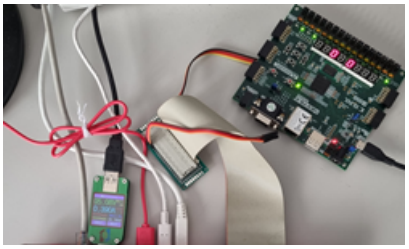
TÉCNICO LISBOA



Evaluation of Real Time Operating System in RISC-V

Rui Silva Ferreira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Rui António Policarpo Duarte
Prof. Paulo Ferreira Godinho Flores

Examination Committee

Chairperson: Prof. Teresa Maria Canavarro Menéres Mendes de Almeida
Supervisor: Prof. Rui António Policarpo Duarte
Member of the Committee: Prof. José João Henriques Teixeira de Sousa

June 2024

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank my thesis supervisors Professor Paulo Flores and Professor Rui Policarpo Duarte for all the guidance and support they gave me during this work. Also i would like to acknowledge the opportunities and experience shared that helped motivate me.

I would like to thank my parents for supporting me and for providing me with the opportunity to reach this achievement. Finally, i would like to thank my partner and friends for all the help, support and patience they provided during this journey.

Resumo

A arquitetura do conjunto de instruções RISC-V surgiu como uma alternativa devido à sua natureza aberta, simplicidade e modularidade. As implementações de sistemas em circuitos incorporados (SoC) baseados em processadores RISC-V podem utilizar aceleradores de hardware para melhorar o seu desempenho, transferindo tarefas computacionais pesadas do processador. A utilização de um sistema operativo em tempo real é essencial para satisfazer os requisitos temporais e gerir eficientemente múltiplas tarefas. Nesta tese, a SoC SweRVolf foi adaptada para se conectar a um acelerador de hardware. Foram propostos três cenários para gerir o acelerador usando mecanismos do RTOS Zephyr para que o processador RISC-V execute diferentes tarefas enquanto o acelerador está a trabalhar. No primeiro cenário, uma única thread acede ao acelerador, sendo bloqueada por um semáforo enquanto este está a funcionar. No segundo cenário várias threads competem pelo acesso ao acelerador e os semáforos controlam as tentativas de acesso. O terceiro cenário usa uma thread responsável por gerir o acesso ao acelerador. As threads computacionais enviam pedidos à thread gestora para utilizar o acelerador. Para avaliar estes cenários, duas aplicações foram desenvolvidas e executadas na placa Nexys A7, a multiplicação de um conjunto de matrizes e uma rede neural para prever os dígitos do conjunto de dados MNIST. Os tempos de execução dos diferentes cenários mostram que a utilização de uma thread de gestão é mais rápida do que as múltiplas threads competindo pelo acesso ao acelerador e permite um maior controlo sobre a thread que utiliza o acelerador.

Palavras Chave

RISC-V, Sistema Operativo de Tempo Real, Acelerador de Hardware, SweRVolf SoC, Zephyr

Abstract

The RISC-V instruction set architecture has emerged as in the community due to its open and royalty-free nature, simplicity, and modularity. System-on-Chip (SoC) implementations based on RISC-V processors can be extended with hardware accelerators to improve the performance of embedded systems by offloading heavy computational tasks from the processor. Using a Real-Time Operating System (RTOS) is essential in meeting the timing requirements and efficiently managing multiple tasks. In this thesis, the SweRVolf SoC was extended to connect to a hardware accelerator. Three scenarios were proposed for managing the hardware accelerator using Zephyr RTOS mechanisms that enable the RISC-V processor to execute different tasks, while the accelerator works in parallel. In the first scenario, a single thread accesses to the accelerator and a semaphore is used to block the thread while it's working. The second scenario has several threads competing to access the accelerator and semaphores control access attempts. The third scenario uses a thread responsible for managing the accelerator's access. Computational threads send requests to the manager thread to use the accelerator. To evaluate these scenarios, two use case applications were developed and executed on the FPGA board Nexys A7, a set of matrices multiplications and a neural network to predict the digits of the MNIST dataset. The execution times of the different scenarios show that using a manager thread is more time-efficient than the multiple threads competing for access to the accelerator, while providing a larger control over what thread uses the accelerator.

Keywords

RISC-V, Real-time Operating System, Hardware Accelerator, SweRVolf SoC, Zephyr.

Contents

List of Tables

List of Figures

Acronyms

AXI	Advanced eXtensible Interface
ALU	Arithmetic Logic Unit
CDC	Clock Domain Crossing
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
ISA	Instruction Set Architecture
IP IER	IP Interrupt Enable Register
IP ISR	IP Interrupt Status Register
IRQ	Interrupt Request
ISR	Interrupt Service Routine
GPIO	General Purpose Input/Output
GIER	Global Interrupt Enable Register
OS	Operating System
PIC	Programmable Interrupt Controller
PTC	PWM Timer Counter
RTOS	Real-Time Operating System
RTS	Real-Time System
RAM	Random Access Memory
SPI	Serial Peripheral Interface
SoC	System on Chip
UART	Universal Asynchronous Receiver-Transmitter
VIP	Verification IP
WSL	Windows Subsystem for Linux

1

Introduction

Contents

1.1 Motivation

An Operating System (OS) is a software that is responsible to manage the system hardware resources and provide common services for programs. The most common known OSs for computers are Windows, Linux, MacOS. [2]. Other OSs, targeted for embedded systems and called Real-Time Operating System (RTOS), usually can be customized and configured for the unique demands of the devices. RTOS, with requirements of embedded systems, include features like real-time and reactive operations, direct use of interrupts, limitations in power consumption and resource availability. RTOSs manage a system in which the main core is a processor or a microcontroller.

RISC-V is an open Instruction Set Architecture (ISA) that has been establishing itself as a standard in both an academic and industry environment [3]. The royalty-free nature of RISC-V enables a broader community around it that contributes to its development. In particular, in the development of hardware processors that support the RISC-V ISA, in which an RTOS could be used to manage the available resources of the system.

Nowadays, it is common the existence of hardware accelerators targeting different tasks. Hardware accelerators are digital circuits used to speed-up algorithms that usually are the computational bottleneck of the system or require computational intensive tasks.

The usage of a system with an RTOS, to manage the different tasks of the system, and, at the same time, using a hardware accelerator to remove computational bottlenecks, is becoming important is most systems. Since the hardware accelerator and the processor are separate circuits, they can be used simultaneously to perform a variety of computations. Several techniques can be used to manage the accelerator from a RTOS running on a RISC-V processor. However, techniques such as pooling are unable to take advantage of using simultaneous the two different circuits. Therefore, it is important to explore the capabilities of the RTOS to ensure that the processor can deal with other tasks while the accelerator is being used.

1.2 Objectives

This thesis aims to study the impact of using an RTOS for managing the access to a hardware accelerator within a System on Chip (SoC) with a RISC-V based processor. The integration of an RTOS into the system introduces complexities and opportunities for optimization, particularly in terms of accessing and controlling the accelerator and its impact in execution time and power overhead.

For this objective it's necessary to select a suitable implementation of the RISC-V processor integrated into an SoC. Additionally, it's necessary to select an RTOS compatible with the chosen SoC. The RTOS must also provide the necessary mechanisms to use several threads to interact with accelerator.

An hardware accelerator needs to be developed and connected to the selected SoC. This involves

understanding and configuring communication interfaces, memory access, and the interrupt mechanisms.

Software responsible for managing the hardware accelerator access from an application will be developed for an extended SoC, with the RISC-V processor and the designed accelerator connected. This should use the capabilities offered by the selected RTOS, such as task scheduling, resource allocation, and synchronization mechanisms.

Finally, it is pretended to execute the different use cases applications developed in the Field Programmable Gate Array (FPGA) board Nexys A7 to compare and evaluate the cost and benefits of using the mechanisms provided by the RTOS. This comparison will mainly focus on the time efficiency, considering the overhead introduced by the RTOS, and the power efficiency of the applications.

1.3 Document Outline

The next chapter, Chapter 2, provides an overview of the RISC-V Instruction Set Architecture and some of the most relevant Real-time Operating System's (RTOS) mechanisms. This chapter also provides an analysis of the available RISC-V based SoC and open source RTOSs.

Chapter 3 covers the description of the development of the hardware accelerator for matrix multiplication. It also covers the description of the interrupt management system that is implemented in the hardware accelerator.

Chapter 4 provides a description of the base implementation of the SweRVolf SoC, and the modifications done on the SoC implementation to support the developed hardware accelerator.

Chapter 5 presents three different alternative scenarios to manage the hardware accelerator using the RTOS capabilities, such as interrupts, semaphores, and message queues.

Chapter 6 presents a performance assessment of the scenarios previously introduced, using two use cases applications: matrix multiplication and character recognition using Neural Networks. In this assessment, metrics such as time and power efficiency are measured.

Chapter 7 presents the final conclusions of this thesis and also suggest some future work to improve the analysis done with this work.

2

RISC-V ISA and RTOS overview

Contents

This chapter provides an overview of the RISC-V Instruction Set Architecture (ISA) and different RTOS mechanisms used in systems with multiple threads. The chapter covers the RISC-V architecture and several SoC implementations with RISC-V based processors. It also highlights some of the features of the RISC-V ISA, such as its modularity, simplicity and open-source nature.

The chapter also provides a detailed examination of RTOS mechanisms, including task scheduling, context switching, interrupt handling, and resource sharing. This includes an analysis of several open-source RTOS such as FreeRTOS, RIOT, and Zephyr, evaluating their compatibility with RISC-V based SoCs and their community support and documentation.

2.1 RISC-V Architecture and SoC Implementations

2.1.1 RISC-V Instruction Set Architecture

RISC-V [4, 5] is an open-source ISA. The open-source nature of RISC-V encourages development by allowing developers to use it without any additional financial cost. RISC-V International [6] defines four base ISA, each providing a minimal set of instructions that ensure compatibility with compilers, assemblers, and operating systems. Beyond the base ISA, RISC-V International offers standard extensions and the ability for developers to create custom extensions. These extensions allow developers to tailor the processor to suit the specific application requirements.

The four primary base ISA are characterized by the width and number of integer registers. The two main base ISA are RV32I and RV64I, providing 32-bit and 64-bit address spaces, respectively, both with 32 integer registers. RV32E, a subset of RV32I, features only 16 registers and targets small microcontrollers. The fourth base ISA, RV128I, supports a 128-bit address space. Among the standard extensions the 'M' and 'C' are the most notable. The 'M' extension incorporates integer multiplication and division for both signed and unsigned integers. The 'C' extension provides compressed, 16-bit, instructions for common operations. Typically, around 50% of the instructions can be replaced by the compressed instructions, resulting in a 25% code reduction. Other notable extensions are the 'A' extension for atomic instructions, the 'F' extension for floating-point operations, and floating-point registers, and the 'Zicsr' extension that enables access to the control and status register, which facilitates interrupts and exception handling.

RISC-V also includes privileged levels with optional extensions and variants. There are currently three privilege levels: User (U), Supervisor (S), and Machine (M). These privilege modes exist to enforce protection between different components of the software stack. The Machine mode has the most privileges and runs implementation-specific firmware. The Supervisor mode is typically used by operating systems, giving access to most of the hardware. The User mode offers the least privileges and is where user applications generally run. All implementations must provide the Machine mode since this is

the only privileged mode with access to the entire hardware.

2.1.2 RISC-V SoC Implementations

There are several hardware implementations of RISC-V available, however, this study only examines implementations listed under SoC platforms section within the list [7]. Since the focus of this thesis is to study the impact of an RTOS on the existing RISC-V based SoC, the selection of SoC implementations is limited to fully developed SoC. The criteria for selection of the SoC primarily focus on the existence of an open-source license, a robust documentation, and a responsive and active community support. These factors ensure that the chosen SoC offers the necessary resources for the adaptation of the system done in this work.

Three RISC-V based SoC implementations were selected for more in-depth analysis: LiteX [8], SweRVolf [9] and NEORV32 [10]. Table 2.1 presents a comparison of key factors, including the base ISA, the supported extensions, the provided Random Access Memory (RAM) size, the system clock speed, the supported simulation tool, and the current RTOS support.

All of the three SoCs displayed in table 2.1 utilize the RV32I base ISA. LiteX supports a broader range of extensions, M, A, F, D, and C, but NEORV32 supports the extension Zicsr, which facilitates the implementation of interrupts. On the other hand, SweRVolf depends on software to implement interrupts, and the extensions M and C are enough to meet the requirements of this study, since it allows the installation of a popular RTOS such as Zephyr and FreeRTOS. In terms of RAM size, all three offer sufficient capacity to run the necessary programs, since the necessary memory for the RTOS studied in section 2.2 is in the kilobyte range. LiteX supports lower clock speeds, while NEORV32 supports the highest clock speed. This paired with the Zicsr extension makes NEORV32 a very strong candidate for the following work. However, the deciding factor is the documentation available and the community support, in which the SweRVolf has a clear advantage. Amongst the available documentation, the SweRVolf SoC provides guides on how to install and connect external hardware interrupts, even without the Zicsr extension.

Feature	LiteX	SweRVolf	NEORV32
Base ISA RISC-V Extensions	RV32I [M][A][F][D][C]	RV32I [M][C]	RV32I [M][C][Zicsr][Zicntr]
RAM system clock	256MB 1MHz	128MB 50MHz	configurable 150MHz
Simulation tool	Verilator	Verilator	ISIM
Support for RTOS	Zephyr	Zephyr, TockOS, FreeRTOS	Zephyr, FreeRTOS
Documentation	+	++	+

Table 2.1: Comparison between RISC-V SoC

2.2 Real-Time Operating Systems

A Real-Time System (RTS) is a system where its correctness depends on both the output and the time taken to produce it. These systems can be classified into three types:hard real-time systems, soft real-time systems, and firm real-time systems. Hard real-time systems have strict deadlines, and failure to meet those deadlines results in system failure. Soft real-time systems allow tasks to miss deadlines, which can lead to a reduction in the overall system performance. Firm real-time systems have strict deadlines for critical tasks, yet missing the deadline of non-critical tasks does not lead to system failure.

In more complex scenarios, the utilization of an Operating System (OS) becomes necessary for managing resources and meeting systems demands. An RTOS is specifically designed to meet the time constraints in RTS. As an OS, an RTOS must have fundamental functionalities within its kernel, including, a process scheduler, multitasking, inter-tread communication/synchronization, interrupt handling, timers/clocks, and memory management.

One essential capability of an OS is multitasking, which is the capacity of the OS to swap between tasks to optimize system performance. Tasks can be categorized into three states: running, ready, and blocked. The running task is actively executing on the Central Processing Unit (CPU), while ready tasks await execution and blocked tasks are temporarily blocked, usually awaiting specific events. To manage task execution, the kernel uses the scheduler, which is a pre-determined scheduling algorithm that decides what task should be actively executing in the CPU. To run the tasks, the OS uses threads, witch represent the smallest unit of processing that can be executed. Each thread has its own stack, program counter, and register set, which allows it to run independently from other threads. Threads can communicate with each other using shared memory, or inter-thread communication mechanisms provided by the OS.

To move from the blocked state to the ready state, threads may need to wait for events to occur. These events, called interrupts, can be triggered by other tasks or by hardware peripherals in the system. To handle interrupts, the RTOS utilizes what is called Interrupt Service Routine (ISR).

2.2.1 Task Scheduling and Context Switching

The scheduler is the mechanism that is in charge of choosing the task that is executed at any time. The scheduler is especially important in RTOS since it has to ensure that all tasks meet their time constraints. Scheduling algorithms can be divided into two types: preemptive and cooperative. The preemptive scheduler can switch the running task even if that task does not go into the blocked state. The cooperative scheduler does not swap the task that is running, but instead, the running task yields execution periodically. There are a few situations where the scheduler needs to decide what task should run next: when the running task finishes, when the running task moves to the blocked state, or when

an interrupt occurs. In RTOS the most commonly used scheduling algorithms are preemptive since they provide more control over what task is running.

The most commonly used preemptive scheduling algorithm is priority scheduling or highest priority first. In this scheduling algorithm, every task is assigned a priority. The scheduler then makes sure that the task with the highest priority is always running. If a low-priority task is running and a new high-priority task goes to the ready state, then that new task starts running and the old task goes to the ready state. Another scheduling algorithms are first come first served, where the selected task is the first task to arrive at the ready state, or the shortest job first, where the task with the least completion time left executes first. One example of a cooperative scheduling algorithm is round-robin scheduling, where the tasks are not assigned a priority value, but are swapped after a set period of time. It's very common for RTOS to use this type of scheduler to assign running time to tasks of the same priority in priority scheduling.

Context switching is the mechanism that saves the current state of the task for future use, and initializes the state of the new running task. While context switching is essential for task management, it adds overhead that can cause the system to miss the timing requirements. That is why the context switching in RTOS must be as time-efficient as possible. The frequency of context switches is influenced by the scheduling policy, with preemptive scheduling usually resulting in more context switches than non-preemptive schedulers. This means that the scheduler must consider the context-switching delay when determining which task to execute next.

2.2.2 Interrupt Handling

The ISR is a high-priority task that runs every time an interrupt occurs and manages the cause of that interrupt. An interrupt is a signal that can be emitted by hardware, such as a mouse or a keyboard, and by software. After the triggering of an interrupt, the Interrupt Request (IRQ) associated with the interrupt utilizes the interrupt vector table to determine the correct interrupt handler function. All of this adds a delay which is called interrupt latency. More precisely, interrupt latency is the time between the interrupt signal and the conclusion of the ISR.

Before starting the ISR, the CPU must save the context of the thread it was executing. Upon completion of the interrupt handling routine, the scheduler picks what task to execute next. Occasionally, the running thread may need to prevent the ISR from executing, typically due to some time-sensitive task or critical section operation. In those cases, it is possible to disable the ISR, however, this should only be considered as a last resort when no alternative is available, since it may delay other critical tasks.

2.2.3 Resource Sharing

In a system with multiple threads it's necessary to have mechanisms for sharing data between those threads. It's generally unsafe for systems capable of multitasking to access the same memory addresses without protection. Protection may be achieved using RTOS mechanisms such as temporarily masking, synchronization mechanisms and communication mechanisms.

Temporarily Masking

Temporarily masking is when the user temporarily disables interrupts and the ability for the kernel to switch tasks. This way the task can process the data knowing that no other tasks will change that data before its processing finishes. Temporarily masking is the fastest way to make sure that the current task has exclusive access to the system resources. However, it should only be used when the longest path through the protected section is smaller than the interrupt latency since this will prevent any other task from running, even higher priority ones.

Inter-tread Synchronization Mechanisms

A semaphore is an object that controls access to a specific resource. The semaphore stores how many tasks can access that resource and how many tasks are accessing it at that time. When the number of tasks reaches the limit, no other tasks are allowed to access that resource. This way whenever a task needs to access shared resources it increments the semaphore counter. When that thread finishes its processing it decreases the semaphore counter. When only one task can access that resource the semaphore is called a binary semaphore.

Similarly to binary semaphores, there is a mechanism called mutex. A mutex ensures mutually exclusive access to a hardware or software resource. When a task wants to access a mutually exclusive resource it must start by locking the associated mutex. After the task is finished it unlocks the locked mutex. While the mutex is locked no other tasks can access the resources protected by the mutex. These tasks are blocked and only start executing when the mutex is unlocked. One key difference between mutexes and semaphores is priority inversion. Priority inversion happens when a low-priority task locks a mutex and a high-priority task is blocked by that mutex. When this happens the priority level of the original task is elevated so that the higher-priority task does not have to wait for the low-priority task to be assigned CPU time. One problem that comes with mutexes is deadlocking. This happens when one task locks some resource needed by another task but cannot access some other resource that was locked by the second task. In this case, both tasks are waiting for the other task to unlock a mutex and are unable to continue executing. This can be prevented by carefully designing the program.

Inter-tread Communication Mechanisms

Inter-tread communication mechanisms such as FIFOs, stacks, message queues, and mailboxes are used to share data between tasks without utilizing shared memory. With these data-sharing mechanisms, a task only shares the data when its processing is complete. This ensures that the other tasks only start processing the data after receiving it from the first task. Techniques such as priority inversion can be used to guarantee that each task only starts executing when the data is available. Deadlock can also occur when some task is processing data required by another task, yet is waiting for data from a blocked task.

2.2.4 RTOS Comparison

An analysis of several open-source RTOS, listed in [11], was conducted. Immediately, several RTOS were excluded from consideration due to either lack of recent updates or inadequate documentation. Following this initial filtering, the RTOS that stood out were FreeRTOS [12], RIOT [13], and Zephyr [14].

FreeRTOS

FreeRTOS is an RTOS that targets small embedded systems. The RTOS has a minimal footprint and prioritizes its small memory size, low overhead, and fast execution. The RTOS Kernel can be tailored to the specific use case of the system with a configuration file called FreeRTOSConfig.h. In addition, its small and simple kernel makes it a candidate for systems with limited resources.

FreeRTOS has a very small footprint needing only 4.4 KB of storage and 500 Bytes of RAM. FreeRTOS has a very fast context switch delay, needing only 84 cycles to swap between tasks. The RTOS uses a highest priority scheduling algorithm, with Round Robin time slicing. Both of these algorithms can be disabled in the configuration file. FreeRTOS also provides documentation and community support.

RIOT

RIOT is an RTOS designed to cater to the requirements of Internet of Things devices. The main goal of RIOT is to provide a lightweight but versatile RTOS that runs on a low-power microcontrollers but can also use the full power of larger devices with more resources available. RIOT uses a modular architecture that allows developers to include only the components needed for the specific system. The programming model is based on event-driven programming, allowing a highly responsive and efficient system that still supports multitasking. RIOT has small footprint that requires only 3.2 KB of storage and 2.8 KB of RAM. It also has a small interrupt latency that takes only 50 clock cycles. RIOT uses a highest priority first scheduling algorithm and a first come first served algorithm for tasks of the same priority.

Zephyr

Zephyr is an RTOS developed by the Linux Foundation, and adopted by companies such as Google and Facebook in 2020, [15]. It offers a scalable and secure platform for IoT devices. Similarly to RIOT, it offers a modular architecture that allows developers to only select the necessary features of the RTOS. Zephyr provides developers a feature-rich software that can run in a footprint as small as 8kB and as large as to reach megabytes and supports both 32bit and 64bit architectures.

Zephyr uses a highest priority first scheduler and utilizes the algorithm longest waiting first for tasks of equal priority. It also allows the developer to choose the scheduling algorithm for tasks with different and equal priority. It is also possible to define the task queuing strategies used in the scheduler.

Feature Comparison

Three RTOS were compared in terms of compatibility, interrupt processing speed, and documentation First, the RTOS must be compatible with the selected SoC. Additionally, the speed at which the RTOS can process interrupts and switch task context is vital to meet eventual time constraints. Finally, complete documentation and community support are imperative for software development. Table 2.2 summarizes these characteristics. It's worth noting that some of the more precise metrics, such as interrupt latency and context switch delay, do not have any strict value since they depend on the specific use case of the RTOS.

Table 2.2 shows that FreeRTOS has a smaller footprint, making it ideal for systems with less memory. It is also a better RTOS when dealing with small tasks since the context switch delay has a higher impact. At the same time, due to the variety of capabilities available, Zephyr is expected to perform better for larger systems. The support for the chosen SoC is also an important feature to consider. None of the RTOS analyzed have direct support in their documentation page. However, both FreeRTOS and Zephyr have guides on how to use them with the chosen SoC, SweRVolfX. Since FreeRTOS and Zephyr are acceptable for the analysis done in the following chapters, the ample community support and documentation available make Zephyr the best candidate.

Feature	FreeRTOS	RIOT	Zephyr
Minimum storage size Minimum RAM size	4.4 KB 500 Bytes	3.2 KB 2.8 KB	8 KB 5 KB
Interrupt Latency Context Switch Delay	— 84 cycles	50 cycles —	— 264 cycles
Scheduling Policy Time Slicing Policy	Highest Priority Round Robin	Highest Priority none	Highest Priority configurable
Documentation	++	+	++

Table 2.2: Comparison between FreeRTOS, RIOT and Zephyr

3

Hardware Accelerator for Matrix Multiplication

Contents

This chapter describes the design of an hardware accelerator for matrix multiplication. The chapter begins with the description of the necessary pragmas to implement the AXI protocol used to communicate with the CPU and the RAM. It also describes the algorithm used as a base for the circuit design.

The chapter also discusses the interrupt management system implemented in the hardware accelerator, which is crucial in the scenarios developed in Chapter 5. In this chapter provides a description of the registers used in the accelerator to program the interrupt system connected to the Programmable Interrupt Controller (PIC) in the CPU.

3.1 Hardware Accelerator Design

Hardware accelerators are circuits designed to increase the efficiency of a specific task and are used in several different contexts such as neural networks, cryptography and image processing. In these examples, part of the computation done in the CPU is transferred to the accelerator. The hardware accelerator used in this thesis is designed to perform matrix multiplication and replace the CPU whenever that operation is executed.

The communication between the hardware accelerator and the other components is done using the Advanced eXtensible Interface (AXI) protocol [16]. The reason for this choice will be discussed in Chapter 4. The accelerator has an AXI4 Lite interface with the CPU for the control signals and an AXI4 interface with the RAM for data transfers. Listing 3.1 shows the pragmas used in Vitis HLS to define the two communication interfaces. The AXI Lite interface has four ports, one for the control signals, and the last three for the sizes of the two matrices. The AXI interface has three ports for the two input matrices and the result matrix. Since the accelerator circuit requires the memory addresses of the three matrices to access them in the RAM, three ports are automatically added to the AXI Lite bundle. Table 3.1 shows the addresses of the AXI Lite ports.

Listing 3.1: AXI4 Lite and AXI4 interfaces defenition for the hardware accelerator

```
1 #pragma HLS INTERFACE s_axilite port=return bundle=BUS1
2 #pragma HLS INTERFACE s_axilite port=N1 bundle=BUS1
3 #pragma HLS INTERFACE s_axilite port=N2 bundle=BUS1
4 #pragma HLS INTERFACE s_axilite port=N3 bundle=BUS1
5
6 #pragma HLS INTERFACE m_axi port = m1 depth=MAX_DEPTH_SIZE
7 #pragma HLS INTERFACE m_axi port = m2 depth=MAX_DEPTH_SIZE
8 #pragma HLS INTERFACE m_axi port = m3 depth=MAX_DEPTH_SIZE
```

The hardware accelerator is designed using Xilinx's Vitis HLS [17], which allows users to design

Register	Offset
CTRL	0x00
m1	0x10
m2	0x18
m3	0x20
N1	0x28
N2	0x30
N3	0x38

Table 3.1: Offset of the AXI4 Lite registers.

FPGA circuits using a high-level language such as C. The code used to design the accelerators circuit with Vitis HLS is present in Listing 3.2 and consists of three nested *for* loops. The input matrices are copied from the memory to the accelerator in the first two lines. The function *memcpy* takes as arguments the destination of the data, the source of the data, and the number of bytes to be copied. Vitis HLS automatically uses the AXI interface to read and write in the addresses of the RAM. The nested *for* loop, where the matrices are multiplied, is in lines four to thirteen. In line fifteen the output matrix is copied to memory using the function *memcpy*.

Listing 3.2: Hardware accelerator C specification

```
1  memcpy(m1_buffer, (const float*)m1, N1*N2 * sizeof(float));
2  memcpy(m2_buffer, (const float*)m2, N2*N3 * sizeof(float));
3
4  for (int i =0; i <N1; i ++) {
5      for (int j=0; j<N3; j++) {
6          for (int k=0; k<N2; k++) {
7              float mul = m1_buffer[i*N2+k] * m2_buffer[k*N3+j];
8              if (k==0) regc = mul;
9              else regc += mul;
10         }
11         m3_buffer[i *N3+j] = regc;
12     }
13 }
14
15 memcpy((float*)m3, m3_buffer, N1*N3 * sizeof(float));
```

Vitis HLS automatically creates the hardware implementation of the accelerator and provides the resource estimations shown in Table 3.2.

The hardware implementation of the accelerator was tested using Vitis HLS's simulation tool. The simulation consists of writing the sizes and addresses of the matrices in the accelerators ports and

BRAM	DSP	FF	LUT	URAM
6	18	2731	3662	0

Table 3.2: Matrix multiplication accelerator's resource estimation in Vitis HLS

initializing the accelerator using the control signal. In this simulation two four by four matrices are used as an input to the accelerator.

Figure 3.1 shows how the AXI protocol is used to write in the accelerators registers the necessary values to perform the matrix multiplication. The signal *AWADDR* shows the addresses of the accelerators registers and the signal *WDATA* shows the values of the data written in those addresses. Comparing the values of the signal *AWADDR* to the ones on Table 3.1 it's possible to confirm that the registers used are the matrix addresses and their sizes. Looking at the signal *WDATA* its possible to verify that the values being written are correct, since the value being written in the addresses of the matrix sizes is 4. The last value being written with the signal *WDATA* is the control signal at address 0x00, and it starts the execution of the accelerator.

Figure 3.2 shows the waveform of the end of the accelerators execution. The signal *gmem_AWADDR* expresses the address of RAM where the result is written, and has the values 0x0100, which matches the value written in the accelerator in Figure 3.1. The signal *gmem_WDATA* has the sixteen values of the matrix that are written in the address 0x0100. Once the results of the matrix are written in the RAM the control signal of the accelerator changes its value from 0x01 to 0x0e. The signal *RDATA* is used to read the value on address 0x00, expressed by the signal *ARADDR*, and shows the change in values of the control signal that represents the accelerators state change from working to idle.

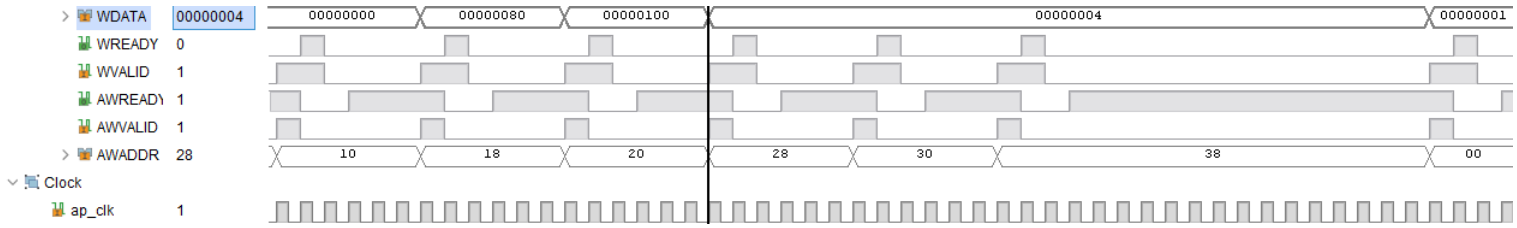


Figure 3.1: Waveform of the initial signals necessary to start the accelerator

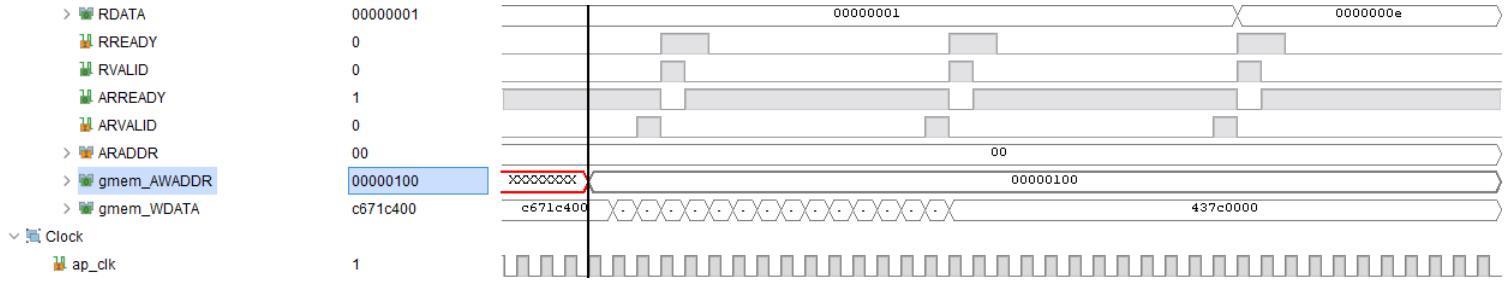


Figure 3.2: Waveform of the signals that represent the conclusion of the accelerator operation

3.2 Interrupt Management System

Vitis HLS includes an interruption system in the hardware accelerators circuit. There are four signals in the accelerator used to activate and receive the interrupts. Three of those signals are registers of the accelerator used to enable the interrupts and to define what triggers the interrupt. The fourth signal is connected to the CPUs PIC to trigger the interrupt itself.

Table 3.3 shows the registers used to enable and disable the interrupts and to define the conditions that trigger the interrupt. The register Global Interrupt Enable Register (GIER) is used to enable all the interrupts generated by the accelerator. GIER is a 1 bit register and can be set to enable the interrupts and reset to disable the interrupts. The register IP Interrupt Enable Register (IP IER) is used to enable interrupts triggered either by the done signal or the idle signal. IP IER is a 2 bit register, where the first bit is set to enable interrupts triggered by the done signal from the accelerator, and the second bit is set to enable interrupts triggered by the idle signal from the accelerator. The register IP Interrupt Status Register (IP ISR) is used to check the status of the interrupt. Similarly to IP IER, the register IP ISR is a 2 bits register, where the first bit indicates the status of the interrupt triggered by the done signal, and the second bit indicates the status of the interrupt triggered by the idle signal. When an interrupt is triggered, the registers bit associated with that interrupt source will be toggled to '1' indicating the state of the interrupt. When IP ISR is toggled to '0', the interrupt has been handled and the accelerator stops sending the interrupt signal to the PIC.

Register	Description	Offset
GIER	Global Interrupt Enable Register	0x04
IP IER	IP Interrupt Enable Register	0x08
IP ISR	IP Interrupt Status Register	0x0c

Table 3.3: Offset of the interrupt enable and status registers

4

RVfpga SoC for Matrix Multiplication Acceleration

Contents

This chapter focuses on the RVfpga SoC, adapted to support the hardware accelerator designed in Chapter 3 and targated to the Nexys A7 FPGA board. The chapter provides a description of the baseline SweRVolf SoC, detailing the RISC-V processor SweRV EH1 and the additional components of the SoC.

The chapter provides some insight on the development of an AXI interconnect to accommodate for the hardware accelerators ports. It also includes the description of the final SoC implementation with the hardware accelerator connected to the AXI interconnect, called RVfpgaNexys-MMA.

The final sections of this chapter provide details of the simulations used to validate the functionality of RVfpgaNexys-MMA, and the description of how the accelerator is used in the software applications developed in Chapter 5.

4.1 Baseline SweRVolfX

4.1.1 SweRVolfX Description

The SweRVolf SoC is developed around the SweRV EH1 Core Complex (RV Core) [18]. The SweRVolf SoC also includes a Boot-ROM, a System Controller, a Serial Peripheral Interface (SPI) controller, and a Universal Asynchronous Receiver-Transmitter (UART). The SweRVolf SoC was extended [1] to add extra functionalities, such as another SPI controller, a PWM Timer Counter (PTC) module, a General Purpose Input/Output (GPIO) controller, and a controller for the 8-digit display included in the System Controller. The extended version, called SweRVolfX, is then targeted to the board Nexys A7 [19]. Figure 4.1 shows SweRVolfX and how the SoC interacts with some of the hardware present in the board.

The SweRV Core is an RV32I based implementation with the extensions C, for compressed instructions, and M, for multiplication and division that only uses machine mode. The RV Core uses a dual issue 9 stage pipeline with four Arithmetic Logic Unit (ALU) in two separate pipelines. The first pipeline supports loads and stores, and the other has a 3 cycle multiplayer. The processor also has a 34 cycle divider outside of the pipeline. Finally, it has four stall points, one in the fetch stage, the second in the align stage, the third in the decode stage, and the fourth in the commit stage.

Besides the CPU, the SweRVolf SoC includes the Boot-ROM, that contains the bootloader, the System Controller, that contains common system functionality such as keeping the SoC version information, the RAM initialization status, and the RISC-V machine timer. The additional functionalities of the controller provided by the extended version are a controller for the 8-digit display and 2 registers for handling interrupts from the PTC module and the GPIO. The 2 SPI controllers [20], the PTC module [21], the GPIO module [22] and the UART [23] are open-source IPs obtained from OpenCores [24].

AXI is a common bus protocol used in many processors. Both the RV Core and the RAM utilize AXI [16] to send and receive data. To connect the RV Core Complex to the RAM and the peripherals the SoC uses an AXI Interconnect crossbar. However, the peripherals use the Wishbone Architecture [25].

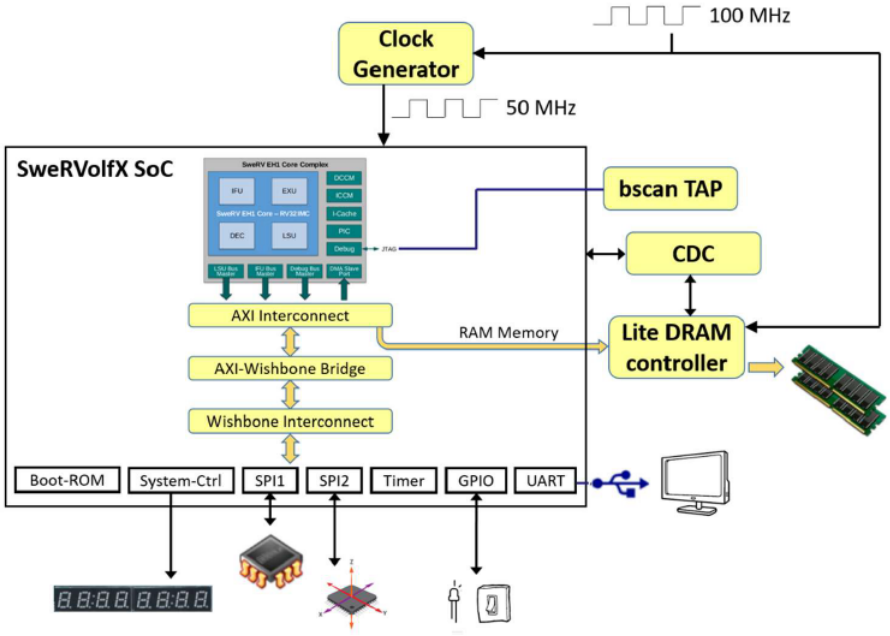


Figure 4.1: RVfpgaNexys block diagram (figure from [1])

System	Adress
RAM	0x00000000 - 0x07FFFFFF
Wishbone Interconnect	0x80000000 - 0x80003FFF

Table 4.1: AXI Interconnect Adress Map

The AXI Crossbar has 3 AXI-slave ports and 2 AXI-master ports. The slave ports are for the SweRV's data, instruction, and debug transfers. The master ports are for the RAM and to access the peripherals Wishbone interconnect. Table 4.1 shows the memory-mapped addresses of the master ports in the AXI interconnect.

An AXI to Wishbone bridge is used to connect the AXI interconnect to the Wishbone interconnect. The Wishbone interconnect is used to connect the RV Core to the peripherals. Table 4.2 shows the memory-mapped addresses of the peripherals connected to the RV Core by the Wishbone interconnect.

System	Adress
Boot ROM	0x80000000 - 0x80000FFF
System Controller	0x80001000 - 0x8000103F
SPI1	0x80001040 - 0x8000107F
SPI2	0x80001100 - 0x8000113F
PTC	0x80001200 - 0x8000123F
GPIO	0x80001400 - 0x8000143F
UART	0x80002000 - 0x80002FFF

Table 4.2: Wishbone Interconnect Adress Map

4.1.2 SweRVolfX Targeted to Nexys A7 FPGA

RVfpgaNexys is the SweRVolfX targeted to the Nexys A7 FPGA. Figure 4.1 illustrates how the SoC connects to other hardware components programmed onto the board, and the peripherals already present in the board. The hardware programmed includes the Clock Generator, the Boundary Scan logic for the JTAG interface, the Clock Domain Crossing (CDC) module and the Lite DRAM controller. The peripherals included in the board are the DDR2 memory, the USB connection, the 8-digit display, the SPI accelerometer, the SPI flash memory, the 16 LEDs and switches, and the USB connection. Figure 4.2 shows the resources used by the implementation of RVfpgaNexys synthesized with Vivado targeting the Nexys A7 board.

In Figure 4.2 is possible to see the total resources utilized by RVfpgaNexys as well as the resources of some of the components of the implementation, such as the SoC and the AXI interconnect.

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)
N rvfpganexys	34012	19106	461	34	10105	33755	257	44	4
cdc (axi_cdc_intf)	636	958	0	0	341	636	0	0	0
clk_gen (clk_gen_nexys)	5	2	0	0	5	5	0	0	0
ddr2 (litedram_top)	2733	2135	101	3	943	2512	221	13	0
swervolf (swervolf_core)	30465	15874	360	31	9081	30429	36	31	4
axi2wb (axi2wb)	377	176	2	0	240	377	0	0	0
axi_intercon (axi_interc	1367	1265	6	0	523	1367	0	0	0
bootrom (wb_mem_wra	0	1	0	0	1	0	0	1	0
gpio_module (gpio_top	249	420	0	0	174	249	0	0	0
spi (simple_spi)	63	69	0	0	32	51	12	0	0
spi2 (simple_spi_0)	62	68	0	0	30	50	12	0	0
swerv_eh1 (swerv_wrap	27824	13173	337	28	8137	27824	0	30	4
syscon (swervolf_syscor	136	299	4	0	112	136	0	0	0
timer_ptc (ptc_top)	93	106	0	0	53	93	0	0	0
uart16550_0 (uart_top)	354	296	11	3	125	342	12	0	0
wb_intercon0 (wb_inter	0	1	0	0	1	0	0	0	0
tap (bscan_tap)	175	121	0	0	90	175	0	0	0

Figure 4.2: Baseline RVfpgaNexys Resource Utilization

4.2 RVfpgaNexys with Support for an AXI Hardware Accelerator

The extension of SweRVolfX to contain a hardware accelerator started with the implementation of the necessary ports for the communication between the CPU and the accelerator, and between the accelerator and the RAM. There are two possible scenarios, using the AXI interconnect or using the Wishbone interconnect.

AXI is an industry-standard protocol with support for advanced features such as burst transfers, that allows for a high data throughput and a low latency. Wishbone is an open-source bus architecture that is flexible and lightweight, with low complexity, which is easy to use. As presented in section 3.1 the

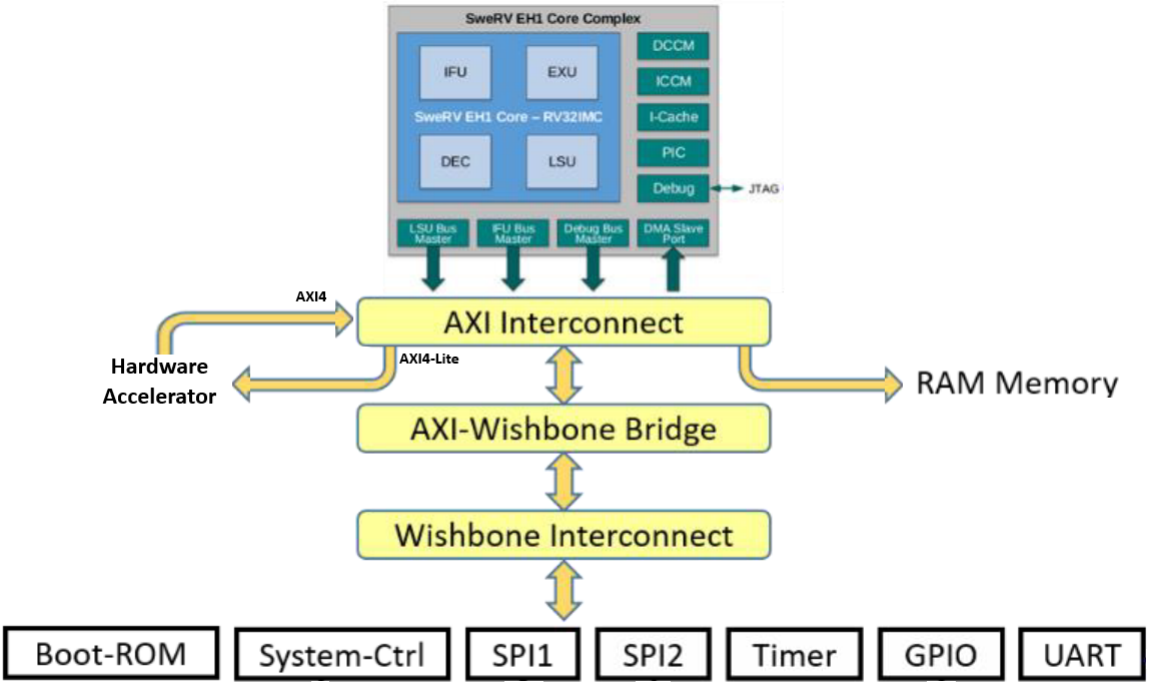


Figure 4.3: SweRVolfX with a Hardware Accelerator

developed matrix multiplication hardware accelerator requires communication with both the CPU and the RAM. So using the Wishbone bus requires the data to pass through the AXI interconnect and the AXI-Wishbone bridge for each transfer, adding overhead. To reduce the overhead of the communication between the CPU and the accelerator, and the accelerator and the RAM, the hardware accelerator was connected to the board using the AXI interconnect.

Figure 4.3 shows the SweRVolfX SoC with the added connections and protocols used for the accelerator. The AXI4-Lite connection allows the CPU to send the control signals to the accelerator and read the status of the accelerator. The AXI4 connection is for the accelerator to read and write into memory.

Autogenerated AXI Interconnect

The AXI interconnect used in the SweRVolfX design was autogenerated by a tool that wasn't available in the SoC description. Since the AXI interconnect files were autogenerated, simply adding the extra ports to the source files is not an option. After inquiring in the official forums of RVfpgaNexys [26], the tool used to generate the interconnect could not be found.

The next option was to find a different autogenerator to generate the AXI interconnect description with the added ports. The autogenerator found in [24] should be able to meet the design requirements. However, it uses a tool that is no longer maintained and could not be installed. There were no other open-source and easy-to-use AXI interconnect generators available. As a result, the alternative was to

stop using automatic generators and use an interconnect developed for another project and adapt it to the current use case.

Axi Interconnect Adapted From Open Source IOb-SoC

IObundle [27] is developing an open-source RISC-V based SoC, IOb-SoC [28]. The IOb-SoC also uses AXI to pass data between the CPU and the peripherals, using an AXI interconnect for this purpose. Using IOb-SoCs interconnect was a promising option, however, the adaptation of the interconnect is a more complicated task than anticipated. Some complications arose, such as not being able to input the address map of the peripherals, not having some signals used by the CPU, and being unable to customize the data signal size to match the CPU and peripherals. This attempt revealed that the adaptation of source files from other projects would use more time than is available and, therefore, is not a viable option.

Xilinx's AXI Interconnect

The final option is the use Xilinx's AXI interconnect [29] that is bundled with Vivado Design Suite. This option prevents the simulation of the system with tools outside of Xilinx's design suit. While Vivado's simulation tool is used to verify the functionality of the SoC, the analysis in this work can still be done without the use of simulation.

Xilinx's AXI Interconnect IP connects up to 16 AXI memory-mapped master devices to up to 16 memory-mapped slave devices. The AXI interfaces conform to the AXI4, AXI4-Lite or AXI3 standards, allowing different standards for each port, depending on the master or slave the port connects to. The IP allows two different architectures, crossbar mode and shared access mode. Crossbar mode uses a crossbar architecture with parallel pathways for read and write channels. This architecture allows for all the slaves to connect to all the masters and vice-versa. The parallel pathways allow for simultaneous access to different connections, for example, the CPU data channel can read the values of the switches and the CPU instruction fetch channel can read the next instruction from memory simultaneously. Shared access mode, uses a shared address pathway. Shared access mode is lighter and uses fewer FPGA resources. Since the board still has plenty of available resources (see Figure 4.2), the performance-optimized mode is the better option. Therefore, the crossbar mode will be selected for the implementation.

Figure 4.4 shows the AXI interconnect IP with the ports to replace the AXI interconnect of the SweRVolfX SoC. The master ports are ifu, an instruction fetch for the CPU, lsu, a data transfer connection for the CPU, and sb, a debug channel for the CPU. The slave ports are io, the connection to the wishbone interconnect that interacts with all the IO peripherals, and ram, a connection to the Lite DRAM controller

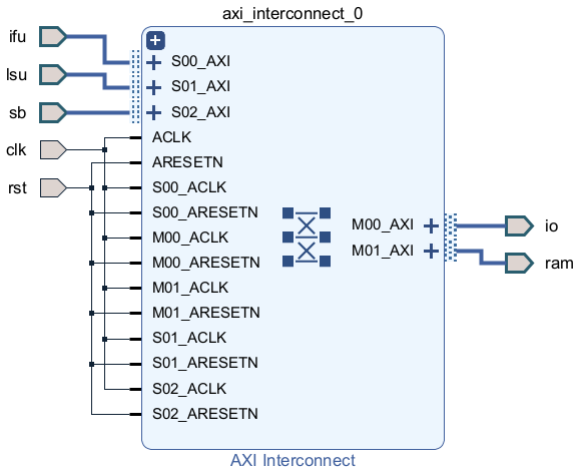


Figure 4.4: AXI Interconnect Block Design

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)
N rvfpganexys	34178	20374	455	34	10084	33728	450	50	4
cdc (axi_cdc_intf)	434	1038	0	0	281	434	0	0	0
clk_gen (clk_gen_nexys)	5	2	0	0	6	5	0	0	0
ddr2 (litedram_top)	2765	2155	101	3	962	2544	221	13	0
swervolf (swervolf_core)	30799	17042	354	31	9001	30570	229	37	4
axi2wb (axi2wb)	354	178	2	0	201	354	0	0	0
bootrom (wb_mem_wra	0	1	0	0	1	0	0	1	0
gpio_module (gpio_top)	249	420	0	0	186	249	0	0	0
interconnect_wrapper (i	1927	2428	0	0	848	1734	193	6	0
spi (simple_spi)	63	69	0	0	30	51	12	0	0
spi2 (simple_spi_0)	62	68	0	0	28	50	12	0	0
swerv_eh1 (swerv_wrap	27562	13176	337	28	7870	27562	0	30	4
syscon (swervolf_sysco	136	299	4	0	109	136	0	0	0
timer_ptc (ptc_top)	93	106	0	0	46	93	0	0	0
uart16550_0 (uart_top)	354	296	11	3	118	342	12	0	0
wb_intercon0 (wb_inter	0	1	0	0	1	0	0	0	0
tap (bscan_tap)	175	121	0	0	99	175	0	0	0

Figure 4.5: RVfpgaNexys with Xilinx's AXI Interconnect Resource Utilization

that reads and writes to memory. Finally, the clk and rst wires are the clock and reset signals necessary for the correct operation of the IP.

The AXI interconnect depicted in Figure 4.4 is generated inside a block design wrapper. The wrapper is an interface that connects the SoC wires to the wrapper's external wires, ifu, lsu, sb, clk, rst, io and ram. Inside the block design wrapper, it's also necessary to define the addresses of the master interfaces. In this case, the addresses are defined as shown in Figure 4.1.

In Figure 4.5 is possible to see the total resources utilized by RVfpgaNexys with Xilinx's AXI Interconnect. It is also possible to see that the resources used in this implementation are very similar to the resources used in the baseline RVfpgaNexys.

The verification of the correct functionality of the design was done using three programs. The first

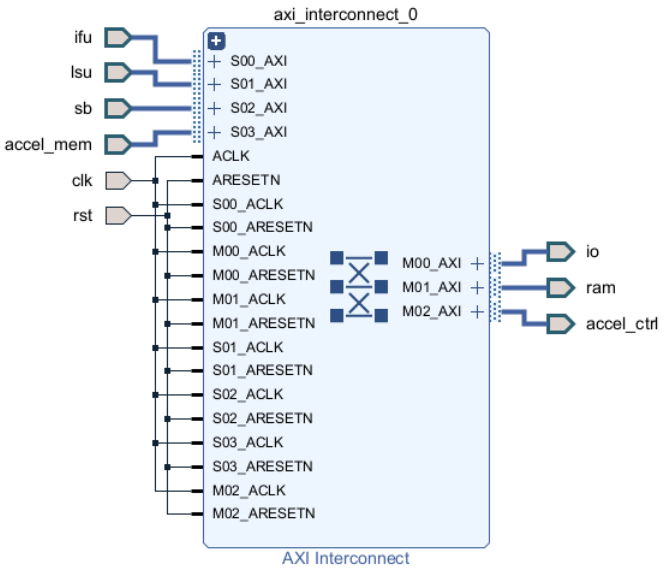


Figure 4.6: Extended AXI Interconnect Block Design

System	Address
RAM	0x00000000 - 0x07FFFFFF
Accelerator	0x90000000 - 0x90003FFF
Wishbone Interconnect	0x80000000 - 0x80003FFF

Table 4.3: Extended AXI Interconnect Address Map

one is a basic Hello World program to validate the comunication using the instruction fetch channel and the io channel. The second program reads the values of the board's switches and turns on the corresponding LEDs, validating the communication through the Wishbone channel. The final program used was the CoreMark benchmark [30] that tests the connections to memory. Since all three programs ran as expected the development continued using Xilinx's AXI interconnect IP.

To connect the accelerator it's necessary to add the connections shown in Figure 4.3. Figure 4.6 shows the AXI interconnect with the two extra ports necessary to communicate to and from the accelerator. The port *accle_ctrl* is an AXI connection to read and write control signals in the accelerators registers, and the port *accel_mem* is an AXI connection for memory access by the accelerator to read and write data.

To access the accelerator is necessary to change the address map of the interconnect. Table 4.3 shows the used memory-mapped addresses of the master ports in the new AXI interconnect.

After synthesising this version of RVfpgaNexys, the total resources used are showed in Figure 4.7. The extended AXI interconnect increased significantly the total resources utilized, since each new port of the interconnect needs to connect to all the others.

The same three programs were used to test this implementation. All programs produced the expected

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)
rvfpganexys	35337	21469	471	34	9961	34724	613	51	4
cdc (axi_cdc_intf)	435	1038	0	0	348	435	0	0	0
clk_gen (clk_gen_nexys)	5	2	0	0	5	5	0	0	0
ddr2 (litedram_top)	2757	2155	101	3	957	2536	221	13	0
swervolf (swervolf_core)	31967	18137	370	31	9008	31575	392	38	4
axi2wb (axi2wb)	355	178	2	0	209	355	0	0	0
bootrom (wb_mem_wra	0	1	0	0	1	0	0	1	0
gpio_module (gpio_top)	247	420	0	0	159	247	0	0	0
intercon (interconnect_w	3083	3523	16	0	1320	2727	356	7	0
spi (simple_spi)	63	69	0	0	30	51	12	0	0
spi2 (simple_spi_0)	62	68	0	0	28	50	12	0	0
swerv_eh1 (swerv_wrap	27576	13176	337	28	7640	27576	0	30	4
syscon (swervolf_sysco	136	299	4	0	105	136	0	0	0
timer_ptc (ptc_top)	93	106	0	0	51	93	0	0	0
uart16550_0 (uart_top)	354	296	11	3	110	342	12	0	0
wb_intercon0 (wb_interc	0	1	0	0	1	0	0	0	0
tap (bscan_tap)	175	121	0	0	115	175	0	0	0

Figure 4.7: RVfpgaNexys with Extended AXI Interconnect Resource Utilization

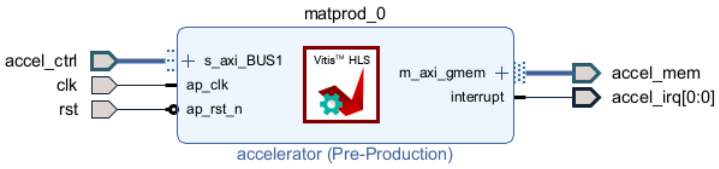


Figure 4.8: Hardware Accelerator Block Design

output, so the development continued into the design and implementation of the hardware accelerator.

4.3 RVfpgaNexys with Matrix Multiplication Accelerator - RVfpgaNexys-MMA

4.3.1 Final RVfpgaNexys Implementation

The hardware accelerator designed in Chapter 3 is integrated into the SoC with a separate Vivado block design that connects the ports added to the AXI interconnect. Figure 4.8 shows the block diagram of the accelerators block design generated by Vivado.

In Figure 4.8 there are five external interfaces connected to the accelerator. The *clk* and *rst* ports are the clock and reset signals. The *accel_ctrl* port connects to the port of the same name in the interconnect, and is the AXI4 Lite connection that writes and reads the control signals that manage the accelerator. The port *accel_mem* connects to the port of the same name in the interconnect, and is the AXI4 connection for the accelerator to read and write data in the RAM. The port *accel_irq* connects to the PIC, and sends the interrupt request from the accelerator once the operation is completed.

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)
N rvfpganexys	37905	25014	505	34	11694	37155	750	56	24
cdc (axi_cdc_intf)	434	1038	0	0	275	434	0	0	0
clk_gen (clk_gen_nexys)	5	2	0	0	6	5	0	0	0
ddr2 (litedram_top)	2773	2155	101	3	902	2552	221	13	0
swervolf (swervolf_core)	34520	21682	404	31	10669	33991	529	43	24
accelerator (accel_wrap	1828	2801	32	0	880	1719	109	4	20
axi2wb (axi2wb)	355	178	2	0	214	355	0	0	0
bootrom (wb_mem_wra	0	1	0	0	1	0	0	1	0
gpio_module (gpio_top)	249	420	0	0	159	249	0	0	0
intercon (interconnect_w	3832	4267	18	0	1443	3448	384	8	0
spi (simple_spi)	63	69	0	0	28	51	12	0	0
spi2 (simple_spi_0)	62	68	0	0	29	50	12	0	0
swerv_eh1 (swerv_wrap	27550	13176	337	28	8149	27550	0	30	4
syscon (swervolf_sysco	134	299	4	0	110	134	0	0	0
timer_ptc (ptc_top)	93	106	0	0	47	93	0	0	0
uart16550_0 (uart_top)	354	296	11	3	107	342	12	0	0
wb_intercon0 (wb_interc	0	1	0	0	1	0	0	0	0
tap (bscan_tap)	175	121	0	0	91	175	0	0	0

Figure 4.9: RVfpgaNexys-MMA Resource Utilization.

The complete design of RVfpgaNexys-MMA was synthesized and the bitstream for this implementation was generated. Figure 4.9 shows the resource utilization of the final RVfpgaNexys-MMA implementation. In the Figure it is possible to see that the resources utilized by the accelerator are similar to the estimation in Table 3.2.

4.3.2 Verification of the Final SoC Design

Three steps were taken to verify the correct implementation of the accelerator. The first one was to run the same three programs as in the previous SoC implementations. These simple programs confirmed that the introduction of the accelerator did not compromise the baseline SoC functionality. The two other verification steps were done using Vivado's simulation tool, and a software testbench to verify the functionality of the hardware accelerator.

Verification with Vivado Simulation

Using Vivado simulation, the accelerator was tested independently and in the context of the RVfpgaNexys-MMA SoC. In both of these situations, the AXI verification IP (VIP) core [31] was used as a way to give commands to the accelerator. The VIP can be used to verify the functionality of AXI masters and slaves.

The first simulation was used to verify the connections of the hardware accelerator IP imported to Vivado from Vitis HLS. Therefore, the connection between the accelerator and the CPU was replaced by Vivados Verification IP (VIP), and the connection between the accelerator and the RAM was replaced by an AXI BRAM controller [32] with a block memory generator [33]. The testbench used writes the matrix sizes and addresses in the accelerators assigned registers and then sets the control port to start the

operation in the accelerator. The control port is read after 10 milliseconds to verify the completion of the operation.

Figure 4.10 shows the waveform of the initial write transactions, done in the simulation from the VIP to the accelerator. The transactions tagged with the numbers 2, 3 and 4 show that the memory addresses of the matrices are being written, and transaction 5 shows that the number of rows of the first matrix is being written. The transactions continue to be done as expected, and, when the accelerator has completed the multiplication, the control signal changes from 0x01 to 0x0e. Since the control register of the accelerator confirms the completion of the operation, the communication with the BRAM controller are correct, otherwise the accelerator would have stopped.

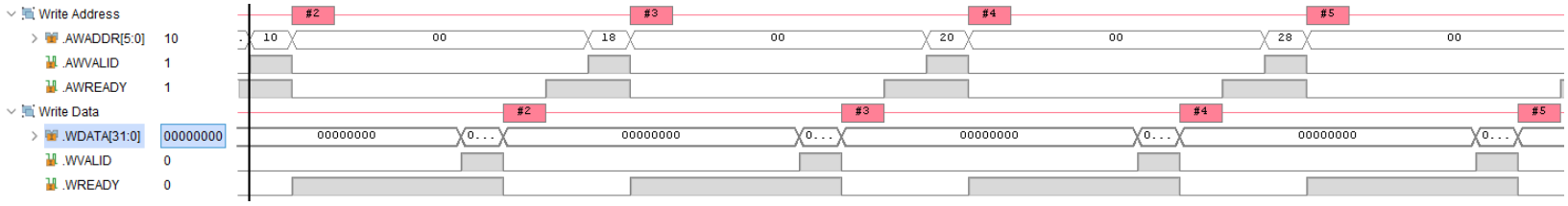


Figure 4.10: Waveform of the first transactions to the accelerator

The purpose of the second simulation was to simulate the interaction between the CPU and the accelerator through the AXI interconnect. This ensures that the crossbar is sending the signals to the correct addresses and that the accelerator is correctly reading the offsets from the full address used in the CPU.

In a real scenario the CPU reads and writes in the accelerator, however, for simplicity, a VIP is connected to the interconnect in place of the CPU. The VIP is programmed in a similar way as the VIP used in the previous simulation. All the transactions performed as expected, and its possible to verify the correct values being written in the accelerator. In this simulation, the accelerator does not change the values of the control signal to 0x0e, since the memory is not initialized and the accelerator is unable to read the necessary data to continue the execution. However, it is possible to verify that the accelerator sends the read request to the RAM.

Verification with a Software Application

The final verification to the RVfpgaNexys-MMA implementation is a software application that generates two matrices and uses the accelerator to multiply them. This application correctly initializes the memory addresses in the RAM and writes the matrix addresses and sizes according to the offsets in Table 3.1.

Listing 4.1 shows the code used to the write the necessary inputs in the accelerator and to start its execution. The first 7 lines in the listing show how pointers are created to write in the accelerators registers, according to the values in Tables 3.1 and Table 4.3. Lines 9 to 14 are the instructions that write the matrix addresses and sizes in the accelerators registers. Line 16 changes the accelerators control signal to 0x01 to start performing the operation. Line 18 is a read instruction inside a *while* loop to constantly check the value of the control signal until the multiplication operation is completed and its value changes to done. Once the results are written in the memory its possible to verify that they are correct.

The tool used to verify the values written in the accelerator and in the RAM is GDB, or the GNU Debugger, which is an open-source debugger used for programming and debugging software. It allows developers to inspect and manipulate the execution of programs, helping to identify and fix errors in the code. GDB allows developers to read the values stored in specific addresses. This feature was used to verify that the matrix addresses and their sizes were being written in the correct accelerator registers, and that the output matrix was being written in the correct RAM address.

Listing 4.1: Software instructions used to validate the hardware accelerator

```
volatile int *do_matp_mem = (int *)0x90000000;
volatile int *a = (int *)0x90000010;
volatile int *b = (int *)0x90000018;
volatile int *c = (int *)0x90000020;
volatile int *rowsA = (int *)0x90000028;
volatile int *colsA = (int *)0x90000030;
volatile int *colsB = (int *)0x90000038;

*a = MATA_START_ADDRESS;
*b = MATB_START_ADDRESS;
*c = MATC_START_ADDRESS;
*rowsA = N1;
*colsA = N2;
*colsB = N3;

*do_matp_mem = 1;

while ((*do_matp_mem & 4) == 0);
```

5

Hardware Accelerator Management Using Multi-threading RTOS Mechanisms

Contents

This chapter describes the three proposed scenarios to use the hardware accelerator. The first scenario is to use the processor's and Zephyr's interrupt management mechanisms to free the processor while the accelerator is being used. The second scenario is to use multiple threads to interact with the accelerator. With this scenario the processor and the accelerator are being used simultaneously. The third scenario is to have a single thread responsible for using the accelerator. The other threads send messages to the first thread with the necessary values.

Two use cases are used to evaluate these three different scenarios. The first use case performs a simple multiplication of sets of matrices. The sets of matrices pairs are pre-generated in the processor to reduce the instructions that are not related to the communication with the accelerator. This makes the overhead added by the RTOS mechanisms more evident. The second use case implements a neural network to perform the prediction of several digits from the MNIST dataset [34]. This is a more realistic scenario, and the additional instructions between matrix multiplication can help hide the overhead added by the RTOS mechanisms used.

The three scenarios pictured below expand upon the preceding one. Therefore, each section focuses only on the changes introduced by each scenario.

5.1 Single Thread With Interrupt Signals

The initial scenario is to use a single thread to perform the designated use case. During a multiplication operation with the accelerator, the CPU is not constantly accessing the accelerator's control signal. Instead, the application enters a semaphore that is locked until the ISR is triggered by the accelerator and unlocks that semaphore.

To use the interrupt generated by the accelerator, it is necessary to link a routine to it. Zephyr defines that an hardware interrupt must be associated to the IRQ 11 or above. In the SoCs interrupt controller, the accelerators interrupt is connected to bit 5. Taking into consideration the other interrupt sources, the accelerators IRQ number is 15.

Figure 5.1 shows how the ISR is programmed into the interrupt request table of Zephyr. The first two instructions enable the interrupt signal on the accelerator, using pointers to the addresses specified in table 3.3. The third instruction connects the accelerator IRQ to the ISR, in this case, called the *my_isr* function, and associates a priority to the interrupt. The last instruction enables the interrupt to be generated by Zpehyr after being triggered by the accelerator.

Figure 5.2 depicts the ISR triggered by an interrupt generated from the hardware accelerator. The first line in the ISR toggles the accelerator's interrupt status, informing it that the interrupt is being processed and signaling it to stop sending the interrupt request to the processor. The second instruction of the ISR is to signal the thread that called the accelerator. That thread was locked with the semaphore

Listing 5.1: ISR installer code

```
1 void my_isr_installer(void) {
2     *acceleratorGIER = 0x1;
3     *acceleratorIP_IER = 0x1;
4     ARCH_IRQ_CONNECT(ACCEL_IRQ, ACCEL_PRIO, my_isr, NULL, 0);
5     arch_irq_enable(ACCEL_IRQ);
6 }
```

accel_sem waiting for the accelerator's completion. When the semaphore unlocks the thread can proceed.

Listing 5.2: ISR code

```
1 void my_isr(const void *arg) {
2     *acceleratorIP_ISR = 0x1;
3     k_sem_give(&accel_sem);
4 }
```

Figure 5.1 shows the flow chart of the thread that is using the accelerator. The thread starts by preparing the matrix multiplication operation in the processor. In the matrix pair multiplication use case it is necessary to get the correct memory addresses for the new matrix pair. In the neural network use case there are more complex operations such as the ReLU or the Softmax operation. In the neural network example some of the matrices have more elements then the accelerator allows. In these cases, the preparation step also involves dividing the matrix to fit the accelerator maximum size.

After the inputs are prepared, the thread sends them to the accelerator. Since the Accelerator Semaphore is initialized in the not available state, the thread is unable to take it and goes to the blocked state. The ISR unlocks the semaphore when the processor receives the interrupt request, changing the threads state from blocked to ready.

When the scheduler awakes the thread the first instruction is to take the Accelerator Semaphore so that no proceeding accelerator calls can proceed without their results. Finally, the thread processes the result in the processor.

5.2 Multiple Threads Competing for Accelerator Access

In this scenario, multiple threads compete for access to the accelerator at the same time. The main concern introduced with this scenario is that a thread may try to use the accelerator while it is already in use. One way to solve this is issue to introduce a second semaphore. This semaphore is used by

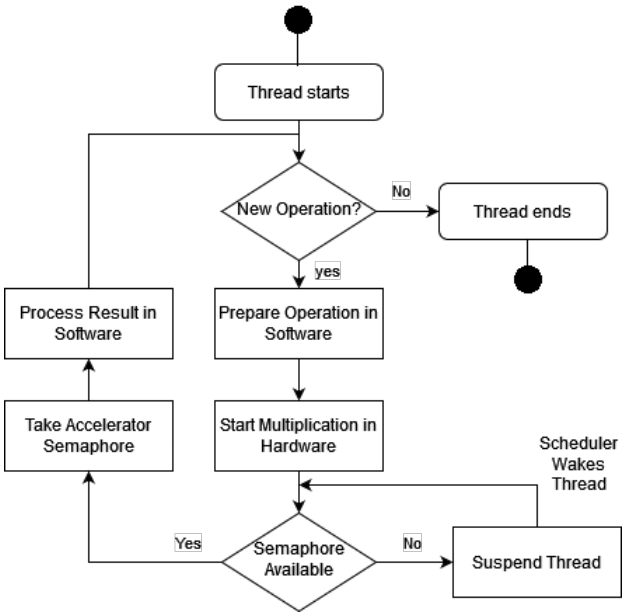


Figure 5.1: Single thread accessing the accelerator flow chart

the different threads to check if another thread is using the accelerator. Before using the accelerator, the new semaphore is locked. Consequently, the other threads have to wait for the semaphore to be unlocked. The semaphore is unlocked by the ISR, and when the next thread starts executing, it locks the semaphore, ensuring that no other thread can use the accelerator before the current one finished the operation.

Figure 5.2 illustrates the flow chart of each thread. After the inputs are prepared, the thread tries to take the Thread Semaphore. If the semaphore is not available, the thread suspends and enters the blocked state. If the semaphore is available, no other threads are using the accelerator, and the thread may take the Thread Semaphore and proceed. Finally the input matrices are sent to the accelerator and the Accelerator Semaphore is used in the same way as in the previous section.

When the ISR is triggered by the accelerators interrupt the thread changes to the ready state and the scheduler wakes the thread to process the results. The ISR also unlocks the Thread Semaphore, allowing one thread that was previously blocked to go to the ready state.

In this scenario there may be multiple threads waiting in the blocked state for the semaphores to be unlocked. The scheduler may wake a thread that is trying to use the accelerator before waking the thread that has already used the accelerator. Because of this, in the matrix multiplication use case, all matrix pairs and results have different predefined addresses. In the neural network use case, all threads have different address ranges to put the inputs and results of each operation.

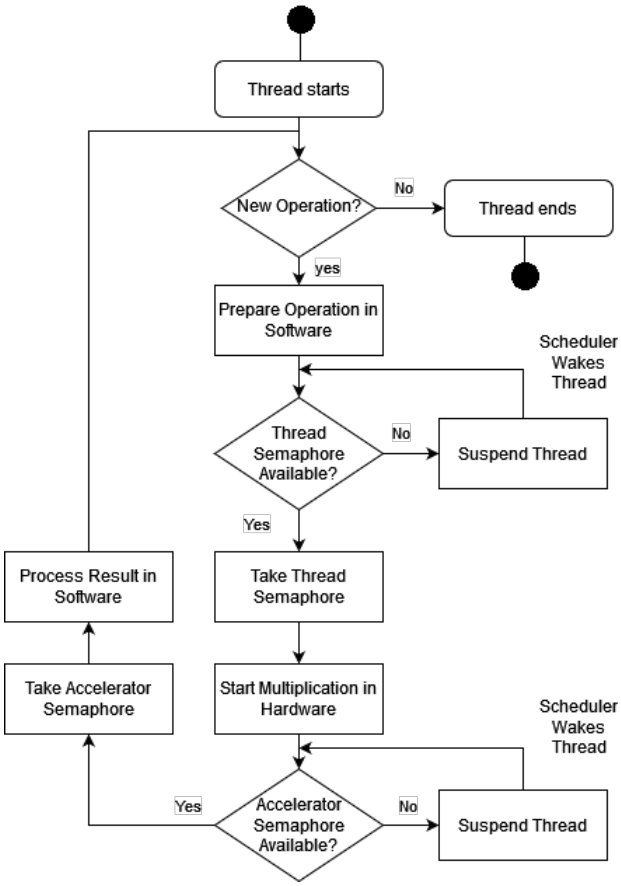


Figure 5.2: Multiple threads competing for accelerator access flow chart

5.3 Centralized Control of Accelerator Access

The final scenario is to have one thread responsible for managing the accelerator. This Manager Thread receives requests from other threads when they need to use the accelerator. The Manager Thread has a higher priority then the threads that make the requests to minimize the requests in the queue at a time.

In this scenario the Manager Thread receives communicates with the other threads using two message queues. The first message queue receives the requests, and the second message queue sends the replies. When a thread has a multiplication operation and needs the accelerator, it puts a message in the queue. That thread then waits for a reply in the reply queue. Each message consists of the addresses used by the accelerator, the matrices size, and the ID of the thread that sent the request. The ID is used to ensure that the threads only take their replies from the queue.

Figure 5.3 shows the flow chart of the Manager Thread. The thread is in a loop that checks if there are any messages in the queue. If there aren't it is suspended until a new messages is added to the queue and the scheduler wakes the thread. When the new message arrives the thread uses the accelerator in the same way as in Section 5.1. When the thread starts executing after using the accelerator, a new message is sent to the reply queue.

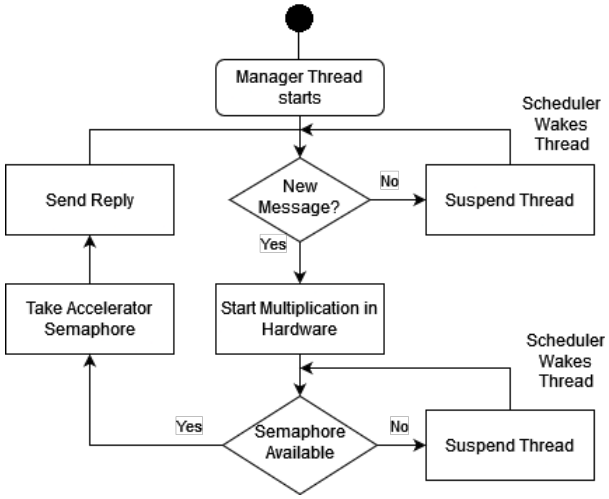


Figure 5.3: Manager thread flow chart

Figure 5.4 illustrates how the other threads use the Manager Thread to perform multiplication using the accelerator. The thread creates the message to be sent to the Manager Thread and they put it in the requests queue. After sending the request they check the reply queue for new messages. In the reply queue there may be replies to other threads, so the have to check if the ID of the message is the same as their own ID. If the IDs match, the message is removed from the queue and the results are processed. If the IDs don't match or the queue is empty, the thread goes to the blocked state allowing for other threads to run.

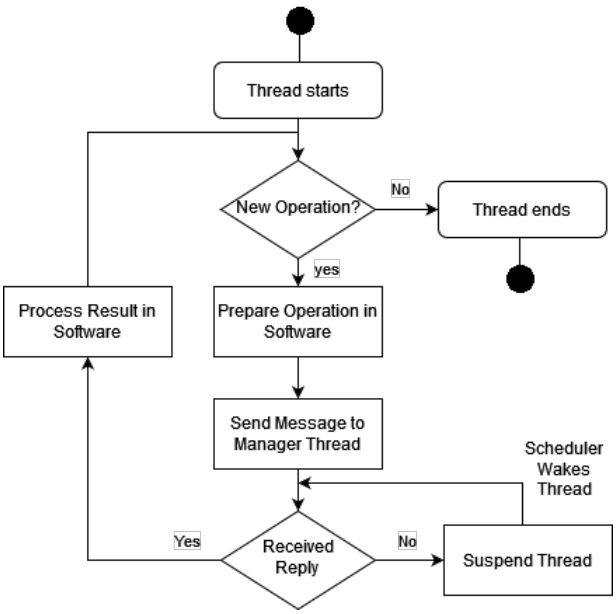


Figure 5.4: Thread sending requests to Manager Thread flow chart

6

Performance Assessment of the RTOS Management Mechanisms on an Accelerator

Contents

This chapter provides the performance assessment of the different scenarios defined in Chapter 5. This study was done in the environment described in Appendix A. The Vivado tool was used to generate the bitstream, targeting the Nexyz A7 board, of the hardware implementation defined in Chapter 4. And application code was generated using the Zephyr toolchain.

The evaluation done in this chapter focuses on the execution time and current consumption of the board on each scenario. To do so, four applications were developed for each use case. The first application performs the operations without the RTOS capabilities, both using the accelerator and without using it. The second application performs the multiplication operations with the use of interrupts as described in Section 5.1. The third application performs the operations using the approach described in Section 5.2, with 1, 2, 4, 16, 32 and 64 threads competing for the use of the accelerator. The fourth application, described in Section 5.3, uses 1, 2, 4, 16, 32 and 64 threads that communicate with a single accelerator manager thread.

The current consumption of the applications was measured using RUI DENG's USB Color Display Tester model UM24C [35]. The device was connected to the source device and to the Nexys A7 board to measure the current used by the board, and has a current measurement resolution of 1 mA. When the SoC implementation is programmed into the FPGA the USB tester measures a constant base current consumption of 409 mA. The execution time of each scenario was measured using the the Zephyr function $\mathrm{k\_uptime\_get}$, which returns the systems uptime in milliseconds. While this routine returns time in milliseconds, the precision is only 10 milliseconds.

6.1 Matrix Multiplication Acceleration Without RTOS Mechanisms

To obtain basic reference metric values the selected the application performs specific use cases without the RTOS. The first metric values are determined when the application executes on RISC-V using only software instructions (without the hardware accelerator). The second metric values, the baseline metrics, the application uses the hardware accelerator to aid the processor employing a technique called pooling. In this situation the processor is actively waiting for the result from the accelerator by constantly checking the control register of the accelerator.

Multiplication of Set of Matrices

In the matrix multiplication use case, 100 sets of matrices were multiplied. Each matrix has 32 rows and 32 columns, creating a result matrix of the same size. The multiplication of the 100 sets of matrices took 158.770 seconds using only the processor and 200 milliseconds when introducing the accelerator. In terms of current consumption, both the software only execution and the execution with the accelerator used 413 mA.

These results show how an hardware accelerator can accelerate this task, achieving a speedup of over 700, without increasing the current consumption of the SoC. To reach more meaningful conclusions in the following sections when comparing the different scenarios the performance of 512 multiplication was also tested and took 1.140 seconds, with the same consumption of 413 mA.

Neural Network

In the feed forward pass of the neural network, 50 digits were initially predicted. The neural network has 3 hidden layers with 128, 64 and 10 neurons and each digit has 784 pixels. This translates to three different matrices multiplications per digit. However, since the first two weight matrices are larger then the maximum allowed size of the accelerator, which is 1024, the multiplication was separated per column. In the end, 193 matrix multiplications per digit were performed when using the accelerator. In these conditions, the prediction of the 50 digits took 232.030 seconds and used 415 mA, using only the processor, and took 8.810 seconds and used 415 mA with the accelerator.

Since the neural network use case has a significant amount of software instructions in relation to multiplication operations, the speedup achieved was 18.33, a lower value as expected by the Amdahl's law. At the same time, the software only execution does not have any restraints on the size of the matrices, and, therefore, has fewer instructions before performing the multiplication. Similarly to the first use case, to reach more meaningful conclusions in the following sections, the number of digits tested with the accelerator using pooling was increased to 256, taking 45.140 seconds and using 415 mA.

6.2 Single Thread With Interrupt Signals

This scenario introduces the RTOS and its multitasking mechanisms, such as interrupts and semaphores. The introduction of interrupts allows the processor to perform other tasks while the accelerator is being used. In this scenario, the processor stays in a waiting state, which is similar to the baseline application, however, the processor is not being used to check if the accelerator has completed execution. When the hardware accelerator completes the operation, it signals the processor through an interrupt.

Multiplication of Set of Matrices

The multiplication of 512 sets of matrices took 1.200 seconds. There is an increase of 60 milliseconds, when comparing this time to the baseline application with the same number of matrix sets. This increase corresponds to a 0.12 milliseconds increment per multiplication. This increment is expected because switching contexts with every interrupt adds some overhead.

Threads	1	2	4	8	16	32	64
Time (s)	1.21	1.23	1.23	1.23	1.23	1.23	1.24

Table 6.1: Time measurements of the multiplication of 512 matrices using the second scenario

Neural Network

The execution of the neural network in this scenario for the prediction of 256 digits took 52.640 seconds. When comparing the execution time of this scenario to the baseline there is a 7.620 seconds increase, which corresponds to a 0.15 millisecond increment per multiplication.

The introduction of the RTOS and the interrupt mechanism added some overhead in both use cases, however it allows the processor to be used in other tasks. In terms of current consumption, the USB tester displayed a similar value (413 mA in the first use case and 415 mA in the second use case) in all the following scenarios, having a variation of $\pm$ 1 mA while performing the tests. This variation corresponds to 0.25% of the base current of the SoC, meaning that the different scenarios do not significantly change the power consumption of the system.

6.3 Multiple Threads Competing for Accelerator Access

The second scenario introduces several threads that use the accelerator. A semaphore is used to manage the access to accelerator, ensuring that only one thread is performing a multiplication at a time.

Multiplication of Set of Matrices

Table 6.1 shows the times of the multiplication of 512 sets of matrices when using 1, 2, 4, 8, 16, 32, and 64 threads trying to access the accelerator. The 512 sets where divided uniformly among the different threads. Comparing the time performance to the previous scenario, there is an overhead of 20 upto 40 millisecond. Figure 6.1 shows that the time performance is not significantly affected by the number of threads.

Neural Network

Table 6.2 shows the times of the prediction of 256 digits using a neural network when using 1, 2, 4, 8, 16, 32, and 64 threads trying to access the accelerator. When using only one thread the application works similarly to the previous scenario, and the difference in the time measurements comes from the addition of the new semaphore.

Figure 6.2 shows how the time performance is affected by the number of threads used. It's possible to see that for 1, 2, 4 and 8 threads the time is consistent, however for 16 and 32 threads the time

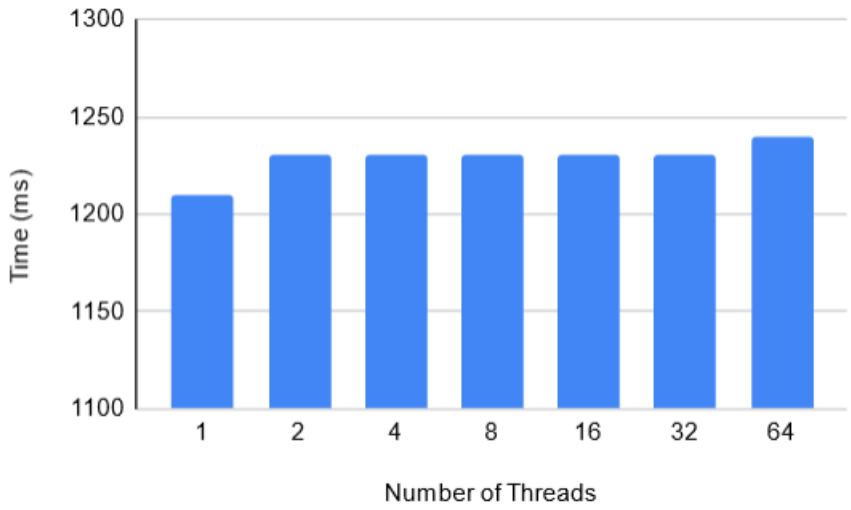


Figure 6.1: Time measurements of the multiplication of 512 matrices using the second scenario

Threads	1	2	4	8	16	32	64
Time (s)	54.32	54.30	54.11	54.12	55.37	56.88	56.99

Table 6.2: Time measurements of the prediction of 256 digits using the second scenario

increases reaching a plateau at 32 threads. One possible explanation for this behavior is that the cache is filled with the stacks of the threads, and having cache misses increases the duration of the context switch, however the CPU does not have a data cache, so this explanation is invalid. Due to the time limitations of this work it was not possible to further investigate the causes of this behavior.

The additional semaphore allows for different tasks to use the accelerator concurrently. This adds overhead seen by the time increase from the previous scenario to this scenario while using only one thread. In the first use case there is an increase of 10 millisecond and in the neural network there is an increase of 1.680 seconds. The current consumption measured with the USB tester remained similar to the previous scenario, being 413 mA on average in the multiplication of sets of matrices and 415 mA on average in the prediction of the MNIST digits.

6.4 Centralized Control of Accelerator Access

The third scenario introduces a thread responsible for managing the use of the accelerator. This addition removes the necessity of the extra semaphore added in the previous scenario, but introduces message queues to communicate between threads. The message queues allow for a precise control of witch thread is accessing the accelerator, since the manager thread has access to the requesting thread's id.

The utilization of the message queues also allow to queue requests for the accelerator as a substitute

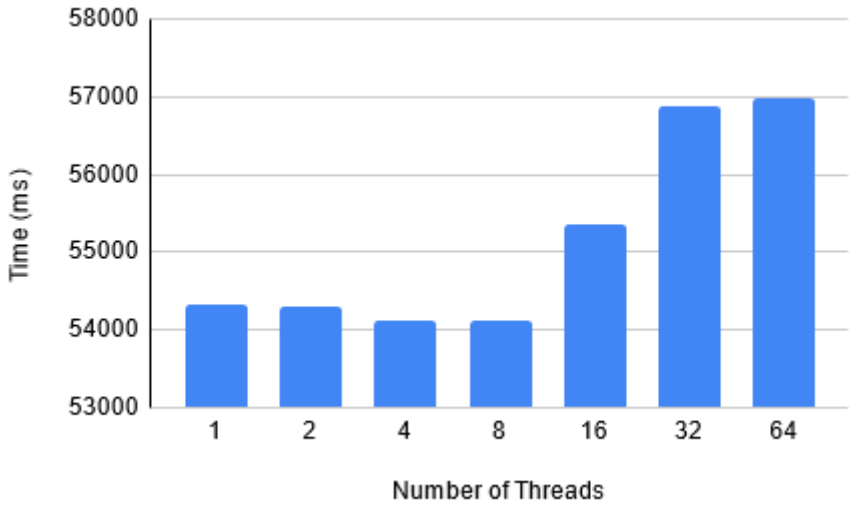


Figure 6.2: Time measurements of the prediction of 256 digits using the second scenario

Threads	1	2	4	8	16	32	64
Time (s)	1.43	1.28	1.28	1.28	1.28	1.28	1.31

Table 6.3: Time measurements of the multiplication of 512 matrices using the third scenario

to immediately trying to access the accelerator. This reduces the number of times a thread is suspended when a request is made. In the previous scenario each thread is suspended before unsuccessfully trying to start the accelerator and after starting the accelerator when its waiting for the results. In this scenario, the threads only suspend after sending the request to the manager thread, and they start executing when the manager thread sends the reply.

Multiplication of Set of Matrices

Table 6.3 presents the execution times of the multiplication of 512 sets of matrices when using differing number of threads. When a single thread is used, processor execution time is not effectively utilized, resulting in longer execution times compared to when multiple threads are used. With two or more threads, while one thread is waiting for the result of the accelerator, the other threads are used, hiding some of the overhead added by the communication between the threads. THe Figure 6.3 shows that the time performance is not significantly affected by the number of threads, when using more then one thread.

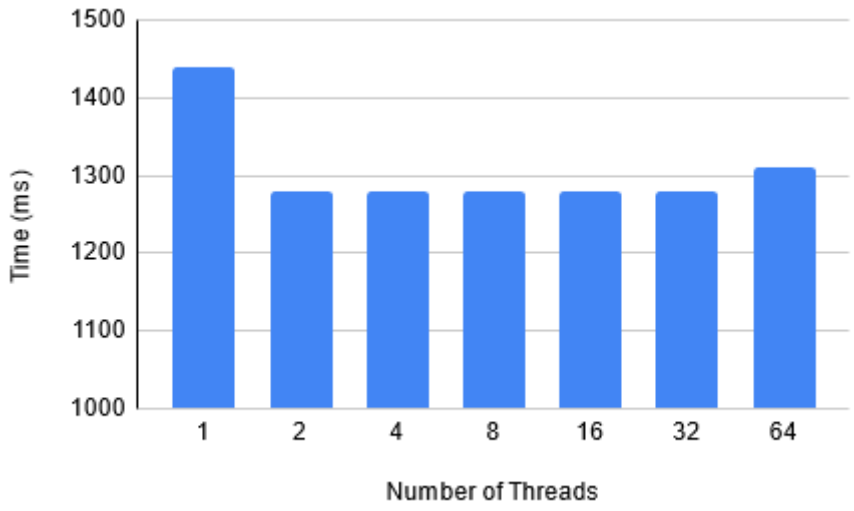


Figure 6.3: Time measurements of the multiplication of 512 matrices using the third scenario

Threads	1	2	4	8	16	32	64
Time (s)	53.16	52.92	53.39	53.52	53.18	53.30	53.12

Table 6.4: Time measurements of the prediction of 256 digits using the third scenario

Neural Network

In this scenario, the main goal is to have the hardware accelerator as independent from the other threads as possible, since it allows for a larger abstraction when developing the application. Because of this, the threads send the full matrices that need to be multiplied, and it's the manager threads job to make sure that the accelerators inputs fit the accelerator. One benefit of this approach is that it vastly reduces the number of requests done to the manager threads, reducing, in turn, the overhead added by the message queues.

Table 6.4 shows the execution times of the prediction of 256 using the neural network with the differing number of threads. In opposition to the previous use case, there is no significant difference between using a single thread and using multiple threads. At the same time, the weight matrices sent to the manager thread are the full matrices to reduce the number of messages between the threads. This adds more work to the manager thread, but reduces the number of context switches. Because of this, the time measured for this scenario using one thread is closer to the first scenario then the second scenario with one thread. Similarly to the multiplication of a set of matrices, Figure 6.4 shows that the execution time of the application does not increase with the number of threads.

This scenario uses a manager thread to control the use of the accelerator. Similarly to the previous scenario, this allows for multiple tasks to use the accelerator concurrently, and to use the processor while the accelerator is occupied. However, the use of a manager thread performed better then the second

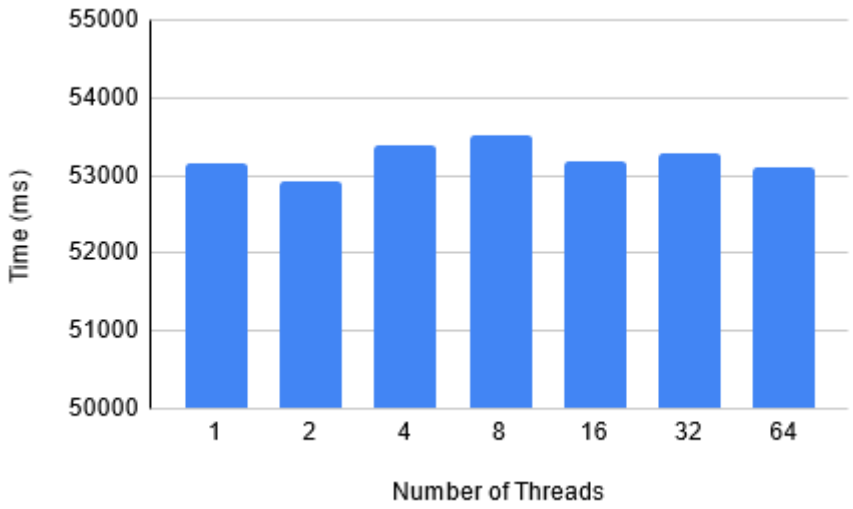


Figure 6.4: Time measurements of the prediction of 256 digits using the third scenario

scenario, even if it was a more complex implementation.

7

Conclusions and Future Work

The main objective of this thesis was to measure how an RTOS performs in a RISC-V based SoC, when managing an hardware accelerator. To do this, the SoC SweRVolf was modified to support a hardware accelerator for matrix multiplication, and three different methods were developed to manage the accelerator using the RTOSs mechanisms.

In systems where different tasks need to be managed by an RTOS, the managements of peripherals such as hardware accelerators can aided by the RTOS. In these systems it's necessary to comply to the time restrictions of the different tasks and the processors execution time can not be spent constantly checking for the results of the accelerator, or it may need to process the output of the accelerator as soon as it is available. Interrupts solve both of these obstacles, allowing the processor to switch between the tasks according to their assigned priority.

In Chapter 2 there is an analysis of the open-source SoCs with RISC-V based CPUs, and of the open-source RTOS. Chapter 3 describes the development process of an hardware accelerator for matrix multiplication with an AXI interface and an interrupt system. Chapter 4 provides an overview of the adaption of the SweRVolf SoC to support the hardware accelerator in the AXI Interconnect and to manage the interrupts generated in the accelerator using the PIC in the CPU. The implementation of Zephyr applications were done to manage the hardware accelerator and use the interrupts generated in the accelerator to allow the processor to perform other tasks while the accelerator is working. Finally, these applications where compared by measuring the time and power efficiency of the developed use cases in the FPGA board Nexys A7.

The three scenarios described in Chapter 5 attempt to use the interrupt and multitasking mechanisms of the RTOS to provide adaptability where techniques such as pooling can not. The scenarios provided range from situations where only one task can access the hardware accelerator to complex systems where multiple tasks can use the accelerator without the developer having to worry about the conditions of the accelerator. These scenarios enable the use of the processor at the same time as the accelerator with a small current consumption cost. In the end, using a manager thread enabled a more precise control over the use of the accelerator and performed better then the scenario where the multiple threads attempt to use the accelerator independently from each other.

While the main objective was achieved, there is room for further analysis on this topic. The analysis of the performance of individual operations and mechanisms would help better understand the execution time differences between the different scenarios, specially in the second scenario, allowing for further optimizations. Even though the Neural Network use case was developed with the intention to near the gap between this study and real world applications, those applications are not solely focused on a single operation. In the use cases studied all tasks perform the same operations, however there are scenarios where the different tasks operate independently and the RTOS still needs to manage the access to the hardware accelerator. There are other scenarios where multiple accelerators are connected to the SoC.

In these scenarios the RTOS must keep track of, not only the thread that is using an accelerator, but also what accelerator is being used. Finally, more complex operations can be accelerated using hardware accelerators, leading to larger time periods where the threads are suspended, allowing for other tasks to execute for longer while the accelerator is performing the assigned operation.

The environment used to design the accelerator and implement it into the SoC, as well as to develop the Zephyr application is detailed in Appendix A and the source code of all the hardware designs and software applications are available in the Github repository [36].

Bibliography

[1] S. L. Harris, D. Chaver, L. Piñuel, J. Gomez-Perez, M. H. Liaqat, Z. L. Kakakhel, O. Kindgren, and R. Owen, "RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 145–150.

[2] "Operating System Market Share Worldwide," https://gs.statcounter.com/os-market-share/, accessed: 2024-07-202.

[3] S. Greengard, "Will RISC-V Revolutionize Computing?" *Commun. ACM*, vol. 63, no. 5, p. 30–32, apr 2020. [Online]. Available: https://doi.org/10.1145/3386377

[4] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, 20191213th ed., RISC-V International, December 2019.

[5] K. A. Andrew Waterman and J. Hauser, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, 20211203rd ed., RISC-V International, December 2021.

[6] "RISC-V International," https://riscv.org/, accessed: 2023-04-15.

[7] "RISC-V Cores and SoC Overview," https://github.com/riscvarchive/riscv-cores-list, accessed: 2023-04-15.

[8] "LiteX Documentation," https://github.com/enjoy-digital/litex, accessed: 2023-05-06.

[9] "SweRVolf Documentation," https://github.com/chipsalliance/Cores-SweRVolf, accessed: 2023-05-06.

[10] "The NEORV32 RISC-V Processor," https://github.com/stnolting/neorv32, accessed: 2023-05-06.

[11] "List of Open-source Real-Time Operating Systems," https://www.osrtos.com/, accessed: 2023-05-28.

[12] "FreeRTOS: Real-Time Operating System for Microcontrollers," https://freertos.org/, accessed: 2023-05-28.

[13] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[14] "Zephyr Project, a Linux Foundation Project," https://www.zephyrproject.org/, accessed: 2023-05-28.

[15] "Google and Facebook Select Zephyr RTOS for Next Generation Products," https://zephyrproject.org/google-and-facebook-select-zephyr-rtos-for-next-generation-products/, accessed: 2024-05-19.

[16] "AMBA AXI4 Interface Protocol," https://www.xilinx.com/products/intellectual-property/axi.html, accessed: 2024-01-22.

[17] "Vitis HLS," https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html, accessed: 2024-01-30.

[18] "SweRV EH1 RISC-V Core," https://github.com/chipsalliance/Cores-VeeR-EH1, accessed: 2023-05-06.

[19] "Nexys A7 Digilent Reference," https://digilent.com/reference/programmable-logic/nexys-a7/start, accessed: 2023-05-06.

[20] "Simple SPI Core," https://opencores.org/projects/simple_spi, accessed: 2024-01-22.

[21] "PWM/Timer/Counter (PTC) Core," https://opencores.org/projects/ptc, accessed: 2024-01-22.

[22] "General-Purpose I/O (GPIO) Core," https://opencores.org/projects/gpio, accessed: 2024-01-22.

[23] "UART 16550 core," https://opencores.org/projects/uart16550, accessed: 2024-01-22.

[24] "Generic AXI Interconnect Fabric," https://opencores.org/projects/robust_axi_fabric, accessed: 2024-01-24.

[25] "SoC Interconnection: WISHBONE," https://opencores.org/howto/wishbone, accessed: 2024-01-22.

[26] "RVfpga: Understanding Computer Architecture & Introduction to SoC," https://university.imgtec.com/forums/rvfpga/, accessed: 2024-01-29.

[27] "IObundle," https://www.iobundle.com, accessed: 2024-01-24.

[28] "IOb-SoC," https://github.com/IObundle/iob-soc, accessed: 2024-01-24.

[29] "Xilinx's AXI Interconnect," https://www.xilinx.com/products/intellectual-property/axi_interconnect.html, accessed: 2024-01-24.

[30] “CoreMark,” https://www.eembc.org/coremark/, accessed: 2024-01-22.

[31] “Xilinx’s AXI Verification IP,” https://www.xilinx.com/products/intellectual-property/axi-vip.html, accessed: 2024-02-05.

[32] “Xilinx’s AXI BRAM Controller,” https://www.xilinx.com/products/intellectual-property/axi_bram_if_ctlr.html, accessed: 2024-02-05.

[33] “Xilinx’s Block Memory Generator,” https://www.xilinx.com/products/intellectual-property/block_memory_generator.html, accessed: 2024-02-05.

[34] “Neural Network in C++,” https://cognitivedemons.wordpress.com/2018/06/08/neural-network-in-c-part-2-mnist-handwritten-digits-dataset/, accessed: 2024-04-16.

[35] “RUI DENG’s USB Color Display Tester model UM24C,” https://www.mediafire.com/folder/0jt6xx2cyn7jt/UM24, accessed: 2024-05-27.

[36] “Github repository with source code of the hardware designs and software applications,” https://github.com/Rui-Ferreira3/MSc_Dissertation, accessed: 2023-06-10.

A

Environment Description

The instructions bellow are for the development of the hardware implementation, the development of the software application and the programming of the bitstream and the application binary files into the fpga board. While the full development environment can be used in most linux distributions, this guide describes how to use a combination of Windows for the Vivado Design Suite, Windows Subsystem for Linux (WSL) for the application development, and Raspberry Pi OS to program the files into the board. The Raspberry Pi is only used to facilitate remote work, thus the board is connected to it.

A.1 Installation of Software Tools

A.1.1 Vivado Design Suite

1. Navigate to https://reference.digilentinc.com/vivado/installing-vivado/start

2. You will be guided to Xilinx's download page to download Vivado Design Suit

3. It is recommended to download the ”Self Extracting Web Installer”

4. The vivado installer will guide through the installation process

5. After Vivado and Vitis HLS are installed, it’s necessary to install the cable drivers (follow the digilent guide above)

A.1.2 Zephyr Development Environment

1. Install VScode by following the link https://code.visualstudio.com/Download

2. Install pip: sudo apt install python3-pip

3. Install pyelftools: sudo apt-get install -y python3-pyelftools python-pyelftools

4. Install FuseSoC: sudo pip3 install –upgrade fusesoc

5. Install OpenOCD:
5.1 Install the required dependencies:

```
sudo apt-get install libusb-1.*
sudo apt-get install pkg-config
sudo apt-get install libtool
```

5.2 Clone the riscv-openocd github repository and install OpenOCD:

```
git clone https://github.com/riscv/riscv-openocd.git
cd riscv-openocd
./bootstrap
./configure --prefix=/opt/riscv --program-prefix=riscv- --enable-ftdi --enable-jtag_vpi
make
sudo make install
```

6 Install the required dependencies:

```
sudo apt install --no-install-recommends git cmake ninja-build gperf \
ccache dfu-util device-tree-compiler wget \
```

```
python3-dev python3-pip python3-setuptools python3-tk python3-wheel xz-utils file \
make gcc gcc-multilib g++-multilib libsdl2-dev libmagic1 \
```

Note: Make sure that the dependencies intalled are the correct versions according to this forum post.

7 Install cmake:

```
wget -O - https://apt.kitware.com/keys/kitware-archive-latest.a sc 2>/dev/null |
sudo apt-key add -
sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ bionic main'
sudo apt update
sudo apt install cmake
```

8 Install west:

```
pip3 install --user -U west
echo 'export PATH=~/.local/bin:"$PATH"' >> ~/.bashrc
source ~/.bashrc
```

9 Install Zephyr SDK:

9.1 Download the 0.12.4 SDK installer

9.2 Run the installer:

```
chmod +x zephyr-sdk-0.12.4-x86_64-linux-setup.run
./zephyr-sdk-0.12.4-x86_64-linux-setup.run -- -d ~/zephyr-sdk-0.12.4
```

A.1.3 FPGA Programming Environment

A Raspberry Pi board connected to the FPGA was used to facilitate remote work. This Raspberry Pi board was running the Raspberry Pi OS. On it, the RISC-V Toolchain and OpenOCD were installed so that the Raspberry Pi can program the SoC bitstream and the application's binary files. If the Raspberry Pi board is not bwing used some of the following steps may not be necessary.

1. Install the RISC-V Toolchain:

```
sudo apt-get install git autoconf automake autotools-dev curl \
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex \
texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
```

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain/
./configure --prefix=/opt/riscv --with-arch=rv32imc
sudo make
export PATH=$PATH:/opt/riscv/bin
```

2. Install OpenOCD:

```
sudo apt-get install libusb-1.*
sudo apt-get install pkg-config
git clone https://github.com/riscv/riscv-openocd.git
cd riscv-openocd
./bootstrap
./configure --prefix=/opt/riscv --program-prefix=riscv- --enable-ftdi --enable-jtag_vpi
make
sudo make install
```

A.2 Developing the Bitstream and Binary Files

A.2.1 Generation of the Bitstream

1. Open the rvfpga Vivado project in the Thesis github repository

2. Include two folder for the Pulp Platform. In the FLow Navigator pane click on *Settings*. In the opened window click on *General* and then next to the *Verilog Option* click on the three dots to select the directories. Navigate to the following directories and select the include foulder.

```
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/AxiInterconnect/
    pulp-platform.org_axi_0.25.0/include

[RVfpgaPath]/RVfpga/src/OtherSources/pulp-platform.org_common_cells_1.20.0/include
```

3. Generate the bitstream by clicking on *Generate Bitstream* under the *Flow* dropdown menu. The bitstream will be created in the following path:

```
[RVfpgaPath]/RVfpga/rvfpga.runs/impl_1
```

A.2.2 Generation of the Zephyr Application Binary

1. Create a new directory that will be the workspace to develop the Zephyr application. In this guide the workspace folder is called SweRVolf.

2. Navigate to the workspace directory and create an environment variable with the path to the workspace:

```
export WORKSPACE=\$(pwd)
```

3. Add the FuseSoC base library to the workspace:

```
fusesoc library add fusesoc-cores https://github.com/fusesoc/fusesoc-cores
```

4. Add the swrvolf library:

```
fusesoc library add swervolf https://github.com/chipsalliance/Cores-SweRVolf
```

Note: If fusesoc can't find the swervolf library swap the URL to https://github.com/chipsalliance/VeeRwolf.

5. Set the swervolf directory as a new environment variable:

```
export SWERVOLF_ROOT=$WORKSPACE/fusesoc_libraries/swervolf
```

6. Create a west workspace in the same directory as the fusesoc workspace:

```
west init
west config manifest.path fusesoc_libraries/swervolf
west update
```

7. Compile the Zephyr application: 7.1 Navigate to the the application samples directory:

```
$WORKSPACE/zephyr/samples
```

7.2 Copy the zephyr application folder from the Thesys repository to the samples folder.

7.3 Select what application to compile by uncommenting the target sources in the file *CMakeLists.txt*.

7.4 Compile the Zephyr application using west:

```
west build -b swervolf_nexys
```

Note: If you had to download the veervolf library above use the command:

```
west build -b veerwolf_nexys
```

A.2.3 Executing a program on RVfpgaNexys using the Nexys A7 board with OpenOCD

1. Copy the bitstream and the binary files into the Raspberry Pi board.

2. Copy the directory ConfigFiles located in the Thesis repository: rvfpga/OtherSources/ConfigFiles
3. Download the bitstream into the fpga using OpenOCD:

```
riscv-openocd -c "set BITFILE rvfpganexys.bit" -f \
ConfigFiles/swervolf_nexys_program.cfg
```

4. Connect OpenOCD to the SoC:

```
riscv-openocd -f /rvfpga/ConfigFiles/swervolf_nexys_debug.cfg
```

5. Connect gdb and execute the application:

5.1 Open a new terminal window.

5.2 Start GDB:

```
riscv32-unknown-elf-gdb zephyr.elf
```

5.3 Execute the application:

```
target remote localhost:3333
load
continue
```