Investigation of using a PRET processor on a low-cost, low-power FPGA

SAM ZAMANI
Investigation of using a PRET processor on a low-cost, low-power FPGA

Sam Zamani
Abstract

Mixed-criticality is a current trend in real-time embedded systems, where software tasks are integrated onto fewer hardware platforms. The basic idea is to use one processor to execute multiple tasks with differing requirements of certification, importance or safety. In systems where time is an important key factor the behavior of that system must be predictable at all times, which is hard to achieve when optimizations made to achieve good performance lower predictability at the same time. In 2007 Stephen A. Edwards and Edward A. Lee made a case for the precision-timed (PRET) machine as a solution, arguing that temporal behavior is to be treated equal to functional behavior. One of those PRET machines is FlexPRET, which is the processor we are studying in this thesis.

This thesis aims to study the generation, synthesis and programming of FlexPRET, a fine-grained multithreaded RISC-V based PRET processor developed at UC Berkeley. This is part of a larger-scale project to port FlexPRET to be used as a node processor in a NoC mesh generated by ForSyDe, a programming methodology developed at the Royal Institute of Technology. Previous synthesis of FlexPRET has been done on FPGAs from the Virtex-5 and Spartan-6 families by Xilinx. The FPGA used for this thesis is from the Cyclone IV family by Altera.

Evaluation of the synthesis results have been performed by running a real-time application on FlexPRET that blinks LEDs on the Altera DE2-115 board.
I would like to thank my family for pushing me and standing by me through this whole journey. Special thanks to my mother for all the financial support and for believing in me. Thanks to my friends Robi Al-Aa and Karl Weisser for pushing me hard to achieve results. Thanks to Michael Zimmer of UC Berkeley for his help with technical details regarding FlexPRET generation and RISC-V, and David Broman of Royal Institute of Technology for helping me get in touch with Michael Zimmer. Thanks to my supervisor and examiner Johnny Öberg for his great advice, guidance and support throughout the project.

Sam Zamani

Stockholm, June 2016
**Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABI</td>
<td>Application Binary Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSR</td>
<td>Control and Status Register</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>ELF</td>
<td>Executable and Linkable Format</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose Input Output</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>PRET</td>
<td>Precision-Timed</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>SBT</td>
<td>Simple Build Tool</td>
</tr>
<tr>
<td>SCALA</td>
<td>SCAlable Language</td>
</tr>
<tr>
<td>SUDO</td>
<td>Substitute User DO</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread Level Parallelism</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

ABSTRACT .................................................................................................................................................. 2
PREFACE .................................................................................................................................................... 3
NOMENCLATURE ....................................................................................................................................... 5
TABLE OF CONTENTS ............................................................................................................................. 6
1 INTRODUCTION ......................................................................................................................................... 8
  1.1 Mixed-Criticality Systems .................................................................................................................... 8
  1.2 Predictability Issues in Mixed-Criticality Systems .............................................................................. 9
  1.3 Precision Timed (PRET) Machines ....................................................................................................... 9
  1.4 Project Objectives and Thesis Overview ........................................................................................... 10
2 BACKGROUND ......................................................................................................................................... 11
  2.1 The principle of PRET .......................................................................................................................... 11
  2.2 The FlexPRET Processor ..................................................................................................................... 12
  2.3 The RISC-V ISA .................................................................................................................................. 14
  2.4 Scala and Chisel .................................................................................................................................... 16
3 IMPLEMENTATION .................................................................................................................................... 17
  3.1 The Generation of FlexPRET ............................................................................................................... 17
  3.2 The Synthesis of FlexPRET ................................................................................................................ 19
  3.3 Programming FlexPRET .................................................................................................................... 21
4 RESULTS ................................................................................................................................................ 29
  4.1 Synthesis ............................................................................................................................................. 29
  4.2 Programming ....................................................................................................................................... 32
5 DISCUSSION AND CONCLUSIONS ...................................................................................................... 37
  5.1 Discussion ........................................................................................................................................... 37
This thesis focuses on the generation, synthesis and programming of a precision-timed (PRET) processor named FlExPRET on an Altera FPGA, as part of a larger-scale project to port that processor for use on a NoC mesh generated by the ForSyDe tool suite. In this chapter we discuss the concepts of Mixed-Criticality Systems, predictability issues, and PRET machines. We will also introduce the area and the problems dealt with in this thesis.

1.1 Mixed-Criticality Systems

A current trend in real-time embedded systems is integrating many software tasks onto fewer hardware platforms. This trend is driven by demands in automotive and avionic domains to reduce cost, size, weight and power of the embedded hardware, while at the same time system complexity keeps growing [1][16][17][18][19][20]. The idea is that a single processor must run tasks that have two or more distinct levels of requirements regarding certification, importance or safety, often defined as criticality levels. Such systems are known as mixed-criticality systems, where criticality designates the required assurance level against failure required for a system component or task, i.e. what measures must be taken in design and verification to avoid failure in meeting the requirements. The number of criticality levels and how they are defined may vary, but in order to classify as a mixed-criticality system the number of levels should be at least two; one critical level and one non-critical level. Standard frameworks for safety within avionics and automotive often define more than two criticality levels; one example is the DO-178C standard which describes five safety integrity levels for avionics [14]. Criticality levels are graded from the lowest to the highest, where applications having the highest criticality level have the highest cost in both design and verification, and are also most prone to failure. Flight control systems in aircrafts are an example of systems whose failure will cause a plane crash. In railroad rolling stock, e.g. trains and trams, if the brake system fails there will be a crash or derailing. The same goes for automobiles and autobuses.

In a mixed-criticality system a single processor must run tasks with different levels of criticality, some of who may be critical, others non-critical, and the rest may grade between those two extremities. For critical tasks that grow in complexity, design and verification require temporal and spatial isolation of those tasks, as they must not be interfered with by other tasks. Isolation can be achieved either in software with a Real-Time Operating System (RTOS) or in hardware with multi-core processors or multi-threaded processors. RTOSes do not require much hardware, but do require verification and certification themselves. Hardware-based isolation is achieved by executing each task on a separate core or thread, which can be inefficient in the former case for tasks with low processor utilization. Hardware threads, the latter case, are executed on a single pipeline with one task per thread, and can be scheduled according to their needs. This can be more efficient than running on multi-core systems [1][16]. One major need rises from the importance of not missing deadlines, i.e. finish time is greater than what is allowed for the task. The hardware interacts with components that require fast response times to operate properly, and must thus provide support for the execution of tasks with varying temporal behavior and requirements [1][16]. Tasks are categorized based on their consequences of deadline misses; a task is called hard real-time if a deadline miss causes total system failure with disastrous effects as a result, firm real-time if deadline misses cause the system’s quality of service to deteriorate due to zero usefulness of the results, and soft real-time if deadline misses cause the system’s quality of service to deteriorate due to deteriorating usefulness of the results. Consequently, hard real-time tasks must always be guaranteed to meet their deadlines, whereas firm and soft real-
time tasks should at least meet a subset of their deadlines to maintain acceptable system behavior.

One important aspect of a real-time system is to ensure predictability of its system behavior at all times, which is the requirement for hard real-time tasks. This is something that has grown very difficult on modern processors due to their growing complexity. Architectural optimizations and improvements have resulted in processors with high performance, but it has also become almost impossible to accurately predict the execution time of a simple computer program.

1.2 Predictability Issues in High Performance Systems

Achieving high performance on an average case in modern processors has almost always been the main goal of every optimization and improvement made to computer architecture since the 1980s. Cache hierarchies were introduced to reduce the time for processors to access main memory in order to load and store data, and branch predictors were introduced to minimize the number of branch hazards in pipelines. Caches and branch predictors are also, unfortunately, major sources of unpredictability. Branch predictors, although they are found to be quite accurate, do not always make the correct predictions. Incorrect predictions may cause the wrong instructions to be fetched into the pipeline. The correct data or instructions may in addition not be found in the cache, which means that thousands of clock cycles will be wasted on miss penalties. If the problems with branch predictors and caches are combined (i.e. the wrongly fetched instruction was not found in the cache) and that happens a lot, the running task may have problems meeting its deadline as a result. Deadline misses are incredibly unforgiving in a highly time critical system, because the effects are disastrous.

Yet, worst-case execution time (WCET) analysis is the common measurement standard for computer systems when we compare them to one another, and when we develop real-time software for real-time systems. It is indeed a nontrivial task to perform thanks to unpredictability. The proposed solution for that would have to be an entire rethinking of processor architecture [3]. The idea is to treat timing behavior equal to function.

1.3 Precision-Timed Machines

As raising average-case performance has been the main focus of every computer architect, predictability is believed to have been neglected and forgotten about. Therefore a complete rethinking of architecture has been necessary. In 2007, Stephen A. Edwards and Edward A. Lee made a case for the precision-timed (PRET) system as a solution. They argued that temporal behavior should be treated as a first class citizen, equal to functional behavior [2]. One obvious way to achieve predictability would be to reintroduce architectures of the 1980s, as they were known to be very predictable, but then performance would have to be sacrificed. With PRET machines a new era is born where predictability and performance coexist.

An integral part of the PRET solution is to employ as many register sets (known as hardware threads) as there are pipeline stages, although one thread less than the number of pipeline stages is sufficient. The pipeline must furthermore be thread-interleaved, i.e. it must fetch instructions from different threads frequently. A fine-grained thread-interleaved pipeline fetches instructions from different threads every cycle. Coarse-grained thread-interleaved pipelines on the other hand are event triggered, i.e. thread switches occur less frequently. The PRET solution uses the former case because it eliminates all hazards that give rise to unpredictability. Data and instructions are stored in scratchpad memories, which are on-chip memories like caches, but have distinct address space from main memory, so that they hold their own data instead of copies of the main memory contents. Scratchpad memories also lack hardware controllers for content management,
which is instead done through software. To further ensure predictability and to implement semantics for time control, the ISA is extended with timing instructions for reading time from an internal register and put lower and upper time bounds on program flow.

The PRET system we are studying in this thesis is FlexPRET, a fine-grained thread-interleaved RISC-V based processor for mixed-criticality systems, developed at UC Berkeley [1]. FlexPRET has a RISC based fine-grained thread-interleaved pipeline and a thread scheduler that implements flexible scheduling to support mixed-criticality systems. It also implements an extended RISC-V ISA with instructions to handle timing semantics.

1.4 Project Objectives and Thesis Overview

This thesis aims to study the generation, synthesis and programming of FlexPRET, as a part of a larger-scale project to make a port to the ForSyDe programming methodology developed at the Royal Institute of Technology [21][22]. The idea is to use FlexPRET as a node processor in a NoC mesh generated by the NoC System Generator of the ForSyDe tool suite [23][24][25].

FlexPRET has previously been synthesized for FPGAs of the Virtex-5 and Spartan-6 families from Xilinx [1][4]. The goal of this project is to port FlexPRET for a FPGA device from Altera. The target FPGA is a Cyclone IV family device and our results will be evaluated by executing software applications on it that set and clear GPIO bits.

The objectives for the project are the following:

I. Study the FlexPRET processor. Document how it works.

II. Generate and synthesize FlexPRET from customizable open-source code to be deployed on an Altera FPGA. Document how FlexPRET is specified and how synthesis should be carried out for the target FPGA.

III. Evaluate the synthesis results by executing a simple code application on FlexPRET.

The first chapter of this thesis describes concept of mixed-criticality systems, as well as stating the problems with predictability and a solution to that. Finally the goal and objectives of the thesis project are given. The second chapter of this thesis goes deeper into the concepts of PRET machines, the FlexPRET processor and the RISC-V Instruction Set Architecture (ISA) with its timing semantics extensions. The third chapter of this thesis describes the implementation of the project, everything from the generation of FlexPRET to its synthesis, and working with the programming environment. Finally the fourth, fifth and sixth chapters of this thesis describe results, discussions, conclusions and proposals of future work.
To understand the FlexPRET processor, this chapter begins with explaining the underlying principle of a PRET system and the architectural solutions that form the PRET infrastructure, which is basically a set of architectural changes that bypass the sources of unpredictability without sacrificing too much performance, and that extend the ISA with semantics to control time. We also look at the FlexPRET processor’s architecture and the RISC-V ISA. The final section describes Scala and Chisel, a hardware description language which generates synthesizable Verilog code.

2.1 The Principle of PRET

It is a great challenge to raise both performance and predictability, as one is always raised at the cost of the other. The basic idea behind PRET machines is unpredictability rising from modern architectures, because they only focus on achieving high average performance. Branch predictors and caches that generally raise performance unfortunately also tend to raise unpredictability, which may have disastrous effects [3][5]. The PRET principle is therefore to treat temporal behavior equal to functionality in order to achieve good predictability. If one is willing to forgo performance then predictability is not very hard to achieve, as microprocessors of the 1980s were known to be highly predictable [2][6]. Thus, the true challenge with PRET machines is to achieve both high performance and high predictability. In this section we discuss the principles of the PRET machine and compare them to those of ordinary processors.

Hardware threads

Thread-Level Parallelism (TLP) is a technique that can be used with great favor in parallel computer systems. It can be exploited in multicore processors through software, which will utilize entire cores to share the workload of programs between them for example. Each thread state has its own portion of memory where the state of the thread is stored upon context switch. Multithreaded processors have hardware support for TLP through hardware threads. Each hardware thread has its own set of registers to save their state, and the processor switches between threads through thread interleaving, i.e. fetching instructions from different hardware threads into the pipeline.

Scratchpad memories

While caches are a great way to reduce latency in transfers between main memory and CPU, they are a major source of unpredictability. However, just removing the caches would not do any good, as performance would drop very fast back to the standards of the 1980s. Instead we use scratchpad memories to replace the caches. Scratchpad memories can be thought of like L1 caches in the sense that they are close to the arithmetic logic unit, and thus work as fast access memories in the same way. But they do not have hardware controllers for checking memory content. Instead, scratchpad memories are thought of as a distinct part of the memory system with distinct memory addresses, so that we access the scratchpad memory when we access a scratchpad memory address, and similarly we access main memory when we access a main memory address [6]. Scratchpad memory contents are managed through software and use DMA-based data transfers between scratchpad and main memory. Unpredictability is avoided, but it comes with the price of slight overhead additions due to the extra code needed, i.e. instructions that explicitly manage the contents.
Thread-interleaved pipelines

One way to get around the branch predictor issue is to use a thread-interleaved pipeline instead of an ordinary deep pipeline, and employ as many hardware threads as there are pipeline stages. In a thread-interleaved pipeline each stage can have an instruction from different hardware threads, which will eliminate all hazards if the scheduler implements a fine-grained policy, as the pipeline will be scheduled to fetch instructions from different threads every single cycle. As a result branches are always resolved before the next instruction in that thread is fetched into the pipeline, making sure that the correct instruction is always fetched. Dependent instructions never have to stall the pipeline either. Thus, the pipeline is full at all times. If the scheduler implements a coarse-grained policy, however, hardware thread scheduling would basically work similarly to software thread scheduling, i.e. a thread may run in the pipeline until a thread switch is triggered by some event. Coarse-grained multithreading won’t slow down the execution of threads thanks to less thread switches, but it won’t eliminate hazards that stall the pipeline either.

Instruction Set Architectures with timing instructions

To be able to make sure that predictability and treat time as a first class citizen, a major part of the PRET system infrastructure is the ability to control temporal behavior. The idea is to implement time control within the Instruction Set Architecture (ISA), as it provides instructions that are tied to specific parts of the hardware based on their semantics. Most modern day ISAs, unfortunately, do not provide any means to control time explicitly; time control can only be done indirectly through software and existing hardware [5]. However, it is possible to extend ISAs to think outside its boundaries, i.e. add new semantics associated with some new hardware. Previous work known as PTARM, assumed a designated platform clock along with instructions that used it to read time and provide upper and lower bound time slots to tasks [6].

2.2 The FlexPRET Processor

FlexPRET is a fine-grained multithreaded processor developed at UC Berkeley. It is based on the RISC-V ISA and uses a thread-interleaved RISC pipeline with 5 stages. It is extended with timing instructions that use a designated platform clock, like what was implemented with PTARM [6]. It is made to support mixed-criticality systems by a scheduler that classifies threads as either hard real-time or soft real-time. Hardware-based isolation is provided for hard real-time threads through scheduling guarantees and slicing of the memory, meanwhile soft real-time tasks can execute on vacant cycles to provide efficient processor utilization.

Figure 2.1 – Diagram of FlexPRET’s datapath
(Picture from http://github.com/pretis/flexpret/tree/riscv-2.0)
Scratchpad Memories

FlexPRET's local memory system uses two scratchpad memories; one for storing instructions (I-SPM) and one for storing data (D-SPM). The I-SPM has a read port for fetching instructions and a read/write port connected to the load-store unit and also externally exposed to allow DMA operations, i.e. the read/write port is connected to the peripheral bus. The D-SPM has two read/write ports, one connected to the load-store unit and the other externally exposed. Since scratchpads are independent memories with distinct address space, the peripheral bus, I-SPM and D-SPM are assigned different segments of a program using addresses, and the load-store unit decides using those addresses which memory to access [16]. The scratchpads are divided into 8 equally sized slices that can be assigned to threads and therefore protected by writing to dedicated control and status registers (CSR). If protected, the slices can only be accessed by their corresponding threads, whereas if they are shared they can be accessed by all threads. The scratchpads are useful for reducing WCET and ensuring predictability, although they are not required if the loss in predictability with caches is acceptable [1].

Pipeline

The FlexPRET pipeline is based on the RISC pipeline which consists of an instruction fetch stage (F), an instruction decode stage (D), an execution stage (E), a memory stage (M) and a write back stage (W), as can be seen in figure 2.1. Since the pipeline is arbitrarily thread-interleaved it can hold one instruction from up to 5 threads at the same time, and doesn’t have any restriction to thread scheduling. The minimum number of threads employed for predictability would be 5. However, let’s consider two cases:

1. A branch instruction is fetched from a thread that was scheduled to run. The branch is resolved in the E-stage at the latest, given that it is a conditional branch. Until that has happened, the thread should not be scheduled again, which is after 3 cycles.

2. Another thread has a load instruction and an immediately following data dependent instruction. Since the dependent instruction has to wait for the load to finish, it cannot be fetched until 2 cycles after, given that forwarding will present data to the dependent instruction using the ALU.

Based on both cases FlexPRET could run on at least 3 threads, but since it supports arbitrary scheduling of threads, any number of threads above one would be sufficient. If one were to keep the round-robin fashion, then 4 threads is the minimum. PTARM doesn’t support arbitrary thread-interleaving and thus always needs spacing between instructions, as it uses round-robin scheduling [6]. FlexPRET on the other hand schedules threads based on their scheduling frequencies, i.e. per how many clock cycles that thread must execute exactly once [1].

Thread Scheduler

The thread scheduler uses constant scheduling frequencies to schedule. While this is not an absolute requirement for soft real-time threads, the isolation and timing predictability of every hard real-time thread strongly depends on it. A hard real-time thread must always finish in time; therefore it has to have priority over the soft real-time threads, which will then be scheduled to use vacant clock cycles in a round-robin fashion. This is implemented by allowing threads to be active; and thus allowed to be scheduled. Or they can be sleeping; and thus not allowed to be scheduled. Since threads also can be either hard real-time or soft real-time, there are actually 4 modes that a thread can be in. This information is stored in a thread mode control register that is used by the thread scheduler. Threads can also be disabled, and active hard real-time threads need to be guaranteed cycles, therefore the thread mode control register alone doesn’t suffice. There is also need for a slot control register, a register that is used to prescribe cycles to certain
threads [1]. The register has 8 4-bit slots that can be disabled, given to soft real-time threads’ round-robin scheduling, or dedicated to one of FlexPRET’s threads given by its ID. Depending on values of both registers the thread scheduler can assign cycles to threads that are active using non-disabled slots. A slot that is dedicated to a thread in active hard real-time thread mode always guarantees that the thread is given cycles. If that thread however is in sleeping hard real-time thread mode, the slot dedicated to that thread can be used by a soft real-time thread instead. Another way to give a cycle to a soft real-time thread is to assign a slot the soft real-time thread value. Setting a thread into any hard real-time mode, but not assigning any cycles to it in the slot control register, renders it disabled. Both the thread mode and the slot registers are CSRs, which is an integral part of the privileged architecture.

Timing Semantics

In order to support timing semantics, the RISC-V ISA is extended and given timing instructions. These timing instructions read the value of a special register known as the internal clock, which represents elapsed time in nanoseconds. The first version of FlexPRET had a 64-bit register which would overflow every 584 years according to the representation. The revised version of FlexPRET has a 32-bit internal clock which overflows about every 4.29 seconds instead of every 584 years for reduced complexity reasons. Software support is required for longer relative timing behavior [16]. The internal clock allows the timestamp to be read and tasks to be bounded by time constraints. Instructions that perform these kinds of operations can be found in older PRET machines as well, like PTARM [6] for instance, what differentiates FlexPRET from PTARM is that the former is especially designed for mixed-criticality purposes and the latter is not. Thus threads that are delayed until a certain time do not waste any clock cycles, instead they yield their slots to other soft real-time threads. Timing instructions will be covered more in detail in section 2.3.

2.3 The RISC-V ISA

The ISA implements semantics for all functions in a processor, everything from the basic integer operations to exception handling. What it does is provide a link bridge between the programmer and the machine code. FlexPRET is based on RISC-V, which is an open source ISA that was originally made for research and education in computer architecture, with the extensive ambition to bring RISC-V to industry and development, to make it a standard ISA for commercial developers [7]. RISC-V is as its name suggests the fifth major RISC-based ISA developed at UC Berkeley, with the capital V, besides representing 5 in the Roman numeral system, signifying the support for varieties of research [7]. Being an ISA originally designed to support varieties of research RISC-V is highly customizable. All implementations must include a base integer ISA, beyond of which extensions and customizations are allowed. RISC-V supports two different base integer ISAs, they can be either 32-bit (RV32I) or 64-bit (RV64I). The base integer ISAs denote the size of an integer register, i.e. the 32-bit base integer ISA uses 32-bit integer registers and the 64-bit base integer ISA uses 64-bit integer registers. FlexPRET uses the 32-bit architecture and holds one set of 32 32-bit registers for each thread, so it can have up to 256 32-bit registers, as it can be configured to have up to 8 threads.

The Base Integer ISA Instructions

The base integer ISA has 4 instruction type formats; I, R, S and U-type. The I-type format specifies a source register for its first operand and a destination register for the result, the second operand is a 12-bit immediate value encoded in the instruction. The R-type format is a pure register format, both operands come from source registers and the result is stored in the
destination register. The S-type format specifies two source registers and an immediate value, which adds to the base source register to create an effective address for stores. The U-type format has a 20-bit immediate value which is shifted 12 bits left and added to the upper 20 bits of either the destination register or the program counter. There are further two variants of the S and U-types, namely SB and UJ. The SB-type is an S-type instruction format with an immediate to encode conditional branch offsets in multiples of 2 bytes, and the UJ-type is a U-type instruction format with an immediate value that instead of being shifted 12 bits is only shifted 1 bit to encode unconditional jump offsets in multiples of 2 bytes [7].

Privileged Architecture

The interaction between hardware and users can be abstracted with a software level stack where user level is at the top and hardware level is at the base. Applications may run directly on hardware, the only communication interface between the application and the hardware would then have to be the Application Binary Interface (ABI), which contains the rules and guidelines, i.e. the ISA, for how to emit code to communicate with the hardwired hardware platform, or a configured hardware platform. The ABI may also provide a link for communications with an operating system, and the operating system itself may be communicating with a virtual machine or the hardware. The virtual machine also communicates with its platform, and so on. For each of the levels communicating with each other through binary interfaces, there is a certain privilege with the hardware level having the highest privilege. There are up to 4 so called privilege levels which are part of the privileged architecture. The 4 levels are labeled user (U), supervisor (S), hypervisor (H) and machine (M). Applications run on user level, operating systems run on supervisor level, virtual machines run on hypervisor level, and hardware obviously runs on machine level, which is the only level that must always be provided in all hardware implementations because of its unrestricted access to the entire hardware platform [15].

The privileged architecture is a hierarchy of protected domains in hardware to where access is only granted to hardware threads running at least at the lowest allowed privilege level. Attempts to access higher privileged domains, or trying to perform such operations, will cause exceptions. Exceptions can also be caused by hardware interrupt requests, and once anything causes an exception. Such information is stored in control and status registers (CSRs). CSRs are basically the foundation of the privileged architecture; they store system information, interrupt flags, and general-purpose input/output (GPIO) data for example. Information regarding hardware thread modes is also stored in them [15].

The privileged instruction set is a list of instructions all using the I-type format. This instruction set includes instructions that atomically read or write a CSR, change privilege level, redirect traps, the wait for an interrupt and manage memory. All privileged instructions are encoded as system instructions [15].

Extensions to the RISC-V ISA

As mentioned above RISC-V is extensively customizable as far as to the base integer ISA, i.e. the base integer ISA can be extended for special purposes, but not redefined [7]. New instructions can be added to the set to support special semantics normally not supported by the base integer ISA. One example is the real-time semantics which is a major part of the PRET infrastructure. Instructions are added for the ability to read timestamps and define time boundaries.

The ability to read time is provided by the get_time pseudo-instruction which reads the current time from a CSR since the last boot or reset. The returned value is stored in a destination register. The clock can be triggered to put a lower and upper bound on timed execution by setting the compare register, which is another CSR. This is done by executing the set_compare
pseudo-instruction. The set_compare must then be followed by an instruction using the comparison result. Note that trigger conditions for interrupts can also be disabled by executing set_compare with the x0 register as input, which is hardwired to constant zero. 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. During the stall the hardware thread is put to sleep in order to allow its cycles to be shared, only waking up upon interrupt or time expiration. The delay_until keeps the program counter locked at its address, so it keeps repeating after return from an interrupt, and then going to sleep, whereas the wait_until increments the program counter before sleeping the hardware thread so that an interrupt would eventually make it resume execution. An upper time bound is provided by the interrupt_expire and exception_expire pseudo-instructions, which both set the timer to trigger an interrupt or exception depending on what cause identifier is being used. The interrupt or exception is triggered when the timer expires [16].

### 2.4 Scala & Chisel

The Scala language is an object-oriented programming language that fully supports functional programming. It brings together concepts from both domains and the result is a language that is rich in coverage and doesn’t need so much code. It was built on the Java Virtual Machine platform, thus it is fully compatible with the Java library. Similarly Java is fully compatible with the Scala library. The name Scala is a portmanteau, i.e. a contraction, of Scalable Language, which is to signify its ability to scale with the demands of its users.

Scala has many built in features; among them is the Chisel hardware description language, which FlexPRET is distributed in. Chisel can be thought of as a high level HDL compared to VHDL and Verilog, i.e. it can produce Verilog code for synthesis in analogy with compilers that produce instructions to be executed on a machine. The same way a compiler produces instructions from source code in several internal functions in one file or external functions in separate files, does Chisel take the contents of one or many Scala files to synthesize a Verilog file. Chisel is also able to produce C++-code for emulation purposes of the hardware configuration.
In this chapter we discuss the implementation methods to deal with the problem of the thesis, from generating a Verilog file from Chisel code to programming the synthesized processor. The first section deals with the generation of the FlexPRET Verilog file from a set of files written in Chisel, a hardware description language embedded within the Scala programming language. We discuss how to invoke Chisel, how to set up the file system for compilation and generation, and how to input values for configuration. Furthermore we go into detail with configurations and discuss why some configurations are valid and some are not.

The second section deals with synthesis process of the generated Verilog file. Here we realize that providing proper code for synthesis is a key factor when it comes to FPGA resource utilization. Especially when it comes to memories, as they synthesize into arrays of flip flops, and they occupy a huge chunk of the logic elements. Therefore the Verilog file that is generated by Chisel and Scala cannot be synthesized as it is; the memories need to be written in a supported coding fashion. One solution is however to scale down the memories, but a better one is to create black box style code to infer megafunctions that provide access to the memory blocks of the FPGA and instantiate them in the main code.

The third and final section deals with the programming of the synthesized processor. Since FlexPRET is a RISC-V based processor, the executable code file must contain RISC-V code as well, i.e. the written code must be compiled with a RISC-V cross compiler. A cross compiler is a compiler that is hosted by one system, but compiles code for any other target system. We will deal with how to set up the RISC-V newlib cross compiler from the RISC-V GNU toolchain, in order to compile C/C++ code into executable RISC-V code. We list and explain all the required package prerequisites for the building process, as well as explaining the importance of using a UNIX-like operating system, and not just an emulated UNIX-like environment. We also deal with how to execute the code on FlexPRET.

### 3.1 The generation of FlexPRET

FlexPRET can be configured in many ways, such as the number of threads, i.e. allowing or not allowing flexible thread scheduling, size of scratchpad memories, instruction and cycle counting and enabling or not enabling one or more timing instructions. The initial source files are found in a Git repository on GitHub [4]. Git is a distributed revision control system, i.e. a file sharing system for members of a project group to gain instant access to the project files on any computer that is connected to the internet. Revision control allows users to keep track of version histories and older versions of files. Everything is stored in a repository on a Git server, which any authorized user can access anywhere. GitHub implements this as a web-based service, where repositories can be either private or open-source. FlexPRET goes under the latter, which means that anyone can download and use the contents. The source files are written in Chisel that will generate a Verilog file from a description that is given in the console input to Scala. Chisel is embedded in the Scala language, and must be invoked through a command line in the project’s build.sbt or build.scala file. The build.scala file that is included in the FlexPRET repository on GitHub invokes Chisel through the line:

```
libraryDependencies += “edu.berkeley.cs” %% “chisel” % “latest.release” [9]
```
The build.scala or build.sbt is the project definition file in a Scala project and it must always be included in the project directory. It contains a list of settings defined for the project that will apply as transformation to the project’s map. The simple build tool will always look for the build definition file when building the project. Furthermore the build.sbt must be in the root directory of the project, or more correctly, in the directory where the simple build tool is executed. If it is a build.scala file it must be in a project subdirectory to the directory where the build tool is executed. In the downloaded FlexPRET repository from GitHub [4] the build.scala has the search path root/sbt/project/build.scala, which means that the build tool must be executed from the root/sbt directory. In that same directory there is a file named sbt-launch.jar to be found, this is a self-executable launch file that doesn’t require a prior installation of the simple build tool. The generation of FlexPRET is now done by executing the following command in a command shell:

```
root\sbt> sbt “project Core” “run <config_string> --backend v --targetDir <directory>”
```

The <config_string> placeholder is replaced with the configuration string for FlexPRET and the <directory> placeholder is replaced with the search path to the location of the output file as related to the current directory, i.e. root\sbt. The subcommand “project Core” tells sbt to look for files in the src\Core directory, as specified in the build.scala. The subcommand “run” passes the <config_string> as an input argument to the main function. This is where FlexPRET gets configured. The --backend v command tells Chisel to generate a Verilog file that is the FlexPRET Core. Chisel can also generate C++ files for emulation; this is done by executing the sbt command with --backend c instead of --backend v. Finally the --targetDir <directory> tells Chisel to output the Verilog or C++ file to the specified output folder. The configuration string should look as the following:

```
<number_of_threads>tf-<size_of_I-SPM>I-<size_of_D-SPM>D
```

The <number_of_threads> placeholder is an integer number between 1 and 8, t stands for threads and f should denote flexible scheduling. The <size_of_I-SPM> and <size_of_D-SPM> are sizes of scratchpad memories in kilobytes (kB). FlexPRET is configurable for a large variety of combinations of thread counts, scratchpad sizes, and whether or not scheduling should be flexible, that the user is free to choose between. However, not all combinations are valid. Flexible scheduling is possible for \(2 \leq \text{number\_of\_threads} \leq 8\), whereas strict round-robin scheduling requires a minimum thread count that is one less than the number of pipeline stages. Thus, round-robin scheduling is only possible for \(4 \leq \text{number\_of\_threads} \leq 8\), since the number of pipeline stages is 5. Finally we have \(\text{number\_of\_threads} = 1\) which denotes the base configuration of FlexPRET where neither flexible scheduling nor round-robin scheduling would make any sense.

The resulting output is a Verilog file Core.v in destination folder given by the placeholder <directory>. A typical output address would begin with ./ to get back to the root directory, and end with the chosen output directory. Preferably the output directory should be named after the configuration of FlexPRET and its target, e.g. emulator/generated-src/4tf-2I-2D for a 4 threaded 2x2 kB configuration for emulation, or fpga/generated-src/8tf-16I-16D for an 8 threaded 16x2 kB configuration for FPGA deployment.

18
3.2 The Synthesis of FlexPRET

FlexPRET is synthesized in Quartus II by Altera, which provides support for Cyclone IV devices. The device we are synthesizing for is EP4CE115F29C7 of the Cyclone IV family, which has 114 480 logic elements and 3 981 312 dedicated memory bits that can be utilized. However, if we are not careful with coding style the synthesis tool may end up not fitting the design into the correct resources. One major problem comes with synthesizing the scratchpad memories; the memories may turn up as arrays of flipflops and will turn up to use a huge portion of the logic elements. This is something we must avoid at all costs and this is why we have to be careful about how we deal with memories. Therefore, if needed, we must modify parts of FlexPRET for better resource utilization. Altera fortunately provides support with the Quartus II software to compile memories in the megawizard plug-in manager. The output of the megawizard is a file with HDL code, which Quartus II will replace with a functional block that is known as a megafuction. A megafuction is a pre-built module that performs a required function, such as arithmetic/logic functions, DSP modules, memories and shift registers. They can be created using the Altera megawizard plug-in manager and instantiated in the design, or they can be inferred from previously written HDL code if Quartus II recognizes such behavior in that code. Megafuctions can save a lot of time as they do not need to be written if they are created; the code is already written. They also help save area as they are optimized for Altera devices, and in addition they are required for gaining access to architecture specific features [11].

The full compilation process performs analysis and elaboration to structure up the design. First it analyzes the code to check for any syntax and semantic errors, and then it elaborates on the logic to synthesize and structure up the design elements. After synthesis the design is viewable in the RTL viewer. At this state we have a synthesized design, but the compilation process will move on to the fitter to try to fit the design onto the FPGA. This is where designs that occupy more area than there is capacity for will cause the compilation process to fail. As an example the 8tf-16I-16D configuration requires more than 130 000 logic elements and is therefore way too big for the target device. As another example the 4tf-4I-4D configuration requires around 50 000 logic elements, which does make it fit into the device, however it should be using memory bits. The reason why to this is that with the synthesis of the scratchpad memories, which itself is the result of HDL code where Quartus II doesn’t verify the memory logic as RAM logic, ROM logic, or any other recognizable memory logic [11], instead of inferring a megafuction the synthesis tool will use the logic elements which require a lot of FPGA space. Altera provides megafuction support for optimized resource utilization, one of them is the access to special memory blocks on the FPGA. But in order for it to work, Quartus II has to recognize and verify the RAM logic during synthesis, and this does not happen if the code is written in an unsupported fashion.

Figure 3.1 – The Black Box approach used by Quartus II for megafuctions.
The RAM logic is basically a 2-dimensional array of registers in the HDL code. When recognized and verified by Quartus II, the RAM logic is put in a black box, i.e. a functional block that is mapped to the memory blocks of the device. In case the module contains more than just a RAM block, i.e. additional logic, it cannot be verified, and thus not mapped to the memory blocks, as that logic would also have to be treated as a black box for formal verification [11]. Obviously we cannot run compilation on the Core.v file that was generated in Chisel as it is, because the scratchpad modules that are synthesized with the design contain extra logic. We have to use more suitable Verilog memories that can be instantiated. We have to create black-box style code to instantiate in the memory modules, i.e. the RAM logic must be separated from the memory modules. The megawizard plug-in manager has a memory compiler in which it is possible to create and customize RAM/ROM logic, FIFO-buffers and shift registers. Logic that is recognizable and verifiable by the Quartus II software. RAM logic can implement single-port RAMs, dual-port RAMs, or tri-port RAMs; suitable for whichever purpose they are used. In the customization wizard we can, if we’re creating a dual-port RAM, check whether both ports should be readable and writable, or if one should be read and the other should be written. We can also choose if we want a word sized or bit sized memory. We can specify the word size, the word count and the memory block type, as well as the clocking methods and initial memory contents.

![Architecture and hardware resources of the Cyclone IV family of FPGAs.](http://www.altera.com)

The memory blocks in the Cyclone IV devices are of the type M9K, i.e. memory blocks each containing 9 kb (9216 bits) of memory bits, out of which 1024 are parity bits, which means that each block can provide mainly 8192 bits and 9216 bits in packed mode. 8192 memory bits can support up to 1024 bytes of memory, as 8 bits make 1 byte. Depending on how many M9K blocks the FPGA has available, one can fit considerably large memories into the device. It is in fact not a question of how many logic elements the device has, but rather how many M9K blocks it has. According to technical specifications for the Cyclone IV device family published by Altera [12], our target device, the EP4CE115F29C7 FPGA, has 3 981 312 memory bits (432 M9K blocks) and should support up to 486 kB of memory, which means that a 16x2 kB configuration of FlexPRET should by all means fit perfectly into the device.

The fitter will route the input and output ports of the design to suitable pins on the FPGA, it is of course possible to modify the pins in the pin planner once synthesis is successful. But if one tries to synthesize the Core.v file as the top level module, pin planning will be incredibly hard to
straighten out. Instantiating the core module in a designated top level module with only the relevant input and output ports is ideal here.

Quartus II will also perform timing analysis on the design to help determine critical path and what clock input pin to employ in the pin planner, during the compilation process.

After successful compilation and pin planning we have an SRAM offset file (.sof) that can be launched on the specific Altera FPGA. Launching such files is done in the Quartus II programmer.

### 3.3 Programming FlexPRET

The programming environment for FlexPRET has two parts; one is a RISC-V cross compiler, and the other one is a set of makefiles with support for multithreaded execution and creating hexdumps for initial memory content. FlexPRET is a RISC-V processor; programs running on it must be compiled using a RISC-V cross compiler, i.e. a compiler that runs on one platform to compile code for another platform. There is a RISC-V GNU toolchain that can be downloaded and installed, which will build the RISC-V-GCC cross compiler. And then to emit code that can execute on FlexPRET we must execute makefiles that contain compiler calls and include a startup routine for execution on multiple threads. Source code must include header files containing macro definitions of CSRs, thread initializers and implementations of real-time semantics among other things. To execute the code we must create memory initialization files or Intel format hexfiles from the contents of the hexdumps, and synthesize FlexPRET with these files specified for initial memory content.

#### Building the Cross Compiler

The newlib cross compiler is found in a GitHub repository [10], where it is fully downloadable as a toolchain containing GCC, newlib, glibc, Linux headers and binutils for RISC-V. The binutils contain assemblers and linkers for target architectures, plus other programming tools for managing and creating programs, objects, assembly code and libraries. Newlib is a C language standard library implementation for embedded systems. Glibc is the GNU implementation of the C standard library. In order for the installation to succeed there are a few packages that must be installed first, i.e. prerequisites that are needed for building the GCC cross compiler for RISC-V and modifying it:

- gawk: GNU implementation of awk, an interpreted programming language, i.e. it performs instructions without prior compilation of a language. Awk was made for text processing and running scripts. This is required for creating some of the generated source files for GCC.

- autoconf: Part of the GNU build system. Autoconf generates configure scripts based on what is in the configure.ac file, which is a file containing user specific desired behavior written in the m4 language in form of instructions. The m4 language is a general-purpose macro preprocessor which can usually be found embedded in compilers and assemblers, where they process text and transform it to something else. The configure script generates a configure.status script from its build environment, and the configure.status script in turn converts input files (such as Makefile.in files) to Makefile output files. A Makefile is a howto prescription for make that tells the utility how to compile and link programs written in a given language. The Makefiles are executed in make to create executable files and libraries from source code. This is needed for modification of GCC.
• automake: Part of the GNU build system. Automake automatically generates Makefile.in files, which can be used to generate Makefiles. The Makefile.in files are generated from Makefile.am files that contain variable definitions for compiled software, such as compiler and linker flags, or dependencies. This is needed for modification of GCC.

• bison: General-purpose parser generator for conversion of annotated context free grammar into left-right text. Bison is part of the compiler; it reads sequences of tokens and decides whether or not they are consistent with grammar specified syntax. A syntax tree is built and is successful if the source code is correct.

• flex: Short for Fast Lexical Analyzer Generator. A lexical analyzer is a program that converts sequences of characters into sequences of tokens. They are also known as tokenizers. Flex has a Finite State Machine with every possible recognizable symbol available for identification. Flex is used together with bison to analyze the syntax of source code and is part of the compiler. Lexical analysis has to be performed before parsing because the source code may contain preprocessor directives such as library/header inclusions, pragmas and macro definitions. Those must be resolved before we can build a syntax tree out of the code.

• texinfo: Typesetting syntax that is used to generate documentation in several filetypes such as html, pdf and info.

• patchutils: Utilities for manipulating patch files. A patch file is an instruction file that is run by the patch program to update the contents of a text file. The instructions in the patch file tell how to do it. The patch file merely contains differences between two versions of a file, so the utility creating such files is called a diff program. Patchutils contains a set of diff utilities that can create and modify patch files. This is necessary for patching over sources.

• autotools-dev: Installation of up to date versions of config.guess and config.sub. Config.guess is a script that tries to determine the host’s name, and config.sub is an autoconf generated file that converts system aliases into full canonical names.

• libmpc-dev: Multiple precision complex floating-point library development package. It is a library written in C that supports floating-point arithmetic with correct rounding on complex numbers, exponential functions, logarithmic functions and trigonometric functions. This is a dependency that is needed to build GCC.

• libmpfr-dev: Multiple precision floating-point library development package. This is the same as above, but unlike libmpc it doesn’t support complex numbers. It does however provide support for floating-point numbers with correct rounding. Libmpc is based upon libmpfr, and therefore libmpfr is as well needed to build GCC.

• libgmp-dev: GNU multiple-precision arithmetic library. This is a library for arbitrary precision arithmetics including integers, rationals and floating-points. The mpfr and mpc libraries build upon gmp. The development packages of the libraries above contain headers and symbolic links for compilation and linking of programs that use the named libraries. The gmp library is needed to build GCC.

• build-essential: Contains a reference list for all the packages that are needed to build a Debian package, like GNU. Debian is one of the biggest Linux distributions and it’s known to be the foundation of Ubuntu. One of its key features is the APT tool that
enables downloading and updating of packages and package dependencies from repositories.

All of the packages above are installed on Ubuntu by executing the following command:

$ sudo apt-get install <package_0> <package_1> <package_2> … <package_n>

The sudo command executes a program in Ubuntu that grants superuser privileges to the user, this is equivalent to executing programs in administrator mode in Windows. Apt-get tells the apt tool to search for the packages above and retrieve them, and install simply tells the system to install all of them. In order to install the packages, all the names must be added to the command line separated by a blank space. When all those packages are installed, the RISC-V GNU toolchain can be downloaded and built. A very simple way to download contents from a repository is by having Git installed on the system, which is done by running apt in the same fashion as with the prerequisites, i.e. the command to be executed is

$ sudo apt-get install git

With Git installed downloading contents from a Github repository is simply done by cloning the repository. Cloning a repository is done by executing

$ git clone <repository_address>

The repository address is found on the right side of every repository. It generally starts with github.com, and then continues with names of the subdirectories, and it ends with the .git top domain. Both the RISC-V GNU Toolchain and the FlexPRET Github repository are downloadable this way, or as zip-files from their corresponding Github repository pages. Once the content of the repository has been downloaded, the toolchain is ready to be built. This is done by running the makefile in the downloaded directory. It should however be given an install search path first, for instance /opt/riscv/bin. This is done by executing

$ export PATH=$PATH:/opt/riscv/bin

The makefile then gets the install path /opt/riscv by executing

$ ./configure --prefix=/opt/riscv

This configuration being the default configuration builds a compiler named riscv64-unknown-elf-gcc, which by default emits 64-bit instructions. We will get 32-bit instructions by passing architectural flags when compiling, as there are 32-bit object files available as well after building. The newlib toolchain cannot be installed on Windows through Cygwin or any other UNIX-like environment emulator without support for backtracing. The Linux header file libexec.h is not included and that will cause a fatal error in the installation process since it cannot
be found. Installation has to be done in a UNIX-like system, such as Ubuntu or Debian. However, a virtual machine running Ubuntu is indeed sufficient. The building process is initiated by executing the make command. However, failure is still possible if access cannot be granted to all directories, i.e. the make command has to be executed in superuser mode to make sure that access is not denied anywhere. The command to execute to start building the riscv64-unknown-elf-gcc compiler is:

$ sudo make

Once the RISC-V-GCC cross compiler has been built the FlexPRET Git repository contains every useful header file and assembly routine for programming, as well as makefiles with commands to output hex-files and splitting contents for data and instruction scratchpads. These FlexPRET specific header files must be included and called from the program.

**Compiling Programs**

To build an executable binary machine code file with 32-bit version code out of an existing C or C++ file, and naming it, the following command should be executed in a terminal window.

$ riscv64-unknown-elf-gcc -m32 -o <name> <file>.<file_extension>

riscv64-unknown-elf-gcc is the name of the compiler that we have built. Since FlexPRET actually requires 32-bit instructions, we have to pass the -m32 flag in the command line to tell the compiler and the linker to look for the 32-bit files. Compiling with the -o flag tells the compiler that the output file should be named using a name given by the user from the command line. The <name> placeholder in the above command is replaced with the chosen name. The <file> placeholder is the name of the written program file and the <file_extension> placeholder is the file extension of the existing source file, which is given according to the used programming language, i.e. c for a C-file and cpp for a C++-file respectively. For example, if we have a C-file named bar and we want to compile it and call the output of the compiler foo, the command to be executed should then be

$ riscv64-unknown-elf-gcc -m32 -o foo bar.c

Now since foo actually is a RISC-V code binary file, it cannot be executed on the host machine, it has to be executed on a RISC-V machine. For execution we must extract machine code from the executable file foo and upload it to a memory. In a dedicated development environment this would be done automatically by the software once we initiate execution, we need to do it manually. For execution on FlexPRET we have to split foo into two separate hexdumps; one containing the instructions segment and the other one containing the data segment. This, including compilation, is done by running a makefile that is included in the root directory of the downloaded repository. Running this makefile will automatically compile files in the tests/isa directory and output two hexdumps with respective memory content. This is done by executing

$ sudo make run
This command not only compiles the test files, it also runs tests on them using a target core specified in config.mk in the same directory. If needed, the makefile also installs sbt, downloads Chisel and generates a test core for emulating programs. If we only want the files to be compiled and split into hexdumps, we can do that by executing

$ sudo make prog

Initially the files that are to be compiled are found in the tests/isa subdirectory. This can be changed to any other directory by editing the config.mk file in FlexPRET’s base directory. The config.mk file contains configuration parameters for FlexPRET, and target and directory information. The TARGET variable can be either fpga or emulator, the DEBUG variable can be either true or false, and finally the PROG_DIR variable is the one that should be set to the desired program directory, i.e. isa, dev/<subdirectory>, or examples. The includes directory contains header files that will be described more in detail below. The dev directory contains several subdirectories; bootloader, complex-mc, fpga, malardalen-wcet, simple-mc and timing-instructions. One of them must be included on the <subdirectory> placeholder. Running the makefile with a new PROG_DIR in config.mk will autocompile the files in that directory instead of tests/isa. Additional files can be included in any directory, but if they are to be compiled, they must be included in the PROG array of the local test.mk, i.e. we must edit the test.mk and add the name of the new file into PROG, as follows from the simple example below.

```
PROG ?=\n foo \n bar \n bletch \n <new_file_0> \n <new_file_1>
```

If the compiler is not found when we run the makefile there will be an error, the reason why is simply that tests.mk needs to be edited so that the variables RISCV_GCC, RISCV_OBJDUMP and RISCV_OBJCOPY are set to the absolute paths to riscv64-unknown-elf-gcc, riscv64-unknown-elf-objdump and riscv64-unknown-elf-objcopy respectively. The path to the directory where they are found can be found in the $PATH environment variable if it was exported when building the toolchain. In our previous build example we added “/opt/riscv/bin” to $PATH, the absolute paths following from that should be:

```
RISCV_GCC = /opt/riscv/bin/riscv64-unknown-elf-gcc
RISCV_OBJDUMP = /opt/riscv/bin/riscv64-unknown-elf-objdump
RISCV_OBJCOPY = /opt/riscv/bin/riscv64-unknown-elf-objcopy
```

The makefile will output two memory hexdumps after compiling a source file <file>.c, named <file>.data.mem and <file>.inst.mem. If we compile bar.c, we get bar.data.mem and bar.inst.mem. Contents from those files should be added to the scratchpads as initial memory content.
Writing Source Code

The FlexPRET programming interface consists of a list of c and assembly language headers whose inclusion is required. These contain the instructions and definitions that are needed to run the system properly, i.e. macros for activating threads, writing the control registers, and defined operation codes. This is the binary interface of FlexPRET, and its default values assume a configuration with 4 threads and 16 kB data memory. The header files are found in the tests/include directory in the Git repository.

The flexpret_threads.h header contains initialization, startup and yielding routines for hardware threads with functional pointers, thread IDs and stack pointers. There are also definitions for setting thread states and scheduling, i.e. setting the thread mode control and slot control registers. When initializing a thread, we must input its thread id, the name of the function that we intend to execute on that thread, and a desired stack pointer address that is only set if the input value is any other than null. The thread states are all stored in an array of 4 elements, 4 being the preset number of elements, which is consistent with the default value. It can be changed to match different configurations with up to 8 threads. The flexpret_io.h header contains IO-routines for string handling, and for reading and writing the Control/Status Register (CSR) unit, for example for General Purpose Input Output (GPIO). The flexpret_timing.h header contains definitions for delay and sleeping times in nanoseconds, microseconds and milliseconds. These headers must be included in FlexPRET software applications to utilize the hardware threads. The initializer in the threads header must be called to start a thread and assign code to it. The encoding.h header contains op-codes, interrupt request numbers and control register function declarations. This header is included in all the headers above.

The application that we are going to run is a simple one that blinks LEDs by setting and clearing GPIO bits in the CSR, provided that the GPIO outputs are assigned to LED pins. Blinking is achieved by setting one or more bits, and clearing the same bits a short while after that. This turns the corresponding LEDs on and off respectively. If done fast, but slow enough to be visible for the eye, we have a blink.

The header file flexpret_io.h defines function prototypes for setting and clearing GPIO bits, and the functions are macro definitions that infer instructions through inline assembly. The inline assembly instructions are given in the header file encoding.h. The instructions that access the CSR are I-type, and they consist of a 12-bit address where the two most significant bits (11 and 10) set the CSR to either read/write mode or read-only mode, the second two most significant bits (9 and 8) set the lowest privilege mode, i.e. the lowest privilege that may access the CSR (machine level being the highest and encoded as 11, whereas user level is the lowest and encoded as 00). The rest of the address maps to the chosen CSR. The instruction also has either a source or a destination register encoded by five bits, a 3-bit encoded function and a 7-bit encoded operation code. CSR-accessing instructions are encoded on the following form.

[ src/dest address (12 bits) | src r1 (5 bits) | function (3 bits) | dest r2 (5 bits) | OP-code (7 bits) ]

The GPIO function calls found in flexpret_io.h that map to the CSR instructions are further explained below along with their corresponding instructions that are found in encoding.h.

- gpo_set( mask ) – uses the csrrsi instruction, i.e. Atomic Read & Set bit in CSR, to set the bit corresponding to the value mask to the CSR given by a specific address. The read value is stored in the destination register. The mask is a zero-extended 5-bit immediate passed as the function argument.
• **gpo_clear( mask )** – uses the csrrci instruction, i.e. Atomic Read & Clear bit in CSR, to clear the bit corresponding to the value mask in the CSR given by a specific address. The read value is stored in the destination register. The mask is a zero-extended 5-bit immediate passed as the function argument.

• **gpo_write( val )** – uses the csrrwi pseudoinstruction, which is encoded as csrrwi with x0 as destination register r2, and a zero-extended 5-bit immediate coming from the input argument val. The csrrwi instruction writes the value val to the CSR given by a specific address. The read value is stored in x0, i.e. it is discarded.

• **gpo_read( )** – uses the csrr pseudoinstruction, which is encoded as csrrs with x0 as source register r1, i.e. no value is written to the CSR after reading. This is a special case that applies to both csrrs and csrrc, no value is written when the source register is x0.

• **gpi_read( )** – The same as gpo_read( ), but reads other CSRs.

The CSR addresses that are used according to encoding.h for GPIO are 0xcc0 to 0xcc3 for GPIO input and 0xcc4 to 0xcc7 for GPIO output, which are all non-standard user level read-only addresses according to the RISC-V instruction set manual [15]. It means that user level is the lowest privilege level to access the CSR, and any level above may access it.

**Executing Programs**

Once a program is written and successfully compiled by running the makefiles, we have code that is ready for execution on FlexPRET. However, the hexdumps cannot be executed as they are. Their contents must first be transferred to Intel-format hexfiles (.hex) or memory initialization (.mif) files in Quartus II. Such files can be created as text files, and then be saved as .hex or .mif files. Memory files must contain the following information:

```
DEPTH = <number_of_words>; -- Number of words in memory
WIDTH = <word_size>; -- Word size in bits
ADDRESS_RADIX = <radix>; -- Radix for addresses
DATA_RADIX = <radix>; -- Radix for words
CONTENT
BEGIN
<address_0> : <data>; -- Data is pasted here
<address_1> : <data>; -- Each word must have an address
<address_2> : <data>; -- Semicolon marks the end of a line
: 
<address_n> : <data>;
END; -- Everything ends here
```

The radix can be specified as hexadecimal (HEX), signed decimal (DEC), unsigned decimal (UNS), octal (OCT) or binary (BIN), either notation works fine. However, the data must conform to the specified radix. The output data from riscv64-unknown-elf-gcc is hexadecimal, specifying the radix as anything else would indeed require converting all data to that radix too. The depth and width should conform to the memories’ dimensions, if the word size is 32 bits and there are 4096 lines, the memory files should have a width of 32 bits and depth of 4096 words. Between the CONTENT BEGIN and END keywords we should place the actual memory content in “address : data” pairs, i.e. each address is associated with some data. It is also possible to assign data to multiple addresses using brackets to group addresses and adding space between
the data words. We do not need to assign data to all addresses; the uninitialized addresses will be assigned zeros by Quartus II during synthesis anyway.

The memory files are then specified as initial memory content in the megawizard plug-in manager, where we created the memories. Since we are actually replacing the verilog files in which the memories are specified, or just changing their contents, we must synthesize the project again, which will create a new .sof file to program the FPGA device.
The results of our work are explained and elaborated in this chapter. We begin with discussing the synthesis process. Many things can go seriously wrong, if we are not careful with coding style, and really mess up the resource utilization. Section 4.1 will thoroughly go through the attempts to fit FlexPRET onto the FPGA by scaling down the memory system, as well as the optimized utilization of FPGA resources by providing proper HDL code to the synthesis tool. Section 4.2 discusses the attempts to install the RISC-V cross compiler on different UNIX-like environments, and the small application that was developed for evaluating the synthesis results.

4.1 Synthesis

FlexPRET is initially found as a set of files written in Chisel in a Git repository on GitHub. We have downloaded the RISC-V 2.0 branch files which contain an updated version of the processor and we have modified the function prototypes of the scratchpad memory files from class name(input_arguments) extends BlackBox into class name(input_arguments) extends module. This is done as a first attempt to get around the memory issues, as they prove to be a major hurdle in the synthesis process. The generation of the FlexPRET verilog code is done by from the sbt subdirectory of the root directory executing the following command in a command prompt:

```
root\sbt> sbt "project Core" "run 4tf-16I-16D --backend v --targetDir ../fpga/generated-src/4tf-16I-16D"
```

We have generated a flexible 4-threaded configuration with a 16 kB instruction scratchpad memory and a 16 kB data scratchpad memory. Synthesis process on the above FlexPRET configuration instantiated in a top level module not only fails, but the fitter does not map the memories to the correct FPGA resources, i.e. our synthesized processor requires 130 000+ logic elements when it should map to dedicated memory bits. This is one of the issues with synthesizing memories; they easily synthesize into flipflops, and this is a very time consuming operation. Attempts to scale down the memories in size obviously result in big drops in logic element requirements, as they synthesize into less flipflops. The 4tf-4I-4D configuration uses around 50 000 logic elements, but as can be seen from the compilation report that can be viewed in listing 4.1 below Quartus II has also inferred megafuncions from the memories, which occupy a total of 40960 memory bits. According to the compiler messages, however, it is the instruction scratchpad that has synthesized into two altsyncram megafuncions; i-spm~0 and i-spm~1, whereas the data scratchpad was ignored due to an asynchronous read port, which is not available for the Cyclone IV family. The 4tf-4I-4D configuration does fit on the FPGA, but this is a very time consuming process and an example of how we should not synthesize.

Looking at the number of available memory bits, i.e. M9K memory blocks, we find that the device can fit much larger memories. 432 M9K blocks according to the Cyclone II device handbook would hold as much as 486 kB of memory, i.e. 2 x 16 kB of memory would fit perfectly if we help with the resource optimization [12]. Thus, we should create the memories using the megawizard plug-in manager. Using the two Verilog scratchpad memories found in the FPGA directory of the Git repository as templates, two memories have been compiled with the megawizard plug-in manager; both memories being synchronous single clocked, dual-port RAMs styled with two 32-bit ports, as word size is 32 bits. The I-SPM has a read port for the datapath to read instructions from and a write port that can be connected to externally. The D-
SPM has two read/write ports, one connected to the load-store unit and the other can be connected to externally. The D-SPM also has byte enables on both ports to enable writes on specific bytes. In order to get 16 kB memories with 32 bit words, we need a word count of 4096 words. As a result of this, both memories get 12 bit address inputs. The instruction scratchpad is created with a write enable input signal, and the data scratchpad is created with exactly four byte enable input signals, all according to the structure of the original memories. The two compiled memories are instantiated in the place of the original memories in the scratchpad memory codes, i.e. the RAM logic is commented out and the wires are connected to the ports of the megafnctions. The results of the synthesis performed on the 4tf-16I-16D configuration with compiled memories optimized for Altera devices can be viewed in listing 4.1 as well.

Flow Status: Successful
Quartus II 64-Bit Version: 10.0 Build 218 06/27/2010 SJ Full Version
Revision Name: FlexPRET
Top-level Entity Name: Top
Family: Cyclone IV E
Device: EP4CE115F29C7
Timing Models: Final
Met timing requirements: N/A
Total logic elements: 47,692 / 114,480 (42 %)
Total combinational functions: 29,076 / 114,480 (25 %)
Dedicated logic registers: 37,898 / 114,480 (33 %)
Total registers: 37898
Total pins: 18 / 529 (3 %)
Total virtual pins: 0
Total memory bits: 40,960 / 3,981,312 (1 %)
Embedded Multiplier 9-bit elements: 0 / 532 (0 %)
Total PLLs: 0 / 4 (0 %)

Listing 4.1 – Differences between results of incorrect synthesis (left) and correct synthesis (right)

As can be seen above in listing 4.1, where the 4tf-4I-4D configuration’s incorrect compilation report is compared to that of the optimized 4tf-16I-16D configuration, the memories are now mapped to the memory bits. 262 144 memory bits equals 2x16 kB (2x8x16x1024) memory bits. We have optimized the utilization of the FPGA resources to such an extent that only true registers are synthesized into flipflops now whereas memories map to the M9K blocks. Logic usage has shrunk tremendously now that both scratchpads use memory blocks.

Figure 4.1 – M9K block utilizations for I-SPM (left) and D-SPM (right)
Each of the blue bars in figure 4.1 represents an M9K block, and each set of M9K blocks represent a scratchpad memory. The left one is the instruction scratchpad (I-SPM) and the right one is the data scratchpad (D-SPM). The external port of the D-SPM is grounded, but can be wired from the top level module. The I-SPM could be realized the same way as the D-SPM, i.e. with two read/write ports, but it works fine with only a read port and an external write port for our implementation.

The top level module for the 4tf-16I-16D configuration comes with inputs from the clock and reset, plus the General Purpose Input Output (GPIO) which also provides output ports. The GPIO consists of 1-bit sized inputs and 2-bit sized outputs that can be read and written to through dedicated CSRs, which are a set of special purpose registers that handle GPIO and interrupts among other internal and external events. The input is connected to the switch array of the DE2 board, and the output is connected to the red LED array. The external connectors to the scratchpad memories can be connected to instantiations in the top level module; they are grounded in our implementation. For DMA transfers of instructions the I-SPM would have to be reconfigured in the megawizard plug-in manager with two read/write ports, and then the write enable at the instruction fetch port should be grounded.

The top level module for the 4tf-4I-4D configuration is slightly different with GPIO inputs and outputs that are 1 byte wide. GPIO input and output widths can be tailored for any purpose necessary in the top level module, and that can be used in a huge variety of applications. The synthesized 4tf-16I-16D configuration of the FlexPRET processor with control, datapath and scratchpad modules is found in figure 4.2 below.

Figure 4.2 – The FlexPRET processor with control, datapath and scratchpad memory units
4.2 Programming

The first question with programming the FlexPRET system is how to compile source code on any system into RISC-V machine code, since FlexPRET is based on the RISC-V ISA. The answer to that question is to build a so called cross compiler. Fortunately there is a cross compiler for RISC-V, and the one we’re using is the Newlib GNU cross compiler [10]. Building a cross compiler in a UNIX-like system is the first part of setting up the FlexPRET programming environment, the other part is setting up scripts that compile and create hexdumps.

Attempts have also been made to build the Newlib cross compiler on Windows. Attempts on Windows 8.1 have been made with a MinGW+MSYS cross compiler system, however we decided that it would be much easier to have a UNIX-like environment, as the presented building commands would all be UNIX commands. Executing UNIX commands in Windows would require a UNIX-like environment, and thus, attempts have also been made to build the cross compiler in Cygwin, in which the prerequisites could be selected to be installed automatically. Cygwin, however, does not support backtracing and therefore the installation would fail when looking for libexec.h, a UNIX-specific header file containing calls for backtracing routines. Therefore that installation must be done in a UNIX-like system, i.e. any system running Debian, Ubuntu or any other Linux distribution. A virtual machine running Ubuntu would be enough.

The synthesized FlexPRET processor has to have code and data in separate memory files for initial content in the compiled memories. This requires that we hexdump the contents of a compiled executable file, and copy the data and bss segments into one file named <name>.data.mem and the code segment into another file named <name>.inst.mem. This is done by running the makefile found in the FlexPRET Git repository, which accesses a bunch of tests.mk-files that contain information about what files to compile and how. Additional information about what directory to compile in and what target to build is found in config.mk in the root directory. In order for compilation through the makefiles to work, the variables in tests.mk containing the commands to riscv64-unknown-elf-gcc with the -m32 flag, riscv64-unknown-elf-objdump and riscv64-unknown-elf-objcopy need their absolute search paths, which should be the bin subdirectory of the path set by the --prefix option in the RISC-V configuration file. The absolute path should also be found in the $PATH variable if it has been exported. Additionally the files that are to be compiled must be added to the .mk-file found in the corresponding subdirectory. For each file the makefile in the root directory will output a data memory hexdump and an instruction memory hexdump. The contents from the hexdumps are then used as data for the memory initialization files or Intel format hexfiles. For execution we must create memory initialization files or Intel format hexfiles in the Quartus II software, and specify them as initial memory content in the megawizard plug-in manager, which is where we have created the memories.

Two applications that blink the LEDs of the DE2 board and evaluate both single-threaded and multi-threaded execution have been developed for FlexPRET. The single-threaded application just executes a blinking sequence in a loop on hardware thread 0. The multi-threaded application executes the same blinking sequence on hardware thread 0 as in the single-threaded application, and also sets hardware threads 1 and 2 to execute their own blinking sequences. Hardware thread 0 is always assumed to be the master thread and according to startup.S it is assigned to execute the main function. The main function assigns the two functions named func1 and func2 to the hardware threads it wakes up. The main function of the single-threaded application is shown in listing 4.2, and the output blinking sequence is 11 – 00 – 10 – 00 – 01 – 00 – 10 – 11 – 00. The included headers contain all the important functions needed for setting and clearing GPIO bits, and handling timing semantics. There is another header included in the multi-threaded application which contains the functions and array needed for waking up hardware threads.
#include "flexpret_timing.h"
#include "flexpret_io.h"

int main()
{
    while(1)
    {
        gpo_set_0(3);
        gpo_clear_0(3);
        gpo_set_0(2);
        gpo_clear_0(2);
        gpo_set_0(1);
        gpo_clear_0(1);
        gpo_set_0(2);
        gpo_set_0(1);
        gpo_clear_0(3);
    }
    return 1;
}

Listing 4.2 – Single-threaded application for evaluating synthesis result.

The first thing both applications do is executing the startup routine startup.S, which assigns the main function to run on hardware thread 0, and then they start executing the main functions. The single-threaded main function executes the loop directly, whereas the multi-threaded main function begins with waking up thread 1 and thread 2, and assigning functions to them before continuing to the loop. The hwthread_start() function has 3 arguments; thread ID, function pointer and stack address. The thread ID is an integer number between 1 and one less than the number of hardware threads available, hardware thread 0 always executes main. The function pointer takes the address of the function that the thread is going to run. The stack address is set to null because it is already initialized in the array startup_state[ ], which is declared in startup.S at compile time and made global for external access. So we do not need to specify any other stack address when waking up hardware threads. This array is the one that hwthread_start() stores all the hardware thread states in. Slots and thread modes are then set to run all hardware threads in a round-robin fashion, which can be achieved by assigning one slot to each hardware thread and running them all in active real-time mode. Hardware thread 3 is sleeping in hard real-time mode and we do not need it to run. Orders of slots and hardware thread modes are always from the least significant bit nibbles to the most significant bit nibbles, i.e. the rightmost arrays of bits correspond to slot 0 and hardware thread 0 respectively, and then follow slot 1 and hardware thread 1, etc. In the threads header we find that all modes and slot assignments are encoded as macros, and all we have to do is use the macros with the corresponding function calls.

The function calls are, just like the GPIO function calls, inline assembly invocations of instructions that access CSRs. Slots are handled by 0x503 which is labeled badvaddr, and thread modes are set in 0x504 which is labeled ptbr. The GPIO function calls use 0xcc0 to 0xcc7, these CSRs are labeled uarch0 to uarch7. There are actually 16 uarch CSRs, but in our configuration we only use the first 7 for GPIO, where uarch0 to uarch3 are used for GPIO inputs and uarch4 to uarch7 are used for GPIO outputs. Configurations with 8 hardware threads may use uarch0 to uarch15. As explained in section 4.1 the GPIO has 4 1-bit inputs and 4 2-bit outputs, so the least significant bit uarch0, uarch1, uarch2 and uarch3 each comes from their corresponding switches on the board, and the two least significant bits of uarch4, uarch5, uarch6 and uarch7 each are sent to the red LEDs.

We use uarch4, uarch5, and uarch6, which correspond to the first, second, and third pairs of GPIO bits for our applications. Hardware thread 0 writes to uarch4, hardware thread 1 writes to uarch5, and hardware thread 2 writes to uarch6. This is by no means a mandatory solution, but a solution that we use to separate hardware threads from one another.
```c
#include "flexpret_threads.h"
#include "flexpret_timing.h"
#include "flexpret_io.h"

int main()
{
    hwthread_start(1, func1, NULL);
    hwthread_start(2, func2, NULL);
    set_slots(SLOT_T0, SLOT_T1, SLOT_T2, SLOT_D, SLOT_D, SLOT_D, SLOT_D, SLOT_D);
    set_tmodes_4(TMODE_HZ, TMODE_HA, TMODE_HA, TMODE_HA);
    while(1)
    {
        gpo_set_0(3);
        gpo_clear_0(3);
        gpo_set_0(2);
        gpo_clear_0(2);
        gpo_set_0(1);
        gpo_clear_0(1);
        gpo_set_0(1);
        gpo_clear_0(3);
    }
    return 1;
}
```

Listing 4.3 – Multi-threaded main function for evaluating synthesis results.

The code section in listing 4.3 above shows the same main loop executing in the single-threaded application, in this example it is set to run in parallel with func1() and func2(), which are described and shown below. Listing 4.4 shows the loop in func1(), which sets and clears the second pair of GPIO bits and outputs the following blinking sequence: 01 – 11 – 01 – 00.

```c
while(1)
{
    gpo_set_1(1);
    gpo_set_1(2);
    gpo_clear_1(2);
    gpo_clear_1(1);
}
```

Listing 4.4 – The main loop in func1.

Listing 4.5 shows the loop in func2(), which sets and clears the third pair of GPIO bits and outputs the following blinking sequence: 01 – 11 – 00.

```c
while(1)
{
    gpo_set_2(1);
    gpo_set_2(2);
    gpo_clear_2(3);
}
```

Listing 4.5 – The main loop in func2.

The source code in listings 4.2 to 4.5 has been compiled and two separate hexdumps have been created with content that has been copied into .mif files for execution. There’s either an issue with the compiler that makes it calculate incorrect jump and branch addresses, or with the architecture that makes it place the program counter at incorrect branch addresses. When we execute the code, the program counter does not fetch instructions from the expected addresses, but rather from the following addresses. And as a result of this, execution does not carry out as we might expect because important instructions are not executed. The first thing the main
function in the multi-threaded example does is waking up the hardware threads, but that doesn’t happen because an important address is not stored properly in a register as the base value for the store word instructions that store the start addresses to func1( ) and func2( ). This results in single-threaded execution of only the main function from which another problem rises, namely branch delay slots. Branch delay slots are required when additional hardware threads are not active, because the branch may not resolve until the execution stage. So by the time the branch actually happens there are other instructions in the pipeline and if they are incorrect, i.e. data that does not mean anything, the execution is forced to restart from the beginning, i.e. fetching again from address 0x0.

We have used two methods to fix these issues; adding CSR accesses to the source code that do not set any bits, i.e. a mask with only zeros, and hand assembly to alter the compiled machine code, i.e. recalculating branches and adding nops to fill out the branch delay slots. The alterations to the source code is to make sure that any instructions that store important addresses to registers at the beginning of functions indeed do get executed, whereas the hand assembly is to correct branches and making sure that no illegal instructions get fetched.

As can be seen in figure 4.3, adding nops and recalculating unconditional jump target addresses result in the expected behavior of single-threaded execution. The instructions that access CSRs are fetched and executed in a sequence, resulting in the output waveform we see on the LED section. We can see that the application writes to uarch4 at address 0xcc4, which is the CSR for the first pair of GPIO output bits. The CSR is written 4 cycles after the instruction is fetched, and then the output shows up on the LEDs on the very next cycle. After the last CSR access we have an unconditional jump (0xfd9ff06f) and 3 following nops (0x00000013), which corresponds to the jump first being fetched and issued at the first pipeline stage, decoded at the second and resolved at the third. After resolving there is a delay between the program counter and the actual fetch, which means that we need 3 nops to fill out the branch delay slots.

Figure 4.4 shows execution of the multi-threaded application. Execution is really messy and we do not get any outputs, but as can be seen the hardware thread scheduler does indeed schedule all active hardware threads in a round-robin fashion, and each hardware thread is given exactly one execution slot. The rest is messy and hard to straighten out. The CSRs uarch5 at address 0xcc5 and uarch6 at address 0xcc6 are being accessed, but no outputs show up on the LEDs. This is because the instructions accessing the CSRs actually do not set any bits, but rather clears them or are just redundant accesses, i.e. the access uses a zero-valued mask. The latter is one of the methods that we have used to make sure that any important calculation happens. These redundant CSR accesses can also be replaced with hand assembled nops.

The hardware threads actually seem like they only start executing their assigned functions, but do not get into the loops. Hardware thread 0 only manages to wake up the other hardware.
threads, but then it doesn’t get into its own loop, and neither do hardware threads 1 nor 2. Hardware threads are however scheduled for execution, so it is obvious that something happens in the program.

Figure 4.4 – Signaltap II Logic Analyzer output waveform from execution of a multi-threaded application.

We have experimented a little further with the machine code for an attempt to get some kind of output on the LEDs. Some of the CSR instructions accessing uarch5 and uarch6 prior to the loops, and that were fetched according to what can be read from figure 4.4, have been changed using hand assembly to set both GPIO bits in their respective CSRs to 1. Figure 4.5 shows that after a triggering reset, uarch5 is set to 0x3, and then gpo_clear_1(1) happens, whereas uarch6 is constantly set to 0x1 and nothing else happens.

Figure 4.5 – Signaltap II Logic Analyzer output waveform from execution of a multi-threaded application with modified GPIO output instructions.


5 DISCUSSION AND CONCLUSIONS

5.1 Discussion

We have tried to synthesize FlexPRET for the Cyclone IV family of FPGAs by Altera, which we from the results can conclude was successful. But if one is not careful with coding style, especially with memories, things can go very wrong. From the synthesis results we see that required FPGA space is strongly dependent on memory size, if we do not write proper HDL code from which the synthesis tool can recognize and verify RAM-logic. Unverified RAM-logic will synthesize into huge arrays of flipflops, and they will consume a lot of FPGA space, which results in rather small configurations in terms of memory to be synthesizable for our device. The FPGA, however, has dedicated memory blocks that are accessible only through the use of megafunctions, which can be inferred by synthesizing proper HDL code, but also by creating memories using the memory compiler of the Altera megawizard plugin manager in the Quartus II software.

We have created a configuration of FlexPRET in Chisel with 4 threads and 2 scratchpad memories at 16 kB each, which in its initial form was not synthesizable for the Altera device. Synthesis with smaller sized memory configurations showed that most configurations required more logic elements than what is available on the device. Interestingly enough only one memory, the data scratchpad, caused this behavior, whereas the instruction scratchpad caused two instances of synchronous RAM to be inferred. The solution to this was to compile two synchronous dual-port memories, one for the instruction scratchpad and one for the data scratchpad. The I-SPM was realized with a read port and a write port with the external read/write signals connected to the write port and datapath reading the read port. The D-SPM is realized with two read/write ports, one connected to the load-store unit and the other one connected externally, and byte enables on both. The FPGA has 432 M9K blocks available, each M9K block has the capacity of 8192 memory bits and 1024 parity bits, i.e. each M9K block holds up to 9216 bits and a total of 486 kB can be fitted into the device. Thus, it turned out quite possible to fit in 2 scratchpads of 16 kB on the target FPGA. Listing 4.1 also shows that the usage of logic elements drops dramatically when instantiating compiled memories, alternatively when synthesizing RAM-logic that is supported for the device family. The register file and the CSR would be the only registers to actually synthesize into flipflops when we optimize the resource utilization.

We have compiled a software application for evaluating the synthesis results. The RISC-V cross compiler has to be built in a UNIX-like environment. It does not work in environmental emulators like Cygwin, as they do not support backtracing. Rather it must be built in a UNIX-like operating system like Ubuntu or Debian. Several attempts were made to build from the GNU toolchain in Cygwin, all of them failed during search for the Linux-specific header file libexec.h. Failed attempts to build the cross compiler in Ubuntu had to do with not running make in superuser mode, i.e. the installation requires access to all directories in order to succeed. Furthermore, in order to successfully compile programs for FlexPRET we have had to build an older version of the RISC-V Toolchain, and compile using makefiles in the FlexPRET root directory. Ultimately everything boils down to how well software applications perform on the target. We have attempted to execute a number of applications, both single threaded and multithreaded, which show varying results. We have compiled and executed a number of single threaded applications that communicate with the open world through CSRs and GPIO. Compilation was done with makefiles to create separate instruction and data hexdumps, and memory initialization files were created with content from the hexdumps to provide initial memory content. We have
used the Signaltap II Logic Analyzer to extract data from output nodes among a few to verify functionality, from which we can conclude successful execution of single-threaded applications. We have also compiled and executed a multi-threaded application that did indeed execute, but the data extracted from the output nodes shows a very messy execution. Hardware threads are scheduled, but program counters tend to be corrupted somehow by faulty updates so that they run away and potentially land on illegal instructions. Execution is however possible, even though we are required to modify instructions through hand assembly and adding redundant operations before compiling.

5.2 Conclusions

Memories are not easy to synthesize, as they must be written in coding styles that follow the device vendors’ recommendations. One coding style that works for some devices may simply not work for other devices, and therefore the synthesis tool will not infer memory functions that access the dedicated memory blocks. Instead the logic of the code synthesizes into a huge array of flipflops and occupies a huge portion of the logic elements, sometimes more logic elements than are actually available if the memory is very big. That code must be rewritten so that the synthesis tool recognizes the specific behavior and can verify it. One alternative, however, is instantiating compiled memories in the main code if such options are supported by the software. Either way it all boils down to vendor and device specific macros, ultimately, for optimized resource utilization.

The RISC-V cross compiler is best built in Debian, Ubuntu, Linux, or any other UNIX-like system. Cygwin that is a UNIX-like environment emulator for Windows does not support backtracing, which will cause the building process to quit abruptly when looking for the libexec.h file. It is possible to build the 64-bit toolchain, and then use 32-bit flags to direct the compiler and the linker to the 32-bit object files so that the compiler emits the correct instructions for FlexPRET. And in order to gain access to all directories, the makefile has to be executed in superuser mode, or else building may fail. Makefiles are also used to compile and create hexdumps in FlexPRET’s programming environment, and separate hexdumps are required for the creation of initial memory content.

Compiled programs may need some extra hand assembly for proper execution, especially if the intention is multi-threaded execution. There are problems with jumps and branches where the main problem seems to be the wrong address, and if programs are executed on a single thread, then one must add nop instructions and perhaps recalculate the addresses. Multi-threaded execution may be carried out somehow, but this is something that requires further investigations.

5.3 Future Work

In order to make FlexPRET available as a node processor in the ForSyDe generated NoC mesh, future work includes porting FlexPRET towards the ForSyDe methodology. We should investigate in particular how the Altera SoPC builder and Xilinx systems work in order to create necessary wrapper functions for FlexPRET to connect with the ForSyDe/NSG tool suite.

Deeper investigations regarding how multi-threaded execution should be successfully carried out are required. We should experiment with hand assembly and source code to overcome issues that may be architectural and hardware related. We should try synthesizing for more powerful FPGAs, such as the Stratix series, and try other approaches for compiling scratchpad memories, e.g. implementing the I-SPM with two read/write ports and having one write enable grounded.
Investigations of using cache memories along with or instead of scratchpads must focus on minimizing loss in predictability as caches cannot eliminate that issue. We can experiment with several cache optimizations, such as cache size, block size and set associativity, or software optimizations in both code and compilation.


[13] A. Burns, R I. Davis; Mixed Criticality Systems – A Review

[14] DO-178C Software Considerations in Airborne Systems and Equipment Certification


[16] M. Zimmer; Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O


Specifications for the EP4CE115F29C7 FPGA of the Altera Cyclone IV family:

Device name: EP4CE115F29C7
Family name: Cyclone IV
Number of Logic Elements: 114 480
Number of M9K blocks: 432 (3 981 312 memory bits)
Number of Embedded Multipliers (18x18): 266
Number of PLLs: 4
Number of Global Clock Networks: 20
Number of User I/O Banks: 8
Number of Maximum User I/O pins: 528
Installing Git on UNIX-like platforms

```
sudo apt-get install git
```

Generating the FlexPRET processor from Scala source files

Files downloaded from:
https://github.com/pretis/flexpret/tree/riscv-2.0
and extracted to a directory named “flexpret-riscv-2.0”, which is the root directory.

Generating the FlexPRET processor on the Windows command prompt:

```
root\sbt> sbt "project Core" "run <config_string> --backend v --targetDir <directory>”
```

<config_string>: Configuration parameter string for FlexPRET; <number_of_threads>tf-
<size_of_I-SPM>I-<size_of_D-SPM>D
<directory>: Chosen directory where Chisel saves the Core.v output file.
<number_of_threads>: Number of hardware threads for the configuration.
<size_of_I-SPM>: Instruction scratchpad size in kB.
<size_of_D-SPM>: Data scratchpad size in kB.

For this thesis:

```
root\sbt> sbt “project Core” “run 4tf-16I-16D --backend v --targetDir ..\fpga/..\generated-src\4tf-16I-16D”
```

Generating the FlexPRET processor on a UNIX shell with parameters defined in config.mk:

```
$ cd flexpret-riscv-2.0
$ sudo make fpga
```

Note: The makefile emits the same sbt command as above, making it executable both on the
Windows command prompt and UNIX shell.

Optional way to download the master branch source files with git installed:

```
$ git clone https://github.com/pretis/flexpret.git
```

Building the RISC-V GNU toolchain

Files downloaded from:
https://github.com/riscv/riscv-gnu-toolchain/tree/9a8a0aa98571c97291702e283fc1056f3ce2e2
and extracted to a directory named “riscv-gnu-toolchain”.

The following commands are executed on a UNIX shell:

```
$ sudo apt-get install gawk autoconf automake bison flex texinfo patchutils autotools-dev
$ libmpc-dev libmpfr-dev libgmp-dev build-essential
$ cd riscv-gnu-toolchain
$ export PATH=$PATH:/opt/riscv/bin
$ ./configure --prefix=/opt/riscv
$ sudo make
```

Note: The RISC-V GNU toolchain must be built in a native or virtual UNIX environment.

Uninstalling the toolchain:

```
$ cd riscv-gnu-toolchain
$ sudo make clean
```
Compiling programs

Compiling programs in general for 32-bit RISC-V machines:
$ riscv64-unknown-elf-gcc -m32 -o <name> <file>.<extension>

<name>: Chosen name of output binary file
<file>: Name of compiled source file
<extension>: Language-specific file extension of compiled source file, i.e. S, c or cpp.

Note: The following commands must be executed in FlexPRET’s root directory, i.e. “flexpret-riscv-2.0”

Compiling the FlexPRET test files and running tests on them:
$ sudo make run

Compiling the FlexPRET test files without running tests on them:
$ sudo make prog

Building target specified in config.mk and compiling test files:
$ sudo make

Deleting everything produced by the makefile:
$ sudo make cleanall
// Single-threaded application

#include “flexpret_timing.h”
#include “flexpret_io.h”

int main()
{
    while(1)
    {
        gpo_set_0(3);
        gpo_clear_0(3);
        gpo_set_0(2);
        gpo_clear_0(2);
        gpo_set_0(1);
        gpo_clear_0(1);
        gpo_set_0(2);
        gpo_set_0(3);
        gpo_clear_0(3);
    }
    return 1;
}

// Multi-threaded application

#include “flexpret_threads.h”
#include “flexpret_timing.h”
#include “flexpret_io.h”

void func1()
{
    while(1)
    {
        gpo_set_1(1);
        gpo_set_1(2);
        gpo_clear_1(2);
        gpo_clear_1(1);
    }
}

void func2()
{
    while(1)
    {
        gpo_set_2(1);
        gpo_set_2(2);
        gpo_clear_2(3);
    }
}

int main()
{
    hwthread_start(1, func1, NULL);
    hwthread_start(2, func2, NULL);
    set_slots(SLOT_T0, SLOT_T1, SLOT_T2, SLOT_D, SLOT_D, SLOT_D, SLOT_D, SLOT_D);
    set_tmodes_4(TMODE_HZ, TMODE_HA, TMODE_HA, TMODE_HA, TMODE_HA);
    while(1)
    {
        gpo_set_0(3);
        gpo_clear_0(3);
        gpo_set_0(2);
        gpo_clear_0(2);
        gpo_set_0(1);
        gpo_clear_0(1);
        gpo_set_0(2);
        gpo_set_0(3);
        gpo_clear_0(3);
    }
    return 1;
}
Startup code for multithreaded application, assuming hardware thread 0 is the master thread. Thread 0 will run set the stack pointer and run the 'main' function. Currently, a hardware thread cannot modify the PC of another hardware thread, so all other hardware threads monitor a shared data memory location. When a non-zero value (desired PC) is set, the stack address is set from the subsequent word address, and the hardware thread jumps to this value. On completion, the thread will reset this value and sleep.

Assumptions:

**HW:**
- 4 hardware threads, 16kB data
- thread 0 scheduled
- All PC on boot/reset is 'reset'

**SW:**
- 'main' function

Michael Zimmer (mzimmer@eecs.berkeley.edu)

```
.text
.align 4

# code that all threads will run on reset
reset:
    # read thread ID
csrr a0, hartid
    # t0 (thread ID 0) will run initialization code (init)
    beqz a0, init
    # all other threads will watch memory location for non-zero value
    # compute address (using thread ID)
    la a2, startup_state
    slli a0, a0, 3
    add a2, a2, a0
    # check data at address and just loop if 0
    wait:
    lw a0, 0(a2)
    beqz a0, wait
    # desired PC set, load stack address and run startup code
    lw a1, 4(a2)
    j startup

# executed only by thread 0
init:
    # thread 0 will run main
    la a1, startup_state
    lw a0, 0(a1)
    la a0, main
    lw a1, 4(a1)
    j startup

# startup thread, assumes a0=func, a1=stack address
startup:
    # Ignored: 0 to all registers
    # Ignored: 0 .bss (would require 2 more arguments)
    # initialize the stack
    mv sp, a1
    # run C program
    jalr ra, a0, 0
    # when done, behavior depends on thread
    csrr a1, hartid
    # for all threads except thread 0, sleep and back to reset on wake
    beqz a1, done
    la a2, startup_state
    slli a1, a1, 3
    add a2, a2, a1
    sw x0, 0(a2)
    #TODO: thread sleep
    j reset
    # for thread 0, indicate pass/fail by setting tohost to return value from program (if 0, set to 15; ends simulation)
done:
    bnez a0, 1f
    addi a0, a0, 15
    1: csrw tohost, a0
    2: j 2b
```
# Format is t0_PC, t0_stack, t1_PC, t1_stack, ...

.data
.global startup_state
startup_state:
  .word main
  .word startup_state+0x0FFC
  .word 0
  .word startup_state+0x1FFC
  .word 0
  .word startup_state+0x2FFC
  .word 0
  .word startup_state+0x3FFC
## APPENDIX D: CONTROL AND STATUS REGISTERS

<table>
<thead>
<tr>
<th>CSR Address</th>
<th>Label</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td>fflags</td>
<td>internal clock</td>
</tr>
<tr>
<td>0x2</td>
<td>frm</td>
<td></td>
</tr>
<tr>
<td>0x3</td>
<td>fcsr</td>
<td></td>
</tr>
<tr>
<td>0xc0</td>
<td>stats</td>
<td></td>
</tr>
<tr>
<td>0x500</td>
<td>sup0</td>
<td></td>
</tr>
<tr>
<td>0x501</td>
<td>sup1</td>
<td></td>
</tr>
<tr>
<td>0x502</td>
<td>epc</td>
<td></td>
</tr>
<tr>
<td>0x503</td>
<td>badvaddr</td>
<td>scheduler slots</td>
</tr>
<tr>
<td>0x504</td>
<td>ptbr</td>
<td>hthread modes</td>
</tr>
<tr>
<td>0x505</td>
<td>asid</td>
<td>imem protection</td>
</tr>
<tr>
<td>0x506</td>
<td>count</td>
<td></td>
</tr>
<tr>
<td>0x507</td>
<td>compare</td>
<td></td>
</tr>
<tr>
<td>0x508</td>
<td>evec</td>
<td></td>
</tr>
<tr>
<td>0x509</td>
<td>cause</td>
<td></td>
</tr>
<tr>
<td>0x50a</td>
<td>status</td>
<td></td>
</tr>
<tr>
<td>0x50b</td>
<td>hartid</td>
<td></td>
</tr>
<tr>
<td>0x50c</td>
<td>impl</td>
<td></td>
</tr>
<tr>
<td>0x50d</td>
<td>fatc</td>
<td></td>
</tr>
<tr>
<td>0x50e</td>
<td>send_ipi</td>
<td></td>
</tr>
<tr>
<td>0x50f</td>
<td>clear_ipi</td>
<td></td>
</tr>
<tr>
<td>0x51d</td>
<td>reset</td>
<td></td>
</tr>
<tr>
<td>0x51e</td>
<td>tohost</td>
<td></td>
</tr>
<tr>
<td>0x51f</td>
<td>fromhost</td>
<td></td>
</tr>
<tr>
<td>0x586</td>
<td>counth</td>
<td></td>
</tr>
<tr>
<td>0xc00</td>
<td>cycle</td>
<td></td>
</tr>
<tr>
<td>0xc01</td>
<td>time</td>
<td></td>
</tr>
<tr>
<td>0xc02</td>
<td>instret</td>
<td></td>
</tr>
<tr>
<td>0xc80</td>
<td>cycleh</td>
<td></td>
</tr>
<tr>
<td>0xc81</td>
<td>timeh</td>
<td></td>
</tr>
<tr>
<td>0xc82</td>
<td>instreth</td>
<td></td>
</tr>
<tr>
<td>0xcc0</td>
<td>uarch0</td>
<td></td>
</tr>
<tr>
<td>0xcc1</td>
<td>uarch1</td>
<td>gpi_0</td>
</tr>
<tr>
<td>0xcc2</td>
<td>uarch2</td>
<td>gpi_1</td>
</tr>
<tr>
<td>0xcc3</td>
<td>uarch3</td>
<td>gpi_2</td>
</tr>
<tr>
<td>0xcc4</td>
<td>uarch4</td>
<td>gpi_3</td>
</tr>
<tr>
<td>0xcc5</td>
<td>uarch5</td>
<td>gpo_0</td>
</tr>
<tr>
<td>0xcc6</td>
<td>uarch6</td>
<td>gpo_1</td>
</tr>
<tr>
<td>0xcc7</td>
<td>uarch7</td>
<td>gpo_2</td>
</tr>
<tr>
<td>0xcc8</td>
<td>uarch8</td>
<td>gpo_3</td>
</tr>
<tr>
<td>0xcc9</td>
<td>uarch9</td>
<td></td>
</tr>
<tr>
<td>0xccb</td>
<td>uarch10</td>
<td></td>
</tr>
<tr>
<td>0xccc</td>
<td>uarch11</td>
<td></td>
</tr>
<tr>
<td>0xccd</td>
<td>uarch12</td>
<td></td>
</tr>
<tr>
<td>0xccf</td>
<td>uarch13</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>uarch14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>uarch15</td>
</tr>
</tbody>
</table>
APPENDIX E: SYNTHESIS MESSAGES

Info: Instantiated megafucntion "Core:flexpret|DSpm:dmem|dspm_dpram:dscratchpad|altsyncram:altsyncram_component" with the following parameter:
Info: Parameter "address_reg_b" = "CLOCK0"
Info: Parameter "clock_enable_input_a" = "BYPASS"
Info: Parameter "clock_enable_input_b" = "BYPASS"
Info: Parameter "clock_enable_output_a" = "BYPASS"
Info: Parameter "clock_enable_output_b" = "BYPASS"
Info: Parameter "init_file" = "./Users/Leonel/Documents/KTH/Exjobb/FlexPRET/fpga/generated-src/4tf-161-16d/demo_dspm.mif"
Info: Parameter "intended_device_family" = "Cyclone II"
Info: Parameter "lpm_type" = "altsyncram"
Info: Parameter "numwords_a" = "4096"
Info: Parameter "numwords_b" = "4096"
Info: Parameter "operation_mode" = "DUAL_PORT"
Info: Parameter "outdata_aclr_b" = "NONE"
Info: Parameter "outdata_reg_b" = "CLOCK0"
Info: Parameter "power_up_uninitialized" = "FALSE"
Info: Parameter "ram_block_type" = "NMH"
Info: Parameter "readcontrol_reg_b" = "CLOCK0"
Info: Parameter "read期间 write mode mixed ports" = "DONT_CARE"
Info: Parameter "width_a" = "12"
Info: Parameter "width_b" = "12"
Info: Parameter "width_a" = "32"
Info: Parameter "width_b" = "32"
Info: Parameter "width_byteena_a" = "4"
Info: Parameter "width_byteena_b" = "4"
Info: Parameter "wrcontrol_wraddress_reg_b" = "CLOCK0"
Info: Found 1 design units, including 1 entities, in source file db/altsyncram_lt62.tdf
Info: Found entity 1: altsyncram_lt62
Info: Elaborating entity "altsyncram_lt62" for hierarchy "Core:flexpret|DSpm:dmem|dspm_dpram:dscratchpad|altsyncram:altsyncram_component|altsyncram_lt62:auto_generated"

Info: Instantiated megafucntion "Core:flexpret|DSpm:dmem|dspm_dpram:dscratchpad|altsyncram:altsyncram_component" with the following parameter:
Info: Parameter "address_reg_b" = "CLOCK0"
Info: Parameter "byteena_reg_b" = "CLOCK0"
Info: Parameter "byte_size" = "8"
Info: Parameter "clock_enable_input_a" = "BYPASS"
Info: Parameter "clock_enable_input_b" = "BYPASS"
Info: Parameter "clock_enable_output_a" = "BYPASS"
Info: Parameter "clock_enable_output_b" = "BYPASS"
Info: Parameter "indata_reg_b" = "CLOCK0"
Info: Parameter "init_file" = "./Users/Leonel/Documents/KTH/Exjobb/FlexPRET/fpga/generated-src/4tf-161-16d/demo_dspm.mif"
Info: Parameter "intended_device_family" = "Cyclone IV E"
Info: Parameter "lpm_type" = "altsyncram"
Info: Parameter "numwords_a" = "4096"
Info: Parameter "numwords_b" = "4096"
Info: Parameter "operation_mode" = "BIDIR_DUAL_PORT"
Info: Parameter "outdata_aclr_a" = "NONE"
Info: Parameter "outdata_aclr_b" = "NONE"
Info: Parameter "outdata_reg_a" = "CLOCK0"
Info: Parameter "outdata_reg_b" = "CLOCK0"
Info: Parameter "power_up_uninitialized" = "FALSE"
Info: Parameter "ram_block_type" = "M9K"
Info: Parameter "read期间 write mode mixed ports" = "DONT_CARE"
Info: Parameter "read期间 write mode port_a" = "NEW_DATA_NO_NBE_READ"
Info: Parameter "read期间 write mode port_b" = "NEW_DATA_NO_NBE_READ"
Info: Parameter "width_a" = "12"
Info: Parameter "width_b" = "12"
Info: Parameter "width_a" = "32"
Info: Parameter "width_b" = "32"
Info: Parameter "width_byteena_a" = "4"
Info: Parameter "width_byteena_b" = "4"
Info: Parameter "wrcontrol_wraaddress_reg_b" = "CLOCK0"
Info: Found 1 design units, including 1 entities, in source file db/altsyncram_oh03.tdf
Info: Found entity 1: altsyncram_oh03
Info: Elaborating entity "altsyncram_oh03" for hierarchy "Core:flexpret|Datapath:datapath|RegisterFile:regfile|regfile"

Info: Found 1 instances of uninferrned RAM logic
Info: RAM logic "Core:flexpret|Datapath:datapath|RegisterFile:regfile|regfile" is uninferrned due to asynchronous read logic
Info: Generating hard block partition "hard_block:auto_generated_inst"
Info: Implemented 15567 device resources after synthesis - the final resource count might be different
Info: Implemented 11 input pins
Info: Implemented 9 output pins
Info: Implemented 15254 logic cells
Info: Implemented 290 RAM segments
Info: Quartus II 64-Bit Analysis & Synthesis was successful. 0 errors, 120 warnings
Info: Running Quartus II 64-Bit Fitter
Info: Low junction temperature is 0 degrees C
Info: High junction temperature is 85 degrees C
Info: Design uses memory blocks. Violating setup or hold times of memory block address registers for either read or write operations could cause memory contents to be corrupted. Make sure that all memory block address registers meet the setup and hold time requirements.
Info: Evaluating HDL-embedded SDC commands
  Info: Entity sid_hub
  Info: create_clock -period 10MHz -name altera_reserved_tck [get_ports {altera_reserved_tck}]
  Info: set_clock_groups -asynchronous -group {altera_reserved_tck}
Info: Detected timing requirements -- optimizing circuit to achieve only the specified requirements
Info: Found 2 clocks
  Info: Period  Clock Name
  Info: ========= =============
  Info:  100.000 altera_reserved_tck
  Info:  20.000  Sys_clk

Info: Fitter placement operations beginning
Info: Fitter placement was successful
Info: Running Quartus II 64-Bit Fitter was successful. 0 errors, 8 warnings
Info: Quartus II 64-Bit TimeQuest Timing Analyzer
Info: Low junction temperature is 0 degrees C
Info: High junction temperature is 85 degrees C
Info: Analyzing Slow 1200mV 85C Model
Info: Worst-case setup slack is 5.697
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  3.697  0.000  Sys_clk
  Info:  43.361  0.000  altera_reserved_tck
Info: Worst-case hold slack is 0.306
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  0.306  0.000  Sys_clk
  Info:  0.403  0.000  altera_reserved_tck
Info: Worst-case recovery slack is 48.317
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  48.317  0.000  altera_reserved_tck
Info: Worst-case removal slack is 1.601
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  1.601  0.000  altera_reserved_tck
Info: Worst-case minimum pulse width slack is 9.547
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  9.547  0.000  Sys_clk
  Info:  49.485  0.000  altera_reserved_tck
Info: Analyzing Slow 1200mV 0C Model
Info: Worst-case setup slack is 5.002
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  5.002  0.000  Sys_clk
  Info:  44.049  0.000  altera_reserved_tck
Info: Worst-case hold slack is 0.320
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  0.320  0.000  Sys_clk
  Info:  0.354  0.000  altera_reserved_tck
Info: Worst-case recovery slack is 48.583
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  48.583  0.000  altera_reserved_tck
Info: Worst-case removal slack is 0.906
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  0.906  0.000  altera_reserved_tck
Info: Worst-case minimum pulse width slack is 9.556
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  9.556  0.000  Sys_clk
  Info:  49.398  0.000  altera_reserved_tck
Info: Analyzing Fast 1200mV 0C Model
Info: Worst-case setup slack is 11.556
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info: 11.556  0.000  Sys_clk
  Info:  47.024  0.000  altera_reserved_tck
Info: Worst-case hold slack is 0.122
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  0.122  0.000  Sys_clk
  Info:  0.179  0.000  altera_reserved_tck
Info: Worst-case recovery slack is 49.514
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  49.514  0.000  altera_reserved_tck
Info: Worst-case removal slack is 0.489
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
  Info:  0.489  0.000  altera_reserved_tck
Info: Worst-case minimum pulse width slack is 9.199
  Info: Slack End Point TNS Clock
  Info: ========= ============= =============
Critical Warning: Partially connected in-system debug instance "auto_signaltap_0" to 437 of its 454 required data inputs, trigger inputs, acquisition clocks, and dynamic pins. There were 0 Illegal, 0 Inaccessible, and 17 missing sources or connections.

Critical Warning: The following clock transfers have no clock uncertainty assignment. For more accurate results, apply clock uncertainty assignments or use the derive_clock_uncertainty command.
  - Critical Warning: From altera_reserved_tck (Rise) to alteraReserved_tck (Rise) (setup and hold)
  - Critical Warning: From alteraReserved_tck (Fall) to alteraReserved_tck (Fall) (setup and hold)
  - Critical Warning: From Sys_clk (Rise) to Sys_clk (Rise) (setup and hold)

Critical Warning: The following clock transfers have no clock uncertainty assignment. For more accurate results, apply clock uncertainty assignments or use the derive_clock_uncertainty command.
  - Critical Warning: From alteraReserved_tck (Rise) to alteraReserved_tck (Rise) (setup and hold)
  - Critical Warning: From alteraReserved_tck (Rise) to alteraReserved_tck (Fall) (setup and hold)
  - Critical Warning: From Sys_clk (Rise) to Sys_clk (Rise) (setup and hold)

Critical Warning: The following clock transfers have no clock uncertainty assignment. For more accurate results, apply clock uncertainty assignments or use the derive_clock_uncertainty command.
  - Critical Warning: From alteraReserved_tck (Rise) to alteraReserved_tck (Rise) (setup and hold)
  - Critical Warning: From alteraReserved_tck (Rise) to alteraReserved_tck (Fall) (setup and hold)
  - Critical Warning: From Sys_clk (Rise) to Sys_clk (Rise) (setup and hold)

Critical Warning: The following clock transfers have no clock uncertainty assignment. For more accurate results, apply clock uncertainty assignments or use the derive_clock_uncertainty command.
  - Critical Warning: From alteraReserved_tck (Rise) to alteraReserved_tck (Rise) (setup and hold)
  - Critical Warning: From alteraReserved_tck (Rise) to alteraReserved_tck (Fall) (setup and hold)
  - Critical Warning: From Sys_clk (Rise) to Sys_clk (Rise) (setup and hold)