# FPGA-Assisted Verification

Author: Grace Fieni, Ali Hoffmann, Xiran Wang
Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Hardware verification is traditionally done using simulation software, which is notoriously slow. Our project aims to speed up this process by synthesizing the design under test (DUT) and running tests on an FPGA instead, taking advantage of its fast hardware clock and reconfigurability. The goal is for our method to achieve 3x speedup over simulation. To test our method, we are creating a simple single-cycle processor as our DUT on the FPGA.**

***Index Terms*—FPGA, Hardware acceleration, Simulation, SoC, SystemVerilog, Verification**

## I. Introduction

Before today's SoC chips are manufactured and shipped, the design needs to be extensively tested using simulation software to verify functionality. This step, known as verification, is crucial because a bug that makes its way into a released chip could lead to recalls, potentially costing a company in the billions [1]. At the same time, verification also tends to be the bottleneck of the chip design process due to simulation often taking weeks or even months total to run [2]. Verification therefore takes up both significant compute resources and man-hours, not to mention the increasing complexity of designs only exacerbates the problem.

Our project aims to speed up verification by using custom hardware, instead of simulation software, to run tests. More specifically, we will be using the DE2-115 FPGA, which is a reconfigurable and relatively cheap integrated circuit. The benefits of this approach are two-fold: by leveraging the fast hardware clock of the FPGA, tests can complete in shorter times than they would in simulation, and the design is tested on actual hardware. For our project, the goal is to achieve runtime speedup of 3x over VCS simulation for test cases of various sizes (ranging from one instruction to 20,000 instructions). The test cases will be randomizable and customizable by the user.

## II. Design Requirements

For our system to be useful, there are numerous requirements we need to satisfy. We divide the requirements into three categories: performance, functionality, and ease of use.

### A. Performance

The most critical of our requirements is about performance, prompted directly by the problem at hand. As mentioned in the introduction, our goal is to achieve 3x speedup in comparison to the traditional method of verification: simulation. This goal is motivated by the fact that many tests in industry take 24 hours to complete. Reducing the runtime to 8 hours allows tests to run overnight, meaning a verification engineer can analyze the results when they come to work next morning.

We'd like to note that while simulation runtime consists only of processing time by the simulator, runtime for our approach consists of communication time from a PC to the FPGA and back, in addition to computation time on the FPGA (see section III for system dataflow). Communication is the bottleneck of our system, and we chose to use Gigabit Ethernet as our communication channel to maximize our chances of meeting speedup (see section V for communication tradeoff studies).

### B. Functionality

Our next set of requirements deal with the basic functionality of our system, a.k.a. what sort of tests we can support. We will create our own design under test (DUT)—a single-cycle processor supporting a subset of RISC-V instructions—and use our FPGA framework to test this DUT. (See section IV for DUT details). Our framework naturally must support testing all instructions implemented by the DUT. Furthermore, we will support running test cases of any size ranging from one to 20,000 instructions. Users can use small test sizes to isolate bugs, while large test sizes can be used to stress the design.

### C. Ease of use

Our last set of requirements relate to user experience while working with our framework. Our users will be able to both customize and randomize the test cases being fed to the DUT. This method of combining direct and indirect testing is known as constrained random verification. It is widely used in simulation-based approaches for its effectiveness at discovering both anticipated and unanticipated bugs. Therefore, we would like to provide the same experience in our framework. More specifically, we will allow the user to choose the number of instructions per test, which instructions to test, which registers to write to, etc., while unspecified parameters are randomized.

Another feature we will provide is cycle by cycle correctness checks of the DUT's output against that of a golden model (a C model of the design). This gives the user an exact pinpoint of which instruction the DUT begins faulting at, thereby easing their debugging effort, in contrast to offering only final output comparisons (which will require the user to trace through the test themselves to discover the divergence point).

Finally, if we have time at the end, we would like to implement a GUI to display to the user statistics about tests they have run (how many times each instruction has been tested, failure rates, etc.). This is to help with coverage closure—determining if enough tests that exercise different portions of the DUT have been run.

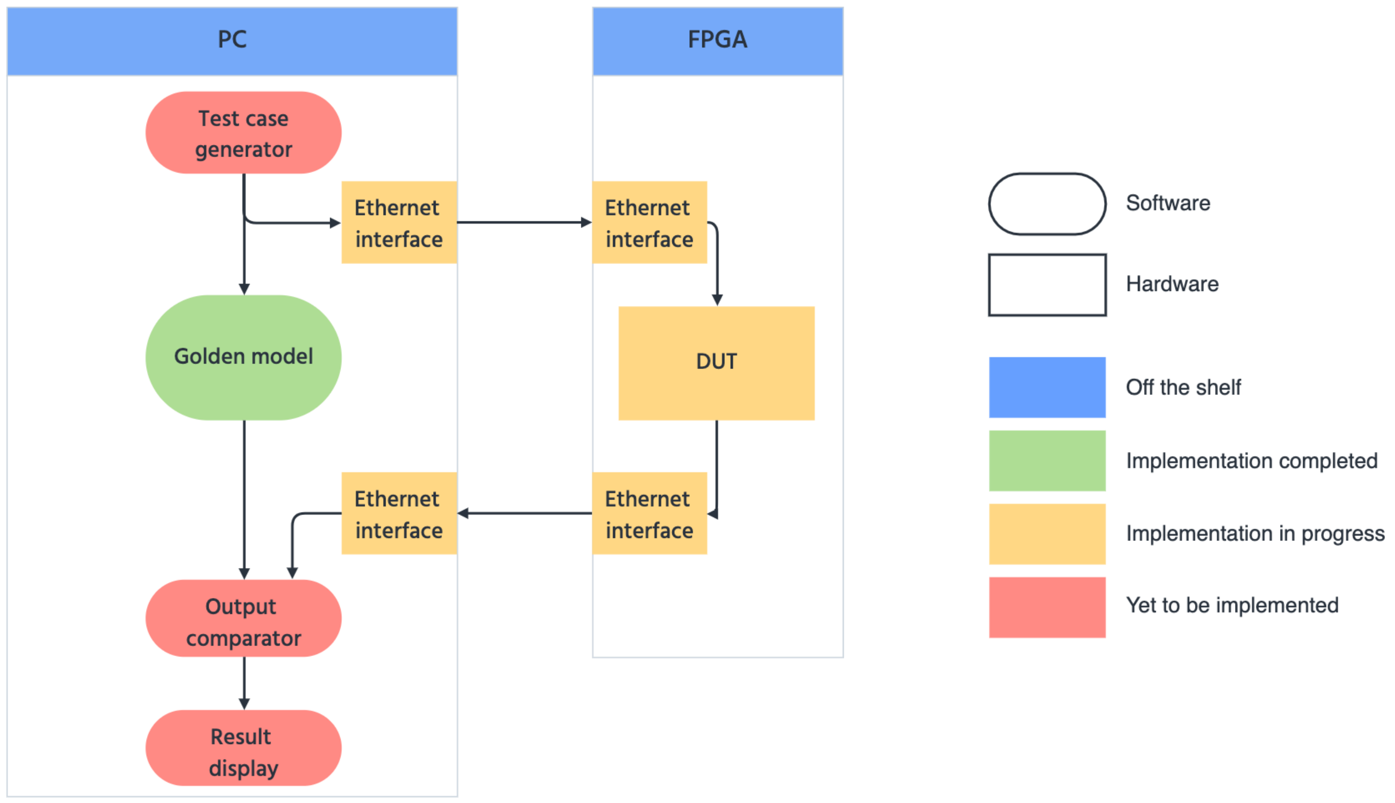


Fig. 1: System overview

## III. Architecture of System

As shown in Fig. 1, our project consists of two subsystems: the PC (a machine with Quartus installed and with an Ethernet port) and the FPGA (a DE2-115 Altera board). The two subsystems communicate through Ethernet. In this section, we will describe how data flows through our system at a high level. In section IV, we will describe the DUT and the Ethernet interfaces in more detail.

The information entry point of our project is the test case generator, which will be a Python script that takes user input and outputs a test case. User input will be obtained by prompting the user with a series of questions on the command line (e.g., “How many instructions would you like to have in this test?”, “Which instructions would you like to test?”), to which the user may either respond with the desired customization or ask the system to randomize. The test case generator will then create a test adhering to the user’s needs and output the test as a text file containing line-separated instructions.

This test file will then be sent to both the golden model and the DUT on the FPGA. They will both execute the instructions in the test and write to separate files register dumps after each cycle. The golden model, which serves as the reference for checking the DUT’s functionality, is a simple C program executed on the PC that parses the input file and computes the needed arithmetic. We have chosen to write the golden model in a fourth-generation language for simplicity, speed, and ease of ensuring correctness. The DUT, on the other hand, is written in SystemVerilog and synthesized onto the FPGA.

Because the DUT is on the FPGA, not the PC, we need a communication protocol to send the test to it and obtain the register dumps back. This communication protocol will be implemented using Gigabit Ethernet. The FPGA board has two Ethernet ports, but we will only need one of these ports to send all of our data (instruction bytes streaming to the FPGA and register dump bytes streaming out). We will elaborate on both the DUT and the communication protocol in the next section.

After the test completes on both the golden model and the DUT, we will have two files on the PC containing register dumps from each execution. Those two files will then be sent to an output comparator, again a Python script. The comparator is responsible for detecting that either the register dumps are the same for every cycle (meaning the DUT passed the test) or, in the case of a discrepancy (meaning the DUT failed the test), the exact cycle at which the discrepancy began.

We will then take the result and display it to the user. The scope of this part of the project is not yet clear due to time constraints. For the simplest implementation, we will just output, to standard out or to a text file, if the test passed or failed, and which cycle the failure occurred if there’s any. If time permits, however, we will develop a GUI that displays additional statistics about the test (e.g., graphs of the instruction distribution in the test), to help the user make better sense of what they had tested. For now, we are not prioritizing this part of the project because it is not a part of the core functionalities of our system.

## IV. System Description

The DUT and the Ethernet interfaces, briefly mentioned in the previous section, are the most complex modules in our project and the modules with the most design choices. In this section, we will describe those modules in greater detail. In section V, we will describe the design choices we made regarding those modules.

### A. DUT

Our DUT implements a single-cycle processor capable of executing some 16-bit ALU instructions inspired mainly by RISC-V. Each instruction is 16 bits, while our register file is also 16 registers by 16 bits. Note that we do not support memory operations or control flow, so each register is general purpose. The instruction format is as follows in Fig. 2.

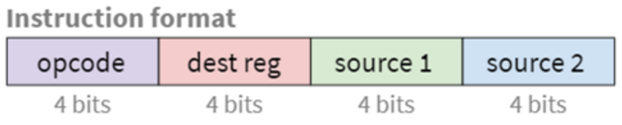


Fig. 2: Instruction format

The instruction bits are evenly divided into four components. At a high level, the instruction takes the two sources, applies the opcode on the sources (performs some computation), then writes the result to the destination. Source 1 must be a register, while source 2 may be a register (for a register-register instruction), a direct value (for a register-immediate instruction), or not used (for MOV). The destination can only be a register. We'd like to note two caveats here. The first is that register 0 is always 0 (a value written to it will simply be disregarded). The second is that for immediate instructions, the 4-bit immediate in the instruction will be sign-extended to 16 bits before the computation is performed. Both of those features are taken from RISC-V.

Table 1 outlines the full instruction set that we support. Each line in our test cases will be one instruction in hex (e.g., 9325 means r3 = r2 + 0x5).

Table 1: Supported instruction set

| Instr | Opcode | Type | Description |
|---|---|---|---|
| MOV | 0x0 | Reg-Mov | Register move |
| ADD | 0x1 | Reg-Reg | Register add |
| SUB | 0x2 | Reg-Reg | Register subtract |
| AND | 0x3 | Reg-Reg | Register bitwise and |
| OR | 0x4 | Reg-Reg | Register bitwise or |
| XOR | 0x5 | Reg-Reg | Register bitwise xor |
| SLL | 0x6 | Reg-Reg | Reg. shift left logical |
| SRL | 0x7 | Reg-Reg | Reg. shift right logical |
| SRA | 0x8 | Reg-Reg | Reg. shift right arith. |
| ADDI | 0x9 | Reg-Imm | Imm. add |
| ANDI | 0xa | Reg-Imm | Imm. bitwise and |
| ORI | 0xb | Reg-Imm | Imm. bitwise or |
| XORI | 0xc | Reg-Imm | Imm. bitwise xor |
| SLLI | 0xd | Reg-Imm | Imm. shift left logical |
| SRLI | 0xe | Reg-Imm | Imm. shift right logical |
| SRAI | 0xf | Reg-Imm | Imm. shift right arith. |

Fig. 3 shows the block diagram for our DUT implementation in SystemVerilog. On each clock, the DUT takes a 16-bit instruction from the Ethernet interface. We have an instr_valid signal to indicate whether the instruction is ready to be executed (has fully propagated through Ethernet). The instruction then goes through a decoder, which outputs the four components of the instruction in Fig. 2, along with some other control signals that will be passed along to the register file and/or ALU (e.g., whether this is a register-immediate instruction).

We then access the operands from the register file. The register file will be asynchronous read, synchronous write, meaning the operand values will be available in the same clock cycle. After obtaining the operands, they are passed along to the ALU. On the next clock cycle, the ALU's output will be written back to the destination register in the register file.

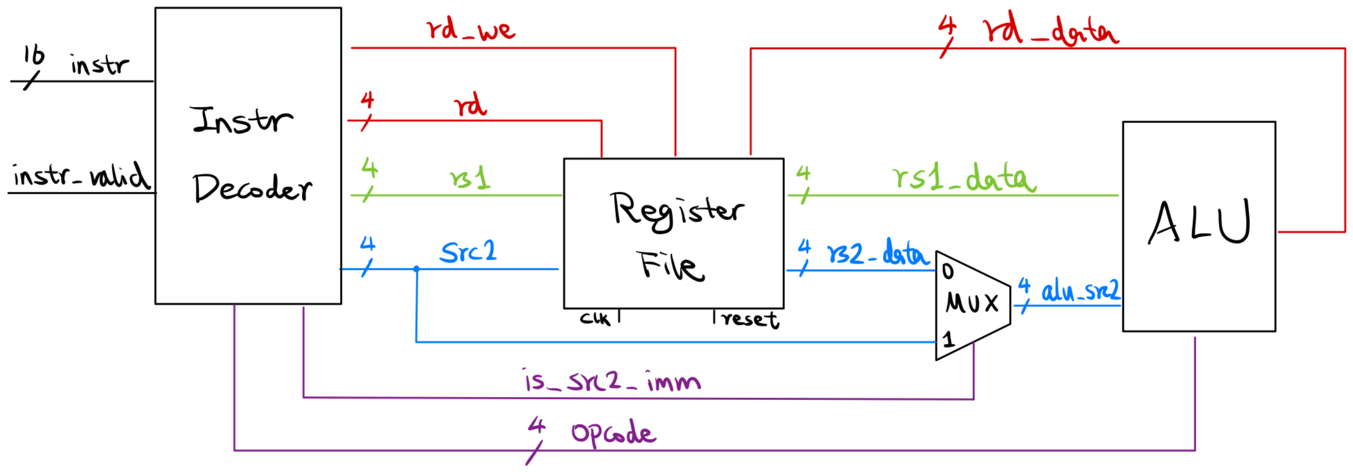


Fig. 3: DUT datapath

### B. *Ethernet interfaces*

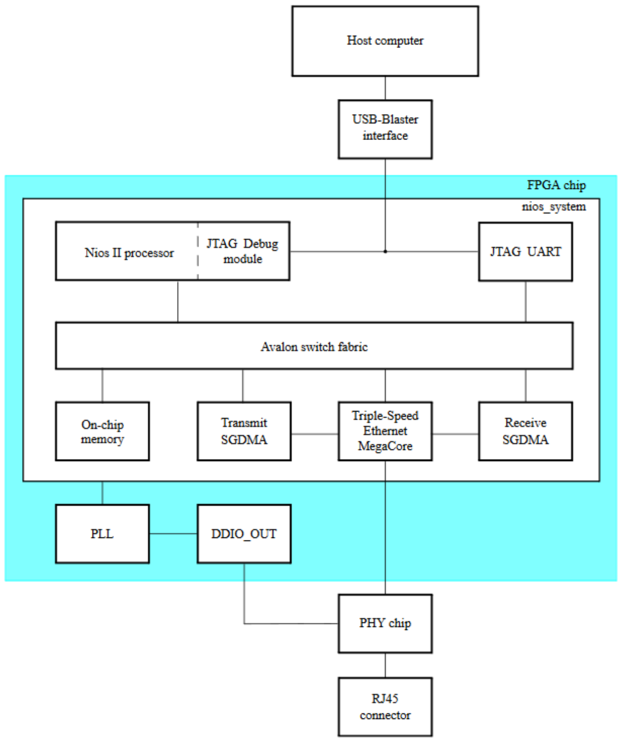
As mentioned in section III, the two subsystems of our project—the PC and the FPGA—will be connected using Gigabit Ethernet, with the PC sending the FPGA instructions to execute, and the FPGA sending the PC register values back.

Most of the communication will be controlled on the low-level by Ethernet directly. However, we will need to handle the end points of communication: that is, choosing where the data is being read from / written to. On the PC, we will simply use files for reading/writing. On the FPGA, we will be using the NIOS II/e to place Ethernet data in memory. The NIOS II/e is an off-the-shelf soft processor developed by Altera, which we can configure and then instantiate on the FPGA board along with the DUT. This processor will allow us to run C/C++ programs on the FPGA board.

Fig. 4 is obtained from Altera's triple speed Ethernet guide [3]. As shown in the figure, the Nios II Processor has a Triple Speed Ethernet MegaCore connected to a transmit and a receive Scatter-Gather Direct Memory Access (SGDMA). This SGDMA is a conversion point between memory and streaming interfaces: it converts serial data to words of data and interfaces with the main memory. We can therefore use main memory to save the data that is being received from the PC and the data that will be sent to the PC. To simplify the protocol, we will keep the memory location the same for all reads and the same for all writes.

We will be implementing a handshaking protocol for alerting when new data is ready to be read or sent back to the PC. The end goal is that at every positive clock edge, the DUT can assume that it is getting a new instruction unless the signal telling it to wait is asserted. As we develop the protocol, we anticipate having to add more signals to the DUT to indicate different events.

Fig. 4: General Nios II Processor diagram



## V. Design Trade Studies

Throughout the design phase of our project, we made design decisions to balance implementation complexity, meeting our requirements, and the usefulness of our system. Those design decisions can be divided into three categories: regarding the hardware platform for our project, regarding the DUT, and regarding the communication protocol.

### A. *Hardware platform*

With our goal being accelerating simulation by leveraging the fast clocks of hardware, we first needed to choose what hardware platform to use.

#### 1) *FPGA vs. ASIC*

FPGAs and ASICs are both commonly used as hardware accelerators. We chose to use a FPGA for the reconfigurability and low cost. Verification, fundamentally, is the process of discovering bugs and fixing the design to be bug free. Our platform therefore needs to accommodate changes in the design, making an ASIC—a static design—not suitable for our use case. In addition, the cost of manufacturing an ASIC is well beyond our budget, while FPGAs are much cheaper in comparison, and the ECE department has some readily available.

#### 2) *Altera vs. Xilinx*

We could use either an Altera or a Xilinx FPGA owned by the ECE department. While the Xilinx FPGAs available have a built-in SoC on the board (which makes communicating data to and back from the FPGA chip simple), none of us have experience programming such a FPGA using the necessary toolchains. After several people who do have experience with Xilinx FPGAs informed us that learning how to use said toolchains in the time frame we have is incredibly difficult, we decided to use an Altera FPGA instead. However, the lack of an on-board SoC means we need to generate the test cases on a PC and send data between the PC and the FPGA through a communication protocol we develop.

#### 3) *DE2-115 vs. Cyclone V*

As for which Altera board to use specifically, we again have two options: the DE2-115 board and the Cyclone V board. The Cyclone V is overall more powerful (it has more memory and more logic elements), while the DE2-115 has more I/O ports. Keeping in mind the need for a communication protocol, we chose the DE2-115 because more I/O ports give us more options (and fallbacks) for how to send/receive data. The downside of having limited memory and logic elements then influenced the design of our DUT.

### B. *DUT*

We chose to implement a processor for our DUT because the project arose from our shared struggles in trying to verify a CPU core. However, the limited amounts of memory and logic elements on the DE2-115 FPGA dictate that we cannot create a very complex DUT. The challenge here is then to limit the scope of the DUT while keeping it realistic and interesting.

First, we ruled out making a pipelined processor our DUT out of concerns for complexity and the design not fitting on the board. A non-pipelined design is expected to fit because we have experience synthesizing, in 18-240, such a design onto the same board. Next, we need to determine what instructions our DUT could support.

*1) Instruction size*

Modern processors typically have 32-bit or 64-bit ISAs. Having such large bit widths though would mean having high communication cost in our system (since for each instruction, we need to send the instruction to the FPGA and obtain register information back). Moreover, having a large register file raises the chances of the design not fitting on the board. In the end, we settled for 16 bits to lower those risks, while keeping the ISA large enough that it does not become a toy example.

*2) Instruction format*

To keep the instruction decoder simple, we chose the simplest scheme possible for the instruction format, illustrated in Fig. 2.

*3) Instruction set*

A typical ISA consists of compute, memory, and control flow operations. However, in the interest of making our communication protocol simple and again due to FPGA size limitations, it is not feasible for us to support changes in control flow or maintain a large memory. Our instruction set is thus comprised solely of compute instructions.

The instructions themselves are chosen after studying three ISAs: RISC-V, ARM, and x86. While our instruction set is mainly inspired by RISC-V, we wanted to look at others in order to determine what instructions are important and essential. We ended up choosing the most common instructions across these ISAs.

## C. Communication

Although the FPGA has a fast clock, communication to off-board is time-consuming and is the bottleneck of our system. To meet 3x speedup despite this challenge, we designed various ways to reduce/hide the communication latency and calculated which channel would allow us to meet our requirement.

*1) Reducing latency*

We can reduce latency by reducing the number of bits sent between the PC and FPGA. For every instruction processed, the DUT needs to receive the instruction and the PC needs to receive the register dump (in order to perform cycle-by-cycle correctness checks). In a naïve approach, this would mean a total of 272 bits (16 instruction bits + 16 registers * 16 bits) of communication per instruction processed.

However, we observed that it is not necessary to send the entire register dump after each instruction; we can instead send the delta (which register changed to what value) and reconstruct the dump at the PC end. This reduces the number of communication bits to 36 bits per instruction processed (16 instruction bits + 4 bits for register index + 16 bits).

*2) Hiding latency*

In addition to reducing latency, we can also hide latency through pipelining. In contrast to the serial approach of waiting for one instruction to finish propagating through our system (to the FPGA, then computed, then back from the FPGA) before processing the next instruction, we can overlap processing of multiple instructions.

Table 2: Communication pipeline illustration

| | | | | | |
|---|---|---|---|---|---|
| TO | COMP | BACK | | | |
| | TO | COMP | BACK | | |
| | | TO | COMP | BACK | |
| | | | TO | COMP | BACK |

Table 2 illustrates this pipeline. Each row represents a different instruction, and the pipeline consists of three stages: TO is sending data from the PC to the FPGA, COMP is the computation on the FPGA, and BACK is sending data from the FPGA back to the PC.

In a pipelined system, throughout is bounded by the latency of the slowest stage, which is BACK in our case. Whereas in the serial approach, we have one instruction processed after the latency of TO + COMP + BACK, in our pipelined system, we have one instruction processed after only the latency of BACK.

We recognize that while the pipelined approach offers speedup gains, it will also complicate the communication protocol significantly. Therefore, we decided we will start with the serial approach, then implement the pipelined approach if time permits. Calculations from this point onwards will assume we are communicating a reduced number of bits per instruction (36 bits) as explained in C1), but without pipelining.

*3) Communication channel*

After exploring ways to reduce the communication cost, we needed to determine which communication channel to use to meet our speedup requirement. The Altera DE2-115 board has three types of I/O that we could use: JTAG, USB, and triple speed Ethernet. Table 3 gives the speed for each of these options (found in the DE2-115 User Manual [4]).

With the values in Table 3, we can calculate, for each channel, how much time it takes to transmit 36 bits:

$$communication\ time\ per\ instr = \frac{36\ bits}{\#\ bits/s} \quad (1)$$

We also know the FPGA clock is 50 MHz (again from the User Manual). We can then obtain the total time per instruction by adding communication time and FPGA compute time. Because of our DUT's single-cycle microarchitecture, each instruction takes one cycle on the FPGA:

$$total\ time\ per\ instr = comm.time + \frac{1}{50\ MHz} \quad (2)$$

Table 4 gives the results of these calculations.

The next step was to find how much time it would take to simulate one instruction so we could choose a communication

Table 3: DE2-115 communication channels and speeds

| Communication channel | Data transmission speed (# bits/s) |
|---|---|
| JTAG | 4 Mbit/s |
| USB | 12 Mbit/s |
| Triple speed Ethernet | 10, 100, or 1000 Mbit/s |

Table 4: Time per instruction for each channel

| Communication channel | Comm. time per instr. | Total time per instr. |
|---|---|---|
| JTAG | 9 µs | 9.02 µs |
| USB | 3 µs | 3.02 µs |
| Triple speed Ethernet | 3.6 µs, 360 ns, or 36 ns | 3.62 µs, 380 ns, or 56 ns |

protocol that meets speedup. Unfortunately, this metric is difficult to determine. Simulation runtime varies significantly across different designs, making it impossible for us to have an exact number of how long our design would take before we have the design ready. However, we do know that simulation time grows as design complexity grows, so we can obtain a lower bound time by simulating a smaller design.

We ended up creating an adder and simulating its execution. Our benchmarks show that it takes roughly 2 μs to simulate a single add operation. This means in order to attain 3x speedup, each instruction in our approach must be processed in less than 2 μs/3 ≈ 666 ns. We can achieve this using either 100 Mbit/s or 1000 Mbit/s Ethernet. We are planning on using 1000 Mbit/s Ethernet (also called Gigabit Ethernet) to give us the most speedup we can get and to leave room for spending bits not taken into account in this calculation for handshaking.

We'd like to note, once again, that the 2 μs is a lower bound approximation. However, this estimate is still useful for providing us order-of-magnitude insights.

#### *4) Nios II Processor*

After deciding to use Ethernet as our communication channel, we found an Altera tutorial for how to use Ethernet on DE2-115 boards [5]. Implementing the system following the tutorial requires using a Nios II processor. This presents another design decision, as there are 3 versions of the Nios II processor: Nios II/e, Nios II/s, and Nios II/f. The versions increase in functionality and area in the order listed, but all of them have the needed functionality for us to use Ethernet.

We decided to use the Nios II/e (the simplest one) in the interest of saving area on the FPGA. Another benefit of using the Nios II/e is that it is the only one out of the 3 versions that does not require a license to use. This means instead of having to go on CMU lab computers (which have the license) to work, we can use our personal computers.

## VI. Test and Validation

Recall that our requirements are divided into three categories: performance, functionality, and ease of use. In this section, we outline testing plans for each requirement.

### *A. Speedup*

Our 3x speedup requirement can be tested by measuring simulation runtime and runtime of our approach. For simulation runtime, we will use CPU time outputted by VCS at the end of simulation. To get the runtime of our approach, we will utilize software system clocks. Because the dataflow of our system begins on the PC and ends on the PC, we can place a time() call before the test begins and another time() call after all the output has been received. The different between the two calls is then the runtime for our solution.

Speedup is defined as follows:

$$Speedup = Simulation\ time\ /\ Solution\ time \qquad (3)$$

Again, having speedup > 3 means we have met this requirement.

### *B. Functionality*

We have a functional system if we can support test cases ranging from one to 20,000 instructions. We will start by testing that we can send test cases containing one instruction for all 16 instruction types we have through our system.

We are successful if our approach gives the same output as simulation (register dump at the end + whether the test passed). We will then increase the size of test cases and apply the same validation method until we reach 20,000 instructions.

### *C. Ease of use*

As a reminder, ease of use deals with allowing the users to customize and randomize test cases, in addition to providing cycle-by-cycle correctness checks.

Checking that test case customization and randomization are successful is a matter of manual inspection, plus some analysis of randomization distribution. As an example, if the user chooses to only test ADD, we will manually inspect the generated test case to check that it does indeed only contain ADD instructions. If the user chooses to randomize what instructions to test, we will analyze the distribution of instructions in the generated file to see if they are about even. We will repeat this process for different user customizations.

On the other hand, to ensure that when a test case fails, we accurately tell the user which cycle the test fails on, we will be inserting intentional bugs into the DUT. Because we know what the bug is, we can check that our system indicates a failure at the cycle that we expect. The bugs we insert will either be realistic hardware bugs or RTL designer mistakes. Some examples are having a register that is not r0 stuck at 0, or not performing sign-extension on an immediate value.

### *D. Other testing of subsystems*

In addition to validating that we meet our requirements, we will be testing our subsystems as their implementations complete throughout the project.

Not much testing effort is expected to be spent on our software components, as they are relatively simple. We will apply the standard software engineering technique of feeding modules different inputs and observing if the outputs are as expected. Testing the output comparator, for example, involves giving the comparator different input file pairs and observing if it flags differences at the right line.

Our communication protocol, on the contrary, is much more substantial and requires more planned-out testing. We are staging our testing by beginning with simply sending data to the FPGA board and echoing it back to the PC. To be more precise, the echoed back data will be slightly different to ensure that the read and write buffers on the board are not unintentionally coupled somehow.

Once echoing succeeds, we will move on to sending actual instructions to the DUT and obtaining register values back. As mentioned earlier, we will start with small test cases.

## VII. Project Management

### A. Schedule

We give an overview of our schedule here. Please see Appendix A for the Gantt chart of our project.

Our project is divided into three phases. Phase one focuses on setting up our system. This includes creating a basic echo communication protocol, determining the ISA, and programming the corresponding golden model. In phase two, we move on to implementing the core functionality of our system. This entails finalizing the communication protocol, as well as implementing the DUT and output comparator. By the end of phase two, we will have a system that can send test cases to the FPGA and obtain back results. The testing interface is not fully complete, but this is our MVP. In phase three, we will focus on adding features to fulfill our ease-of-use requirements. This means mainly creating scripts for test case customization/randomization, as well as potentially creating a UI to output results to the user.

We can currently in the middle of phase one and two. We are working on the echo protocol, while development of the DUT is also underway.

### B. Team member responsibilities

We have divided up the work so that each team member is responsible of two large tasks, roughly one for the first half of the project and one for the second half.

#### *1) Ali*

Ali's responsibilities are to develop, along with Grace, the communication protocol and to test different parts of the design. Developing the communication protocol includes both the initial echo protocol as well as the final version to be used in our system. Afterwards, Ali will transition to a testing role. She will test components as they complete and perform periodic benchmarking to see if we are on track to meeting speedup.

#### *2) Grace*

Grace's responsibilities are to develop, along with Ali, the communication protocol and the input part of our interface. She and Ali will work together to research, implement, and test our communication protocol. Afterwards, Grace will develop the test case generator, allowing the user to customize and randomize test cases.

#### *3) Xiran*

Xiran's responsibilities are to implement the golden model and DUT, and to develop the output end of our testing interface. She begins by designing an ISA for our use case and will then create the design in both software and hardware. Afterwards, she will develop the output comparator and an output display to the user.

### C. Bill of materials and tools

Please see Appendix B for a list of what equipment we used and their costs.

In addition to this equipment, we will be using Quartus Prime II for synthesis onto the FPGA.

### D. Risk management

The biggest risk in our project is in data communication between the PC and the FPGA. This is because communication is both challenging and essential: while none of us have experience working on FPGA communication to an offboard component, we must use a fast communication protocol in order to meet our speedup requirement.

We are actively mitigating this risk with three strategies: (1) perform thorough research before implementation, (2) allocate more time and manpower to this part of the project, and (3) come up with backup plans.

As shown in section V, we chose the communication protocol that would maximize our chances of meeting speedup after quantitative analysis. This mitigates the risk of us having spent significant effort on implementation, only to discover that we cannot meet our requirement. After deciding to use Ethernet, we continued to be cautious by looking into implementation details from multiple sources. Some of the tutorials and projects we found helpful are listed under References.

The communication protocol task is also the first task we worked on when the project began and is the only task with two team members responsible for it. Ali and Grace can support each other and balance the workload.

Despite these efforts, in the case that we cannot implement a successful Ethernet protocol, we have investigated backups. The first set of backups is to switch to using either USB or JTAG as our communication channel. We all have experience with the USB protocol, making the implementation difficulty lower than that of Ethernet. JTAG is even simpler: it is already set up on the board and should not require much configuration from our end. Unfortunately, switching to either of these protocols would mean we will most likely not meet our speedup requirement, but that is better than not having a functional product at all in the end.

If we are still not able to get offboard communication working, we have yet another backup: store test cases in and write results to the FPGA's memory. We then have no need for any complex communication protocol because everything is completed on chip. Of course, this would mean we most likely need to reduce the size of our test cases and revamp our system architecture. We therefore save this option as a last resort.

Regardless of these challenges, Grace and Ali are close to running a test with the Nios II Processor and Ethernet. They are mainly working on debugging the synthesis process and are looking forward to making Ethernet work.

## VIII. Related Work

While researching the feasibility of our project in the beginning stages, we were surprised to learn that hardware acceleration of simulation is in fact a hot area of research. A quick Google search of “FPGA accelerated verification” or the like reveals a plethora of recent research papers ([6], [7], [8]). Common trends across these papers include the flexibility of the framework proposed (can move some portions of the DUT onto the FPGA while keeping others in simulation) and much greater speedup (up to two or three orders of magnitude).

Although we cannot hope to achieve as much gains as these works from academia, we are encouraged by their existence (as they show there is clearly a problem to be solved) and excited by the new technology to come.

We also looked into works that used the Nios II processor and/or Ethernet ports to learn more about how to use them. One of the projects we found uses an Altera DE2-115 and a Nios II Processor to handle image and video processing [8]. This project helped Grace and Ali understand how to use the Nios II Processor with C/C++ code.

Another project we found useful is a project that designed an interface for the FPGA to communicate with other devices through Ethernet without using the Nios II processor [9]. It details the Ethernet communication protocol, which is useful to reference when designing our own packet.

## IX. Summary

The main goal of our project is to speed up RTL simulation using a FPGA. However, the FPGA we are using has limited memory and logic elements, thereby limiting what kind of DUT we can place in our system. We encourage future work to be done using more powerful FPGAs, ideally with a SoC onboard, as this would enable testing of more complex designs. In addition, by significantly reducing communication cost through the use of onboard communication, much greater speedup can be attained.

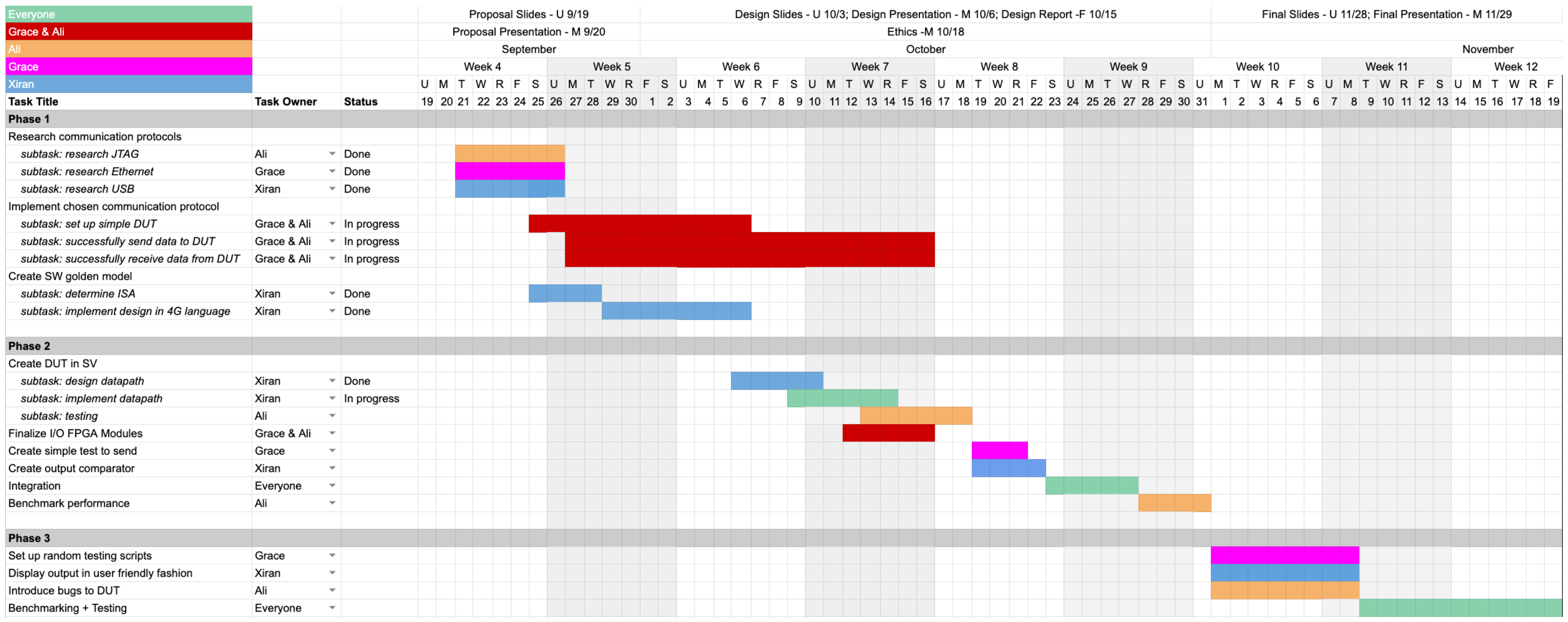
## Glossary of Acronyms

ALU – Arithmetic Logic Unit
DUT – Device Under Test
FPGA – Field Programmable Gate Array
GUI – Graphical User Interface
ISA – Instruction Set Architecture
MVP – Minimum Viable Product
PC – Personal Computer
RTL – Register Transfer Level
SoC – System on a Chip

## References

[1] https://www.crn.com/news/components-peripherals/229200131/intel-assesses-damage-of-cougar-point-chipset-flaw.htm
[2] http://www.iie.uz.zgora.pl/iie_archiwum/desdes01/files/ref/IV-7.pdf
[3] https://bohr.wlu.ca/nznotinas/altera_reference/DE2_115/using_triple_speed_ethernet.pdf
[4] https://www.intel.com/content/dam/www/programmable/us/en/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf
[5] https://bohr.wlu.ca/nznotinas/altera_reference/DE2_115/using_triple_speed_ethernet.pdf
[6] https://escholarship.org/uc/item/0vt3c73p
[7] http://trilobit.fai.utb.cz/Data/Articles/PDF/fba3fd06-6222-4e25-910c-989553226dde.pdf
[8] https://ieeexplore.ieee.org/document/1329526
[9] https://www.secs.oakland.edu/~ganesan/ece576f14project/index.htm
[10] https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2011/mis47_ayg6/mis47_ayg6/

## Appendix A: Gantt chart



| | |
|---|---|
| 🟩 | Everyone |
| 🟥 | Grace & Ali |
| 🟧 | Ali |
| 🟪 | Grace |
| 🟦 | Xiran |

| Task Title | Task Owner | Status |
|---|---|---|
| **Phase 1** | | |
| Research communication protocols | | |
| *subtask: research JTAG* | Ali | Done |
| *subtask: research Ethernet* | Grace | Done |
| *subtask: research USB* | Xiran | Done |
| Implement chosen communication protocol | | |
| *subtask: set up simple DUT* | Grace & Ali | In progress |
| *subtask: successfully send data to DUT* | Grace & Ali | In progress |
| *subtask: successfully receive data from DUT* | Grace & Ali | In progress |
| Create SW golden model | | |
| *subtask: determine ISA* | Xiran | Done |
| *subtask: implement design in 4G language* | Xiran | Done |
| **Phase 2** | | |
| Create DUT in SV | | |
| *subtask: design datapath* | Xiran | Done |
| *subtask: implement datapath* | Xiran | In progress |
| *subtask: testing* | Ali | |
| Finalize I/O FPGA Modules | Grace & Ali | |
| Create simple test to send | Grace | |
| Create output comparator | Xiran | |
| Integration | Everyone | |
| Benchmark performance | Ali | |
| **Phase 3** | | |
| Set up random testing scripts | Grace | |
| Display output in user friendly fashion | Xiran | |
| Introduce bugs to DUT | Ali | |
| Benchmarking + Testing | Everyone | |

## Appendix B: Bill of materials

| Item Name | Quantity | Price | Manufacturer | Model Number | Description |
|---|---|---|---|---|---|
| Altera DE2-115 FPGA | 1 | $675.00 | Altera | DE2-115 | This is an FPGA that the ECE department had in stock |
| Cat 5 Ethernet Cable | 1 | $11.84 | Mediabridge | 31-399-25X | Ethernet cable used to communicate between PC and FPGA |
| Dual ended USB cable | 1 | $6.99 | UGREEN | 10369 | USB Male to USB Male cable to use to test demo project that comes with FPGA |
| USB to ethernet dongle | 1 | $16.55 | Amazon Basics | U3-GE-1P | USB-ethernet converter used to easily connect ethernet cable to PC/Desktop. Because it uses USB 3.0, it should not limit the speed of ethernet. |