Thesis Report for Partial Fulfilment of Major Project

On

Developing a Hardware Verification Framework for the Resource Manager of a Multicore AI Accelerator

Submitted by

191178EE121. Hariharan Ayappane

Under the Guidance of

Dr.S.K.Nandy

Dept. of Electrical and Electronics Engineering,

NITK, Surathkal

Date of Submission: July 5, 2023

Department of Electrical and Electronics Engineering
National Institute of Technology Karnataka, Surathkal.

2022-2023

Acknowledgement

I am grateful to a number of individuals who played a vital role in the completion of my thesis work. Firstly, I express my sincere appreciation to Prof. Nandy Soumitra for agreeing to supervise my project. His extensive knowledge of many-core processor systems and insightful queries about my work helped me to attain the desired outcomes. I am also thankful to him for offering me the wonderful opportunity to collaborate with brilliant individuals and for providing me with all the necessary resources from the CAD lab at IISc, Bangalore for the successful implementation of my thesis.

I would like to extend my gratitude to Ravi Kiran, whose constant support and guidance were indispensable for the accomplishment of this project. Additionally, I want to thank my co-supervisor Prof. Yashwant Kashyap, who was always ready to assist me whenever needed.

Abstract

Our goal will be to automate the process of running test cases in the Redefine Resource Manager (RRM), which will be present as a core inside the Rocketchip based SoC. This will be done by designing customized hardware that will load an elf file onto the SoC using a series of IPs, such as memory mapped registers and FIFOs. Once the pipeline for running test cases on the SoC is setup, we would like to expand the project to allow optimization of the customized hardware, add the redefine core to the SoC and run some benchmarks to evaluate the custom hardware's performance.

Certificate

This is to certify that the Internship Report entitled

"Developing a Hardware Verification Framework for the Resource Manager of a Multicore AI Accelerator"

is submitted by
Hariharan Ayappane (191EE121)

as a record of the work done by him is accepted as the Major Project submission in the partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Electrical and Electronics Engineering, National Institute of Technology Karnataka, Surathkal

Supervisor Name:	Nandy Soumitra
Designation:	Professor
Department:	Computational and Data Sciences (CAD)
University Name:	Indain Institute os Science (IISC), Bangalore

Project Guide

Contents

1 Introduction

1.1 Accelerators for Artificial Intelligence

Artificial Intelligence (AI) has been a rapidly growing field in recent years and has found applications in many areas such as healthcare, finance, manufacturing, and more. However, the increasing demand for AI has also resulted in a need for faster and more efficient AI acceleration methods. This includes hardware and software methods for both training and inference.

AI acceleration is important because it allows us to train and run complex AI models faster and more efficiently. AI models require large amounts of data and complex algorithms to process this data, which can be time-consuming and resource-intensive. Acceleration techniques such as hardware accelerators, specialized processors, and software optimization techniques can help improve the performance of AI models and reduce the time and resources required to train and run them.

Furthermore, the need for AI acceleration is not only driven by performance but also by cost. As the amount of data generated by organizations and businesses continues to increase, the cost of processing and storing this data becomes a major concern. Acceleration techniques can help reduce the cost of processing and storing large amounts of data, making it more feasible for businesses to implement AI in their operations.

In summary, AI acceleration is critical for improving the performance and efficiency of AI models, reducing the cost of processing and storing large amounts of data, and making it more feasible for businesses to implement AI in their operations.

1.2 Project Goals

The REDEFINE processor was developed by Morphing Machines Pvt. Ltd, India. It is a multi core AI accelerator IP that is made up of a mesh of compute elements (CEs) and acts as a device that be connected to a host. However it cannot be directly connected to a host and needs an intermediate IP called Redefine Resource Manager (RRM) to supervise the allocation of tiles on the fabric. The proper functioning of the RRM is crutial for the REDEFINE processor. Hence, in this project we will design and synthesize an IP on Zynq UltraScale+ MPSoC (ZCU102), to run test cases on the RRM core. The core itself will be present within the Rocketcore SoC. The successful completion of a test case being written will be verified by reading values from the output FIFO buffer. IF the value read is 1, then the test case is executed successfully and the RRM functioning is verified. The verification will followed by a brief attempt to quantify the amount of clock cycles (FPGA) it takes to write a test case using a performance counter IP.

2 Background

2.1 Multicore processors

Multicore processors enable a significant upgrade from a single core processors using parallelism. There are essentially three types of parallelism to improve the performance of modern processors:

1. Instruction Level Parallelism (ILP): ILP involves executing multiple instructions at the same time in a single processor core. This is achieved by breaking down a single instruction into multiple smaller instructions that can be executed in parallel. This is typically done using techniques such as pipelining, superscalar execution, and out-of-order execution.

2. Data Level Parallelism (DLP): DLP on the other hand, involves dividing data into smaller chunks and processing them in parallel across multiple processor cores. This is achieved through techniques such as SIMD (Single Instruction, Multiple Data) and vector processing. In SIMD, the same instruction is executed across multiple data elements, while in vector processing, multiple data elements are processed simultaneously using different instructions.

3. Thread Level Parallelism (TLP): TLP involves dividing a single task into multiple threads and executing them concurrently across multiple processor cores. This is typically done using techniques such as multi-threading and multi-core processors. By dividing a task into multiple threads, TLP enables multiple tasks to be executed simultaneously on a single processor, improving performance.

All three forms of parallelism have their advantages and disadvantages. ILP is good for tasks with large instruction-level parallelism but is limited by the size of the instruction pipeline. DLP is good for tasks with large data-level parallelism but is limited by the size of the data set. TLP is good for tasks with large thread-level parallelism but can be limited by the number of available processor cores. The choice of parallelism technique depends on the specific task and the hardware available. In general, modern processors use a combination of ILP, DLP, and TLP to achieve the best possible performance. Multicore processors primarily use DLP to increase the efficiency that which data is processed.

To compare the execution of a small piece of assembly level code running on a single core processor and a multi-core processor, let's consider an example. Suppose we have a task that involves performing two independent calculations, A and B, which take 10 seconds each to complete. In a single-core processor, the task will take a total of 20 seconds, since the processor can only perform one calculation at a time. However, in a multi-core processor with two cores, we can perform the two calculations simultaneously, which reduces the total time to 10 seconds. This is because each core can perform one calculation at a time, so while the first core performs calculation A, the second core can perform calculation B. Thus, the total time taken for the task is the time taken by the slower of the two calculations, which is 10 seconds in this case. This example clearly demonstrates the advantages of using a multi-core processor over a single-core processor. In fact, the performance gain achieved by using a multi-core processor is not limited to just two cores. As the number of cores increases, the performance gain also increases proportionally, subject to limitations such as memory bandwidth and synchronization overheads.

To further illustrate the efficiency of multi-core processors, we can calculate the theoretical speedup achieved by using multiple cores. The speedup is defined as the ratio of the execution time of a task on a single core processor to the execution time on a multi-core processor. Assuming that the task can be perfectly parallelized, the theoretical speedup is equal to the number of cores. For example, if a task takes 100 seconds to complete on a single-core processor, and 20 seconds on a four-core processor, the theoretical speedup achieved is 5 (100/20). This means that the task can be completed 5 times faster on a four-core processor compared to a single-core processor.

In summary, the use of a multi-core processor can significantly improve the performance of tasks that can be parallelized. The performance gain increases with the number of cores and can be quantified by calculating the theoretical speedup.

2.2 ELF Files

An ELF (Executable and Linkable Format) file is a standard file format for executables, object code, shared libraries, and core dumps. It is widely used in many operating systems, including Linux, Unix, and macOS. ELF files provide a way to organize and package code and data in a format that can be loaded and executed by an operating system. According to Lu et al. (1995) [4] it provides programmers with the ability to dynamically control the execution flow during runtime through the utilization of appropriate tools. This makes ELF a powerful and versatile format for creating executable files.

The ELF format consists of a set of headers and sections that describe the different parts of the file. The headers include information such as the file type, architecture, entry point, and section table offset. The sections contain the actual code and data, along with additional metadata such as symbol tables, relocation information, and debug information.

ELF files are typically created by compilers and linkers, which take source code and produce an executable or library in ELF format. They can also be analyzed and manipulated using a variety of tools, including object dumpers, debuggers, and disassemblers.

Overall, ELF files provide a flexible and standardized way to package and distribute code and data in a format that can be easily loaded and executed by an operating system, and we'll be using it to load test cases into the memory.

2.3 Redefine Architecture

To understand the purpose of the Redefine Resource Manager, we must first examine the REDEFINE architecture in more detail. According to Khamvilai et al. (2019) [3] REDEFINE is a many core architecture that dynamically allocates applications to different cores. The whole setup consists of a host, external memory and the redefine chip. As per [7] the redefine chip has four main components:

1. Host: The host is responsible for configuring the FPGA, programming it with the required logic circuits which is nothing but the bitstream, and communicating with it during runtime if required. The host is usually a CPU (Central Processing Unit) or a microcontroller that controls the FPGA by sending it commands and data through an interface such as PCIe (Peripheral Component Interconnect Express) or JTAG (Joint Test Action Group).

2. External Memory: The device usually has limited memory resources. Therefore, external memory serves as a way to store data that is not immediately needed by the device. The host can also use the external memory to load data into the device's memory during initialization or during runtime. The external memory can be implemented using various technologies, such as DRAM, SRAM, or flash memory, depending on the requirements of the system. It is typically connected to the host or device via a high-speed bus interface such as PCI Express, DDR or HBM.

3. The Execution Fabric: It is a toroidal mesh of tiles connected through an Network on Chip (NoC). Tiles run user defined application node and can even contain fault actuating and detection mechanisms.Each tile consists of a router to direct the signals and a Compute Resource (CE) where he actual calculations take place. The edges of the fabric also contain extra tiles made of just routers (no CEs) to support communication with the host.

4. The Redefine Resource Manager (RRM): It is the interface between execution fabric and the host. It works on decomposing a given task into basic elements called HyperOps and managing the allocation of tiles for them on the execution fabric. The HyperOps are generated from C code in an offline compilation process and are the most fundamental element in the redefine tile. Once the HyperOps are launched and executed the results are obtained through the gateway tiles and transferred to the host. RRM supervises the fabric and is informed about an potential faults on the fabric. It is generally run on a raspberry pi board, separately from other raspberry pi boards that emulate the fabric itself.

3 Methodology

3.1 Tools Used

3.1.1 Chipyard

Chipyard is an open-source framework for building custom SoCs (system on a chip) developed by UC Berkeley's Architecture Research Group. It provides a modular hardware design flow that allows users to customize and create their own SoC designs using a collection of parameterized, open-source hardware components, called generators.

One of the key benefits of using Chipyard is its flexibility and ease of use. The framework allows users to easily select and configure hardware components to meet their specific needs, such as adding custom accelerators, changing cache configurations, or modifying interconnect topologies. This customization can lead to significant performance improvements and reduced power consumption compared to using off-the-shelf SoCs.

Additionally, Chipyard provides a complete software development environment that includes support for popular tools like GCC and LLVM compilers, standard Linux kernel and user space, and a variety of drivers and libraries. This allows users to develop and test their software on their custom SoC design, ensuring that it works seamlessly with the hardware.

Overall, Chipyard is a powerful tool for both researchers and engineers in the field of computer architecture. Its flexibility and ease of use make it an attractive option for those looking to design and test custom SoCs for a variety of applications, from high-performance computing to embedded systems. Here Chipyard was used to generate the RocketChip SoC.

3.1.2 RocketChip SoC

Rocket Chip is an open-source hardware design platform developed at the University of California, Berkeley. It is based on the RISC-V instruction set architecture and provides a flexible framework for designing and implementing custom processors and SoCs (systems on chips).

Rocket Chip includes several key components, including the Rocket processor core, the Chisel hardware construction language, the TileLink coherent interconnect fabric, and the Rocket Chip Generator tool. The Rocket processor core is a high-performance, in-order, single-issue RISC-V core, while Chisel is a domain-specific language for building hardware generators that can be easily customized and extended.

The TileLink interconnect fabric provides a scalable and efficient mechanism for communication between different components of a system, such as processors, memory controllers, and I/O devices. It supports coherent caching and memory ordering, allowing for efficient sharing of data between different processors or cores.

The Rocket Chip Generator tool is a flexible and customizable framework for generating and customizing SoC designs based on the Rocket Chip platform. It includes a set of parameterizable templates that can be used to quickly generate different types of designs, along with a set of tools for simulation, synthesis, and verification.

Overall, Rocket Chip provides a powerful and flexible platform for designing and implementing custom processors and SoCs based on the RISC-V instruction set architecture. Our focus will be to generate a generic framework with RocketChip and modify it according to our needs.

3.1.3 Vivado

Vivado is a popular software tool developed by Xilinx, a leading company in the field of programmable logic devices. It is an integrated development environment (IDE) used to design, simulate, implement, and debug Xilinx FPGAs, SoCs, and other devices. Vivado provides an intuitive graphical interface to design and configure hardware using a variety of languages including VHDL, Verilog, and SystemVerilog. The tool offers a range of features such as design entry, IP management, simulation, synthesis, place-and-route, and timing analysis. It also includes advanced features such as system-level design, partial reconfiguration, and high-level synthesis. Vivado is widely used by hardware designers, system architects, and researchers to create complex designs for various applications such as telecommunications, data centers, aerospace, and defense. Its versatility and ease of use make it a popular choice for hardware design projects of all sizes and complexities.

Vivado was used to design the hardware/PL (insert the diagram) The hardware consists of a Zynq processor connected to a core like rocketchip via a memory mapped register and FIFO (input and output) as intermediaries

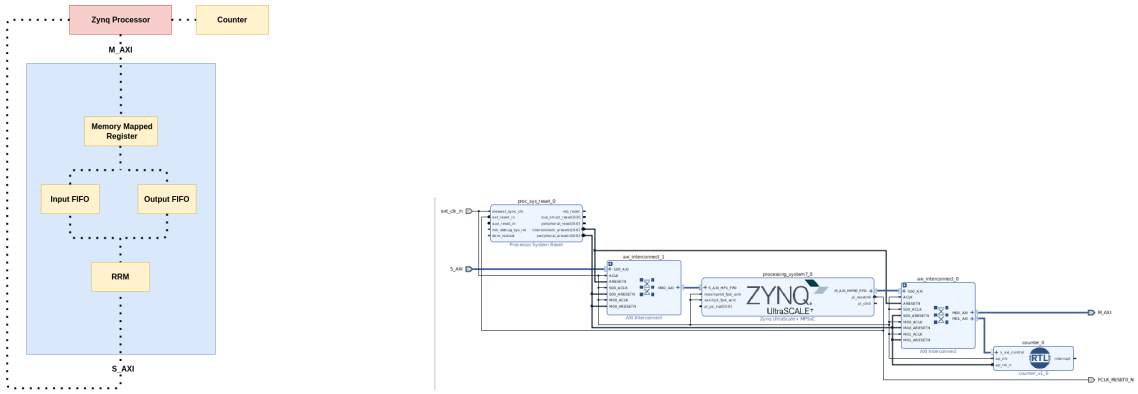
3.1.4 Vitis

Vitis is a software platform developed by Xilinx, a leading company in the field of programmable logic devices. The Vitis platform provides an integrated development environment (IDE) for software developers, allowing them to create and deploy accelerated applications on Xilinx's FPGA and SoC devices. The platform includes a wide range of development tools, such as compilers, debuggers, and profilers, as well as libraries and pre-built components that can be used to accelerate applications. Vitis also supports popular programming languages such as C, C++, and Python, making it accessible to a broad community of developers. With Vitis, developers can take advantage of the unique capabilities of Xilinx's devices, such as hardware acceleration and custom logic, to build high-performance applications for a wide range of industries, including data center, automotive, aerospace, and defense.

Use Vitis to run the software on the Processing System (PS/Arm core) in C++ The software consists of the elf file with 32bit instructions loaded in the form of an array Software starts by reseting the system Uses Xil_Out32 and Xil_In32 to write values to in FIFO and read values from out FIFO via the memory mapped register

3.2 Implementation of Verification Hardware

The custom hardware to run test cases on RRM contains the following units:



(a) (b)

Figure 1: Caption place holder

- Processor: In our pipeline we manually extract the individual instructions of the test case from the ELF file and place them in an array within processor memory. The processor then starts to write these instructions, one-by-one to the input FIFO. The original verification framework had a frontend server which used two important addresses to communicate with the RRM: A) TO HOST 8001000 and B) FROM HOST: 8001040. TO HOST is where the core puts the results of the computation. This where the result sits in. The original test setup of RocketChip uses something similar where the Zynq processor is sending the same data that SimSerial is sending to a memory mapped register, based on the memory go to input FIFO. All the ELF files are stored in the Memory Mapped register. Whenever RRM is ready to read the data. The Zynq processor has it's internal DDR memory. This memory is used as our memory mapped unit. The input FIFO loads all the contents from the memory. It sends it sends a software interrupt to CLINT, which is located inside RRM. This it wakes the core up from the wait for instruction mode (WF mode) and allows it to start taking instruction and data from the memory and start execute it. and then RRM will start fetching instruction and data from memory and write back to this address (output FIFO). The only way RRM interacts with the processor is through the FROM/TO HOST addresses.

- Memory mapped Register: Memory mapping is a technique used to access files and device memory beyond the physical RAM. Mapping device memory to main memory allows CPU to control the device with the same instruction set. This allows for efficient data transfers and reduced CPU involvement. Our design uses a memory mapped register to load the contents of the ELF file.

- Redefine Resource Manager (RRM): The resource manager for redefine that allocates tiled on the execution fabric. The RRM is added to the design after generating the RocketCore SoC using Chipyard and replacing the RocketCore with the RRM. This will need some minor

modifications in the interface of the RocketCore according to the requirements of the RRM interface.

- Input FIFO: It is where the memory mapped register data writes the input data and instructions. The write_check and write operations are used to interact with the input FIFO.
- Output FIFO: It is where the memory mapped register data writes the output data and instructions after execution. The read_check and read operations are used to interact with the output FIFO.
- Counter: It's a peripheral that is used to count the number of clock cycles it takes to finish executing a certain section of the code in the processor. It's discussed in detail under the bench marking section.

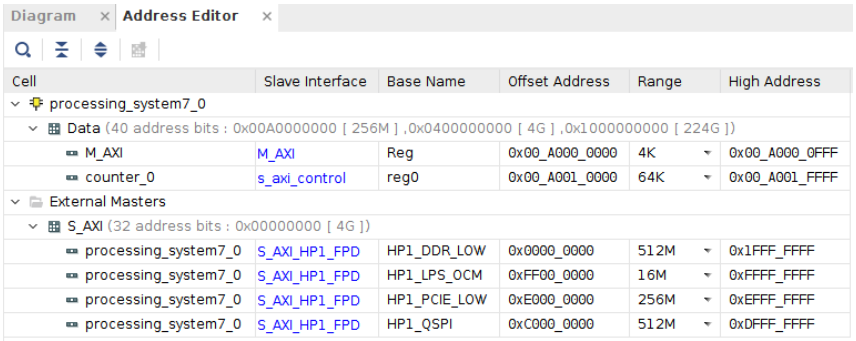
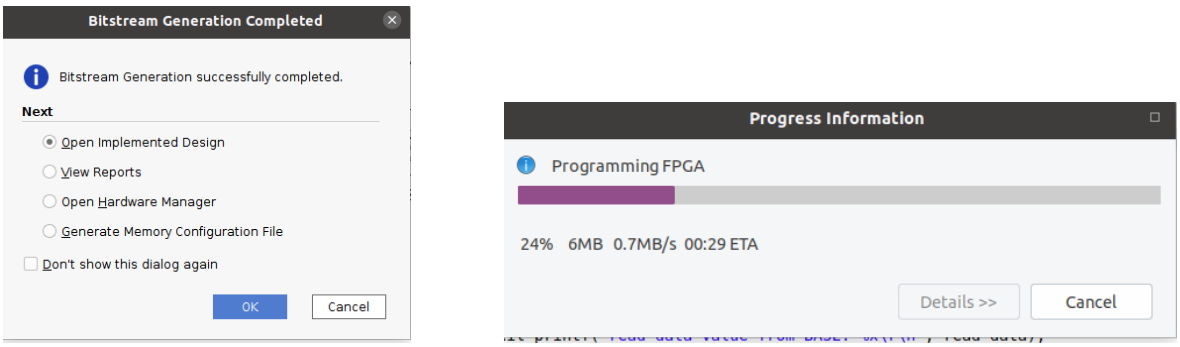


Figure 2: Address Mapping for the design



(a) Bitstream Generation Complete (b) Programming of the FPGA

Figure 3: Bitstream generation and programming

3.3 Implementation of Verification Software

C program to extract the test cases and produce the output ELF file. Instructions are extracted from an ELF file and output into a new file. The extracted instructions are placed in the form of

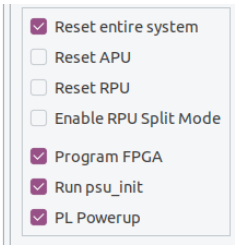
an array. The est used here has 1520 lines of instructions and so an array of size 1520 elements (1520 * 4 bytes) will be required. There are 4 main address locations which the read and write operations need to function. These addresses are expressed as offsets from a base address.

The software program run from Vitis, the following task :

- It first checks if the input FIFO is full, by reading data from the read_check (refer table below) address. If the FIFO is full, no more writing takes place to avoid stack overflow.
- If the input FIFO is not full, the program (or processor) writes all the required data and instructions to the write address of input FIFO.
- Once all the data and instructions are written and execution is completed, the program polls the read_check address of the output FIFO to check if it's empty.
- As long as the output FIFO is empty the polling process continues, until the value read from the output FIFO is 1. This indicates that the execution of the test case has completed successfully.

```
Xil_Out32(XPAR_M_AXI_BASEADDR + offset_reset, 1);
Xil_Out32(XPAR_M_AXI_BASEADDR + offset_reset, 1);
Xil_Out32(XPAR_M_AXI_BASEADDR + offset_reset, 1);
Xil_Out32(XPAR_M_AXI_BASEADDR + offset_reset, 1);
Xil_Out32(XPAR_M_AXI_BASEADDR + offset_reset, 1);
Xil_Out32(XPAR_M_AXI_BASEADDR + offset_reset, 1);
Xil_Out32(XPAR_M_AXI_BASEADDR + offset_reset, 0);
```

(a) Set reset (active low) before starting



(b) Run configuration setup

Figure 4: Software Setup

Address Name	Function	Offset
write check	check if input FIFO is full	0x0c
write	location to write values	0x08
read check	check if output FIFO is empty	0x04
read	location to write values	0x00
reset	System reset	0x10

Table 1: Address descriptions

3.4 Project Setup

3.4.1 Project Setup: From Scratch

Follow these steps to setup the project:

- Clone the following repository `https://github.com/li3tuo4/rc-fpga-zcu`

- The work directory here is /home/redefine/work/fpga-zynq/common/src.
- The model generated works only for Vivado versions 2017.1 and other vivado versions might compilation or upgrade errors for certain IPs, So make sure that the right version is installed.
- Open Vivado and run rc-fpga-zcu/zcu102/src/tcl/zcu102_bd.tcl from the TCL command line. This should start building the project.
- Once the project is generated, you open the block diagram for additional changes. Generate bitstream when done.
- The project contains the RocketChip core with RocketCore, which can later be replaced with RRM code. Both rockecore and RRM are quite similar and can be used interchangeable, with some minor changes in the ports used by RRM.
- After generating bitstream, generate the hardware for the project and export as a XSA file to preferred location.
- Start a new Application project with hardware as the XSA file generated in the previous step. Use the Hello World template for the software.
- Once the project is setup, replace the code helloworld.c with the elf loader described in the previous stage.
- Open the putty terminal in serial mode with the BAUD rate (speed) as 115200 and /dev/t-tyUSB0 as the serial line. Sometimes if ttyUSB0 fails to open, make sure to check other lines as well (ttyUSB1 or ttyUSB2).
- Build the project and Launch the project onto hardware. The tool will start programming the FPGA. Make sure the the FPGA is switched on and connected to the system.
- The terminal will change display each instruction as it is written into the write address. After completing all the write values, the read stage will begin and the value 1 will be displayed on the terminal upon successful execution.
- The ILA debugger can be used to debug any potential errors. Simply right click on the lines you wish to debug and select mark debug. Running connections after this will automatically connect an ILA debugger.

  Note: The Vitis application ma throw an executive context error. This can simply be overcome by restarting the FPGA.

3.4.2 Project Setup: From backups directory (Recommended)

- The backups directory contains backups of the most important working and failed versions os the design. You can select a version and open it using vivado, either directly from backups directory or after copying into a fresh working directory.
- Open a terminal in root and run the vivado startup script (./vivado2017.sh) script. The script automates running Vivado with the correct configurations.

- In Vivado navigate to ~/Desktop/backups/zcu102_new1_25_5_23_working_basic/zcu102_new1/zcu102_rocketchip_Zyn and open the project folder (.xpr) using open project.
- The ~/Desktop/backups also contains other versions of the project, like with/without DMA and with/without performance counter.

3.4.3 Debugging

Debugging will play a major role in the process of obtaining the expected results. Keep the following points in mind while debugging he project:

- If the amount of data displayed on the ILA debugger is less, try increasing the number of samples. This can be done by clicking on the debugger in the block diagram and changing the samples to values higher than 1024.
- RRM is based on the RocketCore in the rocket chip SoC, so setting up RocketChip on FPGA might give some insights.
- Make sure the array containing the ELF instructions is of the correct size. As a good practice, declare the array inline and use a size variable to keep track of it's size. If the size is incorrect then the Vitis window may crash or stall.

Listing 1: Inline declaration of array

```
//inline array decleration
int arr[] = {1, 2, 3, 4};

//size variable to keep track of array size
int size = sizeof(arr)/sizeof(arr[0]);
```

- If there are are any executive context running errors in Vitis try deleting the cache (.Xilinx/Vitis). If that doesn't work, restart the project from a new folder.

3.5 Benchmarks

A performance counter [1] is utilized to benchmark the speed of the ARM core processor to write the test case successfully. The objective is to measure the number of clock cycles in terms of the clock present on the FPGA board, it takes to write a test case so that we can have a quantification of the system's performance and how it changes with the addition of new peripherals if needed. We will use HLS to generate the hardware and co-responding drivers (API) for it.

The counter function is written in C. The RTL and co-responding drivers are generated using High Level Synthesis in Vitis HLS 2022.

Listing 2: Counter function in C for HLS

```
int counter(){
    #pragma HLS INTERFACE s_axilite port=return
    static int ctr = 0;
    ctr = ctr + 1;
```

```
    return ctr;
}
```

- Import IP or just use verilog code and import the counter module
- Drivers already present or can be imported to the include folder under project_name_bsp-/psu_cortexa53_0/include.
- Initialize the counter and start it using appropriate API calls from the drivers.

The drivers present a vast number of API calls to interact with the counter. The following example demonstrates the calls that will be used in our verification software to initialize and use the counter.

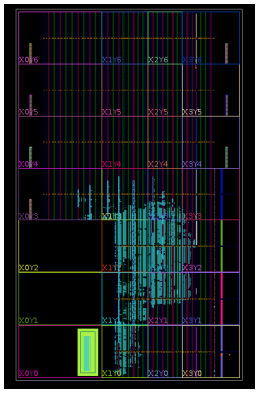
- First a counter object is declared from the XCounter struct (defined by drivers). The fields of the counter object are initialized to the base address of the counter and the ready flag.
- The counter is started and the auto-restart flag is enabled. This allows the counter to continue counting and not stop after a single iteration.
- After starting the counter it's value can be obtained by checking it's retuen value at any point.

Listing 3: Performance Counter API

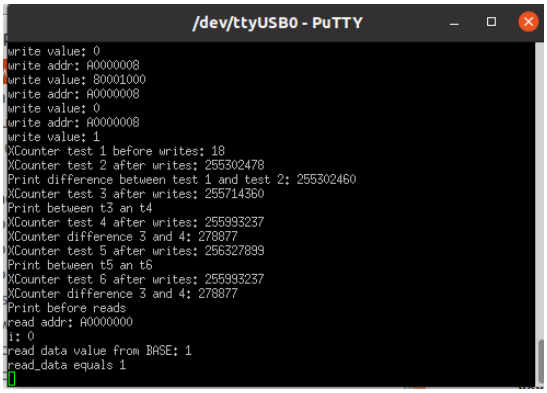
```
//Declare a counter object
XCounter xc;
XCounter *xcptr = &xc;

//Initialize the counter object
xcptr->Control_BaseAddress = 0x00A0010000;
xcptr->IsReady = XIL_COMPONENT_IS_READY;

//Start the counter (Allow to automatically increment)
XCounter_EnableAutoRestart(xcptr);
XCounter_Start(xcptr);

//Read the counter value indicating number of cycles passed
int cycles = XCounter_Get_return(xcptr);
xil_printf("Number␣of␣Cycles:␣%ld\n\r", cycles);
```

4 Results

The value read from the output FIFO after polling is 1, indicating that the test cases have run successfully.The counter IP developed using HLS has measured the number of cycles taken for the test cases as 255391989 (in terms of the FPGA clock). The LUT and RAM resource consumption for the IP is given below:



(a) Design Implementation



(b) Successful output at the putty terminal (USB0)

Figure 5: Results

WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
							27108	**12646**	**136.00**	**0**	**4**
10.262	0.000	0.013	0.000	0.000	4.168	0	27390	14134	136.00	0	4

Figure 6: Resource Utilization

5 Conclusion

We have developed a custom verification IP that can load and run an ELF file onto the Redefine Resource Manager (RRM) and emulated it on the Zynq UltraScale+ MPSoC (ZCU102). The IP has successfully loaded and executed a test case from an ELF file. The performance counter quantifies the speed at with the instructions of the test case are written, by using the clock cycles as a metric. This metric will allow us to further optimize the the IP if required and even measure the impact of adding new peripherals on performance. With this design as a base we can explore further optimizations to improve the resource consumption, memory utilization and speed of operation.

6 Future Work

6.1 Inclusion of PCIe using Quartus

The program-debug cycle using the Intel Quartus software stack is fairly cumbersome. The difficulties involved in setting the Signal Tap tool further slows down progress. Therefore, this project is focused on setting up Rocketchip on FPGA using Vivado to quickly find errors using ILA debugger. Since the bitstream generation is much faster on the beast system, the program-debug cycle is more feasible, and progress is satisfactory. Once a working design is obtained, we attempt to recreate it on the Quartus stack to leverage the benefits of PCIe.

6.2 Addition of Direct Memory Access (DMA)

The present design requires the processor to write each instruction from a test case to the input FIFO address. One possibility for optimization is to store the data in a small memory and outsource the data movement task to a DMA. The DMA would read data from the memory and write the data to the input FIFO buffer, treating it as a peripheral. DMA implementations in Vivado 2017.1 (eg: Central Direct Memory Access) use a AXI Stream (AXIS) interface which is not compatible with the AXI Lite interface required for the custom hardware we developed. Adding a new peripheral to convert AXI Stream connection to an AXI Lite connection might resolve this issue, but at the cost of increasing hardware (LUT, RAM) usage and will bring a additional complexity to the design. It's possible that newer versions of vivado will possess DMA IPs with AXI Lite interfaces, but adapting to a newer version would raise compatibility with the Zynq 7000 processor. Designing a new DMA IP that uses AXI Lite interface or is also an option. Custom DMA designs can also accomodate intelligent systems that use machine learning to improve efficiency in data movement and optimize resource and power usage.

6.3 Design Space Exploration (DSE)

During the course of the project we have explored multiple designs for running the test cases. However not all these designs are effective. We can use the performance counter to run comparisons across designs and finalize the most efficient design.

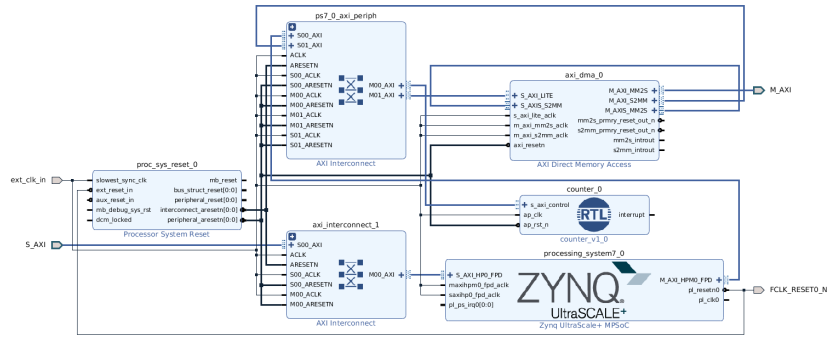


Figure 7: DSE: DMA placed between processor and AXI_S lead to bitstream fail

6.4 Test Case Automation

Presently the test cases are being generated by the spike simulator on a host machine, extracted manually and loaded using the software running on Vitis/Xilinx SDK onto the device (Zynq PS). This can be automated by moving the spike simulator onto the processing system (PS) directly. This would allow us to write hundreds of test cases without any manual intervention. However, the spike simulator contains a large number of libraries that cannot be compiled on bare metal directly. There are three approaches to deal with this problem:

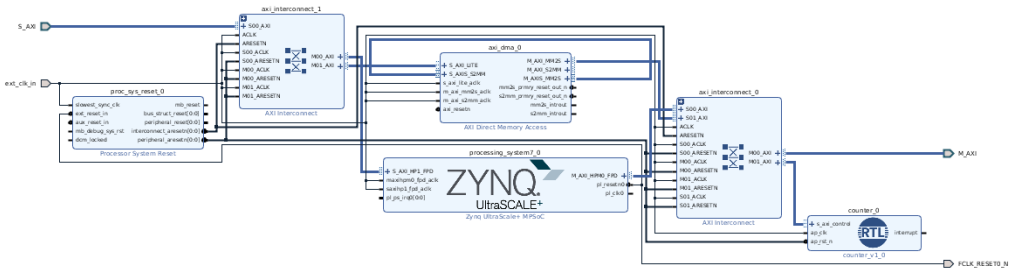


Figure 8: DSE: DMA placed parallel with processor and between AXIS_M and AXI_S lead to bitstream success

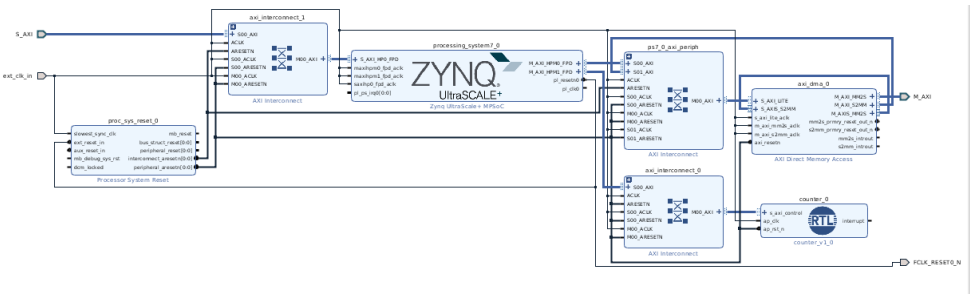


Figure 9: DSE: Design for AXI Stream based peripherals

6.4.1 Compiling equivalent libraries for embedded systems

There are various open source projects that demonstrate the use of POSIX and related libraries for embedded systems. Some libraries use a FreeRTOS instead of a standalone OS, the article [2] explains how one can use the FreeRTOS+POSIX library to run POSIX on bare metal. Any use of the pthread library will also require you to link the pthread library to the compiler [9]. The newlib library [5] consolidates various smaller C/C++ libraries, functions and system calls. It can be set up with the help of Linaro ABE script [6]. Another approach involves wrapping the required the C or C++ code for the g++ compiler as explained by [8]

6.4.2 Use a script to generate test cases on host and update to the SDK

Adding multiple custom libraries onto the PS will raise a lot of compatibility issues. An alternate approach would be to complete generation of all test cases on a host system that can support all the required libraries and only automate the movement of these test cases to the Xilinx SDK. These test cases can be loaded onto the PS during execution, using a custom peripheral like a data source IP. However, this means that the hardware needs to recompiled every time a new test case is added.

6.4.3 Using an operating system like Petalinux

The most direct way is to move away from bare metal and use an operating system like Petalinux. This will allow the software to compile all libraries and run spike simulator to produce the test

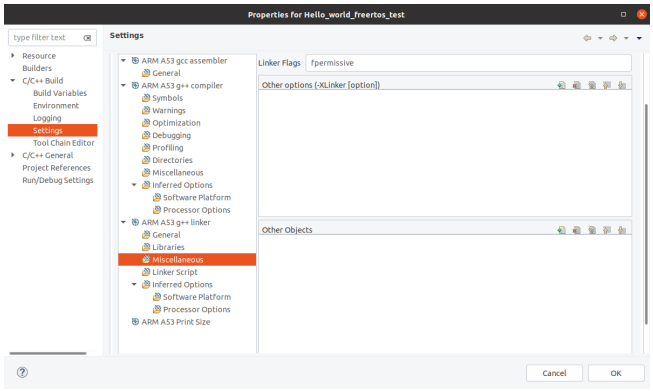


Figure 10: Link pthread before compilation of POSIX API

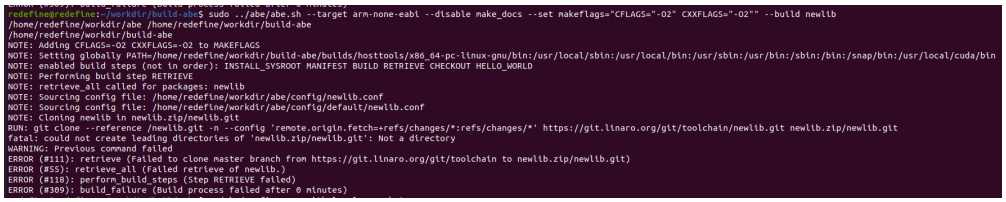


Figure 11: Newlib compilation wih the help of ABE

cases. However setting up an operating system will need more time.

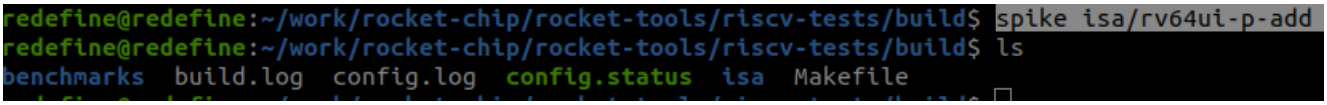


Figure 12: Spike was successful on host device with operating systems

References

[1] Nitin Chandrachoodan. *Lec90 - Demo: Performance counter AXI peripheral*. YouTube video. URL: `https://www.youtube.com/watch?v=J0pUoYpFqck`. 2019.

[2] REDS institute. *FreeRTOS+POSIX for Zynq*. Article. URL: `https://blog.reds.ch/?p=1671`. 2022.

[3] Thanakorn Khamvilai et al. "Task allocation of safety-critical applications on reconfigurable multi-core architectures with an application on control of propulsion system". In: *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*. IEEE. 2019, pp. 1–10.

[4] Hongjiu Lu. *Elf: From the programmer's perspective*. 1995.

[5] Newlib. *Newlib for embedded systems*. Article. URL: `https://sourceware.org/newlib/`. 2007.

[6] Newlib$_a$*be*. *Building newlib*. Article. URL: https://support.xilinx.com/s/question/0D52E00006hptKcSAI/instructions-for-building-newlib-in-xsdk-20182?language=en_US. 2018.

[7] Louis Sutter et al. "Experimental allocation of safety-critical applications on reconfigurable multi-core architecture". In: *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. IEEE. 2018, pp. 1–10.

[8] Xilinx. *SDK C++ projects using C source files*. Article. URL: https://support.xilinx.com/s/question/0D52E00006hpjZ8SAI/tips-for-sdk-c-projects-using-c-source-files?language=en_US. 2018.

[9] Xilinx. *SDK linux app error*. Article. URL: https://support.xilinx.com/s/question/0D52E00006hpcItSAI/sdk-linux-app-error-undefined-reference-to-pthreadcreate?language=en_US. 2017.