CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR'S THESIS

Prasoon Dwivedi

FlexPRET real-time processor in heterogenous systems

Department of Cybernetics

Thesis supervisor: **Ing. Martin Košťál**

Supervisor:

Ing. Martin, Košťál
Department of Control Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Technická 2
160 00 Prague 6
Czechia

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Dwivedi Prasoon** Personal ID number: **481622**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Electrical Engineering and Computer Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

FlexPRET real-time processor in heterogenous system

Bachelor's thesis title in Czech:

FlexPRET real-time procesor v heterogenních systémech

Guidelines:

Get familiar with Chisel HDL and FlexPRET CPU
Modify the CPU for usage as softprocessor in an FPGA
Verify, that the softprocessor is working by simple assembly program
Prepare a benchmark to measure performance in real-time application domain and compile it for RISC-V, get results for FlexPRET
Compare the results with other RISC-V processor available as soft-core

Bibliography / sources:

[1] E. Lee, J. Reineke and M. Zimmer, "Abstract PRET Machines," 2017 IEEE Real-Time Systems Symposium (RTSS), Paris, 2017, pp. 1-11, doi: 10.1109/RTSS.2017.00041
[2] Chisel language reference https://www.chisel-lang.org
[3] RISC-V specification https://riscv.org/technical/specifications/
[4] RISC-V intepreter https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/

Name and workplace of bachelor's thesis supervisor:

Ing. Martin Košťál, Department of Control Engineering, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **07.01.2021** Deadline for bachelor thesis submission: _____

Assignment valid until: **30.09.2022**

Ing. Martin Košťál	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Declaration of Independent Work

I hereby declare that this bachelor's thesis is the product of my own independent work and that I have clearly stated all information sources used in the thesis according to Methodological Instruction No. 1/2009 – "On maintaining ethical principles when working on a university final project".

In Prague on ....

Prasoon Dwivedi

Acknowledgements

First, I would like to thank Ing. Martin Košťál for his constant support and guidance. He has been a great advisor, both accommodating my research interests and providing valuable insights that improved my research focus. I would also like to thank everyone at the Industrial Informatics Research Lab for providing such a stimulating environment and valuable feedback on state exams and thesis committees.

Dr. Michal Sojka sparked my initial interest in FPGA development and taught me relevant background knowledge.

I would like to thank my friends for their unwavering support throughout my studies. I was very fortunate to have a fun and thoughtful group of friends. I am deeply grateful to my parents for the values they taught me, the opportunities they have given me, and their love, advice, and encouragement.

Abstract

Mixed-criticality systems, where tasks with different levels of safety criticality are integrated on a single hardware platform to share resources and reduce costs: complicate design and verification. Precision-timed (PRET) machines treat temporal behavior the same way as functionality to achieve good predictability and, this way, attempt to solve mixed-criticality issues.

This thesis aims to study the architectural techniques, generation, and synthesis of one such PRET machine, called FlexPRET: a fine-grained multithreaded RISC-V-based processor. FlexPRET was designed using Chisel, a hardware construction language that generates both C++ and Verilog code. We have deployed FlexPRET on an FPGA and also attempted to evaluate benchmarks using the cycle-accurate emulator.

Keywords: Real-time systems, FPGA design, Chisel HDL, soft-core processor, Instruction sets, Timing, Processor scheduling, Mixed-criticality, Temporal isolation, RISC-V.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Cyber-Physical Systems are integrations of computation, networking, and physical processes [1]. Embedded computers and networks control the physical processes using sensing and actuation. Application areas such as avionics, industrial automation, and medical devices contain real-time embedded systems fulfilling their diverse requirements. These are systems where the timing behavior affects the physical world. As a result, software timing behavior is essential to develop.

Driven by the demands in real-time domains to reduce costs, size, and power of embedded hardware while maintaining system complexity, mixed-criticality is a current trend in real-time embedded systems. It is how embedded systems can integrate tasks with different safety criticalities on a single hardware platform to share resources and reduce costs. The number of criticality levels and how they are defined may vary, but more than one level forms a mixed-criticality system. For example, the DO-178C standard describes five safety criticality levels for avionics [2]. Tasks with higher criticality levels cost more, both in design and verification.

1.1 Problem Statement

Critical tasks that grow in complexity and demand expensive design and verification require temporal and spatial isolation while running in a mixed-criticality system. Isolation can be achieved in software by a real-time operating system (RTOS). Although this can reduce hardware costs, the RTOS itself must be verified and certified. Hardware-based isolation can be achieved by deploying each task on separate components: processors, cores on a multicore processor, or hardware threads on multithreaded processors. One task per thread can better utilize resources by allowing multiple tasks to execute on a single processor, but thread scheduling must preserve hardware-based temporal isolation.

Some existing fine-grained multithreaded processors can preserve isolation, but they have inflexible thread scheduling mechanisms. PTARM isolates each thread, but precisely four threads must be constantly active to utilize the processor fully [3]. The problem now becomes not just meeting task deadlines but also utilizing hardware efficiently while maintaining flexibility for the different types of deadlines (criticality levels).

1.2 Structure of this Thesis

This thesis aims to study the architectural techniques, generation, and synthesis of FlexPRET: a fine-grained multithreaded RISC-V-based processor. This thesis work is structured into six main chapters :

- The first chapter gives an introduction to the problem, the challenge, and the source of inspiration.
- In the second and third chapters, we establish initial conditions and discuss various principles necessary to understand the problem and the solution design fully.
- Chapter four introduces more profound concepts of the FlexPRET processor and the implementation of the project.
- Finally, the fifth and sixth chapters describe results, discussions, conclusions, and proposals of future work.

Chapter 2

Preliminaries

Contents

2.1 Introduction to FPGA

FPGA stands for Field Programmable Gate Array. FPGAs consist of massive collections of unconnected digital components like multiplexers, logic gates, and more complex components. Programming an FPGA means creating connections between these different components to digitally create a complex system while providing a very high level of flexibility and parallelism. They are the closest one can get to simulating hardwired circuits. This section introduces FPGA concepts and discusses how the design flow of FPGAs is different from that of a microcontroller.

2.1.1 Historical connection with digital electronics

Digital electronics is concerned with circuits that represent information using a finite set of output states. Most applications use just two states, often labeled '0' and '1'. Different mappings between these states and the corresponding output voltages or currents characterize logic families.

From the first Transistor-Transistor logic families, improvements to satisfy the demand of programmability have led to the invention of complex programmable logic devices(PLDs) and FPGAs. Programmability here means the ability of a designer to affect the logic behavior of a chip after it has been produced in the factory. FPGAs contain many simple logic blocks with increased programmable interconnections, illustrating the peak of programmability in modern electronics.

2.1.2 Basic FPGA architecture

Although it is logical to think of an FPGA as an array of unconnected digital components, in reality, FPGAs still have a fixed structure. The basic architecture of a typical modern FPGA is composed of:

- CLBs (configurable logic blocks) - They are the main building blocks of an FPGA and typically consist of a few inputs, look-up tables (LUTs), multiplexers, and some random access memory (RAM).
- I/O (input/output) blocks - These physical ports get data in and out of the FPGA.
- Configuration flash memory - FPGA uses this to configure interconnections as well as other internal components.

FPGA architectures incorporate these essential elements along with additional computational and data storage blocks such as embedded memories or clocking. All the components listed above typically account for less than 20% of the silicon inside an FPGA chip. The figure does not show the large amounts of programmable interconnect and the auxiliary circuits that 'program' the generic block to become a well-defined piece of logic. This silicon inefficiency is the price to pay for programmability and is the reason why FPGAs have been more successful in high-end, low-volume applications.

2.1.3 FPGA design flow

A microprocessor executes instructions sequentially, one after another. This only changes when an interrupt occurs, and then the interrupt only changes the order of execution from whatever it was doing, to doing something new. When finished with processing the interrupt, the code returns to what was interrupted and continues to execute the instructions where it left off. A hardware description language (HDL), like Verilog, is different. While a program describes an algorithm or a task, the HDL describes a circuit, or a hardware description of a design, which forms a machine to solve an algorithm or work on a task.

The most common flow used in the design of FPGAs involves the following phases:

- Design entry. This step consists of transforming design ideas into some form of computerized representation. This is accomplished using HDLs. The two most popular HDLs are Verilog and VHDL (Very-High-Speed-Integrated-Circuit HDL). However, the language we are about to deal with in this thesis will be Chisel HDL, an alternative to classic HDLs. HDLs are not tools to design electronic circuits. They differ from conventional software programming languages because they do not support the sequential execution of statements in code.

- Synthesis. The synthesis tool receives HDL and a choice of FPGA vendor and model. Using these two pieces of information, the tool generates a netlist that satisfies the logic behavior specified in the HDL files. Most synthesis tools go through additional steps such as logic optimization, register load balancing, and other techniques. The resulting netlist is a very efficient implementation of the HDL design.

- Place and route. The placer takes the synthesized netlist and chooses a place for each of the primitives inside the chip. The router's task is then to interconnect all these primitives together and satisfy the timing constraints. The most apparent constraint for a design is the frequency of the system clock, but there are more constraints designers can place using software packages supported by vendors.

- Bitstream generation. FPGAs are typically configured at their power-up time from some kind of configuration flash memory. Once the place and route process is finished, the resulting configuration must be stored in a file to program the flash. That file is called a bitstream.

Out of these four phases, only the first one is human-labor intensive. Designers have to type in the HDL code, which can be tedious, for example, lots of digital signal processing. This tediousness is the reason for the appearance of alternative design flows, which include a preliminary phase in which the user can draw blocks at a higher level of abstraction and rely on software tools for the generation of the HDL. It is also one of the motivations behind the invention of Chisel HDL.

2.2 Chisel HDL

Chisel [4] is an open-source HDL used to describe digital electronics and circuits at the register-transfer level. It is an alternative to classic HDLs like Verilog or VHDL. Chisel is based on Scala as an embedded DSL (domain-specific language), and it inherits the object-oriented and functional programming aspects of Scala. Using Scala this way provides designers with the power of a modern programming language to write complex circuit generators. Circuits described in Chisel can be converted to a description in Verilog for synthesis and simulation. This generator methodology also enables the creation of reusable components and libraries. The need for Chisel, or rather the need for alternative HDLs that improve designer productivity, is discussed in the section titled 'Chisel vs. Classic HDLs'.

2.2.1 Chisel code

Consider an FIR filter implementing a convolution operation, as depicted in Figure 2.1.

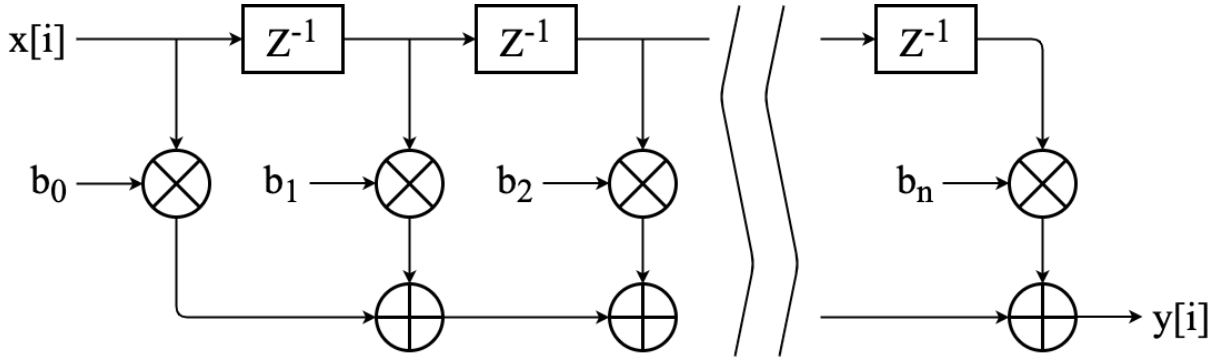

Figure 2.1: FIR filter

Chisel provides similar base primitives as synthesizable Verilog, and one could write 'Verilog-like' Chisel. That would mean control over every multiplexer (MUX) or every bit-width in the design. That type of control is entirely possible with Chisel. In other words, Chisel provides no loss of expressibility. As an example, given below is a Chisel module. It computes a moving average, and its structure is similar to an FIR filter where we register some input over a few cycles, then do some computations where we weight them equally and add them together to get a moving average.

```
// 3-point moving sum implemented in the style of a FIR filter
class MovingSum3(bitWidth: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt(bitWidth.W))
  })

    val z1 = RegNext(io.in)
    val z2 = RegNext(z1)
    io.out := (io.in * 1.U) + (z1 * 1.U) + (z2 * 1.U)
}
```

While this approach works perfectly, the power of Chisel comes to form the ability to create generators. This "Verilog-like" approach would prove inefficient if we wanted, for example, a generic FIR filter and not this specific instance of an FIR filter.

```
// Generalized FIR filter parameterized
// by convolution coefficients
class FirFilter(bitWidth:Int,coeffs:Seq[UInt]) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt(bitWidth.W))
  })
// Create the serial-in, parallel-out shift register
  val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))
  zs(0) := io.in
  for (i <- 1 until coeffs.length) {
    zs(i) := zs(i-1)
  }

  // Do the multiplies
  val products=VecInit.tabulate(coeffs.length)(i =>zs(i)*coeffs(i))

  // Sum up the products
  io.out := products.reduce(_ + _)
}
```

This new instance shows a massive increase in parameterizability and is starting to look like "software-like" Chisel. We are still parameterizing by the bit-width, but now we're passing in a sequence of coefficients for the FIR filter. Now, we can programmatically iterate over these coefficients and construct all the registers to do the delaying and compute all the products which we can reduce and sum together. This kind of meta-programming enables powerful parameterization and if we think back to the three-point moving average filter, we can call this generic FIR filter and pass in the sequence of constants.

```
val movingSum3Filter=Module(new FirFilter(8,Seq(1.U,1.U,1.U)))
// same 3-point moving sum filter as before
```

More importantly, we can now use this generic filter in lots of other places too, for example, to create a delay filter or a triangle filter, all with the same code. This freedom to write generic generators that capture design patents is what Chisel enables. All of which can be converted to synthesizable Verilog.

2.2.2 Chisel development

To meet the ever-growing demand for improving computation in hardware design, Chisel uses a software stack, but for hardware. Software development employs compilers like clang [5], and built upon the compilers, is a powerful language like C++, and built upon the language, are libraries and projects. Similarly, Chisel uses a compiler stack that interacts with the Chisel language frontend and enables more RTL (register transfer level) transformations. This is where FIRRTL (flexible intermediate representation for register transfer level) comes in: it is an extendable hardware compiler framework [6]. FIRRTL represents the standardized elaborated circuits produced by Chisel. FIRRTL's structure is composed of different transformations. The designer passes in a circuit and metadata and annotations, a simple transformation occurs, and outputs a modified circuit with modified metadata. This continues in a kind of pipelined fashion. Since it has been expressed in this way, it is very straightforward to add a custom transform.

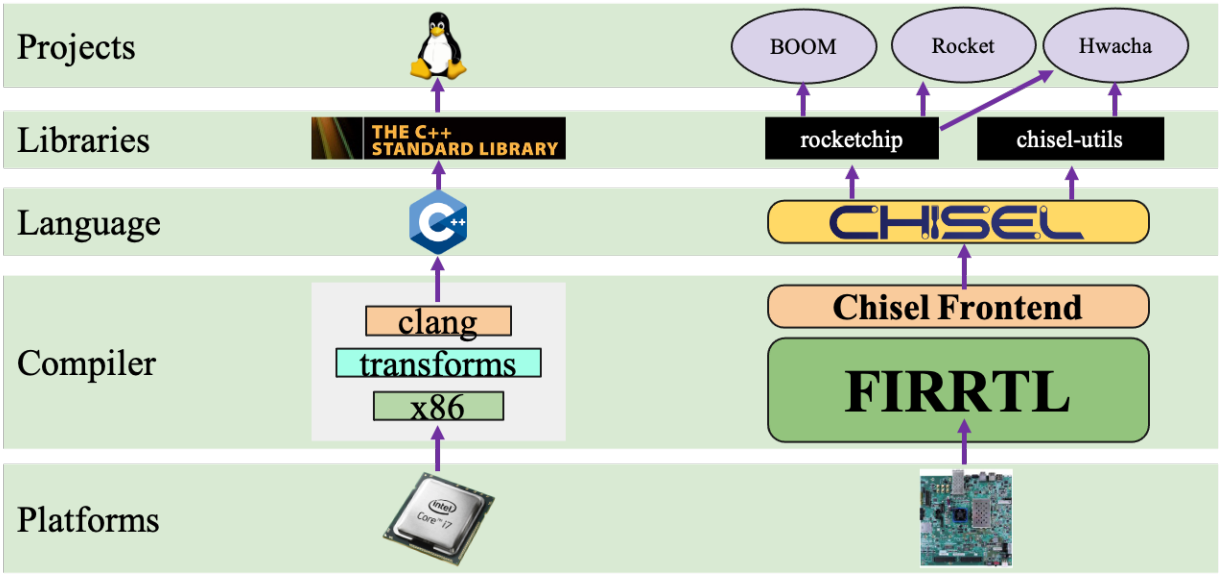

Figure 2.2: Software stack, but for hardware

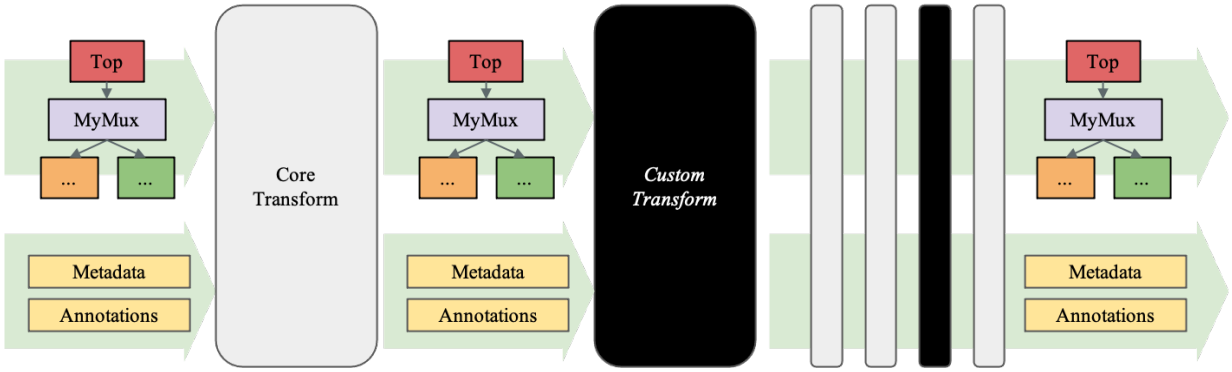

Figure 2.3: FIRRTL: an extendable hardware compiler framework

A FIRRTL compiler is constructed by chaining together these transformations as shown in Figure 2.3, then writing the final circuit to a file. The Chisel-to-Verilog process takes form in multiple stages: The Chisel stage or the frontend compiles Chisel to an intermediate circuit representation called FIR (flexible intermediate representation). The FIRRTL stage, or "mid-end" optimizes FIR for RTL and applies user custom transformations. Finally, the Verilog stage or backend emits Verilog based on the optimized FIRRTL transformations. The Chisel frontend can be very lightweight, and additional HDLs written in other languages can also target FIRRTL and reuse most of the compiler toolchain.

2.2.3 Chisel vs. classic HDLs

Let's think about comparing languages and try to find the advantages of Python over C. We could bring up the following points:

- Everything that I can write in C, I can write in Python
- C has features that Python doesn't, like inline assembly.
- Both are Turing complete.

The problem with this comparison is that it ignores the fact that Python brings new programming paradigms that increase productivity, like object-oriented programming or functional programming or support for libraries. This way, Python can be considered as a stronger language from a design productivity and code reuse perspective. But the existence of such paradigms does not force you to use them. Because it is entirely possible to write Python code that feels just like C. This means a better question to compare the languages would be, "what can be built with Python that would be incredibly hard with C?" Answering this question is quickly out of the scope of "Hello World" comparisons. Complex problems like building machine learning libraries would be incredibly difficult in C, but much easier in Python because of the new programming paradigms brought by Python.

Chisel is a domain-specific language embedded in Scala. By its very nature, it provides very similar constructs to Verilog. This causes the most basic Chisel examples to look precisely like Verilog. We could use this as an argument and dismiss Chisel, but that would be analogous to making a choice based on the structure of "Hello World" syntaxes. But complex projects like Rocket-Chip [7] [8], which is a generator of System-on-Chip designs, would be much easier to design in Chisel because of the new programming paradigms Chisel brings. Rocket-Chip can be used as a library, which means that a designer can virtually "import a RISC-V microprocessor" the same way they would "import Matplotlib" [9].

The Chisel-to-Verilog process takes form in different stages, which enables two things:

- The frontend and backends are decoupled. This means that other frontends and backends can be written. For example, Magma [10], which is another HDL embedded in Python, can directly target the FIRRTL stage in Chisel's compiler stack. New frontends get all the benefits of mid-end optimizations and available backends. New backends can also be written. For example, a VHDL backend.

- Chisel's compiler framework enables automated specialization and transformation of circuits. This means circuits that are transformed to FPGA optimized versions run faster than unoptimized versions. The framework can also enable hardware breakpoints or assertions and add run-time configurable fault injection capabilities. Doing these optimizations in Verilog would be very complex and brittle.

The best way to compare Chisel to classic HDLs would be by comparing the set of programming paradigms Chisel enables. The way to tackle this is to do deep dives on mature Chisel codebases, which takes time. Additionally, the skillsets for reading these code bases and making judgments are not ordinary in hardware engineers. Hardware engineers are usually very proficient with C, but not with object-oriented programming, functional programming, or complex projects that use modern software engineering principles. This sets up biases against Chisel or similar languages.

Chapter 3

Processors and Architectures

Contents

A processor is the heart of an embedded system. It is the basic unit that takes input and produces output after processing the data. Processor architectures categorize how data is moved around inside a processor. This includes things like pre-fetch cues, parallel execution paths, stack operations, and caching. Processor architecture is an abstract model of a computer and might be the most crucial type of hardware design. This chapter will discuss architecture terminology and set foundations for the flexPRET processor design.

3.1 ISA

An ISA (instruction set architecture) is an abstract model of a computer. In general, it defines the supported data types, the registers, the hardware support for managing main memory, fundamental features (such as memory consistency, addressing modes, virtual memory), and the input/output model of a family of implementations of the ISA. The purpose of ISAs is to allow designers to express a program at a higher level, making it easier to understand and less implementation-specific.

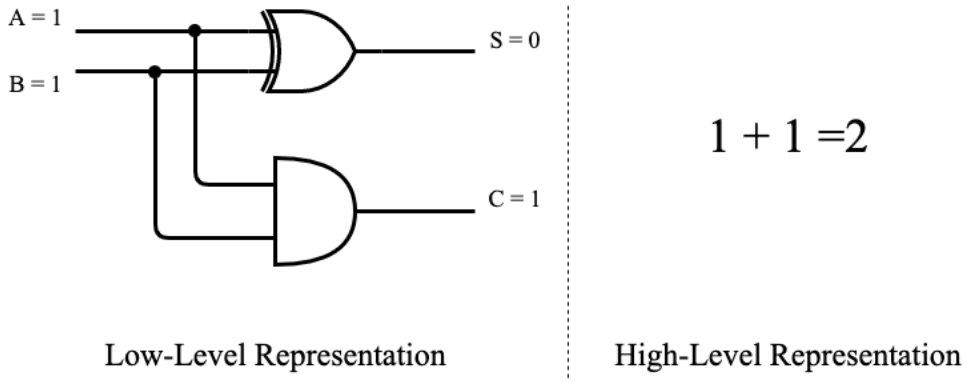

Figure 3.1: The concept behind an ISA is abstraction

Most ISAs allow designers to work with a CPU using simple instructions that are bit strings encoded with crucial information. Each instruction contains an operation code, which defines what operation the CPU must carry out and the operation parameters. Interestingly, the instruction itself does not necessarily say anything about which components are in charge of doing what. Instructions are implementation-independent. This independence provides binary compatibility between implementations.

3.2 RISC

RISC (reduced instruction set computer) is a type of processor architecture that utilizes a small, highly optimized set of instructions rather than a more specialized set of instructions often found in other types of architectures. RISC has five design principles:

- Single-cycle execution - Since RISC works with simple instructions, RISC designs emphasize single-cycle execution even on complex CPUs.
- Hard-wired control with little microcode - Microcode adds a layer of interpretive overhead, raising the number of cycles per instruction.
- Simple instructions, few addressing modes - Simple instructions, hence simple instruction decoding. Complex instructions which entail microcode or multicycle instructions are avoided.

- Load and Store; large number of registers - Only loads and stores access memory; all others perform register-register operations. A large number of registers prevent interactions with memory.
- Efficient, deep pipelining - Pipelining makes use of hardware parallelism without the complexities of horizontal microcode. An n-stage pipeline keeps up to 'n' instructions active at once.

Pipelining is a standard feature in RISC processors. Because the processor works on different steps of instruction at the same time, more instructions can be executed in a shorter period of time. While different processors have different numbers of steps, they are basically variations of these basic five:

- Fetch instructions from memory (IF)
- Read registers and decode the instruction (ID)
- Execute the instruction or calculate an address (EX)
- Access an operand in data memory (MEM)
- Write the result into a register (WB)

Instruction no.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 3.2: Classic five-stage RISC pipeline. In the green column, the earliest instruction is in the WB stage, and the latest instruction is fetched.

3.3 RISC-V ISA

RISC-V (pronounced "risk five") is a well-organized and categorized ISA. It defines the software interface for hardware. RISC-V is open source and can be used to build hardware designs free of intellectual property (IP) and licensing restrictions. This leads to the reason why there is faster adoption of RISC-V in the industry; the fact that it is open-source paves a new business model for hardware vendors. The RISC-V ISA is on par with modern CPUs in terms of performance, code density, and power consumption. RISC-V has a modular design consisting of alternative base parts, with added optional extensions. The ISA base and its extensions are developed collectively between industry, the research community, and educational institutions.

The ISA base [11] specifies instructions (and their encoding), control flow, registers (and their sizes), memory and addressing logic (i.e., integer), manipulation, and ancillaries. The base alone can implement a simplified general-purpose computer with full software support, including a general-purpose compiler. The base integer ISA has four instruction type formats [11]:

- R-format:
  - **opcode**: partially specifies operation
  - **funct7** + **funct3**: combined with opcode, describe operation to perform
  - **rs1**: first operand
  - **rs2**: second operand
  - **rd**: destination register

Bit Length	7	5	5	3	5	7
Field Name	funct7	rs2	rs1	func3	rd	opcode

Table 3.1: R-format instruction

- I-format:
  - **opcode**: uniquely specifies instruction
  - **rs1**: specifies register operand
  - **rd**: specifies destination operand
  - **imm**: 12-bit signed immediate

Bit Length	12	5	3	5	7
Field Name	imm[11:0]	rs1	func3	rd	opcode

Table 3.2: I-format instruction

- S-format (store):
  - **rs1**: register for base memory
  - **rs2**: register for data and immediate offset
  - Stores do not write to the register file, so no 'rd'
  - Register names are more critical than immediate bits in hardware design

Bit Length	7	5	5	3	5	7
Field Name	imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode

Table 3.3: S-format instruction

- U-format (upper immediate):
  - 20-bit immediate in upper 20 bits of 32-bit instruction word
  - One destination register, rd
  - Used for two instructions: Load upper immediate or add upper immediate to PC

Bit Length	20	5	7
Field Name	imm[31:12]	rd	opcode

Table 3.4: U-format instruction

The RISC-V base integer ISA can be extended but not redefined [11]. New instructions can be added to the set to support special semantics. One such example is real-time semantics, a significant part of FlexPRET's infrastructure. Instructions such as `get_time` and `set_compare` are added for the ability to read timestamps and define time boundaries. These extended timing instructions are also discussed in section 3.5 "PRET machines".

3.4 Mixed-Criticality Systems

Real-time computing refers to a real-time constraint, for example from event to system response. Real-time systems must guarantee a response within their specified time constraints. These time constraints, combined with verification requirements, form the concept of criticality. When tasks with different criticality levels (different time and verification constraints) are executed on a single platform, it is called a mixed-criticality system.

Criticality also represents the required level of assurance against failure for a task or a component. For domains like automotive or avionics, each criticality level has different certifications, which specify design and verification methodologies. The number and definition of criticality levels may vary, but the minimum is two levels, one critical and the other non-critical. Since we are concerned with cyber-physical systems, failures in critical tasks directly impact the behavior of the physical systems they associate with. For example, the DO-178C standard defines five design assurance levels in avionics: (A) Catastrophic, (B) Hazardous, (C) Major, (D) Minor, and (E) No Safety Effect [2]. However, the only criticality levels contained in the scope of this thesis will be hard real-time (must meet deadline) and soft real-time (reduced utility with deadline miss).

Since both critical (hard real-time) and non-critical (soft real-time) tasks bust execute on the same hardware platform, partitioning the hardware platform in both space and time is an essential technique used in mixed-criticality systems. This is done by privatizing memory segments and I/O devices to partition space and allocating time segments for processor or shared resource usage to partition time. Such isolation can either be achieved by software (using a real-time operating system) or by hardware (using different cores or threads). The methodology for deploying mixed-criticality applications uses a combination of hardware-based and software-based partitioning:

- Only tasks of the same criticality level are assigned to each hardware thread, with fewer tasks per hardware thread at higher criticality levels.
- For hardware threads with higher criticality tasks, hard real-time threads (HRTTs) are used, and scheduling resources are over-allocated.
- For hardware threads with lower criticality tasks, soft real-time threads (SRTTs) are used, and scheduling resources are under-allocated.
- The scheduling algorithm with the highest confidence is used if multiple tasks are mapped on the same hardware thread. Static scheduling algorithms are preferable over dynamic scheduling algorithms for higher criticality tasks.

A simple mixed-criticality example being executed on FlexPRET is discussed in Chapter-4.

3.5 PRET Machines

For embedded software applications, computer architecture, software, and networking have gone too far down the path of emphasizing average-case performance over timing predictability. Therefore, a complete rethinking of architecture has been necessary. In 2007, Edwards and Lee made a case for precision timed (PRET) machines as a solution [12]: temporal behavior is as important as logical function. This was an enormous problem because it spanned nearly all abstraction layers in computing, including programming languages, virtual memory, memory hierarchy, pipelining techniques, power management, I/O, etc. PRET systems ushered in a new era where predictability and performance coexist. To describe the FlexPRET processor, this chapter begins with explaining the underlying principle of PRET machines and then the architecture of FlexPRET.

The PRET principle is to treat temporal behavior the same way as functionality to achieve good predictability. Integral features of the PRET solution are:

3.5.1 Hardware Threads

A hardware thread is logically a separate processor with its own program counter and registers, but it shares the pipeline with other hardware threads. For example, four hardware threads appear to each execute at 25 MHz if interleaved on a 100 MHz pipeline (each hardware thread enters the pipeline once every four cycles). Each hardware thread either executes a single task or uses software-based scheduling for multiple tasks. PRET machines employ as many hardware threads as pipeline stages, although one thread less than the number of pipeline stages is sufficient. Furthermore, the pipeline must be thread-interleaved. A fine-grained thread-interleaved pipeline fetches instructions from different threads every cycle. Fine-grained multithreading also enables hardware-based isolation between tasks that are deployed to separate hardware threads, but isolation still depends on hardware thread scheduling.

Multithreaded processors have hardware support for thread-level parallelism through hardware threads. Each hardware thread has its own set of registers to save its state, and the processor switches between threads through interleaving. Similar to virtual machines in general-purpose computing, the hardware threads in FlexPRET can be considered virtual real-time machines; they provide guarantees on execution resources at the cycle level and hardware support for mechanisms typically provided by a real-time operating system. Each hardware thread can be programmed using different languages or techniques and isolated from other threads' interrupts. The classification and working of hardware threads in FlexPRET are discussed in the section titled 'FlexPRET'.

3.5.2 Scratchpad Memories

Instead of a cache, processors can use scratchpad memories, which are local memories with contents controlled by software and not by hardware [13] [14]. Scratchpad memories can be thought of as a distinct part of the memory with their own memory addresses, and they can be accessed in a way similar to the main memory. Scratchpad memory management has many similarities to cache locking, with memory addressing being the primary difference. Most scratchpad allocation techniques optimize for average-case execution time or energy consumption, not for worst-case execution time. Techniques can be static or dynamic depending on if the contents change during run-time. FlexPRET uses scratchpad memories but could be adapted to use other predictable memory hierarchies.

3.5.3 Thread-interleaved pipelines

Branch predictors do not always make correct predictions. Incorrect predictions may cause the wrong instructions to be fetched into the pipeline, and the correct instruction may not even be found in the cache. This would mean wasting thousands of clock cycles and missing penalties. This makes branch predictors a significant source of unpredictability.

One way to get around this problem is to use a thread-interleaved pipeline instead of an ordinary deep pipeline and employ as many hardware threads as pipeline stages. The pipeline will be scheduled to fetch instruction from different threads every single cycle in a fine-grained thread-interleaved pipeline. As a result, branches will always be resolved before the next instruction in that thread is fetched into the pipeline, ensuring that the correct instruction is always fetched. FlexPRET removes dynamic branch prediction and hides branch latency with hardware thread concurrency, and it isolates interrupts to particular hardware threads.

3.5.4 ISAs with timing instructions

ISAs are extended with timing instructions for reading time from an internal register and putting lower and upper time bounds on program flow. Timing instructions set and clear deadlines for each task. Most ISAs do not provide any means to control time explicitly; time control can only be done indirectly through software and existing hardware [15], and of course, by extending ISAs. Timing instructions add temporal semantics to programs but do not fully specify behavior. For FlexPRET, the RISC-V ISA has been extended with timing instructions to express real-time semantics. In FlexPRET, time is represented by a nanosecond value instead of counting clock ticks, starting at zero when powered on.

Chapter 4

FlexPRET

Contents

4.1 Description of the processor

The PRET system studied in this thesis is FlexPRET, a fine-grained thread-interleaved RISC-V-based processor for mixed-criticality systems, developed at UC Berkeley [16]. FlexPRET uses a thread-interleaved 5-stage RISC pipeline. It is based on the RISC-V ISA extended with timing instructions that use a designated platform clock. It employs a thread scheduler that implements flexible scheduling to support mixed-criticality systems. This section discusses the microarchitectural design techniques involved in FlexPRET.

4.1.1 Complexity

The general complexity of a processor is the highest-level design involved. An 8-bit non-pipelined processor is simpler and cheaper than a 64-bit superscalar processor with a dozen pipeline stages but has lower performance. In the real-time embedded domain, processors trade-off between cost and performance. FlexPRET is a 32-bit, 5-stage, fine-grained multithreaded processor implementing the base RISC-V ISA. The base ISA is small but usable and enables an efficient minimal hardware implementation, such as a soft-core on an FPGA. Optional extensions add target-specific functionality, such as floating-point operations, but for smaller code and hardware size, FlexPRET uses the 32-bit version without any extensions (RV32I [11]).

4.1.2 Pipeline

FlexPRET's pipeline is based on the classic 5-stage RISC pipeline. It supports an arbitrary interleaving of hardware threads and enforces scheduling decisions. Once an instruction is fetched, its execution through the pipeline is isolated from the behavior of other hardware threads. An overview of the datapath is shown in Figure 4.1. The tall rectangles with small triangles at the bottom represent pipeline registers that store signals between pipeline stages.

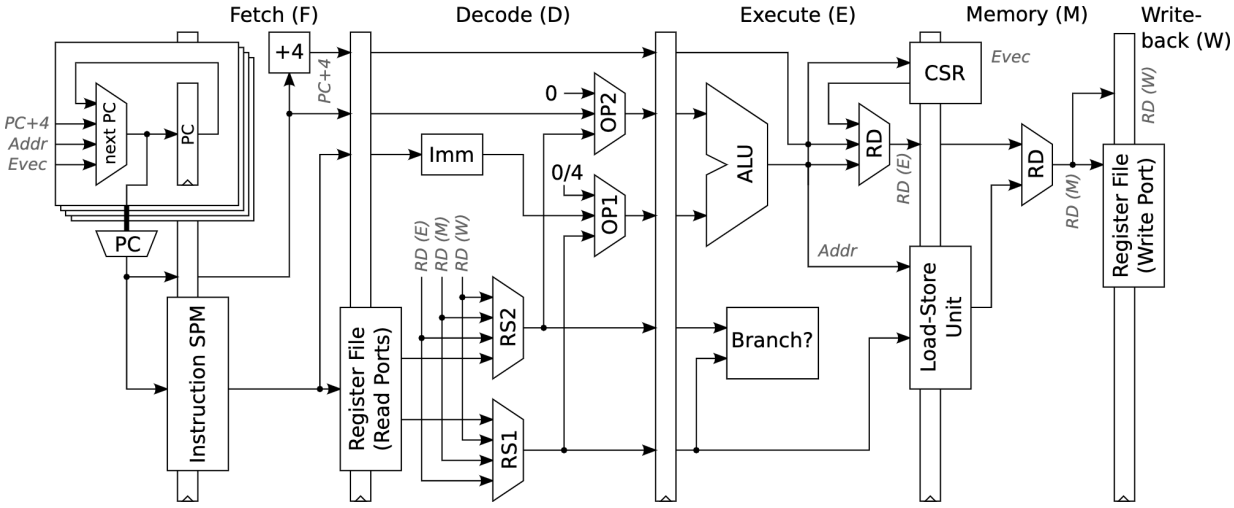

Figure 4.1: A high-level diagram of FlexPRET's datapath

The *fetch* stage retrieves the instruction stored in the ISPM (instruction scratchpad memory) at the program counter of the scheduled hardware thread. In the *decode* stage, the control unit determines the control signals required for the datapath to properly execute the instruction. The *execute* stage is the most complex in this pipeline. The ALU (arithmetic logic unit) computes the specified operations, either producing the destination register data, or an address used as a program counter or memory location. The execute stage also contains the control and status register unit that handles interrupts, thread scheduling configuration, and timing instructions. The load-store unit handles memory operations to the ISPM read-write port, the DSPM (data scratchpad memory) , or the peripheral bus, depending on the address provided. In the *memory* stage, the load-store unit returns data for any load instruction. The destination register data, either from the load-store or execute stage is connected to the write port of the register file. The *writeback* stage registers provide a copy of the data being stored in the register file.

4.1.3 Hardware Thread Scheduler

The temporal isolation of each hardware thread depends on how it is scheduled. The most compelling technique used in FlexPRET is its flexible, software-controlled thread scheduler. FlexPRET classifies hardware threads as either HRTTs (hard real-time threads) or SRTTs (soft real-time threads). FlexPRET's hardware thread scheduler provides predictable and isolated execution to HRTTs while allowing SRTTs to efficiently utilize spare cycles. Since the pipeline will support an arbitrary interleaving of hardware threads, the scheduler must only meet the HRTT and SRTT property requirements.

Consider a mixed-criticality system that consists of three independent periodic tasks τ_A, τ_B, τ_C where each task has a deadline equal to its period T_i. Each task executes on its own thread, with hard real-time tasks τ_A and τ_B on HRTTs and soft real-time task τ_C on an SRTT. FlexPRET's thread scheduler ensures that hard real-time tasks are executed at a constant rate for isolation and predictability, and when a cycle is not being used for a hard real-time task, that cycle is used by a soft real-time task.

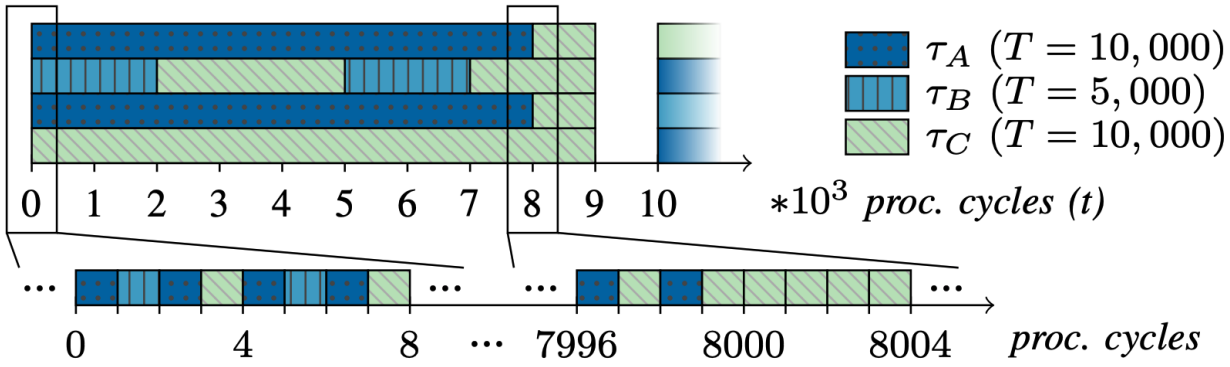

Figure 4.2: FlexPRET executing a simple mixed-criticality example.

In figure 4.2, the vertical direction shows the thread from which an instruction is fetched each cycle over a four-cycle interval. In each four-cycle interval, τ_A is allocated the first and third cycle, τ_B the second cycle, and the fourth cycle is unallocated. Initially, both τ_A and τ_B execute during their allocated cycles. When τ_B completes at (t = 2,000), its allocated cycles are not needed until its next period (t = 5,000). This means the soft real-time task τ_C can use these empty cycles. τ_A, being a hard real-time task would be verified to meet all deadlines with only its allocated cycles and does not benefit by completing earlier, so τ_A's scheduling is unchanged. When both τ_A and τ_B complete at (t = 8,000), τ_C temporarily uses every cycle. This way, by only using their allocated cycles, the hard real-time tasks τ_A and τ_B are temporally isolated and can be verified independently. τ_C efficiently uses every cycle not needed by the hard real-time tasks but has sacrificed temporal isolation, since its timing behavior depends on when the hard real-time tasks start and end.

Each hardware thread, either an HRTT or SRTT, is always in one of two states: sleeping if it does not need to be scheduled until some later cycle or active otherwise. The hardware thread scheduler will only schedule active threads. If there were more than one SRTT in the above example, the thread scheduler would select and active SRTT in a round-robin order. FlexPRET's thread scheduler is designed to provide flexible scheduling options for both HRTTs and SRTTs while minimizing complexity.

4.1.4 Timing Instructions

The RISC-V ISA has been extended and given timing instructions for expressing real-time semantics. In contrast to PRET architectures supporting timing instructions, FlexPRET's design is targeted for mixed-criticality systems [17]. The first version of FlexPRET had a 64-bit register to store the number of nanoseconds since the processor was booted. However, to reduce complexity, the revised version of FlexPRET has a 32-bit internal clock that overflows about every 2^{32} or 4.29 human seconds. Software support is required for longer relative timing behavior [18]. The internal clock allows the timestamp to be read and tasks to be bounded by time constraints.

Expressing real-time semantics has been possible by extending the RISC-V base ISA with timing instructions. Here are some examples used by FlexPRET's clock [18].

- The `get_time` pseudo instruction reads the current time and allows for storing the value in a destination register.
- The `set_compare` pseudo instruction enables the compare register, which triggers the clock by putting upper and lower bounds on timed execution.
- A lower time-bound is provided by the `delay_until` and `wait_until` pseudo instructions, which both stall the execution until the compare value set by `set_compare` has expired.

4.2 Implementation

The following two sections discuss the deployment of FlexPRET as a soft-core on FPGA and the evaluation using a cycle-accurate simulator. The simulator is useful for prototyping, debugging, and benchmarking, while the FPGA deployment demonstrates the feasibility and provides analytical hardware costs.

4.2.1 FlexPRET Generation

The Chisel source files are located in a Git repository [19] on GitHub. GitHub is a web-based version-control and collaboration platform for software developers. Chisel allows parameterization of code, which helps produce different processor variations. This means FlexPRET can be configured in many ways by changing arguments in the 'config.mk' file :

- `THREADS=[1-8]` Specify the number of hardware threads
- `FLEXPRET=[true/false]` Use flexible thread scheduling
- `ISPM_KBYTES=[]` Size of instruction scratchpad memory (32-bit words)
- `DSPM_KBYTES=[]` Size of instruction scratchpad memory (32-bit words)
- `SUFFIX=[min,ex,ti,all]`
  - `min`: base RV32I
  - `ex`: `min` + exceptions (necessary)
  - `ti`: `ex` + timing instructions
  - `all`: `ti` + all exception causes and stats

These different configurations can also be deployed on FPGAs to evaluate the incremental cost of FlexPRET's hardware device properties. Verilog can be generated by running make FPGA, and a resulting Verilog file called "Core.v" will be created.

4.2.2 FlexPRET simulation

We have synthesized FlexPRET in Vivado by Xilinx [20], which provides support for the Zynq UltraScale+ family of devices [21]. The device we are synthesizing FlexPRET for is the Zynq UltraScale+ MPSoC ZCU102 evaluation board. First, Vivado analyzes the top-level module (generated Core.v file) for any syntax and semantic errors. Then, we describe the behavior of the circuit using input signals, output signals, and delays. The process is called simulation in Vivado, and it is used to verify the functionality of the circuit. The simulation also does a RTL analysis and returns an elaborated design file. This is a high-level representation of all the I/O banks and shows the ports connected to their respective modules in the Verilog file, along with all unassigned ports. The RTL file is a text file containing bits of code that represent real hardware structures. This file is read by Vivado and converted to a diagram of abstract generic technology cells.

Figure 4.3: Elaborated design read from RTL file (netlist of generic technology cells) (orange rectangles represent utilized cells)

4.2.3 FlexPRET synthesis

An elaborated design is necessary since the next step (high and low-level optimizations) is timing-driven and needs constraints. Constraints can not be applied directly to our RTL file (which is text); they need to be applied to a netlist. Constraints are a little human-labor intensive because writing them demands knowledge of not just the design but also the hardware being synthesized for. Constraints in Vivado are stored in '*.xdc' files [22]. Although the file is applied to the design at once, it can be considered a collection of individual tool command language (TCL) shell commands.

The first and the most important constraint we need to write is clock creation. We do this using the `create_clock` TCL command [23]. Here, we connect the clock signal from the top-level module to pin `CLK_74_25`. This pin is a fixed frequency onboard clock source with a frequency of 74.25 MHz (or period 13.5 nanoseconds) to the clock signal in the top-level module. The waveform function describes the duty cycle. The arguments 0-3.375 imply a clock 'HIGH' of 3.375 nanoseconds out of a period of 13.5 nanoseconds, meaning a 25% duty cycle.

```
create_clock -add -name CLK_74_25 -period 13.5 -waveform{0 3.375}
             [get_ports {clock}]
```

The next constraint is to set up input and output delays from the clock source to all the ports being used. This was done using the `set_input_delay` and `set_output_delay` TCL commands. The argument for max and min are the maximum and minimum delays and the argument for the `get_ports` function is the relative pin or port utilizing the clock. This needs to be written for every port or pin using the clock signal.

```
set_input_delay -clock CLK_74_25 -max 2.000 -min 1.000
                [get_ports {args}]
set_output_delay -clock CLK_74_25 -max 1.0 -min -0.1
                 [get_ports {args}]
```

The last notable constraint would be to define the I/O standards for every pin being used by our design. I/O standards define the voltage level our interface operates on and the kind of signaling it uses. The recommended standards for every pin can be found in the user guide for the ZCU102 board [24] and can be written to the constraints file using the `set_property` TCL command. Here, pin AE2 uses the LVCMOS18 standard [24] and is being connected to the `io_bus_data_out[0]` pin in our design.

```
set_property -dict {PACKAGE_PIN AE2 IOSTANDARD LVCMOS18}
             [get_ports {io_bus_data_out[0]}]
```

Once we have these constraints, Vivado will make high-level and low-level optimizations and produce a synthesized design. Synthesis is essentially converting RTL code to a netlist using our constraints. The synthesized design is an interconnected netlist of hierarchical and basic elements like flip-flops, block RAMs, I/O elements, etc.

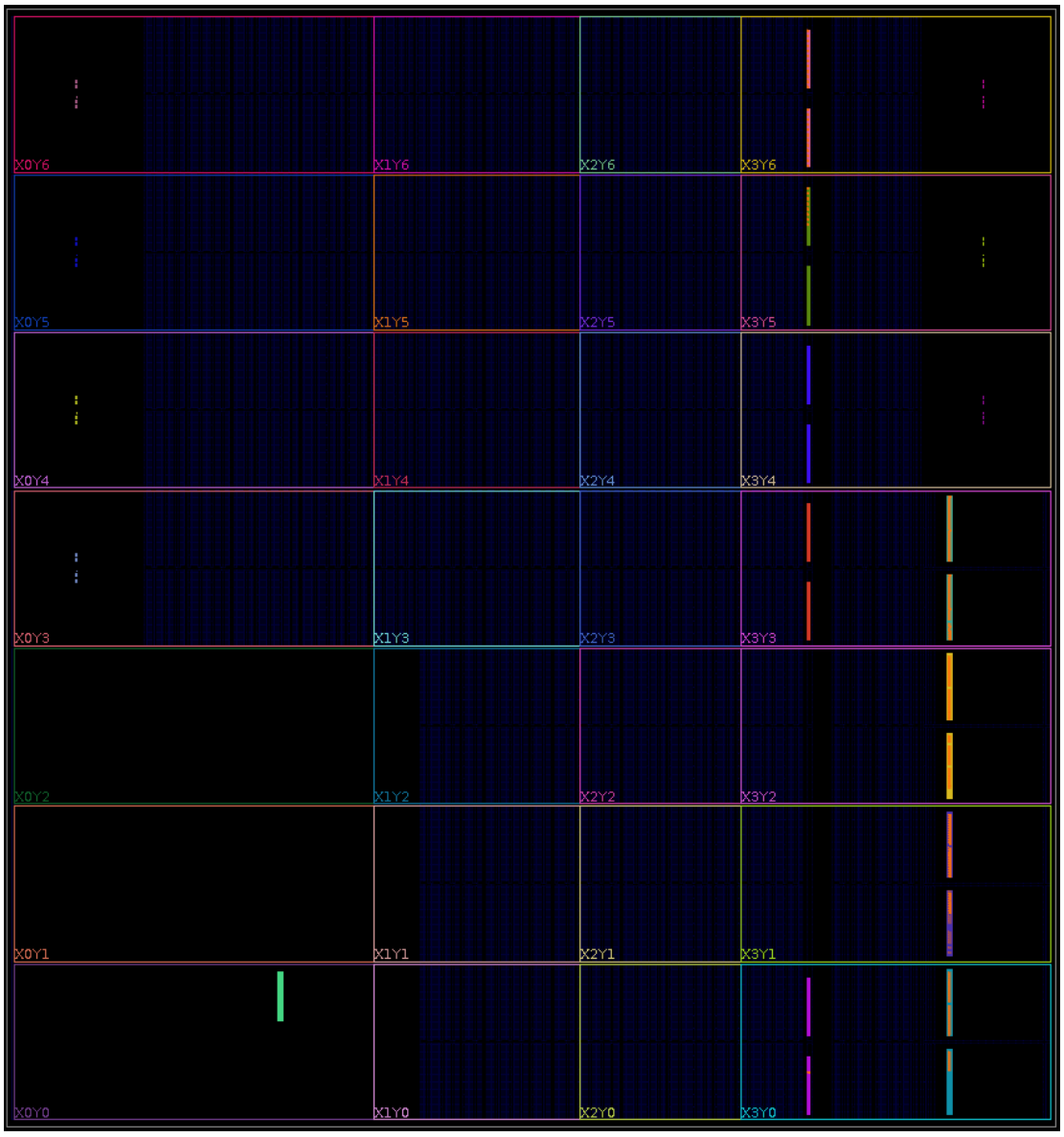

Figure 4.4: Synthesized design: each tile represents a collection of basic elements

4.2.4 FlexPRET Implementation

Vivado implementation includes all steps necessary to place and route the netlist onto the FPGA device resources while meeting the logical, physical, and timing constraints of a design. An implemented design is structurally similar to the synthesized design in the sense that cells have locations and nets are mapped to their specific routing channels, but different in the sense that the implemented design is optimized for hardware placing and routing.

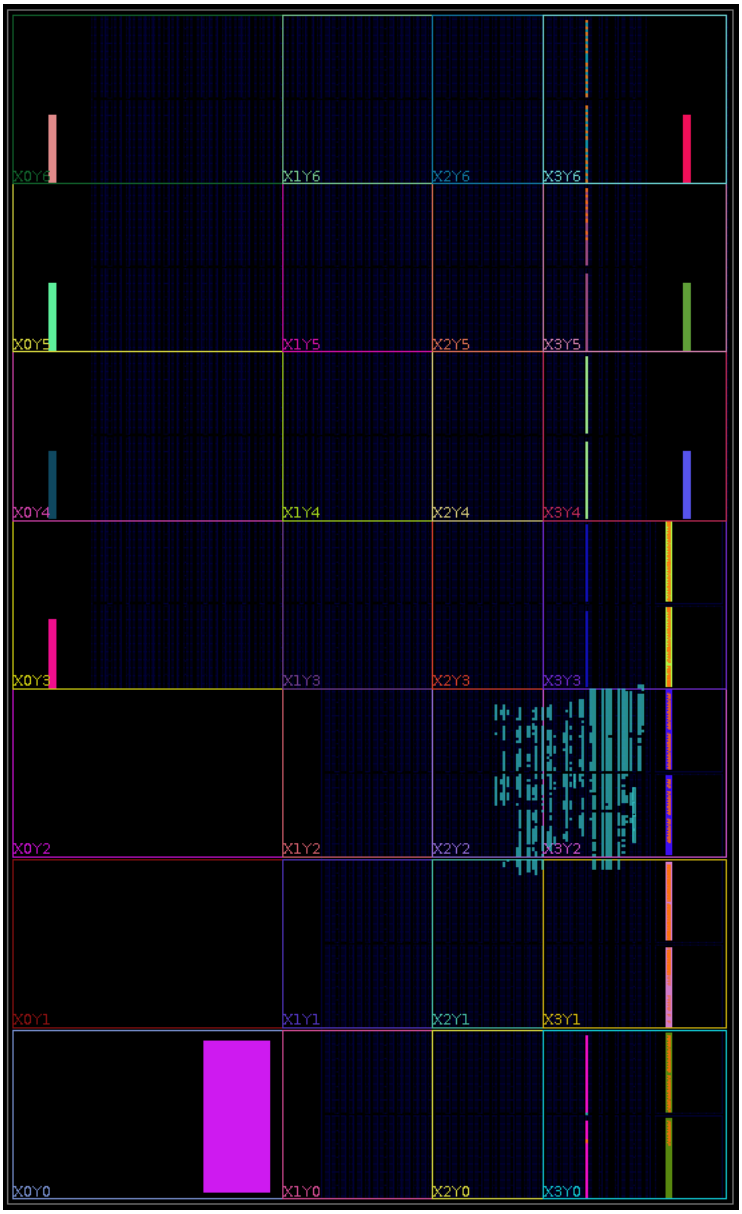

Figure 4.5: Implemented design: blue tiles represent utilized elements

4.2.5 FPGA deployment

An FPGA bitstream is a file that contains the programming information for an FPGA. Bitstream generation is the final step in deploying our design. At a high level, a bitstream file is similar to an executable program. It includes the description of the hardware logic, routing, and initial values of registers and on-chip memory. We generated the bitstream using the `write_bitstream` TCL command and then, used the Vivado hardware manager to upload the bitstream to our board.

4.3 Simulator

Chisel generates not only synthesizable Verilog code but also a cycle-accurate C++ based simulator for FlexPRET. The simulator also generates waveform files to aid debugging. This section discusses a simple hello world program for the simulator, and the next chapter will discuss executing our more complex benchmarks.

The RISC-V GNU toolchain consists of GCC, Binutils, newlib and gilbc ports. The GCC port is based on GCC 6.1.0 and receives commits frequently. A script for building the entire toolchain for RISC-V 32 is provided within the toolchain repository (located on GitHub [25]), and no issues were encountered during the build. Once the toolchain was built, it was possible to compile custom programs for the FlexPRET simulator.

The script file "compile.sh" [19] compiles a RISC-V C program with a start script and creates an objdump [26] [27]:

- `-march=rv32i`: generates code for the rv32i base integer variant of the ISA
- `-mabi=ilp32`: means that long and pointers are 32-bit wide.
- `riscv32-unknown-elf-objdump`: creates *.dump.text file for generated objdump

```
riscv32-unknown-elf-gcc -Iinclude -g -static -O1 -march=rv32i \
-mabi=ilp32 -nostartfiles -Wl,-Ttext=0x00000000 -o \
"$output_name" start.S "$@"

riscv32-unknown-elf-objdump -S -d "$output_name" > \
"$output_name.dump.txt"
```

The "parse_disasm.py" generates a hex file of the program for simulation. It parses the output of the riscv32-unknown-elf-objdump and puts it into a Scala array constant or readmemh hex file. This hex file can now be run as a simulation using the C++ emulator. Running the emulator creates a waveform file, useful for debugging and benchmarking.

Chapter 5

Benchnmarks

To investigate functionality and timing behavior, we executed C programs from the TACLeBench benchmark collection [28], which was created to support worst-case execution time (WCET) research [29].

5.1 TACLeBench

TACLeBench is a collection of open-source programs adapted to a standard coding style. It is available from GitHub. TACLeBench is a collection of 53 benchmark programs from several research groups and tool vendors around the world. The source codes are a hundred percent self-contained, with no dependencies to system-specific header files via `#include` directives or an operating system. All input data is part of the C source code, and potentially used functions from math libraries are also provided as C source code. This makes the TACLeBench collection useful for general embedded systems where no standard libraries are available. The latest version of FlexPRET's emulator has two header files with support for I/O ports and CSR (control and status register) instructions. We experimented heavily with combining the functions defined in these header files with those in the TACLeBench collection.

Since almost all benchmarks are processor-independent and can be compiled and evaluated for any kind of target processor, they are executable with RV-32I. They are compiled with the RISC-V gnu toolchain. All the benchmarks can be classified into:

- **Kernel benchmarks**: they implement small kernel functions, and the size of these benchmarks is in the range of 18 to 992 source lines of code (SLOC).
- **Sequential benchmarks**: they implement large function blocks, such as encoders and decoders, used in many embedded systems. The size of such benchmarks is in the range of 117 to 2710 SLOC.
- **Artificial test benchmarks**: they are used to stress test WCET analysis tools.
- **Application benchmarks**: they are derived from real applications and provided with a simulated input. For example, Lift is a lift controller that has been deployed in a factory in Turkey.

FlexPRET's microarchitectures can be divided into two interacting parts: the datapath and the control. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. FlexPRET uses a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

Figure. 5.1 shows the waveform of a simple binary search program generated by FlexPRET's emulator. `datapath_io_control_next_pc_sel[1:0]` represents the program counter selector for the first and only thread being utilized by the binary search program. `datapath_io_control_wb_rd_addr[4:0]` represents the destination register being written to a register file. `datapath_io_imem_rw_address` represents the addresses in the instruction memory being accessed every cycle.

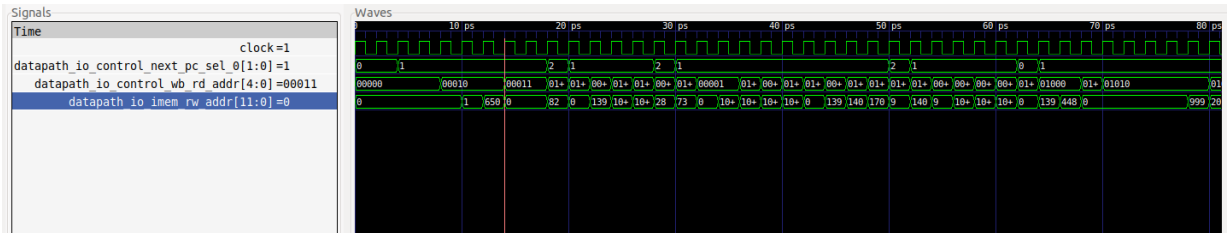

Figure 5.1: datapath_io_imem_rw_address represents addresses in instruction memory

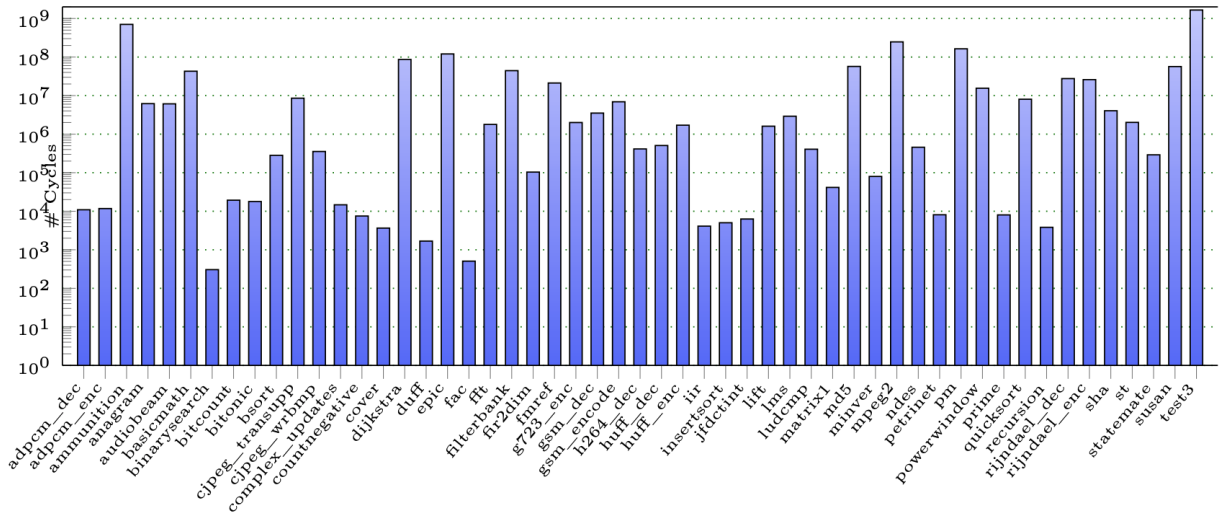

Figure 5.2: Sample execution times of programs in TACLeBench range from 305 cycles up to more than 1,600,000,000 cycles on the Patmos architecture

Figure 5.2 [29] shows the execution times of all 53 programs being executed on a Patmos Architecture [30]. On the Patmos architecture, the execution times range from 305 cycles (binarySearch) up to 1,658,333,567 cycles (test3). To put this into relation, the benchmark program test3 runs approximately for 21 seconds on the Patmos platform assuming a CPU Frequency of 80 Mhz. From this evaluation we make the main observation that TACLeBench consists of both short and long running benchmarks with a huge variety in execution time.

Chapter 6

Conclusion

6.1 Conclusion

Cyber-physical and real-time embedded applications require high confidence in average-case performance and timing predictability. Integrated hardware platforms share resources to support the increasing functional complexity of applications, but interference and different levels of criticality complicate design and verification. The common approach of running a real-time operating system (RTOS) on processors optimized for average-case performance results in unpredictable behavior—mainly caused by interrupts and hardware prediction mechanisms—that is difficult to verify and certify. FlexPRET's architectures provides high confidence in software functionality and timing behavior without sacrificing overall processor throughput. It uses fine-grained multithreading to enable trade-offs between predictability, hardware-based isolation. By supporting the specification, repeatability, and predictability of timing behavior, FlexPRET includes time in the abstraction level between software and hardware, allowing compilers and analysis tools to optimize and guarantee timing behavior.

6.2 Future Work

Since FlexPRET only works with the RV32I [11] base instruction set, real-world applications might not be realizable. However, promising future applications include investigation of the RISC-V extensions M (Standard Extension for Integer Multiplication and Division), F (Standard Extension for Single-Precision Floating-Point), P (Standard Extension for Packed-SIMD Instructions).

Deeper investigations regarding thread synchronization with extension A (Standard Extension for Atomic Instructions) are also required.

Bibliography

[1] Lee, E. A. Cyber Physical Systems: Design Challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 363–369, doi:10.1109/ISORC.2008.25.

[2] *DO-178C: Software Considerations in Airborne Systems and Equipment Certification.* RTCA Std., 2012.

[3] Liu, I.; Reineke, J.; Broman, D.; et al. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, 2012, pp. 87–93, doi:10.1109/ICCD.2012.6378622.

[4] Chisel/FIRRTL Developers. Chisel: Home. `https://www.chisel-lang.org`, online; accessed frequently.

[5] LLVM Developer Group. clang - the CLang C, C++, and Objective-C compiler. `https://clang.llvm.org/docs/CommandGuide/clang.html`, online; accessed frequently.

[6] Chisel/FIRRTL Developers. Chisel: FIRRTL. `https://www.chisel-lang.org/firrtl/`, online; accessed frequently.

[7] Berkeley Architecture Research. Rocket-Chip Generator. `https://chipyard.readthedocs.io/en/latest/Generators/Rocket-Chip.html`, online; accessed 29 January 2021.

[8] CHIPS Alliance. Rocket Chip Generator. `https://github.com/chipsalliance/rocket-chip`, online; accessed 29 January 2021.

[9] Hunter, J. D. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, volume 9, no. 3, 2007: pp. 90–95, doi:10.1109/MCSE.2007.55.

[10] Pat Hanrahan. Magma: a hardware design language embedded in python. `https://github.com/phanrahan/magma`, online; accessed 29 January 2021.

[11] Waterman, A.; Lee, Y.; Patterson, D. A.; et al. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technical report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014. Available from: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html`

[12] Edwards, S. A.; Lee, E. A. The Case for the Precision Timed (PRET) Machine. In *2007 44th ACM/IEEE Design Automation Conference*, 2007, pp. 264–265.

[13] Avissar, O.; Barua, R.; Stewart, D.; et al. Heterogeneous Memory Management for Embedded Systems. 12 2001, doi:10.1145/502217.502223.

[14] Banakar, R.; Steinke, S.; Lee, B.-S.; et al. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, 2002, pp. 73–78, doi:10.1145/774789.774805.

[15] Broman, D.; Zimmer, M.; Kim, Y.; et al. Precision timed infrastructure: Design challenges. In *Proceedings of the 2013 Electronic System Level Synthesis Conference (ESLsyn)*, 2013, pp. 1–6.

[16] Zimmer, M.; Broman, D.; Shaver, C.; et al. FlexPRET: A processor platform for mixed-criticality systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 101–110, doi:10.1109/RTAS.2014.6925994.

[17] Liu, I. *Precision Timed Machines*. Dissertation thesis, EECS Department, University of California, Berkeley, May 2012. Available from: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-113.html`

[18] Zimmer, M. *Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O*. Dissertation thesis, EECS Department, University of California, Berkeley, Aug 2015. Available from: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-181.html`

[19] University of California in Berkeley. FlexPRET. `https://github.com/pretis/flexpret`, online; accessed frequently.

[20] Xilinx. Vivado Software Suite. `https://www.xilinx.com/products/design-tools/vivado.html`, online; accessed frequently.

[21] Xilinx. Zynq UltraScale+ MPSoC. `https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html`, online; accessed frequently.

[22] *Vivado Design Suite User Guide: Using Constraints*. Xilinx, 2018.

[23] *Vivado Design Suite Tcl Command Reference Guide*. Xilinx, 2019.

[24] *ZCU102 Evaluation Board: User Guide*. Xilinx, 2019.

[25] RISC-V. RISC-V GNU Compiler Toolchain. `https://github.com/riscv/riscv-gnu-toolchain`, online; accessed frequently.

[26] GCC, the GNU Compiler Collection. riscv32-unknown-elf-gcc Manual. `https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html`, online; accessed frequently.

[27] Linux. objdump Manual. `https://linux.die.net/man/1/objdump`, online; accessed frequently.

[28] TACLe. TACLeBench collection of benchmarks. `https://github.com/tacle/tacle-bench`, online; accessed frequently.

[29] Falk, H.; Altmeyer, S.; Hellinckx, P.; et al. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, *OpenAccess Series in Informatics (OASIcs)*, volume 55, edited by M. Schoeberl, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.

[30] Schoeberl, M.; Abbaspour, S.; Akesson, B.; et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, volume 61, no. 9, 2015: pp. 449–471, ISSN 1383-7621, doi:https://doi.org/10.1016/j.sysarc.2015.04.002. Available from: `https://www.sciencedirect.com/science/article/pii/S1383762115000193`

Appendix A: FPGA Specifications and Resource Utilization

The ZCU102 is a general purpose evaluation board for rapid-prototyping based on the Zynq® UltraScale+™ XCZU9EG-2FFVB1156E MPSoC (multiprocessor system-on-chip). High speed DDR4 SODIMM and component memory interfaces, FMC expansion ports, multi-gigabit per second serial transceivers, and a variety of peripheral interfaces provide a very flexible prototyping platform. Of course the scope of this thesis has been limited to the FPGA logic.

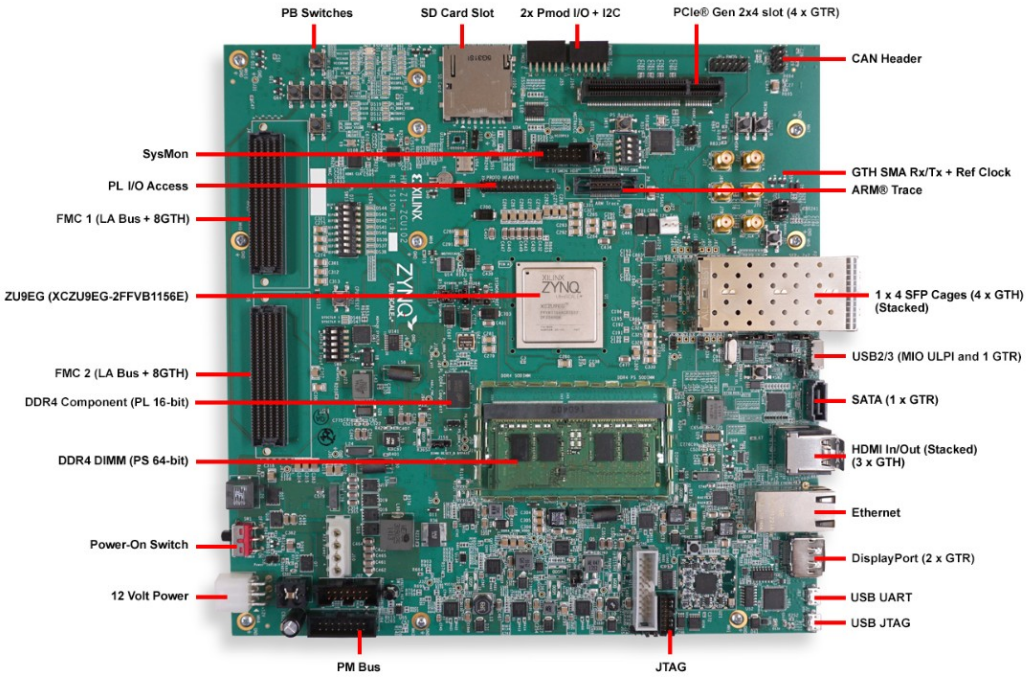

Figure 1: ZCU102 Evaluation Board

Onboard resources utilized by FlexPRET (post-implementation):

Resource	**Utilization**	**Available**	**Utilization%**
LUT	5966	274080	2.1767
FF	1711	548160	0.3121
LUTRAM	3200	144000	2.2222
BRAM	5	912	0.5482
IO	191	328	58.232
BUFG	1	404	0.2475

Table 1: FlexPRET Resource Utilization

Appendix B: Terminal Commands

Generating FlexPRET from the Chisel source files

```
sudo apt-get install git
git clone https://github.com/pretis/flexpret.git
```
(To install Git and download the FlexPRET repository)

```
cd flexpret-master
make fpga
```
(To build a default configuration and generate Verilog)

Building the RISC-V GNU toolchain

```
mkdir /opt/riscv
export PATH="/opt/riscv/bin:$PATH"
git clone https://github.com/riscv/riscv-gnu-toolchain.git
cd riscv-gnu-toolchain
mkdir build
cd build
../configure --prefix=/opt/riscv --enable-multilib
sudo make
```
(To build either cross-compiler with support for both 32-bit and 64-bit. The multilib compiler will have the prefix riscv64-unknown-elf- or riscv64-unknown-linux-gnu-, but will be able to target both 32-bit and 64-bit systems.)

Compiling and simulating custom programs

```
cd flexpret-emulator make emulator
```
(To build the simulator)

```
cd programs
git clone https://github.com/tacle/tacle-bench.git
./compile.sh binarySearch /tacle-bench/bench/kernel/binarysearch/binarysearch.c
```
(To install TACLeBench and compile a binary search program using the riscv32-unknown-elf-gcc)

```
../scripts/parse_disasm.py binarySearch.dump.txt readmemh > imem.hex.txt
```
(To generate hex file of the program for simulation)

```
../emulator/flexpret-emulator
```
(To run the simulation and generate a waveform file from imem.hex.txt)

```
cd programs
sudo apt-get install -y gtkwave
gtkwave Core.vcd
```
(To read the waveform file Core.vcd generated by the emulator with gtkwave)

Appendix C: List of abbreviations

Abbreviation	Meaning
RTOS	Real-Time Operating System
FPGA	Field Programmable Gate Array
PLD	Programmable Logic Device
CLB	Configurable Logic Blocks
LUT	Look-up Table
RAM	Random Access Memory
I/O	Input/Output
HDL	Hardware Description Language
VHDL	Very-High-Speed-Integrated-Circuit HDL
DSL	Domain-Specific Language
MUX	Multiplexer
FIRRTL	Flexible Intermediate Representation for Register Transfer Level
FIR	Flexible Intermediate Representation
ISA	Instruction Set Architecture
RISC	Reduced Instruction Set Computer
IP	Intellectual Property
PRET	Precision Timed
ISPM	Instruction Scratchpad Memory
ALU	Arithmetic Logic Unit
DSPM	Data Scratchpad Memory
HRTT	Hard Real-Time Threads
SRTT	Soft Real-Time Threads
RTL	Register Transfer Level
TCL	Tool Command Language
WCET	Worst Case Execution Time
CSR	Control and Status Register
SLOC	Source Lines of Code

Table 2: List of Abbreviations