PROJECT A

Accelerator Controller Instruction Issue Bandwidth

Students:

Ghassan Shaheen | ghasan.sha@campus.technion.ac.il |ID: 206988461

Anas Mulhem | mulhem.anas@campus.technion.ac.il | ID: 322308966

Supervisor:

Alon Amid | alon@amidfamily.net

Document Version:

1.0

Date:

[08/03/2024]

Table of Contents

List of Figures

Abstract

A key property for accelerator controllers is instruction and task descriptor issue bandwidth. We would like to evaluate the open-source RISC-V processor ecosystem for processors which are most suitable to become accelerator controllers. We will define a small benchmark suit of code sequences representing typical accelerator configuration and offload execution flows for a range of accelerator programming methods such as task descriptors for MMIO accelerators and instruction sequences for extensible cores with custom instructions. We will then evaluate the available RISC-V processors using high-level instruction-count-based simulation. The goal of the project will be to create a taxonomy of processors for their suitability to different types of accelerator complexity based on accelerator input and configuration properties. We found key Accelerator traits that affect its performance significantly. We will discuss 2 main types of Accelerators (i.e. MMIO Peripheral and RoCC), discuss their key traits that affect the CPU's performance, and present the outcome in plotted graphs.

Introduction

Accelerators have emerged as game-changers in the realm of computing, significantly influencing and augmenting the performance of central processing units (CPUs). These specialized hardware units are designed to offload and accelerate specific tasks, alleviating the burden on the CPU and unlocking unprecedented levels of computational efficiency. in this project, we aim to research two types of Accelerators: MMIO (Memory Mapped Input Output) and RoCC (Rocket co-processor). We will discuss each of them later on. We aim to study the impact of each type of Accelerator on different Cores based on their Parameters and Attributes. An Accelerator is defined by its own set of parameters, lets address each of them and how they can affect the efficiency of the Accelerator:

Number of Configuration registers. Each Hardware Unit needs its own set of configuration registers (Configuration registers are necessary to customize and adjust the behavior of digital systems, providing flexibility, optimization…). Each Configuration of those registers is time consuming and will affect the Accelerator's performance.

Configuration registers width. The width of configuration registers can vary depending on the complexity and requirements of the system. In some cases, a few bits may be sufficient to represent all the necessary configuration options, while in other cases, a larger number of bits may be required for more detailed control.

Compute command latency. Eventually, accelerators play the rule of computing a well-defined set of instructions and execute them in a defined period of time. This parameter satisfies this Rule. Example, if the Accelerator performs matrix multiplication, then its custom command will be the matrix multiplication. The Accelerator will execute this custom command in a given amount of time, which is the discussed parameter. Note that the SoC of which the Accelerator is placed in might limit this attribute - if the SoC in which the Core & Accelerator are placed upon can't keep up with the Accelerator's Latency.

Data Width. Data width refers to the number of bits that can be processed simultaneously by the processor or transferred between different components of the chip. Data width is crucial for determining the maximum amount of data that can be processed or transferred at any given time, which directly impacts the performance of the chip.

Address Width. Address width refers to the number of bits used to represent memory addresses within the chip's memory space. It determines the maximum amount of memory that the chip can address directly. The formula $2^{address\_width}$ gives the total number of unique memory addresses that can be accessed. For example, a chip with a 32-bit address width can address up to 2^{32} (4 GB) of memory. This parameter can affect the Accelerator's performance since we may need multiple transactions to pass the address from the Core to the Accelerator. As we will see later on in the project.

Buffer Width. The buffer width parameter will impact our ability to write data, if our Data width is larger than the buffer size then the data will be chopped and divided into more than one beat of data (the processor's responsibility)

Memory Bandwidth. measured in bps (bytes per second)

Background

Accelerators

Accelerators can be added to your SoC in several ways:

- MMIO Peripheral (TileLink-Attached Accelerator)
- RoCC Accelerator (Rocket Co-Processor)

These approaches differ in the method of the communication between the processor and the custom block. In order to understand them, let us illustrate how a chip Tile looks like:

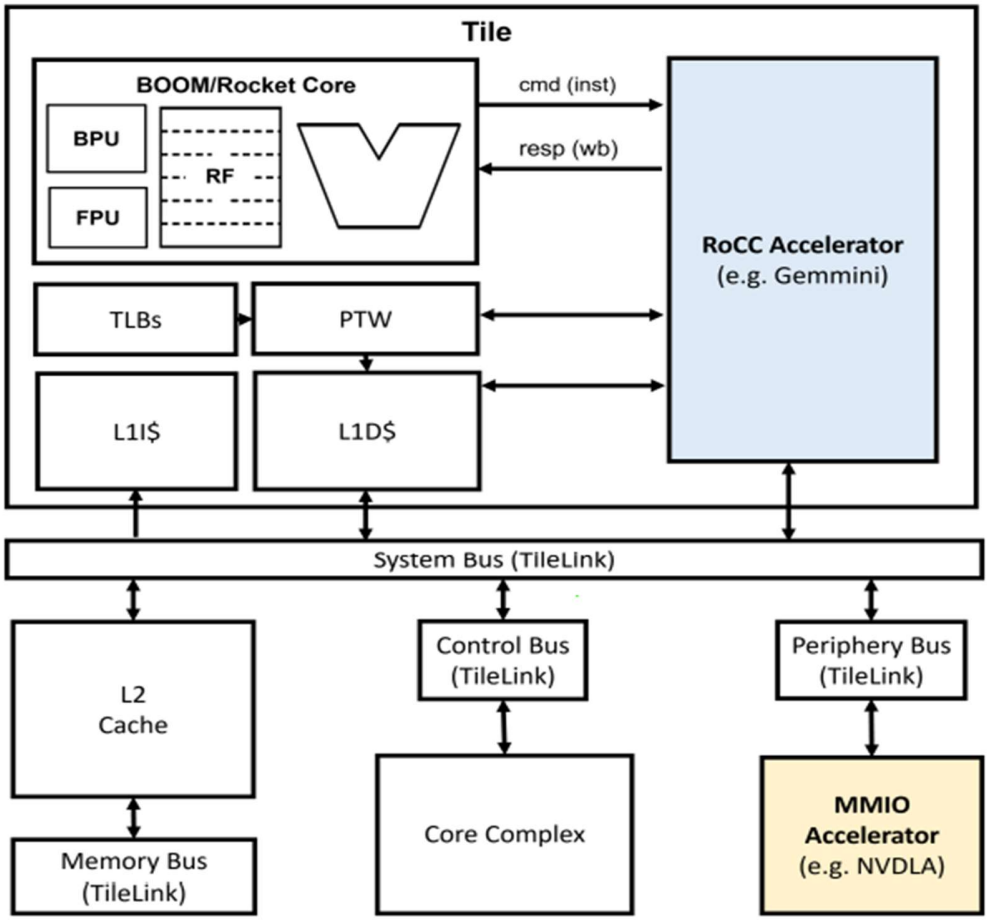

Figure 1

One can see that the RoCC Accelerator is placed beside the Core on the Tile (the processor communicates with a RoCC accelerators through a custom protocol and custom non-standard ISA instructions reserved in the RISC-V ISA encoding space, as we will discuss this later). While in MMIO Peripheral, the processor communicates with the peripheral through memory-mapped registers utilizing the TileLink bus protocol. This is one of the key differences that we aim to address in our project! We shall present each of the Accelerators separately.

RoCC - (Rocket Co-Processor)

RoCC defines a simple interface between a CPU core and a co-processor to support memory system access, pipeline stalls and communication of register values. Signals communicating custom instructions and results between the core and accelerator, as well as some additional interface signals to the core's L1 data cache, PTW, and FPU.

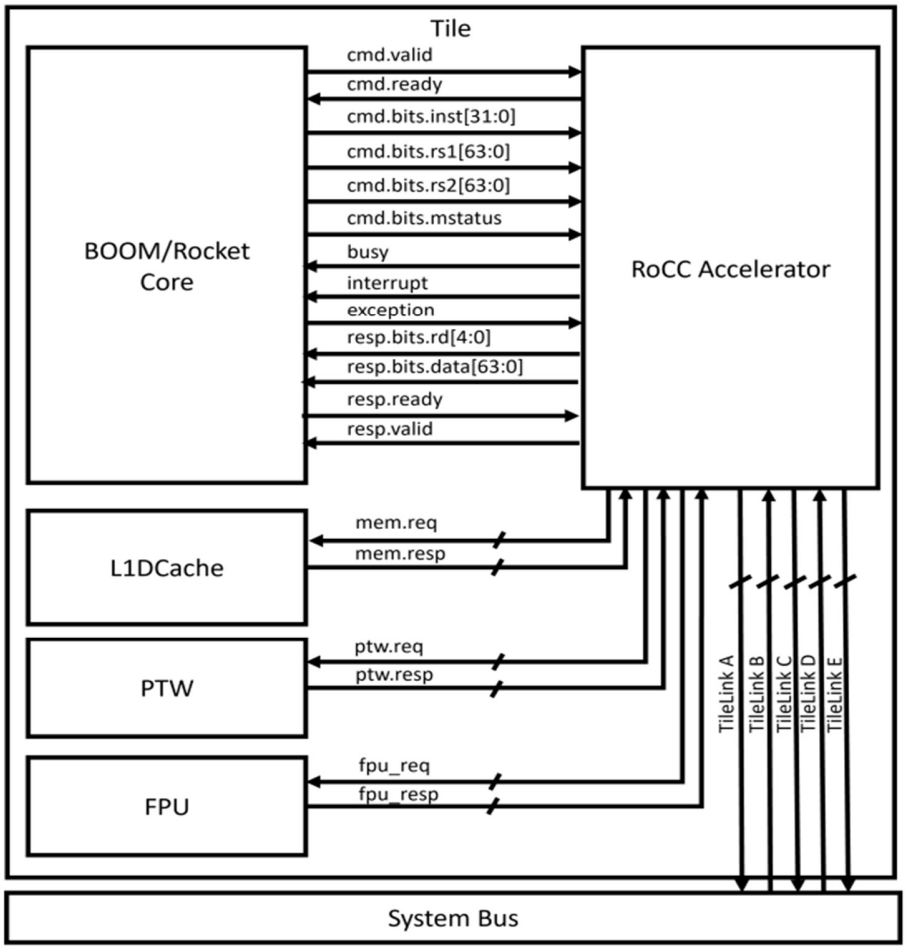

Figure 2 - RoCC-Core interface

The interface from the CPU core to the co-processors includes:

a) Busy signal – a Boolean value which represents the state of the Accelerator.
b) Command – the command communication between the Accelerator and processor is via a master slave protocol which includes a usage of ready valid signals indicators per each transaction. In addition to that, on the interface, we pass (some of the values):
   - Funct – instruction value.
   - Data out (Accelerator's output)
   - Two source registers, rs1 and rs2.
   - One destination register, rd.
   - command Ready & valid, resp ready & valid – implementing the master slave protocol.

C) Exception and interrupt signals.

Cores that support the RoCC interface communicate with the co-processor through a custom protocol and custom non-standard RISC-V instructions reserved in the RISC-V ISA encoding space. The RoCC protocol enables RoCC accelerators to access the L1 data caches, stall the processor pipeline, and pass values through registers. In some sense, a RoCC accelerator can be thought of almost like an additional functional unit within the CPU.

MMIO

Memory-mapped peripherals serve as a prevalent technique for integrating custom accelerators into SoCs. Within a memory-mapped peripheral setup, communication between the processor and the accelerator happens via memory-mapped registers utilizing the TileLink bus protocol. In Chipyard, peripheral accelerators link to the SoC through a periphery bus and are programmed via memory-mapped I/O (MMIO). Given that memory-mapped I/O (MMIO) necessitates fewer assumptions about the CPU compared to RoCC accelerators, it emerges as the favored integration method for third-party accelerators and larger independent subsystems within Chipyard.

The integration of MMIO accelerators requires a modification of the SoC memory map. Different combinations of peripheral accelerators can result in different SoC memory maps. One should be careful when specifying the memory mapping and avoid choosing pre-occupied memory space.

Cores

Cores, an integral component in computing. Within a processor, each core operates independently, executing instructions and performing calculations. Their significance lies in facilitating concurrent task handling, fostering efficient multitasking and parallel processing. Furthermore, cores collaborate seamlessly with accelerators. This relation between cores and accelerators optimizes performance across diverse computing tasks, from executing intricate algorithms to managing system operations. In today's technology characterized by escalating demands for computational power, the relation between cores and accelerators is fundamental.

We intend to implement benchmarks using two specific cores, BOOM and Rocket, to assess how each accelerator interacts with these cores. This approach will enable us to pinpoint optimal configurations and underscore the primary differences between BOOM and Rocket Cores.

Rocket Core

Rocket Chip is an open source SoC originally developed at UC Berkly, Rocket Chip chose to use a Rocket Core which is a 5 to 6 stages pipelined single-issue in-order processor. The Rocket core supports the open-source RV64GC RISC-V instruction set and is written in the Chisel hardware construction language. Chipyard uses the Rocket Chip generator as the basis for producing a RISC-V SoC.

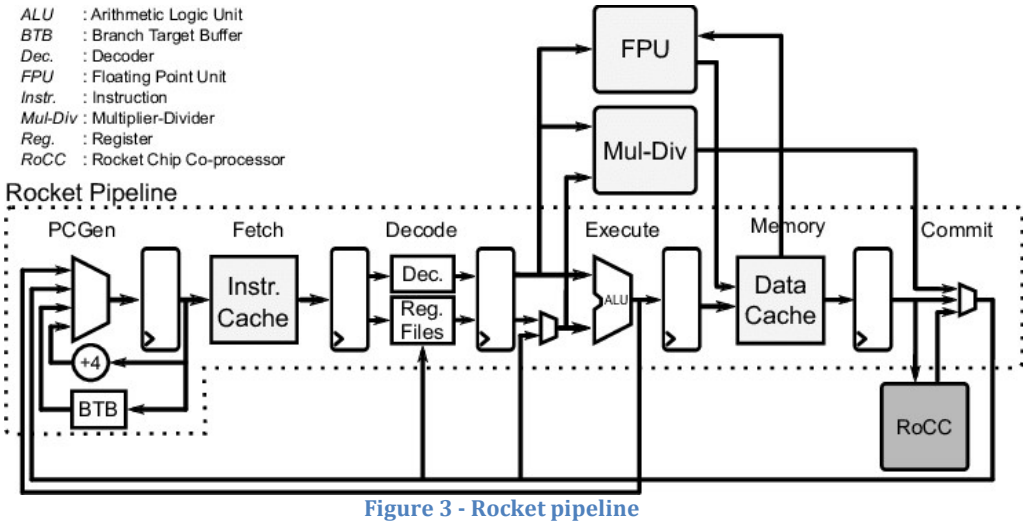

Figure 3 - Rocket pipeline

In Figure 4, one can see a whole Rocket Chip system, using Rocket Core:

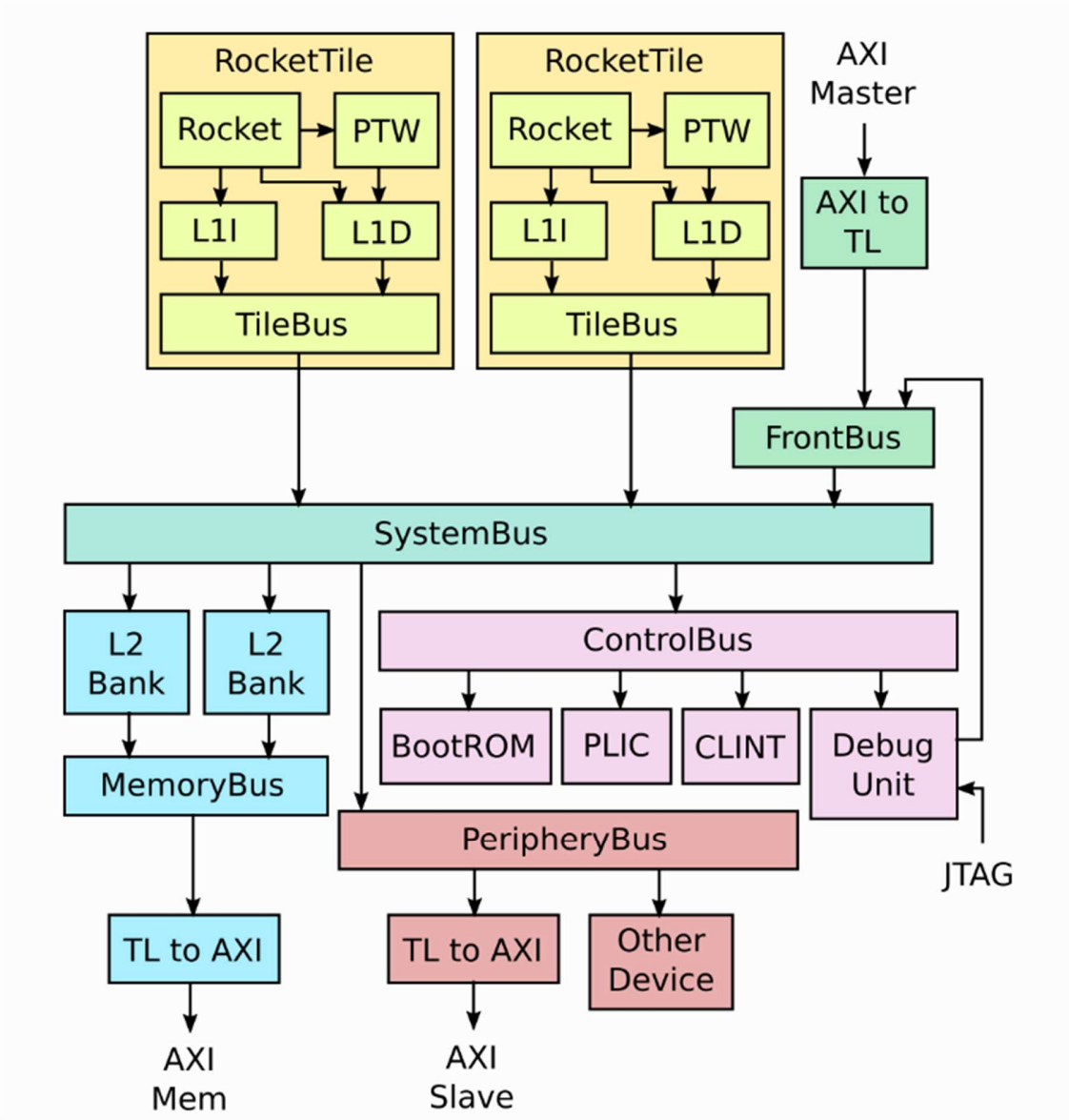

Figure 4 - Rocket Chip system

The figure is for illustration purposes and we shall not dive into its details.

BOOM Core

The Berkeley Out-of-Order Machine (BOOM) is an open-source RV64GC RISC-V core designed in the Chisel hardware construction language. It can substitute the Rocket core provided by Rocket Chip, replacing the RocketTile with a BoomTile. BOOM utilizes a unified physical register file design, also referred to as "explicit register renaming". In principle, BOOM operates across ten stages: Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback, and Commit. However, in the current implementation, several of these stages are merged, resulting in a total of seven stages: Fetch, Decode/Rename, Rename/Dispatch, Issue/RegisterRead, Execute, Memory, and Writeback. (Commit happens asynchronously and therefore is not included as part of the pipeline.)

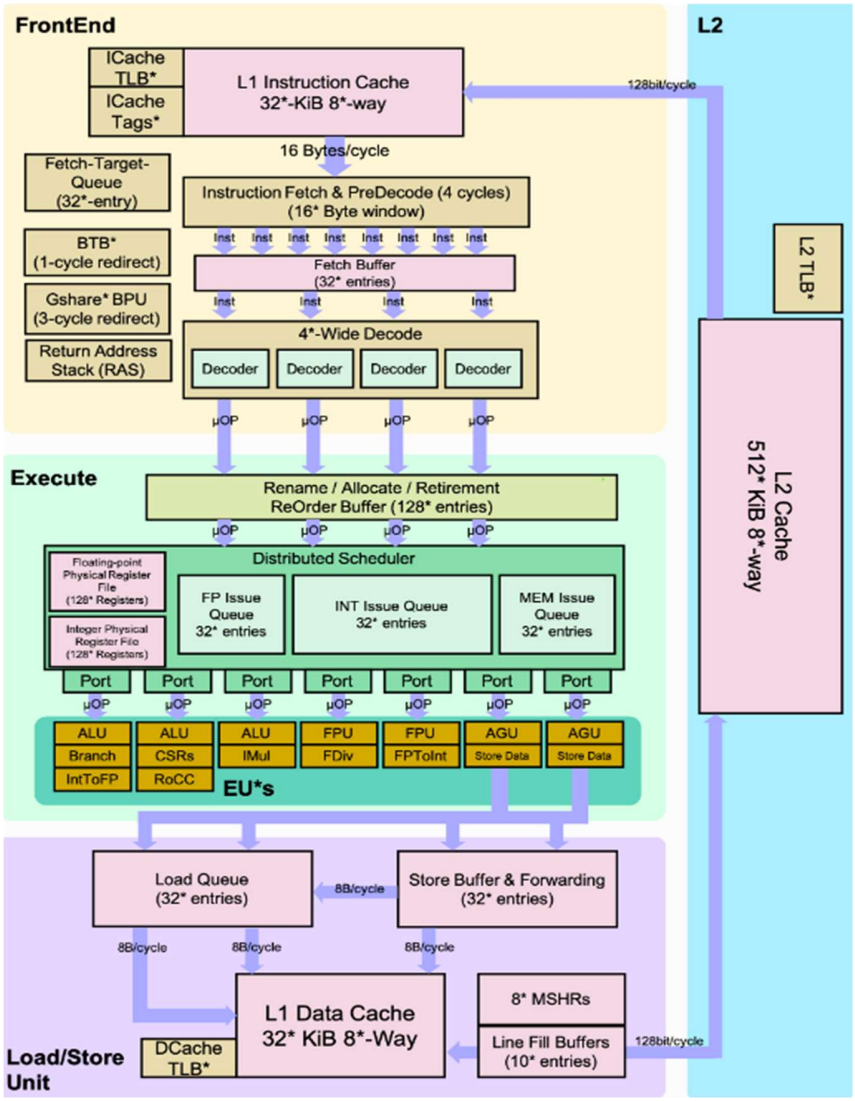

Figure 5 - BOOM Core

Coding languages

- In order to asses our Accelerator's performance, we designed Benchmarks with C programming language.
- To design the Accelerator's RTL, we used Verilog.
- In order to connect our Accelerator's RTL to the CPU tile (with the help of Chipyard's Framework) we used Scala/Chisel Code. Chisel is an embedded language within Scala that provides a set of libraries to help hardware designers create highly parameterizable RTL. More on that later on.

Chipyard

- What is Chipyard?
  Chipyard is a framework for designing and evaluating full-system hardware using agile teams. It is composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of SoCs. Chipyard provides extensive inter-core configurability and composition capabilities.

Using Chipyard's framework was an integral part of our project, it enabled us to design and implement our own Accelerator (further on that later), and incorporate it into an SoC with our own choice of Cores (BOOM/Rocket).

We integrated our RTL and Scala code to Chipyard's framework, connected them to the Top-Level traits and ran our Benchmarks on the integrated SoC.

Chipyard provided many examples of MMIO/RoCC Based accelerators that we used as a compass during our project (For RoCC we used the SHA3 Accelerator, for MMIO we used the GCD Accelerator).

Implementation

Verilog/RTL

Our Verilog code implements a simple generic Accelerator which only counts latency. In other words, a "dummy" Accelerator. Our goal when writing the Accelerator's RTL wasn't to design an intricate RTL implementation of a super Accelerator, but rather define a simple one which can mimic the behavior of a real Accelerator that is placed on a SoC. We don't care about the calculation outcome of the Accelerator (what is the data/values it outputs) because there are many types of Accelerators and we weren't to research a specific one of them, but rather look at the Accelerator's traits/parameters to see how they affect the whole CPU's performance. Our implementation took into account some of the parameters Set that is defined in the project's Introduction section, each one of them affected the Performance in some way or another, as we will discuss this in the results section. You can check out each of our Accelerator's implementations in the project's github link (the MMIO and RoCC RTL) under:

src/main/resources/vsrc/

It is Important to note that the communication between the Verilog Module and the Core is via a master-slave protocol, one toggles output_valid when it is ready to return the data, while the other needs to toggle its ready signal in order to receive the data. And vice versa. Tread carefully and Coordinate, synchronize well between those signals! For a transaction to be completed, the mentioned handshake shall be performed flawlessly.

Per Accelerator, we defined its inputs and outputs. These inputs and outputs differ between RoCC and MMIO, since in RoCC we have to connect the Accelerator to the interface signals as described in the previous section (FPU, Core, PTW, L1 – please refer to Figure 2). While in MMIO we shall connect only the valid ready signals that implement the master-slave protocol with addition to the data signals and funct. Afterall, MMIO peripheral does not implement the Core interface that RoCC needs to supply.

In RoCC, we have many un-used signals within the Core interface, since we want to research the connections between the Accelerator and the Core, we do not care about the signals with the FPU, PTW and L1, hence we tied them to 0. In addition, the Core-Accelerator interface supports interrupt and exception bits, they also are tied to 0. One can further and expand this project by handling interrupts and exceptions within the Accelerator – for RoCC co-processors need to handle their own exceptions and interrupts.

Benchmarks

In order to asses our Accelerator's impact on the CPU's performance, we designed a couple of Benchmarks (one per Accelerator – MMIO/RoCC) written in C Code, that included a set of instructions {COMPUT, CONFIG}.

- CONFIG – this command when sent to the Accelerator, it writes the specified values in the Configuration registers of the Accelerator. Every HW Unit has its own set of configuration registers, this operation is vital for the Unit, for it defines its basic configuration.

- COMPUTE – every Accelerator is defined over a set of specified instructions which it aims to improve performance, this command is supposed to mimic a common Compute command that each Accelerator might have.

The latency of each command above is different, and as another attribute of this project, we assigned each compute command a set of different values and searched for the optimal latency per each command. This comes to hand in our Verilog module implementation.

It is to be noted that the Benchmarks address each of the researched parameters in a different way, this can be easily distinguished within the code. In addition to that, the benchmarks are simple ones which's main goal is to determine the accelerator's performance according to the specified configuration.

Special case:

Before diving into the details of each benchmark, it's important to address a specific case that is handled identically across both benchmarks: where the address width is greater than configuration register width. In such cases, multiple configuration registers are required to accommodate a single address. To illustrate this, let's consider an example using pseudo code.

Address Width = 32 bits // example: address = 0x20001259

Register width = 8 bits // in order to pass the address to the Accelerator, we will need 4 configuration register (32/8 = 4)

The attached pseudo code presents how we intend to pass the data to the Accelerator's registers in those cases:

```
uint32_t Address = 0x20001259
uint8_t my_config_reg; // 8 bits width
for (int i = 0; i < 4; i ++)
{
    write_addr = (uint8_t) (Address >> i*4); // the data to be written
to the config register
    register_write(my_conf_register, write_addr); // write the data to
the register

}
```

Figure 6– Address width > reg width pseudo code

the first register will receive the value of 0x59, the second 0x12, third 0x00, fourth 0x20. Finally, the Verilog module would perform a concatenation on those 4 registers and assemble the desired address (0x20001259).

This relation will be discussed later on as one of the main factors that shall affect the overall Latency.

RoCC Benchmarks

You can view the code in the github link under:

Software/tests/src/gemmini-rocc-tests/bareMetalC/

ROCC_INSTRUCTION command

The processor communicates with a RoCC accelerators through a custom protocol and custom non-standard ISA instructions reserved in the RISC-V ISA encoding space. RoCC coprocessor instructions have the following form:

```
customX, rd, rs1, rs2, funct
```

The X will be a number 0-3, and determines the opcode of the instruction, which controls which accelerator an instruction will be routed to (we chose 2), the custom command also receives 2 source registers rs1 and rs2, one destination register rd. We need to specify the funct input – for it is used in order to indicate to our Verilog module which command it should execute (The funct field is a 7-bit integer that the accelerator can use to distinguish different instructions from each other) – we defined that: funct = 1 refers to CONFIG command, funct = 2 for COMPUTE command.

Choosing the appropriate RoCC custom command might be challenging, one need to consider variable details, such as how many input/output registers, the funct parameter, under which opcode to define the custom command. Please refer to the documentation in the:

- riscv manual in section 2.2 Base Instruction Formats
- Github xcustom.h (offers a set of custom commands)

Time measurement – rdcycle()

We used a rdcycle() command to measure time. When one measures time in riscv based architecture environment better use rdcycle() than standard time library functions within C code. For rdcycle() offers precise results in riscv based systems.

MMIO Benchmarks

You can view the code in the github link under:

Software/tests/src/

In the MMIO Benchmark, we do not need a special custom instruction to write on the peripheral, rather we access it via memory mapping. Our Benchmark relied on 3 main principals:

1. Status register, which is a concatenation between 2 very important signals:
   Status = {input_ready, output_valid}
   Input_ready (as an output to our MMIO peripheral) indicates that our Module is ready to receive a new command.
   Output_valid (as an output to our MMIO peripheral) indicates that the peripheral has finished its calculation and has the result ready.
   As part of the previously discussed hand-shake protocol. We shall use this register in our Benchmarks as an indication to the peripheral's state.

2. Polling – before and after each command accessing the peripheral, we shall wait for the Accelerator to be ready - by reading the status register's value. As we are in MMIO peripheral, the register read will go through the system bus memory system. Example:
   2.1. Before writing to peripheral: we wait for status to be 0x2 (Input_ready = 1, output_valid = 0)
   2.2. Before reading the outcome of the accelerator we wait for the result to be ready, and wait for the status to be 0x1 (Input_ready = 0, output_valid = 1)

3. As previously mentioned, the MMIO peripheral relies on memory Access to be able to read data. Hence, we need to define the base address of our peripheral (we choose it to be 0x50000 – one needs to be weary of not choosing an address which is reserved for other memory usage within the system, as we had encountered some issues when choosing an incompatible address (it was pre-occupied with other memory usage). The base address is defined in the Scala that describes the MMIO peripheral, so one needs to coordinate with it. According to the register mapping (specified in the scala files) one needs to define appropriate address to access various registers throughout the Benchmark (in our case, status and funct registers). It is important to coordinate between the mapped registers width and the Addresses they can occupy. Chipyard supports Addresses of 32 bits width, hence a register of width 64 bits for instance, can't be mapped into one single address and need to special handle those specific cases gently.

With that said, the principal of the benchmark becomes simple to comprehend: define the base address of the peripheral, calculate the registers offsets from it. Before each access to our peripheral we shall wait for it to be ready by perform a polling of the sort:

```
While(register_read(STATUS_REG_ADDR) & 0x2 == 0);
```

Afterwards, write to the Accelerator. In order to read the output of the Peripheral, perform another polling and wait for the polling result to be 0x1. Read the result.

Repeat...

Note: unlike RoCC – there is no special custom command in MMIO in order to access the peripheral. The "custom" command in this case is a memory read & write upon the desired address, and according to the width of the variable mapped to the address.

For time measurements, we also used rdcycle().

Bash Script

When changing the parameters/attributes of the Accelerator and running the Benchmarks, we have to compile and run everything, when manually modifying the desired configuration. So, in order to ease ourselves, we designed a bash script that automatically changes the values of the desired parameters according to our chosen configuration and modifies the desired files. It outputs the result of each test iteration and logs it to our csv output file. We use this csv file to plot our graphs. You can find the script under:

src/main/scala/param_change.sh

it is to be noted that both of our Accelerator have very similar bash scripts, since we aim to check the same traits upon each peripheral.

Chisel & Scala

Chisel is a hardware construction language developed at the University of California, Berkeley. It is designed to support the creation of complex digital systems using a high-level, object-oriented programming approach. One key advantage of Chisel is its ability to generate synthesizable Verilog or VHDL code, which can then be used for FPGA synthesis or ASIC fabrication. This allows designers to leverage the productivity benefits of a high-level language while maintaining compatibility with existing hardware design tools and workflows. We will use this key trait in order to be able to evaluate our benchmarks on the Verilog Accelerator modules we designed, and connect our Accelerator's design to the SoC. As previously mentioned, our Cores, BOOM and Rocket are written in Chisel construction language.

It is to be noted that Chisel is built on Top of Scala code, Scala is a general-purpose programming language known for its scalability, conciseness, and functional programming capabilities.

within the MMIO Scala files, we defined the previously discussed memory mapping, we map every input/output signal of the periphery via address map (RegMap) which specifies the address of each specific input register.

Helpful implementation for Scala code:

- RoCC
- MMIO

Under our github link, one may find the scala code for each Accelerator via:

src/main/scala/

Debug

Debug tools

The tools that we used for debug purposes should be noted:

- gtkwave – a very useful tool that we used to view the waveforms of the benchmark. Waveforms are helpful for debugging RTL designs since they visually represent signal behavior over time, aiding in identifying timing issues. In order to open the gui, simply run gtkwave
- Prints – a very common form of debug…
- Look at a reference model that works and compare the differences. This was a very efficient way of debugging, for this highlights the key differences that might cause the issues, hence leading to a solution, or at least a conclusion.

Debug guidelines

A debug procedure might be challenging. We will illustrate some of the most useful debug approaches that helped us throughout our project:

- When running the make command, the Chipyard env generate Verilog files that connects the Verilog module to the Core/memory map (according to the Accelerator), Command Router and many more modules. It is important to research how the Verilog module is connected to the Core via the Command Router, this is important to verify that the commands do get sent correctly from the core to the Accelerator's Verilog module. For instance, when a test gets stuck (in an infinite loop / some other arbitrary error in the command routing or miscoordination between the Accelerator and Core, etc.) and the command doesn't reach the Accelerator, one can start by viewing the command in the core (the starting point) and see how it should make its way to the Accelerator. By looking at the internal implementation and connections between the Accelerator, Core and command router, one can come to many conclusions.

To view those modules, grep the name of your Verilog module name under chipyard/sims/verilator/generated-src/chipyard.TestHarness.<RocketConfigName>/ and look for the top file (for example, in case of MMIO Accelerator, look for a file named MMIOTL.sv).

- Generate the assembly code of the test, and look at the PC (Program Counter) signal in the waveforms – usually you would find it in the core's module. This is very helpful in order to come for some insights regarding your test and its progress. In order to generate the assembly:

  cd to gemmini-rocc-tests/build/bareMetalC/ and run:
  Riscv64-unknown-elf-objdump -d
  <BINARY_FILE_NAME_OF_C_TEST>

Results

As previously explained in the Introduction section, our aim is to discuss the effect of different Accelerator traits on the total CPU's performance. In this section we wish to present our final results, explain how the different Accelerator configurations affect the overall Benchmark results. We will discuss 2 types of Accelerators: RoCC, MMIO.

In each of the Results sub-sections, we concluded 3 conclusions, each addressing specific parameters, in this form we will be able to present the main differences between both Accelerators upon the same configurations isolating the one (or more) discussed parameter. The parameters that we explored are:

1. Num of Configuration registers.
2. The width of configuration registers.
3. The Address width.
4. Compute command latency.

The aggregate graph is a 5-D graph which will not be very implicit. Hence, we isolated each parameter and discussed it separately.

Please Note that – in many of our conclusions, we address the PC (Program Counter) which relies on the assembly code, please refer for the appropriate attached assembly code.

Note – one can find the appropriate csv file that depicts the Accelerators results for each configuration in the project's github link under results/<Accelerator>.

In the following discussion, we analyze our benchmarks with two different types of Accelerators (MMIO, RoCC) over a Rocket Core, which is a 5 to 6 single-issue pipeline in-order core. For further details please refer to the appropriate section in the Background.

RoCC Results - Conclusions

Conclusion 1 – number of configuration registers

Let us view the case in which our main attribute is the number of configuration registers, let us look at an example where:

I. Latency of compute command = 200
II. ADDR_WIDTH < CFG_REG_WIDTH

The mentioned parameter affects the number of the CONFIG commands that we will execute. Reminder: in our Verilog implementation, a configure command takes a total of 6 cycles to be executed. If we look at the following graph:

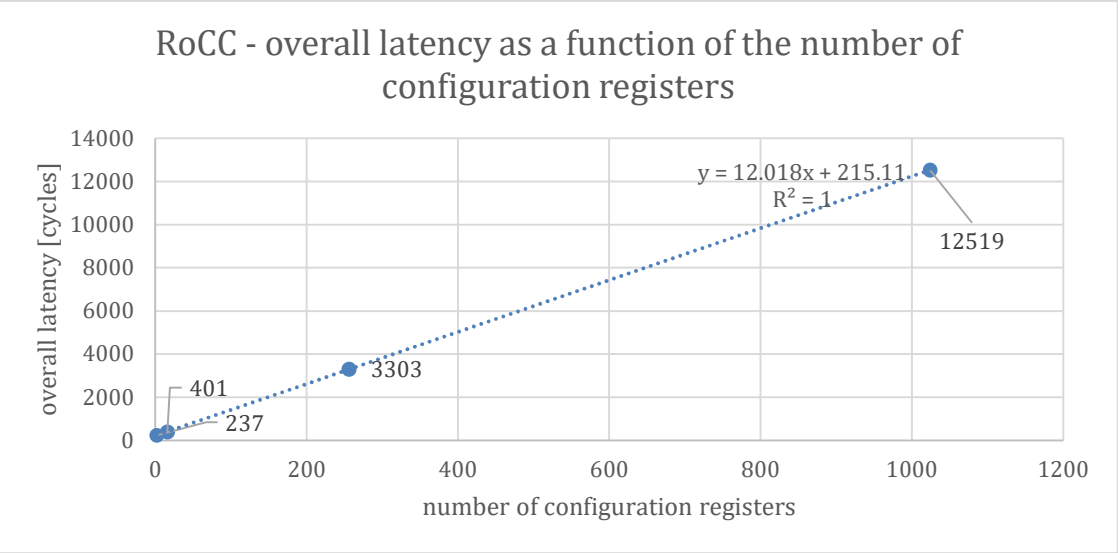

Figure 7 - RoCC result: overall latency as a function of the number of config registers

We may ask ourselves; how did we get the incline and constant of the equation. In order to analyze this, let's look at few samples of the graph:

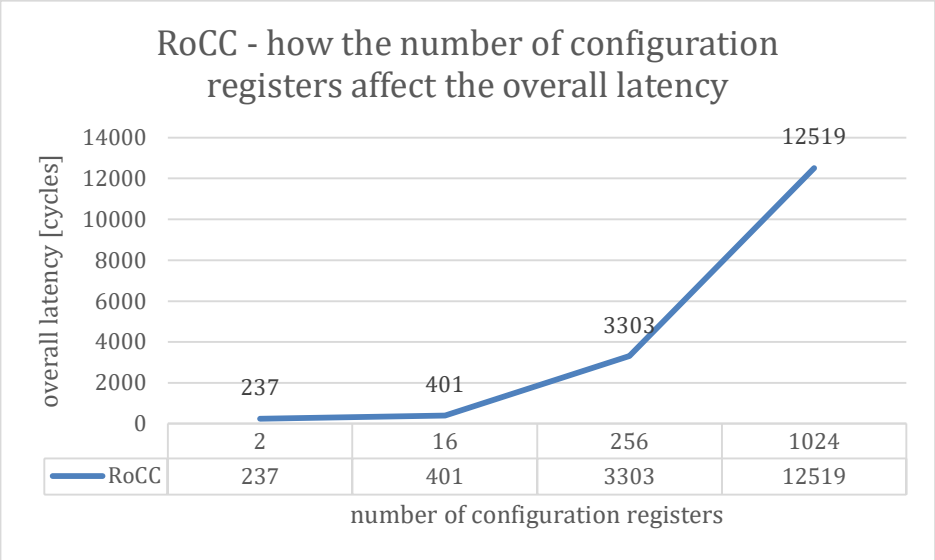

Figure 8 - RoCC, number of config registers effect on overall latency

In Addition to that, let's look at the waveform of a specific test:

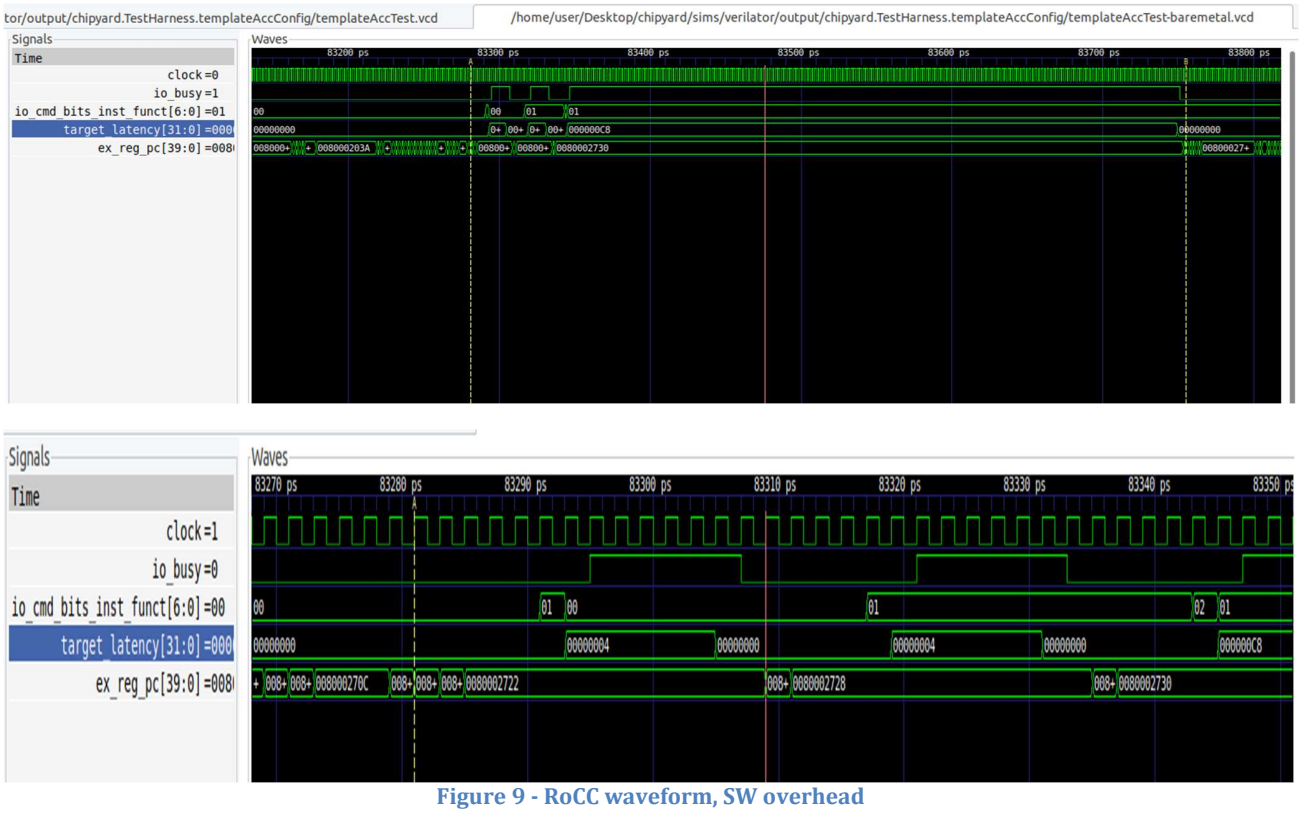

Figure 9 - RoCC waveform, SW overhead

Waveform explanation:

- io_busy - describes the state of the Accelerator Verilog module (io_busy == 1 states that the Accelerator is busy executing a CONFIG/COMPUTE command).
- io_cmd_bits_funct - describes the identifier of the input command (funct == 1 describes a CONFIG command, funct == 2 describes a COMPUTE command).
- PC - Program Counter which describes the current command being executed. Attached a snapshot of the test's assembly (refer to RoCC assembly section).
- Target_latency – an inner signal in our Verilog module, which describes the latency we aim to count to (6 in case of CONFIG command, in case of COMPUTE command, this is the LATENCY parameter that we will discuss later)

As described above, the CONFIG command will take 6 cycles which is the amount if io_busy being toggled. But, let's look at the assembly code, the command with PC=0x80002722, which is the custom ROCC_INSTRUCTION CONFIG command (the command which sends the custom command to the Verilog module while funct = 1) the command takes much more than 6 cycles to be executed. In addition, if we look at the waveform when PC=0x80002728 (which is also a CONFIG command), we can see that the same custom command takes longer time to be executed. Why is that? After research, the software adds additional latency on its own which alters the overall benchmark's latency, hence adding extra latency to each configuration command.

Key observation 1:

We can conclude from the waveform (a detailed example in the next paragraph) that SW adds its own overhead. We will see this phenomenon very frequently in our results, this will play a key Role with the added latency that affects the Accelerator's performance.

Analysis:

In order to analyze the results properly, we want to examine a few points on figure 8 to try and measure the average time of a register configuration command, we shall look at the first 2 sample points in the graph:

The first one states that we have 2 configuration registers and the total number of cycles is 237, the second sample point states that we have 16 registers and a total of 401 cycles of latency. The 401 cycles in the second sampling point includes configuration of 16 registers and a COMPUTE command, while the first one includes 2 configuration commands and one COMPUTE. Hence, approximately, 401-237 = 164[cycles] are the

time that took to configure 14 registers, meaning, each configuration of a register took approximately 164/14 = 11.71[cycles].

If we proceed with the previous notion, and compare different sample points together, we conclude that on average, a configuration command is done in 12 cycles.

We can continue and calculate the average number of cycles associated with the COMPUTE command and get: ~214 cycles.

Conclusion:

We can conclude that the previous equation:

$$y = 12.018x + 215.11$$

Depicts the actual average upon the configuration and compute time, hence we can conclude the incline and constant of the equation!

Key observation 2:

The purpose of the Accelerator is to lighten the load on the processor by efficiently executing specific commands, such as the COMPUTE command in our context. As previously mentioned, each Accelerator has its own array of configuration registers. However, in some instances, the overhead associated with configuring these registers may outweigh the actual benefit gained from executing the COMPUTE command. Consequently, the Accelerator may lose its relevance. Based on our analysis, we determined a threshold for the number of configuration registers:

$$12.018x \leq 215.11 \rightarrow x \leq \frac{215.11}{12.018} = 17.89$$

This indicates that if the Accelerator requires more than 17 configuration registers, the value added by its COMPUTE command will be negated, as the configuration overhead will outweigh the benefits.

We examined the case in which the latency of the special COMPUTE command takes 200 cycles, but different accelerators may have different Latencies for the special custom command, so to generalize this conclusion, we may specify the threshold to be according to the equation:

$$num\ of\ configuration\ registers\ \leq floor\left\lfloor\frac{compute\ time}{12.018}\right\rfloor$$

“Compute constant” is the measured amount of time that takes the Accelerator to execute a single compute command.

Conclusion 2 – Compute Command Latency

We want to address in this section the effect of the Compute command's latency effect on the final results. We argue that its affect is linear with the assigned latency, let us look at some graphs:

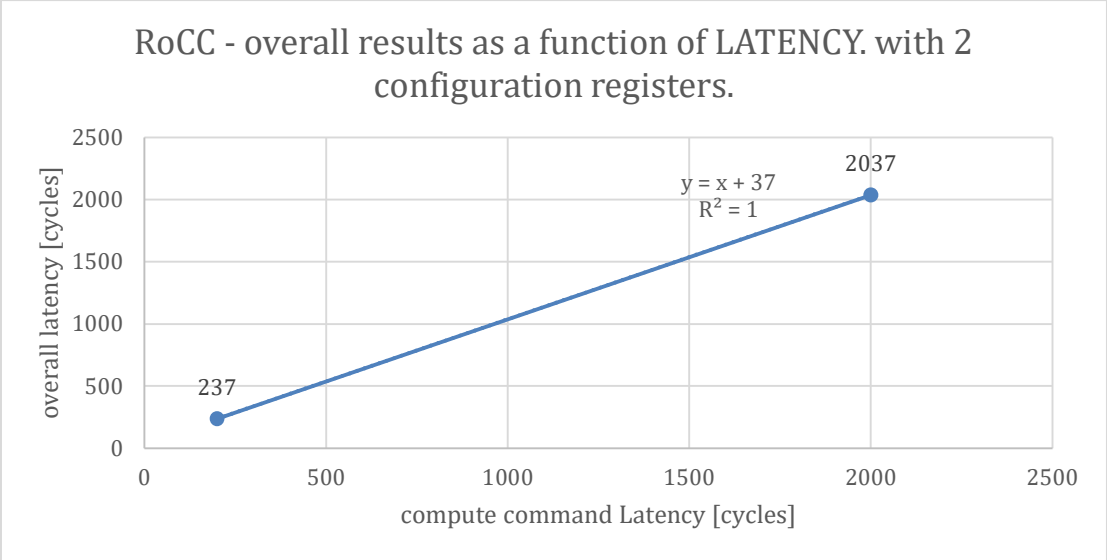

Figure 10 - RoCC, overall result as a function of Compute LATENCY

Note: the benchamrks were performed with 2 configuration registers, address width equals configuration width equals 8 bits.

We have 2 samples presented in the graph:

1. LATENCY = 200
2. LATENCY = 2000

We can see that the plotted equation is:

$$y = x + 37$$

The x qualifier in the equation equals 1 (the incline), which is expected since we have only one COMPUTE command in the Benchmark.

Let's address the constant, we discussed in the previous conclusion the added SW overhead. As previously mentioned, we addressed 2 values of latency: {200, 2000}. In

both benchmarks we received an added latency of 37 cycles, which is actually the SW overhead combined with the configurations of both registers. The configure command takes 6 cycles to complete, so we can conclude that the added SW overhead will be approximately 237 – 200 – 6*2 = 25 cycles on average.

We Shall not try to set a bound on the SW overhead since its unpredictable and related to the system that the benchmark is executed upon!

To summarize the conclusion, the Compute Latency value is effectively determined by both the Accelerator itself and the System-on-Chip (SoC) it is integrated into. When given the option to configure this value, it's reasonable to target minimal latency to enhance the co-processor's efficiency. However, the architecture of the SoC could potentially limit the flexibility to specify desired values. As we saw in our previous conclusions, if the overhead associated with configuring the registers outweighs the efficiency gained from the Compute command, it raises doubts and uncertainties about the necessity of the entire Accelerator.

Conclusion 3 – Config register width and address width relation

We want to discuss in this section the relation between 2 Accelerator's traits:

1. Config register width
2. Address width

Let us examine 2 separate cases:

1. Address width ≤ Config register width
2. Address width > Config register width

regarding the first bullet, a single configuration register can withhold a whole address, hence we do not expect to have additional latency within this case. So far, we have analyzed the results assuming this case.

Addressing the second bullet, the address doesn't fit inside one configuration register and there needs to be some arithmetic operations to be handled (for further information, please refer to the benchmark section in the Introduction and look for – "special Case"). The extra overhead comes to play as some shift register operation, in which we will do a total of:

floor [Address width / Config register width]

With that said, the number of shift operands being performed is related to the number of configuration registers, one should be weary of choosing the appropriate number of configuration registers according to the value of:

floor [Address width / Config register width]

let's demonstrate this, if we choose 2 configuration registers:

In Figure 8: Y axis is the total latency, and x axis is floor [Address width / Config register width]. We chose a COMPUTE latency of 200 cycles.

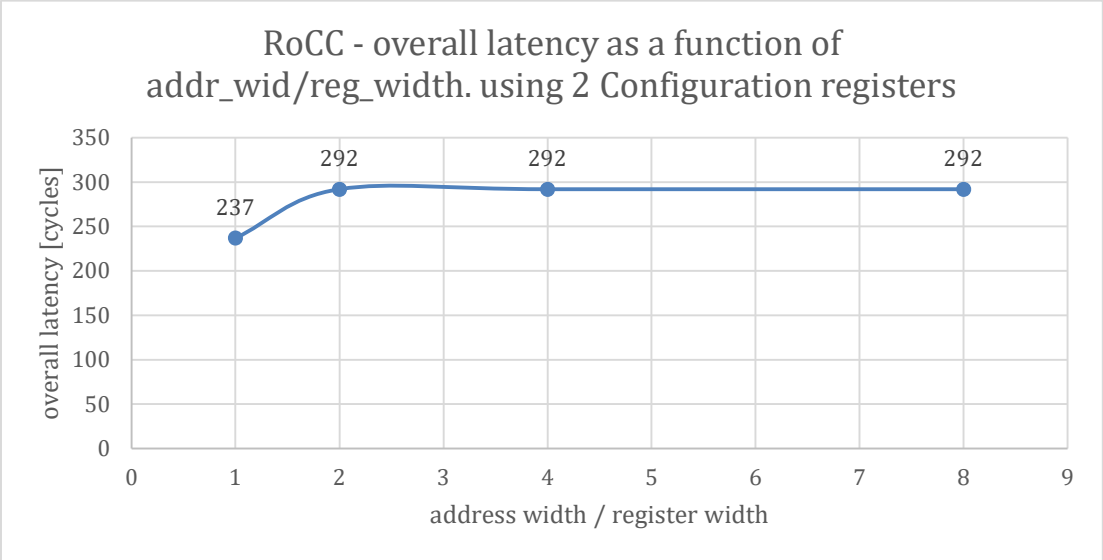

Figure 11 – RoCC results, 2 cfg_regs with addr_width > reg_width

One can notice that once address width ≥ 2 * configuration register width. The graph enters saturation, meaning that there is no longer an effect of address width over the register width. Strengthening the previous notion.

In order to prove our point, we shall examine the case where we have 16 configuration registers and highlight the differences between both cases:

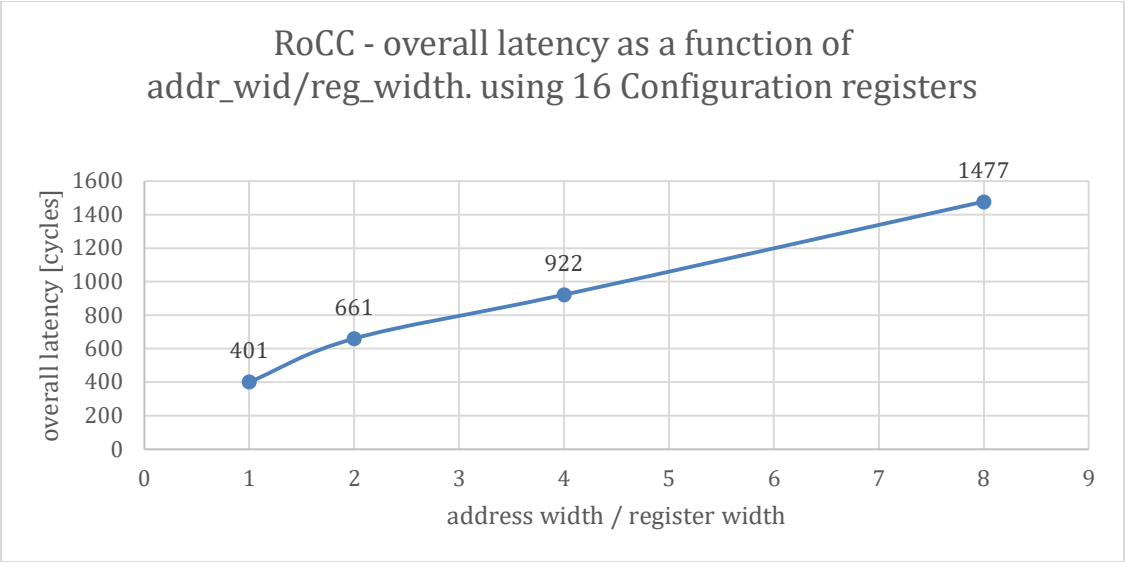

Figure 12 – RoCC results, 16 cfg regs with addr_width > reg_width

From the Figure 11, as expected, we can see that as long as we can perform the shift operands the overall latency still increases until there are no configuration registers to use. In Figure 12 it is noticeable the effect of the shift operands within the assembly code.

```
80002722:   45c1                li      a1,16
80002724:   c402                sw      zero,8(sp)
80002726:   c602                sw      zero,12(sp)
80002728:   47b2                lw      a5,12(sp)
8000272a:   0076f713            andi    a4,a3,7
8000272e:   00e7dc63            bge     a5,a4,80002746 <main+0x5a>
80002732:   47a2                lw      a5,8(sp)
80002734:   4007d79b            sraiw   a5,a5,0x0
80002738:   c43e                sw      a5,8(sp)
8000273a:   47b2                lw      a5,12(sp)
8000273c:   2785                addiw   a5,a5,1
8000273e:   c63e                sw      a5,12(sp)
80002740:   47b2                lw      a5,12(sp)
80002742:   fee7c8e3            blt     a5,a4,80002732 <main+0x46>
```

Figure 13 - shift operand RoCC assembly

In figure 13, we can see the actual added code that performs the shift operands. It is to be noted that, in the cases of register width is smaller (or equal) than the configuration register width, we do not see the attached code section within the assembly generated code. As this is “dead code” – its execution count is 0. The compiler, as part of code optimization, doesn’t generate the specific part of the code into assembly, as it is never executed in the Benchmark.

One can continue and add more configuration registers, but the following conclusions are inevitable:

Conclusion 1:

One should be careful when addressing this relation of configuration register width and address width, if one register can’t be wide enough to accommodate one address then there should be enough configuration registers in order to accommodate the desired number of addresses to be stored!

Conclusion 2:

The added latency of the shift operands enlarges the configuration time overhead, thus minifying the threshold quantity of configuration registers as previously discussed.

RoCC Assembly

The following figure is a snapshot result of objdump upon the binary code of a benchmark. The Benchmark's configuration in the attached snapshot is with 2 configuration registers, Compute latency of 200 cycles and address width equals configuration register width equals 8 bits.

```
00000000800026ec <main>:
    800026ec:   1101                addi    sp,sp,-32
    800026ee:   00000517            auipc   a0,0x0
    800026f2:   0ea50513            addi    a0,a0,234 # 800027d8 <main+0xec>
    800026f6:   ec06                sd      ra,24(sp)
    800026f8:   e822                sd      s0,16(sp)
    800026fa:   e426                sd      s1,8(sp)
    800026fc:   d71ff0ef            jal     ra,8000246c <printf>
    80002700:   00000517            auipc   a0,0x0
    80002704:   10850513            addi    a0,a0,264 # 80002808 <main+0x11c>
    80002708:   d65ff0ef            jal     ra,8000246c <printf>
    8000270c:   4589                li      a1,2
    8000270e:   00000517            auipc   a0,0x0
    80002712:   10a50513            addi    a0,a0,266 # 80002818 <main+0x12c>
    80002716:   d57ff0ef            jal     ra,8000246c <printf>
    8000271a:   4601                li      a2,0
    8000271c:   c0002773            rdcycle a4
    80002720:   4781                li      a5,0
    80002722:   02c7f7db            .4byte  0x2c7f7db
    80002726:   4785                li      a5,1
    80002728:   02c7f7db            .4byte  0x2c7f7db
    8000272c:   c00027f3            rdcycle a5
    80002730:   04c6765b            .4byte  0x4c6765b
    80002734:   0006041b            sext.w  s0,a2
    80002738:   c00024f3            rdcycle s1
    8000273c:   00000517            auipc   a0,0x0
    80002740:   12450513            addi    a0,a0,292 # 80002860 <main+0x174>
    80002744:   8c99                sub     s1,s1,a4
    80002746:   d27ff0ef            jal     ra,8000246c <printf>
    8000274a:   85a2                mv      a1,s0
    8000274c:   00000517            auipc   a0,0x0
    80002750:   13450513            addi    a0,a0,308 # 80002880 <main+0x194>
    80002754:   d19ff0ef            jal     ra,8000246c <printf>
    80002758:   85a6                mv      a1,s1
    8000275a:   00000517            auipc   a0,0x0
    8000275e:   13e50513            addi    a0,a0,318 # 80002898 <main+0x1ac>
    80002762:   d0bff0ef            jal     ra,8000246c <printf>
    80002766:   8622                mv      a2,s0
    80002768:   4599                li      a1,6
    8000276a:   00000517            auipc   a0,0x0
    8000276e:   16e50513            addi    a0,a0,366 # 800028d8 <main+0x1ec>
    80002772:   cfbff0ef            jal     ra,8000246c <printf>
    80002776:   4799                li      a5,6
    80002778:   02f40563            beq     s0,a5,800027a2 <main+0xb6>
    8000277c:   00000517            auipc   a0,0x0
    80002780:   17c50513            addi    a0,a0,380 # 800028f8 <main+0x20c>
```

Figure 14 - RoCC assembly example

MMIO Results

As it was previously mentioned, the MMIO peripheral access differs of that to RoCC in a way that within MMIO communication between the processor and the accelerator happens via memory-mapped registers utilizing the TileLink bus protocol, while in RoCC we can straight access the Accelerator via wires (registers). This main difference comes to hand in the polling query that is executed before and after each MMIO access (refer to section 4.2.2 “MMIO Benchmarks”). We will present its impact upon the performance of the processor. It shall be noted that the parameters/traits that we explored are the same ones that we examined in RoCC.

It is to be noted that the SW overhead that came along the RoCC analysis, is also noted within the MMIO analysis.

Note: the following conclusions address the assembly code of a benchmark, please refer to section “MMIO Assembly” the provided code.

Note: in order to understand the waveform in a better manner, refer to the documentation in the next page.

Important Note: In order to complete the handshake between the MMIO peripheral and processor, the processor need to toggle the output_ready (to indicate to the peripheral is ready to be read) and the Accelerator needs to toggle io_resp_valid, which is the indicator that the peripheral has finished its execution and ready to output the result, on the memory interface. And vice-versa.

Waveform example explanation

in our conclusions, we will address the waveform multiple times, for they will demonstrate some important principles. The following pages aim to explain the purpose of each signal presented in the waveform.

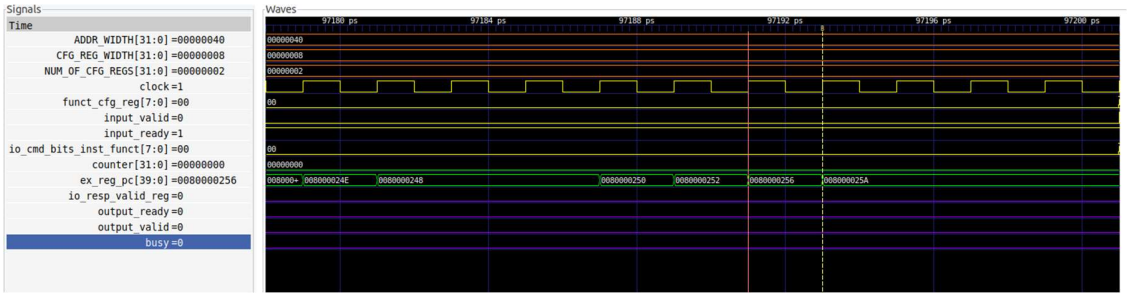

Figure 15 - MMIO waveform, compute command polling

The previous waveform presents a compute command polling, explanation of the signals as seen in the waveform (divided into color groups):

- Orange:
  describes the benchmark's accelerator's configuration – ADDR_WIDTH, CFG_REG_WIDTH, NUM_OF_CFG_REGS.
- Yellow:
  clock – clock signal
  funct_cfg_reg – specifying the desired command from the processor to Accelerator.
  input_valid, input_ready – implementing the master slave protocol between the Accelerator and processor.
- Green:
  counter – an internal counter in the Accelerator's Verilog implementation that counts until the desired latency (according to the specified command / funct). PC – Program Counter, this signal is important because it specifies which assembly command is currently being executed. This will help us monitor the Benchmark's flow.
- Violet:
  io_busy_reg – an indication if the accelerator is busy (Boolean value).
  io_resp_valid_reg – specifies the response status from the accelerator.
  output_ready – indicates if the processor is ready to receive the calculation result in the MMIO peripheral. (this is an input to our module)

As we can see form the previous waveform (between the 2 markers), PC is at 0x80000256 which specifies a memory write command to funct register (as one can see

the assembly instruction in section “MMIO Assembly”). Let’s look at the waveform on a wider time scale:

Figure 16 - MMIO waveform, handshake

We can see (between the 2 markers) that funct arises to 1, which indicates that the following is a CONFIG command. While also the input_valid asserts in the same exact cycle indicating that the input values are valid. In order to complete the handshake and the Accelerator can receive the data according to the master-slave protocol, we need to verify that the Accelerator’s output signal “input_ready” is toggled once the input_valid asserts. Which what actually happens.

When that happens, the accelerator can start its execution and the busy signal arises one cycle later indication that the Accelerator is executing a command.

Afterwards, PC is at 0x8000025A which specifies a read from the memory (start of the polling to check if the peripheral has finished its calculations).

As mentioned earlier: In order to complete the handshake between the MMIO peripheral and processor, the processor need to toggle the output_ready (to indicate to the peripheral is ready to be read) and the Accelerator needs to toggle io_resp_valid, which is the indicator that the peripheral has finished its execution and ready to output the result, on the memory interface.

So, one can see in the waveform that the counter signal has reached the value of 4 (the desired latency of a CONFIG command), meaning that the Accelerator has finished its execution, but the memory system is still not ready to receive the data and complete the handshake, so the peripheral holds the results until the signal output_ready toggles, indicating that the handshake can be completed. As expected, we see that the busy signal de-asserts one cycle later, indicating that the Accelerator is ready to receive more commands.

Now, we can dive into the conclusions:

Polling

In this section we shall explore the effect of MMIO memory polling upon the processor's performance and total latency. In the following section, we will present a few waveform examples that embody the effect of the polling process:

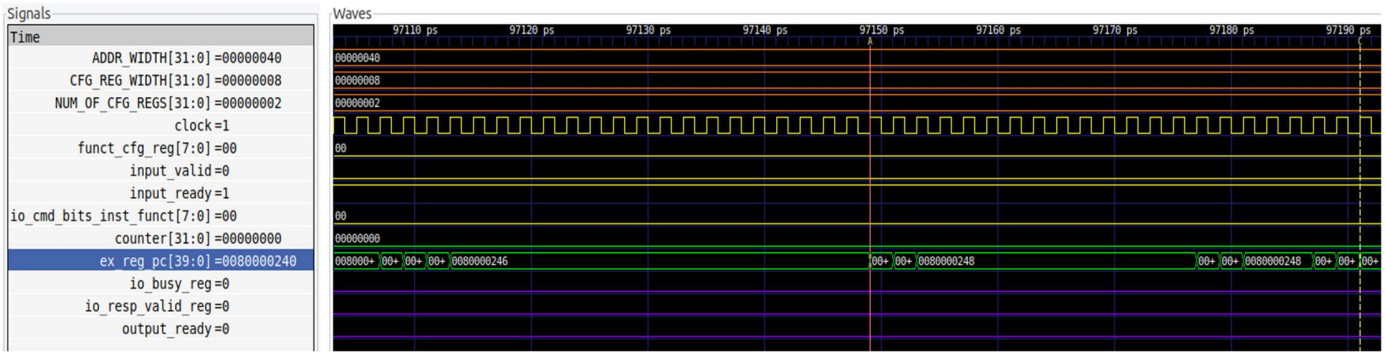

Figure 17 - MMIO waveform, polling example 1

In between the two markers, we can see the first polling – which belongs to PC 0x80000248, it describes the first polling of the status registers, i.e. we wait for the peripheral to be ready, for we poll to the value of 0x2 (i.e. input_ready = 1 & output_valid = 0).

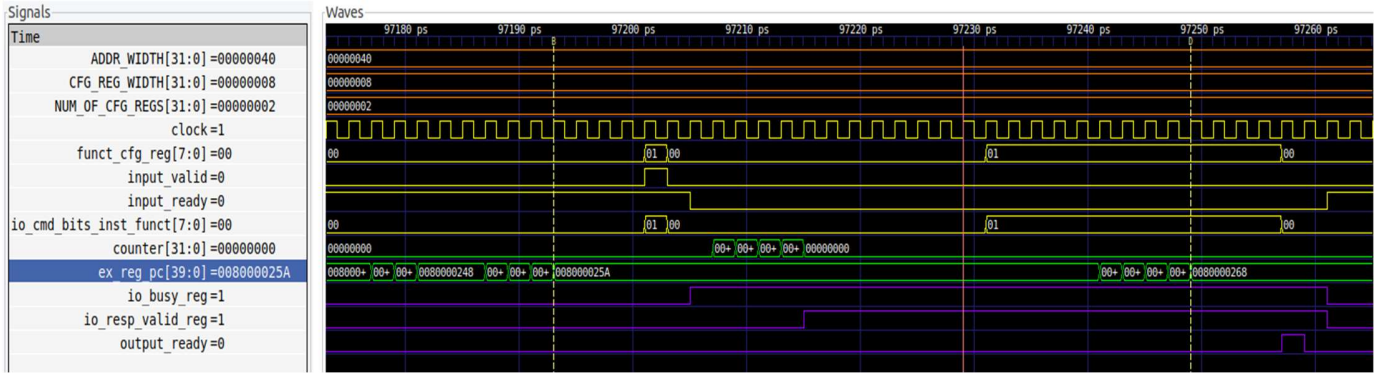

Figure 18 - MMIO waveform, polling example 2

In the previous snapshot, we can view the second polling, in PC 0x8000025A, in which there is a poll to check whether the MMIO has finished its calculation (after a specific write command) – the poll is upon the desired value of 0x1 (i.e. input_ready = 0 & output_valid = 1).

The previous snapshots and comments demonstrate how the effect of the polling comes to display augmenting the overall latency of the benchmark, as we did not have this polling within the RoCC territory, this will be a key difference between both Accelerators.

Important observation:

An important question arises: why does the first polling take a larger amount of time to complete in relation to the second polling? (Figure 17)

The answer to that question relies upon the TLB - Translation Lookaside Buffer. TLB is a cache that stores recent translations of virtual memory addresses to physical memory addresses. The purpose of TLB is to improve the efficiency of memory access in systems that use virtual memory.

The first access to memory typically takes longer because it often results in a TLB miss. The TLB needs to be populated with translations before subsequent accesses can benefit from TLB hits. Once a translation is retrieved from the page tables and stored in the TLB, subsequent accesses to the same virtual memory address can be faster because the translation is readily available in the TLB. Hence, in subsequent accesses to the same virtual memory address, no additional translation penalty will be granted.

Conclusion 1 – number of configuration registers

By considering the previous Polling conclusions and the SW overhead that we discussed in RoCC, we may come to similar conclusions in MMIO as we had in RoCC regarding the effect of each Accelerator parameter, let us present a couple of graphs:

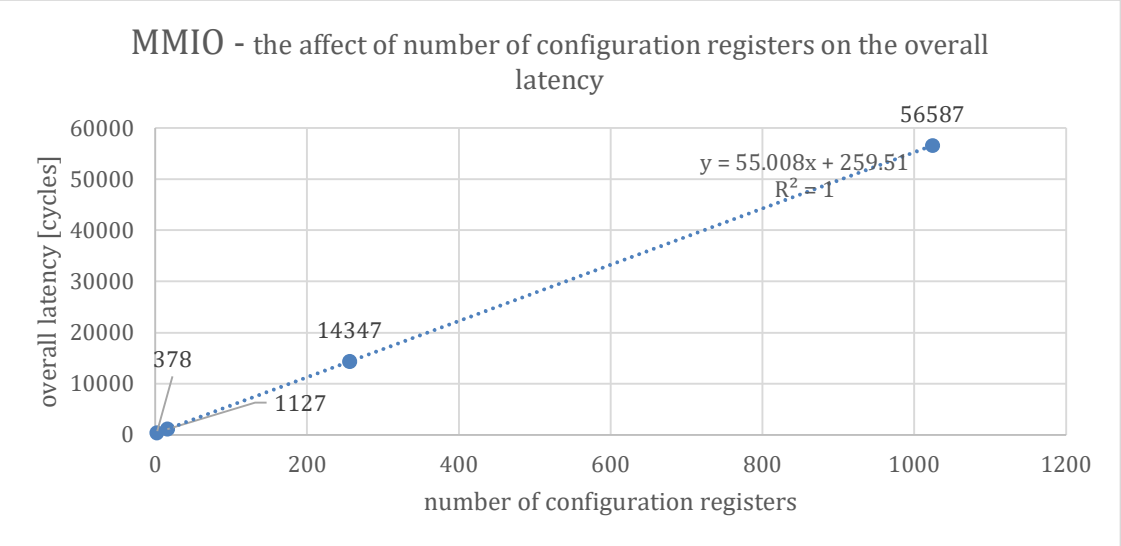

Figure 19 – MMIO result: number of config registers

Note: in the previous graph, the value of the latency parameter is 200. And the ADDR_WIDTH trait is smaller than the CFG_REG_WIDTH.

In order to understand the equation, its incline and constant:

$$y = 55.008x + 259.51$$

we shall perform the same analysis we did in RoCC.

Let's look at a few samples in the graph:

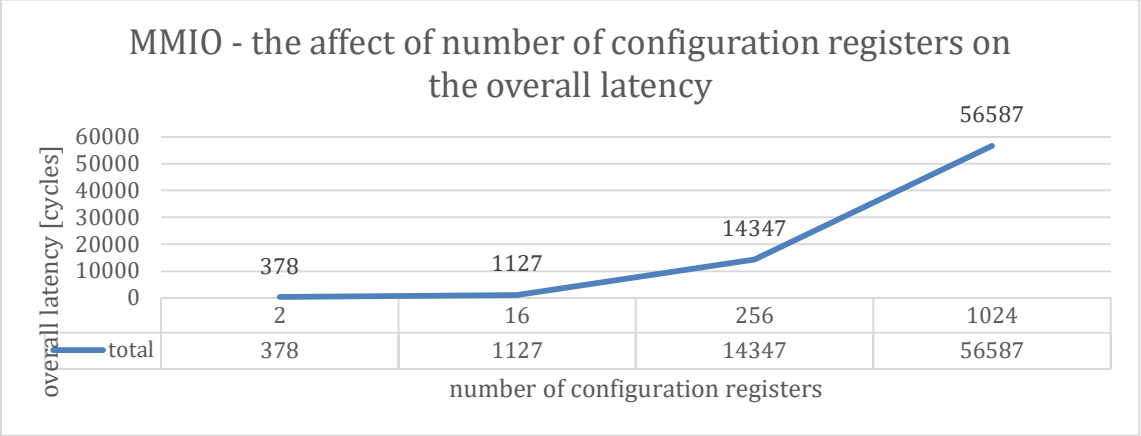

Figure 20 - MMIO results, cfg regs samples

The attached graph represents the relation between the number of configuration register as opposed to the total latency. We can observe that by increasing the number of configuration registers, the overall latency increases drastically. In order to get a better grip on the results, let's look at the first 2 samples in the graph.

The first one states that we have 2 configuration registers and the total number of cycles is 378, the second sample point states that we have 16 registers and a total of 1127 cycles of latency. The 1127 cycles in the second sampling point includes configuration of 16 registers and a COMPUTE command, while the first one includes 2 configuration commands and one COMPUTE. Hence, approximately, 1127-378 = 749[cycles] are the time that took to configure 14 registers, meaning, each configuration of a register took approximately 749/14 = 53.5[cycles].

If we continue with the same attitude and compare different points in the graph, we can conclude that the average time to configure a configuration register is approximately 54.5~55 cycles.

We can continue with this notion to conclude that the average time that took a COMPUTE command to execute is 260 cycles.

The previous results match the equation we received in Figure 19!

We shall compare the RoCC graph with the MMIO graph:

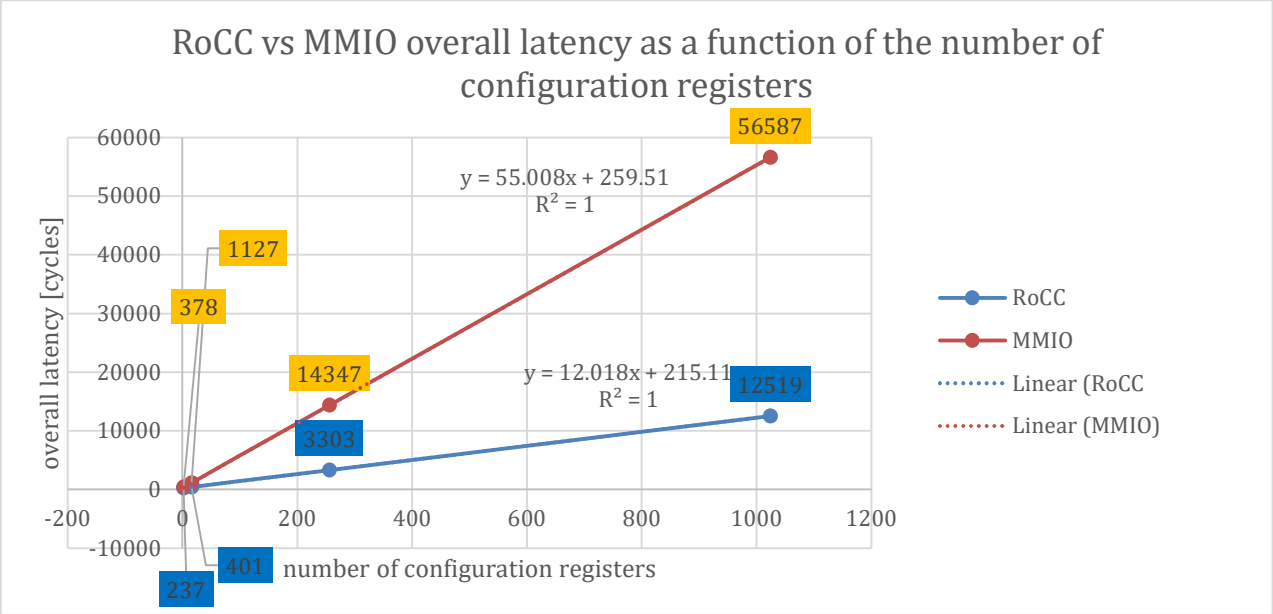

Figure 21 - RoCC vs MMIO overall latency as a function of cfg regs count

Note: the attached graph presents benchmarks under the assumption that the Compute Command Latency is 200 cycles, the configuration register width equals address width which is 8 bits.

In order to compare the results properly, lets analyze the differences between the two command types that we support and their latency:

1. CONFIG:
   RoCC took on average 12.018 cycles to execute a single configuration command while MMIO peripheral took 55.008 cycles.
   Configure time multiplied by a factor of 4.577!
2. COMPUTE:
   RoCC took on average 215 cycles to execute a single COMPUTE command while MMIO peripheral took 260 cycles. An overhead of additional 45 cycles (on average).

this may be attributed to a couple of factors:

1. SW overhead – as we discussed previously, SW adds additional latency to the benchmark, of which we can't bound or determine certainly. So, its plausible to assume that the SW had a role to play within the rule of the added overall latency.

2. Polling – this is without a doubt the main factor here which resulted in the significant overhead within MMIO. As we saw in the first conclusion, each polling takes a heavy amount of clock cycles to be performed, and on average, each command has 2 polling's in total, when each polling performs 2 memory accesses / reads on average! (a polling before we send the command on the memory registers – to check if the peripheral is ready to execute the command, and another one afterwards – if the peripheral has finished the execution. It is difficult to output an equation which describes the behavior of the Benchmark in regard to the parameters being checked, for we can't predict how many cycles it would take the polling to finish.

As we have discussed in RoCC, one may also try to set a threshold upon the maximal number of configuration registers to be used in the peripheral to avoid the case where the configuration time exceeds the compute command latency, disregarding the accelerator's added value:

As mentioned, the average cycles of configuration command: 55.008 cycles. Compute command takes on average 259.51 cycles to complete. Thus:

$$55.008x \leq 259.51 \rightarrow x \leq 4.717$$

Meaning, if one chose a custom command with a latency of 200 cycles, one may have up to 4 configuration registers! Compared to RoCC, where the threshold was set to 17 registers, this has a huge impact!

Conclusion 2 – Compute Command Latency

We shall address the Compute command LATENCY variable effect on the overall benchmark's result. As one can see in the results csv file attached in our project's github, the benchmark's overall latency rises in direct proportion to any increase in the latency variable. Which includes within it the polling and SW overheads.

Let's examine 2 sample points:

ADDR_WIDTH	CFG_REG_WIDTH	NUM_OF_CFG_REGS	LATENCY	Result [cycles]
8	8	2	200	378
8	8	2	2000	2184

If we consider the latency as the x axis, and the overall number of cycles as y axis, one can output the following equation:

$$y = 1.003x + 177.4$$

The factor of 1 (approximately) within the x axis is due to that we have only 1 compute command. Now, let's address the constant, which is 177.4.

If we combine the SW overhead explanation that was discussed in RoCC, add to it the polling overhead we presented in MMIO, we can conclude that the value being presented by the constant 177.4 is a combination between SW & polling overhead combined with the configuration of 2 registers within the Benchmark.

We examined the same accelerator configuration parameters both in MMIO and RoCC, but the effect of the polling overhead that was added in MMIO is very noticeable! 177.4 cycles in comparison to RoCC's 37 cycles.

Note:

Once again, we chose 2 sample points from the results, but we can come to the same conclusions if we pick any 2 points within the results that have the same configuration {same number of configuration registers and their width, the address width. Just with different latency values: 200 as opposed to 2000}

We Shall compare RoCC with MMIO within a graph:

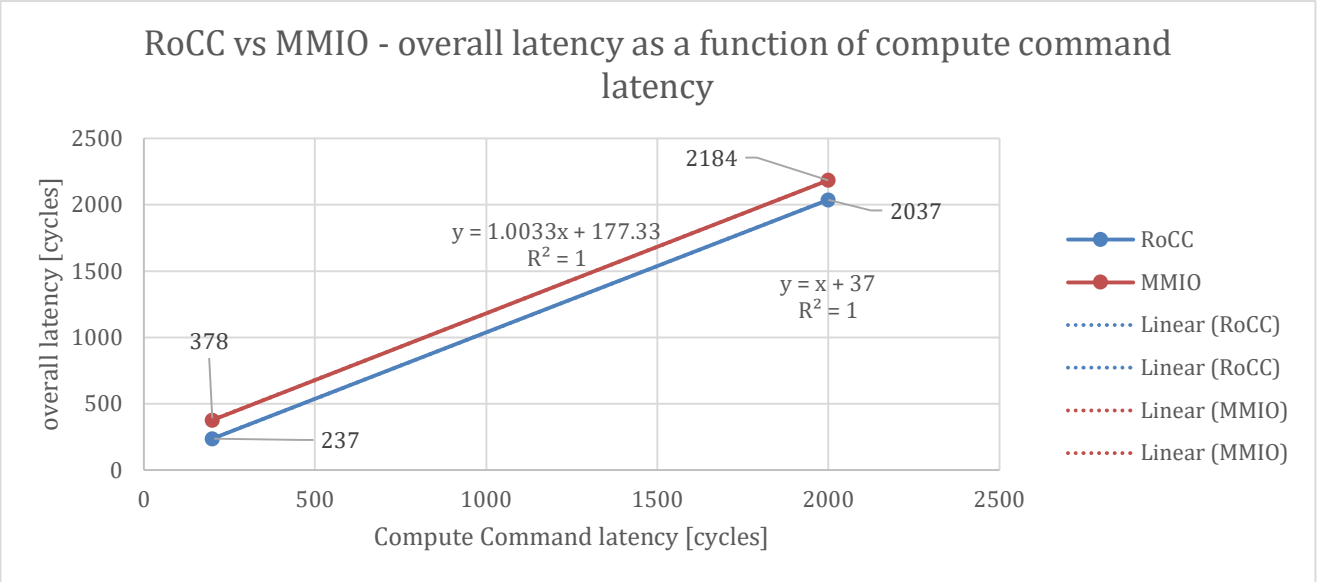

Figure 22 - RoCC vs MMIO overall latency as a function of compute command latency

Note: the attached graph presents benchmarks under the assumption that the Latency of Compute command is a running variable on 200, 2000 cycles. 2 configuration registers with a configuration register width equals address width equals 8 bits.

One can see the polling overhead effect more clearly here, since as we discussed in RoCC, the constant value in the linear equations is attributed to the SW overhead. But in MMIO, it is a combination of the SW and polling overhead. It is important to note that the Benchmarks were performed with regard to the same configurations, so we can isolate and attribute the additional overhead to the polling process.

Conclusion 3 – Config register width and address width relation

In this section we will discuss the relation between two Accelerator's traits:

1. Address width > Config register width
2. Address width ≤ Config register width

We argue that the effect of these changes is identical to what we saw in RoCC. The interesting case to examine here is the first one, since then we would need to perform the shift operands mentioned in the Benchmark section under "special case". As previously mentioned, if the address width is greater than the configuration register width, then we would need multiple registers to accommodate one address. Thus, the overhead of this operation is displayed only in SW and has no relation to the Accelerator type. Meaning, the polling overhead we repeatedly saw in MMIO, should not be a factor here.

In the following graph, we presented in x axis the value of the division:

Address_width / configuration register width

Y axis is the overall latency. The Compute latency is 200 cycles.

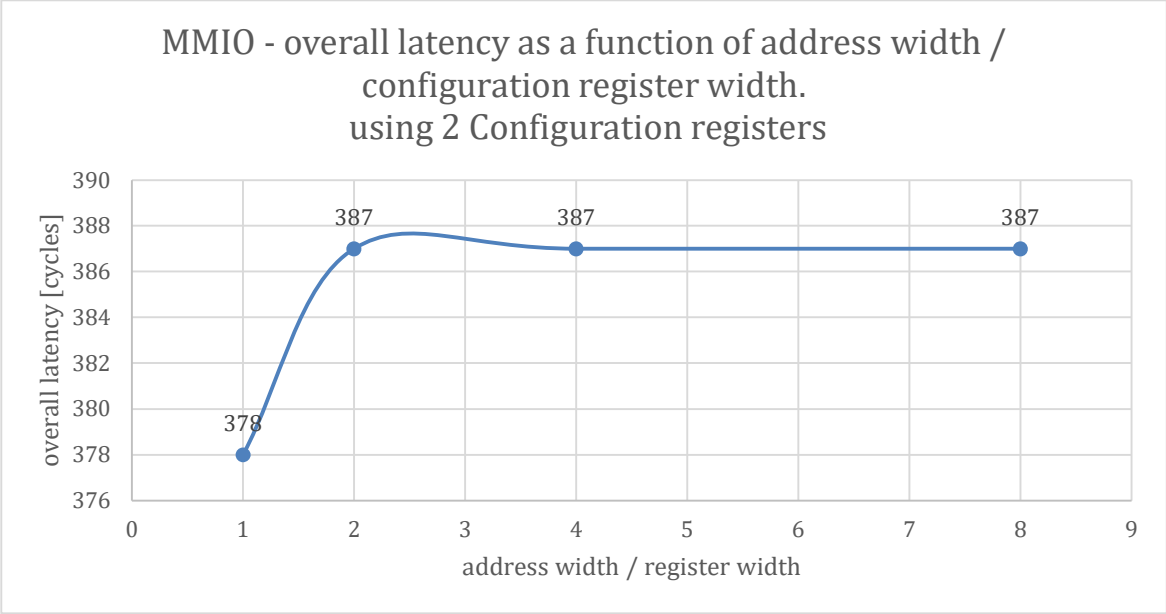

Figure 23 - MMIO results, overall latency as a function of addr_wid/reg_wid, using 2 config regs

Note: the benchmarks were executed with Compute latency of 200 cycles.

As discussed in RoCC's third conclusion, if we do not have enough registers to accommodate one address we will see the above saturation behavior of the graph, since the number of shift operand is limited. Meaning, in each of the benchmarks where we have 2 configuration registers and the address width / register width is greater than 2, we shall get the same result.

similar to RoCC, we will present a graph with 16 configuration registers (again with Compute latency of 200 cycles):

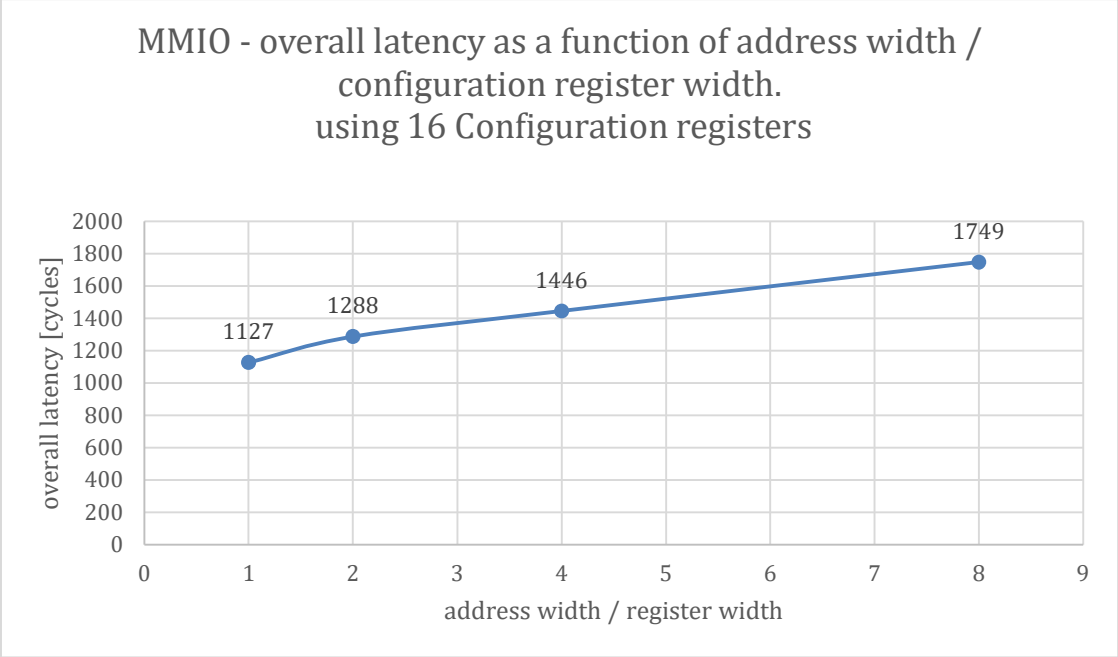

Figure 24 - MMIO results, overall latency as a function of addr_wid/reg_wid, using 16 config regs

Note: the benchmarks were executed with Compute command latency of 200 cycles.

We can see from the graph that the saturation behavior we saw with 2 configuration registers disappears, and we come to the same conclusions we had in RoCC.

In Order to strengthen our notion that the added latency as resulting from address width being greater than register width, has the same effect in RoCC and MMIO, we shall compare the RoCC and MMIO results:

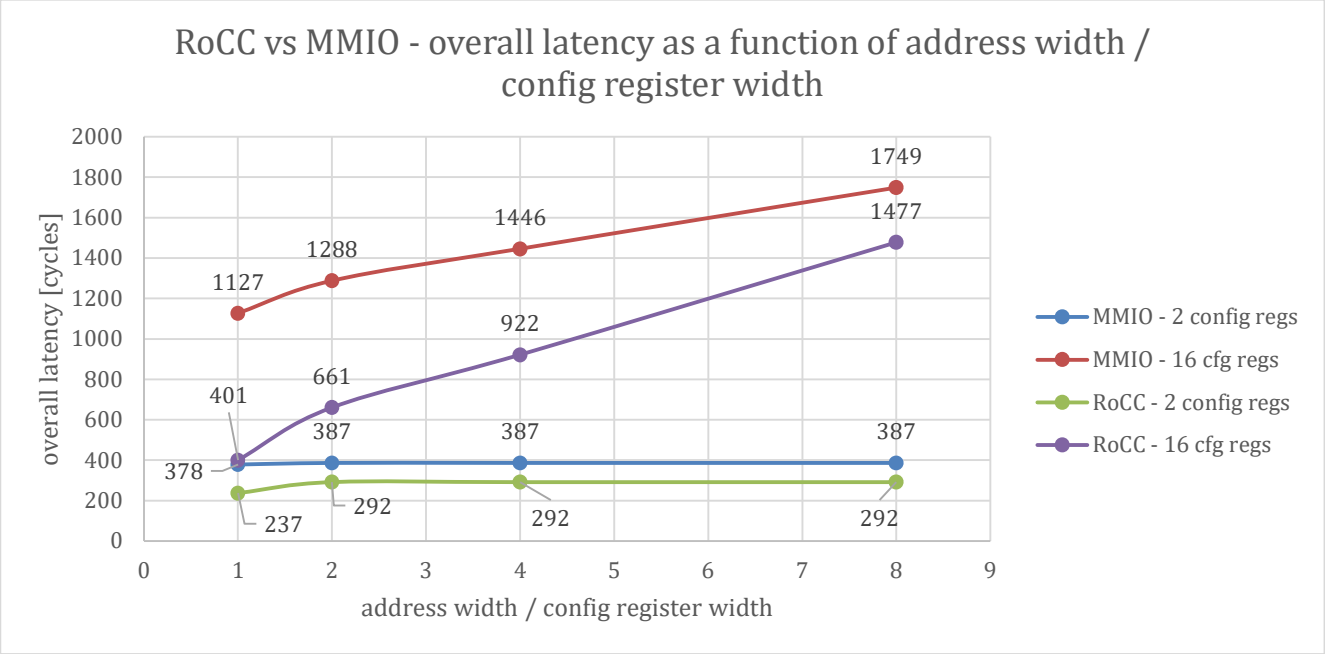

Figure 25 - RoCC vs MMIO, overall latency as a function of addr_wid/reg_wid

Note: the benchmarks were executed with Compute latency of 200 cycles.

It is clear that in efficiency, MMIO is inferior RoCC also in this category. This can be attributed to the polling procedure.

MMIO Assembly

The following is a part of a specific MMIO benchmark within the main() function, one can view the full assembly code within the project's github. Also, one can generate a desired assembly via the instructions presented in section 4.5.

```
80000240:   c00025f3            rdcycle    a1
80000244:   00050737            lui    a4,0x50
80000248:   00074783            lbu    a5,0(a4) # 50000
<__heap_size+0x30000>
8000024c:   8b89                    andi   a5,a5,2
8000024e:   dfed                    beqz   a5,80000248 <main+0x1c>
80000250:   4785                    li a5,1
80000252:   00f70223            sb a5,4(a4)
80000256:   00050737            lui    a4,0x50
8000025a:   00074783            lbu    a5,0(a4) # 50000
<__heap_size+0x30000>
8000025e:   8b85                    andi   a5,a5,1
80000260:   dfed                    beqz   a5,8000025a <main+0x2e>
80000262:   471c                    lw a5,8(a4)
80000264:   00050737            lui    a4,0x50
80000268:   00074783            lbu    a5,0(a4) # 50000
<__heap_size+0x30000>
8000026c:   8b89                    andi   a5,a5,2
8000026e:   dfed                    beqz   a5,80000268 <main+0x3c>
80000270:   4785                    li a5,1
80000272:   00f70223            sb a5,4(a4)
80000276:   00050737            lui    a4,0x50
8000027a:   00074783            lbu    a5,0(a4) # 50000
<__heap_size+0x30000>
8000027e:   8b85                    andi   a5,a5,1
80000280:   dfed                    beqz   a5,8000027a <main+0x4e>
80000282:   471c                    lw a5,8(a4)
80000284:   00050737            lui    a4,0x50
80000288:   00074783            lbu    a5,0(a4) # 50000
<__heap_size+0x30000>
8000028c:   8b89                    andi   a5,a5,2
8000028e:   dfed                    beqz   a5,80000288 <main+0x5c>
80000290:   4789                    li a5,2
80000292:   00f70223            sb a5,4(a4)
80000296:   00050737            lui    a4,0x50
8000029a:   00074783            lbu    a5,0(a4) # 50000
<__heap_size+0x30000>
8000029e:   8b85                    andi   a5,a5,1
800002a0:   dfed                    beqz   a5,8000029a <main+0x6e>
800002a2:   4700                    lw s0,8(a4)
800002a4:   c0002673            rdcycle    a2
```

Figure 26 - MMIO assembly

Summary

It's evident that the RoCC accelerator outperforms the MMIO Accelerator in terms of efficiency, which is plausible considering the location of each Accelerator on the SoC. The RoCC completed the benchmarks remarkably faster than the MMIO Peripheral did. This raises a crucial question: what is the benefit of using the MMIO peripheral instead of RoCC when, upon analysis, RoCC consistently yields superior outcomes?

Resource Sharing:

Since **RoCC** co-processors are tied to a single core or hardware thread and require support with Core interface implementation, hence restricting their scope of application and potentially leading to underutilization of resources in multicore environments. On the other hand, **MMIO** peripherals do not require an interface implementation with the Core but rather communicate with the Core via memory-mapped registers, utilizing TileLink bus protocol. This level of decoupling enables the versatility of integrating and distributing these accelerators across a range of multi-core System-on-Chip (SoC) configurations.

Exception & interrupt handling:

RoCC exception is passed via a single exception bit within its interface with the Core. Limiting the information being passed by the co-processor to the Core. In addition to that, the co-processor needs to handle its own interrupts and exceptions, While in **MMIO** the memory system handles them.

Thus, we conclude that there is no superior Accelerator implementation which can fit all implementations. The decision of which type of Accelerator to use relies on the architectural design one chooses to implement. For there is no optimal Accelerator which can satisfy all requirements, unanimity remains unattainable.

Future work

One can continue and extend this project, include more improvements:

Verilog implementation:

- Pipe Lined Accelerator – implement the RTL accelerator to be pipelined so it would improve its throughput and decrease the total latency. Pipeline in general is a very important concept in the industry which is known to improve performance significantly.
- Use further Accelerator traits defined in the Introduction, that we did not explore.
- In our project we assumed that all of the configuration registers are of the same width, but in reality, this is not true and they may vary differently, so this can be another factor to be checked.
- Add support to interrupt and exception in RoCC accelerator. As previously discussed, RoCC co-processor needs to handle its own interruptus and exception. As opposing to MMIO where the memory system handles these cases. In our Verilog module the exception and interrupt signals are tied to 0. But one may add support to handling interrupts and exceptions.
- Support a larger set of instructions – in the current project we supported only 2 types of instructions: Compute and Config. But one may also support instructions that access memory system for example (Load or Store).

It is to be mentioned, albeit that the previous improvements are mentioned under Verilog improvements, each one of them requires modification within the benchmarks and Scala files in order to accommodate to it, and achieve desired results.

References

[1] Generator-Based Design of Custom Systems-on-Chip for Numerical Data Analysis by Alon Amid (December 1, 2022) https://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-247.pdf

[2] Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration by Alon Amid https://alonamid.github.io/papers/gemmini-dac2021-preprint.pdf

[3] Riscv manual for custom instructions https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

Open-source platforms

Open-source platforms

I. Chipyard's cookbook: https://chipyard.readthedocs.io/en/stable/index.html
II. Chipyard's hierarchy: https://github.com/ucb-bar/chipyard
III. GCD example (MMIO) ; https://github.com/ucb-bar/chipyard/blob/1.6.0/generators/chipyard/src/main/scala/example/GCD.scala
IV. Sha3 Accelerator (RoCC): https://eur01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fgithub.com%2Fucb-bar%2Fsha3&data=05%7C01%7Cghasan.sha%40campus.technion.ac.il%7Ca556a127a755404bb56508db3b53f593%7Cf1502c4cee2e411c9715c855f6753b84%7C1%7C0%7C638169004048020974%7CUnknown%7CTWFpbGZsb3d8eyJWIjoiMC4wLjAwMDAiLCJQIjoiV2luMzIiLCJBTiI6Ik1haWwiLCJXVCI6Mn0%3D%7C3000%7C%7C%7C&sdata=mU9xu6KrEYxdhkmuf82x%2BpxUxFkqCS8YNU0F2FjGGAk%3D&reserved=0
V. More top level file examples will be found in the project's github.

Here are some top level files:
Matrix Multiplication Accelerator (RoCC): https://github.com/ucb-bar/gemmini

- Top level: https://github.com/ucb-bar/gemmini/blob/master/src/main/scala/gemmini/Controller.scala

- Command fields/encodings: https://github.com/ucb-bar/gemmini/blob/master/src/main/scala/gemmini/GemminiISA.scala

Deep Learning Accelerator (MMIO): https://github.com/nvdla/hw

- Top level: https://github.com/nvdla/hw/blob/nvdlav1/vmod/nvdla/top/NV_nvdla.v
- Integrators manual: http://nvdla.org/hw/v1/integration_guide.html

SHA3 Accelerator (RoCC): https://github.com/ucb-bar/sha3

- Top level: https://github.com/ucb-bar/sha3/blob/master/src/main/scala/sha3.scala

Protocol Buffer Accelerator (RoCC): https://github.com/ucb-bar/protoacc

- Top level: https://github.com/ucb-bar/protoacc/blob/master/src/main/scala/protoacc.scala

Compression Accelerators (RoCC): https://github.com/sagark/compress-acc-ae

- Top level 1: https://github.com/sagark/compress-acc-ae/blob/main/src/main/scala/ZstdCompressor.scala
- Top level 2: https://github.com/sagark/compress-acc-ae/blob/main/src/main/scala/SnappyCompressor.scala
- Top level 3: https://github.com/sagark/compress-acc-ae/blob/main/src/main/scala/HufCompressor.scala
- Top level 4: https://github.com/sagark/compress-acc-ae/blob/main/src/main/scala/FSECompressor.scala

Compression Accelerator (Memory Mapped?): https://github.com/opencomputeproject/Project-Zipline

- Top level: https://github.com/opencomputeproject/Project-Zipline/blob/60657e17ac9579aaf3751f0b5c0c0710946b0aa9/rtl/cr_cceip_64/cr_cceip_64.v
- Example Control Register Enumerations: https://github.com/opencomputeproject/Project-Zipline/blob/60657e17ac9579aaf3751f0b5c0c0710946b0aa9/rtl/cr_cceip_64/cr_cceip_64_regs.vh
- Control registers documentation: https://github.com/opencomputeproject/Project-Zipline/tree/60657e17ac9579aaf3751f0b5c0c0710946b0aa9/register_doc

project github link

Provide!

In MMIO we read the values of the registers via memory Map, in a reak MMIO accelerator the values of those registers are pointers to memory addresses which we can load or store from.

MMIO waves: Top/chiptop/system/tile_prci_domain/tile_reset_domain_tile