T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 2.3 Hardware Implementation of Patmos

Version 1.1
20 September 2012
Final

Public Distribution

Technical University of Denmark

Project Partners: **AbsInt Angewandte Informatik**, **Eindhoven University of Technology**, **GMVIS Skysoft**, **Intecs**, **Technical University of Denmark**, **The Open Group**, **University of York**, **Vienna University of Technology**

Project Partner Contact Information

AbsInt Angewandte Informatik Christian Ferdinand Science Park 1 66123 Saarbrücken, Germany Tel: +49 681 383600 Fax: +49 681 3836020 E-mail: ferdinand@absint.com	**Eindhoven University of Technology** Kees Goossens Potentiaal PT 9.34 Den Dolech 2 5612 AZ Eindhoven, The Netherlands E-mail: k.g.w.goossens@tue.nl
GMVIS Skysoft João Baptista Av. D. Joao II, Torre Fernao Magalhaes, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: joao.baptista@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail: silvia.mazzini@intecs.it
Technical University of Denmark Martin Schoeberl Richard Petersens Plads 2800 Lyngby, Denmark Tel: +45 45 25 37 43 Fax: +45 45 93 00 74 E-mail: masca@imm.dtu.dk	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail: s.hansen@opengroup.org
University of York Neil Audsley Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325 500 E-mail: Neil.Audsley@cs.york.ac.uk	**Vienna University of Technology** Peter Puschner Treitlstrasse 3 1040 Vienna, Austria Tel: +43 1 58801 18227 Fax: +43 1 58801 918227 E-mail: peter@vmars.tuwien.ac.at

Contents

Document Control

Version	Status	Date
0.1	First draft	1 August 2012
0.2	Version for internal review	30 August 2012
0.3	Version for group review	7 September 2012
0.9	Eindhoven meeting	14 September 2012
1.0	Final version	18 September 2012
1.1	Update SDRAM integration status	20 September 2012

Executive Summary

This document describes the deliverable *D 2.3 Hardware Implementation of Patmos* of work package 2 of the T-CREST project, due 12 months after project start as stated in the Description of Work. This document presents the design and implementation of an FPGA prototype of the base Patmos processor.

1 Introduction and Background

Current processors are optimized for average case performance, often leading to a high worst-case execution time (WCET). Many architectural features that increase the average case performance are hard to be modeled for the WCET analysis. In this deliverable we present Patmos, a processor optimized for low WCET bounds rather than high average case performance. Patmos is a dual-issue, statically scheduled RISC processor. The instruction cache is organized as a method cache and the data cache is organized as a split cache in order to simplify the cache WCET analysis. To fill the dual-issue pipeline with enough useful instructions, Patmos relies on a customized compiler, which is developed in the context of work package 5. The compiler also plays a central role in optimizing the application for the WCET instead of average case performance.

Real-time systems need a time-predictable execution platform so that the worst-case execution time (WCET) can be estimated statically. It has been argued that we have to rethink computer architecture for real-time systems instead of trying to catch up with new processors in the WCET analysis tools [18, 4, 25].

However, time-predictable architectures alone are not enough. If we would only be nterested in time predictability, we could use microprocessors from the late 1970s to the mid-1980s, where the execution time was accurately described in the data sheets. With those processors it would be possible to *generate* exact timing in software, e.g., one of the authors has programmed a wall clock on the Zilog Z80 in assembler by counting instruction clock cycles and inserting delay loops and nops at the correct locations.

Processors for future embedded systems need to be time-predictable *and* provide a reasonable worst-case performance [20]. Therefore, we present a Very Long Instruction Word (VLIW) pipeline with specially designed caches to provide good single thread performance. We intend to build a chip-multiprocessor using this VLIW pipeline to investigate its benefits for multi-threaded applications.

We present the time-predictable processor Patmos as one approach to attack the complexity issue of WCET analysis. Patmos is a statically scheduled, dual-issue RISC processor that is optimized for real-time systems. Instruction delays are well defined and visible through the instruction set architecture (ISA). This property simplifies the WCET analysis tool and helps to reduce the overestimation caused by imprecise information. Memory hierarchies having multiple levels of caches typically pose a major challenge for the WCET analysis. We attack this issue by introducing caches that are specifically designed to support WCET analysis. For instructions we adopt the method cache, as proposed in [15], which operates on whole functions/methods and thus simplifies the modeling for WCET analysis. Furthermore, we propose a split cache architecture for data [17, 22], offering dedicated caches for the stack area, for constants and static data, as well as for heap allocated objects. A compiler-managed scratchpad memory provides additional flexibility. Specializing the cache structure to the usage patterns of its data allows predictable and effective caching of that data, while at the same time facilitating WCET analysis.

The processor and its software environment is intended as a platform to explore various time-predictable design trade-offs and their interaction with WCET analysis techniques as well as WCET-aware compilation. We propose the co-design of time-predictable processor features with the WCET analysis tool, similar to the work by Huber et al. [8] on caching of heap allocated objects in a Java processor.

Within T-CREST, the co-design of processor features and their WCET analysis is performed, based in the work in work package 6, in Task 2.4 (M21-M24): "WCET Driven Design Space Exploration". Only features where we can provide a static program analysis shall be added to the processor. This includes, but is not limited to, time-predictable caching mechanisms, chip-multiprocessing (CMP) with a time predictable network-on-chip [21] (WP 3), as well as novel pipeline organizations. Patmos is open-source under the Simplified BSD License. [1]

The presented processor is named after the Greek island Patmos, where the first sketches of the architecture have been drawn; not in sand, but in a (paper) notebook. If you use the open-source design of Patmos for further research, we would suggest that you visit and enjoy the island Patmos. Consider writing a postcard from there to the T-CREST group.

1.1 Related Work

Edwards and Lee argue: "It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function" [4]. A first simulation of their PRET architecture is presented in [10]. PRET implements a RISC pipeline and performs chip-level multi-threading for six threads to eliminate data forwarding and branch prediction. Scratchpad memories are used instead of instruction and data caches. The shared main memory is accessed via a time-division multiple access (TDMA) scheme, called memory wheel. The ISA is extended with a *deadline* instruction that stalls the current thread until the deadline is reached. This instruction is used to perform time-based, instead of lock-based, synchronization for accesses to shared data. Furthermore, it has been suggested that the multi-threaded pipeline explores pipelined access to DRAM memories [3]. Each thread is assigned its own memory bank. The actual version of the PRET architecture supports the ARM instruction set and it called PTARM [11, 12].

Thiele and Wilhelm argue that a new research discipline is needed for time-predictable embedded systems [25]. Berg et al. identify the following design principles for a time-predictable processor: "... recoverability from information loss in the analysis, minimal variation of the instruction timing, non-interference between processor components, deterministic processor behavior, and comprehensive documentation" [2]. The authors propose a processor architecture that meets these design principles. The processor is a classic five-stage RISC pipeline with minimal changes to the instruction set. Suggestions for future architectures of memory hierarchies are given in [29].

Time-predictable architectural features have been explored in the context of the Java processor JOP [16]. The pipeline and the microcode, which implements the instruction set of the Java Virtual Machine, have been designed to avoid timing dependencies between bytecode instructions. JOP uses split load instructions to partially hide memory latencies. Caches are designed to be time-predictable and analyzable [15, 17, 23, 8]. With Patmos we will leverage on our experience with JOP and implement a similar, but more general, cache structure.

Heckmann et al. provide examples of problematic processor features in [7]. The most problematic features found are the replacement strategies for set-associative caches. In conclusion Heckmann et al. suggest the following restrictions for time-predictable processors: (1) separate data and instruction caches; (2) locally deterministic update strategies for caches; (3) static branch prediction; and (4)

[1] `https://github.com/t-crest/patmos`

limited out-of-order execution. The authors argue for restriction of processor features. In contrast, we also provide additional features for a time-predictable processor.

Whitham argues that the execution time of a basic block has to be independent of the execution history [27]. To reduce the WCET, Whitham proposes to implement the time critical functions in microcode on a reconfigurable function unit (RFU). With several RFUs, it is possible to explicitly exploit instruction level parallelism (ILP) of the original RISC code – similar to a VLIW architecture.

Superscalar out-of-order processors can achieve higher performance than in-order designs, but are difficult to handle in WCET analysis. Whitham and Audsley present modifications to out-of-order processors to achieve time-predictable operation [28]. Virtual traces allow static WCET analysis, which is performed before execution. Those virtual traces are formed within the program and constrain the out-of order scheduler built into the CPU to execute deterministically.

Multi-Core Execution of Hard Real-Time Applications Supporting Analyzability (MERASA) [26] is a European Union project that aims for multicore processor designs in hard real-time embedded systems. An in-order superscalar processor is adapted for chip multi-threading (CarCore) [13]. The resulting CarCore is a two-way, five-stage pipeline with separated address and data paths. This architecture allows issuing an address and an integer instruction within one cycle, even if they are data-dependent. CarCore supports a single hard real-time thread to be executed with several non-real-time threads running concurrently in the background.

In contrast to the PRET and CarCore designs we use a VLIW approach instead of chip-level multi-threading to utilize the hardware resources. To benefit from thread-level applications we will replicate the simple pipeline to build a CMP system. For time-predictable multi-threading almost all resources (e.g., thread local caches) need to be duplicated. Therefore, we believe that a CMP system is more efficient than chip multi-threading.

1.2 FPGA Technology and On-Chip Memories

Current FPGAs consist of following building blocks: logic cells, on-chip memories, and often hardware multipliers. A logic cell (LC) consists of a 4 or 6-bit lookup table (LUT) and a flip-flop. With LCs any digital circuit can be built. However, building larger storage elements out of LCs is very costly. Therefore, basically all available FPGA architectures offer on-chip memory blocks. Common sizes of those on-chip memories are 4 KBit, 8 KBit, and 16 KBit. The data width is usually configurable in powers of 2 between 1 and 32 bits.

The question is now how much memory and LC resources are available for a design. It is interesting that current low-cost FPGAs from Xilinx and Altera (Spartan and Cyclone series) settle for a logic to memory ration of 200–400 LCs per on-chip memory block [19]. This ratio is, quite surprising, constant relative to the number of memory blocks, not the memory capacity. Therefore, for a plain RISC pipeline, which might be implemented in about 3000 LCs, the optimal memory consumption is about 10 blocks of on-chip memories. This is equivalent, dependent on the device family, 5, 10, or 20 KB of available on-chip memory. This is a very small number when register files, local scratchpad memories, communication memories, and various caches consume on-chip memory blocks. It is easy to see that the on-chip memory will be the main resource bottleneck for a T-CREST platform in an FPGA.

Current FPGAs only support synchronous on-chip memories. That means that the read and write addresses, the write data and the write enable are registered.[2] Therefore, the registers at the memory input signals are part of the pipeline registers. Furthermore, these registers cannot be reset and cannot be read out. If the value of such a register needs to be read back (e.g., representing the program counter (PC)), it has to be duplicated with normal LC based registers.

A further implication of having part of the pipeline registers in the on-chip memories is a less structured implementation. We intend to encapsulate each pipeline stage in a VHDL component, with an input from the former stage, and as output the pipeline register. However, if the next stage uses an on-chip memory, we also have to provide the unregistered output of the pipeline stage. As an example consider the fetch stage: an instruction is fetched from the instruction memory (scratchpad, cache, or method cache) and stored in the instruction register. This instruction register is the pipeline register for the fetch stage. However, during the decode stage also the register file is read. When the register file is implemented using on-chip memories, it needs the combinational output from the fetch stage.

2 The Architecture of Patmos

Patmos is a 32-bit, RISC-style microprocessor optimized for time-predictable execution of real-time applications [24]. In order to provide high performance for single-threaded code, a two-way parallel VLIW architecture was chosen. For multi-threaded code, the T-CREST chip-multiprocessor system, with statically scheduled accesses to shared resources, will be explored for time-predictable performance enhancements.

2.1 Overview

Patmos is a statically scheduled, dual-issue RISC microprocessor. The processor does not stall, except for instructions that wait for data from the main memory. All instruction delays are thus explicitly visible at the ISA-level, and the exposed delays from the pipeline need to be respected in order to guarantee correct and efficient code. Programming Patmos is consequently more demanding than for usual processors. However, knowing all delays and the conditions under which they occur simplifies the processor model required for WCET analysis and helps to improve accuracy.

The modeling of memory hierarchies with multiple levels of caches is critical for practical WCET analysis. Patmos simplifies this tasks by offering caches that are especially designed for WCET analysis. Accesses to different data areas are quite different with respect to WCET analysis. Static data, constants, and stack allocated data can easily be tracked by static program analysis. Heap allocated data on the other hand demands for different caching techniques to be analyzable [9]. Therefore, Patmos contains several data caches, one for each memory area. Furthermore, we will explore the benefits of compiler managed scratchpad memory. [3]

[2] The read data output can also be registered for maximum clock frequency, but need not to be.
[3] The exploration of the split cache will be the topic of the following Task 2.3.

The primary implementation technology is in a field-programmable gate array (FPGA). Therefore, the design is optimized within the technology constraints of an FPGA. Nevertheless, features such as preinitialized on-chip memories are avoided to keep the design implementable in ASIC technologies.

The main features of Patmos are:

- Dual-issue pipeline
- Full predication
- Split caches with typed load/store instructions
- Split load

2.2 Software Development with Patmos

The architecture design of Patmos adopts ideas from the RISC and VLIW design philosophies. In particular, the idea that architecture design is *interdependent* on the software development environment. The first RISC machines made some architectural constraints visible on the instruction set level in order to push complexity from the hardware design to the software tools or programmer. The VLIW philosophy took this idea even further and assigned the compiler a central role in exploiting the available hardware resources in the best possible way [6].

We make the case that this architecture philosophy is particularly suited to address the problems encountered in today's real-time system design. Time-predictable architectures following this approach, such as Patmos, not only unveil optimization potential to the compiler, but more importantly provide the opportunity for developing more accurate program analyses, e.g., in order to derive tighter bounds for the WCET. The compiler and the program analysis tools are thus first class citizens of the real-time system engineer's toolbox and need to be accounted for in the architecture design. As a side-effect the use of high-level programming languages is facilitated or even favored, since the necessary software tools are readily provided.

2.3 Exploiting Patmos' Features

Some design decisions for Patmos are based on a pragmatic assumption that the engineer best knows the system under development. It is thus important to enable the programmer to fine tune the system. Care has been taken that those features are *accessible* from high-level programming languages. The typed memory loads and stores are a good example of such a feature, which allows the programmer to explicitly assign variables and data structures to specific memory areas. The typed memory operations are a natural match to named address spaces in Embedded C, an extension of the traditional C language. The computation of tight WCET bounds is simplified, since the target memory is apparent from the operation itself. The tedious tracking of possible pointer ranges is thus avoided.

The stack cache provides a time-predictable and analyzable way to reduce the penalty for accessing objects residing on the stack frame of the current function. For most functions it is trivial for the compiler to immediately exploit the stack cache. Special care has to be taken that function-local variables accessible through pointers are not placed in the cache, because the cache's memory is not accessible using regular memory operations. Those variables need to be kept in a *shadow stack*

residing in general purpose memory. Note that other variables of the same function are nevertheless assigned to the stack cache.

Exploiting the method cache is more involved and requires a global analysis of the complete real-time program, including all external modules and libraries linked to it. Using a regular call graph we can determine function calls potentially leading to conflicts in the cache and adopt the placement of the involved functions accordingly. Similar techniques have successfully been applied in the context of scratchpad memories and overlay memories [5]. The design of Patmos' method cache, however, combines the predictability of a static code layout in a scratchpad memory with the flexibility of a cache.

The predicated instructions supported by Patmos allow the elimination of branches. This idea was first applied for wide-issue VLIW machines in order to keep the parallel execution units busy and avoid the expensive branch penalty. The single-path programming paradigm [14] adopts the very same idea to compute tighter WCET bounds. While it is true that for a given single-path program the WCET bound is generally closer to the actual WCET, the actual WCET and its computable bound is *not* guaranteed to be better than for regular programs. The problem arises from the blind elimination of branches independent from their relevance to the WCET. We thus propose WCET-aware if-conversion and global scheduling in order to eliminate branches and exploit the parallel execution units of Patmos to actively reduce the WCET.

2.4 Instruction Set

The instruction set of Patmos follows the conventions of usual RISC machines such as MIPS. All instructions are fully predicated and take at most three register operands. Except for branch and accesses to main memory using loads or stores, all instructions can be executed by both pipelines.

The first instruction of an instruction bundle contains the length of the bundle (32 or 64 bits). Register addresses are at fixed positions to allow reading the register file parallel to instruction decoding. The main pressure on the instruction encoding comes from constant fields and branch offsets. Constants are supported in different ways. A few ALU instruction can be performed with a sign-extended 12-bit constant operand. Two instructions are available to load 16 bits into the lower (with sign extension) or upper half of a register. Furthermore, a 32-bit constant can be loaded into a register by using the second instruction slot for the constant. Branches (conditional and unconditional) are relative with a 22-bit offset. Function calls to a 32-bit address are supported by a register indirect branch and link instruction.

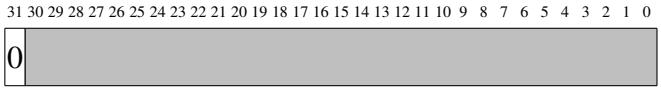
To reduce the number of conditional branches and to support the single-path programming paradigm [14], Patmos supports fully predicated instructions. Predicates are set with compare instructions, which themselves can be predicated. A complete set of compare instructions (two registers and register against 0) is supported. The optimum number of concurrently live predicates is still not settled, but the current prototype supports 8 predicates.

Access to the different types of data areas are explicitly encoded with the load and store instructions. This feature helps the WCET analysis to distinguish between the different data caches. Furthermore, it can be detected earlier in the pipeline which cache will be accessed. A detailed list of the instructions can be found in the Patmos simulation deliverable D 2.1.

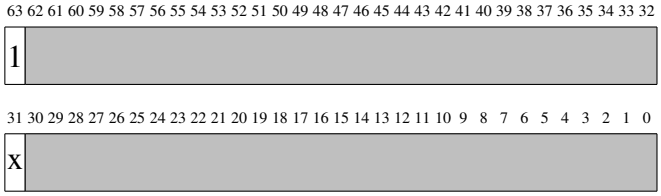
2.4.1 Bundle Formats

Most Patmos instructions are 32 bits wide and are structured according to one of the instruction formats defined in the following section. Up to two instructions can be combined to form an instruction bundle; Patmos bundles are thus either 32 or 64 bits wide. The bundles sizes are recognized by the value of the most significant bit, where 0 indicates a short, 32-bit bundle and 1 a long, 64-bit bundle. Furthermore, a 64-bit bundle can be used to support a single ALU operation with a 32-bit constant (in the second word). The following figures illustrate these two bundle variants:

- 32-bit bundle format



- 64-bit bundle format



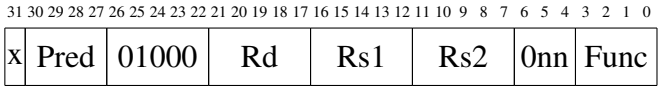
2.4.2 Instruction Types

Patmos supports following instruction types: ALU operations with register operands, ALU operations with short constants, ALU operations with long constants, compare, predicate combination, typed load and store instructions, branch instructions, and cache related special instructions.

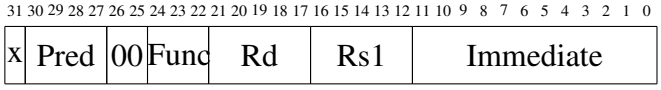
The detailed list of the Patmos instructions is already listed in the appendix of the simulator deliverable (D 2.1) and therefore not repeated in this document.

ALU Instructions ALU instructions can be register/register instructions or register/immediate instructions. The immediate instructions come in two flavors: one with a short immediate, which supports only a subset of the ALU operations, and one with a long immediate, which consumes two issue slots. The instruction formats for ALU instructions are:

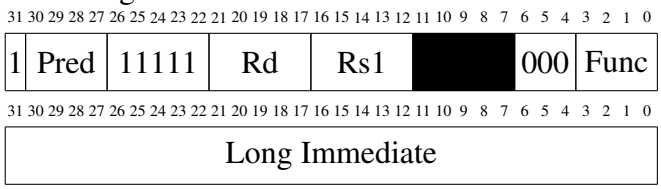
- ALU register



- ALU immediate

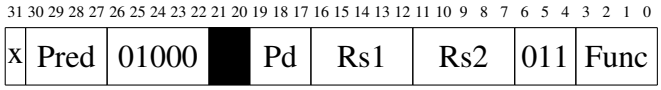


- ALU long immediate

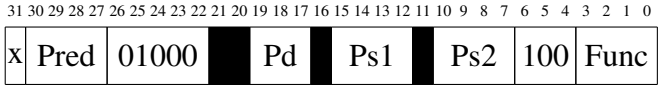


Compare and Predicate Instructions Compare instructions and predicate instructions set or reset a predicate. A compare instruction operates on the two source registers. The predicate instructions perform operations on the predicate register.

- Compare

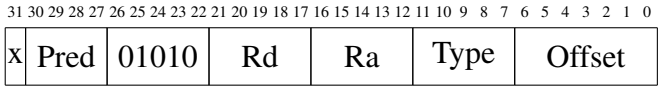


- Predicate

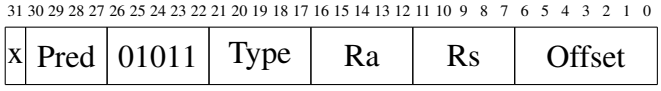


Load and Store Instructions Load and store instructions use register indirect addressing with an offset. 32-bit word, 16-bit half word, and byte access is supported. Furthermore, several different types of memory areas are distinguished by different load and store instructions. The instruction format is as follows:

- Load



- Store



2.5 The Pipeline

The register file with 32 registers is shared between the two pipelines. Full forwarding between the two pipelines is supported. The basic features are similar to a standard RISC pipeline. The data cache is split into different cache areas. The distinction between the different caches is performed with typed load and store instructions.

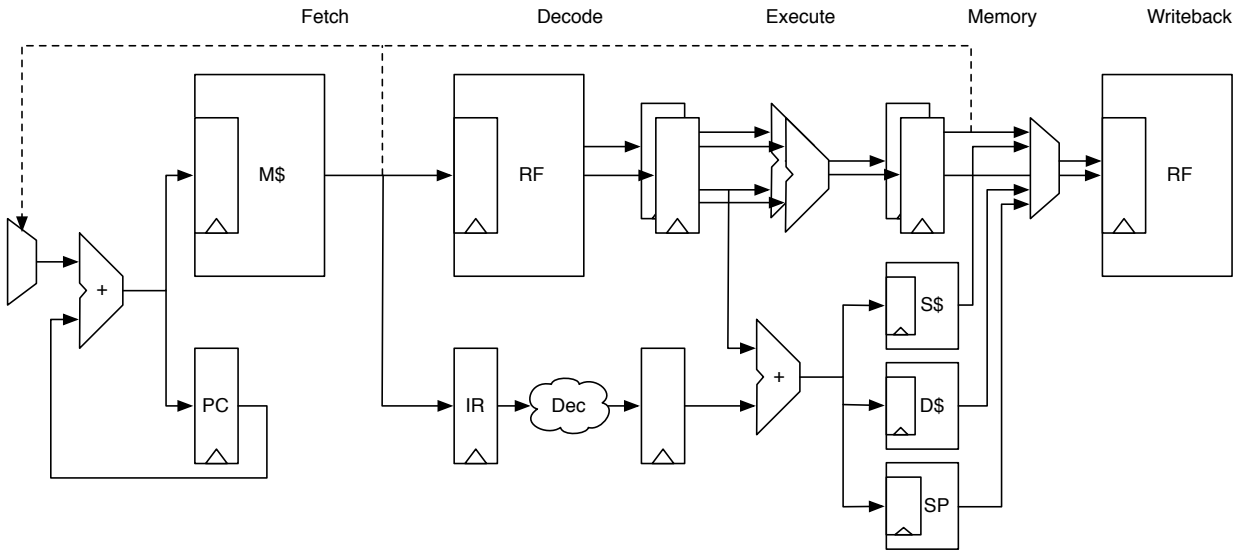


Figure 1: Pipeline of Patmos with fetch, decode, execute, memory, and write back stages. RF denotes the register file, which appears in the decode stage for reading and in the write back stage for writing. The on-chip memories shown are: method cache (M$), stack cache (S$), data cache (D$), and scratchpad memory (SP).

Figure 1 gives an overview of Patmos' pipeline. The pipeline consist of 5 stages: (1) instruction fetch (IF), (2) decode and register read (DR), (3) execute (EX), (4) memory access (MEM), and (5) register write back (WB). To simplify the diagram, forwarding and external memory access data paths are omitted and not all typed caches are shown. The method cache (M$), the register file (RF), the stack cache (S$), the data cache (D$), and the scratchpad memory (SP) are implemented in on-chip memories of an FPGA. All on-chip memories of Patmos use registered input ports. As the memory internal input registers can not be accessed, the program counter (PC) is duplicated with an explicit register. The instruction fetched from the method cache is stored in the instruction register (IR) and also used in the register file to fetch the register values during the decode stage.

For a dual-issue RISC, the RF needs four read ports and two write ports. Current FPGAs offer on-chip memories with one read and one write port. Additional read ports can be implemented by replicating the RF on several on-chip memories. However, to implement the dual write ports, the RF needs to be double clocked. To save resources, double clocking is also used for the read ports. The resulting RF needs *only* two block RAMs. As read during write at the same address in the on-chip memories of current FPGAs either delivers the old value on the read or an undefined value, the RF contains an internal forwarding path.

At the execution stage up to two operations are executed and the address for a memory access is calculated. Predicates are set on a compare instruction. The following stage performs memory/cache load or store and the last stage writes back the result from the execution or from a load operation.

The PC manipulation depends on two pipeline stages. The control signals for the next PC are sketched with dashed lines in Figure 1. At the fetch stage the single bit that determines the instruction length is fed to the PC multiplexer. A predicate for a conditional branch is calculated by a compare instruction. That predicate is set in the execute stage and available for the following instruction in the execute

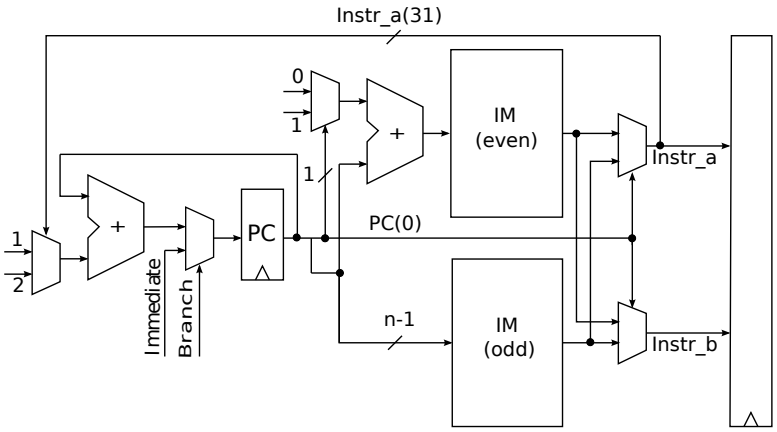


Figure 2: Fetch stage in Patmos

stage. The conditional branch is therefore executed in the excute stage, reading the predicate register (conceptually form the memory stage). Update of the PC in the execute stage results in two branch delay slots after the branch. An unconditional branch could be implemented by a feedback from the decode stage and would only insure a single branch delay slot. However, to keep the instruction set orthogonal, there are no unconditional branches.

2.5.1 Fetch

Patmos is a dual-issue processor. Thus the fetch stage in Patmos is quite different from a fetch stage in a single issue processor. It is capable of reading two instructions at the same cycle. Furthermore, Patmos supports instructions with long immediate constants (32-bit). ALU instructions with a long immediate use the second instruction slot for the immediate constant. A dual instruction fetch or long immediate instruction is identified by a one in bit position 31 of the instruction in the first issue slot. In the fetch stage, this bit is checked to determine if the PC will be incremented by one or two. To support variable length fetching in a single cycle, we use two memory blocks for the instruction memory. One provides the instruction words at even addresses and one the instruction words at odd addresses.

If the instruction bundle starts at an odd address, the address for the even memory needs to be incremented by one to support this unaligned instruction bundle. An odd instruction address has bit 0 of the PC set to 1 (The PC counts in instruction words, not in bytes.) Furthermore, the output multiplexers of the two memories depend on whether the instruction bundle is aligned or not. Figure 2 shows the principal function of the dual-issue fetch in Patmos. This figure assumes asynchronous memories for the instruction memory (IM) and only the PC and the instruction register (= pipeline register) are synchronous registers.

As discussed before, only synchronous on-chip memories are available in FPGAs. In the current state of Patmos implementation, on-chip memories are used as instruction memories. Therefore the memories should be synchronous which needs the registered signals going through the instruction memories to be replaced by their not registered version. Since in the current implementation the instruction memories are implemented as ROM, the only input which is modified is the address

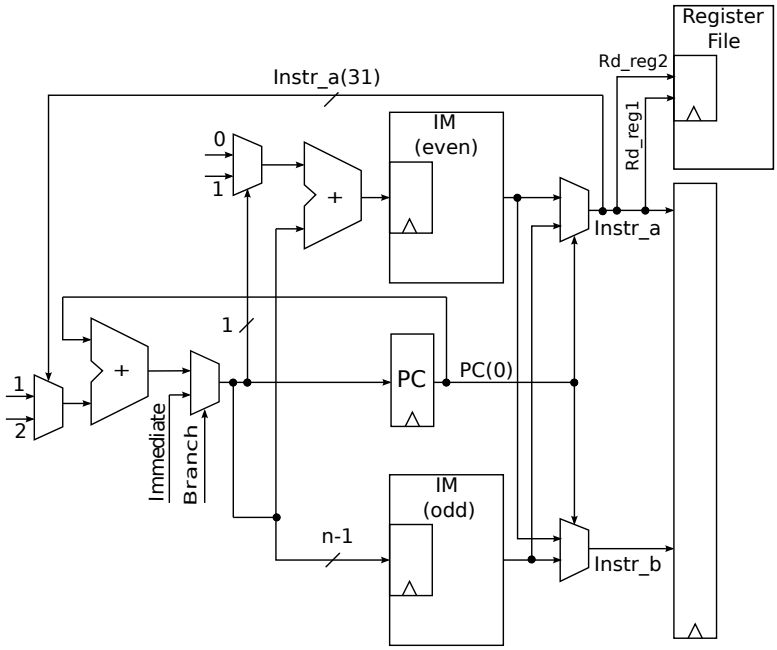


Figure 3: Concrete implementation of the fetch stage with on-chip memories with registered inputs.

signal. Additionally, the select bit of the multiplexer which decides on the odd increment for even addresses, uses the not registered value of PC(0). This idea is presented in Figure 3.

2.5.2 Decode

An instruction is decoded and the register file is read in the decode stage. Figure 4 shows the decode stage of a single-issue version of Patmos, with support for long immediates.

In the decode stage, the instruction fetched from the instruction cache is split into different parts. The type of each instruction is defined based on the bit positions [26:22]. In case of an ALUi instruction, bits [24:22] are used to define the function of the ALUi instructions (e.g. add, sub, etc.). Registers are read in this stage of the pipeline too. The first source register address of every instruction is located at bit positions [16:12] and the second source address (used only in a register/register ALU operation) is located at bit positions [11:7]. Otherwise, immediate values are defined in the instruction at bit positions [11:0]. In case of load instruction bit positions [11:7] define the type of the load, (e.g. from stack cache or scratchpad memory) and bit positions [6:0] provide the offset for the effective address. The destination register address is at bit positions [21:17] (if a destination is needed in the instruction).

Reading the register file in the decode stage is based on the combinational output of the instruction fetch, as the addresses are registered in the register file.

The decode stage is responsible for defining the control signals, which are piped to the following pipeline stages. As mentioned above, some of the instruction do not write into the register file (such as load instructions). This property is defined using a control signal to enable or disable writing into the register file. Also deciding where to read the second operand of each instruction from, is done

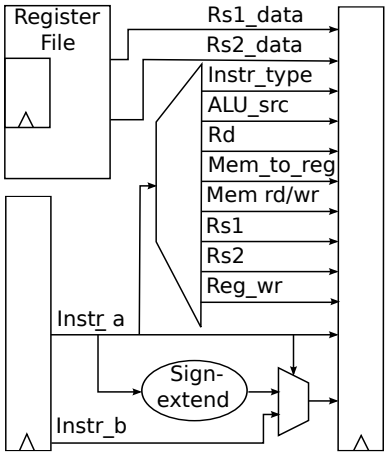


Figure 4: Decode stage in Patmos.

in the decode stage. This control signal is used in the next pipeline stage as select of a multiplexer. Also the value which has to be written to the register file can come from either memory (the load instruction) or from the execution stage. There is a single control signal in the decode stage output that represents this selection. Since writing to the memory is just performed in store instructions, another control signal is defined in the decode stage as memory enable which makes writing in the memory only possible in store instructions. Due to the existence of different types of loads and stores in the Patmos ISA, different control signals are defined in the decode stage to handle writing/reading to/from different types of memories.

One important feature of Patmos is conditional execution, which is provided through the predication bits. Predication bits are located in bit positions [30:27]. These bits are piped to the next stage too.

2.5.3 Execute

The execution stage is responsible for performing the desired operation on its operands based on the type of the instruction. In case of the memory operations, the desired operation is calculating the address. The execution stage is shown in Figure 5.

The ALU in the Patmos, as the main part of the execution unit, performs the operations based on the type of the instruction (provided by the signal Instr_type from decode stage) and the function code. For most of the instructions the operations are performed on the operands provided by the forwarding multiplexers. The inputs of the forwarding multiplexers come from three different sources. The highest priority is dedicated to the case where the registers which are read are the same as the destination register in the previous clock cycle (instruction). In this case the result is not yet written to the register file and needs to be forwarded. The second priority is if the destination register of two clock cycles before is the same as the one read in the present cycle. If none of the cases are valid, the operand is provided by the register file. The values for the first and second inputs of these multiplexers are depicted in the Figure 5 as Ex_wr_reg and Mem_wr_reg, which are the outputs of the execution stage and memory stage pipeline registers consecutively. Third inputs of multiplexers which are coming from register file are shown by Rs1_data and Rs2_data in the Figure 5. The select

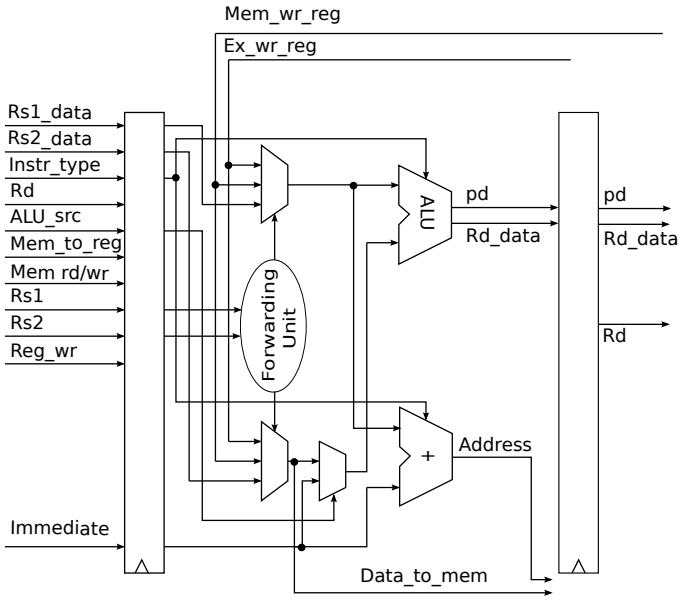


Figure 5: Execution stage in Patmos.

bits of the forwarding multiplexers are the result of comparison between number of the register which is being read (i.e. Rs1 and Rs2 in Figure 5) and the number of destination registers from outputs of the execution stage and memory stage pipeline registers. Case of forwarding from write back stage is handled in the register file (see Section 2.5.5). For the second operand of the ALU, the result of the forwarding multiplexer is fed to another multiplexer which chose whether the second operand of the ALU is the register value or the immediate value. Select bit of this multiplexer is provided by the ALU_src control signal generated in decode stage. The generated result of ALU instructions is depicted in Figure 5 by the Rd_data signal (which may later be written to the register file). For load and store instructions an adder is used to generate the address which is accessed by the instruction (shown as Address signal in the Figure 5). This signal does not go through the execution stage register since the on-chip synchronous memories are used to implement the memories at this stage of development. Although there are other sets of instructions which do not affect the Rd_data signal.

Compare instructions read their source operands from forwarding multiplexers but write to the predicate registers (which is presented by pd signal in Figure 5). Calculating the predicate for a branch instruction is a common example of usage of compare instructions. Another set of instructions which do not use the registers from the register file as destination (nor do they read their source operands from the register file) are predication instructions, e.g., *por* where the ALU operation is performed on predicate registers. The operation on these registers is affected by a negation bit which causes the value read from the predicate register to be negated and then used as an operand in the predicate instructions.

Patmos is capable of conditional execution based on predicate registers. This is accomplished through bits [30:27] in every instruction. The three lower bits represent the number of predicate register and the highest bit represents the negation. The difference imposed from the predicated execution is that controlling signals in the execution stage are only piped to the next stages if the predication is true. This means that control (enable) signals for writing to the memory or the register file only retain

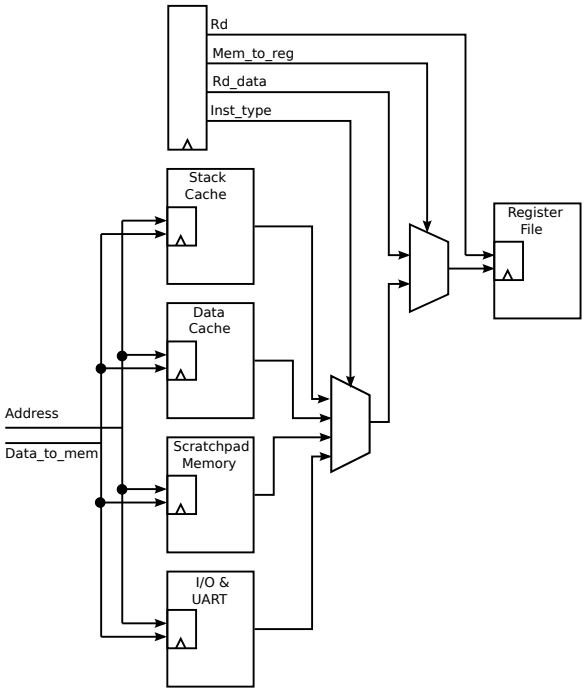


Figure 6: Memory stage in Patmos

their original values if the predication of the instruction is true, otherwise the write enable signal for memory or register file is not activated (this is equal to the instruction not being executed at all because the state of the processor would be intact).

2.5.4 Memory

The memory stage contains the local memories such as scratchpad memory as shown in Figure 6.

Since Patmos uses different types of loads and stores, each with different length, access to individual bytes of memory is required. This is accomplished through using four RAM modules with individual enable signals. In case of store instructions, the write enable signals of the RAMs are defined using a decoder based on type of the store instruction and the lower address bits. Load instructions read from different number of RAM blocks (i.e., 4, 2, or just one) using another decoder based on the type of the load instruction. At the current development state of the Patmos, the data-cache and the global memory are mapped to the scratchpad memory.

2.5.5 Write Back

The write back stage in Patmos performs the writing in to the register file. To avoid hazards in the write back stage, the register file provides the correct result if the instruction in the decode stage reads the same register as the register written by the instruction in the write back stage. Such a register file provides another form of forwarding based on the result of comparing the two mentioned

register numbers. If the registers are the same, the value which is being written to the register is read (forwarded) instead of the previous value of the register.

2.6 Memory and Caches

Access to main memory is done via a split load, where one instruction starts the memory read and another instruction explicitly waits for the result. Although this increases the number of instructions to be executed, instruction scheduling can use the split accesses to hide memory access latencies deterministically. For instruction caching a method cache is used where full functions/methods are loaded at call or return [15]. This cache organization simplifies the pipeline and the WCET analysis as instruction cache misses can only happen at call or return instructions. For the data cache a split cache is used [17]. Data allocated on the stack is served by a direct mapped stack cache, heap allocated data in a highly associative data cache, and constants and static data in a set associative cache. Only the cache for heap allocated data and static data needs a cache coherence protocol for a CMP configuration of Patmos. Furthermore, a scratchpad memory can also be used to store frequently accessed data. To distinguish between the different caches, Patmos implements typed load and store instructions. The type information is assigned by the compiler (e.g., the compiler already organizes the stack allocated data). To simplify Figure 1, only the stack and data cache are shown as an example of the split cache.

2.7 Processor State

The (visible) state of the processor consists of: the register file, the program counter (PC), the predicate register, the function base pointer, the stack cache pointers, and content of the scratchpad memory. The register file is read in the decode stage and written in the write back stage. However, full forwarding makes the to be written value available for following instructions in earlier pipeline stages. The predicate register is read in the execution stage and also updated in the execution stage. Therefore, no forwarding is needed for the predicate register. The program counter is read in the fetch stage. However, it is updated in two different stages: normal increment is performed in the fetch stage, whereas branches are performed in the execution stage. The reason to update the PC in the execution stage is that a changed predicate from an instruction before is available in the execution stage. The branch is followed by two branch delay slots, i.e., the following 2 (up to 5 in the dual-issue version) instructions are executed independent of the branch outcome.

3 Implementation Details

The basic pipeline of Patmos that we have implemented, can run in the ModelSim simulation and also executes in an FPGA. The prototype consumes about 2100 logic cells and can be clocked at 65 MHz in a Cyclone II FPGA. The prototype of Patmos is not optimized and we assume that there is room for improvement. Our target is a maximum clock frequency of 80% of a plain RISC pipeline (such as NIOS II [1] from Altera) for the dual-issue version of Patmos. In a low-cost FPGA, such

Address	Memory area
0x00000000–0x0fffffff	Scratchpad memories
0x10000000–0xdfffffff	Main memory
0xe0000000–0xefffffff	NoC interface and communication memory
0xf0000000–0xffffffff	I/O devices

Table 1: Address mapping

as Cyclone II, this may be in the range of 100 MHz, in a high-performance FPGA, such as Altera's Stratix or Xilinx's Virtex, the clock frequency may be in the range of 200 MHz.

3.1 Supported Instructions

The instruction set architecture (ISA) of Patmos is already listed in the appendix of the simulator (Deliverable D 2.1) and therefore omitted from this document. The cycle accurate simulator of Patmos is the reference for the hardware design. However, if some aspects/instructions become expensive or impractical in the hardware implementation, the simulator and the compiler needs to be changed. Furthermore, instruction usage by the compiler will guide future iterations of the ISA.

For the hardware prototype a single issue pipeline is implemented (the compiler currently has only a single issue backend). However, as long immediate instructions are needed, the fetch stage supports fetching of 32-bit and 64-bit bundles.

The cache and local memory system has been simplified to use only scratchpad style local memories. The design of time-predictable caches is part of Task 2.3, which starts in month 13 of the project.

3.2 Memory Mapping and I/O Devices

Main memory, SPMs, communication memory, and I/O devices are all accessed via memory load and store operations (some memories also support DMA transfers). To simplify address decoding, the top four bits (A31–A28) are used to distinguish between different memory and I/O areas. Table 1 shows this address mapping.

Within the I/O device memory area bits 11–8 are used to distinguish between different devices. I/O device registers are mapped and aligned to 32-bit words. If a register is shorter than a word, the upper bits shall be filled with 0 on a read. With this mapping each I/O device can have up to 64 32-bit registers.

In the initial prototype of Patmos we have 3 I/O devices: a UART for basic communication, LEDs on the FPGA board, and cycle counters. One counter ticks with the clock frequency and the second counter ticks with 1 MHz for clock frequency independent time measurements. Table 2 shows the I/O devices and the registers.

Address	I/O Device	read	write
0xf0000000	UART	status	control
0xf0000004	UART	receive buffer	transmit buffer
0xf0000100	counter	clock cycles	–
0xf0000101	counter	time in micro seconds	–
0xf0000200	LED	–	output register
0xf0000300	SDRAM		

Table 2: I/O devices and registers

3.3 SDRAM Access

For first experiments with the external SDRAM memory, TU/e has delivered an early version of the SDRAM controller (the first version is originally due in M18). The SDRAM controller is configured to support up to four processors and has a DTL-based interface to the processors.

For a first test we have added an I/O mapped interface between Patmos and the SDRAM controller. The I/O device provides Patmos memory mapped I/O registers with single cycle latency. Therefore, no pipeline stalling is needed. The I/O device is also connected to the SDRAM controller implementing the DTL interface via a small state machine.

The processor reads and writes to the memory via this I/O based indirection and polls for availability of data. A read request works as follows:

1. The read address is written to the I/O device,
2. the I/O device translates this to the DTL transactions towards the SDRAM controller,
3. the SDRAM controller performs the memory read and delivers the data back,
4. the I/O device reads the data into a buffer,
5. the processor polls the I/O device until the data is ready, and
6. reads the data.

A write indirection works as follows:

1. The write address is written to the I/O device,
2. the write data is written to the I/O device and triggers the write,
3. the I/O device translates this to the DTL transactions towards the SDRAM controller,
4. the SDRAM controller performs the memory write and delivers done back,
5. the I/O device reads the acknowledgement, and

6. the processor polls the I/O device until the acknowledgement is received.

The smallest operation supported by current memory controller is 64 bytes. Therefore, the writes of smaller sizes from the processor must be implemented as Read(full block)-Modify(relevant part)-Write(block). The current I/O device supports four memory operations: load/store of a 64-byte block and read/write of a single word.

The full write acknowledgment is currently not supported. There is an optional mechanism for it in the DTL specification (with tag and tag_ack signals), but it is not included in the interface of the controller.

Status The SDRAM controller supports a Xilinx FPGA board, whereas the initial version of Patmos has been developed on an Altera FPGA board. Minor changes have been applied to make Patmos compatible to the Altera and Xilinx FPGAs and make Patmos an EDK component. The SDRAM controller with the test driver and Patmos have been combined in the same FPGA sharing the clock and reset. The IO based interface between Patmos and the SDRAM has been developed and tested in the hardware with assembler based test programs.

3.4 Future Work

The next steps can be summarized as:

- Testing of Patmos with the full set of MiBench integer benchmarks
- Extension to the dual-issue architecture
- Design and implementation of various caches (method cache, stack cache,..) to evaluate the concept of a split-cache in Patmos [22]
- A paper on the stack cache
- Optimization of resource usage and maximum clock frequency
- An overview paper on the Patmos architecture
- Integration with the network-on-chip (Deliverable D 3.2)
- Integration with the memory controller (Deliverable D 4.2), which will be available in month 18 of the project

4 Programming and Testing

Patmos is intended to be programmed in C/C++. Therefore, the LLVM compiler is adapted in work package 5 to support the Patmos instruction set. For first and detailed tests of single instructions or small sequences it is easier and more convenient to write test programs in assembler.

4.1 Assembler Programming

Initial testing is performed with small assembler programs. A simple assembler is part of the Patmos simulator (Deliverable D 2.1), which is also used for the FPGA implementation of Patmos. In a

first step the plain binary, generated by `patasm`, is converted with a small tool (`Bin2Vhdl`) into a VHDL table. That table represents a read only memory (ROM) for the instruction memory.

Initial tests are performed during the development of the pipeline with ModelSim and "starring at the waveforms". Parallel to using the VHDL simulation with ModelSim the VHDL source is synthesized for an FPGA to avoid constructs that are not synthesizeable or generate latches. The scaled-up version of testing is performed via co-simulation with the cycle accurate Patmos simulator (Deliverable D 2.1).

4.2 C Programming

We spent not too much effort in writing (or porting) a fancy macro assembler for Patmos, as the LLVM compiler framework (Deliverable D 5.2) is adapted to support Patoms. A version that generates single-issue Patmos code was reasonable early available during the development of Patmos. The C compiler produces linked ELF binaries. As a first step towards using small C programs for testing, we added a small tool (`elf2vhd`) to convert the essential part of the ELF file to a binary, which was then converted to the VHDL table. With this simple approach we bootstrap the download process by writing the boot loader, which understands different program segments, in C itself.

4.3 Co-Simulation

When building a complex hardware, such as the Patmos processor, testing, having a good test coverage, and actually checking the outcome of the tests is very important. However, usual testing with looking at VHDL simulation waveforms or writing individual test benches becomes tedious and at some point unmanageable time intensive.

However, we have developed, as part of Deliverable D 2.1, a cycle accurate simulator of Patmos in C++. This simulator is currently able to cycle accurate execute the integer benchmarks form MiBench (see Deliverable D 5.2). Therefore, this simulator can serve as gold reference for the hardware implementation (in VHDL) of Patmos.

The idea is to compare on a cycle-by-cycle base the execution of `patsim` and the execution of the VHDL simulation in VHDL. To compare the two simulation we consider the most important state of a processor: the register file. A difference in other program visible state (program counter, predicate registers) might also be interesting, but a difference there between the two simulations will at some point (some cycles later) show up in the register file - if not, the failure would not be visible during a normal program execution.

We have setup scripts to automate the testing. A collection of assembler programs is co-simulated in a single run. This list is continuously extended to increase the test coverage. Furthermore, we have setup a regression testing machine (`procell.imm.dtu.dk`) that also contains an FPGA board. On that machine every night the source is cloned from the git repository, the tools and processor built, and the regression tests run. The result is sent out per email to the Patmos developers and interested parties. [4]

[4] To be included in this email list edit `testsuite/recipients.txt` and commit the change or send a git patch to a Patmos developer.

5 Source Access

Patmos is available as open-source under the Simplified BSD License. The source code is provided through the Patmos repository via the `git` source code management tool:

`https://github.com/t-crest/patmos`

5.1 Requirements

To build Patmos and develop for Patmos only open-source or free tools are needed (except an FPGA board and a PC):

- A Unix like environment with `git`, `make`, and a C/C++ compiler, such as: Linux, Mac OSX, or cygwin/Windows
- A recent version of `cmake` and `boost` for the assembler
- ModelSim for simulation (the free version from Altera is good enough) [5]
- Altera Quartus for synthesizing for an FPGA (the free Web edition is good enough) [6]
- An FPGA board, the default target is currently the Terasic DE2-70 board [7]
- A serial interface on a PC to read the program output
- The library `libelf`, when executing C code on Patmos

5.2 Retrieving the Source Code and Building Patmos

The source code of Patmos and its simulator can be retrieved with as follows:

```
git clone git://github.com/t-crest/patmos.git
```

or downloaded as .zip file from GitHub:

`https://github.com/t-crest/patmos/zipball/master`

The build process and the simulation of Patmos is `make` based. The simulator and assembler of Patmos use `cmake`. However, the build of the simulator and assembler is included in the main Makefile.

A plain, and simple

```
make
```

will: (1) build all tools, (2) assemble the default program `ams/test.s`, (3) generate the instruction ROM table, (4) synthesize Patmos for the FPGA board DE2-70 (Altera Cyclon II FPGA), and (5) configure the FPGA.

Following `make` targets are available to perform only part of the steps, or different functions:

[5] `https://www.altera.com/download/software/modelsim-starter`
[6] `https://www.altera.com/download/software/quartus-ii-we`
[7] `https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=226`

`make tools` build all tools (C++ and Java based)

`make rom` assemble a program and generate the VHDL ROM table

`make comp` compile a C program and link to an ELF binary

`make crom` convert an ELF binary to the VHDL ROM table

`make sim` compile the VHDL files with `vcom` and run the ModelSim simulation

`make hsim` run the high-level (C based) Patmos simulation

`make test` run the test suit

`make patmos` synthesize Patmos and configure the FPGA

`make synth` just synthesize

`make config` just configure the FPGA

`make clean` remove (most) temporary files [8]

The default application can be changed by setting the variable `APP`, e.g., to run the ModelSim simulation of the program `asm/branch.s` the command is:

```
make rom sim -e APP=branch
```

The `Makefile` is intended to support: Linux, a cygwin environment under Windows, and Mac OSX. Under Mac OSX the Windows version of ModelSim is supported via `wine`.

6 Requirements

In this section all requirements in aspect CORE and scope NEAR from Deliverable D 1.1, which are relevant for the processor work package, are listed. NON-CORE and FAR requirements are not repeated here. The requirements are followed by a comment to what extent it is fulfilled by the prototype or if it is not (yet) relevant to the processor development.

P-6-001 The processor (and local memory/cache) shall be a fully timing compositional architecture to support modular WCET analysis.

The current prototype is fully timing compositional.

P-2-002 The processor shall support a standard RISC instruction set (similar to the MIPS ISA).

The instruction set of Patmos is a RISC instruction set with full predication.

P-2-003 The processor shall have two execution pipelines and a shared register file with full forwarding.

The prototype of Patmos contains currently only one execution pipeline. The fetch stage is already able to fetch variable length instruction bundles for the immediate long instructions.

[8] All untracked files can be removed with `git clean -df`

P-2-004 The processor shall support time-predictable (WCET analysable) caches.

This requirement is not applicable for the prototype of Patmos. Cache design is part of Task 2.3

P-2-005 The processor shall support single-path code with predicates.

All instructions are predicated.

P-2-006 The processor shall have a single network port for memory access.

This requirement is not applicable for the prototype of Patmos.

P-0-061 The processor shall have a cycle counter, which can be read out for performance analysis.

Patmos contains two cycle counters: one ticking at the clock frequency and one ticking at 1 MHz.

P-4-014 The processor shall support time control instructions.

This requirement is not applicable for the prototype of Patmos.

P-4-015 The processor shall have variants of load/store instructions to support the different varieties of memory: main memory, stack and SPM.

Patmos supports typed load and store instructions. In the prototype they all map to the same on-chip data memory.

P-4-016 The processor shall provide access to the DMA controller via special block memory copy instructions.

This requirement is not applicable for the prototype of Patmos.

P-4-019 The processor shall be time-predictable (i.e.: temporal bounds on the time to produce and consume outstanding requests, to execute configuration code and to transport the configuration settings to the DRAM controller shall be provided).

The architecture of Patmos is time-predictable.

P-5-021 The processor shall have a local memory from/to which data has to be moved explicitly to/from the (shared) main memory.

The prototype contains only local memory.

P-5-022 The latency of instructions (except instructions to access main memory) shall be invariable with respect to their operands and statically known.

All instructions (except memory access) have a constant execution time.

P-5-023 The processor shall support full predication on its ISA.

All instructions are predicated.

P-5-025 The processor shall provide instructions to bypass the cache.

Patmos contains typed load and store instructions. One type is cache bypass to main memory.

P-5-065 The processor shall provide non-blocking memory operations.

This requirement is not applicable for the prototype of Patmos.

P-6-038 The processor shall implement disjoint instruction and data caches.

The current prototype uses disjoint on-chip memories for instructions and data.

P-6-039 The processor shall use LRU replacement policy for both caches.

This requirement is not applicable for the prototype of Patmos. Cache design is part of Task 2.3

P-6-042 Caches shall be private.

This requirement is not applicable for the prototype of Patmos. Cache design is part of Task 2.3

P-0-505 Preemption The system shall provide means to implement preemption of running threads; these means shall enable an operating system or execution library to suspend a running thread immediately and make the CPU available to another thread.

This requirement is not applicable for the prototype of Patmos.

P-0-506 Priority-preemptive scheduling The system shall provide means to implement CPU-local priority-preemptive scheduling without migration of threads between CPUs.

This requirement is not applicable for the prototype of Patmos.

P-0-508 Interrupts: The CPU shall support interrupts.

This requirement is not applicable for the prototype of Patmos.

P-0-509 Exceptions: The CPU shall support exceptions.

This requirement is not applicable for the prototype of Patmos.

P-0-525 Cache: The architecture shall contain predictable instruction and data caches.

This requirement is not applicable for the prototype of Patmos. Cache design is part of Task 2.3

P-0-526 Data cache flushing and bypassing: The platform shall provide a cache control interface that allows application code to flush and bypass the data cache.

This requirement is not applicable for the prototype of Patmos. Cache design is part of Task 2.3

P-0-543 Data cache freezing: The platform should provide a cache control interface that allows application code to freeze the data cache.

This requirement is not applicable for the prototype of Patmos. Cache design is part of Task 2.3

P-0-527 Scratchpad: The platform shall contain data scratchpad memory with bounded access time.

the current prototype contains only scratchpad memories for instructions and data with single cycle access time.

P-0-528 Scratchpad Control: The tool-chain shall provide a scratchpad control interface (e.g., code annotations) that enables managing data in scratchpads at design time.

This requirement is not applicable for the prototype of Patmos. It is a compiler feature.

P-0-538 Cache Bypass: The processor shall provide mechanisms to bypass the data cache on store and load instructions.

Patmos supports typed load and store instructions, where one type is cache bypass.

7 Conclusion

In this deliverable we presented the design of the time-predictable processor Patmos and a first prototype implementation. We believe that future embedded real-time systems need processors, interconnect, and memory controller that are designed to minimize the WCET and only implement architectural features that are WCET analyzable. To provide good single thread performance Patmos implements a statically scheduled, dual-issue pipeline. Patmos will serve as platform for future research on co-development of time-predictable architecture features and their WCET analysis within the T-CREST project.

References

[1] Altera Corporation. Nios II processor reference handbook. Available from `http://www.altera.com/literature/lit-nio2.jsp`, May 2011. Version NII5V1-11.0.

[2] Christoph Berg, Jakob Engblom, and Reinhard Wilhelm. Requirements for and design of a processor with predictable timing. In Lothar Thiele and Reinhard Wilhelm, editors, *Perspectives Workshop: Design of Systems with Predictable Behaviour*, number 03471 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2004. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[3] Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proceedings of IEEE International Conference on Computer Design (ICCD 2009)*, Lake Tahoe, CA, October 2009. IEEE.

[4] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 264–265, New York, NY, USA, 2007. ACM.

[5] Heiko Falk and Jan C. Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *DAC '09: Proceedings of the Conference on Design Automation*, pages 732–737, 2009.

[6] Joseph A. Fisher, Paolo Faraboschi, and Young Cliff. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann (Elsevier), 2005.

[7] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[8] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. WCET driven design space exploration of an object cache. In *Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)*, pages 26–35, New York, NY, USA, 2010. ACM.

[9] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. Worst-case execution time analysis driven object cache design. *Concurrency and Computation: Practice and Experience*, 24(8):753–771, 2012.

[10] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In Erik R. Altman, editor, *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, pages 137–146, Atlanta, GA, USA, October 2008. ACM.

[11] Isaac Liu. *Precision Timed Machines*. PhD thesis, EECS Department, University of California, Berkeley, May 2012.

[12] Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proceedings of IEEE International Conference on Computer Design (ICCD 2012)*, October 2012.

[13] Jörg Mische, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. How to enhance a superscalar processor to provide hard real-time capable in-order smt. In *23rd International Conference on Architecture of Computing Systems (ARCS 2010)*, pages 2–14, University of Augsburg, Germany, February 2010. Springer.

[14] Peter Puschner. Experiments with WCET-oriented programming and the single-path architecture. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 2005.

[15] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[16] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[17] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16, Tokyo, Japan, March 2009. IEEE Computer Society.

[18] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[19] Martin Schoeberl. Leros: A tiny microcontroller for FPGAs. In *Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL 2011)*, pages 10–14, Chania, Crete, Greece, September 2011. IEEE Computer Society.

[20] Martin Schoeberl. Is time predictability quantifiable? In *International Conference on Embedded Computer Systems (SAMOS 2012)*, Samos, Greece, July 2012. IEEE.

[21] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*, pages 152–160, Lyngby, Denmark, May 2012. IEEE.

[22] Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, DOI: 10.1007/s11241-012-9159-8:1–28, 2012. doi: 10.1007/s11241-012-9159-8.

[23] Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, number 5860 in LNCS, pages 180–191. Springer, November 2009.

[24] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[25] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[26] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, and J. Wolf. Merasa: Multi-core execution of hard real-time applications supporting analysability. *Micro, IEEE*, 30(5):66–75, 2010.

[27] Jack Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction*. PhD thesis, University of York, 2008.

[28] Jack Whitham and Neil Audsley. Time-predictable out-of-order execution for hard real-time systems. *IEEE Transactions on Computers*, 59(9):1210–1223, 2010.

[29] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.