AXI connected hardware accelerator in a RISC-V Processor

Progress report

Callum Stew

Department of Computer Science

University of Warwick

1 Introduction

In recent years processors have been running into limits in single-threaded processing speed [12]. Overall performance has been increased by taking advantage of parallelism. This can be achieved using multiple cores to gain general parallelism or through hardware accelerators specifically designed to exploit the parallelism inherent in an algorithm. An example would be multiplying matrices. Many matrix operations involve computing many simpler calculations on the components of the matrix. These calculations are often independent and can be performed in parallel. A processor would compute these sequentially but a specific hardware implementation would be able to compute them in parallel.

As this project will use a hardware implementation, a platform is needed that can allow fast development and iterations on hardware designs. An FPGA (Field Programmable Gate Array) is a semiconductor device based around a matrix of configurable logic blocks [14]. The user can define what logic function these blocks perform and how they are connected using HDL (Hardware Description Language) programming. Hardware can be designed in HDL and constructed on the FPGA without needing to manually build the circuit.

For this project, a processor is needed to integrate the matrix accelerator with. RISC-V is an open standard instruction set architecture that can be implemented in a variety of devices such as high-performance boards by SiFive and smaller microcontrollers such as Espressif's ESP32-C3. This adoption means that more software is available for the RISC-V instruction such as Debian's risk64 port. To implement a RISC-V core on an FPGA the Rocket Chip Generator will be used. This is an open-source SoC generator that produces synthesizable RTL for implementation on an FPGA. The FPGA being used in this project should be able to run one rocket64b core. Rocket Chip uses the Chisel HDL based on the Scala programming language. This allows for modern programming features to be used and then generate Verilog from this code. The Verilog code can then be synthesised for an FPGA.

1.1 Objectives

Develop a hardware accelerator for a RISC-V processor communicating over the AXI bus and compare it against a bare-metal software solution.

- Generate a RISC-V core on an FPGA and run bare-metal code on it (Must).
- Design an IP block that connects to the RISC-V's AXI bus that can send and receive data (Must).
- Design an IP block that performs the hardware-accelerated functionality (Must).
- Develop a driver in bare-metal C code to use the accelerator (Must).
- Design and implement a test for the accelerator (Must).
- Implement the test using a software approach and compare results (Should).
- Implement a hardware accelerator using an alternative method of communication such as an APB bus and compare (Could)
- Run Linux on the processor and make the accelerator available for use (Won't)

1.2 Related work

1.2.1 Implementation of a RISC-V Processor with Hardware Accelerator [7]

This paper covers the development of a hardware matrix multiplier for a RISC-V processer. They used a ZedBoard which uses the same Artix-7 FPGA chip as the Nexys A7 100T that this project uses. They use PULPino [10], a 32-bit single-core RISC-V microprocessor. This project does not support the Nexys A7 100T FPGA used in our project so vivado-risk-v [11] is used to generate a 64-bit RISC-V core using the Rocket Chip project [5]. This takes more of the resources available to the FPGA but can be reduced to 32-bit if needed as we will not be running Linux. They used the APB bus to communicate between the accelerator and PULPino microprocessor due to its interface being simpler than the AXI bus. In our project, the AXI interface will be used as it recommended by the base project and provides the option for higher-performance transfers. When tested their accelerator reduced the number of clock cycles required from 603 to 134. This is a good improvement but due to the processor having a clock speed of 5 MHz, it is unable to compete with modern processors. They encountered many errors when trying to build C programs for the PULPino and required much trial and error. The process for compiling code for our project has proved much simpler.

1.3 Matrix Multiplication Accelerator [8]

This paper also used an accelerator for matrix multiplication but with a DE1-SoC board using a built in ARM Cortex-A9 HPS as the processer and a Cyclone V FPGA. They used parallel ports to communicate bettween the HPS and FPGA [1]. Two approaches were used based on the naive method. The first was to store the input matrices in two registeres and use massive parallelization to set the values in the output register. This allowed values to be accessed with zero cycles of latency and in parallel but is very hard to scale. The other method used M10k memory blocks to store the inputs. These only have a single read and write port limiting the ability to explot paralism so they opted to improve performace by pipelining and increasing clock speeds. Their tests showed that the HPS performed better than their accelerator but performed significantly more consistantly than the HPS. The accelertor and HPS ran a very different clock speeds, 100MHz and 925MHz. As the tests measured runtime in ms this would have heavly benifited the HPS. A clock cycle test as used in "Implementation of a RISC-V Processor with Hardware Accelerator" may have provided more comparable results.

2 Background

2.1 Base project: vivado-risc-v

This project uses Rocket Chip to generate RISC-V processors that can be run on various FPGAs. It provides enough documentation to build a core and flash it to the board but very little else. There is some documentation on adding a peripheral device and making it available in Linux but I was unable to get this to work and further research into how to add to the device tree resulted in limited information. It provides an example bare-metal file that can be used as a base for new code but there was no documentation on it.

2.2 AXI protocol

The AXI4 protocol is used to communicate with custom IP blocks [13]. There are 3 interfaces AXI4, AXI4-Lite and AXI4-Stream. AXI4 provides high-performance memory-mapped communication and AXI4-Stream allows for high-speed streaming of data. For this project, AXI4-Lite will be used as it is a simpler interface for low-throughput memory-mapped communication. This will limit the performance of the accelerator but will simplify development. The project will be designed in a modular way to allow the interface block to be changed without needing to modify the accelertor allowing the option to utilise the higher-performance AXI4 interface in the future.

2.3 Hardware Acceleration

A hardware accelerator is a block of hardware designed to fully exploit the parallelism available in an algorithm. An example would be matrix multiplication. The naive method takes $\mathcal{O}(n^3)$ time which can be optimised by more complicated algorithms to $\mathcal{O}(n^{2.3728596})$ [3]. Looking at the naive algorithm, it shows that while many operations are needed all the multiplications can be done in parallel and then added to generate the result. A hardware device would be able to do this allowing for it to be completed in constant time but would face issues with scaling.

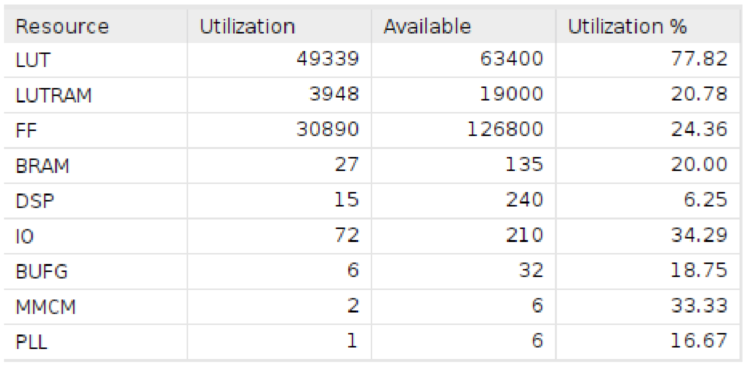
Another possible application is in image processing with local filters [6]. A pixle value is modified based on surrounding pixels in a set window using a matrix of values that scale each pixel in the window and sum to get the resulting pixel value. This can be used for various operations such as a Gaussian blur or edge detection using a Sobel filter. This process can be performed in parallel to calculate all the multiplications in a single window simultaneously or to calculate multiple windows simultaneously. The data required by a window is not all sequential so a cache is needed to hold the unused values from the input data stream until they will be needed. This can be optimised at a hardware level or by re-ordering the input data.

3 Progress

3.1 RISC-V processor

Setting up and running the vivado-risk-v [11] required the use of an Ubuntu 20.04 LTS environment to prevent issues with incorrect package versions being installed. The rocket64b1 core was generated successfully and the SD card was set up with a Linux environment. The `mk-sd-card` program required all partitions to be deleted from the SD card before use. The built-in flash function did not work so the Vivado GUI was used to manually program the FPGA through the hardware manager. This is only temporary and must be re-done every time the board is powered down but is sufficient for this project. Storing the bitstream on the SD card and loading it into the FPGA when powered could be investigated in the future. This used 77% of the FPGAs LUTs limiting the space available for custom hardware (Figure 1).

Putty was used to connect the to RISC-V processors serial port with a baud rate of 115200. It displayed the Linux boot sequence showing the process was successful. Unfortunately, this boot process was extremely slow due to an issue with the udev daemon hindering development with it. There are also issues installing programs with `apt` in the Linux environment.



Resource	Utilization	Available	Utilization %
LUT	49339	63400	77.82
LUTRAM	3948	19000	20.78
FF	30890	126800	24.36
BRAM	27	135	20.00
DSP	15	240	6.25
IO	72	210	34.29
BUFG	6	32	18.75
MMCM	2	6	33.33
PLL	1	6	16.67

Figure 1: Vivado utilisation report for risc-v processor

Due to the boot issue with Linux, bare-metal code proved to be a much better solution. The example hello-world bare-metal code from the vivado-risc-v project was successfully run to test the use of bare-metal code.

3.2 Adding IP

The Vivado development environment provides a block view of the project. The AXI interface can be found in the IO block and custom IP blocks can be connected to this to make them available over AXI. To test this pre-made AXI GPIO block [16] was added and connected to a new master AXI connection on the `io_axi_s` block. The `axi_clk` and `axi_reset` lines were used for the new IPs clk and reset ports (Figure 2). The output of the GPIO block was connected to the FPGA board LEDs. Under the address editor, a new base address can be assigned to the GPIO block (Figure 3).

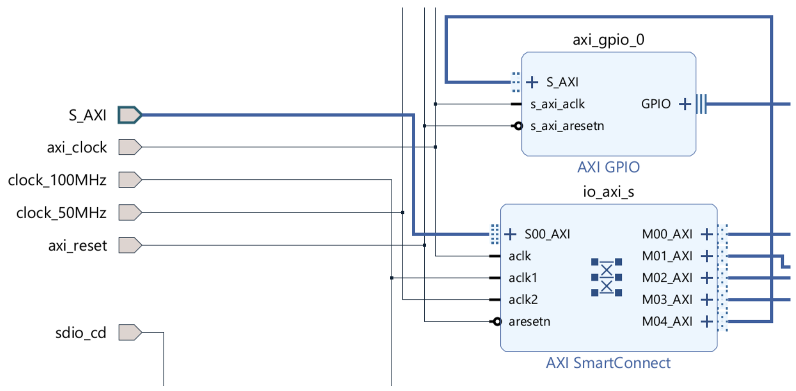
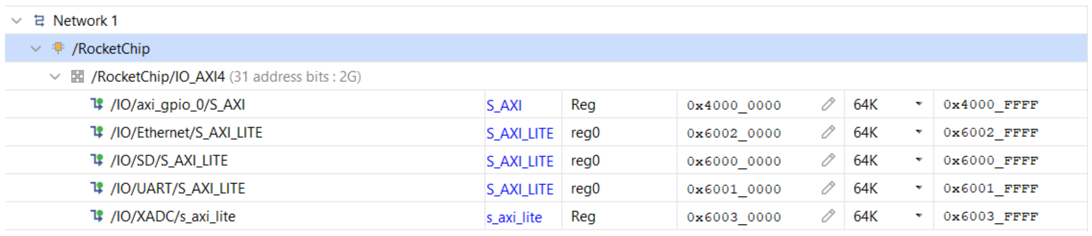


Figure 2: Vivado IP Block Diagram



Network 1

/RocketChip

/RocketChip/IO_AXI4 (31 address bits : 2G)

/IO/axi_gpio_0/S_AXI	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
/IO/Ethernet/S_AXI_LITE	S_AXI_LITE	reg0	0x6002_0000	64K	0x6002_FFFF
/IO/SD/S_AXI_LITE	S_AXI_LITE	reg0	0x6000_0000	64K	0x6000_FFFF
/IO/UART/S_AXI_LITE	S_AXI_LITE	reg0	0x6001_0000	64K	0x6001_FFFF
/IO/XADC/s_axi_lite	s_axi_lite	Reg	0x6003_0000	64K	0x6003_FFFF

Figure 3: Vivado Address Editor

Adding this to the device tree in the Linux environment using the example entry was unsuccessful and bare-metal code will be used this was not pursued further. Xilinx provides drivers [17] for their built-in IP blocks but these are significantly more complicated due to the wide range of devices it must support and include libraries that are not available. The process for interfacing with the IP block is relatively simple. Four 32-bit memory registers start at the assigned base address which represents the data and direction for each of the two 32-bit channels made available. These can be set by writing a uint32 value to a pointer at an address offset from the specified base address. This can be simplified by using a struct (Listing 1).

The example hello-world code was used as a base for developing a script to test this and interact with the serial bus.

Listing 1: Definition of a structure to access virtual memory addresses.

```
typedef struct gpio {
    volatile uint32_t gpio_data;  // Data reg 1
    volatile uint32_t gpio_tri;   // I/O direction reg 1
    volatile uint32_t gpio_data2; // Data reg 2
    volatile uint32_t gpio_tri2;  // I/O direction reg 2
} GPIO;

GPIO * gpio_reg = (GPIO *)base_address;
```

3.3 Custom AXI IP block

When creating custom IP blocks in Vivado, a blank block can be generated or a block with a pre-made AXI interface [15]). To test the use of a custom IP block the pre-made AXI interface was used and custom Verilog code was added to it. This allowed data to be written to one register copied to a second one in hardware and then read back from the new register. A basic shift register was implemented to test processing the data before output but this is not yet working.

4 Timeline

The timeline for this project has changed considerably from the specification. It was initially planned to develop the hardware accelerator first and then integrate it into the RISC-V processor. However, it was decided that starting with the processor and building on that was better as the accelerator depends on the system of communication and that depends on the processor. The updated timeline (Table 1) shows this change.

Week	Task
2	Write specification
4	Generate a working RISC-V processor
6	Integrate an IP block into the processors AXI bus
8	Integrate a basic custom IP block into the processors AXI bus
10	Write the progress report
14	Develop a custom AXI interface IP block and driver code
16	Research and design the structure of the hardware accelerator
20	Implement accelerator in a custom IP block using the AXI interface
21	Design and develop a performance test for the accelerator
23/24	Write presentation
30	Write progress report

Table 1: Updated timeline

The next thing to be developed is a custom AXI interface. This may be based on the code provided by Xilinx or documentation from the developer of ZipCPU [9]. This will be used to interface with the accelerator by transmitting 32-bit values between the bare-metal code and the hardware. This has been assigned 4 weeks due to it going through the winter holiday.

Next research will be made into methods of building the hardware accelerator and a block design will be made to show how it will be structured before beginning development. Different methods should be considered to find what is best for this project. The designed accelerator can then be implemented in Verilog and connect to the AXI interface block. A bare-metal code driver will pass in the input data and validate the output. This will be used to test the functionality of the accelerator. Input data is currently planned to be stored directly in the code but if time is available it may be modified to allow data to be loaded from the SD card.

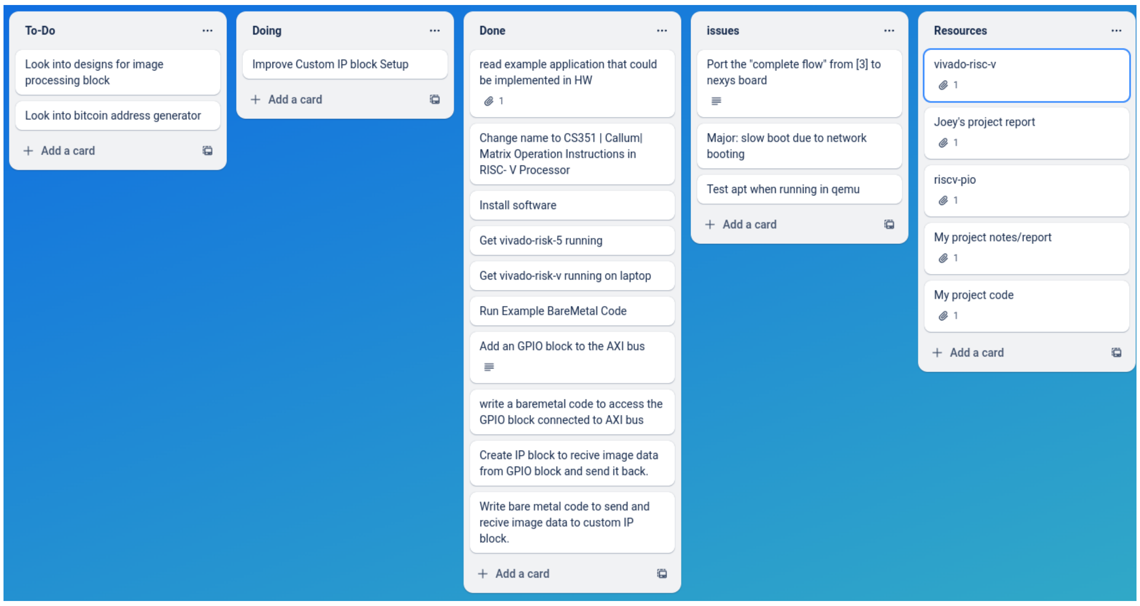
A performance test can then be developed and run using hardware and software approaches. This can then be compared to assess the usefulness of the accelerator.

Finally, the presentation and final report can be written.

5 Project management

A Trello board (Figure 4) has been used to keep track of weekly tasks, resources and issues. The tasks can be marked as "To-Do", "Doing" and "Done". This is reviewed in weekly meetings with the project supervisor and next week's tasks are decided on.

Progress is behind what would be expected if the original timeline had been restructured due to unavoidable personal issues arising and not putting in the hours per week as planned in the specification. A more organised weekly plan will be used to help with time management on the rest of the project and the scope of the project has been changed to allow for the lost time. The switch to using bare-metal code has made the software side considerably simpler and greatly increases the possible speed of development. Using Verilog to develop the hardware instead of Chisle reduces the learning curve allowing for development to start sooner.



To-Do
- Look into designs for image processing block
- Look into bitcoin address generator

Doing
- Improve Custom IP block Setup

Done
- read example application that could be implemented in HW
- Change name to CS351 | Callum| Matrix Operation Instructions in RISC- V Processor
- Install software
- Get vivado-risk-5 running
- Get vivado-risk-v running on laptop
- Run Example BareMetal Code
- Add an GPIO block to the AXI bus
- write a baremetal code to access the GPIO block connected to AXI bus
- Create IP block to recive image data from GPIO block and send it back.
- Write bare metal code to send and recive image data to custom IP block.

issues
- Port the "complete flow" from [3] to nexys board
- Major: slow boot due to network booting
- Test apt when running in qemu

Resources
- vivado-risc-v
- Joey's project report
- riscv-pio
- My project notes/report
- My project code

Figure 4: Current status of the project Trello board

6 Ethics

This project does not require working with people and should not present any ethical issues. There should also be no legal issues as there is no intent to profit from this project.

Glossary

FPGA -	Field-Programmable Gate Array, an integrated circuit designed arround configurable logic blocks allowing for custom logic functions to be defined using HDL [14].
LUT -	Lookup Table, a digital memory element used in FPGAs to implement combinational logic.
HDL -	Hardware Description Language, a language used to describe logic circuits.
RISC-V -	An open standard instruction set for processors [2].
APB -	Advanced Peripheral Bus, a low-cost interface for peripheral devices [4].
AXI -	Advanced eXtensible Interface, a protocol used to comunicate bettween the processor and IP blocks [13].
Bare-metal -	Software that runs directly on hardware without an operating system.

References

[1]

[2] About risc-v. URL `https://riscv.org/about/`.

[3] Alman, Josh & Williams, Virginia Vassilevska. A refined laser method and faster matrix multiplication, 2020. URL `doi.org/10.48550/arXiv.2010.05846`. arXiv:2010.05846 [cs.DS].

[4] Arm, . Amba® apb protocol specification. URL `https://documentation-service.arm.com/static/60d5b505677cf7536a55c245`. (Accessed December 2023).

[5] Asanović, Krste & Avizienis, Rimas & Bachrach, Jonathan & Beamer, Scott & Biancolin, David & Celio, Christopher & Cook, Henry & Dabbelt, Daniel

& Hauser, John & Izraelevitz, Adam & Karandikar, Sagar & Keller, Ben & Kim, Donggyu & Koenig, John & Lee, Yunsup & Love, Eric & Maas, Martin & Magyar, Albert & Mao, Howard & Moreto, Miquel & Ou, Albert & Patterson, David A. & Richards, Brian & Schmidt, Colin & Twigg, Stephen & Vo, Huy & Waterman, Andrew. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[6] Bailey, Donald G. Design for embedded image processing on fpgas, 2011.

[7] Blomkvist, Ludvig & Oscarsson, Jonas Ibrahimoglu & Nilsson, Lucas & Stenseke, Adam & Wennerberg, Joakim. Implementation of a risc-v processor with hardware accelerator, 2019. URL `https://odr.chalmers.se/server/api/core/bitstreams/8a3fa1e8-9dd1-4e4c-a598-6d4967d381bd/content`. (Accessed October 2023).

[8] Dempsey, Brian & Patterson, Liam. Matrix multiplication accelerator, 2020. URL `https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2020/bjd86_lgp36/bjd86_lgp36/index.html`. (Accessed October 2023).

[9] Gisselquist Technology, LLC. Buidilng an axi-lite slave the easy way. URL `https://zipcpu.com/blog/2020/03/08/easyaxil.html`. (Accessed December 2023).

[10] pulp platform, . Pulpino. URL `https://github.com/pulp-platform/pulpino`. (Accessed December 2023).

[11] Tarassov, Eugene. vivado-risk-v. URL `github.com/eugene-tarassov/vivado-risc-v`. (Accessed October 2023).

[12] William, Stallings. *Computer organization and architecture, Global Edition*. Pearson Education, Limited, 2021.

[13] Xilinx, . Axi reference guide, . URL `https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf`. (Accessed December 2023).

[14] Xilinx, AMD. What is an fpga?, . URL https://xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. (Accessed October 2023).

[15] Xilinx, AMD. Axi4-lite ip interface (ipif), . URL https://www.xilinx.com/products/intellectual-property/axi_lite_ipif.html. (Accessed December 2023).

[16] Xilinx, AMD. Axi general purpose io, . URL https://www.xilinx.com/products/intellectual-property/axi_gpio.html. (Accessed December 2023).

[17] Xilinx, AMD. Axi gpio driver, . URL https://github.com/Xilinx/embeddedsw/tree/master/XilinxProcessorIPLib/drivers/gpio. (Accessed November 2023).