FPGA-Assisted Verification

Authors: Ali Hoffmann, Grace Fieni, 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. To test our method, we are creating a simple single-cycle processor as our DUT on the FPGA. The goal is for our method to achieve 3x speedup relative to simulation, and our end system is able to achieve at least 2.2x speedup over the range of test case sizes that we support.**

***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, input end, and output end.

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 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 is where we spent the majority of planning and development effort (see section V for communication tradeoff studies).

B. Input end

Our next set of requirements deals with the input end of our system. First, we will support running test cases of any size ranging from one to 20,000 instructions. Every instruction is a command that will be sent to our own design under test (see section IV for DUT details). Users can use small test sizes to isolate bugs, while large test sizes can be used to stress the design.

Furthermore, 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 functionality in our framework. More specifically, we will allow the user to choose the number of instructions per test, which instructions to test, and which registers to write to, while unspecified parameters are randomized.

C. Output end

Our last set of requirements is about the output end of our system. At the end of our system flow, we will perform an instruction-by-instruction correctness check 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).

We would also like to implement a GUI that displays to the user results from the correctness check in a readable function. This is again to help the verification engineer save time by making information easier to parse. Statistics about the test case will also be displayed here to help the engineer gauge the quality of the test case.

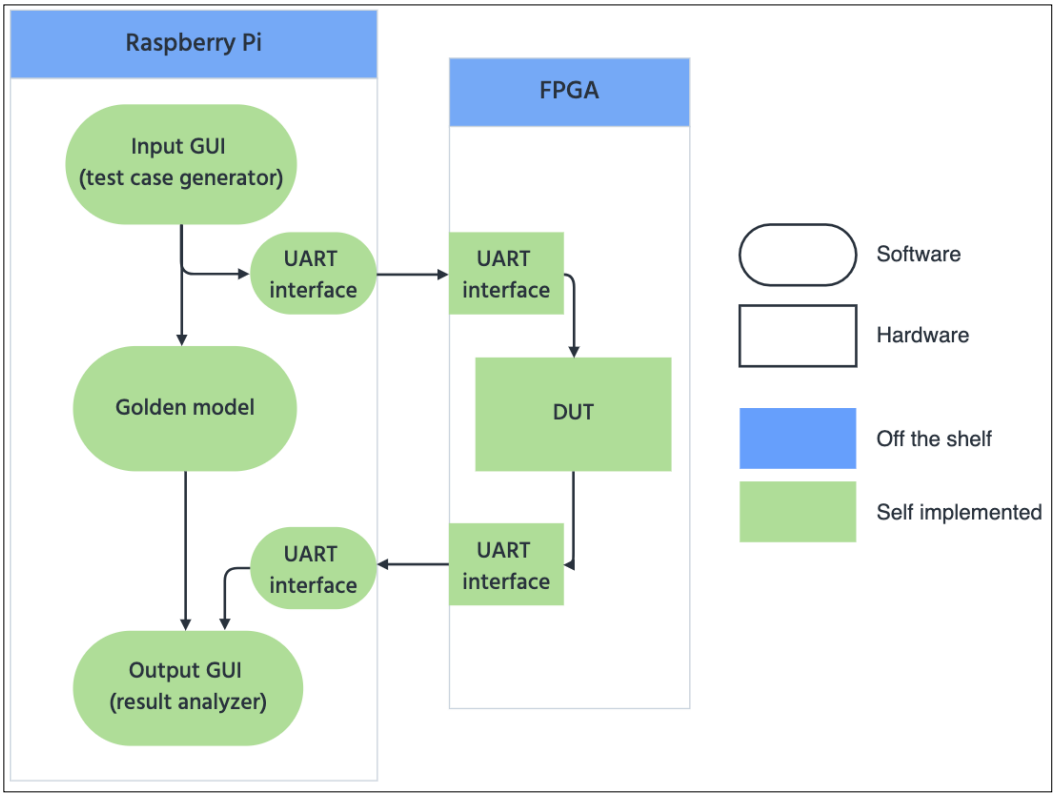


Fig. 1: Our system overview

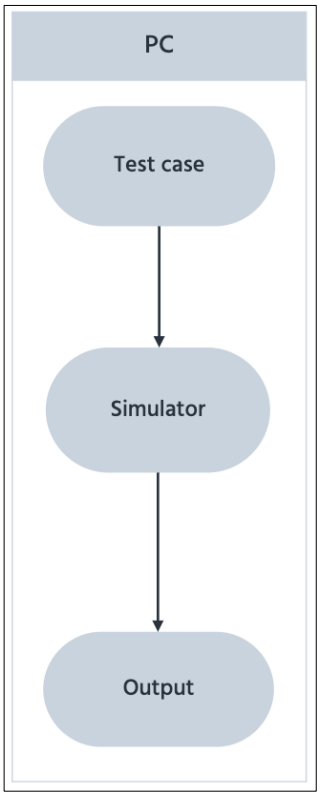


Fig. 2: Traditional system overview

III. Architecture of System

As shown in Fig. 1, our project consists of two subsystems: the Raspberry Pi 4 and the FPGA (DE2-115 Altera board). The two subsystems communicate serially with the UART protocol. In this section, we will describe how data flows through our system at a high level. In section IV, we will describe the DUT, the GUIs, and the UART interfaces in more detail.

The information entry point of our system—the test case generator—is a form that asks the user to answer a series of questions regarding how they would like to customize the test case. It then creates a test adhering to the user's needs and outputs 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 the results after each instruction. The golden model, which serves as the reference for checking the DUT's functionality, is a simple C program executed on the Raspberry Pi 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 Raspberry Pi, we need a communication protocol to send the test to it and obtain the results back. This communication protocol is UART (Universal Asynchronous Receiver-Transmitter). Both the Raspberry Pi and the FPGA have serial ports, and we use a cable to physically connect the two components.

After the test completes on both the golden model and the DUT, we will have two files on the Raspberry Pi containing outputs from each execution. We then upload these two files to the result analyzer. The analyzer is responsible for detecting that either the outputs are the same after every instruction (meaning the DUT passed the test) or, in the case of a discrepancy (meaning the DUT failed the test), the exact instruction at which the discrepancy began.

In a traditional simulation-based system (shown in Fig. 2), the test case is sent to the simulator (and the simulator only), which runs the test case and outputs the result. For our project, the core idea is then to replace the simulation part with a combination of the golden model and the DUT on the FPGA. The speedup of this approach comes from the golden model and the FPGA being able to run faster than the simulator can, although we must incur the overhead of communication.

We'd also like to note two changes that we have made to the overall architecture since the Design Review. The first is that we originally planned to use Ethernet for communication to and from the FPGA, but have since switched to UART due to implementation difficulties. This change was significant, and we explain why the change was made and what it entails with regards to meeting our requirements in sections V and VI, respectively. The second change was that we are now communicating to the FPGA from a Raspberry Pi, not a PC. This is simply because we discovered that a Raspberry Pi is able to handle data transfer at a higher rate than the PC we were using (i.e., it allows us to obtain higher performance).

IV. System Description

The DUT, GUIs, and the UART interfaces, mentioned in the previous section, warrant more explanation because of their complexities. In this section, we will describe those modules in greater detail. Then in section V, we will describe the design choices we made regarding the DUT and the communication protocol.

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.

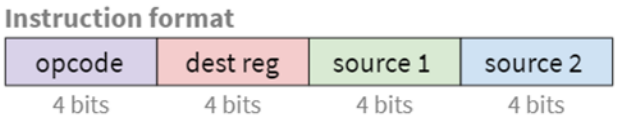


Fig. 3: Instruction format

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 UART interface. We have an instr_valid signal to indicate whether the instruction is ready to be executed (has fully propagated through UART). 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.

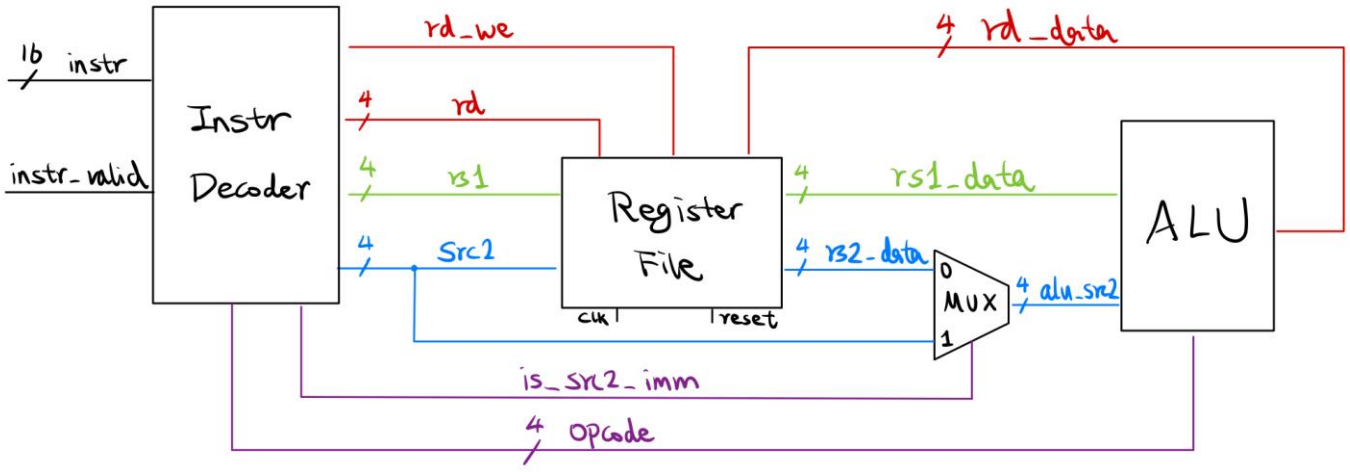


Fig. 4: DUT datapath

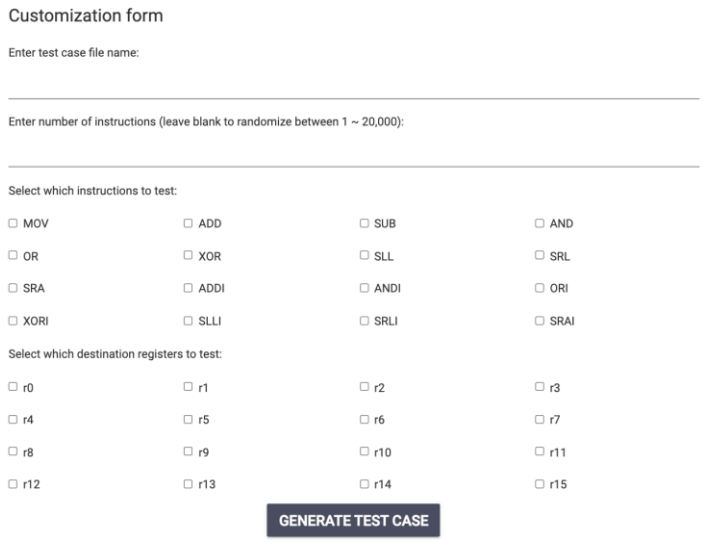


Fig. 5: Input GUI

B. *GUIs*

The information entry and exit GUIs in our system (the test case generator and the result analyzer) are implemented as a web app. To obtain a test case, the user fills out the above form in the web app and clicks on the "Generate test case" button. The form allows the user to customize how many instructions to include in the test case, as well as what types of instructions and destination registers to test for. The size customization is to allow the user to choose whether they are performing small or stress testing. The type customization is for testing individual features.

As for the output end, after the golden model's output and the FPGA's output are uploaded to the result analyzer, it will detect errors and present the earliest discrepancy as shown in Fig. 6. We see a progress bar of how far the DUT made before failing and a register value comparison table showing the mismatches at the failed instruction.

One more important note regarding the GUIs is that the web app is made using a web app builder called Anvil. We chose to use a framework instead of building the app from scratch due to ease of implementation. However, relying on a third-party host does come with security concerns, which we explore in section VIII.

Failure at instruction #8: 6943

Register	Correct Value	Actual Value
r0	0000	0000
r1	fffe	fffe
r2	0001	0001
r3	fffe	fffe
r4	ffff	ffff
r5	fffd	fffd
r6	0000	0000
r7	ffff	ffff
r8	0000	c000
r9	c000	0000
r10	0000	0000
r11	0000	0000
r12	0000	0000
r13	0000	0000
r14	0000	0000
r15	0000	0000

Fig. 6: Example output analysis

C. *UART interfaces*

Recall from Fig. 1 that our system requires a method of sending input and output data to and from the FPGA board. Our final system uses the UART (Universal Asynchronous Receiver-Transmitter) protocol. We made this choice mainly because the protocol is simple to implement and has ample documentation. This is in contrast to another protocol that we were experimenting with—Ethernet—which we ultimately abandoned due to lack of documentation (we touch more on this in section V).

UART sends data at the granularity of packets. Every packet contains a start bit at the beginning, followed by data bits, followed by a stop bit at the end. For our implementation of the protocol, the start bit is an active low bit and the stop bit is an active high bit. We send 8 data bits per packet, for a total of 10 bits per packet (Fig. 7). Although the start and stop bits are overhead, they allow us to send packets without having to obey fixed idle periods (the packets can start whenever). This is important because setting up precise timing is quite tricky.

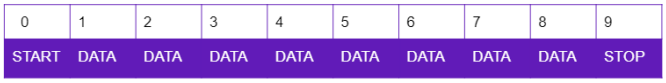


Fig. 7: UART packet

In the first part of our communication, we send data from the Raspberry Pi to the FPGA board. More specifically, we want to send instructions from the test case generator to the DUT, one instruction at a time. Again, in our system, each instruction is 16 bits. After the DUT processes the instruction, the Raspberry Pi needs to receive the output back. This output consists of which destination register was written to (4 bits) and the value in that register after the instruction completes (16 bits) for 20 data bits total. Note that we pipeline the transmit and receive directions (i.e., transmit and receive happen at the same time for different instructions).

Unfortunately, 20 data bits cannot be broken up into whole packets. Our solution is to add 4 bits of padding to the output, so every output ends up being 3 packets, while every input is 2 packets. However, we still have a problem because with the input and output directions being different lengths, our pipeline stages would have misaligned timing. We therefore pad another packet to the input direction to make it 3 packets as well. We can reduce the need for padding by merging data. Although we did not have time for this optimization, we briefly mention its performance effects in section VI.

Performance is tied to how much overhead our protocol incurs and also to the rate at which we can send bits. This rate, known as the baud rate, is configurable depending on what the equipments can support, and it is 576,000 bits per second in our system. The limiting factor here is the cable speed, and we touch more on the performance effects again in section VI.

V. Design Trade Studies

Throughout our project, we made 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 to and from 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 support more I/O types. Keeping in mind the need for a communication protocol, we chose the DE2-115 because more I/O types 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 (pre–Design Review)

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 to and from and FPGA. For every instruction processed, the DUT needs to receive the instruction and send the register dump (in order to perform instruction-by-instruction 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 end. This reduces the number of communication bits to 36 bits per instruction processed (16 instruction bits + 4 bits for register index + 16 bits for register value), not taking into account any overhead.

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 for 4 instructions

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 to the FPGA, COMP is the computation on the FPGA, and BACK is sending data from the FPGA back.

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 in the immediate following section 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 [3]).

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 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.

Table 4: 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 3: 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

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.

D. Communication (post Design Review)

However, as mentioned previously, despite our original plan to use Ethernet as our communication protocol, we ultimately switched to using UART instead. While the FPGA's user manual has instructions for setting up Ethernet communication with the board, those instructions require us to use the NIOS II soft processor: a proprietary processor developed by Altera that we can instantiate on the FPGA. Unfortunately, this processor has both extremely limited documentation and compatibility issues. Given that it is proprietary, we also have no way to figure out its inner workings on our own.

After spending the majority of the semester attempting to use the NIOS II processor to limited success, we decided to forgo relying on proprietary components to instead implement our own end-to-end protocol. UART came to be the choice here given its simplicity. Note that using either the USB or JTAG protocols would have likely put us in the same situation as using Ethernet, given that the user manual says to use special components for those protocols as well.

After we finished implementing the UART interfaces and began integrating, we discovered that the PC we were using could not handle baud rates of above 512,000 bits/second without losing data. We then made the second design change to our system since the Design Review: communicate to the FPGA from a Raspberry Pi instead, since the Raspberry Pi supports baud rates of up to 4,000,000 bits/second. Once we made this change, however, the bottleneck became the physical cable connecting the Raspberry Pi and the FPGA, as the cable begins losing data for baud rates higher than 576,000 bits/second. We did not have time to obtain a better cable, so this is the data rate used in our final system.

In the next section, we will further quantify the performance of our final system.

Fig. 8: Simulation time and solution time comparison

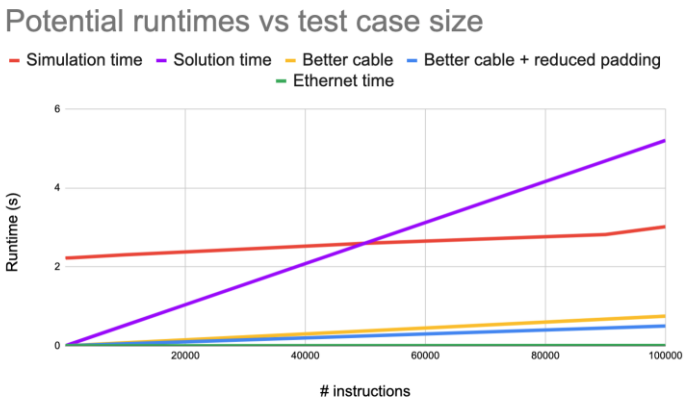


Fig. 9: Potential runtime improvements

VI. Test and Validation

Recall that our requirements are divided into three categories: performance, input end, and output end. While we were not able to meet our performance requirement, we do meet all other requirements. We also briefly talk about some ways to improve performance if we had more time.

A. Performance

Our 3x performance speedup requirement is tested by measuring simulation runtime and runtime of our solution. Speedup is defined as:

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

For simulation runtime, we used elapsed time outputted by VCS at the end of simulation. For the runtime of our solution, we both calculated the expected runtime and confirmed the number with time() calls at the beginning and end of our scripts. Since we implemented pipelining (recall Table 2), the runtime is:

$$Solution\ time = TO + COMP + (\#\ instrs) * BACK \quad (3)$$

COMP is simply one FPGA cycle. TO and BACK are both UART transfers of 30 bits at a rate of 576,000 bits/second (aka TO = BACK = 30/576,000 s).

Fig. 7 shows the comparison between simulation time and solution time for the range of test case sizes we support. For the smallest size (1 instruction), we have speedup of about 20,000x. However, since simulation scales better than our current communication method, speedup diminishes as test case size grows, landing at about 2.3x at 20,000 instructions. We also eventually hit a crossover point at 50,000 instructions, where we have no speedup (see Fig. 8).

Given that we are not able to meet our speedup requirement (and we recognize that scaling is a significant issue), in the next section, we propose ways to improve performance that we did not have time to carry out.

B. Performance improvement proposals

1) Use cable that supports higher data rate

As mentioned in section V, the Raspberry Pi supports baud rates of up to 4,000,000 bits/second, but our cable experiences data loss at baud rates higher than 576,000 bits/second. Using a cable that supports a higher data rate is then one way to improve performance.

Runtime of this improved system with a better cable, assuming we are able to reach the 4,000,000 bits/second upper bound, is plotted in Fig. 8. Note that while our current system has a crossover point due to poor scaling, the improved system scales at an about even rate as simulation.

2) Reduce padding

In addition to improving the rate of data transfer, we can also reduce the number of bits we send to further reduce runtime. As mentioned in section IV, our UART interface requires us to send data at the granularity of whole packets. When we do not have enough data to fill a packet, we must add padding. However, we can merge data from different instructions to form whole packets, thereby reducing the need for padding. This added complexity of merging and unmerging at the other end will bring us a relatively small but still noticeable performance gain, as shown in Fig. 9.

Lastly, we also plot the theoretical performance of using Ethernet in our system. Ethernet provides extraordinary scaling due to its fast speed.

C. Input end

We now move to testing the input end of our system—the test case generator. Testing this component involves ensuring that the customization or randomization request the user makes through the web app form, we generate a test that adheres to those requests. More specifically, the customizations must not be violated (the test case size, if given, must match, and any unspecified instruction or destination register cannot show up in the test case). Furthermore, if the test case is large (greater than 500 instructions), any randomized parameter must have an about uniform distribution.

We emphasize the uniform distribution because randomization failures (always returns the same result) are a major cause of bugs; some parts of the design may in fact never be tested, while the verification engineer thinks they are tested.

Testing that the aforementioned conditions hold is relatively straightforward. We simply give the test case generator different requests (e.g., 10 instructions, only ADD and ADDI, only r1), and observe (both manually and by analyzing the test case's distribution) whether the outputted test case fulfills the conditions. We have tested with 20 such requests, and our test case generator passed all of them.

Table 5: Intentional bugs inserted into FPGA

Bugs
Register-immediate instructions do not sign-extend immediate value.
Register-immediate instructions decode to register-register instructions.
SRA and SRAI perform logical shifts instead.
First instruction always ignored (no effect).
Registers reset at 10000 cycles.
Writes to register 0 not ignored.
Upper 2 bits of register 10 stuck at 0xf.
Writes to registers 8 and 9 are flipped.

D. Output end

To test our last requirement (that is, instruction-by-instruction correctness check), we created 8 buggy versions of our DUT and ran two crafted test cases through each version. One of the test cases exposes the bug, while the other one does not. For the test case that exposes the bug, the result analyzer must report the failure at exactly the location we expect (given that we know both the bug and the test case, we know where the failure should be too). For the test case that does not expose the bug, the result analyzer must report no failures.

The 8 bugs we inserted into our DUT are outlined in Table 5. These are all bugs that we either have personally encountered while implementing processors (aka RTL designer bugs) or have heard of similar bugs existing in chips (aka hardware faults). For all 8 scenarios, our result analyzer behaved exactly as expected. Fig. 6 shown earlier is in fact the output GUI's display when we fail due to bug number 8: writes to registers 8 and 9 are flipped.

VII. Project Management

A. Schedule

Our schedule has changed significantly since the Design Review. The cause was that setting up communication to and from the FPGA took us much longer than expected. While we originally planned about four weeks to implement the communication protocol, this effort has since been stretched out to last the entire semester. Fortunately, because 1) we were able to redistribute tasks to make time for communication and 2) we built lots of slack into our original schedule (we originally planned to finish implementing all subsystems by Thanksgiving), we still managed to finish all components of the project on time.

Please see Appendix A for the final Gantt chart.

B. Team member responsibilities

For the first two thirds of the semester, Ali and Grace worked together towards building an Ethernet protocol. Ali spent more time debugging issues with Ethernet-related demos, while Grace spent more time researching mitigation protocols. In the meantime, Xiran worked on implementing the golden model and the DUT.

After the team decided to switch to using UART, Ali and Grace implemented the UART protocol together. Meanwhile, Xiran implemented the user GUIs. The team then integrated together and completed the final deliverables.

C. Bill of materials and tools

Please see Appendix B for a list of what equipment we purchased and their costs. Note that some of the equipment are not used in our final system.

D. Risk management

The biggest risk in our project was in data communication to and back from 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 have a working communication protocol for our project to function. Ideally, we would also like the protocol to be fast enough to meet our performance requirement, so there are many factors to consider.

In the beginning of the semester, we actively mitigated this risk by performing tradeoff studies on various protocols and allocating more manpower to this part of the project. We chose Ethernet only after studying its performance relative to various other protocols and after having found documentation for how to set it up. 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. The goal is for Ali and Grace to support each other and balance the workload.

However, despite Ethernet being a great fit for our project performance-wise, we continued to run into difficulties once the implementation effort began; the documentation we found turned out to be quite lacking. At this point, we began redistributing tasks to make more time for this riskiest part of the project and also came up with backup plans. The backup plans, which we formed after consulting both online sources and professors, were to either switch to a simpler communication protocol or to forgo communication completely by storing testing cases and results in the FPGA's memory.

When issues with using Ethernet remained unresolved as we head into the last third of the semester, we made the decision to switch to using UART as our communication protocol instead. This turned out to be the right call; we had an incredibly fast turnaround and were able to implement the UART protocol in three weeks. Once we had a completed UART protocol, the risk of not having a functional project was eliminated.

In hindsight, switching to UART earlier would have allowed us to spend more time improving performance. But overall, we are very proud of the progress we've made despite these setbacks.

VIII. Ethical Issues

We observe two major ethical issues that could arise from our project: usage by malicious parties and data leakage.

A. Usage by malicious parties

While the target audience for our project are verification engineers developing consumer electronics, our framework could theoretically extend to verify any silicon hardware. This, of course, includes malicious hardware designed with the intent to attack other systems or even people. To put it plainly, being able to verify good hardware fast means being able to verify bad

hardware fast too.

Although we recognize the potential for misuse, it is hard to envision how such misuse may be prevented or regulated. Denying access to users detected to be verifying malicious systems is one way, but our system is also reproducible by anyone with the same equipment.

B. *Data leakage*

A somewhat more concrete problem is data leakage. Suppose we are offering our framework as a service to companies. Because our framework requires users to upload test data, other users (and system administrators) can potentially gain access to or even tamper with this data. Such vulnerabilities can result in many undesirable outcomes. With only access to data, users can gauge their competitors' verification progress. Reverse engineering the DUT may even be possible depending on the complexity of the design. On the other hand, the ability to tamper with data opens up the possibility of completely sabotaging another's verification work.

The problem is exasperated by the fact that our web app, the component in our system that does data analysis, is hosted on a third-party web app builder that may not have any security protections in place. It is thus important for our system to be vetted by security experts before it is deployed to ensure data privacy for our clients.

IX. 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 ([4], [5], [6]). 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.

As regards to the communication side of things, we found the Serial Lab from CMU ECE course 18-240 created by Professor Bill Nace very helpful. This lab had students use UART to send data between two different FPGA boards. We worked through the lab to get a better understanding of the protocol flow, before we implemented our own version.

X. Summary

Our project aims to make the verification of RTL designs more efficient by speeding up test case runtime and providing features to ease debugging effort. While we came short of our performance requirement, our project meets all other requirements. In section VI, we explored several ways to improve the performance of our final system.

To future students of capstone, we'd like to stress the importance of keeping a flexible mindset when it comes to design choices. No matter how good an idea looks on paper, it may not work when you try to implement it in real life! To that end, it is absolutely necessary to have backup plans. Upon recognizing that an idea is infeasible, being willing to switch to another idea is also important.

Glossary of Acronyms

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

References

[1] Yeraswork, Z., 2021. *Intel Assesses Damage Of Cougar Point Chipset Flaw*. [online] CRN. Available at: https://www.crn.com/news/components-peripherals/229200131/intel-assesses-damage-of-cougar-point-chipset-flaw.htm.

[2] WIŚNIEWSKI, R., BUKOWIEC, A. and WĘGRZYN, M., 2001. *Benefits Of Hardware Accelerated Simulation*. [online] Available at: http://www.iie.uz.zgora.pl/iie_archiwum/desdes01/files/ref/IV-7.pdf.

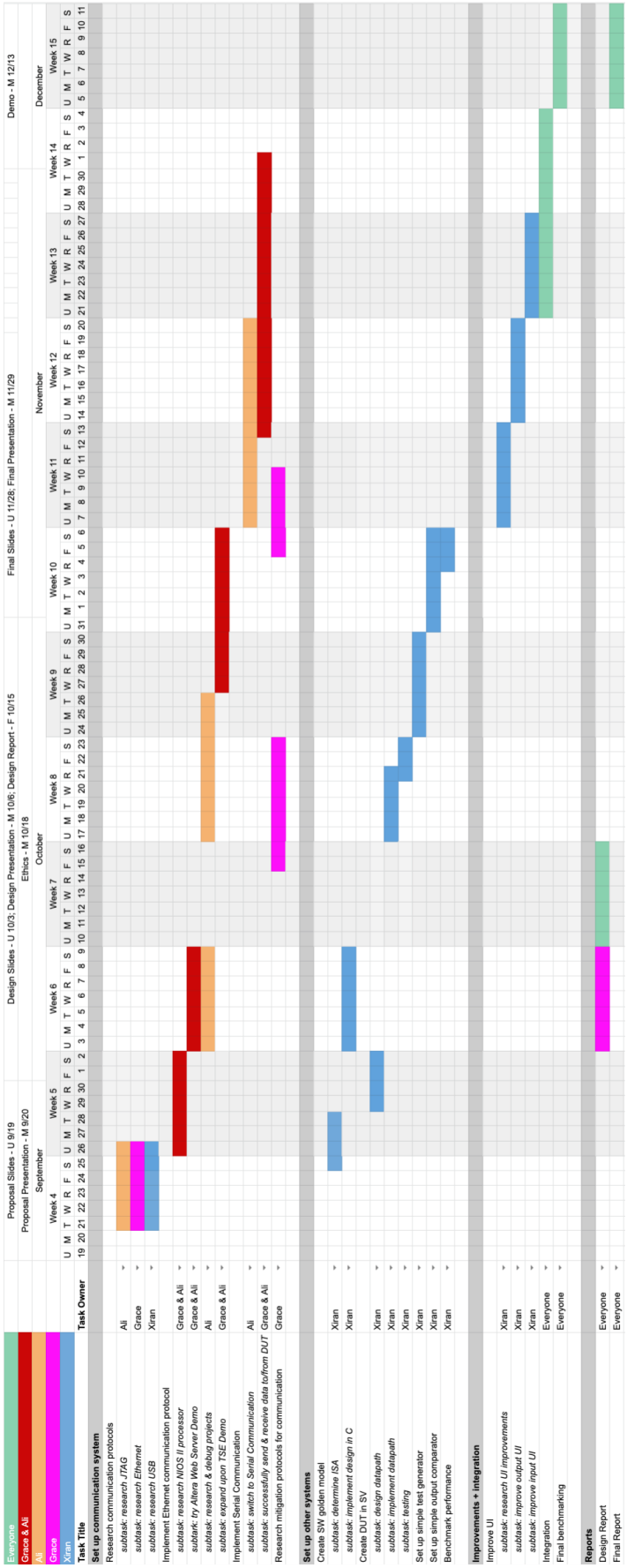
[3] 2013. *DE2-115 User Manual*. [online] Available at: https://www.intel.com/content/dam/www/programmable/us/en/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf.

[4] Kim, D., 2019. *FPGA-Accelerated Evaluation and Verification of RTL Designs*. [online] Escholarship.org. Available at: https://escholarship.org/uc/item/0vt3c73p.

[5] Simkova, M., n.d. *Acceleration of Functional Verification in the Development Cycle of Hardware Systems*. [online] Trilobit.fai.utb.cz. Available at: http://trilobit.fai.utb.cz/Data/Articles/PDF/fba3fd06-6222-4e25-910c-989553226dde.pdf.

[6] Wageeh, Wahba, Salem and Sheirah, 2004. *FPGA based accelerator for functional simulation*. [online] Ieeexplore.ieee.org. Available at: https://ieeexplore.ieee.org/document/1329526.

Appendix A: Gantt chart



Appendix B: Bill of materials

Item Name	Quantity	Used in final system	Price	Manufacturer	Model Number	Description
Altera DE2-115 FPGA	1	Yes	$675.00	Altera	DE2-115	This is an FPGA that the ECE department had in stock.
Cat 5 Ethernet Cable	1	No	$11.84	Mediabridge	31-399-25X	Ethernet cable used to communicate between PC and FPGA.
Dual ended USB cable	1	No	$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	No	$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.
Raspberry Pi 4 2GB Canakit	1	Yes	$101.05	Raspberry Pi & Canakit	PI4-2GB-STR16-C4-CLR-RT	Raspberry Pi 4 2GB used to access web application, and send serial data to the FPGA and process received data.
Serial to Serial Cable	1	No	$9.28	DTECH	B07B4T699J	Serial to Serial cable that we were going to use to transmit data between 2 FPGAs for demonstration purposes (we didn't even end up doing this).
USB to Male Serial Cable	1	Yes	$9.99	CableCreation	B0758B6MK6	USB to Male Serial cable was used to send/receive data to/from the FPGA.
SanDisk Ultra 64GB microSD card with Adapter	1	Yes	$13.60	SanDisk	SDSQUNC-064G-GN6MA	MicroSD card used to store Raspberry Pi OS and data.