ctrl.v` was fixed as follows:

```
            else if (isInfA) begin
                if (sign_A) begin
                    fast_res = {1'b0, 8'd255, `qNaN_sig};
                    mux_fastres_sel_temp = 1'b1;
```

```
                    overflow_fast = 0;
                    invalid_fast = 1;
```

  - In a similar fashion, in `fpu_add_sub_fast.v`, if the input of an operation is NaN, its significand is directly given to the output. Then, if the input is a Signaling NaN, the output also will be. This is not allowed, since the only NaN output allowed by RISC-V is the standard quiet NaN. Moreover, the sign bit of the inputs is also irrelevant in this case, since RISC-V only allows NaN output to be positive. Both of these issues were fixed.
  - The above fix was also applied to all the relevant cases in `fpu_mds_ctrl.v`.

- One of the sign injection instructions, fsgnjn.s, is responsible with injecting the inverse of rs2's sign into rs1. But, in this case, it's only inverting rs2's sign, and not actually using rs1's exponent and significand. The sign injection module simply determines the sign bit, which is used in the top module `fpu_arithmetic_top.v`. As such, the output should always have the exponent and significand of rs1. The relevant line was fixed.

- During sign injection operations, it is only needed to change the sign of the number. However, if a subnormal number's exponent was the output, it was changed from 0 to 1 because of the `fpu_decoder` changing it to 1. The `fpu_decoder` module was changed to give output of the unchanged version and making it the input for the `fpu_sign_inj` module.

- During the minimum and maximum operations, the FPU module needs to give the output without changing anything on the numbers, like the sign injection operation. However, for the subnormal numbers, the exponent was also changed, so the same fix was done for this module.

- In multiplication, multiplying a positive number by -0 should result in -0, and +0 for a negative number. Similarly, dividing a positive number by -0 should result in -Inf. To fix this issue, all relevant cases in the `fpu_mds_ctrl.v` are fixed, with the output using the `sign_O` signal to determine its sign, which is simply the XOR of the input signs.

- The min-max unit gave out the wrong output if one of its inputs were NaN or subnormal. This is caused by the fact that, despite taking the input as separate sign, exponent and significand fields, this module outputs the number in combined 32-bit float format. This can be seen in the following lines of `fpu_min_max.v`:

```
wire [31:0] A = {sign_A, exp_A, sig_A[22:0]};
wire [31:0] B = {sign_B, exp_B, sig_B[22:0]};
...
assign min_max_out = is_both_NaN  ? 32'h7fc00000   :
                     isNaNA       ? B              :
                     isNaNB       ? A              :
                     (min_or_max) ? (A_big ? A : B):
                     (A_big ? B : A);
```

  The reason why this outputs an invalid result is, when the subnormal numbers are being decoded in `fpu_decoder.v`, the exponent is assigned a value of 1, to normalize them. This is made with the assumption that these numbers will be operated on. But the min-max unit simply takes this value and re-outputs it, breaking that assumption. For this reason, this problem can be fixed by giving the raw exponent of the subnormals to this unit (which is 0). Since this affects all subnormals equally, the comparison logic of the min-max unit won't be affected.

- Similarly, when a subnormal number was added to 0, the result had an extra bit in the exponent. This is caused by the fact that `fpu_add_sub_fast.v` was also given the normalized exponent value of 1. This was fixed by directly giving the raw exponent to `fpu_add_sub.v`, which already normalizes the exponent for its non-fast outputs.

- In the `fpu_cvt_to_int.v` module, conversions to unsigned integers were being handled incorrectly. Normally, according to the spec, if a negative float number is converted to an unsigned integer the output must be 0. The module ignored this case. This was fixed by changing the relevant line from:

```
assign final_out = is_unsigned ? int_after_round :
(sign_A ? ~int_after_round + 1 : int_after_round);
```

  to:

```
assign final_out = is_unsigned ? (sign_A ? 32'h0 :
                                  int_after_round) :
                   (sign_A ? ~int_after_round + 1 :
                             int_after_round);
```

- In the same module, the overflow case was determined incorrectly for converting to a signed integer. Initially, only having a (true) exponent of greater than 31 is considered enough for overflow. But we can see that, among floating point values with (true) exponents of 31, even the smallest one is greater than the signed overflow limit of $2^{31} - 1$. As such, changing the overflow check to the following line fixed the issue:

```
assign is_overflow = is_unsigned ? actual_exp > 31 :
                                   actual_exp >= 31;
```

- In addition, the same module results in error when a number is given that is between -1.0 and 1.0 the result was always 0. However, because of the rounding mode, the result can also be -1 and 1. This is solved by changing the condition as the following:

```
assign cvt_to_int_out =  isNaNA ? (is_unsigned ? 32'hFFFF_FFFF :
 ↪  32'h7FFF_FFFF)   :
                         isInfA ? (is_unsigned ? (sign_A ? 32'h0
                          ↪  : 32'hFFFF_FFFF) : (sign_A ?
                          ↪  32'h8000_0000  : 32'h7FFF_FFFF)) :
                         isZeroA ? 32'h0 :
                         is_exp_neg ? (exp_neg_out) :
                         is_overflow ? (is_unsigned ? (sign_A ?
                          ↪  32'h0 : 32'hFFFF_FFFF) : (sign_A ?
                          ↪  32'h8000_0000 : 32'h7FFF_FFFF)):
                          ↪  final_out;
```

- In the module `mds_final_normalizer.v`, when the result of a division is subnormal, and if the significand has 1 in its MSB, this bit is destroyed and the result is wrong. This bit is called the "hidden bit", and is normally 0 for subnormals. This module seems to cut this bit off instead of shifting the bits right, invalidating the number. The relevant line in the module is fixed as follows:

```
assign final_sig = of_fin_norm ? 23'd0 :
                ((proNorm_exp == 8'd0 && sig_after_round[23]) ?
                sig_after_round[23:1] : ( sig_after_round[24] ?
                                          sig_after_round[23:1] :
                                          sig_after_round[22:0] ));
```

- When two equal numbers were subtracted from each other, or when two exact opposite numbers were added together, the output is equal to 0.0f. But the sign of the output depends on the rounding mode. The following adjustment was done to fix this issue:

```
always @(*) begin
    if(sub_op) begin
    //If subtraction, result is 0 if signs are the same.
        if(!(sign_A ^ sign_B)) begin
            //Result is -0.0 if rounding mode is RDN, else +0.0
            if(rounding_mode == 3'b010) sign_O_equal = 1'b1;
            else sign_O_equal = 1'b0;
        end
        else
        //Otherwise the result will be a non-zero number,
        //with the sign equal to the sign of A
            sign_O_equal = sign_A;
    end
    else begin
    //If addition, result is 0 if signs are opposite
        if((sign_A ^ sign_B)) begin
            //Result is -0.0 if rounding mode is RDN, else +0.0
            if(rounding_mode == 3'b010) sign_O_equal = 1'b1;
            else sign_O_equal = 1'b0;
        end
        else
        //Otherwise the result will be a non-zero number,
        //with the sign equal to the sign of A
            sign_O_equal = sign_A;
    end
end
...
...
//sign_O_equal is used as follows
assign sign_O =(exp_A_adjusted == exp_B_adjusted) ?
               (sig_A_adjusted == sig_B_adjusted ? sign_O_equal :
               (sig_A_adjusted > sig_B_adjusted ? sign_A :
                                                  eff_sign_B) ) :
               (exp_A_adjusted >  exp_B_adjusted) ? sign_A :
```

```
                                                    eff_sign_B;
```

6.2.2 Rounding fixes

There are several issues relating to the floating point rounding modes. Floating point numbers are an approximation to the infinitely precise real numbers. As such, there are Floating point rounding modes that define how numbers are rounded when the exact result can't be represented. They exist to control rounding behavior and minimize cumulative errors in computations. The modes available in RISC-V can be seen in the Table 6.1

Encoding	Rounding Mode Description
000	RNE – Round to Nearest, ties to Even
001	RTZ – Round towards Zero
010	RDN – Round Down (towards $-\infty$)
011	RUP – Round Up (towards $+\infty$)
100	RMM – Round to Nearest, ties to Max Magnitude
101	Invalid – Reserved for future use
110	Invalid – Reserved for future use
111	DYN – Dynamic mode selected by instruction's `rm` field

Table 6.1 : Rounding mode encoding of RISC-V [1].

The main issues regarding these modes are:

- In RISC-V, a floating point instruction can choose which rounding mode it wants to use by utilizing the [14:12] bits. However, this is not the only way to set the rounding mode. The alternative is to set these bits to 111, which enables **Dynamic Mode**. In Dynamic Mode, the rounding mode is read from the CSR (Control and Status Register) named "frm". The value of this register can be set with the appropriate CSR writing instruction.

  If we look at the existing solution in Hornet, it's seen that in the control unit, the dynamic rounding mode is tied to the RNE mode. So it's not dynamic at all.

  To address this issue, the required registers were created in the CSR unit, at their required addresses (According to the privileged spec [9]). The wrong assignment in the control unit was fixed. Then, the rounding mode stored in the CSR unit was

sent to the FPU. Logic was added to set the rounding mode to the CSR value if the instruction had 111 as its rounding mode bits. The code to do so is:

```
reg [2:0] round_override;

always @(*) begin
        if(rounding_mode == 3'b111)
            round_override <= csr_dynamic_rounding_mode;
        else
            round_override <= rounding_mode;
end
```

By passing the `round_override` signal to the submodules instead of the previous `rounding_mode` signal, the fix is complete.

- There have been cases where a 1 bit rounding correction was added when it wasn't supposed to, along with the exact opposite case. To understand these issues, a concept in floating point rounding called GRS bits must be explained. These are three additional bits of precision, beyond the 23-bit mantissa that is used in displaying the result. GRS bits correspond to:
  - **G (Guard) Bit**: Bit immediately after the LSB of the significand.
  - **R (Rounding) Bit**: Bit after the Guard bit. Along with Guard bit, used in "Round to Nearest" modes.
  - **S (Sticky) Bit**: The OR of any extra bits of calculated result. In floating point implementations, arithmetic operation results usually have precision significantly higher than the significand. The extra precision is utilized in rounding, in the form of the Sticky bit.

  In Hornet's FPU, there are several issues with the generation and use of these GRS bits. Most of these issues concern the RTZ, RUP and RDN modes. These issues and their fixes are:

  - When rounding in RUP mode, any positive number is assumed to get rounded up by 1 bit, since the mode demands it. This approach sometimes results in incorrect results, because for exact results (without any extra precision that needs to be rounded), rounding shouldn't be done. To detect exactness, the

G, R and S bits are ORed, and rounding is only applied if the result is 1. A similar fix is also done for the RDN mode. The resulting code from one of the rounding units is as follows:

```
    3'b010:begin
            if(sign_O == 1'b0)
                round_out = 1'b0;
            else
                if(|LGRS[2:0]) round_out = 2'b01;
                else round_out = 2'b00;
            end

    3'b011:begin
           if(sign_O == 1'b0)
                if(|LGRS[2:0]) round_out = 2'b01;
                else round_out = 2'b00;
           else
                round_out = 1'b0;
            end
```

- Another issue is caused by the incorrect preservation of the Sticky bit in the division and multiplication modules. The Sticky bit, due to its inherent property, must be preserved when shifting the number to the right. In division and multiplication modules, various normalization operations may shift the result right. In these cases, extra logic was added to preserve the sticky bit. The following excerpt from `fpu_mul.v` shows the fix:

```
shift_amount = inExp_2C + 1;
    // Handle cases where shift_amount exceeds input width
    if (shift_amount > 47) begin
        SigTemp = 47'b0;
        sticky = |inSig;  // All bits shifted out
    end
    else begin
        SigTemp = inSig >> shift_amount;
        // Capture OR of bits shifted out (LSBs of original)
        sticky = |(inSig & ((1 << shift_amount) - 1));
    end
```

- Regarding the RTZ, RUP and RDN modes, especially in addition and subtraction, there are several edge cases that cause off by one errors.

In some cases, where the second operand is very small, it can be normalized to 0, and have no effect on the first operand. But in the above mentioned rounding modes, even these seemingly insignificant numbers have an effect. Consider the round up case. If we add two numbers, one of them small enough to not affect the result, the result is still very slightly higher than the first number, even if we can't represent it.

In this case, the round up mode will round this result up to the 1-bit higher floating point number. There are similar cases for other rounding modes, and for subtraction. All together, the resulting rounding module, `fpu_add_sub_rounder.v` has been modified as follows:

```
...
//By setting second bit of round_out to 1, we can subtract 1 from the result instead
3'b001:begin //RTZ
       if(second_operand_zero) begin
           if(sign_less == 1'b0 && sign_O == 1'b1) round_out = 2'b11;
           //In addition, adding a very small amount to a
           //negative number will round it up if mode is RTZ
           else if(sign_less == 1'b1 && sign_O == 1'b0) round_out = 2'b11;
           //In subtraction, subtracting a very small amount from a
           //positive number will round it down if mode is RTZ
           else round_out = 2'b00;
       end
       else round_out = 2'b00;
   end

3'b010:begin //RDN
       if(sign_O == 1'b0) //For positive numbers
           if(sign_less == 1'b1 && second_operand_zero) round_out = 2'b11;
           //If we subtract a very small amount from a positive number,
           //RDN will pull it down (so magnitude decreases)
           else round_out = 2'b0;
       else begin
           if(sign_less == 1'b1 && second_operand_zero) round_out = 2'b01;
           //If we subtract a very small amount from a negative number,
           //RDN will pull it down (so magnitude increases)
           else if(|LRS[1:0]) round_out = 2'b01; //Round only if R or S bit is set
           else round_out = 2'b00;
       end
   end

3'b011:begin //RUP
      if(sign_O == 1'b0) begin//For positive numbers
           if(sign_less == 1'b0 && second_operand_zero) round_out = 2'b01;
           //If we add a very small amount to a positive number,
           //RUP will pull it up (so magnitude increases)
           else if(|LRS[1:0]) round_out = 2'b01; //Round only if R or S bit is set
           else round_out = 2'b00;
       end
       else begin
           if(sign_less == 1'b0 && second_operand_zero) round_out = 2'b11;
           //If we add a very small amount to a negative number,
           //RUP will pull it up (so magnitude decreases)
           else round_out = 2'b00;
       end
   end
...
```

- Similar cases to above also appear in `fpu_cvt_to_int.v`, the integer conversion unit. In this case, when a very small negative number is rounded down

in the RDN mode, the signed integer output must be -1. Similarly, when a very small positive number is rounded up in the RUP mode, the integer output must be 1.

- These three modes have some specific behavior around Inf values that weren't taken into account. Specifically, when the result is overflowing, in the RDN mode, the output is rounded down from +Inf to the highest possible non-infinite float number. A similar case can be seen with the RUP mode and -Inf. The RTZ mode affects both +Inf and -Inf in the same way. The logic to fix these issues is, from the file `fpu_mds_top.v`:

```
if(overflow) begin //Overflow edge cases
    if(sign_O) begin //For negative numbers
        //For RTZ or RUP, -Inf rounds to largest negative number
        if((rounding_mode == 3'b001) || (rounding_mode == 3'b011))
            OUT_reg <= 32'hff7fffff;
        else OUT_reg <= muldiv_sqrt;
    end
    else begin //For positive numbers
        //For RTZ or RDN, +Inf rounds to largest positive number
        if((rounding_mode == 3'b001) || (rounding_mode == 3'b010))
            OUT_reg <= 32'h7f7fffff;
        else OUT_reg <= muldiv_sqrt;
    end
end
```

- In underflow cases, issues similar to above were detected and corrected.

6.2.3 Fixing insufficient precision issues

There have also been cases where very small inputs to the division and square root units caused loss of precision in the results. Both of these operations were implemented with an iterative algorithm, with a fixed number of iterations. It might be the case that there are not enough iterations for some inputs to gain meaningful output precision. Let's consider the issue in the division unit first:

- When the divisor is a subnormal number, the value is normalized by shifting until the MSB is 1. The shift amount is stored as the `offsetB` signal. This solves the cases where a large number is divided by a subnormal.

- The dividend on the other hand, is not normalized at all. So when a subnormal value is used as the dividend, it has too many leading zeroes. This causes the division algorithm to waste several cycles before any change in the internal values occur, and as such the iteration count is reached before sufficient result digits get computed.
- To fix this problem, the dividend is also normalized, and the shift amount is stored as `offsetA`. Then, since shifting the dividend left would increase the exponent of the result, this shift amount is subtracted from the final exponent.
- There are two instances where this fix causes problems. These cases, and their fixes are:
  - When the `is_exp_underflow` flag is raised, the exponent is underflowing. That is to say, the resulting exponent would have to be less than 0. In this instance, subtracting `offsetA` wouldn't help, as the exponent is already zero. Because of this, the significand is shifted right by `offsetA` instead.
  - When `offsetA` is greater than `inExp`, the subtraction of `offsetA` would cause an underflow. In this instance, a similar fix to above is done, but this time the significand is shifted right by only `offsetA - inExp`.
- Altogether, the division normalizer code that these fixes are applied to, is as follows:

```
always @(*)// normalization
begin
    if (!is_exp_underFlow)
    begin
        if(norm_underflow) begin
            shift_amount = (offSetA - inExp);

            // Handle cases where shift_amount exceeds input width
            if (shift_amount > 27) begin
                SigTemp = 27'b0;
                sticky = |inSig;  // All bits shifted out
            end
            else begin
                SigTemp = inSig >> shift_amount;
                // Capture OR of bits shifted out (LSBs of inSig)
                sticky = |(inSig & ((1 << shift_amount) - 1));
```

```
            end

            ExpTemp = 8'b0;
            underflow = 1'b1;
        end
        else if (inSig[26] == 1'b1) // if mantissa carry is 1
        begin
            ExpTemp = inExp + (offSetB  - 1) - (offSetA - 1);
            SigTemp = inSig;
            sticky = SigTemp[0];
            underflow = 1'b0;
        end

        else if (inExp == 1 | inExp <= zeroCount)
        begin

            ExpTemp = 8'b0;
            SigTemp = inSig << inExp - 1;
            sticky = SigTemp[0];
            underflow = 1'b1;

        end
        else
        begin
            ExpTemp = inExp - zeroCount + (offSetB  - 1)
                                        - (offSetA - 1) ;
            SigTemp = inSig << zeroCount;
            sticky = SigTemp[0];
            underflow = 1'b0;
        end
    end
    else
    begin
        //SigTemp         = inSig >> inExp_2C + 1;
        shift_amount = inExp_2C + 1 + (offSetA - 1);
        // Handle cases where shift_amount exceeds input width
            if (shift_amount > 27) begin
                SigTemp = 27'b0;
                sticky = |inSig;  // All bits shifted out
            end
            else begin
                SigTemp = inSig >> shift_amount;
                // Capture OR of bits shifted out (LSBs of inSig)
                sticky = |(inSig & ((1 << shift_amount) - 1));
            end

        if(SigTemp[26]) begin
            ExpTemp = 1;
            underflow = 1'b0;
```

```
        end
        else begin
            ExpTemp = 0;
            underflow = 1'b1;
        end
    end
end
```

Next, let's consider the issues seen in the square root unit:

- When inputting a very small subnormal number, like 0x00000001, the square root operation does not return the bottom 8-10 bits of the correct result. The issue here, much like the division case, is due to wasted iterations in the algorithm, due to too many leading zeros in the input.

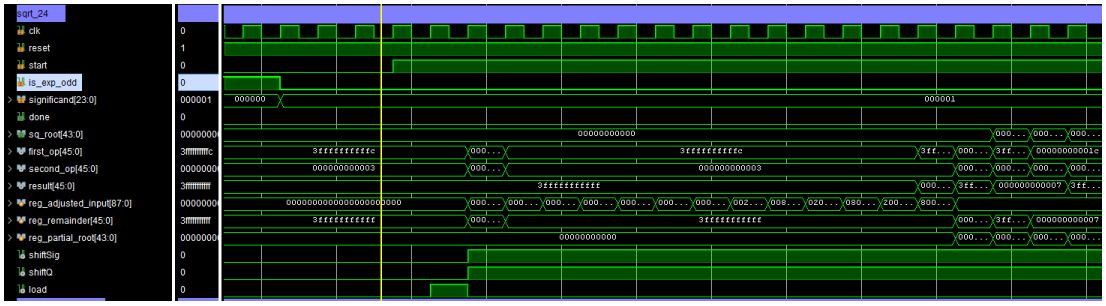


Figure 6.1 : Wasted algorithm cycles. Note that, for several cycles after the start signal is given, signal sq_root is unchanged.

- Unlike division however, normalizing the input before processing isn't effective here. This is because, unlike division, the square root algorithm used here requires the significand to be unchanged, otherwise the results aren't obtained correctly.

- The only solution to this issue, is to increase the iteration count. To do so however, the widths of the square root unit's internal signals must also be increased. All these values are defined in the `sqrt_24.v` file.

- According to [6], a recurrent algorithm that calculates two bits of the result per cycle is used. The output of the module is 27 bits, which gives the required precision and rounding bits. As such, the internal signal that holds the input is 54 bits long, and is shifted to the left by 2 places each cycle. All of these signals' widths need to be increased in order to keep producing correct results.

- After testing with very small floating point numbers, an iteration count of 44 was deemed suitable. This gave correct results for the smallest 10 million subnormal numbers that were tested, and also gave the correct results under all other rounding modes.

- After increasing the iteration count to 44, the following signals were modified as well:

  - **sq_root**: The result signal, increased from 27 to 44 bits.
  - **first_op, second_op, result and reg_remainder**: The internal signals used in the calculation, increased from 29 to 46 bits.
  - **reg_partial_root**: Another calculation parameter, increased from 27 to 44 bits.
  - **reg_adjusted_input**: The extended and shifted version of the input, increased from 54 to 88 bits.

  All the shift operations applied to these signals, and the specific bit indexes of these signals that were used in the calculation process, were also adjusted to the extended sizes.

- The top 26 bits of the result were sent as is to the output, whereas the remaining bits were ORed into a sticky bit.

- Before sending to the output, the result is normalized in the `fpu_sqrt.v` module. To accomplish this, the leading zeros of the result is counted, and the significand is shifted by this amount. Additionally, this shift amount is subtracted from the exponent. This was initially done by a 27-bit counter, but had to be increased to a 44-bit one as follows:

```
module lzc44(
    input  wire [43:0] x,
    output wire [5:0]  z
);

wire      a0, a1, a2, a3, a4, a5;
wire[2:0] z0, z1, z2, z3, z4;
wire[1:0] z5;
reg [5:0] z_result;
```

```
// 5 × lzc8
lzc8 U0 (.x(x[43:36]), .a(a0), .z(z0));
lzc8 U1 (.x(x[35:28]), .a(a1), .z(z1));
lzc8 U2 (.x(x[27:20]), .a(a2), .z(z2));
lzc8 U3 (.x(x[19:12]), .a(a3), .z(z3));
lzc8 U4 (.x(x[11: 4]), .a(a4), .z(z4));

// Final 4 bits with lzc4
lzc4 U5 (.x(x[3:0]), .a(a5), .z(z5));

// Priority encoder
always @(*) begin
    if (!a0)
        z_result = 6'd0  + z0;
    else if (!a1)
        z_result = 6'd8  + z1;
    else if (!a2)
        z_result = 6'd16 + z2;
    else if (!a3)
        z_result = 6'd24 + z3;
    else if (!a4)
        z_result = 6'd32 + z4;
    else if (!a5)
        z_result = 6'd40 + z5;
    else
        z_result = 6'd44;  // all bits zero
end

assign z = z_result;

endmodule
```

This utilizes the existing zero counter sub-modules, just like the previous 27-bit counter.

- It was seen that, if the sticky bit was calculated after normalization, some inputs gave wrong outputs in the RUP (Round up) rounding mode. On the other hand, if the sticky bit was calculated before normalization, this caused issues with the RNE (Round to nearest) rounding mode.

  To fix this issue, a mux was created that chooses between these alternatives depending on the input rounding mode. This fix has given correct results for over 10 million different subnormal inputs, but has no theoretical basis. As such, further investigation to the topic may be required.

7. AI APPLICATION AS A BENCHMARK

After confirming that the Hornet Core works without any error, we then proceed to writing an AI application and confirming that the Hornet Core works while doing real life examples too. MNIST dataset [14] was chosen as the dataset for our purpose. It was chosen since it is a broadly known and one of the most worked on dataset. Our methodology for this purpose will be using TensorFlow [15] to train the model. Then, the inference step will be written in the C programming language and will be compiled for a general purpose computer to verify that the code works. Lastly, the project will be compiled for RV32IMF and run on Hornet Core.

7.1 Training the Model Using TensorFlow

Then the model training on the TensorFlow [15]. The model is trained using the MNIST dataset [14]. This model was chosen because it is broadly used across many number detection projects. MNIST dataset has 70000 28x28 pixel grayscale images. Our first approach was to use 28x28 inputs for the system. However, when this was used it was reaching very high data sizes that the model will not be able to git to Hornet's memory. The input size was reduced to 14x14 pixels with an interpolation operation. Then a hidden layer was implemented using the ReLU function with 32 neurons. The output layer has 10 neurons, where every neuron gives the chance of its corresponding digit, with the softmax function as the output function. The model can be seen in the Figure 7.1. The training was done using Python and the code exports the weights of layers as a C library file. The code can be seen below:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
import tensorflow.image as tfi

# Load MNIST dataset and resize images to 14x14 using interpolation
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
x_train = tfi.resize(tf.expand_dims(x_train, -1), [14, 14]).numpy()
x_test = tfi.resize(tf.expand_dims(x_test, -1), [14, 14]).numpy()

# Normalize and flatten
x_train = x_train.astype(np.float32) / 255.0
x_test = x_test.astype(np.float32) / 255.0
x_train = x_train.reshape(-1, 14*14)
x_test = x_test.reshape(-1, 14*14)

# Define the model for detecting numbers
model = models.Sequential([
    layers.Input(shape=(14*14,)),
    layers.Dense(32, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile and train
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10, batch_size=128, \
validation_split=0.1)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)

# Export weights to C (The path is with respect to main folder)
def export_to_c(model, filename="test/mnist/mlp_mnist1_weights.h"):
    with open(filename, "w") as f:
        for i, layer in enumerate(model.layers):
            if not layer.get_weights():
                continue
            weights, biases = layer.get_weights()
            f.write(f"// Layer {i} weights {weights.shape}\n")
            f.write(f"const float
             ↪  layer{i}_weights[{weights.shape[0]
             ↪  }][{weights.shape[1]}] = {{\n")
            for row in weights:
                f.write("  {" + ", ".join(f"{v:.8f}f" for v in
                 ↪  row)+ "},\n")
            f.write("};\n\n")
            f.write\(f"const float
             ↪  layer{i}_biases[{biases.shape[0]}] = {{\n")
            f.write("  " + ", ".join(f"{v:.8f}f" for v in biases)+
             ↪  "\n};\n\n")
```

```
export_to_c(model)
```

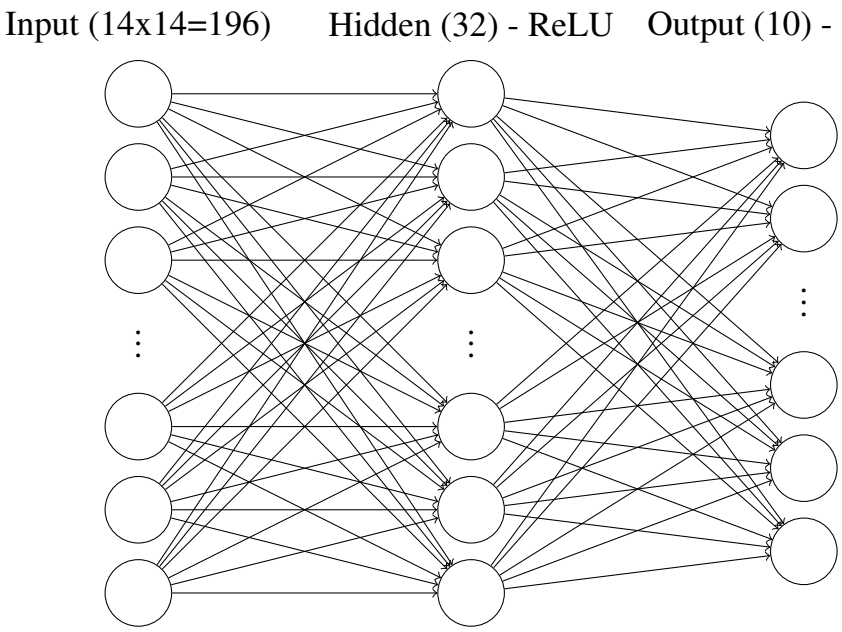


Figure 7.1 : Basic MLP model designed to detect digits.

Also, another Python script was written to load an image and export it as a C library file. The code can be seen below:

```
import numpy as np
from tensorflow.keras.datasets import mnist
from PIL import Image
import tensorflow as tf
import tensorflow.image as tfi

(_, _), (x_test, y_test) = mnist.load_data()

# Select one image (you can change index)
index = 4352
label = y_test[index]
image = x_test[index]

# Resize image to 14x14 using bilinear interpolation
image = image.astype(np.float32)
resized = tfi.resize(tf.expand_dims(image, axis=-1), [14,
↪  14]).numpy().squeeze()

# Export as tif file
resized_uint8 = (resized * (255.0 /
↪  resized.max())).astype(np.uint8)
```

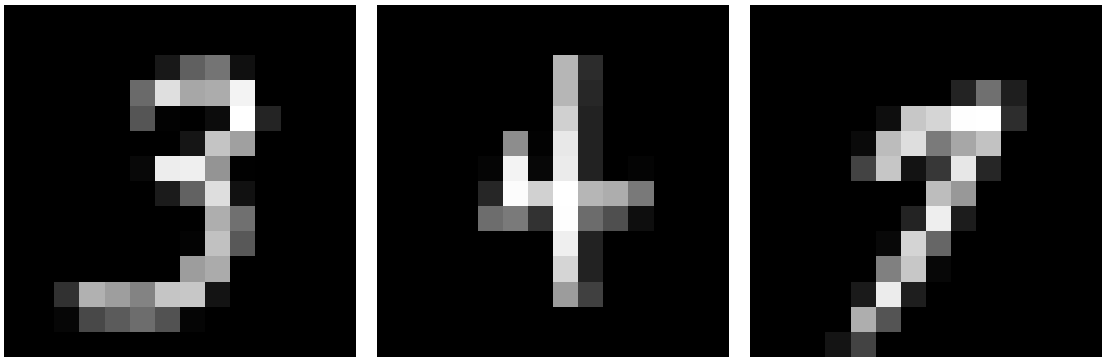
```
img = Image.fromarray(resized_uint8)
img.save("test/mnist/mnist_resized_14x14.tif", format="TIFF")

# Normalize to [0, 1]
resized = resized / 255.0
flattened = resized.flatten().astype(np.float32)

print(label)

# Export to C (The path is with respect to main folder)
with open("test/mnist/input_image.h", "w") as f:
    f.write("#ifndef INPUT_IMAGE_14X14_H\n#define
    ↪  INPUT_IMAGE_14X14_H\n\n")
    f.write("const int label = {};\n".format(label))
    f.write("const float input_image[196] = {\n")
    for i in range(196):
        f.write("  {:.8f}f".format(flattened[i]))
        if i < 195:
            f.write(",")
        if (i + 1) % 14 == 0:
            f.write("\n")
    f.write("};\n\n#endif\n")
```

The resized images can be seen in Figure 7.2. Although reducing the pixel size from 28x28 to 14x14 might have resulted in as bad accuracy because of having much less features, it did not happen and we still get about 95.18% accuracy which is sufficient for our purposes.



((a)) Digit 3 ((b)) Digit 4 ((c)) Digit 7

Figure 7.2 : Some digits from MNIST dataset that are reduced to 14x14 pixels.

7.2 Running the Model on the Hornet Core

After the model was trained without any issues it was time to implement it for our Hornet Core. Firstly, the C code was written for and tried to be run on a general use

computer. Exponentiation function was realized as Taylor Series sum. The ReLU and softmax functions were also implemented as matrix multiplication. The final C code can be seen below:

```
#include <stdio.h>
#include "mlp_mnist_weights.h"
#include "input_image.h"

#define INPUT_SIZE 196
#define HIDDEN_SIZE 32
#define OUTPUT_SIZE 10

float expf_approx(float x) {
    float result = 1.0f;
    float term = 1.0f;
    int n;

    for (n = 1; n <= 50; n++) {
        term *= x / n;
        result += term;
    }

    return result;
}

float relu(float x) {
    return x > 0 ? x : 0;
}

void softmax(const float* input, float* output, int size) {
    float max_val = input[0];
    for (int i = 1; i < size; ++i) {
        if (input[i] > max_val) max_val = input[i];
    }

    float sum = 0.0f;
    for (int i = 0; i < size; ++i) {
        output[i] = expf_approx(input[i] - max_val);
        sum += output[i];
    }
    for (int i = 0; i < size; ++i) {
        output[i] /= sum;
    }
}

void dense_relu(const float* input, float* output,
                const float weights[][HIDDEN_SIZE], const float*
                 ↪  biases,
```

```
                int in_size, int out_size) {
    for (int i = 0; i < out_size; ++i) {
        float acc = biases[i];
        for (int j = 0; j < in_size; ++j) {
            acc += input[j] * weights[j][i];
        }
        output[i] = relu(acc);
    }
}

void dense_softmax(const float* input, float* output,
                   const float weights[][OUTPUT_SIZE], const float*
                    ↪  biases,
                   int in_size, int out_size) {
    float logits[OUTPUT_SIZE];
    for (int i = 0; i < out_size; ++i) {
        float acc = biases[i];
        for (int j = 0; j < in_size; ++j) {
            acc += input[j] * weights[j][i];
        }
        logits[i] = acc;
    }
    softmax(logits, output, out_size);
}

int mlp_infer(const float input[INPUT_SIZE]) {
    float hidden[HIDDEN_SIZE];
    float output[OUTPUT_SIZE];

    dense_relu(input, hidden, layer0_weights, layer0_biases,
     ↪  INPUT_SIZE, HIDDEN_SIZE);
    dense_softmax(hidden, output, layer1_weights, layer1_biases,
     ↪  HIDDEN_SIZE, OUTPUT_SIZE);

    int predicted = 0;
    float max_val = output[0];
    for (int i = 1; i < OUTPUT_SIZE; ++i) {
        if (output[i] > max_val) {
            max_val = output[i];
            predicted = i;
        }
    }

    return predicted;
}

int main() {
    volatile int result = mlp_infer(input_image);
```

```
    printf("Predicted digit: %d\n", result);
    return 0;
}
```

As a test, digit four in the Figure 7.2 was given to the model. The results came up satisfactory as `Predicted digit: 4`. After confirming the C code works as expected, then the focus was shifted on the Hornet Core. The code was changed so that it is sent 1 to debug interface if the result is found as expected or it is sent 0 to debug interface when the result was found wrong. The code was compiled and run on simulation to see the results. The results followed the results we found on the general use computer. In addition, the result `Success!` was seen in the terminal meaning that it is sent 1 to the debug interface. After that, the test was run in the test environment to see if any mismatch came up with the instruction set simulator. No difference were examined expect a CSR flag mismatch, as it is expected. The results can be seen in Figure 7.3. Therefore, it can be said that the Hornet Core with its floating point unit is ready to use in real life examples without any error occurring because of the processor itself.

```
WARNING: CSR Flag Mismatch at 0x00010140:
fsub.s  fa0, fa0, fs0, CSR=c1_fflags
RTL:0x00000003  ISS:


============================================================
                    PASS (With Warnings)
============================================================
ERROR:0
WARNING:1
```

Figure 7.3 : Comparison result of MLP model.

8. REALISTIC CONSTRAINTS AND CONCLUSIONS

8.1 Practical Application of this Project

As a result of this project, the in-house RISC-V core of ITU, Hornet, has gained a functionally checked and corrected Floating Point Unit. With this project's results as a springboard, further research and additions can be done to the core. Thanks to the verification environment created, these further additions can be more easily and completely checked compared to before. The final version of the project can be reached from [16].

8.2 Realistic Constraints

8.2.1 Social, environmental and economic impact

RISC-V is a license-free ISA. This means that companies or groups do not have to pay for a licensing fee to produce and/or sell RISC-V processors.

8.2.2 Cost analysis

The RTL simulator used in this Project, Xilinx Vivado is free to use for educational purposes and for smaller FPGA models. If the design is extended enough to necessitate the use of larger/faster FPGA models not in the free license support list, a Vivado license fee may be required.

8.2.3 Standards

Throughout this project, the RISC-V Unprivileged [1] and Privileged Specifications [9] were followed. The floating-point implementations within these specifications follow the IEEE-754 Floating Point standard [8].

8.2.4 Health and safety concerns

This project does not involve health and safety concerns.

8.3 Future Work and Recommendations

There are several possible additions possible. Firstly, the missing multiply-accumulate instructions can be implemented. Apart from that, other floating point extensions can be added without much extra work. The double-precision float extension, D, might require some extra logic, but the recently added BF16 extension can be implemented with minimal changes.

Apart from ISA level additions, a standalone AI accelerator unit can be designed and integrated into the core. Moreover, further optimizations to the design, for example adding an extra pipeline stage in the FPU, can be done to improve the performance of the core.

REFERENCES

[1] **Waterman, A. and Asanović, K.** The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture Version 20240411, https://riscv.org/specifications/ratified/, (Accessed: Oct. 13, 2024).

[2] **Accellera Systems Initiative**, Universal Verification Methodology (UVM) 1.2 User's Guide, https://www.accellera.org/downloads/standards/uvm, (Accessed: Mar. 17, 2025).

[3] **lowRISC**, (2017), Ibex Verification, https://github.com/lowRISC/ibex/blob/master/doc/03_reference/verification.rst, (Accessed: Oct. 13, 2024).

[4] **Google**, (2020), RISCV-DV, https://github.com/chipsalliance/riscv-dv, (Accessed: Oct. 13, 2024).

[5] **Tozlu, Y.S. and Yılmaz, Y.** Design and Implementation of a 32-bit RISC-V Core, https://web.itu.edu.tr/~orssi/thesis/2021/YavuzTozlu_bit.pdf, (Accessed: Oct. 13, 2024).

[6] **Daysal, S. and Tuzcu, M.E.** Extending the Instruction Set of RISC-V Processor for Floating Point Arithmetic, https://web.itu.edu.tr/~orssi/thesis/2022/SalihDaysal_bit.pdf, (Accessed: Oct. 13, 2024).

[7] **RISC-V Foundation**, (2020), Spike: The RISC-V ISA Simulator, https://github.com/riscv/riscv-isa-sim, (Accessed: Oct. 13, 2024).

[8] **IEEE** IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 1–84.

[9] **Waterman, A. and Asanović, K.** The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 20240411, https://riscv.org/specifications/ratified/, (Accessed: Oct. 13, 2024).

[10] **RISC-V Foundation**, (2015), RISC-V GNU Compiler Toolchain, https://github.com/riscv-collab/riscv-gnu-toolchain, (Accessed: Jan. 07, 2025).

[11] **Schiavone, P.D.**, **Rossi, D.**, **Pullini, A.**, **Di Mauro, A.**, **Conti, F. and Benini, L.** Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX, *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pp.1–3.

[12] **SymbioticEDA**, (2016), RISC-V Formal Interface, https://github.com/SymbioticEDA/riscv-formal/, (Accessed: Oct. 13, 2024).

[13] **Kahan, W.** PARANOIA - Kahan's Floating Point Test Program, https://people.math.sc.edu/Burkardt/c_src/paranoia/paranoia.html, (Accessed: Jan. 09, 2025).

[14] **Deng, L.** The MNIST Database of Handwritten Digit Images for Machine Learning Research, *IEEE Signal Processing Magazine*, *29*(6), 141–142.

[15] **Abadi, M.**, **Agarwal, A.**, **Barham, P.**, **Brevdo, E.**, **Chen, Z.**, **Citro, C.**, **Corrado, G.S.**, **Davis, A.**, **Dean, J.**, **Devin, M.**, **Ghemawat, S.**, **Goodfellow, I.**, **Harp, A.**, **Irving, G.**, **Isard, M.**, **Jia, Y.**, **Jozefowicz, R.**, **Kaiser, L.**, **Kudlur, M.**, **Levenberg, J.**, **Mané, D.**, **Monga, R.**, **Moore, S.**, **Murray, D.**, **Olah, C.**, **Schuster, M.**, **Shlens, J.**, **Steiner, B.**, **Sutskever, I.**, **Talwar, K.**, **Tucker, P.**, **Vanhoucke, V.**, **Vasudevan, V.**, **Viégas, F.**, **Vinyals, O.**, **Warden, P.**, **Wattenberg, M.**, **Wicke, M.**, **Yu, Y. and Zheng, X.**, (2015), TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, https://www.tensorflow.org/, software available from tensorflow.org.

[16] **Özden, M.K. and Eroğlu, D.Z.**, (2025), Hornet RISC-V Core, https://github.com/mkozden/HornetRISC-V, (Accessed: Jun. 10, 2025).