

Sindre Jacobsen Faeroe

# **Fine-grained Multithreading for Deterministic Concurrency in Safety-Critical Systems**

Master thesis  
for the degree Master of Science in Embedded Systems

Trondheim, June 2022

Norwegian University of Science and Technology  
Faculty of Information Technology and Electrical Engineering  
Department of Electronic Systems

**NTNU**

Norwegian University of Science and Technology

Master thesis  
for the degree of Master of Science in Embedded Systems

Faculty of Information Technology and Electrical Engineering  
Department of Electronic Systems

© 2022 Sindre Jacobsen Faeroe

# Abstract

Integrating multiple tasks of differing criticality levels onto a single hardware platform, known as a mixed-criticality system, has become a growing research topic in real-time embedded systems. The correctness of these systems depends not only on the software's functionality but also on the timing behavior. Therefore, the underlying hardware platform must provide both spatial and temporal isolation guarantees and predictability to have high confidence in the timing behavior of the software. Unfortunately, even though existing hardware consisting of multiple single-core, multi-core, or multithreaded processors supports hardware-based isolation, they do not manage to exploit the hardware resources available on the platform entirely. Under-utilizing hardware resources can, as a result, cause a decrease in the overall throughput for a set of tasks.

This report presents a microarchitecture intended for mixed-criticality systems that allow exchanging hardware-based isolation between tasks to increase hardware resource utilization and vice versa. The microarchitecture has been designed and simulated in Simulink, a model-based design tool that enables fast feedback on design requirements and decisions. The design incorporates cycle-by-cycle hardware thread interleaving, also known as fine-grained multithreading, to increase the resource utilization of the platform. Additionally, a configurable hardware thread scheduler and timing instructions have been added to the microarchitecture to enable the programmer to make compromises between hardware-based isolation, predictable timing behavior, and overall instruction throughput.



# Preface

The work presented in this thesis was carried out in the Department of Electronic Systems, Faculty of Information Technology and Electrical Engineering at the Norwegian University of Science and Technology (NTNU). The Master Thesis was supervised by Professor Per Gunnar Kjeldsberg (NTNU) and former co-worker at UBIQ Aerospace, Shibarchi Majumder, Ph.D. (Nordic Semiconductor). I am thankful for their well-informed insight while working on the project. In addition, they provided me with exceptional guidance and motivation throughout this process.

I am grateful to my friends and co-workers for their vast support. They have given me inspiration and exceptional advice through their work and study experiences.

Finally, I would also like to thank my family, my father, mother, and brother, for encouraging me to pursue a Master's degree. I am also grateful for my lovely girlfriend's support during the course of the year. Her limitless patience through long evenings of work and the affection that she offers me have kept me motivated during my Master's Thesis at NTNU.



# Contents

<b>Abstract</b>	<b>v</b>
<b>Preface</b>	<b>vii</b>
<b>List of abbreviations</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Introduction and motivation . . . . .	1
1.2 Objectives . . . . .	2
1.3 Contributions . . . . .	3
1.4 Thesis Structure . . . . .	4
<b>2 Background and Related Work</b>	<b>5</b>
2.1 Mixed-Criticality Systems . . . . .	5
2.2 Timing Instructions . . . . .	7
2.3 Hardware Multithreading Techniques . . . . .	9
2.4 Model-Based Engineering . . . . .	11
2.5 RISC-V . . . . .	13
2.5.1 RV32I Base Integer Instruction Set . . . . .	13
2.5.2 Control and Status Registers . . . . .	14
2.5.3 Machine-level Timer Registers . . . . .	15
2.6 Previous Work . . . . .	16
2.6.1 Single-operation RISC-V Architecture . . . . .	16
2.6.2 FlexPRET . . . . .	16
<b>3 The Microarchitecture Design</b>	<b>19</b>
3.1 Architecture . . . . .	19
3.1.1 5-Stage RISC-V Pipeline . . . . .	19
3.1.2 Control Unit . . . . .	23
3.1.3 Memory Unit . . . . .	29
3.2 Hardware Thread Scheduler . . . . .	31
3.2.1 Hardware Thread Scheduling CSRs . . . . .	32
3.2.2 Main Scheduler . . . . .	32
3.2.3 SRTT Scheduler . . . . .	35
3.2.4 Thread Select Logic . . . . .	38
3.2.5 Stall Unit . . . . .	39
3.2.6 Hardware Thread Scheduler Example . . . . .	39
3.3 Timer Unit . . . . .	42
3.3.1 Timing Instruction Set . . . . .	42
3.3.2 Memory Mapped Timer Registers . . . . .	43
3.4 Assembler for the Microarchitecture . . . . .	46

<b>4</b>	<b>System Analysis</b>	<b>49</b>
4.1	System Configurations . . . . .	49
4.1.1	System Execution Time . . . . .	49
4.1.2	System Response . . . . .	56
4.2	Coarse-grained Multithreading . . . . .	62
<b>5</b>	<b>Conclusion and Future Work</b>	<b>65</b>
<b>Appendices</b>		
<b>A</b>	<b>Code</b>	<b>69</b>
A.1	Python code . . . . .	69
A.2	Assembly code . . . . .	77
	<b>Bibliography</b>	<b>81</b>

# List of Tables

4.1	The slots configuration for the single-threaded system. . . . .	50
4.2	The mode configuration for the single-threaded system. . . . .	50
4.3	Real-time constraints of the tasks. . . . .	51
4.4	The slots configuration for the multi-threaded system using only soft real-time threads (SRTT). . . . .	52
4.5	The mode configuration for the multi-threaded system using only soft real-time threads (SRTT). . . . .	53
4.6	The mode configuration for the multi-threaded system using only hard real-time threads (HRTT). . . . .	54



# List of Figures

2.1	Mixed Criticality Systems Application Integration . . . . .	6
2.2	DO-178C Design Assurance Levels . . . . .	6
2.3	Coarse-grained Multithreading . . . . .	10
2.4	Fine-grained Multithreading . . . . .	11
2.5	RV32I instruction type formats . . . . .	13
2.6	I-type operation instruction formats . . . . .	14
2.7	RV32I 5-stage pipeline . . . . .	14
2.8	Slot CSR . . . . .	17
2.9	Mode CSR . . . . .	18
3.1	High-Level Model of Microarchitecture . . . . .	20
3.2	Control Unit Simulink Design . . . . .	23
3.3	CSR Models . . . . .	25
3.4	Branch Data Hazards in Schedule . . . . .	27
3.5	Timer Compare First Solution . . . . .	28
3.6	Timer Compare Second Solution . . . . .	29
3.7	Memory Unit Simulink Design . . . . .	30
3.8	Memory Address Check . . . . .	31
3.9	Hardware Thread Scheduler Simulink Design . . . . .	32
3.10	Thread Scheduler Configuration Example . . . . .	33
3.11	Main Scheduler Logic Example . . . . .	33
3.12	Simple Ring Buffer Model . . . . .	35
3.13	Main Ring Buffer Example . . . . .	36
3.14	SRTT Scheduler Logic Example . . . . .	37
3.15	SRTT Ring Buffer Example . . . . .	38
3.16	Thread Select Logic . . . . .	39
3.17	Stall Unit . . . . .	40
3.18	Hardware Thread Scheduler Example . . . . .	40
3.19	Timing Diagram of Hardware Thread Scheduler . . . . .	41
3.20	Hardware Thread Scheduler Example with Stall . . . . .	41
3.21	Timing Diagram of Hardware Thread Scheduler with Stall . . . . .	41
3.22	Timing Instruction Set . . . . .	43
3.23	Memory Mapped Timer Registers . . . . .	44
3.24	Compare Block . . . . .	45
3.25	Delay Until Unit . . . . .	46
3.26	Timer Interrupt Unit . . . . .	46
4.1	Execution times for single threaded configuration . . . . .	52
4.2	Execution times for multi-threaded SRTT configuration . . . . .	54

4.3	Execution times for multi-threaded HRTT configuration . . . . .	55
4.4	Execution times for multiple tasks . . . . .	56
4.5	Total execution times . . . . .	57
4.6	Model of Response and Reaction Times . . . . .	58
4.7	Response Time Analysis SingleThreaded . . . . .	59
4.8	Response Time Assembly Instructions . . . . .	59
4.9	Response Time Analysis Multithreaded 8 threads . . . . .	60
4.10	Response time as a function of thread scheduling frequency . . . . .	61
4.11	Overall Latency . . . . .	64

# Source code

2.1	timingcontrol.a . . . . .	8
2.2	timingcontrolcase2.a . . . . .	8
3.1	branchthread.a . . . . .	22
3.2	timinginstructions.a . . . . .	43
3.3	assemblerbefore.a . . . . .	47
3.4	assemblerafter.a . . . . .	47
4.1	singlethreadedconfig.a . . . . .	50
4.2	fibonacci.a . . . . .	51
4.3	srttconfig.a . . . . .	52
4.4	srttmain.a . . . . .	53
4.5	hrttconfig.a . . . . .	55
4.6	responsetimetask.a . . . . .	58
A.1	RiscvAssembler.py . . . . .	69
A.2	DataHazardsSingleThreaded.a . . . . .	77
A.3	DataHazardsMultiThreaded.a . . . . .	78



# List of abbreviations

<b>CSR</b>	Control and Status Register
<b>DAL</b>	Design Assurance Level
<b>FPGA</b>	Field-Programmable Gate Array
<b>GPS</b>	Global Positioning System
<b>HDL</b>	Hardware Description Language
<b>HRTT</b>	Hard Real-Time Thread
<b>I/O</b>	Input/Output
<b>IPC</b>	Inter-Process Communication
<b>ISA</b>	Instruction Set Architecture
<b>LSB</b>	Least Significant Bit
<b>MBD</b>	Model-Based Design
<b>MBE</b>	Model-Based Engineering
<b>MCS</b>	Mixed-Criticality System
<b>MMIO</b>	Memory Mapped Input/Output
<b>MSB</b>	Most Significant Bit
<b>PC</b>	Program Counter
<b>PLL</b>	Phase-Locked Loop
<b>PMP</b>	Physical Memory Protection
<b>RF</b>	Register File
<b>RTC</b>	Real-Time Clock
<b>RTL</b>	Register-Transfer Level
<b>RTOS</b>	Real Time Operating System
<b>SRTT</b>	Soft Real-Time Thread
<b>VGA</b>	Video Graphics Array

**VHDL** Very high-speed integrated circuit Hardware Description Language

**WCET** Worst-Case Execution Time

# Chapter 1

## Introduction

### 1.1 Introduction and motivation

A real-time system is a computer system that must react to events in the given environment within a predetermined period, called a deadline [1, 2]. Consequently, the system's behavioral correctness depends not only on a calculation's logical correctness but also on the moment in which the system produced these results [3, 4]. In contrast to a general-purpose system, where the objective is to have fast computation to minimize the average response time of a given set of tasks, the goal of a real-time system is to meet the timing requirement of each task. The significance of an average-case execution time is minor for the real-time system behavior if several tasks are executed with different timing constraints. For example, consider the following quote made by Howard Marks:

Never forget the 6-foot-tall man who drowned crossing the stream that was 5 feet deep on average.

As the quote implies, even though the average depth was less than the height of the man, it could still be deeper in various places. Similarly, although the average-case performance of the system is less than the timing constraints placed on the system, a specific task execution may still be too slow to meet its timing constraint. Hence, a real-time system should have a predictable timing behavior rather than having a fast average-case performance.

In a real-time embedded system, a software task is assigned a criticality level based on how important it is to finish the task within a specified deadline. The tasks required to meet their respective deadlines to avoid catastrophic consequences are hard real-time tasks. On the other hand, a task is referred to as a soft real-time task if the system will still function correctly even though the deadlines are occasionally missed [5]. For example, an autonomous drone contains several real-time tasks that must keep up with external changes affecting the drone [6]. The obstacle avoidance and the global positioning system (GPS) tasks are among these tasks. If the obstacle avoidance task cannot react quickly enough to external changes, it can result in the drone crashing into an object, potentially destroying the drone and objects, and can lead to injury. Meanwhile, a late transmission from the GPS task reduces the reliability of the GPS location of the drone, meaning that the received location lags behind the actual location of the drone. Thus, the obstacle avoidance task must have a high criticality level as the task must finish within a given deadline. On the other hand, the GPS task can have a lower criticality level as missing a deadline is tolerable.

Real-time systems traditionally located multiple tasks with different criticality levels on separate hardware platforms to prevent unwelcome interference between tasks [7]. However,

having respective hardware platforms results in excessive resource consumption. For example, a drone with multiple tasks of different criticality levels would require numerous hardware platforms. As a result, the drone would be costly and energy inefficient due to the extra components needed to isolate the tasks. The drone would additionally be large and heavy due to the number of components required and the increased battery size due to the energy usage. Therefore, a solution integrates multiple components of more than one distinct criticality level onto a shared hardware platform. This integration method is an increasingly popular approach to designing real-time embedded systems, known as a mixed-criticality system [8].

With the integration of multiple criticality levels onto the same hardware platform, two interference issues arise; spatial and temporal interference. Temporal interference is the ability of a group of tasks on the same hardware platform to interfere with each other's time constraints [9]. This type of interference can cause a highly critical task to miss its deadline due to a lower criticality task obstructing the predicted timing schedule of the highly critical task. Spatial interference is the ability of a group of tasks on the same hardware platform to alter each other's code or private data [10]. Spatial interference can cause a highly critical task to have corrupted data due to the lower criticality task modifying the private data of the highly critical task, resulting in either incorrect or delayed task execution.

For a mixed-criticality system to be certifiable, there is a need for each task of varying criticality levels to meet their timing constraints to different levels of assurance. For example, the RTCA DO-178C avionics standard categorizes the criticality levels into five levels of assurance, ranging from A to E [11]. The worst-case execution time (WCET) used for analysis and certification is usually a conservative upper bound that exceeds the actual WCET of the system [12, 13]. Having tight bounds on the WCETs of the tasks is thus desirable, where it is preferable to have the WCET values as close as possible to the actual WCET. Furthermore, by maintaining sound spatial and temporal isolation, each independent task can avoid being negatively influenced by other tasks, resulting in predictable task execution. This predictable execution due to sound task isolation further leads to tight bounds on the WCET [14].

Because of the inefficient hardware resource utilization of hardware-based isolation using one processor per task, substantial research on software scheduling of mixed-criticality systems has been carried out over the past two decades [15, 16, 17]. In particular, research into real-time operating systems (RTOS) provided a method to enable software-based isolation while drastically reducing hardware costs. However, the RTOS must be certified, where overheads such as task switch latency, preemption time, and inter-process communication (IPC) must be considered [18, 19]. This overhead results in timing requirements in milliseconds, which is not precise enough for some applications. For example, applications that use software to replace functionality implemented initially in hardware or that interact with various hardware devices demand precision-timed input/output (I/O) with timing accuracy and precision on the order of nanoseconds which is at clock cycle granularity for processors [20].

## 1.2 Objectives

Instead of having a hardware platform for each application, it is possible to implement a processor where each task is deployed on separate computational components, such as cores in a multicore processor or hardware threads in a multithreaded processor. However, allocating a core to each task can result in the hardware resources being vastly underutilized because a less computationally heavy task can occupy an entire core. Meanwhile, a multithreaded processor allows multiple hardware threads to share a pipeline, resulting in fewer hardware resources for each computational component. Thus, having each task allocated a hardware thread enables better hardware resource utilization when multiple tasks are executing on the same

processor. However, many multithreaded processors do not have hardware thread scheduling mechanisms that manage to preserve hardware-based temporal isolation between tasks. Thus, a thread scheduling mechanism that manages to keep the hardware-based temporal isolation while utilizing the hardware resources better must be explored.

A typical processor can contain several components leading to a context-dependent execution time, such as memory, caches, pipelines, and branch prediction. This context-dependency leads to the execution time of individual instructions and memory accesses depending on the execution history [21]. For example, dynamic branch prediction allows the program to have insight into the execution history to avoid flushing pipeline stages when a branch is taken. However, there are instances where the information received from the execution history is not enough to make an assumption based on branch outcomes. Because the branch prediction is dynamic, it is challenging to foresee until the program run-time how the execution of instructions will behave exactly. The outcomes of the dynamic branch predictions may mainly vary when external influences coming from sensors are presented to the system, resulting in less predictable instruction latencies. Because the instruction's execution time depends mainly on the execution history, the processor lacks fine-grained predictability. As a result, the architectural features such as complex branch prediction and multilevel cache used to optimize the average-case performance reduce the coarse-grained predictability of the processor and make the WCET analysis cumbersome [14].

Fine-grained multithreading is a technique used in recent processor designs where instructions from different hardware threads are interleaved every clock cycle in the pipeline. This technique can enable hardware-based temporal and spatial isolation between tasks by allowing each task running on a hardware thread to use the pipeline for one clock cycle periodically. As a result, several tasks can run on the same mixed-criticality system without interfering with each other's timing constraints. Furthermore, the performance penalty caused by removing dynamic branch prediction and caches can be significantly reduced because of the concurrent execution of tasks provided by fine-grained multithreading. By removing the dependency on execution history, the fine-grained predictability of the processor can be maintained, and the WCET analysis becomes less demanding. However, even though fine-grained multithreading enables hardware-based isolation and predictable timing behavior, it cannot fully utilize the processor unless all threads are constantly running.

### 1.3 Contributions

This report presents a fine-grained multithreaded microarchitecture designed in Simulink that provides a means for each hardware thread to select between predictability and hardware-based isolation or increased instruction throughput. A hardware thread focusing on predictability will simplify the verification and certification of the safety-critical tasks. Additionally, the hardware-based isolation of the hardware thread allows the task to meet its timing constraints accurately. In comparison, having the hardware thread focusing on increased instruction throughput allows the less critical tasks to maintain good hardware resource utilization.

The designed microarchitecture contains a hardware thread scheduler and timing instructions inspired by the theory provided by the FlexPRET research paper [14]. Like the research paper, the microarchitecture contains a hardware-thread scheduler that uses two active round-robin schedulers to decide the hardware thread schedule. However, the microarchitecture utilizes a unique design to implement configurable round-robin schedulers of variable sizes. These configurable schedulers use CSRs to configure the hardware schedule, allowing the software to adjust the content and length of the thread schedule. The designed

microarchitecture also introduces a unique implementation of the four timing instructions presented by FlexPRET. The timer unit presented in this microarchitecture incorporates memory-mapped timer registers that expand on the `mtime` and `mtimecmp` registers described in the RISC-V documentation. Here, in addition to providing the original real-time timer interrupt functionality, the timer instructions give direct control over the thread schedule with fine-granularity resolution. Thus, the timer instructions allow the timing behavior of a task with hard real-time execution requirements to be specified in software while reallocating spare hardware resources to tasks with soft real-time execution requirements.

By designing the microarchitecture in Simulink, it was possible to continuously test and verify the design decisions made from start to finish. This design process made it possible to spot bugs early in the design process, thus drastically reducing the number of mistakes made. Additionally, the understanding received from previous research theory was used to inspire some of the design requirements during the design process. By using Simulink to test and verify the design, these design requirements could be adjusted or discarded based on the understanding and results provided by the simulations. As a result, a custom microarchitecture molded by inspiration from previous theory was achieved.

## 1.4 Thesis Structure

This thesis presents a fine-grained multithreaded microarchitecture incorporating a configurable hardware thread scheduler aimed at mixed-criticality and safety-critical systems. Any essential background information and prior work in the field that is necessary to explain the design decisions and reasoning is described in Chapter 2. Further, some of the design choices made throughout the microarchitecture design are discussed in Chapter 3. Additionally, some design issues and other enhancements or modifications are explained here. Chapter 4 will configure the hardware thread schedule of the microarchitecture in several ways to perform analyses on the execution times, response times, and its ability to hide instruction latencies. These results are then used to reason why specific threads provide high predictability, making them better for higher criticality tasks, while other threads are better for low criticality tasks prioritizing high instruction throughput. Finally, Chapter 5 will conclude the work done during this master thesis and discuss any problems that would be interesting in future works.

## Chapter 2

# Background and Related Work

### 2.1 Mixed-Criticality Systems

In Real-time systems, temporal correctness is just as important as logical correctness [3, 4]. A real-time system is required to respond to stimuli from its given environment within time intervals decided by that particular environment. The moment in time that a result must be available is called the deadline [1, 2]. The timing constraints of a real-time system are usually represented by how it enforces the deadlines placed on executing tasks. A deadline is classified as soft if the result of a task still has some value after the deadline has elapsed; otherwise, the deadline is firm.

In some cases, however, the result of a missed firm deadline can have severe consequences such as destruction of property or loss of life [22]. These deadlines are hard. Any real-time system classifies a system with at least one hard deadline as a hard or safety-critical real-time system. The system is a soft real-time system when there are no hard deadlines. For a hard real-time system to achieve functional correctness, all tasks with hard deadlines must meet their assigned deadlines. If such a task fails to meet its deadline, the executed task is considered a failure, and the system did not function as intended. Sometimes, failure in a hard real-time system could lead to catastrophic ramifications. In contrast, a soft real-time system has more relaxed constraints on its tasks. In these types of systems, it may be acceptable for a task to finish its execution after its deadline. However, the usefulness of the results in a soft real-time system decreases gradually with an increase in delay.

A safety-critical system can consist of functionalities with different levels of criticality. Criticality is used as a label to specify the level of assurance against failure needed for a system [11]. For example, a weather drone can contain flight control and weather sensing applications. If a failure within the functionality of the flight control application occurs, it could lead to the drone's destruction and possibly cause injury. As a result, the flight control application must be considered highly critical to avoid such circumstances. In comparison, the weather sensing application is of a lower criticality, where the drone could experience a failure by being unable to gather data for the weather station. This failure will only be perceived as an inconvenience for the weather station as it will at most require the drone to return to the weather station to fix the issue. Suppose both of these applications have separate physical hardware units on the drone. In that case, there is isolation between the functionality of both applications, meaning that the low-criticality task will not affect the high-criticality one. This isolation provides high safety guarantees and simplifies the certification process by limiting it to only the critical functions [22]. However, having a physical hardware unit for each application on the drone will result in inefficient resource utilization, an increase in costs and size, and a heavy drone with a significant amount of

energy consumption. Alternatively, as illustrated in Figure 2.1, it is possible to integrate the applications on the same hardware unit to reduce the drone’s cost, weight, size, and energy consumption. This integration will require the applications on the hardware unit to be assigned individual criticality levels to avoid having the drone prioritize gathering data the same way as the flight controls. Because of the mentioned advantages, integrating multiple applications with different levels of criticality onto a common hardware platform, known as a Mixed-Criticality System (MCS), is a growing trend in the design of real-time and embedded systems [23, 8].

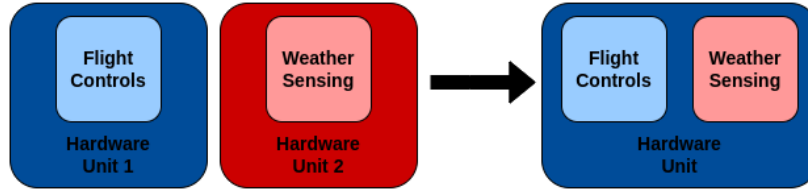


Figure 2.1: Applications of different criticality levels can be located on separate hardware units to achieve isolation between applications. Integrating these applications onto a shared hardware unit reduces the hardware’s cost, weight, size, and energy consumption.

To allow a safety-critical system to be implemented and deployed, it must be certified due to the risks that could occur during system failures. To become certified, a third party known as the Certification Authority must perform a certification process to verify that the safety-critical system is safe. The certification standard usually used in the avionic systems’ certification process is the RTCA DO-178C standard [11]. The RTCA DO-178C standard defines five Design Assurance Levels (DAL), categorized by their criticality from the highest criticality level (DAL-A) to the lowest (DAL-E) [17]. Because of the consequences of a failure or malfunction in an application of higher criticality levels, it is clear that the higher the DAL, the more activities, and objectives must be performed and met. In total, DO-178C includes 71 objectives, where 43 of these are related to verification. Therefore, for an application to pass with a DAL-A assurance, all 71 objectives must be met, as seen in Figure 2.2. In comparison, a DAL-E assurance does not require any objectives to be met because there are no consequences to the safety of the aircraft if a failure should occur. Thus, the flight control application would be regarded with a DAL-A assurance for the drone example above. In contrast, the weather sensing application could be as low as DAL-E.

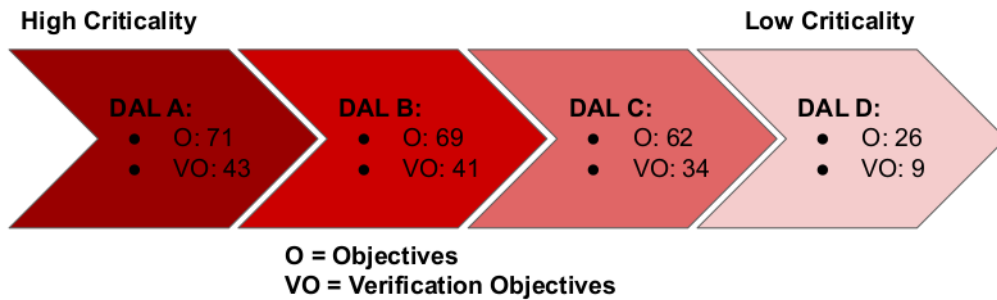


Figure 2.2: The number of objectives and verification objects required in each DAL decreases from left to right [11]. Notice how DAL-E is not on the scale due to not requiring any objectives to be met.

A vital characteristic of any real-time system is that it behaves predictably. This char-

acteristic means the system can perform all applications correctly (functional predictability) and within a given timing bound (timing predictability). A significant concern when integrating multiple applications on the same hardware is the partitioning of the system [8]. Unfortunately, by combining various applications onto the same hardware, the tasks from one application will share compute resources with tasks from other applications. This sharing of resources increases the interference between the tasks regardless of the criticality level [24]. This interference means testing and verification must also account for the interference's timing behavior when determining the system's functional correctness. Introducing spatial and temporal isolation in the system can prevent independent tasks from affecting each other's behavior [20]. Spatial isolation is responsible for protecting a task's state that is stored in various forms of memory, while temporal isolation protects the timing behavior of a task [7]. When complete isolation is introduced, each task can be tested and verified for correctness by itself and still have identical behavior when integrated into the system. As a result, an isolated task will have better timing predictability, which will provide tighter bounds on the worst-case execution time (WCET) analysis [21]. These WCET bounds resulting from the WCET analysis determine the timing partitions that a scheduling tool or system designer must reserve for executing each task [12, 13]. Thus, as a consequence of having tighter bounds on the WCET of a system, fewer system resources will be required to maintain guarantees in the system behavior [16].

## 2.2 Timing Instructions

Real-time systems assign deadlines and periods to tasks. To meet every deadline and know when the real-time system should perform the task, the system requires a sense of time. An instruction that specifies the minimum amount of execution time for a section of code was introduced by Ip and Edwards [25]. A deadline instruction was implemented that forces the program to pause its execution while a timer register decrements. The pause lasts until the specified timer register reaches zero. Once the timer is zero, it is reloaded with the source value of the deadline instruction, and the program continues to the following instruction. This deadline instruction provided a software solution for specifying a lower execution time-bound on specific program code segments. Finally, they demonstrated the deadline instruction by showing how a video controller could be implemented in software and hardware. They concluded that an application that would usually only be possible in hardware is now much easier to write and debug in software, where they wrote roughly four times as many lines of code in VHDL compared to assembly.

Lickly et al. added this deadline instruction to their fine-grained multithreaded processor [26]. In addition, they included a replaying mechanism to the deadline instruction that allowed a particular thread to be stalled without stalling the entire pipeline. Each thread has twelve deadline registers, where eight of these count instruction cycles while the other four count at a different frequency specified by a phase-locked loop (PLL). Because the architecture uses a 6-stage pipeline, the instruction cycle is six clock cycles, as this is the time it takes an instruction to pass through the pipeline. Finally, they illustrated that the architecture could use the deadline instruction to meet VGA real-time constraints using a video game example with three main tasks running on separate threads.

To provide temporal isolation in a multiprocessing architecture, Bui et al. identified four timing control cases that are desirable to have at an ISA level [27]:

- 1 - Executing a code segment takes a specified *minimum amount of time* [25].

- 2 - Execute a code segment and *branch afterward* if the code segment exceeds the specified execution time limit.
- 3 - Execute a code segment and *branch immediately* if the code segment exceeds the specified execution time limit.
- 4 - Finish a code segment within a *maximum execution time*.

The first three capabilities can be added to a given ISA relatively simply, where the following four pseudo-instructions, proposed in previous work, can express this [20]:

- **get\_time:** Loads the current time of a timer register into a destination register.
- **delay\_until:** Stall the program execution until the value in the timer register exceeds the value in the provided register.
- **interrupt\_on\_expire:** Interrupt program execution when the value in the timer register exceeds the value in the provided register.
- **deactivate\_interrupt:** Disables the interrupt\_on\_expire operation that is already in progress.

Consider a weather sensing drone that reads a sensor and adds up to 1000 samples to an average before transmitting the value to the weather station, shown in Listing 2.1. The sensor task uses the interrupt\_on\_expire and deactivate\_interrupt to achieve case 3 mentioned above. Here, the sensor task is allocated 1 ms of time, interrupting the task’s execution if it exceeds the time limit. To ensure that the task uses a specific amount of time, the delay\_until task is added to achieve case 1, where the task will stall until 1 ms has elapsed since the retrieval of the current time.

Listing 2.1: timingcontrol.a

---

```

1  addi x6, x0, 1000           //Set loop limit
2  read_sensor:
3      get_time x1             //Get current time
4      addi x1, x1 1000000     //Calculate time 1 ms in future
5      addi x2, x0 0           //Set iterator to 0
6      addi x5, x0 0           //Set sensor sum register to 0
7      interrupt_on_expire x1   //Set a timer interrupt to trigger in 1 ms
8  poll_input:
9      lw x4, 0(x3)            //Read sensor
10     addi x2, x2, 1           //Increment iterator
11     ... Add to Average Calculation Code ...
12     blt x2, x6, poll_input   //Branch if less than loop limit
13  output_average:
14     deactivate_interrupt     //Deactivate interrupt
15     sw x5, 4(x3)             //Write sensor average
16     delay_until x1           //Stall until 1 ms from current time

```

---

To achieve case 2, a comparison between two get\_time instructions can be performed. By reading the timer before and after some code segment, the program can see whether the code segment’s duration lasted too long, where the program will branch to deadline\_miss if more than 1 ms elapsed.

Listing 2.2: timingcontrolcase2.a

---

```

1  get_time x1
2  addi x1, x1, 1000000
3  ... Code ...
4  get_time x2
5  bgt x2, x1, deadline_miss

```

---

Implementing the mentioned pseudo-instructions in hardware is relatively simple, where many possible alternative implementations exist [20]. The timing control cases' general idea is to stall, branch, or interrupt program execution due to a timer comparison. As a result, increased predictability in the timing behavior of a task can be achieved, where the timing instructions can allocate a specific amount of time to a task.

## 2.3 Hardware Multithreading Techniques

Hardware multithreading is a popular method used to improve the utilization of processor resources by integrating multiple hardware threads into the same processor core [28]. A hardware thread is logically equivalent to a processor, where each hardware thread has individual state registers, such as a register file (RF), a program counter (PC), and control and status registers (CSR) [29]. However, a hardware thread shares a pipeline and frequently the same memory space as other hardware threads within the same processor. By sharing the same or parts of the same memory space, the hardware threads can effortlessly access the data of other hardware threads, thus enabling data sharing [30].

In an operating system with multiple software threads, a software thread switch would spend many clock cycles storing the state of the software thread before fetching the state of a different software thread. As a result, having multiple software tasks allocated a certain amount of computing time will add significant latency to the overall execution time due to the context switching performed in software. In comparison, by having a software task allocated to a hardware thread in a multithreaded processor, the processor can perform hardware thread switching at clock cycle granularity, thus removing the latency due to the storing and retrieving of the task state.

In addition to removing the latency caused by context switching, hardware multithreading has an additional latency-related benefit. In a single-threaded processor, branching and cache miss results in short and long stalls in the task execution, respectively. Short stalls refer to a delay in executing a task of a few clock cycles, while a long stall can be many hundreds of clock cycles. Because of the poor hardware resource utilization caused by these stalls, much research is being performed on branch and cache prediction mechanisms to hide the added delay to the program execution [31, 32, 33]. In addition to these prediction mechanisms, it was discovered that an inherent benefit of hardware-based multithreading is that it can be used to hide these delays in program execution by switching to a different thread instead of stalling.

Coarse-grained multithreading, or block interleaving, is a hardware multithreading technique introduced to hide long stalls [30]. For example, the thread schedule of a coarse-grained multithreaded processor with five pipeline stages can be seen in Figure 2.3. When the coarse-grained technique notices a more prolonged stall such as a cache miss, it performs a thread switch [29]. This thread switch allows a different thread to execute while the thread that was stalled fetches the required data from a higher level cache, such as the L3 cache. However, when the thread has managed to retrieve the necessary data from the L3 cache, it must either wait until the other thread gives away the compute resources, or the thread has to preempt the thread currently occupying the pipeline. As a result, the thread that was stalled may have significant latency added to its task execution.

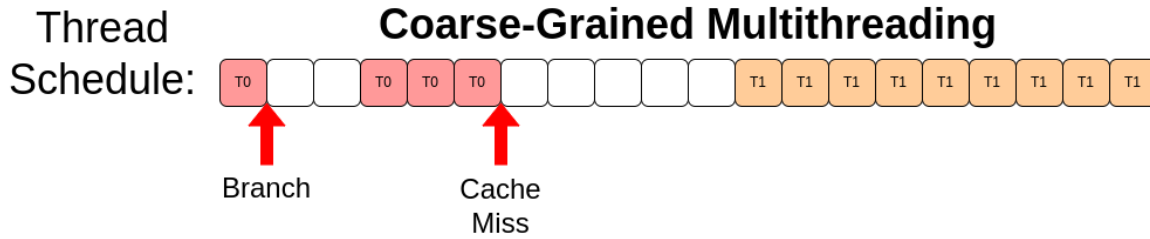


Figure 2.3: An example thread schedule using the coarse-grained multithreaded technique. The technique manages to hide most of the latency caused by cache miss, but is incapable of hiding the branch latency.

As can be seen in Figure 2.3, the program execution latency in a coarse-grained multithreaded processor caused by the cache miss is five clock cycles compared to the hundreds of clock cycles latency present in a single-threaded processor without cache prediction. The five-clock cycle delay is because the pipeline must be emptied of instructions from the previous thread and filled with instructions from the new thread. Thus, the number of clock cycles delayed depends on the number of pipeline stages in the processor architecture. Because of this pipeline delay, the coarse-grained multithreaded technique does not benefit from switching on shorter stalls. If the coarse-grained multithreaded processor example performs thread switching on branch stalls, the latency introduced to the pipeline will be three clock cycles longer than if it did not perform any thread switching. As a result, coarse-grained multithreading is used to hide a large portion of the latency caused by longer stalls such as cache misses. In contrast, it cannot conceal branch latencies without a branch prediction mechanism.

Fine-grained multithreading, or cycle-by-cycle interleaving, is another hardware multithreading technique introduced to hide long and short stalls [30]. For example, the thread schedule of a fine-grained multithreaded processor with five pipeline stages and five hardware threads can be seen in Figure 2.4. The fine-grained multithreaded technique performs thread switching at the clock cycle granularity. The thread scheduler usually switches between available hardware threads each clock cycle in a round-robin fashion [29, 34]. This scheduling method means that if there are as many threads as pipeline stages, the next instruction from any particular thread is, in principle, fed to the pipeline once the previous instruction is completed. Thus, the need for data forwarding is eliminated, and any data hazards that would otherwise cause a flush of the pipeline stages are avoided. As a result, the fine-grained multithreaded technique can avoid short stalls by having other threads use the pipeline stages that would otherwise be flushed. This avoidance can be seen in Figure 2.4, where the branch operation performed by thread0 does not lead to a stall because thread1 and thread2 are using the thread cycles where a pipeline flush would otherwise have occurred.

An example thread schedule using the coarse-grained multithreaded technique. The technique manages to hide most of the latency caused by cache miss, but is incapable of hiding the branch latency.

In addition to hiding shorter stalls by performing hardware thread interleaving every clock cycle, the fine-grained multithreaded technique can hide longer stalls, as seen in Figure 2.4. Here, thread0 notices a cache miss where the thread scheduler is notified of this occurrence. While thread0 fetches the required data from the L3 cache, the hardware thread scheduling technique will decide how the unused cycles are allocated. For example, PTARM is a fine-grained multithreaded processor that uses fixed round-robin scheduling, alternating between

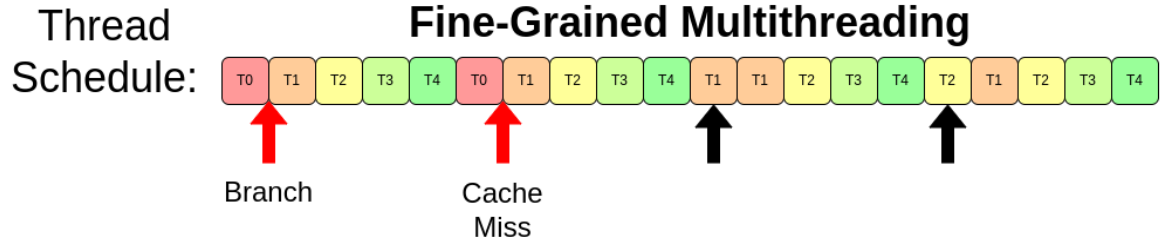


Figure 2.4: An example thread schedule using the fine-grained multithreaded technique. The technique manages to hide the cache latency, except the cycle causing the cache miss. Here, other threads are using the spare cycles while Thread0 is stalling. Additionally, the technique manages to hide branch latency by scheduling other threads during the cycles that can be flushed due to a branch.

all hardware threads regardless of the thread status [35]. Because the scheduler does not care about the thread status, the thread0 cycles will be left unused until the thread stops stalling. Furthermore, if other threads result in cache misses, those cycles will also be left unused.

On the other hand, XMOS is a fine-grained multithreaded processor that uses active round-robin scheduling, alternating between all hardware threads ready to execute [36]. This scheduling method can be seen in Figure 2.4, where thread1 and thread2 use the cycles initially allocated to thread0. The disadvantage of this technique is that it can introduce some latency due to branch stalls, where two adjacent thread cycles can potentially be assigned to the same thread. For example, the figure shows thread1 being adjacent to another thread1 cycle and thread2 having a single thread cycle between another thread2 cycle. If the first thread cycle, in either case, performs a branch, the other cycle will result in a stall. Thus, there is a potential for some branch latency in the active round-robin scheduling method. However, active round-robin scheduling can allocate most unused cycles to other threads, resulting in better hardware resource utilization than fixed round-robin scheduling.

Although the fine-grained multithreaded technique is capable of hiding both short and long stalls, there is a trade-off that must be made. Because the method is performing cycle-by-cycle interleaving, where the scheduler will interleave several threads, the frequency of any specific thread will be reduced. For example, using the fine-grained processor above running at 100MHz clock frequency, each thread will execute at 20MHz. As a result, the frequency of each thread is inverse proportional to the number of actively scheduled threads. However, even though each thread is executing slower, concurrent thread execution results in a total throughput equivalent to that of a single-threaded processor when disregarding stall latency. In comparison, the coarse-grained technique has a similar thread-specific throughput as the single-threaded processor. For example, with a 100MHz frequency coarse-grained multithreaded processor, a single thread will execute at 100MHz due to the seemingly sequential behavior of the thread scheduling method. Finally, the overall throughput of the coarse-grained technique is equivalent to the fine-grained approach when disregarding stall latency.

## 2.4 Model-Based Engineering

Model-based design (MBD) is an effective and efficient method for understanding various systems, such as those found in microcontrollers and processors. MBD helps reduce the complexities that arise during the development of a system through visual prototyping and

continuous testing and validation of system characteristics [37, 38]. By continuously testing and validating the system throughout the design process, spotting bugs and areas in the design or requirements that need modification is made easy. This continuous checking enables rapid prototyping by fast feedback on requirements and design decisions that result in an overall reduction of development risks [39, 40]. Furthermore, traditional design processes usually handle the design information in a text-based approach that can lead to misinterpretations and be challenging to understand. Standard document-based procedures can also be more time-consuming and prone to errors due to the code and data being created manually in a text-based form. These reasons are essential when designing large and complex systems, such as satellites, aircraft, and automobile systems, where mistakes in the system can lead to catastrophic failure. With a model-based approach, it is possible to break a complex system into smaller digital models. These models are then verified to reduce the number of defects injected due to a lack of overview and validation of design decisions in a traditional document-based approach. These designed models do not necessarily need to represent the system perfectly. However, they serve as a method to supply the designer with valuable knowledge and feedback on the devised system sooner and more cost-effectively than system implementation alone.

Because of its ability to reduce development time, resolve design problems early, and provide less ambiguous system documentation, Model-Based Engineering (MBE) is an expanding field in multiple industries, including the automotive and aerospace industries [41, 42]. Amongst the many areas within these industries, MBE can also be used to accelerate FPGA development [43]. In a traditional FPGA design process, a systems engineer produces a high-level computer simulation of the system to be designed [44]. Further, the designer communicates with a hardware developer that will write the code for the model to be implemented on an FPGA. The hardware developer must understand the simulated system's logic before engaging in the HDL design. MBE significantly speeds up the design and verification process by allowing the hardware developer and systems engineer to cooperate using functional models of the design specifications. These are executable models and are easier for designers coming from different fields of expertise to interpret similarly. Consequently, the hardware developer does not need to spend the same amount of time trying to decipher the simulated design and can directly start the process of implementation and verification on an FPGA.

Simulink is an effective MBE tool that can further accelerate FPGA development using the available HDL Coder add-on. The tool can generate HDL code from the model by having the designer create an HDL-compatible Simulink Model [45, 46]. Thus, it is possible to design functional models that can be continuously tested and verified throughout the entire design process and later used to generate HDL code. Furthermore, Simulink provides methods for reusing reference models in design verification and performing FPGA-based debug and verification [47, 48]. This debug and verification method allows the RTL code to be verified against the reference model through simulation and on an FPGA. As a result, less time is spent correcting design mistakes, reducing the total development time considerably.

Moreover, as the generated HDL code has been designed and verified on an FPGA, Simulink also provides the capability to integrate the model into a more extensive system and perform Hardware-in-the-loop (HIL) simulation [49]. This simulation will allow the design to be further tested by providing the design implemented on an FPGA with various stimuli. These stimuli are then handled by the FPGA, which further outputs signals relative to the stimuli. The simulation then provides new stimuli dependent on the output signals retrieved from the FPGA. As a result, the algorithms implemented on the FPGA can be verified in an environment that simulates the real world, allowing critical systems to be tested in detail to ensure that no errors are possible.

## 2.5 RISC-V

RISC-V is a non-profit organization with billions of chips on the market and a large and growing community [50]. This organization provides the community with an open-source Instruction Set Architecture (ISA). An ISA is a computer model that defines how the processor is controlled by software, allowing developers to understand better what the instruction set can do, allowing for more efficient code to be written [51, 30]. Because RISC-V's ISA is open-source and the growing community, there is a myriad of resources available related to custom microarchitecture design based on this ISA [52, 53, 54]. RISC-V has a collection of available extensions that can be added to the baseline ISA, making it possible for designers to pick between various functionality options to add to their design, reducing the amount of custom functionality required.

### 2.5.1 RV32I Base Integer Instruction Set

The 32-bit base integer instruction set, also known as RV32I, is one of the most straightforward architectures available in the RISC-V extension libraries. The RV32I is an ISA designed to reduce the required hardware with minimal implementation while forming a compiler target able to support modern operating system environments [55]. RV32I contains 40 unique instructions that can be used to emulate all of the ISA extensions available except for the atomic-instruction extension, which requires additional hardware [56]. Creating a hardware implementation using this ISA, including the machine-mode privileged architecture, would require the addition of the 6 CSR instructions mentioned in Section 2.5.2.

RV32I also has several types of instructions. Namely, R-type, I-type, S-type, B-type, U-type, and J-type, shown in Figure 2.5. These instructions have their unique purpose, where, e.g., B-type instructions are branch instructions and I-type instructions are register-immediate operations [55]. First, the operation code stored in the seven least-significant bits (LSB), called opcode, is checked to determine which type of instruction is being executed. Then, based on the instruction type that has been established, it is possible to extract the instruction fields. As can be seen in the figure, most fields come from the same set of bits regardless of the instruction type. The immediate field, however, is constructed of different bits depending on the kind of instruction.

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7				rs2			rs1	funct3		rd			opcode		R-type
imm[11:0]						rs1	funct3		rd			opcode		I-type	
imm[11:5]				rs2		rs1	funct3		imm[4:0]			opcode		S-type	
imm[12]	imm[10:5]			rs2		rs1	funct3		imm[4:1]	imm[11]	opcode			B-type	
imm[31:12]									rd			opcode		U-type	
imm[20]	imm[10:1]				imm[11]	imm[19:12]			rd			opcode		J-type	

Figure 2.5: The types of instructions available in the RV32I ISA.

Figure 2.6 shows the instruction encoding formats for several of the math operations that the RV32I architecture is capable of performing. The term I-type means that the instruction is of Immediate-type. These instructions only use one source register, rs1, whereas the other

value, `imm[11:0]`, is derived from the instruction. The 3 bits in the middle of the 32-bit instruction format are the functional bits, which specify which operation is to be performed. These bits are used along with the opcode in the 7 LSBs to select the instruction operation. Once the operation has been performed on these two values, the result is placed in the destination register, `rd`.

<code>imm[11:0]</code>	<code>rs1</code>	0 0 0	<code>rd</code>	0 0 1 0 0 1 1	I-type	<code>addi</code>	<code>rd,rs1,imm</code>
<code>imm[11:0]</code>	<code>rs1</code>	0 1 0	<code>rd</code>	0 0 1 0 0 1 1	I-type	<code>slti</code>	<code>rd,rs1,imm</code>
<code>imm[11:0]</code>	<code>rs1</code>	0 1 1	<code>rd</code>	0 0 1 0 0 1 1	I-type	<code>sltiu</code>	<code>rd,rs1,imm</code>
<code>imm[11:0]</code>	<code>rs1</code>	1 0 0	<code>rd</code>	0 0 1 0 0 1 1	I-type	<code>xori</code>	<code>rd,rs1,imm</code>
<code>imm[11:0]</code>	<code>rs1</code>	1 1 0	<code>rd</code>	0 0 1 0 0 1 1	I-type	<code>ori</code>	<code>rd,rs1,imm</code>
<code>imm[11:0]</code>	<code>rs1</code>	1 1 1	<code>rd</code>	0 0 1 0 0 1 1	I-type	<code>andi</code>	<code>rd,rs1,imm</code>

Figure 2.6: Example instruction formats for various I-type operations.

The instructions of RV32I are of the same lengths and can be fetched in a single cycle. Because of this, it is beneficial to divide the microarchitecture into pipeline stages [57]. Thus, a reduced clock time and throughput improvement can be achieved, where multiple instructions are overlapped in execution [58]. An example of a pipelined RV32I is the 5-stage pipeline, shown in Figure 2.7. This pipelined microarchitecture contains the fetch, decode, execute, memory access, and write-back stages, performed in the respective order [59] [60].

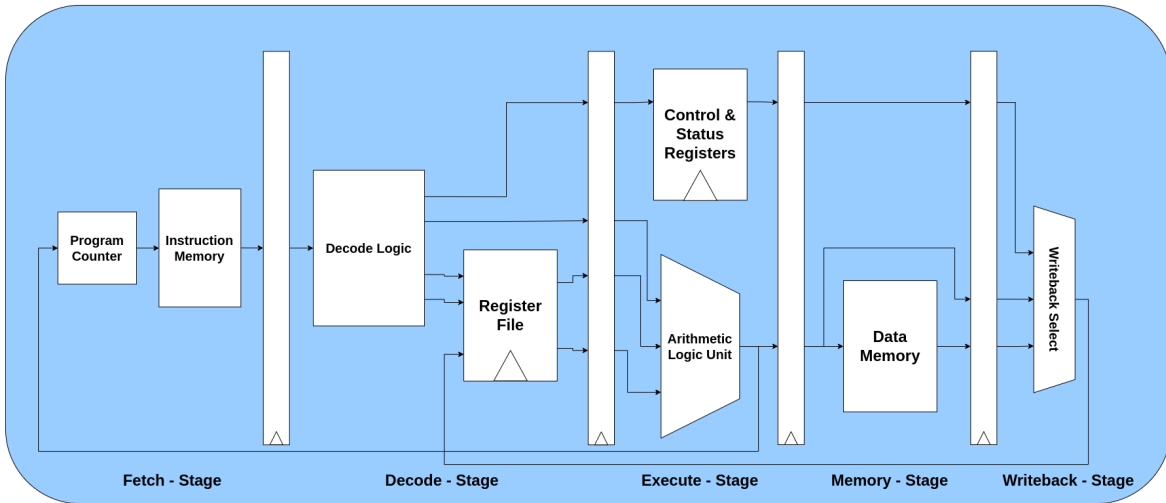


Figure 2.7: An example of a 5-stage pipeline containing the stages fetch, decode, execute, memory access, and write back, separated by a register.

### 2.5.2 Control and Status Registers

To perform any control and status register (CSR) operations, RISC-V defines specific CSR instructions to read or write a CSR, known as `Zicsr` [61]. There are six available CSR instructions within `Zicsr`, where three (`CSRRW`, `CSRRS`, and `CSRRC`) use a source register. The other three instructions (`CSRRWI`, `CSRRSI`, and `CSRRCI`) use a 5-bit zero-extended immediate value to read/write a CSR. These instructions within the standard RISC-V ISA have a 12-bit encoding space set aside for up to 4096 CSR. The 2 MSB of this encoding space indicates whether the register is read/write or read-only. The following two bits encode the lowest privilege level that can access the CSR. This CSR address convention constrains the mapping of CSRs into the address space but makes the hardware easier to check for errors

and provides a larger CSR space [61]. The highest privilege mode available in a RISC-V system is the machine mode. This mode is used for low-level access to the hardware platform and is the first mode entered at a hardware reset. The machine mode additionally allows the implementation of features that would otherwise be implemented in hardware directly but is either too difficult or expensive to realize [62].

The machine-level ISA contains a list of several important CSRs that were added to the designed microarchitecture, described in Chapter 3, to allow certain features to be added [63]. For example, embedded systems rely heavily on handling asynchronous events, known as interrupts, which are designed to be managed by the processor [64]. Therefore, within the machine-level ISA, there are several CSRs described that are used to configure interrupts. These CSRs are part of the CSRs referred to as trap setup and trap handling CSRs, which are used to manage interrupts and exceptions. The term trap refers to the synchronous transfer of control to a trap handler caused by either an interrupt or an exception. The term exception refers to unusual conditions associated with the instruction in the currently running thread that can occur during run-time, such as illegal instructions and instruction address misalignment.

The CSRs required as a bare minimum to handle interrupts are the `mstatus`, `mie`, and `mtvec` trap setup CSRs and the `mepc`, `mcause`, and `mip` trap handling CSRs [64]. By having these CSRs, a RISC-V architecture can enter and exit the interrupt handler, although it will lack certain features that are not within the scope of this report. In addition to the CSRs necessary to handle traps, the `mhartid` CSR is essential in a multithreaded system. The `mhartid` is a read-only register that holds an integer value corresponding to the ID of the hardware thread that is running some code. Due to the occasional call for startup configuration, e.g., a system reset, one of the threads must have a thread ID of 0. Additionally, the magnitude of the largest thread ID used in the system should be as low as possible for higher efficiency.

### 2.5.3 Machine-level Timer Registers

The machine-level timer register, `mtime`, is a memory-mapped register that provides the RISC-V architecture with a real-time counter [65]. This timer register is required to increment at a constant frequency, and the architecture must have a mechanism that determines the time base of the counter. Once `mtime` reaches the maximum value, the register will overflow, and the register will wrap around. In both 32-bit (RV32) and 64-bit (RV64) RISC-V systems, the `mtime` register is 64 bits.

In addition to the `mtime` register, another register known as `mtimecmp` is defined [65]. This register is a 64-bit memory-mapped timer compare register, which in combination with `mtime`, is used for timer interrupts. Whenever `mtime` contains a value greater than or equal to the value in `mtimecmp`, a timer interrupt becomes pending [64]. This interrupt will remain pending until the value of `mtimecmp` becomes greater than `mtime`, which typically results from writing `mtimecmp`. To allow a timer interrupt to occur, the timer interrupt bit, `MTIE`, must be set in the `mie` register. Otherwise, the architecture will not be able to treat the pending interrupt.

An accurate real-time clock (RTC) is relatively expensive and must be able to run even when the rest of the system is powered down [65]. As a result, the RTC is usually shared between threads in the same system, and accessing the RTC may result in the penalty of a voltage-level-shifter and clock-domain crossing. Thus, it is more natural to have the `mtime` register exposed as a memory-mapped register rather than a CSR. Finally, although the `mtime` register is shared between threads, having at least one `mtimecmp` register for each thread is typical, allowing the thread to perform timer interrupts.

## 2.6 Previous Work

### 2.6.1 Single-operation RISC-V Architecture

The single-operation RISC-V architecture was implemented in Simulink during the specialization project, TFE4590, last semester at the Norwegian University of Science and Technology (NTNU) [66]. This project aimed to implement a microarchitecture with a 5-stage pipeline capable of performing the addi operation from the RISC-V's base integer instruction set (RV32I) using a custom-made design methodology. The first step of this methodology was to design the functionality of each stage using Matlab scripts, followed by the design of the microarchitecture in Simulink. The third step of the methodology was to implement the microarchitecture in Verilog, where the knowledge from the Simulink design reduced errors during code writing. Finally, the last step was to use the Simulink model (step 2) as a golden reference model to test the Verilog code implemented on an FPGA. The idea of the specialization project was to develop a proper design methodology that could be used to design a fine-grained multithreaded microarchitecture. However, the time required to implement the fine-grained multithreaded microarchitecture, including the testing, resulted in only step 2 being considered. It should be noted, however, that the single instruction microarchitecture used in the previous semester was further used in this project as the baseline. However, the entire 5-stage pipeline was redesigned due to a better understanding of Simulink, resulting in better model-based testing and more readable models.

### 2.6.2 FlexPRET

FlexPRET is a fine-grained multithreaded RISC-V processor that allows thread-level hardware-based isolation and predictability to be traded for instruction throughput [14]. The processor enables an arbitrary interleaving of threads controlled by a unique hardware thread scheduler. This scheduler allows for a flexible thread schedule where threads can either be characterized by isolation and predictability or efficient processor utilization. FlexPRET also utilizes timing instructions that enable direct control over timing in nanoseconds resolution, as mentioned in Section 2.2.

As described in Section 2.3, a fine-grained multithreaded system performs clock-by-clock thread interleaving. To achieve such fine-granularity hardware thread switching, some mechanism is required to decide which thread should be selected for each clock cycle. In the FlexPRET processor, the unit responsible for this is the hardware thread scheduler, a configurable scheduler capable of scheduling anywhere between 0 and 8 threads in a round-robin fashion.

Unlike most multithreaded systems, the hardware thread scheduler presented in FlexPRET classifies the threads as either hard real-time threads (HRTTs) or soft real-time threads (SRTTs). An HRTT is a thread that can only use the thread cycles allocated to that specific thread, meaning that the thread is locked to a slot in the repeating sequence. If the thread is not assigned any slots in the sequence, then the thread will stay inactive. In contrast, if a thread is configured as an SRTT, then the thread can have specific thread cycles allocated like an HRTT in addition to using any spare thread cycles available. This means that if unused slots are in the repeating sequence in the hardware thread scheduler, an SRTT can use these cycles. The hardware thread scheduler utilizes two active round-robin schedulers for the threads; one that schedules both HRTT and SRTT for fixed slots in the sequence and one that schedules SRTTs for spare cycles.

To configure the hardware thread scheduler, FlexPRET implemented two configuration registers: `mthreadmode` and `mthreadslot`. The `mthreadslot` register, shown in Figure 2.8,

is responsible for selecting the repeating thread sequence that is delivered from the thread scheduler to the fetch stage of the pipeline. The register is divided into eight 4-bit registers, called slots. As the name implies, a slot is a placeholder for a thread ID in the repeating sequence. Each of these slots can have a value from 0 to 15, specifying which thread should have priority for that slot in the sequence. For example, the values 0 to 7 indicate specific thread IDs, meaning that if slot0 contains the value 4, then thread4 is prioritized for the zeroth slot in the repeating sequence. The meaning of prioritized in this case is that if the thread is awake, it has priority for that slot. However, if the thread is asleep, other threads, namely SRTTs, can use this slot until the prioritized thread wakes up again. Thus, each slot value specifies the thread ID with priority and not whether the slot is private to a specific thread ID. When the value eight is allocated to a slot, it means that the slot is soft. Any SRTTs can use this particular slot in the repeating sequence. A soft slot functions the same way as a slot containing the value of a thread ID that is sleeping. Finally, it should be possible to disable a slot in the sequence when a repeating sequence of length 8 is undesirable. To disable a slot in the sequence, any other values, 9 to 15, can be stored in the slot the programmer wants to disable.

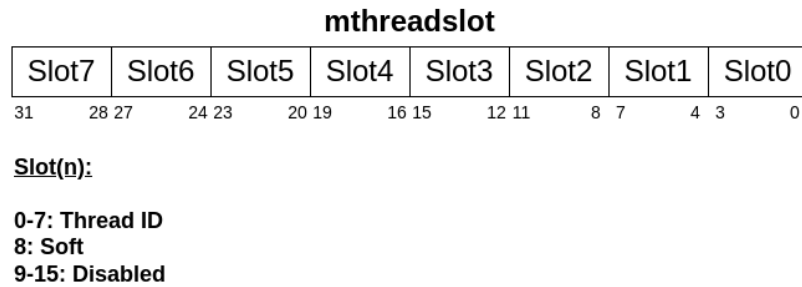


Figure 2.8: The slot CSR of the hardware thread scheduler. Each slot can either contain a thread ID (0-7), a soft thread value (8), or be inactive (9-15).

The mthreadmode register, shown in Figure 2.9, is responsible for selecting whether a hardware thread should be an HRTT or SRTT and whether the thread is awake or asleep. The register only uses the lower 16 bits of the 32-bit register, where the used bits are divided into eight 2-bit registers, called mode registers. Each mode register has a thread ID, where mode0 relates to thread0. Compared to the slots CSR, the mode registers allocates 1 bit to select the mode of the thread and the other to choose the state. The mode register's least significant bit (LSB) is used to determine the thread's state; when the bit is 0, it means that the thread is awake, while a bit value of 1 means sleeping. The most significant bit (MSB) of the mode register selects the thread's mode; when the bit is 0, the thread is configured as an HRTT, while a bit value of 1 means that the thread is configured as an SRTT.

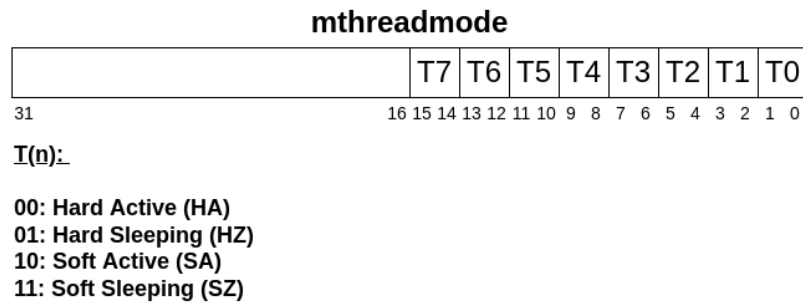


Figure 2.9: The mode CSR of the hardware thread scheduler. This CSR configures each thread as either hard or soft real-time threads and whether the thread is awake or asleep.

## Chapter 3

# The Microarchitecture Design

The architecture presented in Section 3.1 consists of a 32-bit RISC-V microarchitecture with five pipeline stages and uses vectored interrupts. Because the design is made in Simulink and has not been converted to HDL code, the critical paths of the system are unknown, meaning that the microarchitecture cannot provide any specific frequency at the time of this writing. Next, Section 3.2 presents the microarchitecture’s reconfigurable software-controlled hardware thread scheduler. A dynamic scheduler allows the software to freely adjust the schedule and mode of the hardware threads whenever required. These hardware threads can be configured as hard real-time threads (HRTT) or soft real-time threads (SRTT). Classifying these hardware threads as either HRTTs or SRTTs makes it possible to tailor each thread’s predictability, isolation, and throughput. However, changing one of these parameters may cause the other parameters to deteriorate. It is then necessary to select these levels with care. The microarchitecture contains a timing extension that improves the overall throughput of the program, presented in Section 3.3, and makes it possible to add real-time constraints. This extension contains a few timing instructions that allow the program to implement real-time functionality. Finally, Section 3.4 presents a custom assembler designed for this microarchitecture that understands instructions from the base integer RISC-V ISA, the Zicsr extension instructions, and the custom timing instructions.

### 3.1 Architecture

The microarchitecture consists of three primary modules: The 5-stage RISC-V pipeline, the memory unit, and the control unit, as shown in Figure 3.1.

#### 3.1.1 5-Stage RISC-V Pipeline

The pipeline is based on a typical 5-stage RISC-V pipeline containing the stages: fetch, decode, execute, memory, and write-back, as seen in Figure 3.1. To enable the microarchitecture to use fine-grained multithreading, additional state components, such as register files and program counters, must be integrated into the pipeline to allow multiple threads to run on the same microarchitecture. The following paragraphs detail the modifications done to each pipeline.

The pipeline must be expanded with additional program counters in the fetch stage to allow multithreading in the microarchitecture. These program counters enable each thread to point to different locations in the instruction memory. Hence, pointing to separate memory regions allows each thread to perform separate software tasks. Thus, eight program counters were added to the fetch stage, making it possible to schedule eight hardware threads. As a

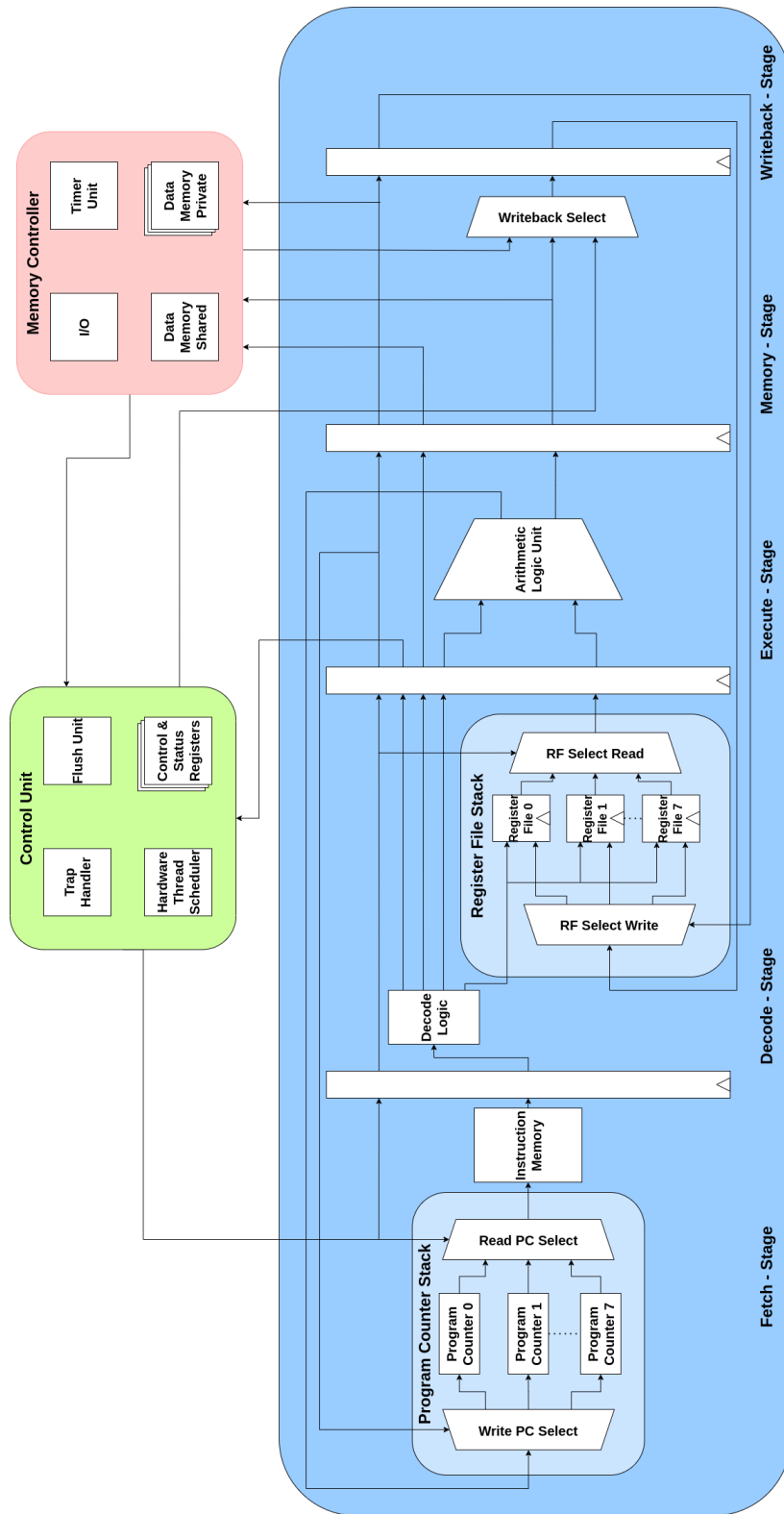


Figure 3.1: A high-level model of the microarchitecture. The design consists of three main parts: a 5-stage RISC-V pipeline, Control Unit, and a Memory Unit, colored in blue, green, and red, respectively.

result, eight distinct software tasks can execute concurrently in the microarchitecture while maintaining hardware-based isolation between the tasks. However, a mechanism must be available to select which program counter to forward and increment. Selection between hardware threads is achievable using a multiplexer to choose which program counter should access the instruction memory. Additionally, the signal used to select the program counter also increments the program counter. Furthermore, the pipeline must know which program counter to update when performing operations such as branching. This update is achievable by adding a demultiplexer to the input to write the branch value to the correct program counter.

The multiplexer and demultiplexer used respectively for reading and writing utilize a thread ID that identifies the program counter to access. As can be seen in Figure 3.1, the multiplexer receives the thread ID from the hardware thread scheduler. In comparison, the demultiplexer receives the thread ID from the execute stage since this is the stage responsible for deciding any jumps in the instruction memory. Thus, the thread ID from the hardware thread scheduler must be forwarded to the other stages to make the pipeline aware of the thread ID residing in each stage. This awareness allows the stages to identify which thread is occupying the stage, thus limiting access to the state components with identical thread ID.

As mentioned previously, each thread must be supplied with a register file. Thus, to equip the microarchitecture with eight threads, the decode stage must contain eight register files, where each register file is private to a thread. By having a private register file, a hardware thread can maintain its state when the pipeline is shared with other hardware threads. However, although the decode stage contains eight register files, a mechanism similar to the program counters must be added to select the correct register file for a thread. As a result, a multiplexer and demultiplexer select port is added to the output and input, respectively. Compared to the program counter, the multiplexer uses the thread ID from the pipeline register between the fetch and decode stages. Since the write-back stage writes the register file, the demultiplexer uses the thread ID from this stage to select the corresponding register file.

The execute stage communicates with the control unit, where the control and status registers (CSRs) reside. To allow the threads to be individually configured, each thread must have separate CSRs. Thus, similar to the fetch and decode stage, the CSRs must be duplicated where a multiplexer and demultiplexer are used to read and write the correct CSRs, respectively. However, compared to the program counter and register file, the CSR stack requires an additional CSR unit where shared CSRs are located. In the shared CSR unit, the configuration registers for the hardware thread scheduler, and the CSR that limits the shared data memory region reside. A different solution is to have only thread 0 to configure these CSRs, as this is the default thread used when starting any software program on the microarchitecture. However, this microarchitecture does not limit thread access to shared CSRs, meaning any thread can modify these registers. This flexibility can accidentally modify the thread schedule or change the shared memory region if the software program is not carefully considered. Additionally, once thread 0 configures the hardware thread scheduler, the other threads will begin fetching instructions from address 0 in the instruction memory. This initialization means that these threads will also run the same configuration procedure if there are no restrictions.

Two potential methods are explored to avoid all threads running the same configuration procedure; one purely software solution and one combined hardware and software. Each thread can read its thread ID from its private CSR register, called *mhartid*, to avoid running the same configuration on all threads in software. This thread ID can decide where the program should branch in memory and begin its thread-specific execution. Listing 3.1 shows

the implementation of this method, where all threads with a thread ID unequal to 0 branches to main. This will, however, require all threads to read the mhartid register from its private CSRs, and branch to a different location depending on the thread id. As a result, each thread's program execution will spend several instruction cycles deciding where the thread should jump.

In contrast, the combined hardware and software solution utilizes an enable signal that is active only when the thread ID is 0. The enable signal allows access to the shared CSRs, meaning that only thread 0 can read and write these CSRs. However, this does not remove the issue of the starting location in memory. Thus, even though only thread 0 can modify and read the shared CSRs, the other threads will still run the configuration instructions. Although, the hardware restricts these registers to thread 0, resulting in the assembly instructions that are trying to modify the shared CSRs to act as NOP instructions. To further improve the startup procedure, additional CSRs that specify which PC value each thread should be initialized to could be added. As a result, instructions must be executed during the startup procedure to configure the PC CSRs. Thus, the latter solution would require modifications to both hardware and software.

Because the additional cycles required by the software solution are negligible, adding the additional hardware logic required to achieve the combined hardware and software solution is unnecessary. Additionally, by carefully designing the software, it should not be necessary to have any limitations on access to the shared CSRs.

Listing 3.1: branchthread.a

---

```

1 startup:
2     csrrsi x1, 3860, 0           //set x1 = mhartid
3     bne x1, x0, main            //if x1 != 0, Jump to main

```

---

Similar to how the execute stage communicates with the control unit, the memory stage communicates with the memory controller. The memory controller must ensure that each thread has a private memory region to ensure that multiple threads can read and write the same memory without data races occurring. The upper and lower boundaries of these private memory regions are configured in the CSRs. Once a thread wants to read or write a memory location, the upper and lower boundaries for that particular thread are fetched from the CSR and matched against the address that the thread tries to access. The thread cannot access the memory location if the address is outside the memory bound. This makes it possible for a single thread to access a region that no other thread can access. A shared memory region can also be configured to share data between threads. For example, if thread 1 depends on some calculations done by thread 2, thread 2 can store the values in the shared memory region so that thread 1 can retrieve the values. Thus, the memory controller provides the memory stage with private and shared memory regions to enable spatial isolation and data sharing between software tasks.

In addition, the memory stage communicates with a timer unit using custom timing instructions. These timing instructions can either read a timer or set a timer compare unit to perform a thread-specific timer interrupt or put a thread to sleep. Similar to the register file, the timer compare unit checks the thread ID to select the unit to access. This check allows the threads to have individual timer compare units, resulting in each thread performing timing control operations without affecting other threads individually.

Due to the addition of these modifications to the 5-stage pipeline, the microarchitecture can have several threads running concurrently. The features added to the microarchitecture enable both spatial and temporal isolation. Spatial isolation is achieved by having private

state registers and memory regions for each hardware thread. A shared memory region is also available to make it possible for threads to share data. Additionally, the private and shared memory regions are configurable through software, thus allowing the programmer to decide how much memory each thread should be assigned. For temporal isolation, the hardware thread scheduler decides which thread should access the pipeline for each clock cycle. In addition, these threads have timing capabilities, enabling the assignment of real-time constraints to the software tasks. First, Section 3.1.2 looks further into the control unit, where all the logic that can change the program's behavior is present. Next, Section 3.1.3 looks further into the memory unit containing the data memory and any memory-mapped registers.

### 3.1.2 Control Unit

As mentioned above, the control unit is responsible for the logic that changes the program's behavior. A top-level Simulink model of the control unit can be seen in Figure 3.2. Each of the components of the control unit will be discussed in this section except the hardware thread scheduler. Instead, the hardware thread scheduler will be discussed in Section 3.2, as this is one of the most significant contributions to the microarchitecture.

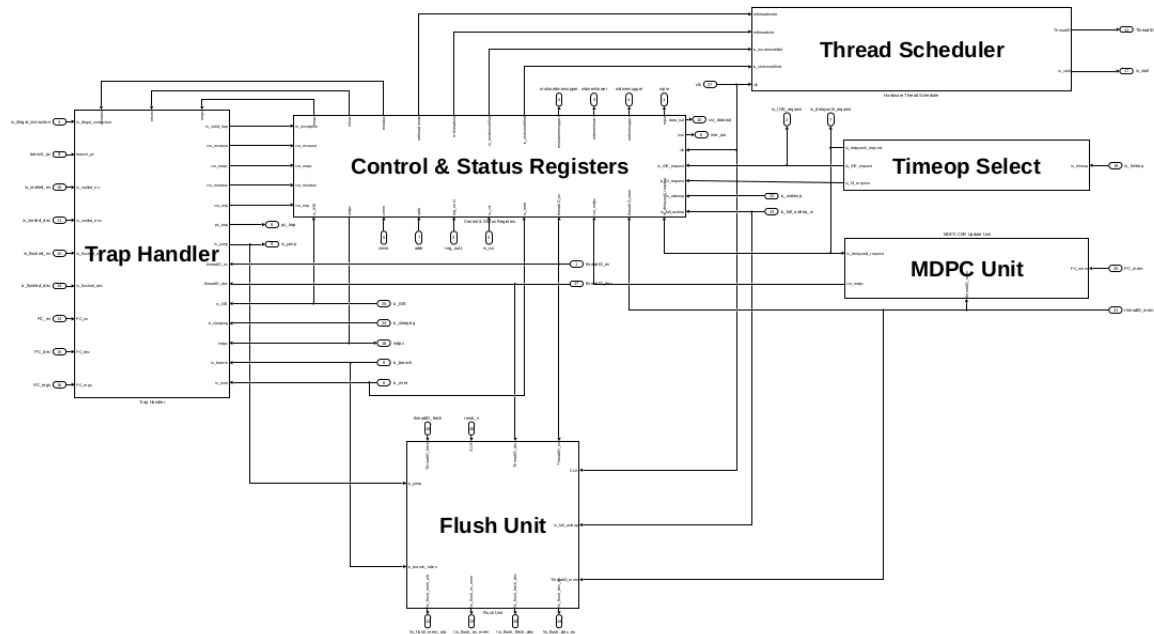


Figure 3.2: A top-level model of the control unit designed in Simulink. The components are responsible for controlling the behavior of the pipeline.

#### Control and Status Registers

The control and status registers (CSRs) are essential in the microarchitecture. This component is responsible for the configurations required in the microarchitecture to enable features such as interrupt handling and hardware thread scheduling. As mentioned in Section 3.1.1, a thread has private and shared CSRs. The models shown in Figure 3.3 are simplified models to illustrate how the CSR unit is structured. The signal bus in the figure is an arbitrary bus containing all signals connected to the CSR unit. Therefore, it does not consider that the bus differs for the private and shared thread CSRs.

As can be seen in Figure 3.3a, there are private CSRs for each thread. The private thread CSRs use the `mhartid` CSR, shown in Figure 3.3b, to select which of the private thread CSRs the execute stage should access; if the thread ID of the execute stage finds a matching `mhartid`, the execute stage gains access to that private thread CSR. However, if the thread ID in the execute stage does not find any matches, it is unable to read or modify any private thread CSRs. Because the `mhartids` available in the private thread CSRs and the thread IDs that are schedulable in the hardware thread scheduler are fixed, the program will never meet this issue. However, the `mhartid` only verifies access to the private thread CSRs. Thus, if the `mhartid` value is modified in the Simulink model while the thread ID check is not modified, e.g., in the program counter demultiplexer, this could cause an issue. A solution would be to retrieve the `mhartids` from the private thread CSRs and map each of these to various thread-duplicated components, such as program counters and register files. Thus, when modifying `mhartid` from "1" to "13", the thread ID of program counter 1 will automatically change to 13. By doing so, only the `mhartid` must be changed, whereas all other locations that the `mhartid` maps to change accordingly. This method has not been added but will be considered as future works.

As seen in Figure 3.3b, each private thread CSR is divided into 4 CSR modules; `mhartid`, trap CSRs, timing CSRs, and memory-mapped CSRs. As mentioned above, the `mhartid` register compares against the thread ID that wants access to the private thread CSRs. This ID match is required to access the other 3 CSR modules in the private thread CSR. Most of the trap CSRs and a few registers from the timing CSRs are based on the specifications given in the RISC-V documentation, which was described in Section 2.6.2 and will thus not be described here. The modification to the trap CSR compared to the RISC-V documentation is that the `interrupt_on_expire` and `deactivate_interrupt` instructions can enable or disable MTIE in the `mie` CSR. When an `interrupt_on_expire` instruction is executed, the 7th bit in the `mie` register is set, while `deactivate_interrupt` clears the same bit. As a result, these timer instructions can enable and disable the timer interrupt of its associated thread without manually configuring the CSRs.

An additional register, `mdpc`, was added to the timing CSR module. This register is used similarly to the `mepc` register in the trap CSR, where a return PC is stored when entering the interrupt handler. In contrast, the `mdpc` is used to hold the return PC of the `delay_until` instruction that is causing the thread to fall asleep. Thus, when the thread wakes up, it retrieves the PC from this CSR. Similarly, when the sleeping thread receives an interrupt it must handle, the value of `mdpc` is stored in `mepc` before entering the interrupt handler. When the interrupt handler executes the MRET instruction, the value of `mepc` is stored in the program counter, like a standard return from an interrupt. The thread will then perform the `delay_until` instruction, where it will return to sleep if the duration has not elapsed.

A different solution would be to update the program counter with the PC value residing in the memory stage during a `delay_until` instruction, similar to how a branch to PC + 0 would be performed in the execute stage. The advantage of this solution is that it removes the need for the additional register since it uses the program counter instead. This removal is possible because the program counter will stay inactive during sleep, significantly reducing the microarchitecture's complexity. The former solution was implemented in the microarchitecture, but the latter could be a better solution in future works.

The memory-mapped CSR module should contain all the boundaries for the memory-mapped registers that are private to the thread. Currently, only the upper and lower boundaries of the private data memory region are present here, named `mdmemupper` and `mdmemlower`, respectively. Software decides on these regions to make the microarchitecture as configurable as possible. Thus, the programmer's job is to decide the optimum upper

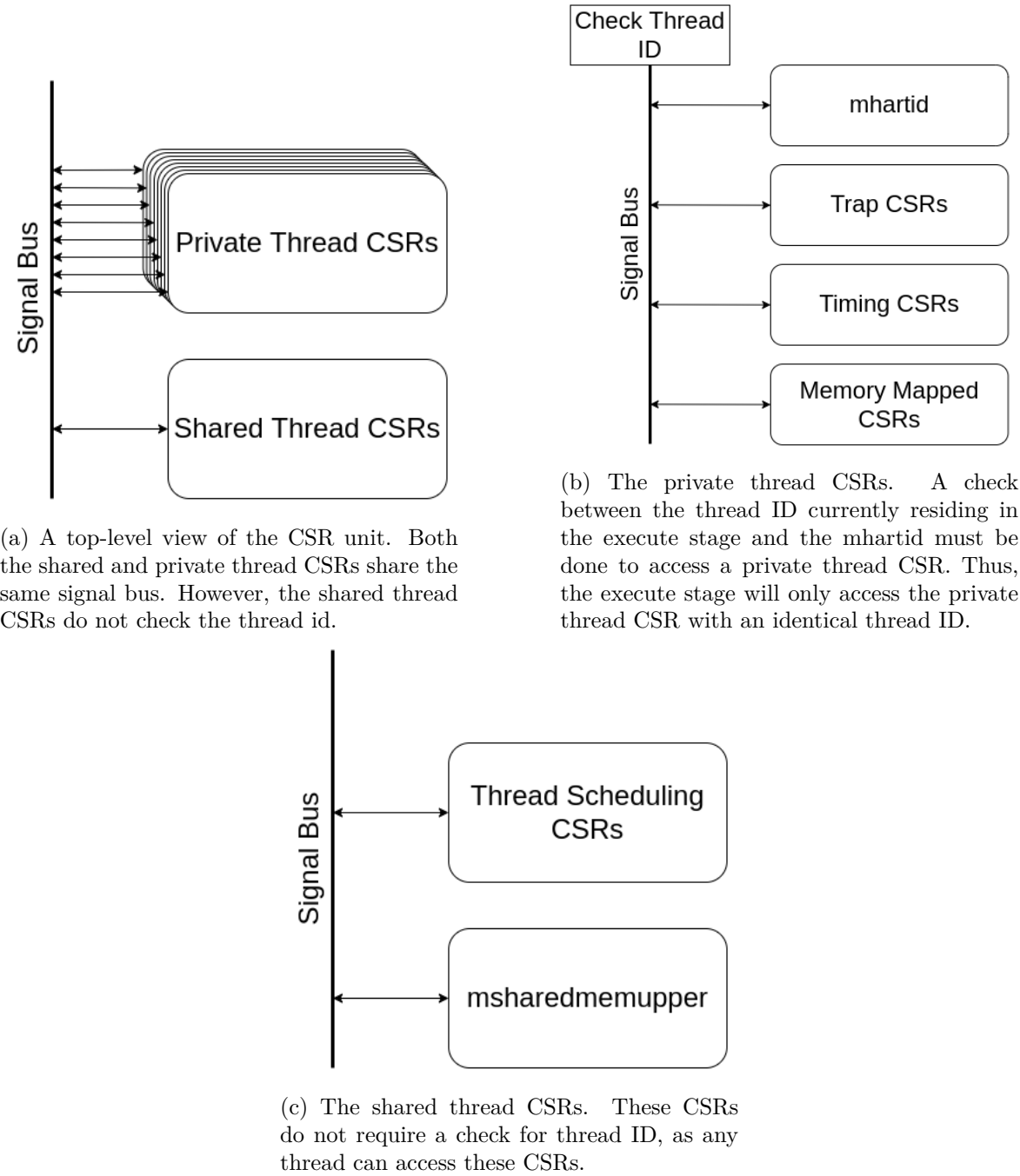


Figure 3.3: Models of the various CSRs available in the microarchitecture.

and lower boundaries for the private data memory regions. If these registers are not written with care, the memory regions of various threads could overlap. This overlap could result in various threads unintentionally sharing a memory region that is supposed to be private. Consequently, the programmer must consider this when selecting the memory boundaries for each thread. Although, this flexibility in selecting the boundaries makes it possible for some threads to share a part of their memory regions while others do not have access. By doing so, a few threads can have private data sharing instead of sharing with all threads like in the shared memory region.

The CSRs shared between the threads can be seen in Figure 3.3c. Within the shared CSR unit, there are two modules; the `msharedmemupper` register and the thread scheduling CSRs. As was mentioned previously, the components related to hardware thread scheduling will be discussed in Section 3.2.1. The `msharedmemupper` register is the upper boundary of the memory region that is shared between all threads. The shared memory region begins at address 0 and ends at the upper boundary specified by this register. By having this register, the programmer can decide how much of the memory should be shared.

In summary, the control and status registers mentioned in this section are new contributions and modifications to existing registers in the RISC-V specification. The CSRs that were modified achieved the original specifications still but had additional features added. The idea was to enable the timing instructions to swiftly set or clear the relevant bits in the CSRs necessary for timer interrupts. The new CSR contributions to the microarchitecture are added to complement the multithreaded feature. Here, CSRs are added to specify which regions in memory each thread can use, how the hardware thread scheduler should be configured, and a register to keep track of the PC when a thread is put to sleep.

## Trap Handler

The trap handler is responsible for performing the hardware operations necessary to place the software program in the interrupt handler. The hardware logic designed in this handler is based on the described operations performed when an interrupt is received. These design decisions are not described in detail in this section since the design is based on the functional description given in the RISC-V documentation. However, parts of the trap handler required some alteration to consider the use of multiple threads. These changes will be described in the following paragraphs.

The primary modification to the trap handler is how it fetches the PC that should be placed in the machine exception program counter (`mepc`). The trap handler fetches the PC of the current instruction and stores it in `mepc` while loading the interrupt PC into the thread's PC. An issue seen during the trap handler's design was during the exchange of PC values. Without a content verification, the value stored in the `mepc` register is the value present in the stage initializing the trap handling, which is the execute stage in this microarchitecture. However, if a branch has recently occurred, the execute stage will be empty because of a flush. Therefore, if the PC value in this stage is fetched, it will store the value "0" in the `mepc` register. Consequently, when the program returns from the interrupt handler, it will start from the first instruction in memory, resulting in a program restart. A solution is to check if the PC value in the execution stage is 0. However, this may prevent a valid instruction memory address from being used. As a result, flushing the entire pipeline stage does not give enough information to the trap handler to avoid fetching the PC of that stage.

An alternative method is to flush everything except the PC value in the pipeline stages. However, the trap handler must then view the other registers in the execute stage to verify whether the stage is flushed. Combinatorial logic must then be added to decide whether the PC present in the execute stage is valid. Additionally, if the execute stage is flushed, the trap

handler must check the prior stages for a valid PC. Thus, the same combinatorial logic must also be added to those stages, leading to considerable extra logic required to ascertain the status of the pipeline stages. Instead, an additional register can be added to each pipeline stage that holds the status of the stage. When the pipeline stage is flushed, the register is set logically high, making the trap handler aware that it can not fetch the PC of this stage. The trap handler will then look through each prior stage (fetch and decode) until it finds a stage where the register is logically low. If neither of these stages contains a valid PC value, the trap handler will fetch from the program counter instead.

Following the addition of the status registers, the trap handler can fetch the correct PC whenever running as a single-threaded microarchitecture. However, when there are other threads present, another issue emerges. As mentioned in Section 2.3, an advantage of fine-grained multithreading is its ability to avoid data hazards by having other threads use these cycles instead. However, the microarchitecture presented in this report has a fine-granularity, configurable thread schedule.

For example, a thread schedule where three branches occur can be seen in Figure 3.4, where the flushed cycles are colored red. The first branch performed by thread0 manages to avoid the data hazards by having thread1 and thread2 allocated to the following cycles. In comparison, the second and third branches do not manage to hide the data hazard. Here, thread1 has another thread cycle scheduled following the branch, resulting in a flush of the decode stage. Finally, the third branch caused by thread2 notices that the same thread is occupying the fetch stage, resulting in a flush.

Consequently, the flush status may tell the trap handler to fetch from a previous stage. The trap handler will then fetch the PC of the decode stage if it has not been flushed. However, the trap handler is unaware that a different thread could be occupying this stage, where it fetches the PC of a thread. Using the weather drone as an example would mean that the thread running the flight controls task could suddenly perform the weather sensing task. As a consequence, two threads are running the weather sensing task, leading to the complete removal of a highly critical task. As a result, a check for thread ID was added to each stage to ensure that the trap handler fetches the PC of the correct thread.

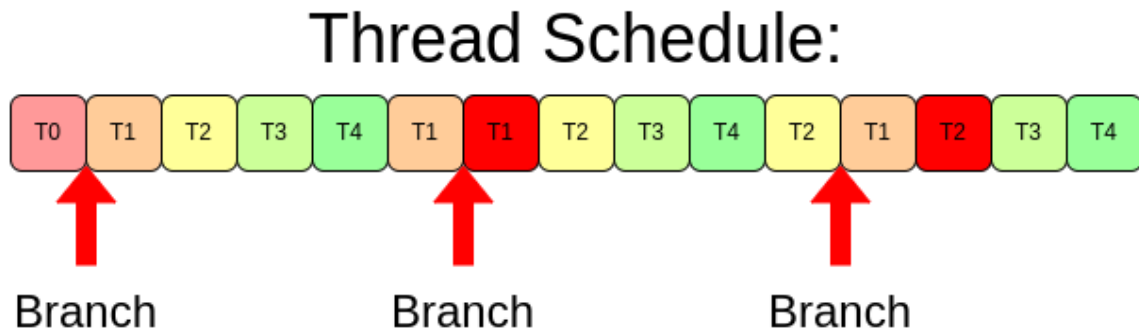


Figure 3.4: There is a possibility of branch latencies in this fine-grained technique when allowing other threads to use the spare cycles. The thread cycles colored red are stalled due to a flush of the stage holding an identical thread ID as the thread that is branching.

While the trap handler is based on the RISC-V specifications, there are some additions to the design. First, the flush status ensures that a flushed PC is not fetched into the mepc register by entering the trap handler routine. Finally, to complement the addition of multiple

threads interleaved every clock cycle, an additional check for a thread ID is required to ensure that an incorrect thread ID is not fetched.

### Flush Unit

In the single-threaded 5-stage RISC-V pipeline, the microarchitecture must flush the decode and fetch stage when a branch, interrupt, or similar is performed in the execute stage. This flush operation would only require using the branch control signal in the execute stage to signal a flush of the decode and fetch stages. However, when instructions from multiple threads are present in the pipeline, it could lead to a stage used by a different thread to be flushed. Consequently, the instructions from other threads in the pipeline are lost since no logic updates the program counters with the flushed PC. As a result, the other threads will have corrupted program execution due to lost instructions caused by a different thread's pipeline flush. Thus, a check for thread ID must be added to the flush logic of each stage to ensure that a thread does not accidentally flush the instructions of other threads.

Furthermore, the `delay_until` timing instruction, presented in Section 3.3, must also be able to flush the pipeline stages with an identical thread ID. However, compared to most other operations causing flushes, the timing instruction is performed in the memory stage. Additionally, the `delay_until` instruction must first set the time compare register, `mtimecmp`, before verifying whether the thread should be put to sleep. Therefore, two solutions were conceived for the flushing of the stages due to the `delay_until` instruction.

The first method, shown in Figure 3.5, compares the `sleep_time` value from the `delay_until` instruction while writing it to `mtimecmp`. Here, the input of the time compare unit uses an `is_delay_until` signal to select between the input or output of `mtimecmp`. The selected value is then compared against `mtime` to see if the thread should be sleeping. If the thread is put to sleep, the output of `mtimecmp` will be constantly selected since the `is_delay_until` signal stays low until the next `delay_until` instruction. As a result, the method will allow the thread to be put to sleep one cycle earlier, where only the stages prior to the memory stage must be flushed.

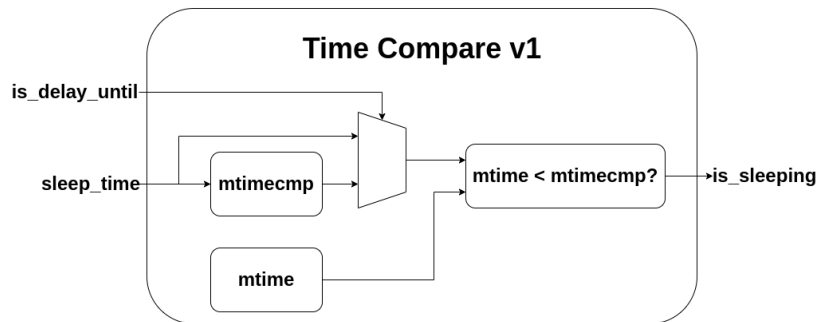


Figure 3.5: A multiplexer can be used to select between the input and output of the `mtimecmp` register to make it possible to check if the thread should be put to sleep before updating the register with the compare value. As a result, threads can be put to sleep faster.

The second method restricts the comparison to the output of `mtimecmp`, as shown in Figure 3.6. As a result, an additional cycle is required before the timer unit will know whether to put the thread to sleep. Consequently, an additional stage, namely the memory stage, must be flushed by the flush unit if the same thread occupies the stage following the `delay_until` instruction.

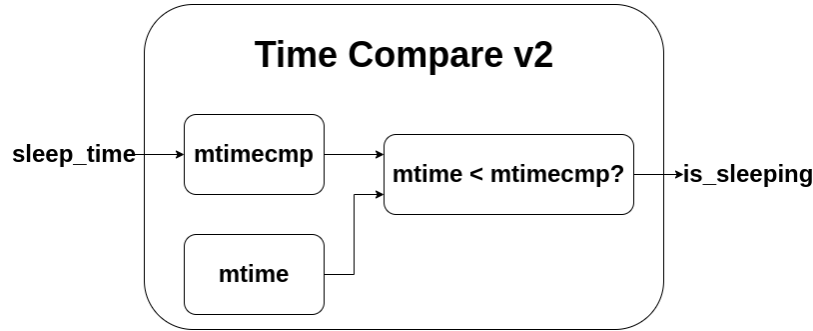


Figure 3.6: Only comparing the output of the mtimecmp register is a simple solution. However, the thread will require an additional cycle to fall asleep. This cycle is because the mtimecmp register must be updated before comparing the new value.

Of these solutions, the latter was selected as it was simple to implement the flush logic needed for the additional stages. Additionally, the added logic by the second solution is less than for the first solution. However, it could be beneficial to implement the first solution as it reduces the number of stages that must be flushed. Also, it would save a clock cycle each time a delay\_until operation is executed. Nonetheless, these minor benefits result in the least complex method being selected.

### Select Modules

The timeop select and mdpc unit seen in Figure 3.2 are minor units that were added to the control unit due to the inputs being signal buses containing signal lines for each thread. These units select which signal lines should be used based on the thread ID.

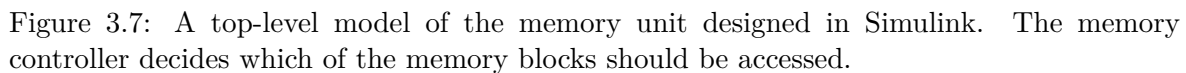
### 3.1.3 Memory Unit

The memory unit, shown in Figure 3.7, is responsible for anything related to the memory of the microarchitecture. The microarchitecture has three types of memory available; the data memory, the memory-mapped timer unit, and the memory-mapped input/output (MMIO) unit. Additionally, the microarchitecture has a memory controller implemented to restrict access to certain memory regions.

The data memory of the microarchitecture is 48 bytes in size and can be accessed using the memory instructions described in the RV32I ISA [67]. The limited data memory is because the compilation time of the Simulink model significantly increases with the size of the system. Thus, a small amount of memory was added to show a proof of concept, whereas an actual implementation would have more memory.

The MMIO unit consists of 3 memory-mapped registers; the output register, the input register, and the direction register. As the name implies, the output register is used to write to general-purpose I/O pins whenever the direction register specifies that the pin is used as an output pin. When the direction register specifies that the pin is an input, the input register will be updated with the value placed on the I/O. The program can then load the value into the register file and operate on the value of the I/O pin.

Because the timer unit is an integral part of the microarchitecture, it will be described in Section 3.3, while the memory controller will be described below.



In most microarchitectures, there is a requirement for some limitation to access various areas in memory. For example, various privilege levels may restrict access to certain memory regions to avoid a higher privilege level from being accessed by a lower privilege level. In this microarchitecture, there are no privilege levels other than the machine level. However, using multiple threads requires the same concept of restrictions on memory access. For example, consider the weather drone from earlier examples. Initially, the microarchitecture does not have any means to restrict the memory accesses for various threads. The weather sensing and flight control tasks will then be able to access the same memory region. Consequently, either task can overwrite the data stored in a memory location used by the other task, resulting in a data race between the tasks. Thus, the memory lacks spatial isolation where a task can modify important data or state information of another task.

The memory controller shown in Figure 3.7 utilizes this method of restricting access to physical memory regions. Here, the memory controller receives both the private and shared memory regions from the CSRs containing the upper and lower memory boundaries, as described in Section 3.1.2. Then, based on those memory regions, the controller will decide whether the thread can access the memory location.

Figure 3.8 shows a simplified model of the memory restriction the memory controller performs. The `mdmemupper` and `mdmemlower` values are fetched from the private thread CSRs of the thread trying to access a memory address. Simultaneously, the `mdmemshared`

value is fetched from the shared CSRs. These values are compared against the address the thread is trying to access. When both the `mdmemlower` and `mdmemupper` comparisons are valid, the thread is allowed access to the address within its private memory region. On the other hand, if either of these compare units returns a false value, then the address is not within the private memory region of the thread. The other possibility for letting the thread access the memory address is through the `mdmemshared` comparison. When this comparison returns true, the memory address is within the shared memory region. Both the private and shared memory regions are used to produce a mask signal. When the mask signal is true, it allows the `is_store` and `is_load` signals to pass through. In contrast, the `is_store` and `is_load` signals are cleared if the mask is a logical '0'.

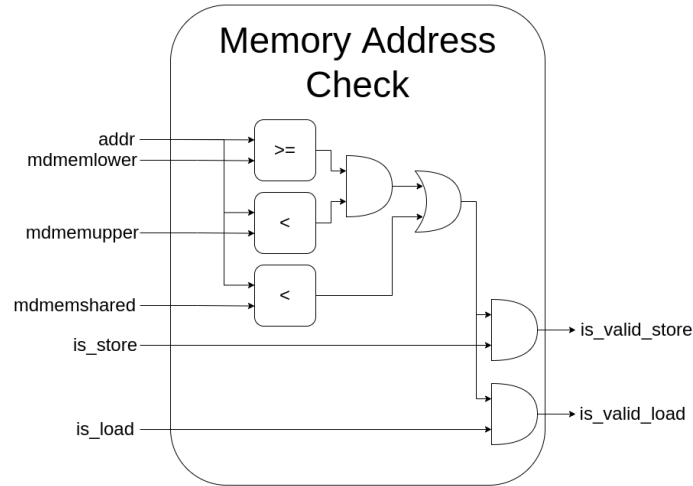


Figure 3.8: A simple solution to restrict access to certain memory regions. The address of the memory operation is compared against the regions to produce a mask signal. This masking signal decides whether the `is_store` and `is_load` signals should be forwarded to the memory.

Adding the memory controller to the microarchitecture makes it possible to restrict memory access to specific threads. This restriction provides the microarchitecture with an increase in spatial isolation. As a result, there is an increase in the predictability of task execution where thread-specific data can be unavailable for other threads.

## 3.2 Hardware Thread Scheduler

The hardware thread scheduler designed in Simulink can be seen in Figure 3.9. This scheduler consists of seven modules: hardware thread scheduling CSRs, main scheduler logic, main ring buffer, thread select logic, SRTT scheduler logic, SRTT ring buffer, and stall logic. The configuration CSRs are responsible for selecting the type of hardware threads to use and how these threads should be scheduled. The main scheduler logic and main ring buffer combined are referred to as the main scheduler, while the SRTT scheduler logic and the SRTT ring buffer refer to the SRTT scheduler. Here, the scheduler logic is responsible for deciding the length of the repeating sequence and what values to load the various slots in the ring buffer. The thread select logic is responsible for choosing which of the ring buffers to output from the hardware thread scheduler to the fetch stage of the RISC-V pipeline. Finally, a stall logic block is added in case a cycle is left unused, and there are no other hardware threads available that can take these thread cycles.

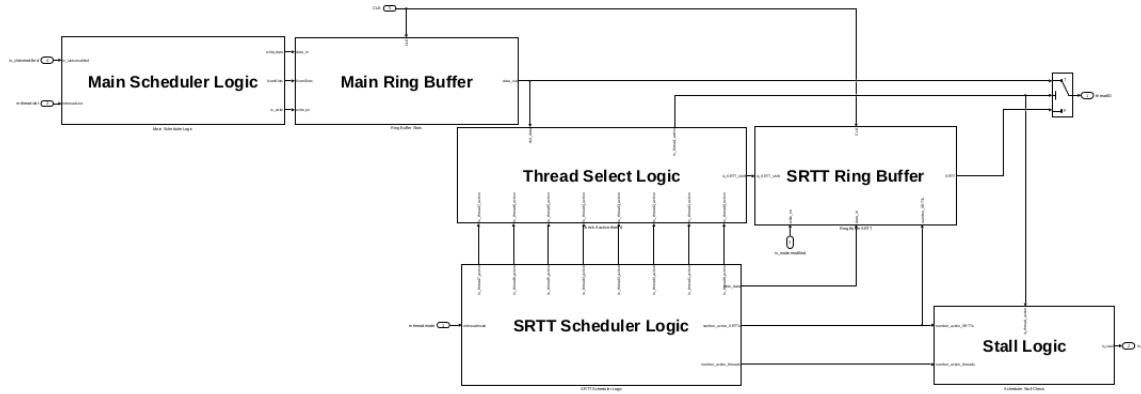


Figure 3.9: A top-level view of the hardware thread scheduler designed in Simulink. The design consists of two schedulers; the main and SRTT schedulers. The main scheduler comprises the main scheduler logic and main ring buffer, while the SRTT scheduler consists of the SRTT scheduler logic and SRTT ring buffer. The thread select logic decides which scheduler should have access to the pipeline. The stall unit ensures that the pipeline only uses active thread cycles.

### 3.2.1 Hardware Thread Scheduling CSRs

The hardware thread scheduling CSR module is the only module residing outside the hardware thread scheduler in the Simulink model seen in Figure 3.9. Instead, the module is placed with the other CSRs shared between threads, as seen in Figure 3.3c. This module contains the `mthreadslot` and `mthreadmode` CSRs described in Section 2.6.2. As a result, these CSRs provide the microarchitecture with a configurable hardware thread scheduler that can be programmatically adjusted through software. Thus, the programmer can decide how many threads that should be scheduled (0 to 8 threads possible), whether the threads are SRTTs or HRTTs, and whether the threads are awake or asleep.

Following the configuration of the scheduling CSRs, two control signals indicate that the hardware thread scheduler must update the ring buffers. To reduce the signal wiring and logic of the system, any modifications that should be done to the thread sequence or the mode of a thread should be configured through the CSR module. For example, any run-time changes to the mode of a thread, such as placing a thread in sleep mode, must update the `mthreadmode` CSR prior to the hardware thread scheduler. Consequently, it would require an additional clock cycle compared to updating `mthreadmode` and the hardware thread scheduler simultaneously. This extra cycle is because the ring buffers will not be updated until the cycle after the updated CSR. However, this added latency is negligible to the system performance since a sleeping thread will generally stay asleep for many cycles.

The modules presented below will be described in detail using an example configuration of the `mthreadslot` and the `mthreadmode` CSRs, shown in Figure 3.9. This example is by no means a good representation of how these CSRs should be configured in a mixed-criticality system, but it is used to showcase the various features of the hardware thread scheduler.

### 3.2.2 Main Scheduler

The main scheduler is the part of the hardware thread scheduler that performs all scheduling based on the data stored in `mthreadslot`. That is, it uses all of the active slots present in the `mthreadslot` CSR to create an active round-robin scheduler. The active slots in this round-robin scheduler can either contain a thread ID in the range of 0 to 7 or be classified

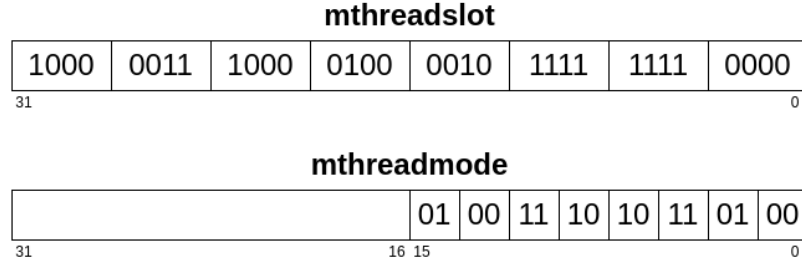


Figure 3.10: An example configuration of the hardware thread scheduler.

as a soft slot when containing the value 8. As the name implies, the main scheduler contains the primary thread schedule for the hardware thread scheduler. Thus, any time the main scheduler outputs a location in the ring buffer with a valid thread ID that is awake, the thread ID will be forwarded to the fetch stage of the RISC-V pipeline. However, when the main scheduler outputs a location containing a sleeping thread ID or a soft slot, the SRTT scheduler will gain access to the output.

### Main Scheduler Logic

The main scheduler logic contributes to the main scheduler by configuring the main ring buffer based on the information retrieved from the mthreadslot CSR. The module continuously reads the value stored in the mthreadslot, performs some operations, and outputs the signals NumSlots and MainSchedule. The NumSlots signal tells the main scheduler's ring buffer the sequence's length. That is, the signal configures the number of registers that the ring buffer should use based on this value, ranging from 0 to 8. Meanwhile, the MainSchedule signal contains the rearranged data received from the mthreadslot CSR. To explain the main scheduler logic, a simplified model that abstracts away the hardware design to focus on the functionality can be seen in Figure 3.11.

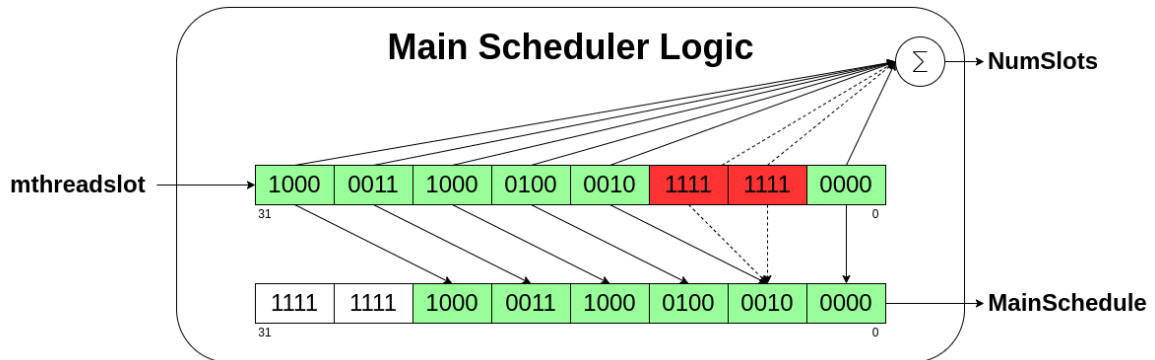


Figure 3.11: The slot CSR is fetched into the hardware thread scheduler when modified. Each slot is checked for a value less than or equal to 8. A slot is placed in the main schedule if it contains such a value. Otherwise, it is discarded. The number of slots with such values is counted up to provide the ring buffer with the correct length of the repeating sequence.

The example shown in Figure 3.11 takes the example mthreadslot configuration from Figure 3.10 as its input value. To show the difference between enabled and disabled slots,

they are colored green and red, respectively. The white slots in the figure are irrelevant data values placed in the MSBs to indicate unused slots.

When the main scheduler receives a value from the `pthreadslot` CSR, it performs 2 operations on the data: counts the number of enabled slots and rearranges them. The module must check the validity of each slot to perform these operations. This check is done by having an active enable signal only when a slot's value is less than or equal to 8. As can be seen, the slots that are colored red contain a value greater than 8. Thus, the enable signal for these slots will stay low while the others are set high.

For the module to obtain the `NumSlots` value, a summation of all the enable signals is performed. Here, the sum is the total number of signals that are set high. In the example, it can be seen that there are six green slots in the `pthreadslot` value, resulting in the value six on the `NumSlots` output.

The complex functionality of the main scheduler logic resides in the rearrangement of the slots provided by the `pthreadslot` CSR. Similar to the `NumSlots` signal, the `MainSchedule` signal requires an enable signal from each slot to indicate whether the slot is disabled or enabled. As can be seen in Figure 3.11, all slots with a value that indicates that the slot is enabled (green) are mapped to `MainSchedule`, while other slots (red) are discarded. The unit responsible for this functionality is called the mapping unit and is responsible for mapping all enabled slots from `pthreadslot` to `MainSchedule`.

When mapping the slots of `pthreadslot` to `MainSchedule`, the mapping unit starts by mapping the LSB. Since `slot0` contains a valid thread ID, the slot can be mapped to `slot0` of `MainSchedule`. In contrast, when trying to map `slot1` of `pthreadslot` to `slot1` of `MainSchedule`, the enable signal alerts the mapping unit that the slot is disabled. As a result, the mapping unit will continue to the next slot of `pthreadslot` and check if `slot2` is enabled. Again, the mapping unit notices that this slot is disabled and moves on to the next slot. Finally, when the mapping unit checks `slot3`, it notices that the enable signal is set high, and the slot can be mapped to `slot1` of `MainSchedule`. The mapping unit will then continue mapping `slot4` of `pthreadslot` to `slot2` of `MainSchedule`. This process is continued until all the slots of `pthreadslot` have been either mapped or discarded. Once all the slots are mapped, the mapping unit will append the remaining empty slots of `MainSchedule` with 1's, as these slots are inactive and thus irrelevant to the ring buffer.

As a result of this example, the main scheduler logic will write the ring buffer with the content of `MainSchedule`, shown in Figure 3.11, and configure the length of the ring buffer to the value of `NumSlots`.

### Main Ring Buffer

The ring buffer is the module responsible for producing a repeating pattern of thread IDs for the hardware thread scheduler to forward to the fetch stage of the RISC-V pipeline. The ring buffer consists of eight 4-bit registers connected as a shift register, where the final register (Reg7) is connected to the first (Reg0). To make the ring buffer more flexible, each register has the possibility of looping back to Reg0, as seen in Figure 3.12. This figure shows a configurable ring buffer that is used by both the main and SRTT schedulers. Here, a value placed on the input signal called `Length` indicates which register should loop back to Reg0. Additionally, the ring buffer provides a 32-bit signal called `Schedule`, and a write enable signal called `is_modified` to update the content. For example, if `Length` is set to 4, the lower 16 bits of the `Schedule` signal will be used, as Reg3 will loop back to Reg0, thus discarding the values stored in registers 4 through 7. Thus, the value placed in `Length` must correspond to the part of `Schedule` that should be used. Finally, the ring buffer has an output signal called

Reg\_Value. This signal outputs the value stored in the upper-most register. That is, if Reg3 loops back to Reg0, then it is Reg3 that is connected to Reg\_Value.

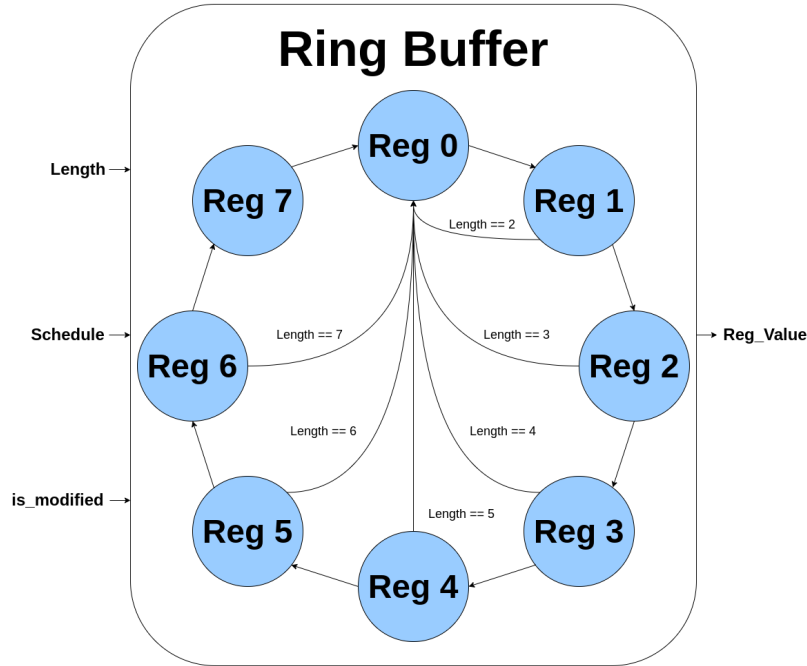


Figure 3.12: A simple model of the generic ring buffer that visualizes how the ring buffer functions. The Length signal decides which of the registers should connect to the input Reg0. The Schedule signal holds the content that is written to the registers in the ring buffer using the is\_modified signal. The upper-most register that connects back to Reg0 is also the register provided to the Reg\_Value output signal.

As mentioned above, the main scheduler uses the configurable ring buffer to produce a repeating pattern. To implement it in the main scheduler, the Length and Schedule signals must be connected to the NumSlots and MainSchedule signals of the main scheduler logic. To update the main ring buffer, a signal called is\_mthreadslot\_modified is connected to is\_modified. This signal looks for a change in the content of the mthreadslot CSR, where it goes high when there is a change. Using the example output of the main scheduler logic, shown in Figure 3.11, the resulting main ring buffer is as shown in Figure 3.13. As can be seen, because there are only 6 slots enabled, there are only 6 registers used in the ring buffer, where the other registers are disregarded. Additionally, the values stored in the used registers correspond to the 6 enabled slots of the mthreadslot CSR mapped onto MainSchedule.

### 3.2.3 SRTT Scheduler

Compared to the main scheduler, the SRTT scheduler creates a repeating sequence based on the values stored in the mthreadmode CSR. For example, if mode0 and mode2 are configured as active SRTTs, then thread0 and thread2 will be placed in the SRTT scheduling sequence. Additionally, the SRTT scheduler is responsible for keeping track of which threads are awake and asleep. This information is forwarded to a different module where it is selected which of the ring buffers that should have access to the output of the hardware thread scheduler. Furthermore, the SRTT scheduler keeps track of the number of active threads and SRTTs in the scheduler. These values are forwarded to other modules to configure the SRTT ring

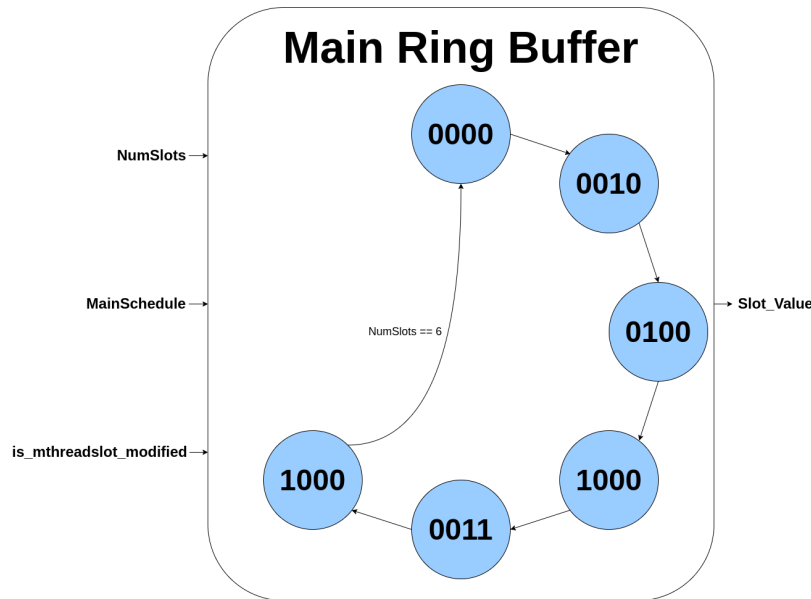


Figure 3.13: An example configuration of the main ring buffer using the values from the main scheduler logic. Notice how the NumSlots (Length) signal selects which register should loop back to the first register.

buffer and potentially stall the pipeline. Section 3.2.3 and Section 3.2.3 will describe the mentioned functionalities further.

### SRTT Scheduler Logic

The SRTT scheduler logic is the module responsible for deciding the status of each thread by reading the content of the mthreadmode CSR. The SRTT scheduler logic provides two enable signals for each thread, the first specifying an active thread and the second an active SRTT. The enable signals indicating an active thread are used as output signals and are given the name `is_threadn_active`, where `n` is the thread ID. These signals are connected to the thread select logic module, described in Section 3.2.4, to select which ring buffer that should have access to the output of the hardware thread scheduler.

The SRTT scheduler logic performs a similar summation to the NumSlots in the main scheduler logic. However, this unit performs two distinct summations of the enable signals; a summation of the total number of active threads (NumActiveThreads) and a summation of the total number of active SRTTs (NumActiveSRTTs). These signals are connected to the stall logic module, described in Section 3.2.5. Additionally, the NumActiveSRTTs are connected to the SRTT ring buffer to configure the length of the ring buffer.

Like the main scheduler logic, the SRTT scheduler logic contains a mapping unit responsible for mapping values onto the schedule signal, called SRTTSchedule. However, in contrast to the main scheduler logic, the value of the mthreadmode is not directly mapped to the SRTTSchedule. Instead, the mapping unit will use the SRTT enable signals to retrieve the thread IDs from a constant vector containing the thread IDs from 0 to 7. To fetch the correct thread ID, the location of the mode value is used as index. That is, `mode0` is at location 0 and will thus collect `thread0`, `mode1` will collect `thread1`, etc. As a result, when the SRTT enable signal is set for a mode, the thread ID of that mode will be mapped onto SRTTSchedule. For example, if `mode2` contains the value "10", then the SRTT enable signal is set high. Thus, the value "0010" will be retrieved from the thread ID vector and mapped

to a slot in the SRTTSchedule. This process will continue until all active SRTTs are mapped onto the SRTTSchedule.

By using the content of mthreadmode CSR value the example configuration shown in Figure 3.10, a simple model of the SRTT Scheduler Logic, shown in Figure 3.14, can be used to explain how the module operates. First, the content of mthreadmode is retrieved and divided into eight 2-bit mode signals, mode0 through mode7. As mentioned above, each bit of these mode signals produce an enable signal. Here, the LSB specifies if the thread is active while the MSB indicates if the thread is an SRTT or HRTT. As can be seen in Figure 3.14, the mode signals have different colors based on these bits. For example, the signal is colored red whenever the LSB indicates that a thread is sleeping. However, if the thread is active, there are two possible outcomes based on the MSB; either the thread is an HRTT (white) or it is an SRTT (blue).

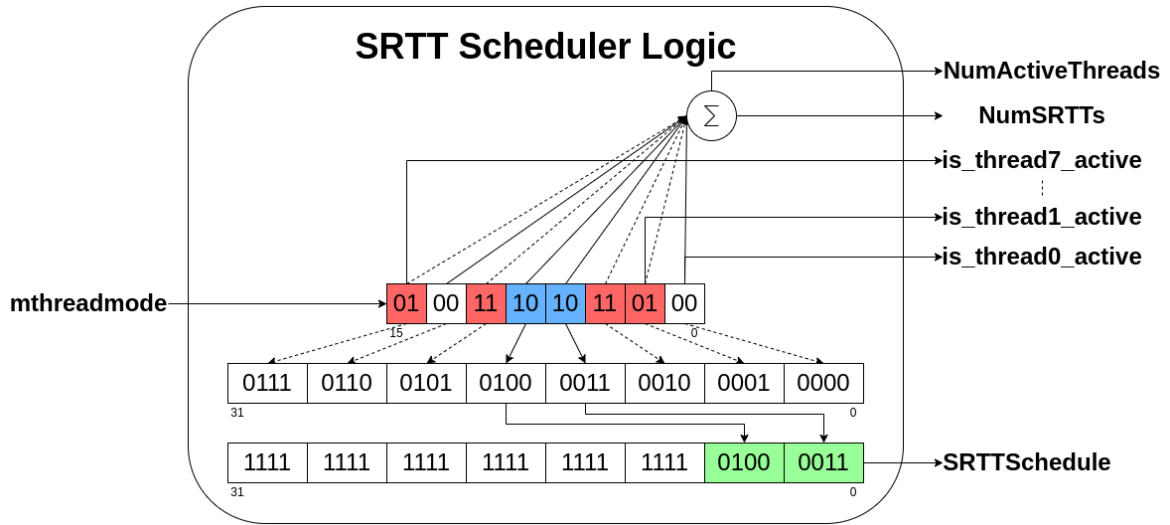


Figure 3.14: The mode CSR is fetched into the hardware thread scheduler when modified, where the content is partitioned into 2-bit mode signals. These mode signals locate the active threads and identify these threads as either soft or hard. The thread IDs of the active SRTTs are further mapped onto the SRTTSchedule signal.

As was mentioned above, the NumActiveThreads signal provides the number of active threads while the NumSRTTs signal yield the number of active SRTTs. Thus, the NumActiveThreads will sum all signals colored both white and blue while the NumSRTTs will sum only blue signals. Similarly, the **is\_threadn\_active** signals consist of the same enable signals as the NumActiveThreads, meaning that all signals colored white or blue have their **is\_threadn\_active** signal set high.

The way the mapping unit of the SRTT scheduler logic functions can be seen below the colored mode signals in Figure 3.14. Here, there are two vector signals containing eight 4-bit slots. The first vector, the thread ID vector, contains all thread IDs available in the microarchitecture. The mapping unit uses the thread ID vector similarly to how mthreadslot was handled by the main scheduler logic mapping unit in Section 3.2.2. However, instead of looking for a valid slot, the mapping unit looks for an active SRTT. If both of these enable signals tell the mapping unit that the thread is an active SRTT, then the thread ID can be mapped to the SRTTSchedule. This mapping can be seen for thread IDs 3 and 4 in the example, colored in green. Finally, the mapping unit performs the same padding operation on the locations in the SRTTSchedule that are left unused.

### SRTT Ring Buffer

The ring buffer of the SRTT scheduler uses the generic ring buffer design described in Section 3.2.2. However, in contrast to the main ring buffer, the SRTT ring buffer should only be shifted when it has access to the output of the hardware thread scheduler. Thus, the SRTT ring buffer must stay dormant whenever it is not accessed to allow each SRTT the same amount of unused thread cycles. Otherwise, occasionally, only some of the SRTTs can utilize the unused thread cycles may occur. For example, the SRTT scheduler is allowed access to the pipeline every fourth clock cycle while there are only 2 SRTTs present in the ring buffer. Therefore, if the ring buffer is not kept dormant, the SRTT that is allowed access to the output will always be the same. To solve this issue, an additional signal is added to the SRTT ring buffer compared to the generic design. Here, the signal tells the ring buffer when the SRTT scheduler has access to the output, thus allowing the ring buffer to shift. Meanwhile, when the signal is low, the SRTT scheduler does not have access, postponing the shift operation. As a result, a ring buffer that is only shifting when the ring buffer is accessed was designed.

Using the example values from Section 3.2.3, a simple model of the SRTT ring buffer was made, shown in Figure 3.15. The module receives the length and the sequence values from the SRTT scheduler logic, known as NumSRTTs and SRTTSchedule, respectively. Using these values the structure of the ring buffer is as shown, where threads 3 and 4 are scheduled. To write these values into the SRTT ring buffer, an `is_mthreadmode_modified` signal is used that is set high whenever the content of `mthreadmode` changes. Finally, the ring buffer will stay dormant until `is_SRTT` is set high, indicating that the SRTT scheduler was accessed. As a result, the pattern 0100, 0011, 0100, 0011, ... will be retrieved from the SRTT scheduler when the hardware thread scheduler accesses this scheduler.

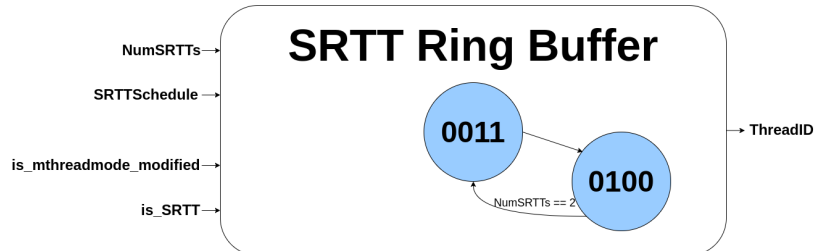


Figure 3.15: An example configuration of the SRTT ring buffer using the values from the SRTT scheduler logic. Compared to the main ring buffer, the SRTT ring buffer uses an additional signal, called `is_SRTT`, to only shift when the ring buffer is accessed.

#### 3.2.4 Thread Select Logic

The thread select logic is the module responsible for selecting which scheduler that should have access to the output of the hardware thread scheduler. This module is one of the least complex parts of the hardware thread scheduler and the functionality can be seen in the flow chart in Figure 3.16. The thread select logic receives the `Slot_Value` signal from the main scheduler, where the value is checked to see if it is an SRTT slot. If the slot is an SRTT slot, the `is_SRTT` signal is set high. On the other hand, if the slot contains a thread ID, a check to see if the thread is active is performed. Thus, the module retrieves the `is_threadn_active` signals from the SRTT scheduler logic to compare against the thread ID. For example, if `Slot_Value` contained `thread5`, then `is_thread5_active` would be checked. If this signal tells

the module that the thread is active, the `is_main` signal will be set high. As a result, the main thread scheduler is allowed access to the output of the hardware thread scheduler. In comparison, if the signal tells the module that the thread is sleeping, the `is_SRTT` signal will be set high, resulting in the SRTT scheduler having access. Thus, the two outcomes of the thread scheduler logic will decide which of the thread schedulers should have access to the output of the hardware thread scheduler.

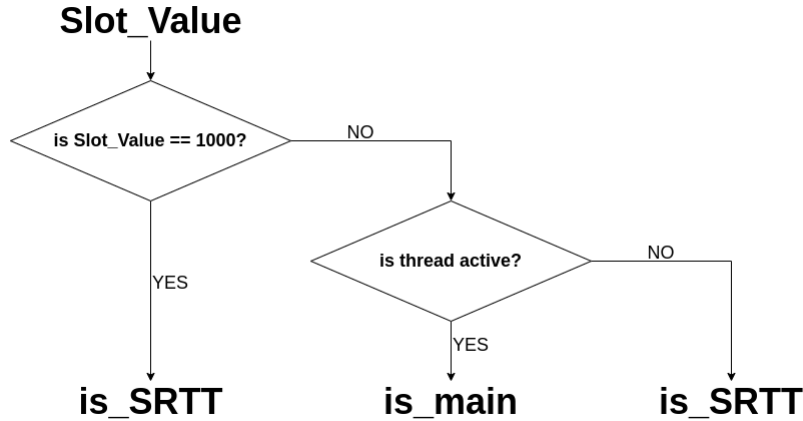


Figure 3.16: Flowchart representation of the functionality of the thread select logic. The module produces two outcomes used to select between the main and SRTT scheduler.

### 3.2.5 Stall Unit

When there are no active threads present in the hardware thread scheduler, or if a thread is sleeping and there are no SRTTs available, a problem arises in the hardware thread scheduler. Although the hardware thread scheduler knows that there are no active threads available that cycle, the RISC-V pipeline does not have this knowledge. As a result, the pipeline can receive a thread ID without knowing that it is inactive. Instead, a "bubble" should be inserted into the pipeline to indicate that the current cycle is left unused. This insertion of an empty cycle can be achieved by adding a module to the hardware thread scheduler that looks for these situations. Thus, the stall unit, shown in Figure 3.17, was added to the design.

As can be seen in the figure, the stall unit looks for two conditions. In the first condition, the thread select logic has set the `is_SRTT` signal high. When this signal is set high, the stall unit must verify that there are SRTTs available in the SRTT ring buffer. Thus, if `NumSRTTs` is zero, then the `is_stall` signal must be set high, indicating to the fetch stage that it should stall the pipeline that cycle. In the second condition, if no threads are active in either scheduler, then the `is_stall` signal should be set high.

### 3.2.6 Hardware Thread Scheduler Example

Following the description of each module, a simple model of the hardware thread scheduler using the same example was made, shown in Figure 3.18. The thread IDs located in the SRTT scheduler are colored green to make it easier to see in the timing diagram, shown in Figure 3.19, which of the schedulers that have access to the ThreadID output signal. Next, inside the main scheduler there is a red thread ID, indicating that the thread is sleeping. Finally, the blue values in the main scheduler indicates that those locations are soft slots.

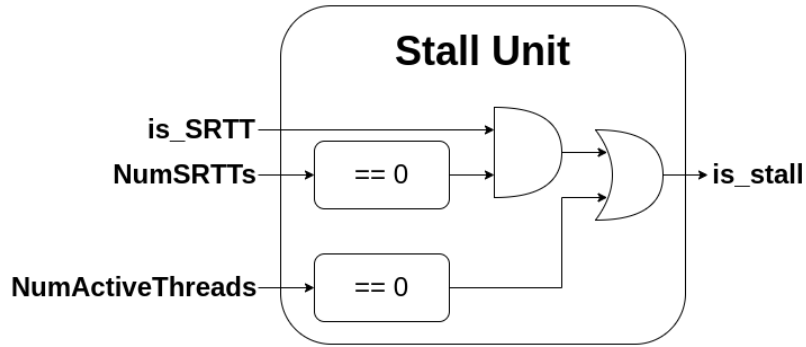


Figure 3.17: The stall unit verifies whether any threads are available that thread cycle. The unit will stall the pipeline stage if there are no threads available.

The thread select logic will consider both the red and blue colored slots as unused slots in the main scheduler, thus allowing the SRTT scheduler to use those cycles. As can be seen in the timing diagram, the thread ID given to the RISC-V pipeline alternates between the main and SRTT schedulers. Also, a thing to note is that even though both thread3 (0011) and thread4 (0100) are SRTTs, they can still have fixed slots allocated in the main schedule while still using the thread cycles of sleeping and soft slots.

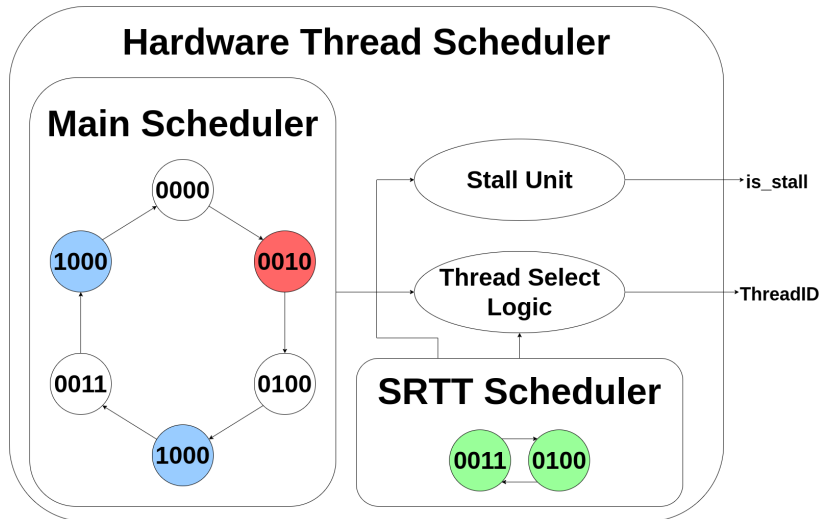


Figure 3.18: An example model of the Hardware Thread Scheduler. The Thread Select Logic decides whether the slot in the Main Scheduler should be allocated to the Main or SRTT Scheduler. The blue slots indicate that it is by default allocated to the SRTT Scheduler due to the slot's content. The red slot indicates that the thread ID allocated to that slot is sleeping, thus distributing those thread cycles to the SRTT Scheduler. The green slots are used to identify thread IDs that are allocated to spare thread cycles. Notice how the Main Scheduler has one colored slot between white slots. As a result, every other thread cycle will be allocated to the SRTT Scheduler

Compared to the sequence shown in Figure 3.19, situations, where all SRTTs are sleeping, may occur. Figure 3.20 shows an example of how the thread scheduling of the hardware thread scheduler would look when the SRTTs are sleeping. Here, the thread select logic and the SRTT scheduler will notify the stall unit every thread cycle when there is an unused slot

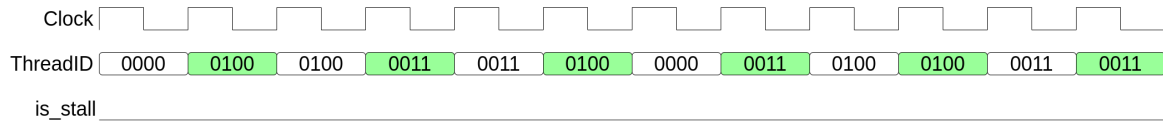


Figure 3.19: A timing diagram of the output of the Hardware Thread Scheduler. Notice how the Thread Select Logic has allocated every other cycle to the SRTT Scheduler.

in the sequence. The stall unit will then set the `is_stall` signal high, alerting the fetch stage of the pipeline that there are no threads available. The fetch stage will then stall the pipeline instead.

The updated timing diagram of the signals provided to the pipeline can be in Figure 3.21. Since thread0 (0000) is an HRTT, meaning that it only uses cycles allocated explicitly to that thread, it is incapable of using the unused cycles. As a consequence, 5 out of 6 thread cycles are left unused. This schedule is an undesirable situation that leads to poor hardware resource utilization of the microarchitecture. As a result, it is important to allocate the threads properly to avoid such circumstances. However, it may be unavoidable in some situations, where no other tasks need processing except for the active HRTTs.

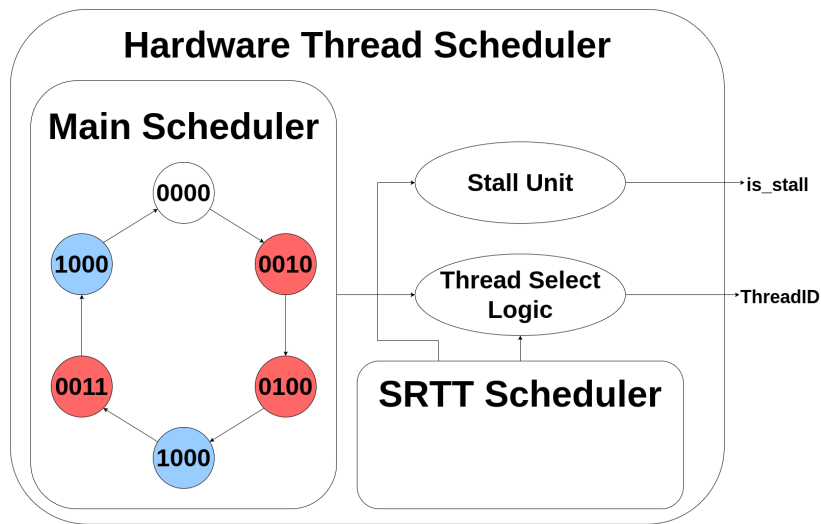


Figure 3.20: Continuation on the example model of the Hardware Thread Scheduler where the SRTTs are sleeping. Thus, there are no SRTTs available that can use the spare cycles, resulting in all thread cycles to be stalling except for the cycles used by the active HRTT.

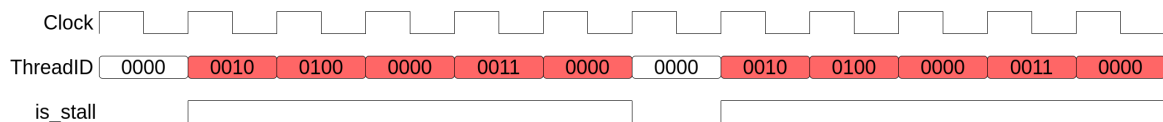


Figure 3.21: The timing diagram of the Hardware Thread Scheduler when there are no SRTTs available. Most of the cycles are now stalling, resulting in the `is_stall` signal being a logical '1' for 5 out of 6 cycles.

### 3.3 Timer Unit

The timer unit of the microarchitecture was designed to provide each task running on a thread with an awareness of time. As a result, the programmer can add real-time timing constraints to the software. These real-time constraints include placing threads in a sleeping state whenever needed, interrupting task execution after a certain amount of time, and branching to some specific location if task takes too long. To access this unit, the microarchitecture provides four timing instructions similar in functionality to the once mentioned in Section 2.2. Section 3.3.1 will describe further how the timing instructions are implemented and what operations they perform in the microarchitecture. Finally, section 3.3.2 will go into detail on the designed memory mapped timer unit.

#### 3.3.1 Timing Instruction Set

The timing instruction set, shown in Figure 3.22, was designed to provide timing instructions capable of doing multiple things simultaneously. That is, in the description of `mtime` and `mtimecmp` in the RISC-V documentation, the trap handling CSRs mentioned in Section 2.5.2 and the value written to `mtimecmp` is set separately [71]. Instead, the `interrupt_on_expire` instruction does both simultaneously; it sets the MTIE bit in the MIE CSR, and stores the value of `rs2` in the `mtimecmp` register.

The `mtimecmp` register is also used by the `delay_until` instruction to sleep a thread for a certain amount of time, specified by `rs2`. Additionally, this instruction will set a register indicating that the `mtimecmp` register is used to sleep a thread. A different solution would be to have separate registers for sleeping and timer interrupt. Initially, this solution was viewed as unnecessary as both functionalities will never be performed simultaneously. However, an advantage of this solution is that the RISC-V convention can be used for both timer interrupts and sleeping threads. Thus, there is no requirement for adding additional instructions to the compiler. Despite this advantage, the method was not implemented as it seemed unnecessary at the time. Although, it may be a good idea for future work to place these functionalities on separate `mtimecmp` registers.

In addition to the `delay_until` and `interrupt_on_expire` instructions, two other instructions are added to the design; `deactivate_interrupt` and `get_time`. The `deactivate_interrupt` instruction is a trivial instruction that clears the MTIE bit in the MIE CSR. The `get_time` instruction retrieves the content of the `mtime` register, and stores it in the `rd` register specified by software. Thus, the task can know the current time using this instruction, making it possible to place accurate timing bounds on tasks.

Looking at the `delay_until`, `interrupt_on_expire`, and `get_time` instructions in Figure 3.22, it can be seen that bits 31-25 are identical. Additionally, bits 11-7 for `delay_until` and `interrupt_on_expire`, and bits 24-20 for `get_time` are identical. Furthermore, notice how `delay_until` and `interrupt_on_expire` both use the `rs2` register while `get_time` uses the `rd` register. The reason behind this is that `get_time` is based on the load instruction of the RV32I instruction set. Here, bits 31-25 and 24-20 combined produce the offset, while `rd` is the destination register and `rs1` is set to 0.

Similarly, the `interrupt_on_expire` and `delay_until` instructions are based on the store instruction of the RV32I instruction set. As a result, the reading and writing of the timer registers are equivalent in behavior to the instructions seen in Listing 3.2. Here, the value 804 is the base address of the timer unit, which was selected as the offset of the timer instructions. To access the `mtimecmp` registers, the memory controller adds the thread ID to the base address to access the correct memory-mapped `mtimecmp` register.

delay_until rs2																											
31-27					26-25		24-20			19-15				14-12			11-7					6-2				1-0	
0	0	1	1	0	0	1	rs2			0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	1

interrupt_on_expire rs2																													
31-27					26-25		24-20			19-15				14-12			11-7					6-2				1-0			
0	0	1	1	0	0	1	rs2			0	0	0	0	0	0	1	1	0	0	1	0	0	1	0	1	1	0	1	1

deactivate_interrupt																													
31-27					26-25		24-20			19-15				14-12			11-7					6-2				1-0			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1

get_time rd																													
31-27					26-25		24-20			19-15				14-12			11-7					6-2				1-0			
0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	rd			0			0	0	1	0	1	1

Figure 3.22: The Timing Instruction Set that was added to the microarchitecture’s Instruction Set Architecture (ISA). These instructions are used to provide timing capabilities that are compatible with the hardware thread scheduler.

Listing 3.2: timinginstructions.a

```

1 SW rs2, 804(0) //interrupt_on_expire/delay_until rs2
2 LW rd, 804(0) //get_time rd

```

By basing the fields of the timing instructions on the SW and LW instructions, the amount of logic required can be reduced significantly. For example, the get\_time instruction is capable of reusing the forwarding logic used by the LW instruction, thus reducing hardware complexity. However, the timer instructions require some additional logic to configure the required registers.

### 3.3.2 Memory Mapped Timer Registers

As mentioned in Section 3.1.3, the memory-mapped timer registers are located in the memory unit. This unit is responsible for making each thread-aware of time by allowing a task to put the thread to sleep or perform a timer interrupt. Figure 3.23 shows a simple model of the memory-mapped timer registers. Here, the is\_store signal is used by interrupt\_on\_expire, delay\_until, and SW to write the registers. More specifically, the interrupt\_on\_expire and delay\_until uses is\_store to write the mtimecmp register of a specific thread, while SW can write any register based on the address. Similarly, the is\_load signal is used by the get\_time instruction to read the mtime register, while LW reads the content of any register based on the address.

The delayuntil\_request signal in the figure is an enable signal connected to the mtimecmp registers that are set high whenever a delay\_until instruction is executed in the memory stage. The data line contains the data that is written or read from all the registers in the memory-mapped timer unit by using the is\_store and is\_load signals, respectively.

The address line is the final signal connected to the memory-mapped registers of the timer unit. This signal is used to select which memory-mapped registers the program wants to access. As mentioned in Section 3.3.1, the timer instructions have the base address of the memory-mapped timer unit contained in the offset value of the instruction. That is, the timer

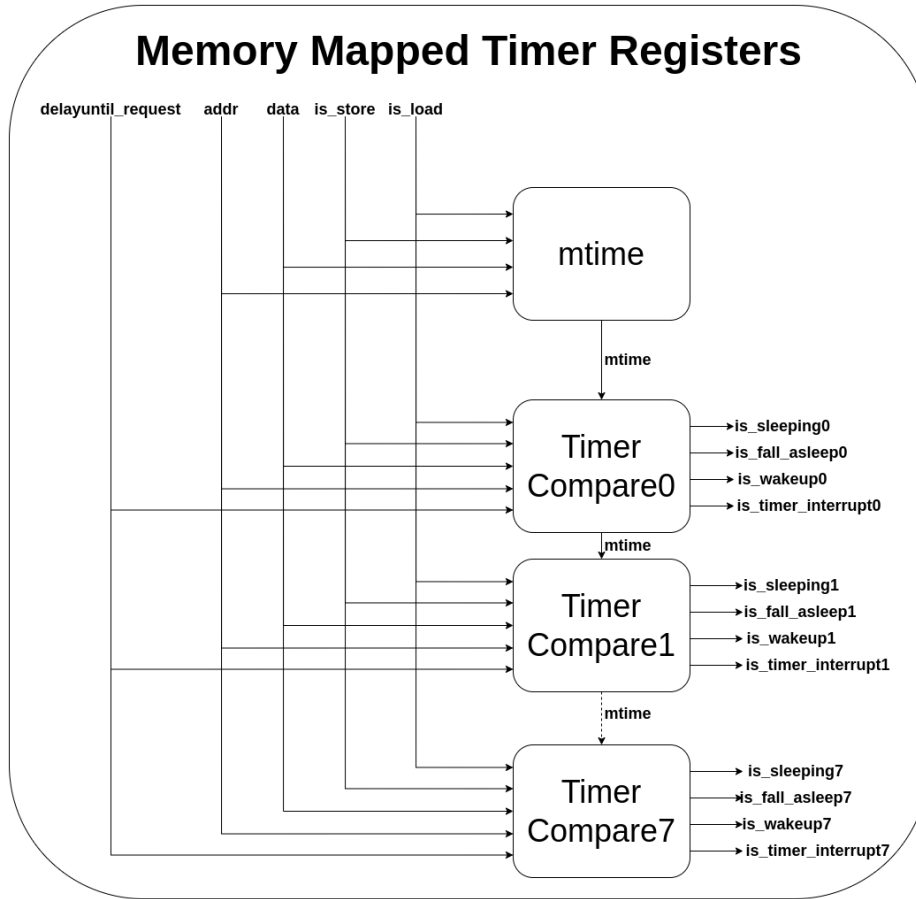


Figure 3.23: A simplified model of the memory-mapped timer registers. The output of `mtime` is delivered to each of the Timer Compare units. These units then check to see if `mtime` is greater than or equal to the value in the `mtimecmp` register. This comparison produces the four output signals on each timer compare unit.

instructions by default contain the offset value 804 to directly point the first register of the timer unit, namely `mtime`. However, the `interrupt_on_expire` and `delay_until` instructions must hold other address values to access the `mtimecmp` registers. Thus, a calculation using the thread ID is performed while performing either instruction to retrieve the correct address. The memory controller is the unit responsible for this, mentioned in Section 3.1.3. That is, the memory controller performs the calculation shown in Equation 3.1 to retrieve the correct memory address for the `mtimecmp` register of a specific thread.

$$addr = base + 4 * (1 + threadID) \quad (3.1)$$

The Timer Compare units shown in Figure 3.23 consists three blocks; the compare block, and the `delay_until` and timer interrupt units. The compare block, shown in Figure 3.24 is responsible for performing a comparison between `mtimecmp` and `mtime`. The block utilizes a technique used in existing hardware timers, shown in Equation 3.2, where an unsigned subtraction of the values is performed. The result from this subtraction is then considered as a signed value [20].

$$signed(unsigned(mtime) - unsigned(mtimecmp)) \geq 0 \quad (3.2)$$

This comparison operation is used to avoid the expiration problem that occurs using the comparison operation, shown in Equation 3.3, due to the possibility of overflows.

$$mtime \geq mtimecmp \quad (3.3)$$

Once the subtraction is performed, the value is checked to see if it is greater than zero. It is enough only to check the MSB of the subtraction result since the value is signed. Thus, when the MSB is a '1' it indicates a negative number, meaning that the value in `mtimecmp` is greater. On the other hand, when the MSB becomes '0', the comparison should set the `is_elapsed` signal high. As a result, the ' $\geq$ ' comparison is the inverse of the MSB of the subtraction result.

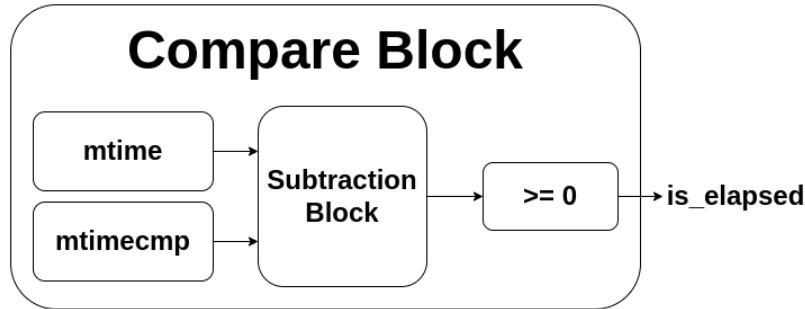


Figure 3.24: The compare block performs a comparison technique used to avoid the possibility of overflows. A subtraction of the `mtime` and `mtimecmp` registers is performed, where the `is_elapsed` signal is set high if the result is negative.

The `delay_until` unit designed for the Timer Compare unit can be seen in Figure 3.25. This unit determines whether the thread should fall asleep, wake up, or if the thread is currently sleeping. The `delayuntil_request` is the signal previously mentioned that is set high whenever the `delay_until` instruction is being executed in the memory stage of the pipeline. When this signal is set high, the `delayuntil` logic will decide whether the `delayuntil` register will be set high, indicating that the thread should fall asleep. Once the `delayuntil` register is set high, the `is_sleeping` signal will be set to '1' iff the `is_elapsed` signal coming from the compare block is '0', indicating that the sleep duration has not passed.

Once the `is_sleeping` signal is set high, the rising edge logic will notice that the signal is going from a '0' to '1', indicating that the `delayuntil` unit wants to put the thread to sleep. The `is_fall_asleep` signal will then be set high, telling the `mthreadmode` CSR to set the LSB of the thread ID's mode to '1'. The hardware thread scheduler will then update the schedule to take into account that a thread is now sleeping. While the thread is still sleeping, it is possible to wake the thread using external interrupts. For example, if an I/O interrupt occurs, the `mpie` CSR will provide an interrupt pending signal to the `delayuntil` logic, telling the `delayuntil` logic to clear the `delayuntil` register. The `is_sleeping` signal will then be cleared, resulting in a falling edge being detected by the falling edge logic. Thus, the `is_wakeup` signal will be set to '1', indicating to the `mthreadmode` CSR that the LSB of the thread ID's mode should now be cleared.

Once the interrupt handling is finished, the thread will perform the same `delay_until` instruction, causing the thread to fall asleep again if the duration has not elapsed. Finally, the `is_elapsed` signal from the compare block will be set high, thus clearing the `delayuntil` register and causing another wake-up procedure.

In comparison to the `delay_until` unit, the timer interrupt unit, shown in Figure 3.26, consists of a single 'AND' gate. This 'AND' gate makes sure that the timer interrupt can only

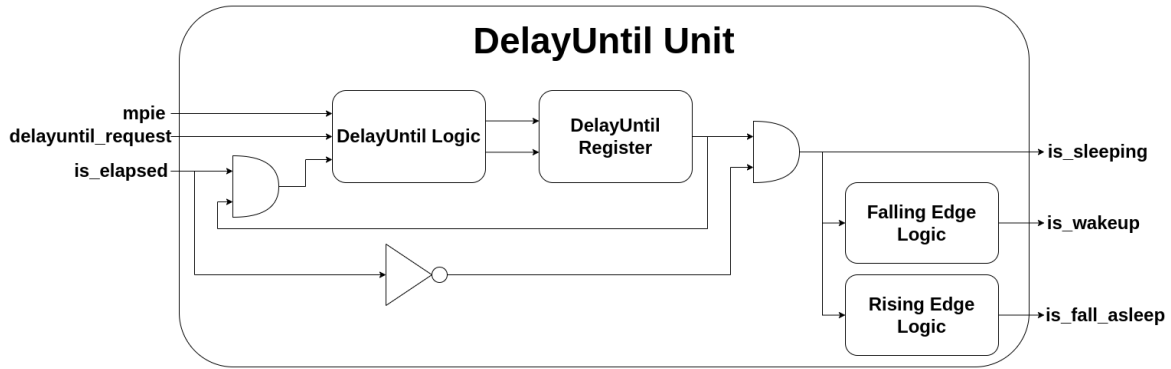


Figure 3.25: The delay until unit is responsible for selecting whether a thread should be sleeping, and for how long. The `delay_until` instruction writes the DelayUntil Register, where the `is_sleeping` signal is set high if the `is_elapsed` signal is low. To write the Hardware Thread Scheduler CSRs, the `is_fall_asleep` signals and `is_wakeup` are used to locate the start and end of the `delay_until` operation.

occur when the MTIE bit in the MIE register is set. By executing the `interrupt_on_expire` instruction, the MTIE bit is set and the `mtimecmp` register is written with the content of `rs2`, thus initializing the timer interrupt.

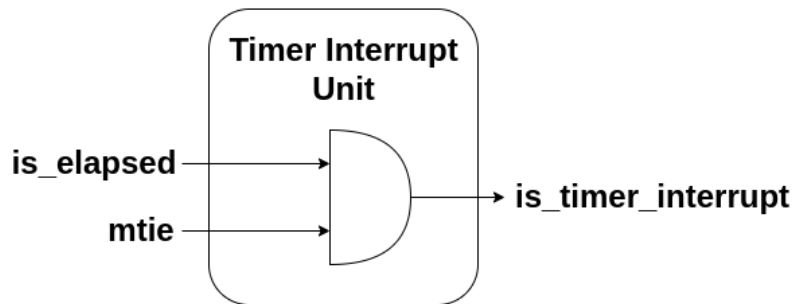


Figure 3.26: The timer interrupt unit is responsible for deciding whether the `is_elapsed` signal is valid as a timer interrupt signal by verifying the machine timer interrupt enable (MTIE) bit.

### 3.4 Assembler for the Microarchitecture

A simple assembler, shown in Appendix A.1, was designed to encode RISC-V and custom assembly instructions to 32-bit binary format to test the microarchitecture. That is, the assembler converts the assembly code to machine code in order for the microarchitecture to understand the instructions. The current assembler is a new design inspired by the assembler created in the previous semester for the single instruction RISC-V microarchitecture [66]. The assembler was designed in parallel with the microarchitecture, where each instruction encoding was added to the assembler once the microarchitecture could execute the instruction. Thus, the assembler was a means to provide a simple method to test each instruction that was added to the microarchitecture. By continuously adding more instruction encodings to the assembler while extending the microarchitecture, a total of 42 assembly instructions were

added. As a result, entire programs could be written in assembly code where the assembler could produce a text file that was easy for the Simulink model to interpret.

As the microarchitecture extended into the multithreading domain, it was necessary to have different threads executing functions from different areas in the instruction memory. Using the assembler with only instruction encoding functionality would be difficult for this purpose, since the assembler could not understand labels or comments. Thus, the programmer had to manually input the branch PC value, where the assembly code would look like Listing 3.3.

Listing 3.3: assemblerbefore.a

---

```
1 beq x1, x0, 60
```

---

Manually calculating the branch PC becomes troublesome if any other instructions are added to the program code. For example, if thread1 is executing a function that needs to be modified, then the branch PC required for thread2 to jump to the starting PC of a task must be recalculated. Otherwise, thread2 will use a branch PC leading to a different instruction, since the task's location is moved in the instruction memory.

Consequently, the possibility of using labels and comments was added to the assembler, shown in Listing 3.4. Furthermore, the assembler also uses the labels to calculate the distance from the branch or jump operations to the label's starting address. Thus, the assembler can automatically calculate the branch and jump PC values instead of being manually calculated by the programmer.

Listing 3.4: assemblerafter.a

---

```
1 beq x1, x0, thread0_function
```

---

Using a simple assembler to convert low-level assembly code to machine code can be an issue when working with more complex functionality. Thus, when testing more extensive, complex code, it would be better to use a compiler such as GCC or Clang and write in a higher-level language, such as C or C++. However, Simulink does not have a method for reading the compiled binary file into the instruction memory. That is, the current instruction memory cannot read a binary file that the microarchitecture can fetch instructions from and understand. A possible solution is to add a Matlab or Python script that converts the binary file format to a format that the microarchitecture can understand. However, this is not part of this thesis and will be part of future work.



## Chapter 4

# System Analysis

This chapter uses several methods to analyze the microarchitecture in various areas. In Section 4.1, the microarchitecture is configured in three modes: single-threaded, multithreaded with only HRTTs, and multithreaded with only SRTTs. Each of these configurations is simulated with various tasks to compare the execution times and the response time to I/O signals. Finally, in Section 4.2, the microarchitecture is theoretically compared against a coarse-grained multithreaded architecture.

### 4.1 System Configurations

#### 4.1.1 System Execution Time

The execution time of a system is an important attribute that specifies how quickly it can complete a task or a set of tasks. In a safety-critical system, poor execution time means requiring looser deadline constraints on hard real-time tasks. However, in a system with multiple tasks of high criticality, the insufficient execution time can become a problem; each task must have looser timing bounds to avoid failure. Consequently, the safety-critical system only applies to areas that do not require fast response and execution times.

For example, a weather drone uses a safety-critical system to retrieve enough weather data to transfer, which takes 4 seconds. Unless the drone aborts the data gathering during this period, it cannot perform other tasks such as flight controls. Thus, if the drone performs each task sequentially, it results in a 4-second lag in the flight controls task. A solution would be to let the higher criticality task interrupt the lower criticality task. However, continuously aborting the data collection for the higher criticality task results in the data collection task being redundant.

In comparison, if the data collection task takes 0.1 seconds, the drone is more than capable of collecting data and, at times, aborts the collection to control the drone. Hence, a faster execution time will result in a system that can be applied to areas requiring speedier computation. Additionally, it enables a more rapid response to external events handled by high criticality tasks, such as the flight controls.

A single-threaded system is good at executing a few tasks quickly. However, when branching, the system flushes all prior stages to the execute stage before fetching from a new location. Thus, the branch instruction takes three cycles to complete instead of one. The stalling caused by the branching is a very common stall that occurs to protect the system from data hazards. A different stall that can happen is a cache miss, in which the pipeline will stall for many cycles. Having such stalls injected into the pipeline significantly degrades the single-threaded system's execution time.

A test program was written in assembly code with eight tasks of differing computation times to analyze the microarchitecture as a single-threaded configuration, shown in Appendix A.2. Table 4.1 and Table 4.2 show the single-threaded configuration of the hardware thread scheduler. To schedule a single thread, either all slots but one can be disabled, or a single thread ID is scheduled, as seen in Table 4.1. Additionally, as a precaution, the other threads are kept in sleep mode in the mode configuration register. As a result, only thread0 will be able to use the pipeline to perform the eight tasks.

The startup procedure required to configure the microarchitecture as a single-threaded system can be seen in Listing 4.1. Here, only the mthreadmode CSR must be set to a new value since the mthreadslot CSR is set to 0, meaning that all slots are already set to thread0. As a result, only two lines of code are required, resulting in a small amount of initialization overhead before being able to perform the tasks.

Slot7	Slot6	Slot5	Slot4	Slot3	Slot2	Slot1	Slot0
Thread0	Thread0	Thread0	Thread0	Thread0	Thread0	Thread0	Thread0

Table 4.1: The slots configuration for the single-threaded system.

Thread7	Thread6	Thread5	Thread4	Thread3	Thread2	Thread1	Thread0
SZ	SZ	SZ	SZ	SZ	SZ	SZ	HA

Table 4.2: The mode configuration for the single-threaded system.

Listing 4.1: singlethreadedconfig.a

---

```

1 addi x1, x0, -4
2 csrrw x0, 1280, x1 // Set mthreadmode (SZ, SZ, SZ, SZ, SZ, SZ, SZ, HA)

```

---

Once the system has been initialized as a single-threaded system, it can begin running the eight tasks. Table 4.3 shows the tasks' deadlines, periods, and execution times used in the example from Appendix A.2. Furthermore, Figure 4.1 shows a graphical representation of the execution times of each task executed in a sequential. This graph results from the simulation result of the tasks running on the microarchitecture designed in Simulink. An example task performed by the microarchitecture can be seen in Listing 4.2. Here, the task computes the first 15 numbers of the Fibonacci sequence and stores the 15th Fibonacci number in the register file. Here, to calculate the 15 numbers, the task must branch 15 times. This results in 45 cycles spent on branching alone due to the single-threaded configuration flushing the pipeline each time a branch is taken.

Listing 4.2: fibonacci.a

---

```

1 thread6_fibonacci:
2     addi x14, x0, 15
3     addi x15, x0, 0
4     addi x16, x0, 0           // Show
5     addi x17, x0, 0           // a
6     addi x18, x0, 1           // b
7     add x16, x17, x18         // show = a + b
8     addi x17, x18, 0           // a = b
9     addi x18, x16, 0           // b = show
10    addi x15, x15, 1
11    blt x15, x14, -16

```

---

As mentioned above, if the execution times of the tasks are too slow, the system will not be able to complete the tasks within the specified deadline. In this example, because the tasks are executed sequentially, each task is dependent on the execution time of other tasks. Consequently, the deadline specified in the example will not be able to meet all deadlines. If all tasks have high criticality levels, multiple failures will be present in the system, specifically tasks 7 and 8 in the example shown in Figure 4.1.

Additionally, these tasks are executed periodically in a round robin fashion, with a period identical to the deadline. Thus, an increasing number of tasks will produce a failure each periodic interval. This means that in the next periodic interval from the example above, tasks 4 through 8 will fail due to the delayed start of Task1 at time 538. As a result, either tasks 7 and 8 must be discarded, or the deadline and period must be extended to beyond the execution times of the tasks.

Task	Thread ID	Thread Mode	Execution Time	Period / Deadline
$T_1$	0	HA	70	400
$T_2$	0	HA	83	400
$T_3$	0	HA	95	400
$T_4$	0	HA	27	400
$T_5$	0	HA	47	400
$T_6$	0	HA	50	400
$T_7$	0	HA	107	400
$T_8$	0	HA	51	400

Table 4.3: Real-time constraints of the tasks.

To configure the microarchitecture as a fine-grained multithreaded system, a setup procedure, shown in Listing 4.3 must be performed. This procedure is only performed by thread0, where the other threads will not be able to access the pipeline until after the mthreads slot CSR is written. Lines 2 and 3 are added to make sure that the other threads do not execute the setup procedure, whereas they will jump directly to the main function. Once the setup procedure has been performed, the resulting slots CSR is shown in Table 4.4 and the modes CSR in Table 4.5. As can be seen in the mode CSR configuration, all threads are configured as SRTTs. Thus, when a thread is finished with its task, it will sleep and let the other threads use the spare cycles. As a result, the configuration will provide dynamic thread scheduling frequency where the scheduling frequency increases for the active threads whenever a thread falls asleep. Consequently, the system configura-

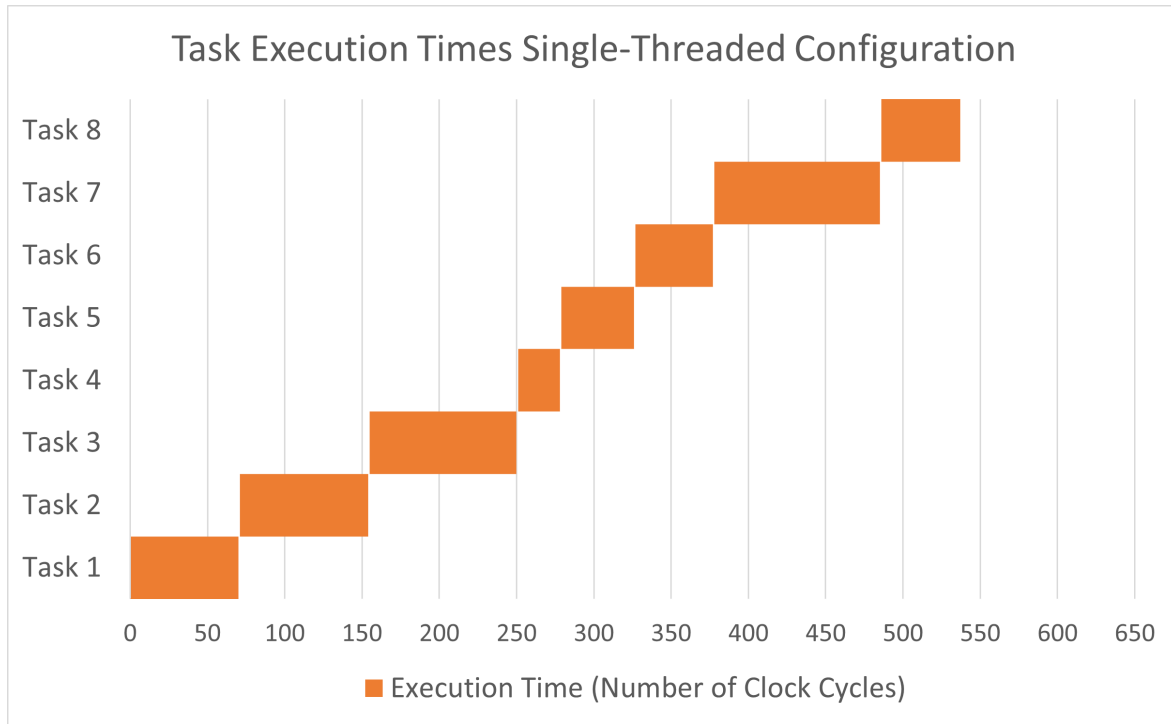


Figure 4.1: The execution times required to finish eight different tasks using the single-threaded configuration. Each of these tasks are executed sequentially. Adding concurrency to this configuration would require additional overhead due to context switching.

tion has traded its isolation and predictability for an increased instruction throughput.

Listing 4.3: srttconfig.a

```

1 startup:
2     csrrsi x1, 3860, 0           //set x1 = mhartid
3     bne x1, x0, main           //if x1 != 0, Jump to main
4     lui x2, 1048565            //x2 = 1111111111111110101
5     xori x2, x2, 2730          //x2 = Sext(1010 1010 1010 1010)
6     csrrw x0, 1280, x2         //Set mthreadmode
7                               //(SA, SA, SA, SA, SA, SA, SA, SA)
8     lui x31, 484675            //Set Upper bits of mthreadslot
9                               //(T7, T6, T5, T4, T3, 0, 0, 0)
10    addi x31, x31, 528          //Set Lower bits of mthreadslot
11                               //(0, 0, 0, 0, 0, T2, T1, T0)
12    csrrw x0, 1281, x31         // Set mthreadslot
13                               //(T7, T6, T5, T4, T3, T2, T1, T0)
14    jal x30, main              //(T0) jump to main

```

When the setup procedure is finished, the program jumps to the main function, shown in

Slot7	Slot6	Slot5	Slot4	Slot3	Slot2	Slot1	Slot0
Thread7	Thread6	Thread5	Thread4	Thread3	Thread2	Thread1	Thread0

Table 4.4: The slots configuration for the multi-threaded system using only soft real-time threads (SRTT).

Thread7	Thread6	Thread5	Thread4	Thread3	Thread2	Thread1	Thread0
SA	SA	SA	SA	SA	SA	SA	SA

Table 4.5: The mode configuration for the multi-threaded system using only soft real-time threads (SRTT).

Listing 4.4. Here, each thread will jump to its designated task if there is a thread ID match. This step allows the threads to be running in different locations in the instruction memory. Because of the increase in the number of instructions performed before the task execution, the startup overhead has significantly increased compared to the single-threaded configuration. Even though the overhead is close to a quarter of the total simulation time for the entire test program, it is disregarded in the analysis because the overhead is static. Meanwhile, the execution time of any task may increase or decrease depending on the amount of computation required. For example, if the Fibonacci function in Listing 4.2 computes 60 values instead of 15, the execution time of the task would be approximately quadrupled. As a result, the overhead becomes negligible with an increase in the execution times of the tasks.

Listing 4.4: srttmain.a

---

```

1  main:
2      beq x1, x0, thread0_incrementer    //if mhardid == 0,
3                                          //Branch to thread0_incrementer
4      addi x2, x0, 1
5      beq x1, x2, thread1_multiplication //if mhardid == 1,
6                                          //Branch to thread1_multiplication
7      addi x2, x2, 1
8      beq x1, x2, thread2_division       //if mhardid == 2,
9                                          //Branch to thread2_division
10     addi x2, x2, 1
11     beq x1, x2, thread3_read_time       //if mhardid == 3,
12                                          //Branch to thread3_read_time
13     addi x2, x2, 1
14     beq x1, x2, thread4_shift_left      //if mhardid == 4,
15                                          //Branch to thread4_shift_left
16     addi x2, x2, 1
17     beq x1, x2, thread5_shift_right     //if mhardid == 5,
18                                          //Branch to thread5_shift_right
19     addi x2, x2, 1
20     beq x1, x2, thread6_fibonacci       //if mhardid == 6,
21                                          //Branch to thread6_fibonacci
22     addi x2, x2, 1
23     beq x1, x2, thread7_even            //if mhardid == 7,
24                                          //Branch to thread7_even
25     beq x0, x0, 0                       //NOP

```

---

Once the system is configured and the threads have branched to their respective tasks, the system will perform the same tasks as in the single-threaded configuration, shown in Appendix A.3. The resulting execution times of the tasks can be seen in Figure 4.2. A major difference from the single-threaded execution is that it performs all tasks concurrently instead of sequentially, with a clock cycle granularity. As can be seen, this leads to Task4 finishing first because it is the least computationally heavy task. Once this task finishes, the slot allocated to thread3 will be used by the other threads in a round robin fashion. As a result, there is an increase in performance for the other tasks that are still executing in the pipeline.

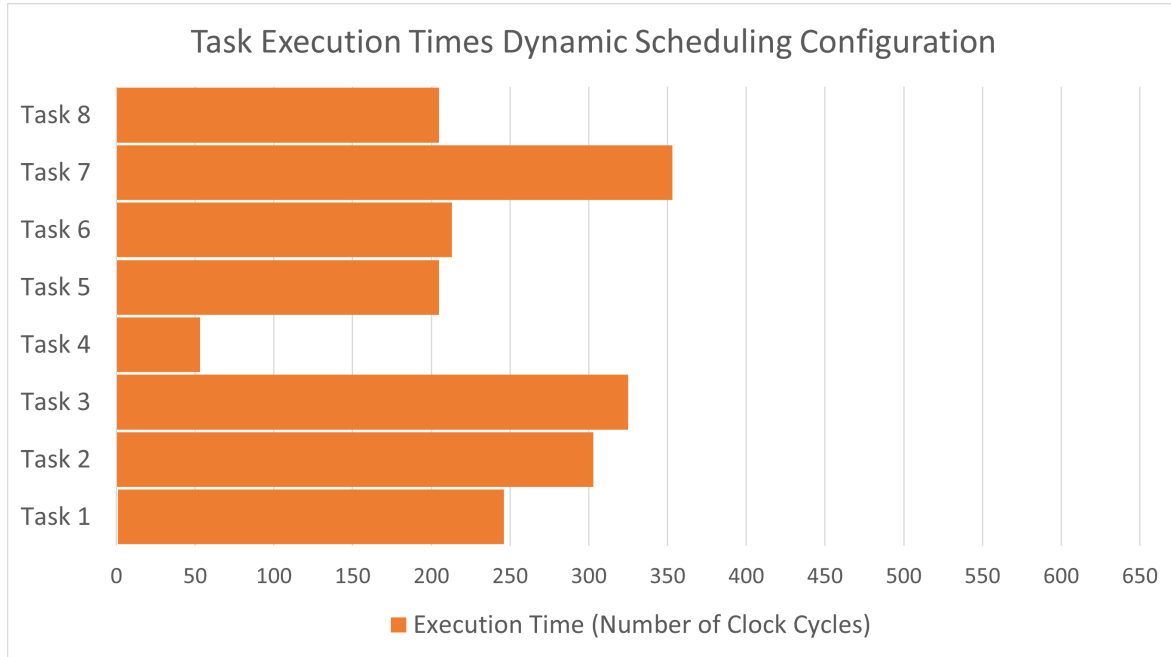


Figure 4.2: The execution times required to finish eight different tasks using the multi-threaded SRTT configuration. These tasks are executed concurrently, thus starting their task execution clock cycles apart.

As mentioned previously, an important benefit of the fine-grained multi-threaded configuration is its capability to avoid data hazard stalls by cycle-by-cycle thread interleaving. As a result, close to all the branch stalls present in the single-threaded configuration are avoided, which significantly reduces the execution time of the system. The reason for the phrasing "close to all" is because as more threads are sleeping, the more the schedule approaches a single-threaded schedule. Thus, when task3 in Figure 4.2 falls asleep, only task7 is scheduled. By avoiding the branch stalls, the total task execution time has been almost halved for the fine-grained multi-threaded system in comparison to the single-threaded system. Thus, because of the significantly reduced execution time, all tasks are now capable of meeting the deadlines specified in Table 4.3.

Although the fine-grained multi-threaded system tested above is capable of executing the tasks significantly faster than the single-threaded system, it can be much slower using other thread configurations. As an example, the mode CSR shown in Table 4.6 depicts such a configuration, where all threads are HRTTs. Consider the same slots CSR configuration as used in the SRTT example. Because all threads are configured as HRTTs, the threads are only capable of using the cycles that are allocated to that particular thread in the slots CSR. That is, each thread has a static thread scheduling frequency of  $1/8$ . Consequently, when a thread finishes its task, the slot will be left empty, and the hardware thread scheduler will instead stall the pipeline that cycle.

Thread7	Thread6	Thread5	Thread4	Thread3	Thread2	Thread1	Thread0
HA	HA	HA	HA	HA	HA	HA	HA

Table 4.6: The mode configuration for the multi-threaded system using only hard real-time threads (HRTT).

When configuring the threads as HRTTs, the CSRs can be to zero since an active HRTT thread is denoted by '00', resulting in the CSR operation shown in Listing 4.5. Once it has been configured with only HRTTs, the same code can be run as in the previous example. As a result of running the simulation with this configuration, the tasks will have the execution times seen in Figure 4.3.

Notice how Task 4 for both configurations have the same execution times while execution times for the other tasks deviate more depending on how many threads are sleeping. As mentioned above, the reason for the significant increase in execution time is because of the fixed schedule. As a result, a stall is introduced to the pipeline each cycle a sleeping thread is scheduled. Consequently, once Task 3 finishes its execution and puts thread2 to sleep, only 1 out of 8 threads are active, meaning that 7 out of 8 cycles are wasted. However, because the scheduling frequency is fixed, the isolation and predictability has been maximized. That is, the timing behavior of the tasks are known and bounded since the timing of each instruction is known and the threads are temporally and spatially isolated. As a result, the configuration has traded throughput for high predictability and isolation.

Listing 4.5: hrttconfig.a

---

```
1 csrrw x0, 1280, x0 //Set mthreadmode(00 00 00 00 00 00 00 00)
```

---

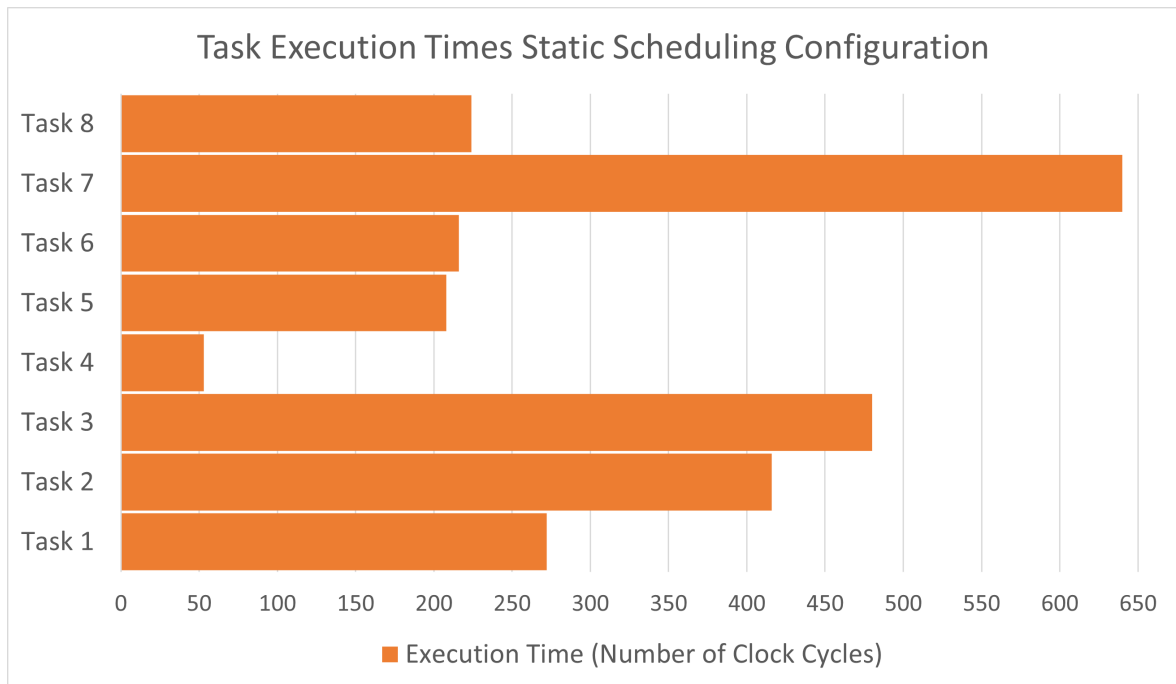


Figure 4.3: The execution times required to finish eight different tasks using the multi-threaded HRTT configuration. These tasks are isolated and have a fixed thread-scheduling frequency. Thus, the tasks will execute the same even when other tasks finishes their execution. As a result, because there is no interference between the tasks, it is less tedious to perform WCET analysis.

A comparison of the task specific execution times for all three configurations can be seen in Figure 4.4. Notice how there is a large gap in the execution times for each of the configurations. As mentioned earlier, the single-threaded configuration is faster at performing

a single task compared to the multi-threaded configurations. Even task 4, the least compute intensive task, has half the execution time in the single-threaded configuration compared to the multi-threaded configurations. Also, notice how the multi-threaded HRTT configuration deviates more when there are fewer threads left active. As a result, the HRTT configuration has almost 6 times as long execution time on Task7 compared to the single-threaded configuration, while the SRTT configuration is only 3 times as long.

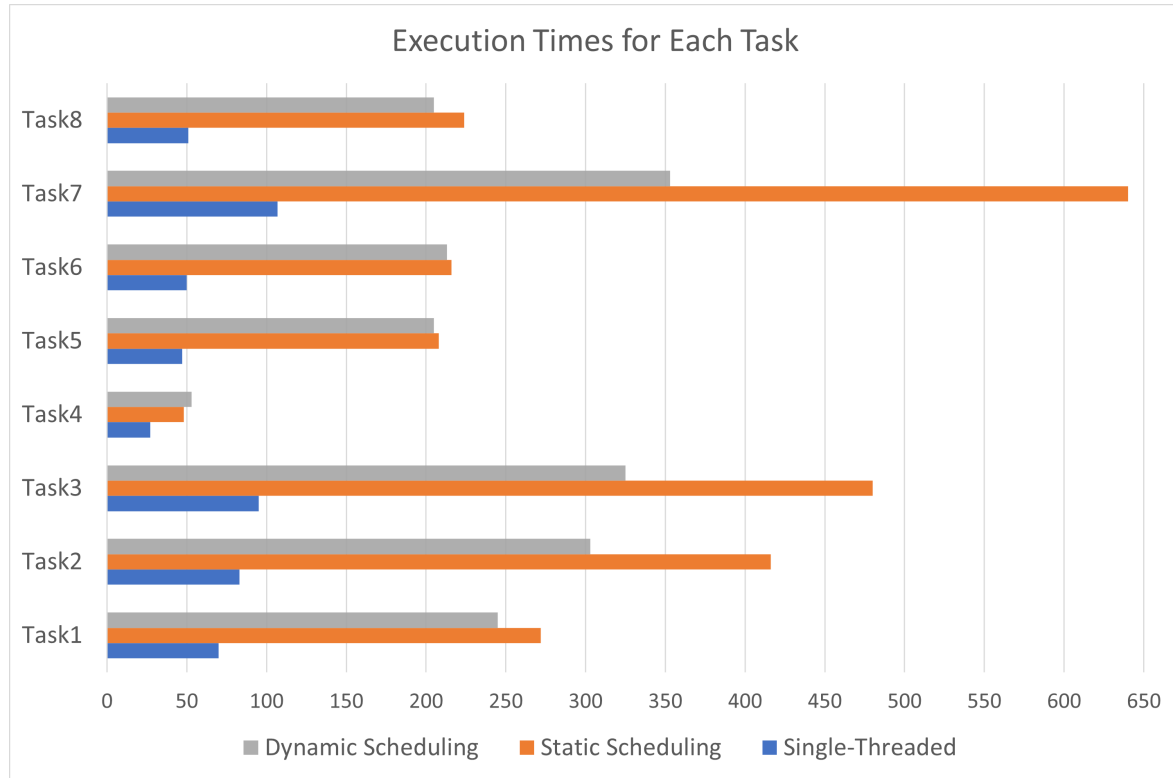


Figure 4.4: The execution times for eight tasks when running on three different system configurations.

In comparison, Figure 4.5 shows how each configuration performed in terms of the total execution time required to complete the same set of tasks. Here, even though the single-threaded configuration outperformed both multi-threaded configurations in terms of specific task execution times, it is not the case for the total execution time. Instead, the single-threaded configuration only managed to outperform the HRTT configuration while the SRTT configuration was faster than both. As mentioned previously, the reason for the faster total execution time for the SRTT configuration is that it avoids most stalls in the pipeline. If there would have been no stalls in the tasks being tested, the SRTT and single-threaded configurations would have identical overall execution times. However, due to the branch stalls, the SRTT configuration managed to reduce the stall latency by almost 200 cycles.

#### 4.1.2 System Response

Certain real-time cyber-physical systems may require fast reaction and response times to external stimuli. If these system behaviors are slow, it may cause the system to be unable to respond quickly enough, resulting in deadline requirements not being met. For example, a strong gust hits the weather drone, causing it become unstable and spin out of control. As a



Figure 4.5: The total execution time each system configuration require to complete all eight tasks.

consequence, the drone will crash if it is not stabilized in time. To counteract this, the drone is equipped with sensors to keep the drone stabilized.

A simple representation of the reaction and response time of the task can be seen in Figure 4.6. The time it takes the drone to retrieve sensor data, compute a counteracting thrust, and output the thrust to the motors, is the reaction time of the drone's stabilization task. However, this time does not consider the duration required to stabilize the drone. The time it takes the drone to react to the problem and stabilize the drone is the response time of the task. Thus, if the reaction time is too slow, the drone will not have enough time available to stabilize. Also, if the reaction time is fast, but the stabilization time is slow, the drone will not reach the stable region before it is too late.

In contrast, if the reaction and response times are fast, the drone will recover from the unstable region. As a result, it is important for the stabilization task to be able to quickly respond to changes in the sensor data to counteract potential instability. Similarly, it is important in other cyber-physical systems with real-time tasks to react swiftly to changes in sensor data.

As was mentioned in Section 4.1.1, the single-threaded configuration is capable of executing singular tasks quickly. Thus, because the execution time of a task is a large portion of the response time, it should be able to respond quickly to changes in sensor input. However, if the drone's software contains multiple tasks of high criticality that must be scheduled periodically, it will result in each task having slower response and reaction times.

A potential solution to this could be to incorporate preemptive task scheduling, where the critical tasks can interrupt other less critical tasks at specific times [4]. However, the timing behavior of the software will become unpredictable, as the stabilization task can preempt other tasks at random moments depending on the behavior of the physical world. This leads

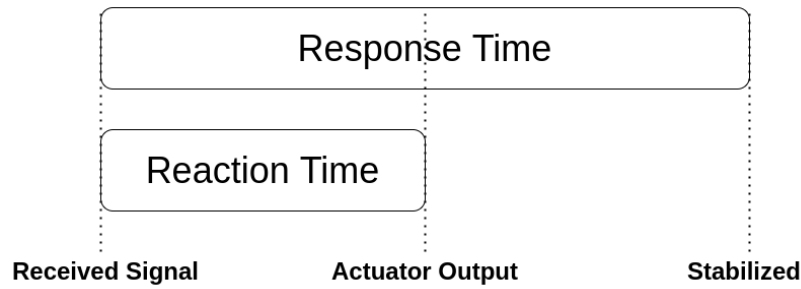


Figure 4.6: A simple model that shows the response time and reaction time of an I/O task. When the sensor notices a signal change, the task should begin execution. The reaction time is the time it takes the task to output a value in response to the sensor data. The response time is the time it takes to finish the response to the sensor data, which in this case is when drone is stabilized.

to preemptive scheduling not being a viable solution for the drone. Additionally, if there are multiple such high criticality tasks that depends on the physical world, it could lead to other important software tasks being ignored completely. Finding a proper scheduling method that takes this issue into consideration can be a tedious task, and will most likely restrict the drone’s software to less optimal timing constraints.

Instead, adding fine-grained multithreading to the drone’s processor could be a potential solution, where multiple tasks are executed concurrently, as seen in Section 4.1.1. Although individual tasks will have a lower instruction throughput, it is possible to place the task interacting with the physical world onto separate threads, allowing the software to have more optimal timing constraints.

To show the difference in response time between the fine-grained multithreaded and single-threaded configurations, an example program with an I/O polling task will be analysed in detail. As mentioned previously, the execution time is a large portion of the response time of a task. Because an execution time analysis was performed in the previous section, where each configuration was compared in terms of its computation ability, it is not necessary to add additional complexity to the response time task. That is, a task, seen in Listing 4.6, will poll the MMIO’s input register until the value is nonzero. Once the value in register x4 is nonzero, the task will store the value of x4 in the MMIO’s output register. Thus, the time it takes from the input register is updated, until the value is present on the output register is referred to as the response time of the task.

By running the program in a Simulink simulation, the best and worst-case response times of the single-threaded configuration were found, shown in Figure 4.7. The best-case response time is when the task executes the load instruction right after the MMIO’s input register has been modified. In contrast, the worst-case response time is when the task executes the load instruction right before the MMIO’s input register has new content.

Listing 4.6: responsetimetask.a

---

```

1 poll_input:
2     lw x4, 0(x3)
3     beq x4, x0, poll_input
4 set_output:
5     sw x4, 4(x3)

```

---

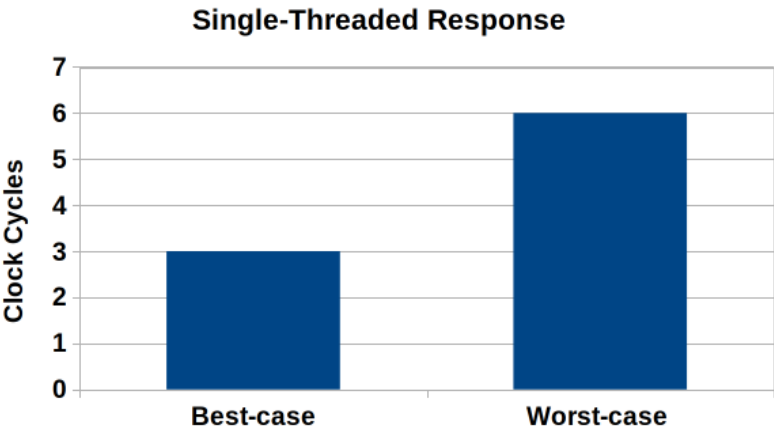


Figure 4.7: The best-case and worst-case response times using a single-threaded configuration.

To illustrate the best and worst-case response times of the single-threaded configuration, a diagram, shown in Figure 4.8, showing the details of the execution of each instruction was made. On the left is the worst-case behavior of the input, output, and x4 registers. The boxes colored red indicates when the worst-case notices the changes in value, and when it manages to output the same value in response. Here, the worst-case behavior receives an update on the input register during the 5th cycle, which is the cycle after reading that register. Consequently, the task is not aware yet of the value update and must branch to read the value again, resulting in two cycles lost to a pipeline flush. The following load instruction will then retrieve the new register value, thus exiting the polling loop. Finally, the task will write the same value to the output register, where the register will be updated with new content in the 11th cycle. As a result, the task spent 6 cycles to respond to an I/O signal. In constrast, the best-case performance can be seen starting during the 8th cycle. Here, the register has just updated as the task starts reading the register. Finally, the best-case scenario writes the output register, where the new content is available in the 11th cycle.

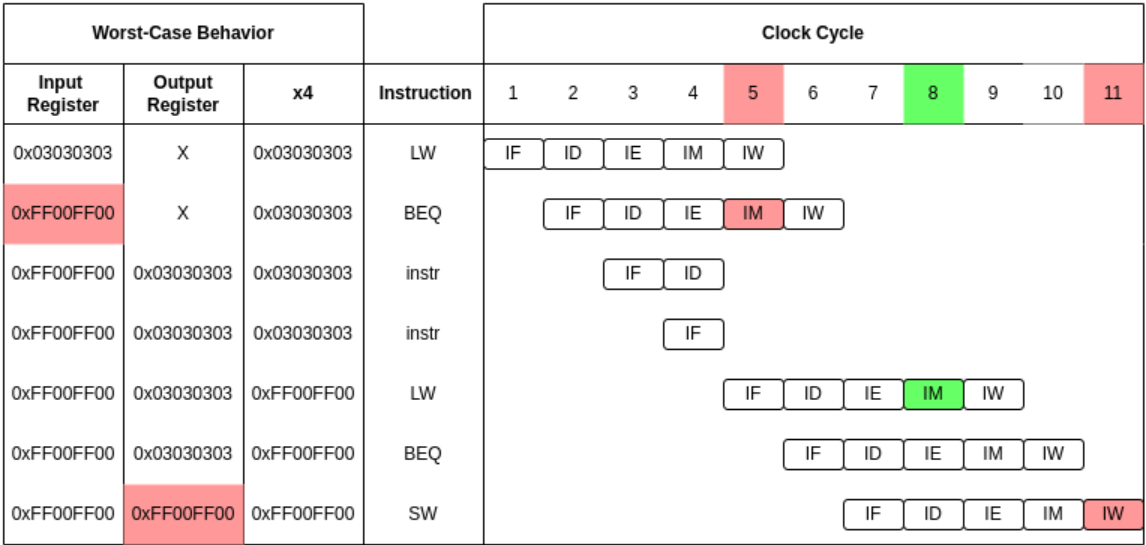


Figure 4.8: The best-case and worst-case response time of the assembly code in Listing 4.6.

The fine-grained multithreaded configuration used in this example is the same as the static scheduling configuration used in Section 4.1.1. As a result, all tasks will be isolated, meaning that the other tasks running on the system will not interfere with the analyzed task. Moreover, the response time analysis will execute the same tasks as in the execution time analysis. However, task 3 has been exchanged for the task seen in Listing 4.6. By simulating the program in Simulink, the best and worst-case response times of the multithreaded configuration with 8 HRTTs were found, shown in Figure 4.9. It can be seen that having 8 threads executing concurrently increases both the best and worst-case response times of the task.

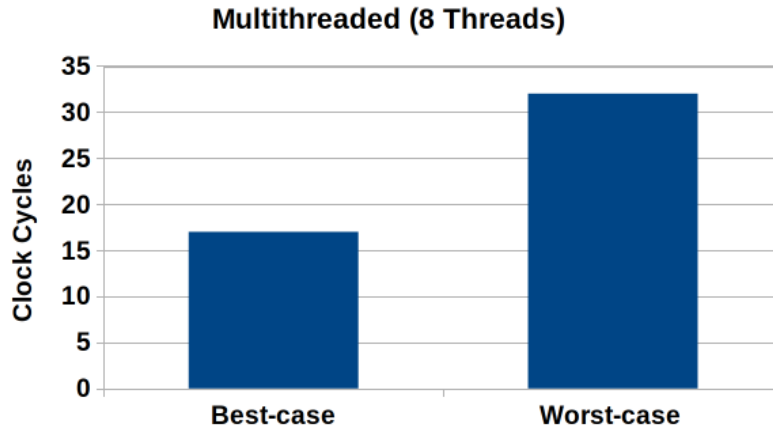


Figure 4.9: The best-case and worst-case response times using a multithreaded configuration with 8 threads.

By simple math, a relationship between the thread scheduling frequency and total latency difference can be established. Equation 4.1 shows the relationship between the worst-case response times of the single-threaded and multithreaded configuration. Similarly, Equation 4.2 shows the relationship of the best-case response times.

$$WCRT_M = \frac{WCRT_S - Latency}{f_{ts}} \quad (4.1)$$

$$BCRT_M = \frac{BCRT_S - Latency + f_{ts} - 1}{f_{ts}} \quad (4.2)$$

where:  $WCRT_M$  = Worst-Case Response Time Multithreaded Configuration  
 $WCRT_S$  = Worst-Case Response Time Single-Threaded Configuration  
 $BCRT_M$  = Best-Case Response Time Multithreaded Configuration  
 $BCRT_S$  = Best-Case Response Time Single-Threaded Configuration  
 $Latency$  = Total difference in instruction latency  
 $f_{ts}$  = Thread scheduling frequency

To verify that the best and worst-case equations are correct, two additional simulations were performed with different thread scheduling frequencies on the thread executing the response time task (Task3). The first test had a thread scheduling frequency of  $\frac{1}{2}$ , which resulted in a best and worst-case response time of 5 and 10 clock cycles, respectively. The second test had a thread scheduling frequency of  $\frac{1}{3}$ , resulting in a best and worst-case response time of 7 and 12, respectively. Using the equations above, it can be seen that the exact same values are achieved. Finally, to show the relationship graphically, the best and worst-case

values from the simulations are plotted, shown in Figure 4.10. These best and worst-case lines form the upper and lower bounds on the response time of the task. Here, it can be seen that the task's response time bounds increases almost linearly with a reduction in thread scheduling frequency. However, there is an exception at a thread scheduling frequency of  $\frac{1}{2}$ , because there is a one cycle branch latency present in the worst-case response time.

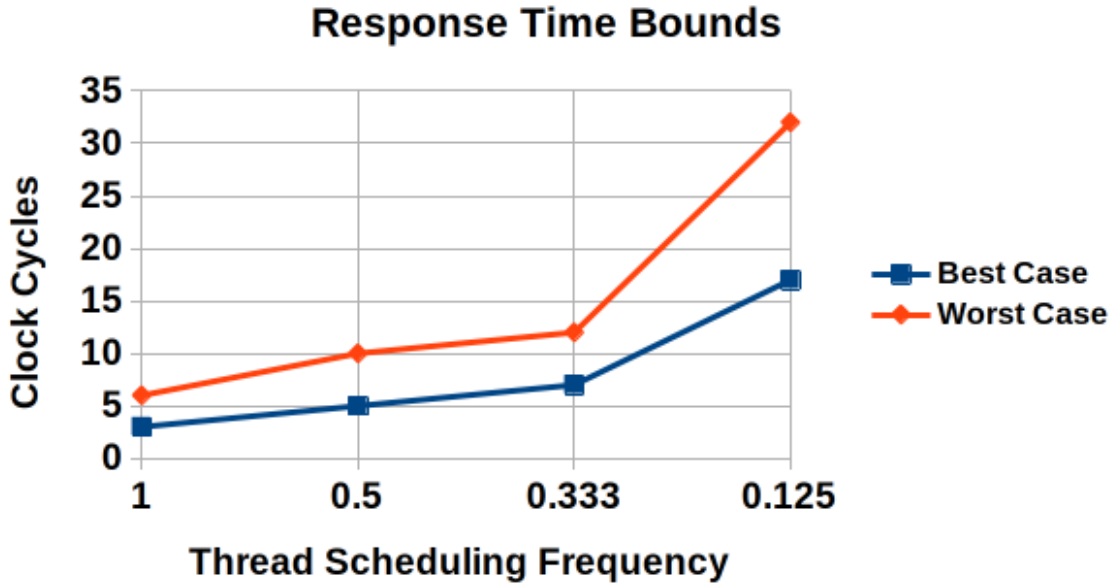


Figure 4.10: The best-case and worst-case response times of the microarchitecture due to the thread scheduling frequency. The graph is only valid when the number of threads used is the inverse of the thread scheduling frequency, or when there are enough thread cycles in-between each thread execution to avoid the branch latency. Thus, there must be at least 2 thread cycles between each time a thread is executed.

As a result of this analysis, it can be seen that there is a direct correlation between response times and the thread scheduling frequency. Here, it was shown that the single-threaded configuration outperformed the multithreaded configuration. However, due to the polling nature of this task, the single-threaded configuration is incapable of executing other tasks. A solution to perform more tasks would be to allocate time slices for each task, where a task can execute for a certain amount of time before letting other tasks use the pipeline. However, because of the overhead of switching tasks in addition to the execution times of other tasks, the response time of the single-threaded system can have a significantly worse response time than its multithreaded counterpart. Additionally, the fine-grained multithreaded configuration manages to hide branch latencies, thus reducing the response time further while also making the timing behavior more predictable. Finally, the multithreaded configuration is capable of executing multiple I/O tasks concurrently with fine granularity, which the single-threaded configuration is incapable of.

In conclusion, while adding threads to the system adds latency to the response time of a single task, it makes it possible to run multiple tasks of differing criticality levels concurrently. These tasks can be scheduled with a static scheduling frequency, thus providing high temporal isolation and predictability. As a result, because of the temporal isolation between the threads, multiple tasks can respond to data from the physical world without degrading the response time of the other tasks.

## 4.2 Coarse-grained Multithreading

A coarse-grained multithreaded system will for short stalls operate the same as a single-threaded system. That is, a branch operation will in a coarse-grained multithreaded system flush the pipeline and continue execution as normal. The reason that this technique does not change threads on a branch operation is because the cycles required to change threads are greater than the stall introduced by flushing the pipeline. Thus, there is no benefit by performing thread switching for a coarse-grained multithreaded system when the program wants to perform a branch.

However, for longer stalls such as cache misses, the coarse-grained multithreaded system will deviate in functionality compared to a single-threaded system. Whereas the single-threaded system would wait for the required data that caused the cache miss, the coarse-grained system will switch threads and continue execution of a different task. This switching will cause the program to lose as many cycles as there are pipeline stages in the system. That is, the pipeline must be flushed and new instructions from the new thread must fill the pipeline. For example, if the pipeline was 5 stages, the program execution will only be stalled for 5 cycles instead of the number of cycles required to fetch the data causing the cache miss, which could be as many as hundreds of cycles.

As mentioned in Section 2.3, the fine-grained multithreaded technique is capable of hiding both short and long stalls while the coarse-grained multithreaded technique is used to hide long stalls. The fine-grained multithreaded microarchitecture is capable of performing concurrent thread execution where the goal is to execute multiple tasks at the same time. It is further capable of configuring the threads as either hard or soft, making it possible to increase isolation between threads for mixed-criticality and safety-critical systems. Meanwhile, the coarse-grained multithreaded technique focuses on execution of a single task as quickly as possible. It is incapable of hiding short stalls, thus making the timing behavior of the real-time system less predictable. As a result, it can be said that the coarse-grained multithreaded technique focuses on task specific throughput and less on timing behavior and predictability. Meanwhile, the fine-grained multithreaded microarchitecture can exchange task specific throughput for better timing behavior and predictability, and vice versa.

As an example, consider both the fine-grained multithreaded microarchitecture and a coarse-grained multithreaded system having 5 hardware threads and a 5-stage pipeline. When a thread has to branch, the fine-grained system will be able to hide the latency by allowing other threads to use those cycles. Similarly, if there is a cache miss it allows other threads to use the spare cycles while the thread is waiting for the data it requires. Moreover, once the thread has managed to get the required data, the thread will return to normal execution. Meanwhile, the coarse-grained system is incapable of hiding the branch latencies, resulting in a two-cycle latency when a branch is taken. However, the coarse-grained system manages to hide most of the cache miss latency by performing a thread switch once. As a result, the coarse-grained system will only have a five-cycle latency on cache misses. Additionally, once the thread manages to retrieve the required data, it will wait until the thread scheduler allows it to switch back. This delay will depend on the type of thread scheduler implemented, but there is at least another five-cycle delay due to the thread switching. As a result, the fine-grained multithreaded system is capable of returning to normal execution faster after a long stall than the coarse-grained system.

Continuing the example, the systems are executing a program where both have a total number of 100 cache misses and 500 branches throughout the program execution. The fine-grained multithreaded design is capable of hiding the branch latencies by having other threads use those cycles. During a cache miss, the fine-grained system will allow other threads to

use the spare cycles. However, the system is incapable of hiding the cycle that resulted in a cache miss. Thus, for every cache miss there is a one-cycle latency added to the program execution, resulting in an overall latency of 100 clock cycles throughout program execution.

In contrast, the coarse-grained system only manages to reduce the cache miss latency to a five-cycle latency, while it is incapable of hiding the branch latency. Thus, the coarse-grained system will have an overall latency of 1500 clock cycles throughout the program execution due to cache misses and branches. As a result, the coarse-grained multithreaded system will have an overall execution time that is 1500 clock cycles slower than the fine-grained multithreaded system. Although, the coarse-grained multithreaded system will have the same throughput as a single-threaded system when executing a single task. Thus, it will have better task-specific throughput than the fine-grained multithreaded microarchitecture.

In addition, consider the designed microarchitecture configured as a single-threaded system executing the same program. Because it only consists of a single thread, it is incapable of hiding cache miss and branch latencies. Consider that a cache miss requires the microarchitecture to spend 200 clock cycles to retrieve the data from main memory. As a result, the single-threaded configuration will have a 21,000-cycle latency added to the program execution. Thus, the execution time for the single-threaded configuration will be significantly deteriorated due to the added latencies caused by the branches and cache misses.

The total latency of each system is shown in Figure 4.11. As can be seen, the coarse-grained multithreaded system is capable of hiding most of the cache latency that is present in the single-threaded configuration. However, it is not capable of hiding the branch latency, resulting in the same branch latency as the single-threaded configuration. Meanwhile, the fine-grained multithreaded system is capable of hiding the branch latency and most of the cache latency by switching threads. As a result, the fine-grained multithreaded system will have a 1400 clock cycles better overall execution time than the coarse-grained multithreaded system. Furthermore, it will have a 21000 clock cycles better overall execution time than the single-threaded configuration. It should be mentioned, however, that this is only the overall execution time, whereas the task specific execution time will be better for the other systems.

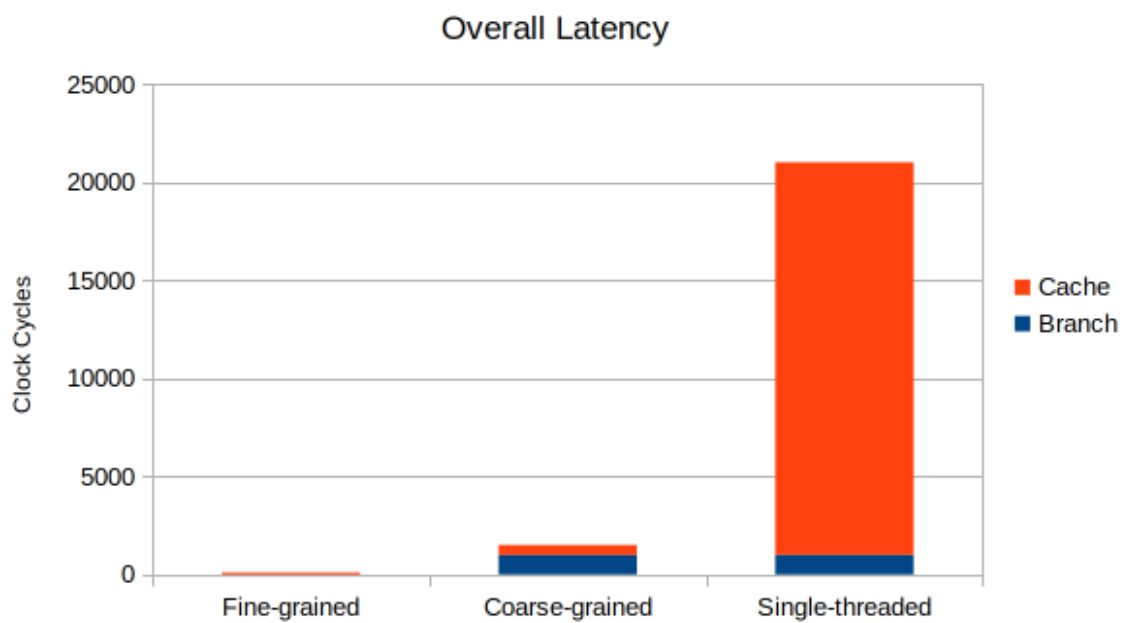


Figure 4.11: The overall latency of the different hardware threaded systems. Notice how the coarse-grained and single-threaded systems have the same branch latency, but most of the cache latency is removed. In comparison, the fine-grained system has almost no latency added.

## Chapter 5

# Conclusion and Future Work

In cyber-physical and real-time applications, it is necessary to have a high degree of confidence in the software functionality and the timing behavior. Notably, it is crucial for safety-critical applications. Integrating such applications of different criticality levels onto a shared hardware platform, known as a mixed-criticality system (MCS), is a trending topic in reducing cost, power consumption, size, and weight. However, this integration results in difficulties with verification and certification of the system due to spatial and temporal resource sharing causing interference between applications of different levels of criticality. Using real-time operating systems (RTOS) on general-purpose processors is a common approach to enable software-based isolation while drastically reducing hardware costs. However, RTOS adds significant overheads, such as task switch latency, preemption time, and inter-process communication (IPC), resulting in timing requirements in milliseconds. Additionally, general-purpose processors result in more unpredictable timing behavior due to, e.g., interrupts and hardware prediction mechanisms. Consequently, the MCS will be harder to verify and certify and can also have timing accuracy and precision that are unsatisfactory for specific input/output (I/O) applications. This report's central theme is on microarchitectures providing confidence in software functionality and timing behavior through hardware-based isolation while maintaining overall instruction throughput and precision-timed I/O with accuracy and precision at clock cycle granularity.

The microarchitecture designed and presented in this report can compromise by exchanging predictability for efficiency, and vice versa, through configurable hardware-based isolation between applications. In addition, the design utilizes a hardware thread scheduler capable of fine-grained multithreading with software customizable threads and schedule. The scheduler can execute two thread types; hard real-time threads (HRTT) and soft real-time threads (SRTT). Furthermore, the scheduler provides static and dynamic thread scheduling frequency capability by restricting HRTTs to predetermined cycles while allowing SRTTs to use predetermined and unused cycles. In addition, the microarchitecture contains custom timing instructions and timing units contributing to the timing control cases identified by Bui et al. [27], resulting in fine-precision timing behavior while producing spare cycles when threads are inactive.

The microarchitecture was designed in Simulink, a tool enabling visual prototyping and continuous testing and validation of system characteristics, thus reducing the number of development risks. In addition, the tool enables the development of a simple test environment, where software programs were simulated on the microarchitecture by reading instructions from a text file. Also, using the simulation time, it was possible to retrieve the test's instruction results with cycle-accurate timing. As a result, it was possible to perform various analyses with Simulink to verify the functional correctness and timing behavior at both

instruction and task granularity.

The microarchitecture was analyzed for two application areas; mixed-criticality systems and precision-timed I/O operations. Here, response time and execution time analyses of several thread scheduling configurations were performed. As a result, the analyses show that tasks should map to threads based on the criticality level, partitioning away the mixed-criticality characteristic. For example, highly critical tasks should apply static thread scheduling to achieve considerable isolation and have predictable timing behavior, resulting in simpler worst-case execution time (WCET) analysis. Meanwhile, less critical tasks should prioritize improving overall efficiency, thus applying dynamic thread scheduling. In addition, by utilizing fine-grained multithreading with proper thread scheduling, the potential branch latency is removed, thus improving instruction throughput and the timing behavior of the software. Furthermore, although multithreading responds more slowly to a single I/O task than single-threaded processors, it allows concurrent execution of multiple tasks while maintaining good response time on the I/O task. Finally, leveraging fine-grained multithreading, predictable instruction timing behavior, and timing instructions enable the software to interact with various separate I/O operations.

During the microarchitecture design, the priority was to implement a readable model that functions correctly, leading to the use of unconventional implementation decisions that the HDL coder tool does not support. Consequently, the design requires some modification to allow the tool to convert the model-based design (MBD) into HDL code. Additionally, Simulink does not provide a method to verify the critical paths of the microarchitecture. Thus, it is unknown at the time of writing what clock speed the design can achieve. As a result, once the HDL code of the model is available, an analysis of the microarchitecture's critical paths should be performed, and optimize the paths if needed. Due to design complexity, there is some suspicion that the hardware thread scheduler is a limiting factor for the maximum clock frequency and may require some modification. Following the critical path analysis, the microarchitecture should be programmed onto an FPGA to perform on-target functional verification and analysis to ascertain that it is functioning as expected in hardware. Furthermore, hardware testing with external stimuli allows MCS and precision-timed I/O analyses on various thread scheduling configurations to compare and verify timing behavior and response times.

A benefit of using Simulink is the extensive testing capabilities that are possible using the tool. For example, performing Hardware In the Loop (HIL) or Processor In the Loop (PIL) testing of the microarchitecture will provide a detailed picture of how the design operates and where it could fail, leading to rapid fixes in the design choices. As a result, the tool can further expand the testing capabilities, thus making both the hardware and software design choices less error-prone and providing higher confidence for certification.

The custom assembler allowed the writing of simple programs that could be tested in simulation. However, the assembler limits the quality of the analyses that can be performed. Hence, it is worthwhile to adjust the instruction memory to allow the use of a proper compiler, such as GCC or CLANG, where cross-compilation for RISC-V architectures is possible. Furthermore, using such a compiler allows the writing of code in C/C++, making it easier to design more extensive test programs for in-depth analyses. Finally, as a consequence of using a higher-level language such as C/C++, it is possible to include existing software scheduling algorithms to perform complex analyses of larger MCS with tight timing bounds.

The current use of memory is a limiting factor for the design to be implemented in hardware. For example, the instruction memory consists of a script storing the instructions from a text file into a data array. Additionally, the data memory available in the Simulink model is 48 bytes made using registers. Therefore, because there is a limited amount of

registers available, the method does not scale if increasing the size of the data memory when implemented on an FPGA. Furthermore, adding cache for instruction and data memory becomes a problem due to the context-dependent execution time. Consequently, adding cache could significantly degrade the fine-grained prediction of the microarchitecture. However, there are methods available to make the cache more predictable; locking cache lines, caching entire functions, or separate cache for stack and heap [72, 73]. Otherwise, FlexPRET demonstrates the use of scratchpad memories for timing predictable memory [14]. Thus, integrating a more predictable cache or scratchpad memory into the design is desirable for future implementation in hardware.



# Appendix A

## Code

In this appendix chapter, the detailed code written during this project is provided.

### A.1 Python code

Python-code A.1: RiscvAssembler.py

```
1 import pathlib
2 import textwrap
3
4 class RISCv:
5
6     i_type_operations = ["addi", "slti", "sltiu", "xori", "ori", "andi", "jalr",
7                          "slli", "srli", "srai"]
8     r_type_operations = ["add", "sub", "sll", "slt", "sltu", "xor", "srl", "sra",
9                          "or", "and"]
10    b_type_operations = ["beq", "bne", "blt", "bltu", "bge", "bgeu"]
11    j_type_operations = ["jal"]
12    s_type_operations = ["sw"]
13    l_type_operations = ["lw"]
14    u_type_operations = ["lui", "auipc"]
15    csr_type_operations = ["csrrw", "csrrs", "csrrc", "csrrwi", "csrrsi", "csrrci"]
16    t_type_operations = ["delay_until", "interrupt_on_expire",
17                        "deactivate_interrupt", "get_time"]
18    ret_type_operations = ["mret"]
19
20    def __init__(self) -> None:
21        self.assembly_instructions = []
22        self.file = ''
23        self.label_pcs = {}
24        self.num_labels = 0
25        self.num_assembly_lines = 0
26        self.encoded_instructions = []
27
28    def print_lines(self):
29        for line in self.assembly_instructions:
30            print(line)
31
32    def print_starting_instructions(self):
33        for label in self.label_pcs:
34            print("PC: ", int(self.label_pcs[label]), " \tInstr: ",
35                  self.assembly_instructions[int(self.label_pcs[label] / 4)])
36
37    def print_labels(self):
38        for label in self.label_pcs:
39            print("Label:\t", label, "\nPC:\t", self.label_pcs[label])
40
41    def print_encoded_instructions(self):
```

```

38     for idx, instruction in enumerate(self.encoded_instructions):
39         print("ID:\t", idx, "\tInstruction:\t", instruction, "\tSize:\t",
40               len(instruction))
41
42     def read_instructions(self, file_name : str):
43         file = pathlib.Path(file_name)
44         if not file.exists():
45             print("The file does not exist!\r\n")
46             return []
47         read_file = open(file, 'r')
48         lines = read_file.readlines()
49         self._append_assembly_instruction(lines)
50         read_file.close()
51         return self.assembly_instructions
52
53     def encode_riscv_instructions(self, assembly_instructions = []):
54         if assembly_instructions:
55             self.assembly_instructions = assembly_instructions
56         for line in self.assembly_instructions:
57             encoded_instruction = ''
58             self._compute_label_branch_value()
59             if any(operation in line for operation in self.s_type_operations):
60                 s_type_immediate, s_type_rs1, s_type_rs2 =
61                     self._convert_to_machine_code(line, 'store')
62                 imm_upper = s_type_immediate[:7]
63                 imm_lower = s_type_immediate[7:]
64                 if 'sw' in line:
65                     encoded_instruction = imm_upper + s_type_rs2 + s_type_rs1 +
66                                           '010' + imm_lower + '0100011'
67                 self.encoded_instructions.append(encoded_instruction)
68             elif any(operation in line for operation in self.l_type_operations):
69                 l_type_immediate, l_type_rs1, l_type_rd =
70                     self._convert_to_machine_code(line, 'load')
71                 if 'lw' in line:
72                     encoded_instruction = l_type_immediate + l_type_rs1 + '010' +
73                                           l_type_rd + '0000011'
74                 self.encoded_instructions.append(encoded_instruction)
75             elif any(operation in line for operation in self.i_type_operations):
76                 i_type_imm, i_type_rd, i_type_rs1 =
77                     self._convert_to_machine_code(line, 'i_type')
78                 i_type_shamt = i_type_imm[7:]
79                 if 'addi' in line:
80                     encoded_instruction = i_type_imm + i_type_rs1 + '000' +
81                                           i_type_rd + '0010011'
82                 elif 'slti' in line:
83                     encoded_instruction = i_type_imm + i_type_rs1 + '010' +
84                                           i_type_rd + '0010011'
85                 elif 'sltiu' in line:
86                     encoded_instruction = i_type_imm + i_type_rs1 + '011' +
87                                           i_type_rd + '0010011'
88                 elif 'xori' in line:
89                     encoded_instruction = i_type_imm + i_type_rs1 + '100' +
90                                           i_type_rd + '0010011'
91                 elif 'ori' in line:
92                     encoded_instruction = i_type_imm + i_type_rs1 + '110' +
93                                           i_type_rd + '0010011'
94                 elif 'andi' in line:
95                     encoded_instruction = i_type_imm + i_type_rs1 + '111' +
96                                           i_type_rd + '0010011'
97                 elif 'slli' in line:
98                     encoded_instruction = '0000000' + i_type_shamt + i_type_rs1 +
99                                           '001' + i_type_rd + '0010011'
100                elif 'srli' in line:
101                    encoded_instruction = '0000000' + i_type_shamt + i_type_rs1 +
102                                            '101' + i_type_rd + '0010011'
103                elif 'srai' in line:
104                    encoded_instruction = '0100000' + i_type_shamt + i_type_rs1 +
105                                            '101' + i_type_rd + '0010011'
106                elif 'jalr' in line:

```

```

92         encoded_instruction = i_type_imm + i_type_rs1 + '000' +
93             i_type_rd + '1100111'
94         self.encoded_instructions.append(encoded_instruction)
95     elif any(operation in line for operation in self.b_type_operations):
96         b_type_imm, b_type_rs1, b_type_rs2 =
97             self._convert_to_machine_code(line, 'b_type')
98         if 'beq' in line:
99             encoded_instruction = b_type_imm[0] + b_type_imm[2:8] +
100                 b_type_rs2 + b_type_rs1 + '000' + b_type_imm[8:12] +
101                 b_type_imm[1] + '1100011'
102         elif 'bne' in line:
103             encoded_instruction = b_type_imm[0] + b_type_imm[2:8] +
104                 b_type_rs2 + b_type_rs1 + '001' + b_type_imm[8:12] +
105                 b_type_imm[1] + '1100011'
106         elif 'blt' in line:
107             encoded_instruction = b_type_imm[0] + b_type_imm[2:8] +
108                 b_type_rs2 + b_type_rs1 + '100' + b_type_imm[8:12] +
109                 b_type_imm[1] + '1100011'
110         elif 'bltu' in line:
111             encoded_instruction = b_type_imm[0] + b_type_imm[2:8] +
112                 b_type_rs2 + b_type_rs1 + '110' + b_type_imm[8:12] +
113                 b_type_imm[1] + '1100011'
114         elif 'bge' in line:
115             encoded_instruction = b_type_imm[0] + b_type_imm[2:8] +
116                 b_type_rs2 + b_type_rs1 + '101' + b_type_imm[8:12] +
117                 b_type_imm[1] + '1100011'
118         elif 'bgeu' in line:
119             encoded_instruction = b_type_imm[0] + b_type_imm[2:8] +
120                 b_type_rs2 + b_type_rs1 + '111' + b_type_imm[8:12] +
121                 b_type_imm[1] + '1100011'
122         self.encoded_instructions.append(encoded_instruction)
123     elif any(operation in line for operation in self.j_type_operations):
124         j_type_imm, j_type_rd = self._convert_to_machine_code(line,
125             'j_type')
126         if 'jal' in line:
127             encoded_instruction = j_type_imm[0] + j_type_imm[10:20] +
128                 j_type_imm[9] + j_type_imm[1:9] + j_type_rd + '1101111'
129             self.encoded_instructions.append(encoded_instruction)
130     elif any(operation in line for operation in self.r_type_operations):
131         r_type_rd, r_type_rs1, r_type_rs2 =
132             self._convert_to_machine_code(line, 'r_type')
133         if 'add' in line:
134             encoded_instruction = '0000000' + r_type_rs2 + r_type_rs1 +
135                 '000' + r_type_rd + '0110011'
136         elif 'sub' in line:
137             encoded_instruction = '0100000' + r_type_rs2 + r_type_rs1 +
138                 '000' + r_type_rd + '0110011'
139         elif 'sll' in line:
140             encoded_instruction = '0000000' + r_type_rs2 + r_type_rs1 +
141                 '001' + r_type_rd + '0110011'
142         elif 'slt' in line:
143             encoded_instruction = '0000000' + r_type_rs2 + r_type_rs1 +
144                 '010' + r_type_rd + '0110011'
145         elif 'sltu' in line:
146             encoded_instruction = '0000000' + r_type_rs2 + r_type_rs1 +
147                 '011' + r_type_rd + '0110011'
148         elif 'xor' in line:
149             encoded_instruction = '0000000' + r_type_rs2 + r_type_rs1 +
150                 '100' + r_type_rd + '0110011'
151         elif 'srl' in line:
152             encoded_instruction = '0000000' + r_type_rs2 + r_type_rs1 +
153                 '101' + r_type_rd + '0110011'
154         elif 'sra' in line:
155             encoded_instruction = '0100000' + r_type_rs2 + r_type_rs1 +
156                 '101' + r_type_rd + '0110011'
157         elif 'or' in line:
158             encoded_instruction = '0000000' + r_type_rs2 + r_type_rs1 +
159                 '110' + r_type_rd + '0110011'
160         elif 'and' in line:

```

```

135         encoded_instruction = '0000000' + r_type_rs2 + r_type_rs1 +
136             '111' + r_type_rd + '0110011'
137         self.encoded_instructions.append(encoded_instruction)
138     elif any(operation in line for operation in self.u_type_operations):
139         u_type_imm, u_type_rd = self._convert_to_machine_code(line,
140             'u_type')
141         if 'lui' in line:
142             encoded_instruction = u_type_imm + u_type_rd + '0110111'
143         elif 'auipc' in line:
144             encoded_instruction = u_type_imm + u_type_rd + '0010111'
145         self.encoded_instructions.append(encoded_instruction)
146     elif any(operation in line for operation in self.csr_type_operations):
147         csr_type_imm, csr_type_rd, csr_type_rs1 =
148             self._convert_to_machine_code(line, 'csr_type')
149         if 'csrrw' in line:
150             encoded_instruction = csr_type_imm + csr_type_rs1 + '001' +
151                 csr_type_rd + '1110011'
152         elif 'csrrs' in line:
153             encoded_instruction = csr_type_imm + csr_type_rs1 + '010' +
154                 csr_type_rd + '1110011'
155         elif 'csrrc' in line:
156             encoded_instruction = csr_type_imm + csr_type_rs1 + '011' +
157                 csr_type_rd + '1110011'
158         elif 'csrrwi' in line:
159             encoded_instruction = csr_type_imm + csr_type_rs1 + '101' +
160                 csr_type_rd + '1110011'
161         elif 'csrrsi' in line:
162             encoded_instruction = csr_type_imm + csr_type_rs1 + '110' +
163                 csr_type_rd + '1110011'
164         elif 'csrrci' in line:
165             encoded_instruction = csr_type_imm + csr_type_rs1 + '111' +
166                 csr_type_rd + '1110011'
167         self.encoded_instructions.append(encoded_instruction)
168     elif any(operation in line for operation in self.t_type_operations):
169         t_type_rd, t_type_rs2 = self._convert_to_machine_code(line,
170             't_type')
171         if 'get_time' in line:
172             encoded_instruction = '001100100000' + '00000' + '100' +
173                 t_type_rd + '0001011'
174         elif 'delay_until' in line:
175             encoded_instruction = '0011001' + t_type_rs2 + '00000' + '011' +
176                 '00100' + '0101011'
177         elif 'interrupt_on_expire' in line:
178             encoded_instruction = '0011001' + t_type_rs2 + '00000' + '011' +
179                 '00100' + '1011011'
180         elif 'deactivate_interrupt' in line:
181             encoded_instruction = '0000000' + '00000' + '00000' + '000' +
182                 '00000' + '1111011'
183         self.encoded_instructions.append(encoded_instruction)
184     elif any(operation in line for operation in self.ret_type_operations):
185         if 'mret' in line:
186             encoded_instruction = '00110000001000000000000001110011'
187         self.encoded_instructions.append(encoded_instruction)
188     return self.encoded_instructions
189
190 def write_instructions(self, file_name, encoded_riscv_instructions = []):
191     file = pathlib.Path(file_name)
192     if not file.exists():
193         print("The file does not exist!\r\n")
194         return False
195     if encoded_riscv_instructions:
196         encoded_riscv_instructions = encoded_riscv_instructions
197     else:
198         encoded_riscv_instructions = self.encoded_instructions
199     write_file = open(file, 'w')
200     for encoded_instruction in encoded_riscv_instructions:
201         encoded_bytes_list = textwrap.wrap(encoded_instruction, 8)
202         for encoded_byte in encoded_bytes_list:
203             write_file.write(encoded_byte.strip() + '\n')
204     write_file.close()

```

```

191         return True
192
193     def _compute_label_pc(self, filtered_line, idx):
194         if filtered_line:
195             if ':' in filtered_line:
196                 self.label_pcs[filtered_line.strip().replace(':', '')] =
197                     self.num_assembly_lines*4
198                 return ''
199             else:
200                 self.num_assembly_lines += 1
201                 return filtered_line
202
203     def _append_assembly_instruction(self, lines : list):
204         for idx, line in enumerate(lines):
205             filtered_line = self._filter_instructions(line)
206             assembly_instruction = self._compute_label_pc(filtered_line, idx)
207             if assembly_instruction:
208                 self.assembly_instructions.append(assembly_instruction.lower().replace(' ',
209                                                                                       '').split(' '))
210
211     def _remove_comment(self, line : str):
212         split_line = line.split("//")
213         return split_line[0]
214
215     def _remove_empty_lines(self, line : str):
216         if line and not line.isspace():
217             return line
218         else:
219             return ''
220
221     def _remove_newlines(self, line : str):
222         return line.replace("\n", "")
223
224     def _filter_instructions(self, line):
225         no_comment_line = self._remove_comment(line)
226         no_empty_line = self._remove_empty_lines(no_comment_line)
227         no_newlines = self._remove_newlines(no_empty_line)
228         return no_newlines.strip(' ')
229
230     def _find_label_pc(self, self, assembly_instruction):
231         for label in self.label_pcs:
232             if label in assembly_instruction:
233                 return self.label_pcs[label]
234         return -1
235
236     def _replace_branch_label_with_value(self, labeled_assembly_instruction,
237                                         branch_immediate, line_number):
238         current_pc = line_number * 4
239         jump_to_pc = branch_immediate - current_pc
240         return labeled_assembly_instruction[:-1] + [str(jump_to_pc)]
241
242     def _compute_label_branch_value(self):
243         for idx, assembly_instruction in enumerate(self.assembly_instructions):
244             branch_immediate = self._find_label_pc(assembly_instruction)
245             if branch_immediate > 0:
246                 self.assembly_instructions[idx] =
247                     self._replace_branch_label_with_value(assembly_instruction,
248                                                             branch_immediate, idx)
249
250     def _convert_to_machine_code(self, assembly_instruction, operation_type):
251         if operation_type == 'store':
252             s_type_immediate = self._extract_mem_immediate(assembly_instruction)
253             s_type_rs1, s_type_rs2 = self._extract_mem_regs(assembly_instruction)
254             return s_type_immediate, s_type_rs1, s_type_rs2
255         elif operation_type == 'load':
256             l_type_immediate = self._extract_mem_immediate(assembly_instruction)
257             l_type_rs1, l_type_rd = self._extract_mem_regs(assembly_instruction)
258             return l_type_immediate, l_type_rs1, l_type_rd
259         elif operation_type == 'i_type':
260             i_type_imm = self._extract_i_type_immediate(assembly_instruction)

```

```

256         i_type_rd, i_type_rs1 = self._extract_i_type_regs(assembly_instruction)
257         return i_type_imm, i_type_rd, i_type_rs1
258     elif operation_type == 'b_type':
259         b_type_imm = self._extract_b_type_immediate(assembly_instruction)
260         b_type_rs1, b_type_rs2 =
261             self._extract_b_type_regs(assembly_instruction)
262         return b_type_imm, b_type_rs1, b_type_rs2
263     elif operation_type == 'j_type':
264         j_type_imm = self._extract_j_type_immediate(assembly_instruction)
265         j_type_rd = self._extract_j_type_reg(assembly_instruction)
266         return j_type_imm, j_type_rd
267     elif operation_type == 'r_type':
268         r_type_rd, r_type_rs1, r_type_rs2 =
269             self._extract_r_type_regs(assembly_instruction)
270         return r_type_rd, r_type_rs1, r_type_rs2
271     elif operation_type == 'u_type':
272         u_type_imm = self._extract_u_type_immediate(assembly_instruction)
273         u_type_rd = self._extract_u_type_reg(assembly_instruction)
274         return u_type_imm, u_type_rd
275     elif operation_type == 'csr_type':
276         csr_type_imm = self._extract_csr_type_immediate(assembly_instruction)
277         csr_type_rd, csr_type_rs1 =
278             self._extract_csr_type_regs(assembly_instruction)
279         return csr_type_imm, csr_type_rd, csr_type_rs1
280     elif operation_type == 't_type':
281         t_type_rd = ''
282         t_type_rs2 = ''
283         if 'get_time' in assembly_instruction:
284             t_type_rd = self._extract_t_type_reg(assembly_instruction)
285         else:
286             t_type_rs2 = self._extract_t_type_reg(assembly_instruction)
287         return t_type_rd, t_type_rs2
288     else:
289         print(assembly_instruction)
290
291     def _extract_mem_immediate(self, assembly_instruction):
292         starting_parenthesis = assembly_instruction[-1].find('(')
293         memop_offset = assembly_instruction[-1][starting_parenthesis:]
294         return self._convert_signed_binary(memop_offset, 'itype')
295
296     def _extract_i_type_immediate(self, assembly_instruction):
297         itype_immediate = assembly_instruction[-1]
298         return self._convert_signed_binary(itype_immediate, 'itype')
299
300     def _extract_b_type_immediate(self, assembly_instruction):
301         btype_immediate = assembly_instruction[-1]
302         return self._convert_signed_binary(btype_immediate, 'btype')
303
304     def _extract_j_type_immediate(self, assembly_instruction):
305         jtype_immediate = assembly_instruction[-1]
306         return self._convert_signed_binary(jtype_immediate, 'jtype')
307
308     def _extract_u_type_immediate(self, assembly_instruction):
309         utype_immediate = assembly_instruction[-1]
310         return self._convert_signed_binary(utype_immediate, 'utype')
311
312     def _extract_csr_type_immediate(self, assembly_instruction):
313         csr_type_immediate = assembly_instruction[2]
314         return self._convert_signed_binary(csr_type_immediate, 'itype')
315
316     def _extract_mem_regs(self, assembly_instruction):
317         starting_parenthesis, ending_parenthesis =
318             assembly_instruction[-1].find('('), assembly_instruction[-1].find(')')
319         rs1_value_binary =
320             self._convert_signed_binary(assembly_instruction[-1][starting_parenthesis+2:ending_parenthesis],
321                                         'reg')
322         rd_rs2_value_binary =
323             self._convert_signed_binary(assembly_instruction[1][1:], 'reg')
324         return rs1_value_binary, rd_rs2_value_binary

```

```

319     def _extract_i_type_regs(self, assembly_instruction):
320         rd_value_binary = self._convert_signed_binary(assembly_instruction[1][1:],
321             'reg')
322         rs1_value_binary =
323             self._convert_signed_binary(assembly_instruction[2][1:], 'reg')
324         return rd_value_binary, rs1_value_binary
325
326     def _extract_b_type_regs(self, assembly_instruction):
327         rs1_value_binary =
328             self._convert_signed_binary(assembly_instruction[1][1:], 'reg')
329         rs2_value_binary =
330             self._convert_signed_binary(assembly_instruction[2][1:], 'reg')
331         return rs1_value_binary, rs2_value_binary
332
333     def _extract_j_type_reg(self, assembly_instruction):
334         rd_value_binary = self._convert_signed_binary(assembly_instruction[1][1:],
335             'reg')
336         return rd_value_binary
337
338     def _extract_u_type_reg(self, assembly_instruction):
339         rd_value_binary = self._convert_signed_binary(assembly_instruction[1][1:],
340             'reg')
341         return rd_value_binary
342
343     def _extract_r_type_regs(self, assembly_instruction):
344         rd_value_binary = self._convert_signed_binary(assembly_instruction[1][1:],
345             'reg')
346         rs1_value_binary =
347             self._convert_signed_binary(assembly_instruction[2][1:], 'reg')
348         rs2_value_binary =
349             self._convert_signed_binary(assembly_instruction[3][1:], 'reg')
350         return rd_value_binary, rs1_value_binary, rs2_value_binary
351
352     def _extract_csr_type_regs(self, assembly_instruction):
353         val = 0
354         if 'x' in assembly_instruction[3]:
355             val = assembly_instruction[3][1:]
356         else:
357             val = assembly_instruction[3]
358         rd_value_binary = self._convert_signed_binary(assembly_instruction[1][1:],
359             'reg')
360         rs1_value_binary = self._convert_signed_binary(val, 'reg')
361         return rd_value_binary, rs1_value_binary
362
363     def _extract_t_type_reg(self, assembly_instruction):
364         reg_value_binary =
365             self._convert_signed_binary(assembly_instruction[1][1:], 'reg')
366         return reg_value_binary
367
368     def _convert_signed_binary(self, decimal_value, conversion_type):
369         if conversion_type == 'itype':
370             return format(int(decimal_value) & 0xFFF, '012b')
371         elif conversion_type == 'btype':
372             return format(int(decimal_value) & 0x1FFF, '013b')
373         elif conversion_type == 'jtype':
374             return format(int(decimal_value) & 0x1FFFFFF, '021b')
375         elif conversion_type == 'utype':
376             return format(int(decimal_value) & 0xFFFFF, '020b')
377         elif conversion_type == 'reg':
378             return format(int(decimal_value) & 0x1F, '05b')
379         else:
380             return ''
381
382     def compare_file_contents(self, file1_name : str, file2_name : str, file3_name
383         : str):
384         file1 = pathlib.Path(file1_name)
385         if not file1.exists():
386             print("The file does not exist!\r\n")
387             return False
388         file2 = pathlib.Path(file2_name)

```

```

377         if not file2.exists():
378             print("The file does not exist!\r\n")
379             return False
380         file3 = pathlib.Path(file3_name)
381         if not file3.exists():
382             print("The file does not exist!\r\n")
383             return False
384         read_file1 = open(file1, 'r')
385         lines_file1 = read_file1.readlines()
386         read_file2 = open(file2, 'r')
387         lines_file2 = read_file2.readlines()
388         read_file3 = open(file3, 'r')
389         lines_file3 = read_file3.readlines()
390
391         for idx, line in enumerate(lines_file1):
392             if line != lines_file2[idx]:
393                 print("ID:\t", idx, "\t IS INCORRECT!!!\n")
394                 print(file1_name)
395                 print("\tValue:\t", line.strip('\n'))
396                 print("\tASM:\t", ' ',
397                       '.join(self.assembly_instructions[int(idx/4)])')
397                 print(file2_name)
398                 print("\tValue:\t", lines_file2[idx].strip('\n'))
399                 print("\tASM:\t", lines_file3[int(idx/4)])
400                 read_file1.close()
401                 read_file2.close()
402                 return False
403
404         read_file1.close()
405         read_file2.close()
406
407         return True
408
409 if __name__ == '__main__':
410     assembler_riscv = RISCv()
411     assembly_instructions =
412         assembler_riscv.read_instructions('src/multi_threaded.txt')
413     encoded_instructions = assembler_riscv.encode_riscv_instructions()
414     is_file_written = assembler_riscv.write_instructions('Output/instructions.txt')
415
416     # if assembler_riscv.compare_file_contents('Output/instr.txt',
417     #     'Output/instructions.txt', 'src/mthread.txt'):
418     #     print("The results are correct!!")

```

Python-code A.1: RiscvAssembler.py

## A.2 Assembly code

Assembly-code A.2: DataHazardsSingleThreaded.a

```

1 startup:
2     addi x1, x0, -4
3     csrrw x0, 1280, x1          // Set mthreadmode (SZ, SZ, SZ, SZ,
        SZ, SZ, HA)
4     addi x29, x0, 25
5     delay_until x29            // delay_until time 300
6
7 thread0_incrementer:
8     addi x1, x0, 16
9     addi x2, x0, 0              // x2 = 0
10    addi x2, x2, 1              // x2 = x2 + 1
11    bne x1, x2, -4              // if x2 != 32, PC = PC - 4
12
13 thread1_multiplication:
14    addi x3, x0, 17
15    addi x4, x4, 1
16    addi x5, x5, 10
17    blt x4, x3, -8              // if x4 < 17, PC = PC - 8
18
19 thread2_division:
20    addi x6, x0, 400
21    addi x7, x0, 20
22    bge x7, x6, 16
23    addi x8, x8, 1
24    sub x6, x6, x7
25    blt x7, x6, -8              // if x7 < x6, PC = PC - 4
26
27 thread3_read_time:
28    get_time x9
29    addi x9, x9, 25
30    get_time x10
31    blt x10, x9, -4
32
33 thread4_shift_left:
34    lui x11, 1
35    addi x12, x0, 1
36    slli x12, x12, 1
37    blt x12, x11, -4
38
39 thread5_shift_right:
40    lui x13, 1
41    srli x13, x13, 1
42    blt x0, x13, -4
43
44 thread6_fibonacci:
45    addi x14, x0, 15
46    addi x15, x0, 0
47    addi x16, x0, 0              // Show
48    addi x17, x0, 0              // a
49    addi x18, x0, 1              // b
50    add x16, x17, x18             // show = a + b
51    addi x17, x18, 0             // a = b
52    addi x18, x16, 0             // b = show
53    addi x15, x15, 1
54    blt x15, x14, -16
55
56 thread7_even:
57    addi x19, x0, 25
58    addi x20, x0, 0
59    addi x20, x20, 2
60    blt x20, x19, -4

```

Assembly-code A.2: DataHazardsSingleThreaded.a

## Assembly-code A.3: DataHazardsMultiThreaded.a

```

1 startup:
2     csrrsi x1, 3860, 0                //set x1 = mhartid
3     bne x1, x0, main                 //if x1 != 0, Jump to main
4     lui x2, 1048565                  //x2 = 1111111111111110101
5     xori x2, x2, 2730                 //x2 = Sext(1010 1010 1010 1010)
6     csrrw x0, 1280, x2               //Set mthreadmode
7                                     //(SA, SA, SA, SA, SA, SA, SA, SA)
8     lui x31, 484675                  //Set Upper bits of mthreadslot
9                                     //(T7, T6, T5, T4, T3, 0, 0, 0)
10    addi x31, x31, 528                //Set Lower bits of mthreadslot
11                                     //(0, 0, 0, 0, 0, T2, T1, T0)
12    csrrw x0, 1281, x31               // Set mthreadslot
13                                     //(T7, T6, T5, T4, T3, T2, T1, T0)
14    jal x30, main                     //(T0) jump to main
15
16 main:
17    beq x1, x0, thread0_incrementer    //if mhardid == 0,
18                                     //Branch to thread0_incrementer
19    addi x2, x0, 1
20    beq x1, x2, thread1_multiplication //if mhardid == 1,
21                                     //Branch to thread1_multiplication
22    addi x2, x2, 1
23    beq x1, x2, thread2_division       //if mhardid == 2,
24                                     //Branch to thread2_division
25    addi x2, x2, 1
26    beq x1, x2, thread3_read_time      //if mhardid == 3,
27                                     //Branch to thread3_read_time
28    addi x2, x2, 1
29    beq x1, x2, thread4_shift_left     //if mhardid == 4,
30                                     //Branch to thread4_shift_left
31    addi x2, x2, 1
32    beq x1, x2, thread5_shift_right    //if mhardid == 5,
33                                     //Branch to thread5_shift_right
34    addi x2, x2, 1
35    beq x1, x2, thread6_fibonacci      //if mhardid == 6,
36                                     //Branch to thread6_fibonacci
37    addi x2, x2, 1
38    beq x1, x2, thread7_even           //if mhardid == 7,
39                                     //Branch to thread7_even
40    beq x0, x0, 0                      //NOP
41
42 thread0_incrementer:
43    addi x29, x0, 200
44    delay_until x29                    //delay_until time 800
45    addi x1, x0, 16
46    addi x2, x0, 0
47    addi x2, x2, 1
48    bne x1, x2, -4
49    lui x29, 10
50    delay_until x29                    //delay_until time 40960
51    beq x0, x0, 0                      //NOP
52
53 thread1_multiplication:
54    addi x29, x0, 200
55    delay_until x29                    //delay_until time 800
56    addi x3, x0, 17
57    addi x4, x4, 1
58    addi x5, x5, 10
59    blt x4, x3, -8
60    lui x29, 10
61    delay_until x29                    //delay_until time 40960
62    beq x0, x0, 0                      //NOP
63
64 thread2_division:
65    addi x29, x0, 200
66    delay_until x29                    //delay_until time 800
67    addi x6, x0, 400
68    addi x7, x0, 20

```

```

69     bge x7, x6, 16
70     addi x8, x8, 1
71     sub x6, x6, x7
72     blt x7, x6, -8
73     lui x29, 10
74     delay_until x29                //delay_until time 40960
75     beq x0, x0, 0                  //NOP
76
77 thread3_read_time:
78     addi x29, x0, 200
79     delay_until x29                //delay_until time 800
80     get_time x9
81     addi x9, x9, 25
82     get_time x10
83     blt x10, x9, -4
84     lui x29, 10
85     delay_until x29                //delay_until time 40960
86     beq x0, x0, 0                  //NOP
87
88 thread4_shift_left:
89     addi x29, x0, 200
90     delay_until x29                //delay_until time 800
91     lui x11, 1
92     addi x12, x0, 1
93     slli x12, x12, 1
94     blt x12, x11, -4
95     lui x29, 10
96     delay_until x29                //delay_until time 40960
97     beq x0, x0, 0                  //NOP
98
99 thread5_shift_right:
100    addi x29, x0, 200
101    delay_until x29                //delay_until time 800
102    lui x13, 1
103    srli x13, x13, 1
104    blt x0, x13, -4
105    lui x29, 10
106    delay_until x29                //delay_until time 40960
107    beq x0, x0, 0                  //NOP
108
109 thread6_fibonacci:
110    addi x29, x0, 200
111    delay_until x29                //delay_until time 800
112    addi x14, x0, 15
113    addi x15, x0, 0
114    addi x16, x0, 0                //Show
115    addi x17, x0, 0                //a
116    addi x18, x0, 1                //b
117    add x16, x17, x18              //show = a + b
118    addi x17, x18, 0              //a = b
119    addi x18, x16, 0              //b = show
120    addi x15, x15, 1
121    blt x15, x14, -16
122    lui x29, 10
123    delay_until x29                //delay_until time 40960
124    beq x0, x0, 0                  //NOP
125
126 thread7_even:
127    addi x29, x0, 200
128    delay_until x29                //delay_until time 800
129    addi x19, x0, 25
130    addi x20, x0, 0
131    addi x20, x20, 2
132    blt x20, x19, -4
133    lui x29, 10
134    delay_until x29                //delay_until time 40960
135    beq x0, x0, 0                  //NOP

```

Assembly-code A.3: DataHazardsMultiThreaded.a



# Bibliography

- [1] M. Colnatic, “State of the art review paper: advances in embedded hard real-time systems design,” in *ISIE '99. Proceedings of the IEEE International Symposium on Industrial Electronics (Cat. No.99TH8465)*, vol. 1, pp. 37–42 vol.1, 1999.
- [2] X. Fan, “Chapter 1 - introduction to embedded and real-time systems,” in *Real-Time Embedded Systems* (X. Fan, ed.), pp. 3–13, Oxford: Newnes, 2015.
- [3] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Incorporated, 2nd ed., 2011.
- [4] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd ed., 2011.
- [5] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Glenview, IL, USA: Addison-Wesley Educational Publishers Inc, 4th ed., 2009.
- [6] V. Kangunde, R. Jamisola, and E. Theophilus, “A review on drones controlled in real-time,” *International Journal of Dynamics and Control*, vol. 9, pp. 1–15, 12 2021.
- [7] A. Crespo, A. Alonso, M. Marcos, J. A. de la Puente, and P. Balbastre, “Mixed criticality in control systems,” *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 12261–12271, 2014. 19th IFAC World Congress.
- [8] A. Burns and R. I. Davis, *Mixed Criticality Systems - A Review : (13th Edition, February 2022)*. White Rose, February 2022. This is the 13th version of this review now updated to cover research published up to the end of 2021.
- [9] Z. Hu, J. Luo, X. Fang, K. Xiao, B. Hu, and L. Chen, “Real-time schedule algorithm with temporal and spatial isolation feature for mixed criticality system,” in *2021 7th International Symposium on System and Software Reliability (ISSSR)*, pp. 99–108, 2021.
- [10] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber, “A comparison of partitioning operating systems for integrated systems,” in *Computer Safety, Reliability, and Security* (F. Saglietti and N. Oster, eds.), (Berlin, Heidelberg), pp. 342–355, Springer Berlin Heidelberg, 2007.
- [11] D. Wright, Z. Stephenson, and M. Beeby, “Efficient verification through the do-178c life cycle,” in *DO-178C handbook*, 2012.
- [12] P. Puschner and A. Burns, “Guest editorial: A review of worst-case execution-time analysis,” *Real-Time Systems*, vol. 18, pp. 115–128, 05 2000.

- [13] S. Vestal, “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance,” in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pp. 239–243, 2007.
- [14] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, “Flexpret: A processor platform for mixed-criticality systems,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 101–110, 2014.
- [15] Renesas, “Issues with real time performance in conventional rtos and performance improvements through hw-rtos,” tech. rep., Renesas, September 2018.
- [16] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, “Mixed-criticality real-time scheduling for multicore systems,” in *2010 10th IEEE International Conference on Computer and Information Technology*, pp. 1864–1871, 2010.
- [17] S. Baruah and S. Vestal, “Schedulability analysis of sporadic tasks with multiple criticality specifications,” in *2008 Euromicro Conference on Real-Time Systems*, pp. 147–155, 2008.
- [18] C. Garre, D. Mundo, M. Gubitosa, and A. Toso, “Performance comparison of real-time and general-purpose operating systems in parallel physical simulation with high computational cost,” in *SAE Technical Paper*, vol. 1, 04 2014.
- [19] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, “Rtos support for multicore mixed-criticality systems,” in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pp. 197–208, 2012.
- [20] M. Zimmer, *Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2015.
- [21] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom, “The worst-case execution-time problem - overview of methods and survey of tools,” *ACM Trans. Embedded Comput. Syst.*, vol. 7, 01 2008.
- [22] R. Kahil, *Schedulability in Mixed-criticality Systems*. Theses, Université Grenoble Alpes, June 2019.
- [23] S. Baruah, H. Li, and L. Stougie, “Towards the design of certifiable mixed-criticality systems,” in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 13–22, 2010.
- [24] D. de Niz, K. Lakshmanan, and R. Rajkumar, “On the scheduling of mixed-criticality real-time task sets,” in *2009 30th IEEE Real-Time Systems Symposium*, pp. 291–300, 2009.
- [25] N. J. H. Ip and S. A. Edwards, “A processor extension for cycle-accurate real-time software,” in *Embedded and Ubiquitous Computing* (E. Sha, S.-K. Han, C.-Z. Xu, M.-H. Kim, L. T. Yang, and B. Xiao, eds.), (Berlin, Heidelberg), pp. 449–458, Springer Berlin Heidelberg, 2006.
- [26] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, “Predictable programming on a precision timed architecture,” in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES ’08, (New York, NY, USA), pp. 137–146, Association for Computing Machinery, 2008.

- [27] D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke, “Temporal isolation on multiprocessing architectures,” in *Proceedings of the 48th Design Automation Conference*, pp. 274–279, Association for Computing Machinery, 2011.
- [28] S. Suijkerbuijk and B. Juurlink, “Implementing hardware multithreading in a vliw architecture,” in *IASTED PDCS*, pp. 674–679, 01 2005.
- [29] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Wiley Publishing, 10th ed., 2018.
- [30] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 6th ed., 2017.
- [31] S. G. Nayak, “Dynamic branch prediction for embedded system applications,” in *2019 International Conference on Communication and Electronics Systems (ICCES)*, pp. 966–969, 2019.
- [32] V. P. Bharadwaj and M. Kohalli, “Dual decode architecture for dynamic branch prediction,” in *2017 2nd International Conference for Convergence in Technology (I2CT)*, pp. 1140–1143, 2017.
- [33] J. E. Bennett and M. J. Flynn, “Reducing cache miss rates using prediction caches,” tech. rep., Stanford University, Stanford, CA, USA, 1996.
- [34] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: a 32-way multithreaded spar processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.
- [35] I. Liu, *Precision Timed Machines*. PhD thesis, EECS Department, University of California, Berkeley, May 2012.
- [36] D. May, *The XMOS XS1 Architecture*. XMOS Semiconductor Ltd, 2009.
- [37] “Why is model-based design important in embedded systems?.” <https://www.einfochips.com/blog/why-is-model-based-design-important-in-embedded-systems>. Accessed: 2022-05-01.
- [38] “Model-based systems engineering.” <https://www.scaledagileframework.com/model-based-systems-engineering/>. Accessed: 2022-05-01.
- [39] N. Shevchenko, “An introduction to model-based systems engineering (mbse).” Carnegie Mellon University’s Software Engineering Institute Blog, Dec. 21, 2020. [Online]. Accessed: 2022-Apr-28.
- [40] “Maximizing the benefits of model-based design through early verification.” <https://embeddedcomputing.com/technology/software-and-os/simulation-modeling-tools/maximizing-the-benefits-of-model-based-design-through-early-verification>, Nov 2011.
- [41] D. Kaslow, B. Ayres, P. T. Cahill, L. Hart, and R. Yntema, “A model-based systems engineering (mbse) approach for defining the behaviors of cubesats,” in *2017 IEEE Aerospace Conference*, pp. 1–14, 2017.

- [42] B. SchÄtzt, M. Broy, S. Kirstan, and H. Krcmar, *What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?*, vol. 1, ch. 13, pp. 343–369. IGI Global, 01 2011.
- [43] S. Sharma and W. Chen, “Using model-based design to accelerate fpga development for automotive applications,” *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, vol. 2, 04 2009.
- [44] J. Luke and C. Swenson, “Model based design and auto coding of an fpga based satellite control system.” Small Satellite Conference, 2016.
- [45] MathWorks, “Create hdl-compatible simulink model.” <https://se.mathworks.com/help/hdlcoder/gs/create-hdl-compatible-simulink-model.html>. Accessed: 2022-05-02.
- [46] MathWorks, “Generate hdl code from simulink model.” <https://se.mathworks.com/help/hdlcoder/gs/example-generating-hdl-code-from-a-simulink-model.html>. Accessed: 2022-05-02.
- [47] MathWorks, “Reusing reference models in design verification.” <https://se.mathworks.com/campaigns/offers/next/verifying-algorithms-on-fpgas-and-asics/design-verification.html>. Accessed: 2022-05-02.
- [48] MathWorks, “Fpga-based debug and verification.” <https://se.mathworks.com/campaigns/offers/next/verifying-algorithms-on-fpgas-and-asics/fpga-debugging.html>. Accessed: 2022-05-02.
- [49] T. KelemenovÄĵ, M. Kelemen, Ä. MikovÄĵ, V. Maxim, E. Prada, T. LiptÄĵk, and F. Menda, “Model based design and hil simulations,” *American Journal of Mechanical Engineering*, vol. 1, pp. 276–281, 11 2013.
- [50] RISC-V, “About risc-v.” <https://riscv.org/about/>, Visited: 2022-06-16, 2015.
- [51] ARM, “What is an instruction set architecture?” <https://www.arm.com/glossary/isa>, Visited: 2022-06-16.
- [52] A. Singh, N. Franklin, N. Gaur, and P. Bhulania, “Design and implementation of a 32-bit isa risc-v processor core using virtex-7 and virtex- ultrascale,” in *2020 IEEE 5th International Conference on Computing Communication and Automation (ICCCA)*, pp. 126–130, 2020.
- [53] J. Gray, “Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator,” in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 17–20, 2016.
- [54] G. Zhang, K. Zhao, B. Wu, Y. Sun, L. Sun, and F. Liang, “A risc-v based hardware accelerator designed for yolo object detection system,” in *2019 IEEE International Conference of Intelligent Applied Systems on Engineering (ICIASE)*, pp. 9–11, 2019.
- [55] F. EmbedDev, “Rv32i base integer instruction set, version 2.1,” *Five EmbedDev*, 2019. <https://five-embeddev.com/riscv-isa-manual/latest/rv32.html#rv32>, Visited: 2022-06-16.

- [56] F. EmbedDev, “A standard extension for atomic instructions, version 2.1.” <https://five-embeddev.com/riscv-isa-manual/latest/a.html#atomics>, Visited: 2022-06-16.
- [57] S. K. Chen C., Novick G., “Pipelining,” *RISC Architecture*, 2000. <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html>, Visited: 2022-06-16.
- [58] W. N. Wawrzynek J., “Discussion 7 - pipelined cpu.” [https://inst.eecs.berkeley.edu/~cs61c/sp18/disc/7/disc07\\_sol.pdf](https://inst.eecs.berkeley.edu/~cs61c/sp18/disc/7/disc07_sol.pdf), Visited: 2022-06-16, 2018.
- [59] W. N. Wawrzynek J., “Pipelining.” <https://inst.eecs.berkeley.edu/~cs61c/sp18/lec/13/lec13.pdf>, Visited: 2022-06-16, 2018.
- [60] I. E., “Five stages of risc pipeline,” *Gitconnected*, 2021. <https://levelup.gitconnected.com/five-stages-of-risc-pipeline-aad0c3eb1233>, Visited: 2022-06-16.
- [61] F. EmbedDev, “Control and status registers (csrs),” *Five EmbedDev*, 2019. <https://five-embeddev.com/riscv-isa-manual/latest/priv-csrs.html#chap:priv-csrs>, Visited: 2022-06-16.
- [62] F. EmbedDev, “Machine-level isa, version 1.12,” *Five EmbedDev*, 2019. <https://five-embeddev.com/riscv-isa-manual/latest/machine.html#machine>, Visited: 2022-06-16.
- [63] F. EmbedDev, “Machine-level csrs,” *Five EmbedDev*, 2019. <https://five-embeddev.com/riscv-isa-manual/latest/machine.html#machine-level-csrs>, Visited: 2022-06-16.
- [64] S. Inc, “Sifive interrupt cookbook,” tech. rep., SiFive Inc, December 2019.
- [65] F. EmbedDev, “Machine timer registers (mtime and mtimecmp),” *Five EmbedDev*, 2019. <https://five-embeddev.com/riscv-isa-manual/latest/machine.html#machine-timer-registers-mtime-and-mtimecmp>, Visited: 2022-06-16.
- [66] S. Faeroe, “Logic driven verification of functionality for custom microarchitecture,” project report in TFE4590, Department of Electronic Systems, NTNU – Norwegian University of Science and Technology, Dec 2021.
- [67] A. Waterman and K. Asanovic, “The risc-v instruction set manual, volume ii: Privileged architecture, document version 1.12-draft,” *Five EmbedDev*, 2019. <https://five-embeddev.com/riscv-isa-manual/latest/rv32.html#sec:rv32:ldst>, Visited: 2022-06-09.
- [68] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, “The risc-v instruction set manual volume ii: Privileged architecture version 1.9,” Tech. Rep. UCB/EECS-2016-129, EECS Department, University of California, Berkeley, Jul 2016.
- [69] A. Waterman and K. Asanovic, “The risc-v instruction set manual, volume ii: Privileged architecture, document version 1.12-draft,” *Five EmbedDev*, 2019. <https://five-embeddev.com/riscv-isa-manual/latest/machine.html#sec:pmp>, Visited: 2022-06-08.

- [70] K. Cheang, C. Rasmussen, D. Lee, D. Kohlbrenner, K. Asanović, and S. A. Seshia, “Verifying risc-v physical memory protection,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) Workshop on Secure RISC-V Architecture Design*, 2020.
- [71] A. Waterman and K. Asanovic, “The risc-v instruction set manual, volume ii: Privileged architecture, document version 1.12-draft,” *Five Embed-Dev*, 2019. <https://five-embeddev.com/riscv-isa-manual/latest/machine.html#machine-timer-registers-mtime-and-mtimecmp>, Visited: 2022-06-09.
- [72] I. Puaut, “Wcet-centric software-controlled instruction caches for hard real-time systems,” in *18th Euromicro Conference on Real-Time Systems (ECRTS’06)*, pp. 10 pp.–226, 2006.
- [73] M. Schoeberl, “Time-predictable cache organization,” in *2009 Software Technologies for Future Dependable Distributed Systems*, pp. 11–16, 2009.