T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE FOR EMBEDDED SYSTEMS

Project Number 288008

D 5.1 ISA and Architectural Support for Generating Time-Predictable Code

Version 1.0
15 March 2012
Final

Public Distribution

Vienna University of Technology, 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**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the Partners accept no liability for any error or omission in the same.

© 2012 Copyright in this document remains vested in the T-CREST Project Partners.

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 Tobias Schoofs Av. D. Joao II, Torre Fernao Magalhaes, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: tobias.schoofs@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail:
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

1 Motivation **2**

1.1 Scope of the Study 2

1.2 Structure of this Document 3

2 Background **4**

2.1 Real-Time Systems: A High-Level View 4

2.2 Aspects of Time-predictability 4

3 Requirements for a Time-predictable Architecture **6**

3.1 Time-predictability at Task Level 6

3.1.1 WCET Analysis 6

3.1.2 Problems of Common Architectural Features 6

3.1.3 Time-predictable Task Code 8

3.2 Time-predictability at System Level 10

3.3 Towards a Time-predictable Hardware Architecture 11

4 The Patmos Design **13**

4.1 Goals 13

4.2 Common Properties of the Patmos Architecture 14

4.2.1 Terminology 14

4.2.2 Basic Features 15

4.2.3 Instruction Timing 16

4.3 Memory Organisation 17

4.3.1 Method Cache 17

4.3.2 Data Cache 18

4.3.3 Stack Cache 19

4.3.4 Data Scratchpad 20

4.4 Function Call Handling 20

4.5 Other Real-Time System Requirements 21

4.5.1 Memory Management Unit 21

5 Architectural Extensions **22**
5.1 Multi-core Support and Synchronisation 22
5.1.1 Remote Data Scratchpad Access 23
5.2 Interrupts and Multi-Threading Support 23
5.2.1 Context Switch Support 23
5.3 Further Instruction Set Extensions 23
5.3.1 DMA Interface 23
5.3.2 Non-Blocking Loads with Implicit Wait on Use 24
5.3.3 Floating-Point Instructions 24

6 Summary **25**

A The Architecture of Patmos **27**
A.1 Pipeline 27
A.2 Register Files 27
A.3 Instruction Formats 29
A.4 Instruction Opcodes 31
A.4.1 Binary Arithmetic 31
A.4.2 Unary Arithmetic 33
A.4.3 Multiply 34
A.4.4 Compare 34
A.4.5 Predicate 36
A.4.6 NOP 37
A.4.7 Wait 37
A.4.8 Move To Special 38
A.4.9 Move From Special 39
A.4.10 Load Typed 39
A.4.11 Store Typed 41
A.4.12 Stack Control 43
A.4.13 Call and Branch 44
A.4.14 Call and Branch Indirect 45
A.4.15 Return 46
A.5 Stack Cache 47
A.6 Method Cache 47

A.6.1 FIFO Replacement 48
A.7 Application Binary Interface 48
A.7.1 Data Representation 48
A.7.2 Register Usage Conventions 49
A.7.3 Function Calls 49
A.7.4 Stack Layout 49
A.7.5 Instruction Usage Conventions 50

Document Control

Version	Status	Date
0.1	First outline	17 November 2011
0.2	Requirements section	31 January 2012
0.3	Patmos description, ISA	15 February 2012
0.4	Introduction and background	20 February 2012
0.5	First draft complete	24 February 2012
0.6	Complete version for Dresden meeting	06 March 2012
0.7	Quality assurance (from feedback)	14 March 2012
1.0	Final version	15 March 2012

Executive Summary

This document contains the deliverable *D 5.1 ISA and Architectural Support for Generating Time-Predictable Code* of work package 5 of the T-CREST project, due 6 months after project start as stated in the Description of Work. This document describes the requirements on the hardware architecture to support the generation of time-predictable code and specifies the instrution-set architecture of Patmos.

1 Motivation

In hard real-time systems, correctness of a program not only depends on the values of the computed results but also on the instant in time at which they are delivered. Typically, these systems have to interact with a physical environment, i.e., they receive signals from sensors, perform some computations based on these signals and control the environment by sending signals to actuators. The response-time, i.e., the time from the occurrence of an event to the reaction observed by the environment, is constrained. In a hard real-time system, failing to satisfy these constraints can result in havoc, threatening human lives.

The validation of timing constraints of a hard real-time system is essential. Unfortunately, modern computer architectures employ features that make this validation difficult. They are built with optimisation towards a high average case performance in mind, without paying much attention to the *time-predictability* of the architectural features employed. The term "time-predictability" is commonly used to describe properties of a system that influence the effort required to validate its timely behaviour, particularly in the worst-case. Hence, these features often make the assessment of the system timing either infeasible or overly pessimistic. Time-predictability is a property required for real-time systems.

In the T-CREST project, we aim at building a system in which time-predictability is a first-order design factor. As a consequence, the exhibited *worst-case performance* should be competitive and considerably better compared to state-of-the-art embedded real-time systems. One essential part of the system to be developed is a compiler that generates time-predictable code and optimises towards its worst-case execution time (WCET).

In this report, we investigate requirements for an architecture to allow the generation of time-predictable code. We provide a high-level view of the Patmos [37] core, highlighting architectural features that influence time-predictability at the instruction set architecture (ISA) level and hence are subject for code generation. In the appendix, we describe a first version of the Patmos ISA in detail.

1.1 Scope of the Study

When we are talking about generation of time-predictable code, we refer to the translation of the source code of tasks down to machine code. Time-predictability cannot be considered only at task level, the overall system architecture plays an important role. This includes the execution environment, how tasks are scheduled by the OS, which resources are shared, etc. Although we will discuss the effect of different design decisions on time-predictability at the system level, our focus lies at the task level. Tasks are subject to WCET analysis, and in the course of the project we strive to achieve a tight interaction between the compiler and the WCET analysis, to effectively reduce the WCET bound. We outline the implications of the overall system architecture on the analysability of tasks, but we do not presume any particular architecture or operating system. A thorough investigation of scheduling theory and models of computation is out of the scope of this project. We also do not discuss the automatic parallelisation of tasks and their distribution onto the processors.

1.2 Structure of this Document

We start with a high-level view on real-time systems to motivate for a time-predictable architecture in Section 2, together with a short description of various aspects of time-predictability. Section 3 elaborates on the requirements of a time-predictable processor architecture. A high-level view on the current Patmos architecture is presented in Section 4. Extensions of the architecture that may enable additional optimisations to improve the WCET or allow for more flexibility are discussed in Section 5. The Patmos ISA is presented in detail in Appendix A.

2 Background

2.1 Real-Time Systems: A High-Level View

Real-time embedded systems impose very specific requirements on the system design. In the T-CREST project, we build a time-predictable system from scratch. Current state-of-the-art software engineering is a hierarchical process, and in real-time systems engineering, the timing behaviour also must be part thereof. It is not sufficient to consider hardware and software components separately. Instead, we first need to consider how the abstractions made in system design allow to assess its timing behaviour, and then identify the requirements they impose on the underlying hardware.

Real-time systems are commonly designed as hierarchical systems. This simplifies the system model by decomposing it into sub-problems such that the system is easier to comprehend and verify. The system is modelled as a set of *tasks* which may cooperate to achieve the desired behaviour. A task is a run-time entity sequentially executing a program written in a programming language, with a single thread of control. In order to make progress, a processor needs to be allocated to a task, and the execution time of a task is determined by the hardware on which it is executed, the initial state thereof, and the input data. Modern state-of-the art processors also feature multiple processing cores that are able to execute multiple tasks simultaneously, allowing to exploit any existing thread-level parallelism in the application.

The allocation of the tasks to processors is called *scheduling*. As a task may process input from the environment (from sensors) or from other tasks and produce output for other tasks or the environment (actuators), dependencies between the tasks need to be respected in the schedule. *Schedulability analysis* aims at constructing a schedule that both satisfies the dependencies between the tasks and the timely requirements of the system. Therefore, an upper bound for the execution times of all tasks must be provided to the analysis. Obtaining the exact *worst-case execution time (WCET)* of a task statically is infeasible due to the large state space that needs to be explored, both induced by the possible hardware states (hardware complexity) and the possible flows of control through the task (program complexity). Therefore, often a safe (over-)approximation is determined by static program analysis. The term WCET is commonly used synonymously with the WCET bound determined by the WCET analysis.

2.2 Aspects of Time-predictability

Time-predictability is a term that covers different aspects of the temporal behaviour of a system. An aspect might apply to the system architecture, a hardware component, a task, a piece of code or a combination of thereof.

Analysability

: is a measure for the complexity of the WCET analysis that reflects the effort required to produce a reasonably tight WCET bound for a task [13]. Both the complexity of the internal hardware state as well as the high number of different input-dependent program paths adversely affect analysability of the code [42, 12].

Repeatability

The behaviour of a system exhibits a *repeatable* property if it is satisfied in every correct execution. With repeatable timing, the timing behaviour of a system will be identical when invoked with identical input data [7].

Stability

is a measure for the variability of code execution times [24, 13]. In a stable system, i.e., a system that guarantees minimal difference between worst-case and best-case execution time, one could make a reasonable prediction for the execution time of any execution of an application by the knowledge of the execution time of a single execution. The prediction will be as good as the stability of the system in this case. In the extreme case of perfect stability, every pair of executions would yield the same execution time, hence the WCET could be exactly determined by measurement. In a perfectly stable system, the timing of the output generated by the application is a repeatable property.

Composability

is a property that states how a component behaves if it is integrated or executed in different contexts [24]. With respect to timing, a composable component will exhibit the same timing behaviour in different contexts. The property not only captures the timely behaviour of a component on its own, but also the influence it has on the timing of other components of the system.

Compostitionality

is a property that states how the behaviour of a component can be derived from the behaviour of its sub-components [24, 30]. For example, the WCET of a component with compositional timing behaviour can be calculated from the WCETs of its sub-components by a simple formula.

Time-predictability covers the notion of *tightness* of the obtained WCET bound, i.e., how close the prediction is to the actual WCET. Note that time-predictability does not imply a *low* WCET bound. Some system components and design decisions allow a higher flexibility that results in a more dynamic execution (i.e., the sequence of executed instructions and the processor state are more susceptible to different input values). This may reduce the WCET bound but may also have a negative impact on at least some aspects of time-predictability. In accordance with the various aspects of time-predictability described in this section, the requirements of the system and the architecture to cover a certain aspect may differ.

3 Requirements for a Time-predictable Architecture

In this section, we explore the requirements for a time-predictable architecture step-by-step. We need to identify properties of contemporary system designs that prohibit a hierarchical, composable timing analysis. Therefore, we discuss both the impact of the processor architecture and the software architecture on the time-predictability of the system. Having outlined the hierarchical decomposition of real-time systems in the last section, following questions need to be answered on our way to a time-predictable architecture design:

- What are the requirements to make the timing behaviour of tasks predictable? How can we obtain an accurate WCET bound for tasks?
- How does the software system organisation influence the timing behaviour of individual tasks? What hardware features support compositionality of timing behaviour at system level?

3.1 Time-predictability at Task Level

3.1.1 WCET Analysis

There are two methods to obtain the WCET of a task: *measurement* and *static program analysis* [42]. Measurement is not considered safe in general, i.e., it is hard to verify that the WCET has been actually triggered, both by the input data considered and the initial hardware state encountered. Therefore, the observed execution times are *WCET estimates*. The other method, static analysis, aims at obtaining *safe upper bounds* for the WCET of a task. The established method combines the results of two sub-analyses:

- *Processor-Behaviour Analysis* (or low-level analysis) determines upper bounds on the execution times of instructions or basic blocks of the program by using a model of the targeted hardware architecture.
- *Control-Flow Analysis* (or high-level analysis) determines information about the possible flow of control through the task.

The problem of finding the WCET of a task is expressed as an optimisation problem, with a method called *implicit path enumeration technique* (IPET) [15, 25]. A system of linear constraints is used to describe the possible flows of control of a program. The function to maximise is the sum of the product of the number of execution counts and the upper bounds on execution times of all basic blocks. The solution consists of the execution counts of basic blocks on the worst-case path.

In the following, we identify what makes both low- and high-level analysis difficult, and propose solutions that do not exhibit these issues.

3.1.2 Problems of Common Architectural Features

Retrospective. In the late 80's, when the timely behaviour of programs became of interest, engineers could simply look at the assembly code sequence and add up the latency of each instruction, which was well defined and either specified in the data-sheets or easily obtainable due to simple

processor design. With the ever since growing demand for processing power, computer architects took advantage of the fact that timing behaviour of a processor was abstracted away at the instruction set architecture (ISA) level. Only the functional semantics that were specified had to be respected. As a result, features that aimed at improving the average processing speed but did not care about the precise effect on the timing of instructions, were developed. Pipelined execution enabled overlapping of instructions, multiple functional units within the processor demanded for dynamically reordering instructions to allow for a better utilisation, dynamic branch prediction was developed to keep the pipeline filled, and the advent of caches (at several levels of the memory hierarchy) aimed at bridging the increasing gap between processing speed and main memory access latency. These features, though very beneficial for the average performance, made it impossible to determine the exact execution time of a code sequence by solely examining its instructions. The execution time of a single instruction became dependent of the current hardware state and hence the execution history. For example, the latency of a load of a memory word can differ in orders of magnitude, dependent whether the word is resident in the cache or not (in the latter case, it has to be fetched from memory first and loaded into the cache). Another level of architectural complexity was added with the advent of multi-core technology, to overcome the physical limits for the clock frequency increase. Shared memory buses and shared caches suffer from interference, such that the access latency and state of these resources are highly dependent on the activity of all cores that access them.

Despite these developments, real-time system designers needed to obtain the WCET of a program. They could not simply use older, very predictable architectures because of their demands concerning processing power. Therefore, methods were devised to take these hardware architectures into account by employing an adequate hardware model for WCET analysis.

Reasons for pessimistic analysis results. Timing analysis of programs on modern processors that employ dynamic optimisations is often infeasible or results in very pessimistic execution time bounds, for several reasons:

- A complex processor requires a complex timing model for WCET analysis.
- The hardware state at a given instruction is highly dependent on the execution history, and the instruction timing is highly dependent on the hardware state. As a result, also instruction timing is highly dependent on the execution history.
- Even if the underlying hardware model is perfect, it is intractable to consider all possible execution paths to get the exact hardware state at a given program point.
- The infeasibility of exhaustive search due to the large state space makes abstraction necessary in the analysis.
- The initial hardware state is unknown and the number of possible hardware states that needs to be tracked is too large.
- The possibility of external interferences (interferences that are not in the scope of the task) on shared resources makes it difficult to keep track of their state.
- The internals of the hardware that are required to build a timing model for analysis are unknown or undocumented.

There has been a lot of research identifying features that impair the time-predictability of an architecture [41, 27, 10, 38, 3, 12]. We give three examples of common dynamic average-case performance enhancing processor features that have a negative impact on time-predictability:

- Superscalar pipeline with out-of-order execution. Instructions are scheduled depending which hardware resources are currently available and which instructions are eligible for execution. If the latency of instructions is dependent on the execution history, then also the pipeline state and the resulting schedule.
- Dynamic speculative execution. For example, dynamic branch prediction is employed to keep the processor pipeline filled and avoid stalling. Whether a branch is predicted as taken or not again depends on the history of the actual branch outcomes, and so does the misprediction penalty.
- Caches. Reineke et al. [27] show that certain cache replacement policies (esp. FIFO and PLRU) are unfavourable because the length of memory access sequences required to recover from an uncertain cache state and to be able to classify subsequent memory accesses as hits or misses is high.
  Their combination with a write-back policy might even worsen the situation, because a potential write-back of a cache line needs to be taken into account for each potential data-cache miss, for noncompositional architectures [41]. Additionally, unified caches for data and instructions cause interferences that contribute to the impreciseness of the analysis result.

Architectures employing these features are prone to *timing anomalies* [16, 28, 40]. Intuitively, a timing anomaly is a situation in which the local worst-case does not entail the global worst-case – a situation which is undesired for timing analysis as it is not sufficient to follow only the worst-case path. The absence of timing anomalies allows the analysis to follow the local worst-case path for the derivation of execution times. Therefore, it is a fundamental criterion for the analysability (i.e. time-predictability) of the architecture.

3.1.3 Time-predictable Task Code

Considering a single task, the task structure, the used language features and additional information available from the design phase influence its analysability. Gebhard et al. [9] make a distinction between requirements that need to be met in order to be *able to* compute a WCET bound in the first place, and challenges that affect the *tightness* of the derivable WCET bound. Knowledge about the maximum number of loop iterations (*loop bounds*), the maximum recursion depth of recursive calls, and targets of function pointers and indirect branches are a prerequisite for analysable code.

The tightness of the WCET bound is affected by any abstraction the analysis has to make due to unavailable information. One abstraction is the information about the possible control-flow paths that stems from input-data dependent control decisions, such that execution paths are considered by the analysis that are infeasible in the real execution context. Unknown memory access addresses are another source of uncertainty that influences the tightness of the analysis result. The analysis tries to keep track of the cache state as caches are part of the timing-relevant processor state. The more sensitive the policy is to memory addresses, the larger part of the abstract cache state is destroyed by an access to an unknown address. For this reason, dynamic heap memory allocation is disadvantageous for predictability of task timing. Note that dynamic heap memory allocation is forbidden in coding standards for the real-time domain, e.g., in MISRA-C [18].

In the following, we present two strategies to make task-timing more predictable. The first strategy aims at improving analysability of memory accesses, while the second strategy is a means to reduce the complexity of path analysis and making task execution times stable.

Predictable Memory Hierarchies. Caches play an important role for bridging the gap between processor speed and memory access times. Execution times for instructions that load a word from memory can vary in orders of magnitude, depending whether the word is resident in a cache or not. For a tight WCET bound, WCET analysis must take cache contents into account. Unfortunately, memory addresses are often input-data dependent and not known at compile time. Therefore, accesses of statically unknown memory locations entail updates to the cache state that are hard to predict, depending on the associativity of the cache and the replacement policy employed. To avoid corruption of the abstract cache state by data accesses, separation of instruction and data caches is mandatory [10]. Schoeberl [32, 33] proposes a cache design that is composed of separate caches not only for data and instructions, but also for data accesses exhibiting regular access patterns. For example, addresses of stack-allocated data and local variables can be determined by analysis of the call tree of a task, making their access predictable. Access to data at hard-to-predict locations is best handled with a cache employing an LRU replacement policy, which exhibits the best analysability properties [27]. Any separation of predictable and non-predictable memory accesses reduces interferences that have to be modelled in the WCET analysis, improving time-predictability. *Scratchpad memories* [2], i.e., explicitly managed, fast local memories, can be utilised to keep WCET critical code and data close to the processor, increasing both predictability and WCET performance [6]. A similar effect can be achieved by locking parts of the cache (*cache locking*) to prohibit the contents from being evicted [8, 39, 19].

The Single-Path Approach. Puschner and Burns [23, 20, 21] propose the *single-path approach* to reduce the effort required for control flow analysis of a task. It is based on a technique known as *if-conversion* [1] that transforms control-dependencies into data-dependencies. The key idea is to eliminate all input-data dependent control-flow branches by speculatively executing input-data dependent alternatives, and fixing the number of input-data dependent loop iterations to the loop bound. The effect of the "wrong path" on the processor state is either masked out or mis-speculated computation results are discarded. The single-path approach reduces the feasible paths for a piece of code under consideration to a single one. Variations in the execution time due to input-data dependencies are avoided by keeping input-data dependent branching local to single instructions with data-independent execution times.

The main benefit of the single-path approach is the reduced complexity of the control-flow structure of a piece of code, resulting from single-path conversion. The WCET can be obtained by analysis of the single execution path with little effort. In simple architectures, the execution time of a piece of single-path code even is constant, allowing for obtaining its WCET simply by measurement. The single-path approach is a means to achieve *stable* execution times for tasks, i.e., not only the computational result but also the timing becomes a repeatable property of a task.

To realise single-path code, the architecture is required to provide some form of predication. The effect of predicated instructions on the processor's state becomes visible only if the predicate associated with that instruction evaluates to true. Otherwise they practically become a NOP (no-operation)

instruction. Many if not most modern microprocessors (e.g., Alpha, ARM, IA-64, Motorola M-Core, Pentium P6) employ some form of predicated instructions. Predication is utilised to reduce the number of conditional branches and hence penalties imposed by pipeline stalls due to branch-mispredictions. Primarily in VLIW architectures it is used to form regions of multiple basic blocks where a predicate is assigned to each basic block instead of employing conditional branches (*region formation*). This increases the utilisation of multiple functional units, as the pool of instructions which the instruction scheduler in the compiler backend can pick from, is larger compared to when only a single basic block is considered. For the single-path approach, the timely behaviour of the predicated instructions must be independent from their operands. Architectural support for predication comes in two variants [17]:

- Partial predication - The ISA specifies at least conditional move instructions, which copy the contents from the source register to the destination register only if the predicate has a specific value.
- Full predication - All instructions of the ISA can be guarded by a predicate.

Full predication has the benefit that no temporary storage for any alternatives is required from which then the “right” result is copied by a conditional move. Also, no provisions to avoid certain exceptions (division by zero, access of an illegal memory address) resulting from speculatively executing computations from any alternative are required with full predication. Transforming code with conditional branches to predicated code requires instructions for the definition and the manipulation of predicates. Nesting of control structures requires the conjunction of predicates, merging of paths in the control-flow graph requires their disjunction.

It must be mentioned that serialising control flow alternatives imposes a penalty in terms of the number of instructions executed and required to be fetched from memory in the general case. Therefore, it is desirable to avoid input-dependent control flow decisions in the first place – a strategy named *WCET-oriented programming* [22].

3.2 Time-predictability at System Level

So far, we have considered time-predictability within a *single execution of one task* only. For our discussion on system-level time-predictability, we assume a *simple task* (or *S-task*) model, as described by Kopetz [14]. Once started, a simple task will have all input data present locally and there is no blocking by I/O or other synchronisation or communication constructs. Availability of outputs at defined locations within the task’s local memory is part of the postcondition. Copying input and output data is part of the task’s execution environment (e.g., the operating system or the physical world). The interface between a task and its environment encompasses temporal constraints, control properties, functional intent and data properties. Interactions between a task and its environment that influence the execution time of the tasks, but are not visible at the task interface (e.g. cache state updates), are called *side effects*. In Section 3.1.3, we have explored how we can make task-timing predictable, focusing on its analysability. In the following, we discuss implications on the system design to reduce the impact of side effects.

Puschner and Schoeberl [26] investigate timing interactions between (periodically executed) tasks in a real-time system. They describe both the reasons for the interactions and the effects on task timing,

and highlight the consequences for timing analysis of the whole system. We summarise the effect on timing with increasing flexibility in the system:

- A single periodic task may encounter a different initial timing-relevant hardware state each time it is executed. Different input values lead to different execution paths, leaving the hardware in different states upon task completion. This state will be the initial state for the next instance of the task.
- If the (timing-relevant) hardware state is shared among tasks, i.e., other tasks are executed on the same processor between invocations, altering cache contents, dynamic branch predictors, etc., the timing of a task instance will not only depend on the hardware state it had left in the previous execution, but also on the tasks and operating system activities in between.
- Another level of complexity is added in systems with preemptive scheduling. Preemptions may have almost arbitrary effects on the state of a task *during* its execution. The number of possible preemptions, the state of the task at preemption time and the state modifications performed by the preempting code impact the task timing.
- In multiprocessor systems, interferences on resources shared among processors (e.g., concurrent access of a shared memory bus or caches) will degrade predictability of task timing.
- In *complex tasks* (or *C-tasks*) [14], tasks can be blocked due to synchronisation with other tasks in the system. Progress and WCET of a C-task not only are dependent on the behaviour of other tasks, but also on the properties of constructs and primitives used for synchronisation.

The bottom line is that time-predictability of tasks degrades with the growing amount of resources shared between them. If variations in the behaviour of a task influence the timely behaviour of other tasks, the system becomes uncomposable. Therefore, for a time-predictable architecture design, interferences on the timing-relevant hardware state of a task must be avoided. In a composable system architecture it is essential that tasks can be analysed in isolation to obtain the timely behaviour of the system. Means to achieve such composability are *temporal* or *spatial isolation* of components.

Kirk [11] has proposed spatial isolation of tasks by splitting the instruction cache into partitions and statically assigning partitions to tasks. Partitioning the cache avoids interference on the task's now private cache state, at the cost that the size of cache memory available for each of the tasks is smaller. Alternatively, a system could allow preemption at the cost for saving and restoring the complete timing-relevant hardware state. A more aggressive approach viable in multiprocessor systems is a complete separation of resources by assigning each task to a separate processor core, with separated caches and local memories. Access to shared memory should be temporally isolated by static, preplanned scheduling of the memory accesses for all cores [29].

3.3 Towards a Time-predictable Hardware Architecture

In the last decade, substantial research in the field of real-time systems has focused in investigation of processor and system architectures that aim at providing a high degree of time-predictability [33, 3, 38, 5, 26, 7, 13]. In the T-CREST project, we take a constructive approach towards time-predictable processor architecture. Having outlined the problems of modern processor architectures, we build an architecture from scratch that is designed for high time-predictability.

Our design philosophy is to

- make timing behaviour an integral part of the ISA and processor components
- use static alternatives for commonly used performance-enhancing features at runtime to reduce the dynamic behaviour and state of the processor that cannot be controlled by the code at the ISA level
- only include features that inherently have a high degree of analysability
- avoid interferences on shared resources to provide both high composability and compositionality

In the following, we outline the requirements and suggestions for architectural features to support the generation for time-predictable code.

Pipeline. The pipeline design must be fully compositional and it must not exhibit timing anomalies. Instead of a deep superscalar pipeline featuring out-of-order execution we propose an in-order VLIW pipeline that makes the instruction schedule explicit at the ISA level, that the processor will adhere to. Dependencies and instruction latencies have to be respected in the assembly code. Stalling of the pipeline should also be made explicit at ISA level. Instructions that decouple the execution of an instruction in the pipeline from its history must be provided, i.e., an instruction that explicitly waits until in-progress memory transactions are completed. Except for instructions that explicitly stall the pipeline or have dependencies on events that are not task local (e.g., accessing the chip interconnect), instruction timing must be constant and independent of the operands. We require a predicated instruction set to support the single-path approach, to decrease the complexity due to input-dependent control flow paths.

Branch prediction. We prohibit dynamic branch prediction and instead propose static schemes like BTFN (backward taken, forward not-taken) or branch delay slots. Branch delay slots provide the advantage that there are no mispredictions and pipeline flushes, and that penalties due to underutilisation in delay slots are expressed in the assembly code (by NOPs).

Memory hierarchy. To increase the time-predictability of memory accesses and to minimise the negative impact of interferences on the cache, we propose separate caches for instructions and data, and distinct, explicitly addressable data caches. Such a *split-cache* architecture [32] will improve the analysability of the cache contents, and the composability of cache costs. The architecture should provide means to force data local to the CPU, either by providing an explicitly managed scratchpad memory or a cache-locking mechanism. We require the data cache to adopt a replacement policy with good analysability properties, more precisely, a highly associative LRU cache. For accesses to hard to predict memory addresses, bypassing the cache should be provided.

In the next section, we present the design decisions made for Patmos, the processor architecture developed in the T-CREST project.

4 The Patmos Design

Patmos is a novel processor core targeted at real-time applications that will be implemented in the scope of the T-CREST project. The Patmos core will be used to develop a multi-core processor. Every core will feature its own data cache, instruction cache and a data scratchpad. A network-on-chip (NoC) will connect the cores to a time-predictable memory controller that provides access to the shared main memory. The caches and the scratchpads of the processor cores are not shared between cores. To enable efficient communication between the cores without the need to use the global shared memory, it will be possible for cores to access data scratchpads of other cores over the NoC. Figure 1 depicts one Patmos processor core and its memory hierarchy connected to the NoC.

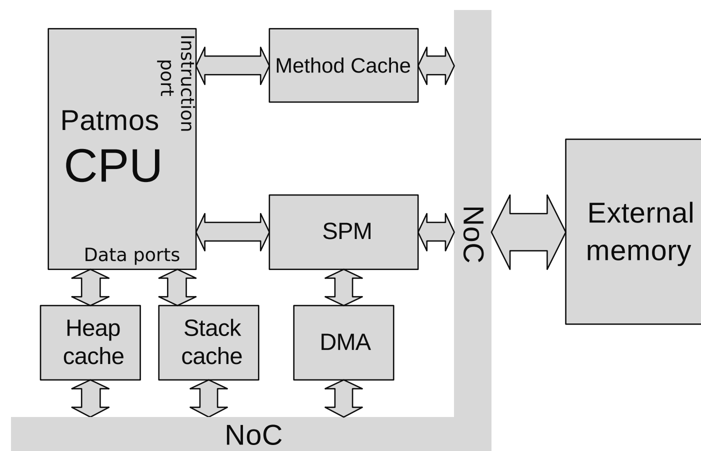


Figure 1: The Patmos processor core and its memory hierarchy connected to the NoC. *(Image from UoY)*

In the remainder of this report, we focus on the instruction set architecture of a single core. We present the current design of Patmos and discuss the design decisions. There are several possible alternatives and possible extensions to current design features to improve performance or predictability, but their potential benefit and their detailed implementation need yet to be determined. We will discuss those possible extensions or alternatives as well as the integration of Patmos into a multi-core processor in Section 5. For a detailed description of the instructions, their behaviour and their encoding, refer to Appendix A.

4.1 Goals

The goal of the Patmos design is to create a time-predictable processor core, i.e., the hardware should have a model that allows a quite precise yet feasible worst-case analysis of the whole system, while still providing a good worst-case performance. To keep the complexity of both the hardware implementation and the hardware model low, the design avoids some dynamic optimisation features that may introduce timing anomalies in the analysis, such as out-of-order executions. The behaviour of an instruction should not depend on the dynamic execution time of previous instructions. Scheduling decisions are taken off-line at compile time by the compiler to avoid scheduling timing anomalies.

In the T-CREST research project Patmos should be used to evaluate compiler optimisation and code generation techniques. Therefore the instruction set architecture should also support efficient single-path code generation and region formation techniques.

Since Patmos will be integrated into a multi-core processor, the memory hierarchy plays a major role. Access to the global shared memory can be expected to be slow, since the resource is shared by all cores. To deliver a good performance, the cores must have local memory, both to store frequently used data for reuse as well as to prefetch data into local memory to avoid stalls when the data is accessed. Caches should allow for a good hit analysis, else the cache analysis would cause large overestimations due to the high cache miss costs. It should be possible for the compiler to prevent memory accesses from changing the cache state, so that the compiler can prevent data for which the cache analysis cannot find an access that is a hit from evicting more useful cache entries. For performance reasons, it should be possible to execute code and perform memory accesses in parallel.

4.2 Common Properties of the Patmos Architecture

Patmos is a 32-bit, RISC-style dual-issue VLIW processor core with variable-length instruction encoding. The Patmos architecture has first been introduced in [37]. Instructions are statically scheduled and executed in-order. The VLIW design therefore shifts the instruction scheduling task to the compiler, simplifying both hardware and WCET analysis. Patmos uses a method cache for instruction caching, and a set-associative data cache and a stack cache for data caching, as well as a data scratchpad memory. The memory organisation is explained in more detail in Section 4.3.

4.2.1 Terminology

First, we define some basic terminology.

Instruction bundle
: An instruction bundle specifies the unit that is fetched by the processor core in a single cycle. The first bit of an instruction bundle specifies whether it is 32-bit or 64-bit wide. One instruction bundle can contain either one 32-bit instruction, one 64-bit instruction or two 32-bit instructions.

Instruction
: An instruction specifies one “unit of work” processed in a pipeline. All instructions are 32-bit wide, except for the *ALU Long Immediate* (ALUl) instruction, which is 64-bit wide.

Operation
: An operation specifies a concrete instance of an instruction, a “kind”, e.g., *addition*, *load from stack cache*, *branch indirect*, etc.

Slot
: A slot specifies the position in an instruction bundle. A 32-bit instruction bundle only has one 32-bit slot. A 64-bit instruction bundle has two 32-bit slots. A 64-bit wide instruction bundle can either contain an instruction in the first and the second slot each, or in the case of an ALUl, one instruction in which the immediate is located in the second slot of the bundle. If

an instruction bundle contains only one 32-bit instruction or an ALUl instruction, the second ALU implicitly executes a NOP.

4.2.2 Basic Features

In this section we give an overview of the basic features of the Patmos ISA.

Register Files Patmos features 32 32-bit general purpose registers that can be used in both slots, i.e., the register file is not clustered. It also has a special purpose register file of 16 32-bit special purpose registers, one of which contains the 8-bit predicate register file (see below).

Arithmetic and Logic The ISA provides common arithmetic and logic operations on 32-bit values. All binary arithmetic and logic operations can take a 32-bit immediate value as second operand. A 32-bit instruction format is provided that allows to use 12-bit immediate values as second operand for some common operations. Patmos also provides an integer multiplication operation, but no hardware division or floating point operations. Overflows and underflows are not detected by the hardware. The compiler must use 32-bit operations to implement 64-bit operations.

Predication To support efficient single-path code generation and to support region formation to reduce the number of branches, the instruction set is fully predicated. The architecture provides eight predicate registers. Operations can be predicated either by one of the predicate registers, or with a negated predicate register (i.e., the operation is executed if and only if the predicate is set to false). The first predicate register is a special predicate register that always evaluates to true, thus always enabling the operation. The instruction set provides instructions to define predicates, by means of arithmetic comparisons, and instructions to combine predicates, i.e., by negation, conjunction and disjunction. Saving the predicate register file to and loading it from main memory is possible.

Control Flow Conditional branches are implemented as predicated jumps, i.e., a predicate has to be defined with a compare instruction first, which then is assigned to a jump instruction. Compare instructions set the value of predicate registers, branches are implemented as predicated jumps. The instruction set features both direct and indirect branches. No branch prediction is employed for control speculation, instead, there are branch delay slots. Calls are similar to branches, but additionally store required return information (cf. Section 4.4). Patmos does not provide support for exception handling at the time of writing.

Pipeline The pipeline of Patmos consists of five stages. Figure 2 shows a simplified diagram of the Patmos pipeline. The pipeline contains two ALU units. Integer arithmetic operations and logic operations can be issued in both slots. Other instructions, such as memory accesses (except stack cache accesses), wait instructions or function calls can only be issued in the first slot. Instruction bundles are executed in-order. The pipeline does not detect data hazards, i.e., the compiler must generate an instruction schedule that honours the minimum latency between all instructions that have

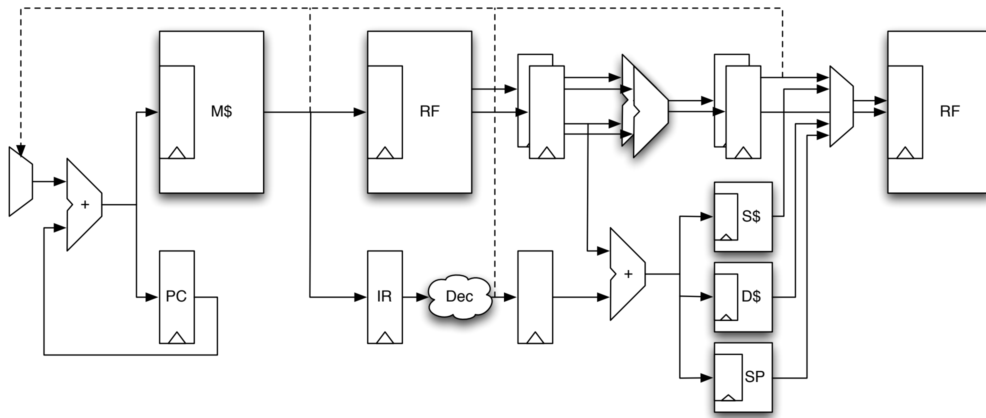


Figure 2: Pipeline of Patmos with fetch, decode, execute, memory, and write back stages (figure from [37]).

a data dependency, if necessary by inserting NOP instructions. Data hazards might occur when instructions that exhibit data dependencies modify data in different stages of the pipeline.

Control hazards are avoided by branch delay slots and call delay slots, i.e., the compiler is responsible for scheduling instructions after a transfer of control instruction that are executed until the processor knows the target address of the branch or call. The compiler must not schedule instructions in the second slot that use hardware resources that are available only in the first slot. However, the hardware avoids structural hazards by stalling the pipeline if a memory operation needs to use the memory controller while the memory controller is busy. If two instructions write to the same register in the same instruction bundle, the hardware gives precedence to the second instruction.

4.2.3 Instruction Timing

A particular characteristic of Patmos is the explicitness of timing at the ISA level. The instructions that may stall the pipeline and the possible causes for the stalls are defined by the ISA.

Only instructions that perform memory operations and dedicated stalling instructions can cause the pipeline to stall. Memory operations include blocking loads and stores (cf. Section 4.3.2), stack cache management instructions (Section 4.3.3), but also control-flow instructions that interact with the memory hierarchy (i.e., the method cache, Section 4.3.1). A `wait` instruction stalls the pipeline until a pending memory operation has finished, and an explicit multicycle-NOP instruction stalls the pipeline for a specified number of cycles. Memory access of data resident in any of the local memories (cf. next section) does neither stall the pipeline nor is a wait operation required.

Since the compiler is responsible for creating a hazard-free instruction schedule for all other instructions, the complexity of the timing analysis and of the hardware implementation is reduced. However, the code size can increase due to the need for explicit NOP instructions in the instruction stream.

Multiplications are executed in parallel in a separate pipeline and finish within a fixed number of cycles. The (64-bit) result of a multiplication of two 32-bit operands is written to a pair of (32-bit) special purpose registers. As multiplication is pipelined, a multiplication can be issued in every cycle.

4.3 Memory Organisation

Patmos employs a number of local memories to reduce the latency of memory operations. The memory organisation follows the ideas presented in [32]. A method cache is used as instruction cache. Data can be stored either in a local data scratchpad memory (D-SPM), or in set-associative data cache. A separate stack cache allows fast access to local variables. The various caches are explained in more detail below.

Load and store instructions are typed, i.e., the ISA provides separate instructions to access the various memories, which simplifies the timing analysis. Memory accesses that are guaranteed to access only local memories, such as accesses to the data scratchpad or to the stack cache, never stall the pipeline. For other memory accesses, such as accesses that use the data cache, the ISA provides blocking loads and decoupled loads. A blocking load stalls the pipeline until the memory access has been completed and stores the result into a general purpose register. No further stalling is required when the result of the load is used. This makes the timing of code sequences easy to compose. On the other hand, this prevents the memory load to be performed in parallel with the execution of the program. Therefore the architecture also provides decoupled loads. In this case the load instruction only starts the load. The result is written to a special register. The result can be copied into a general purpose register by a special move instruction. If the load is not completed when the special result register is accessed, the special move instruction stalls the pipeline until the load is completed. This enables a more efficient resource usage by executing the load and some other ALU instructions in parallel, but at the cost of an additional move instruction compared to the blocking load. An alternative to the decoupled loads is presented in Section 5.3.2.

Any memory instruction that accesses the main memory stalls the pipeline until all previously issued memory instructions have been completed. Memory accesses may not be reordered by the hardware. This guarantees that the stall time of an instruction can only be influenced by the most recently executed main memory access.

4.3.1 Method Cache

The method cache has been first been introduced in [31]. It stores functions as continuous memory blocks in the cache. When the control flow is transferred to a different function, the target function is fetched into the method cache if it is not already present. Assuming that there are no jumps in and out of functions, this happens only at call and return instructions. Call and return instructions must therefore load the target function into the cache on a method cache miss. The instructions stall the pipeline until the target method is fully loaded into the method cache. The call instruction must also update the return information for the function return. Return information is composed of the (absolute) base address of the caller and the (relative) offset of the call instruction within the caller. The return base address is required by the return instruction to load the correct target function into the cache. The size of the function to load is stored in a word before the function code.

The cache is organised in cache blocks in a ring buffer. Functions are loaded into consecutive blocks. Cache blocks are not shared between different functions. On a cache miss, the function is loaded into the blocks following the most recently loaded function. The method cache therefore implements a first-in-first-out (FIFO) replacement policy. If a function in the cache is partially overwritten by

a new entry, all blocks of the old function are cleared. The method cache therefore only contains complete functions. Schoeberl et al. presented a cache analysis of the method cache in [36].

One advantage of the method cache is that instruction cache miss costs only appear at call and return instructions. The cache miss costs are proportional to the size of the functions to load. All other instructions are guaranteed instruction cache hits. It also simplifies the fetch stage since it can assume that the instructions are in the cache. It also allows to perform a burst load to fetch the instructions into the cache, which is usually faster than fetching single instruction words from main memory.

If a method is loaded to the cache, but some part of the method is not executed until it is evicted from the cache, then that part of the function will contribute to the cache miss costs and thus decrease the performance of the instruction cache. To reduce the penalty of loading whole functions into the method cache when only parts thereof are accessed, Patmos implements a variant of the method cache that supports the concept of *subfunctions*. The compiler may split functions into disjoint sequences of code, which are the units for caching. The Patmos ISA provides a function-local *branch with cache-fill* instruction in addition to the (subfunction local) branch, which interacts with the method cache but does not update return information. From the point of view of the method cache, there is no difference between a function and a subfunction. The compiler however generates only a simple branch instruction to enter a subfunction, while calling a function also involves generating code to perform parameter passing and stack management.

Table 1 shows the dichotomy of the different types of branch instructions.

	method-cache fill on miss	guaranteed hit
store return information	call	-
no return information	branch w/ cache fill	local branch

Table 1: Overview of call and branch instructions

A FIFO replacement strategy is not optimal from an analysis point of view, since FIFO is prone to timing anomalies [27]. However, while an LRU replacement strategy would allow for much better analysis, it is not clear if an LRU replacement strategy is feasible to implement for the method cache. Since the functions are not of equal size, more than one function may need to be removed on a cache miss to free enough space for the new function. Since the least recently used methods may not be stored in consecutive order in the method cache, new entries need to be stored in non-consecutive cache blocks, which increases the complexity of the fetch stage and the replacement strategy implementation. Therefore the current Patmos design uses a FIFO replacement strategy, which allows for a comparatively simple hardware implementation.

4.3.2 Data Cache

Patmos includes a data cache to reduce the latency of data loads. The ISA provides typed load and store instructions for word, half-word and byte access. For half-word and byte loads, sign-extending loads can be used. Both blocking loads and decoupled loads are available. Store instructions do not allocate new data in the cache. The cache uses a write-through write policy, i.e., a store always writes to main memory. Therefore a load instruction never needs to write back data to main memory on a

cache miss first. While this can cause a higher number of overall memory transfers compared to a write-back policy, WCET analysis usually performs better for write-through due to the smaller state space that needs to explored [5]. A write buffer is not used.

To prevent data that is not reused from evicting more useful cache entries, the ISA also provides *bypass* loads. A bypass load does not allocate data in the cache. The compiler may use a bypass load to load data that is not reused, or to perform memory loads for which the source address cannot be statically determined and which may therefore reduce the precision of the cache analysis.

Predicated load and store instructions have no effect on the data cache and do not stall the pipeline if the predicate is false.

4.3.3 Stack Cache

The stack cache is a local memory organised as ring buffer that can be accessed from both slots in an instruction bundle. Spilling and filling the stack cache is performed explicitly.

A function may reserve space on top of the stack by using a *reserve* instruction. Typed load and store instructions are used to access the reserved stack cache. Reserved space must be freed using a *free* instruction (e.g., when the function exits). The *reserve* instruction may potentially spill previously reserved data to main memory if needed. The *free* instruction however does not fill previously spilled data back to the stack cache, it just updates the stack pointer that points to the top of the stack. A separate *ensure* instruction must be used to guarantee that the space at the top of the stack resides in the stack cache. The *ensure* may need to fill previously spilled data back to the cache. It does not modify the stack pointer.

It is not allowed to reserve or to ensure space that is larger than the size of the stack cache. It is the responsibility of the compiler to ensure that data resides in the stack cache by using *reserve* and *ensure* instructions before the data is used. The Patmos ISA only defines stack control instructions that use an immediate value for the size of the space to ensure, reserve or free. Therefore, dynamically sized data structures can not be allocated in the stack cache.

The *reserve* and *ensure* instructions are defined as blocking operations, i.e., they stall the pipeline until spilling or filling is completed. All other stack operations never stall the pipeline. Decoupled loads are not required for stack access.

It would be possible to merge the reserve instruction with the call instruction, and the free and ensure instruction with the return instruction. The size of the stack that needs to be reserved or ensured could be stored before the function code similar to the function size. This eliminates the need for stack control instructions in the code, which may reduce the code size. However, the processor would then need to store the stack size of the caller either as part of the method cache entry or in an additional special register which needs to be stored and restored similar to other return information. This reduces the advantage of eliminating the stack control instructions. It also reduces the flexibility for the compiler to ensure stack cache only when needed and prevents prefetching of stack data. Therefore the current Patmos ISA keeps call and return instructions and stack control instructions separate.

4.3.4 Data Scratchpad

The Patmos ISA also provides access to a local data scratchpad. Similar to the data cache, typed load and store instructions are used to access the scratchpad, including word, half-word and byte access and optional sign extension for loads. However, scratchpad instructions are guaranteed not to stall the pipeline. They never incur a memory access to main memory and can therefore be performed in parallel to a main memory access. Decoupled loads are not defined for scratchpad access as they are not required. In contrast to stack cache load and store instructions, scratchpad access is only allowed in the first slot of an instruction bundle though.

To load data from main memory to the scratchpad, the application must load it from main memory first, e.g., by using bypass loads, and then store them to the scratchpad. To improve the performance of large memory transfers from main memory to SPM and back, a direct memory access (DMA) controller might be added in the future (see Section 5.3.1).

The data scratchpad uses the same address space as the main memory. The scratchpads use a different address range than main memory, and in a multi-core setup, the address ranges of all scratchpads of the cores are mutually exclusive. This allows to uniquely address data in any of the data scratchpads or in main memory with a single address. This might be used in the future to directly access remote data scratchpads in a multi-core setup (see Section 5.1.1) to implement communication between cores.

4.4 Function Call Handling

Function calls are coupled both with the method cache and the explicitly managed stack cache.

Ordinary function calls and returns involve argument passing, register (re-)storing and stack frame management. Six general purpose registers are used for argument passing, two general purpose registers for the return value (a 64-bit return value requires two registers). Additional arguments should be passed via a shadow stack, which is described below. About half of the registers of the general purpose register file is caller-saved, and the other half is callee-saved. Also return information, which is held in two special purpose registers, is caller saved and hence must be saved on the stack by the caller before performing the call. The predicate register file is callee saved.

Call stack frame management is performed with *stack control instructions*, which manipulate the stack pointer and perform necessary spilling and filling of the stack cache to or from main memory. A function must *reserve* stack cache memory for the local variables and caller-saved registers in the prologue. This space must be de-allocated by a *free* operation in the function epilogue. If the function contains any calls to other functions, after returning from these, the stack cache memory for the local stack frame must be *ensured*. In case the stack cache contents of the local stack frame have been replaced by the called function (or any function deeper in the call tree), ensuring causes the local stack frame to be filled back into the stack cache from main memory, without actually altering the top-of-stack pointer. Data within a stack frame is always addressed relatively to the top-of-stack pointer.

This scheme imposes following restrictions on the usage of the call stack:

- The stack frame of one function is limited to the capacity of the stack cache.

- Although any location on the call stack maps to a location in main memory, the write-back policy of the stack cache prohibits access of call stack data by other means than loading from the stack cache. In most cases, data on the stack cache and in main memory will be inconsistent, i.e., data from the stack cache has not been written back to main memory. Again, the compiler must ensure that data allocated on the stack cache passed by reference to any callee is only accessed from the stack cache (by accordingly typed load instructions).
- It is not safe to pass pointers to stack-allocated data as arguments to functions called, because any reserving in the callee might evict the data from the stack cache. The approach is only valid if the compiler can ensure the availability of data at uses deeper in the call tree.
- All uses of pointers to stack-allocated data must use stack-typed memory access instructions as the stack cache uses a separate address space.

Due to these restrictions, a *shadow stack* is managed in main memory, which is kept separate from the call stack. Hence, data on the shadow stack is not accessed via the stack cache. This allows to allocate automatic variables and memory of variable size within functions that can be passed to functions deeper in the call tree. On function level, the shadow stack is allocated and de-allocated in the function prologue and epilogue. The general purpose register file contains a register for the (shadow) stack pointer and the (shadow) frame pointer.

4.5 Other Real-Time System Requirements

Generally, time-predictability is not the only property real-time systems must exhibit. Due to their safety-critical context, also properties concerning dependability, fault-tolerance and security must be ensured. Additional architectural support is required to fulfil these properties that are not directly related to timing.

As in the T-CREST project we are concerned with time-predictability, we do not address architectural features targeting real-time system requirements that are not related to timing at the time of writing this report.

4.5.1 Memory Management Unit

Discussions with our industrial use-case partners **GMVIS Skysoft** and **Intecs** revealed that they are using virtual memory for the purpose of achieving spatial isolation between processes. Virtual memory, which requires hardware support in the form of a memory management unit (MMU), provides twofold functionality: On one hand, it achieves spatial separation between processes, as each process has its own private view of memory address space and cannot accidentally overwrite a memory area of another process. On the other hand, paging, i.e., transparent transmission of data between the main memory and secondary storage is handled, to abstract away the fact that main memory is limited. The usage of MMUs is problematic as they are far from being time-predictable at the state-of-the-art. Research into developing a time-predictable MMU is most likely out of the scope of this project. In case the paging capabilities of a memory management unit (MMU) are not required, we could resort to more time-predictable alternatives to achieve spatial isolation between processes, e.g. by employing memory protection registers [4].

5 Architectural Extensions

In this section we discuss various extensions to the current Patmos ISA design that may be added in the future. The extensions presented here are either performance improvements, or extensions to support additional application requirements. They have not been included in the current Patmos design, either because they incur high costs in terms of hardware resources or in terms of implementation effort and it is not yet sufficiently clear if they are required for this project (such as floating-point operations or multi-threading support), or because their implementation and their concrete ISA will be designed at a later time in the project (such as the DMA controller or remote scratchpad access over the network-on-chip).

Since Patmos will be integrated into a multi-core processor, Section 5.1 presents extensions that provide efficient multi-core communication and synchronisation. Section 5.2 discusses changes to the ISA that are required to support interrupts and context-switching. In Section 5.3 we present other possible ISA extensions that target either performance improvements or support for possible user requirements.

5.1 Multi-core Support and Synchronisation

The Patmos processor core will eventually be instantiated multiple times on an FPGA and the instances will be connected to a network on chip (NoC) with multiple ports. Also, a predictable memory controller will be connected to the NoC. At the ISA level, support for multi-core communication and synchronisation must be provided.

In order to support shared-memory synchronisation, hardware primitives for atomic memory access must be provided, which are the basis for higher-level synchronisation constructs implemented in software. Following primitives should be provided:

`SWAP` - Atomically fetch and replace the contents at a memory location. This enables the implementation of simple test-and-set based locks to protect critical sections of code from concurrent access.

`FAI` - Atomically fetch and increment the value at a given memory location. This enables the construction of fair ticket locks and FIFO buffers for data exchange.

`FAD` - Atomically fetch and decrement the value at a given memory location. It is used for the same purpose as FAI. Some architectures offer a more general instruction, e.g. the `LOCK; XADD` instruction of x86, which accepts a register argument and atomically adds the value in the register to the contents at the memory location, and stores the old memory contents in the register.

`CAS` - Compare and swap. Atomically, the contents at a memory location are inspected, and if and only if they are equal to a specified value, the contents are replaced by another specified value. All other primitives can be emulated by CAS, but with reduced time-predictability, e.g., repeatedly executing CAS until the memory contents match a given value, while the number of possible retries needs to be bounded.

5.1.1 Remote Data Scratchpad Access

To allow effective communication between cores and to avoid the bottleneck of all communication taking place through the shared main memory (connected to the NoC with a predictable SDRAM controller), the data scratchpad memories (D-SPM) eventually provide means to be accessed remotely. Recall that the D-SPM of every core has a dedicated address range in the global address space. Then, a core can write to the D-SPM of another core by writing to a address in the range of the SPM of that core. Likewise, a core can read from a remote D-SPM by reading from an address in a non-local D-SPM address range.

5.2 Interrupts and Multi-Threading Support

Interrupts provide means to react to external events with little delay by suspending the current flow of control and switching to an interrupt service routine (ISR). They are commonly used to implement asynchronous I/O interfaces, periodic actions or preemptive scheduling at runtime. It must be possible to disable and (re-)enable interrupts. This way it is possible to guarantee statically that a given sequence of instructions is not disrupted, and any internal state is changed during the execution of the ISR.

5.2.1 Context Switch Support

To enable interrupts and a more flexible system model than a strict one-to-one mapping of tasks (threads) to Patmos cores, support for context-switching is required. At a context switch, all the processor registers including the program counter must be stored to memory such that they can be restored at a later point when resuming the thread of execution. Also, the contents of the stack cache must be replaced at least partially to hold the stack frame of the current function of the newly executing thread. How context-switching affects time-predictability depends on the scheduling policy employed (e.g., cooperative or preemptive) and the measures taken to preserve or store/restore the processor state, including the contents of the caches (e.g. by cache partitioning).

5.3 Further Instruction Set Extensions

5.3.1 DMA Interface

We plan to add a direct memory access (DMA) controller to Patmos, which allows direct block transfers between global shared memory and the local data scratchpad. The controller takes a source address, a target address and the length of the data block to be transferred. Since the global memory and the scratchpad use separate address ranges in the same address space, the controller can determine the source memory and the target memory from the given addresses. This enables a more efficient data transfer between local and global memory. A non-blocking operation mode could be added in addition to blocking DMA transfers. However, it depends on the hardware implementation of the local memories and the connection of Patmos core to the NoC which memory operations can be performed in parallel and hence if the use of non-blocking DMA transfers can be actually beneficial.

5.3.2 Non-Blocking Loads with Implicit Wait on Use

Currently the Patmos ISA specifies either blocking loads or decoupled loads where the result needs to be fetched from a special result register. This can be implemented comparatively simple in hardware, and pipeline stalls due to memory loads can only occur at memory access instructions or at the result fetch instruction.

An alternative approach is to attach a ready flag to all general purpose registers. A load that is a cache miss clears the ready flag of the target register. The ready flag is set again when the load completes. The processor needs to stall the pipeline if the ready flag of any of the operand registers of an instruction bundle is cleared.

This change to the ISA eliminates the need to distinguish between a blocking load (which does not hide any latencies) and a decoupled load (which requires an additional move instruction). The compiler generates code that looks similar to code that uses blocking loads only, while the hardware may hide part of the memory transfer costs. However, any instructions may now cause a pipeline stall, and additional ready flags are required for all general purpose registers. Special registers are still needed to store and restore information such as predicate flags and return information.

5.3.3 Floating-Point Instructions

Although floating-point computation may be relevant in certain signal-processing applications, the costs to implement a floating-point unit in terms of hardware resources are high. Therefore the current Patmos design does not support floating-point operations in hardware.

However, using libraries to implement floating-point operations in software cause large over-approximations by the timing analysis [9]. Hardware floating-point units therefore provide a much better WCET-performance. Depending on the user requirements, floating-point support may be added to Patmos, maybe only to a subset of the cores, due to the high resource usage. On the other hand, floating-point units are not required if the applications solely use integer and fixed-point arithmetic.

6 Summary

In this report, we presented architectural support and the ISA for the generation of time-predictable code. At first, we gave an overview of the abstractions commonly used in real-time systems. From the software side, systems are composed of tasks, and the WCETs of the tasks form part of the schedulability analysis in order to obtain a feasible schedule for the task set at hand. We pointed out the different aspects of time-predictability and that high time-predictability does not necessarily imply a low WCET bound.

In order to identify requirements of a time-predictable architecture, we considered time-predictability at two levels, namely at task level and on system level. WCET analysis is concerned with time-predictability at the task level. We summarised common architectural features that aim at average-case improvement but impede processor behaviour analysis, including superscalar out-of-order pipelines, dynamic branch-prediction, certain cache architectures, etc.

To achieve predictability at the task level, which is the main concern of WCET analysis, all the architectural features that benefit from time being abstracted away at the ISA level must be avoided. We have outlined approaches to make the memory hierarchy predictable. The main idea is separation of (compiler controlled) accesses to predictable and unpredictable memory locations to be able to track the contents of caches with static program analysis more precisely. We also presented the single-path approach, which aims at the reduction of program path complexity, and the architectural prerequisites to support the approach.

In order to be time-predictable at the system level, we argued that spatial or temporal isolation is essential to avoid interferences on timing-relevant hardware state that is shared between tasks. This provides composability in a way that the WCET analysis results obtained for a task in isolation are not invalidated once the tasks are integrated into the system.

We concluded the requirements with design guidelines that allow the construction of an architecture that enables the generation of time-predictable code. These guidelines were followed for the Patmos processor core. We gave an overview of its architectural features that aim to achieve time-predictability and how they are reflected in the ISA. One of the key features is the split-cache architecture that allows to specify the target cache of data fetched from main memory with typed load instructions. Predication will help to reduce the penalties at conditional branches, enable region formation techniques and the generation of single-path code.

As some architectural aspects of the network-on-chip will be defined at a later stage in the project, we considered them as extensions to the current core architecture. Multi-threading capabilities and synchronisation facilities fall in the class of these features.

This report clarified how the Patmos processor core will enable and aid the compiler in the generation of time-predictable code. With this first version of the architecture specification the stage is set for achieving time-predictability at the task level. For a detailed ISA specification, we refer to the appendix.

A The Architecture of Patmos

A.1 Pipeline

The pipeline consist of 5 stages: (1) instruction fetch (IF), (2) decode and register read (DR), (3) execute (EX), (4) memory access, and (5) register write back (WB).

Some instructions define additional pipeline stages. Multiplication instructions are executed, starting from the EX stage, in a parallel pipeline with fixed-length (see the instruction definition). The respective stages are referred to by $EX_1, \ldots, EX_n$.

A.2 Register Files

The register files available in Patmos are depicted by Figure 3. In short, Patmos offers:

- 32, 32-bit general-purpose registers (R) : r0, ..., r31
  r0 is read-only, set to zero (0)
- 16, 32-bit special-purpose registers (S): s0, ..., s15
- 8, single-bit predicate registers (P): p0, ..., p7,
  p0 is read-only, set to true (1).

All register reads to the R, S, and P register files are executed in the DR stage. Register writes to R are performed in the MW stage, while S and P are written immediately in the EX stage.

Concurrently writing and reading the same register in the same cycle will, for the read, yield the value that is about to be written.

When writing concurrently to the same register, i.e., the two instructions of the current bundle have the same destination register, the value of the second slot is taken, unless the predicate of that instruction evaluates to false (0).

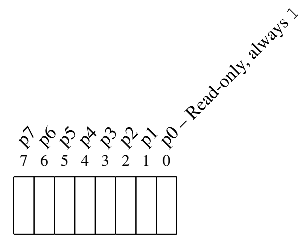
The predicate registers are usually encoded as 4-bit operands, where the most significant bit indicates that the value read from the predicate register file should be inverted before it is used. For predicate operands that are written, this additional bit is omitted.

The special-purpose registers of S allow access to some dedicated registers:

- The lower 8 bits of s0 can be used to save/restore *all* predicate registers at once. The other bits of that register are currently reserved, but not used.
- s1 can also be accessed through the name sm and represents the result of a decoupled load operation. The value is already sign-/zero-extended according to the load instruction.
- s2 and s3 can also be accessed through the names sl and sh and represent the lower and upper 32-bits a multiplication.
- s4 and s5 can also be accessed through the names sb and so. sb denotes the base address of the method containing the most recently executed call instruction. so denotes the offset within that method of the bundle following the bundle containing the call instruction.
- s6 can also be accessed through the name st and represents a pointer to the top-most element of the content of the stack cache spilled to main memory.

31 … 0
r0 (zero, read-only)
r1 (result, scratch)
r2 (result 64-bit, scratch)
r3 (argument 1, scratch)
r4 (argument 2, scratch)
r5 (argument 3, scratch)
r6 (argument 4, scratch)
r7 (argument 5, scratch)
r8 (argument 6, scratch)
r9 (scratch)
r10 (scratch)
r11 (scratch)
r12 (scratch)
r13 (scratch)
r14 (scratch)
r15 (scratch)
r16 (scratch)
r17 (scratch)
r18 (scratch)
r19 (scratch)
r20 (saved)
r21 (saved)
r22 (saved)
r23 (saved)
r24 (saved)
r25 (saved)
r26 (saved)
r27 (saved)
r28 (saved)
r29 (saved)
r30 (frame pointer, saved)
r31 (stack pointer, saved)

(a) General-Purpose Registers (`R`)



(b) Predicate Registers (`P`)

31 … 8	7 … 0	
reserved	p7 . . . p0	s0
sm		s1
sl		s2
sh		s3
sb		s4
so		s5
st		s6
s7		
s8		
s9		
s10		
s11		
s12		
s13		
s14		
s15		

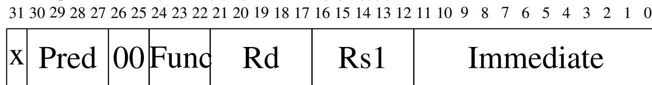
(c) Special-Purpose Registers (`S`)

Figure 3: General-purpose register files, predicate registers, and special-purpose registers of Patmos.

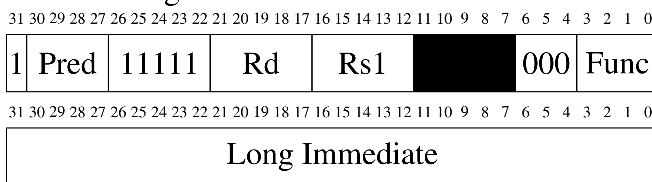
A.3 Instruction Formats

This section gives an overview of all instruction formats defined in the Patmos ISA. Individual instructions of the various formats are defined in the next section. Gray fields indicate bits whose function is determined by a sub-class of the instruction format. Black fields are not used.

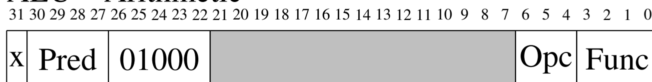
- ALUi – Arithmetic Immediate



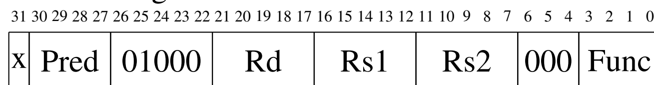
- ALUl – Long Immediate



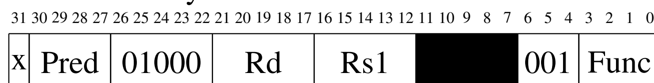
- ALU – Arithmetic



  - ALUr – Register

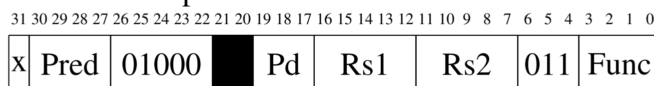


  - ALUu – Unary

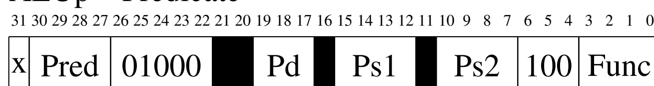


  - ALUm – Multiply

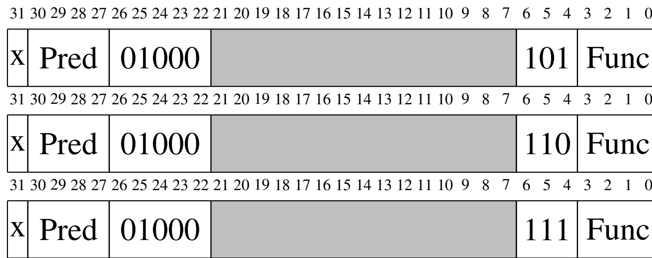
  - ALUc – Compare



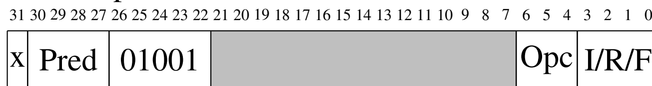
  - ALUp – Predicate



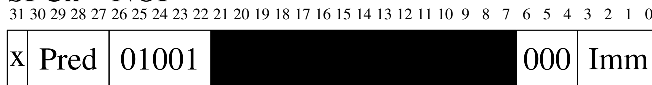
  - Unused



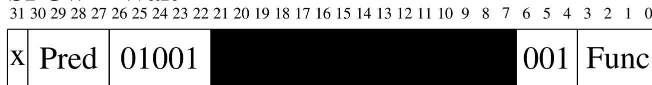
- SPC – Special



  - SPCn – NOP

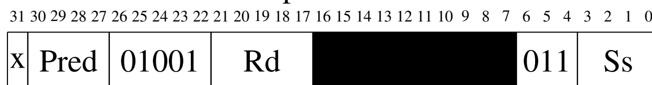


  - SPCw – Wait

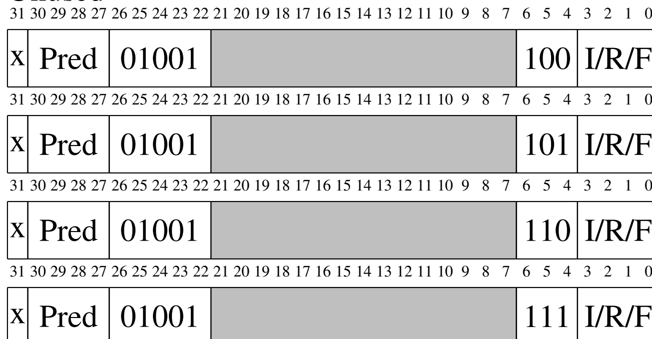


  - SPCt – Move To Special

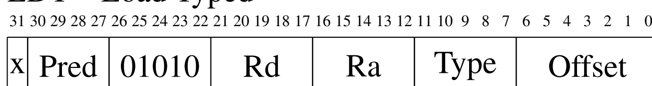
  - SPCf – Move From Special



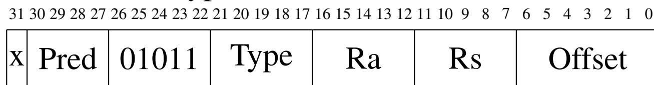
  - Unused



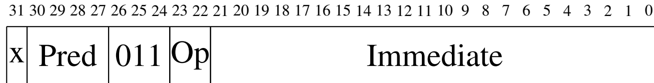
- LDT – Load Typed



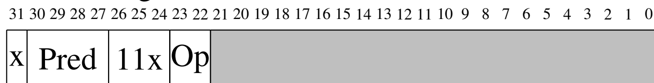
- STT – Store Typed



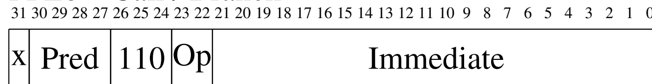
- STC – Stack Control



- PFL – Program Flow

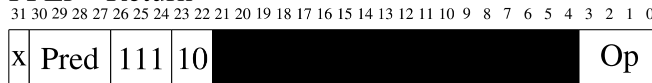


  - PFLb – Call / Branch

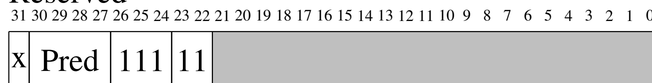


  - PFLi – Call / Branch Indirect

  - PFLr – Return



  - Reserved



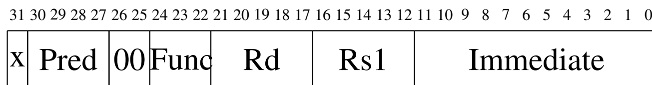
A.4 Instruction Opcodes

This section defines the instruction set architecture, the instruction opcodes, and the behaviour of the respective instructions of Patmos.

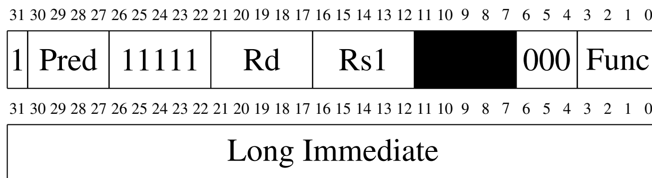
A.4.1 Binary Arithmetic

Applies to the ALUi, ALUl, and ALUr formats. Operand `Op2` denotes either the `Immediate` operand, `Long Immediate` operand, or `Rs2` register operand. For the ALUi format only the first 8 opcodes are supported. The immediate operands are zero-extended. For shift and rotate operations, only the lower 5 bits of the operand are considered.

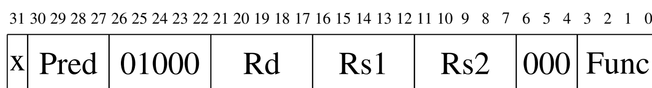
- ALUi – Arithmetic Immediate



- ALUl – Long Immediate



- ALUr – Register



Func	Name	Semantics
0000	add	`Rd = Rs1 + Op2`
0001	sub	`Rd = Rs1 - Op2`
0010	rsub	`Rd = Op2 - Rs1`
0011	sl	`Rd = Rs1 << ` $Op2_{[0:4]}$
0100	sr	`Rd = Rs1 >>> ` $Op2_{[0:4]}$
0101	sra	`Rd = Rs1 >> ` $Op2_{[0:4]}$
0110	or	`Rd = Rs1 | Op2`
0111	and	`Rd = Rs1 & Op2`
1000	rl	`Rd = (Rs1 << `$Op2_{[0:4]}$`) |` `(Rs1 >>> (32-`$Op2_{[0:4]}$`))`
1001	rr	`Rd = (Rs1 >>> `$Op2_{[0:4]}$`) |` `(Rs1 << (32-`$Op2_{[0:4]}$`))`
1010	xor	`Rd = Rs1 ^ Op2`
1011	nor	`Rd = ~(Rs1 | Op2)`
1100	shadd	`Rd = (Rs1 << 1) + Op2`
1101	shadd2	`Rd = (Rs1 << 2) + Op2`
1110	—	unused
1111	—	unused

Pseudo Instructions

```
- mov Rd = Rs ... add Rd = Rs + 0
- neg Rd = -Rs ... rsub Rd = 0 - Rs
- not Rd = ~Rs ... nor Rd = ~(Rs | Rs)
- zext8 Rd = (uint8_t)Rs ... and Rd = Rs & 0xFF)
- li Rd = Immediate ... add Rd = r0 + Immediate)
- li Rd = Immediate ... sub Rd = r0 - Immediate)
```

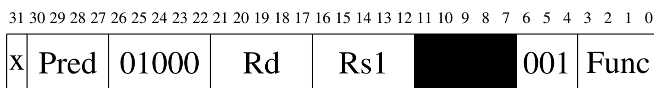
Behaviour

IF –
DR Read register operands `Pred`, `Rs1`, and `Rs2` if needed. If needed zero-extend the `Immediate` operand.
EX By-pass values for `Rs1` and `Rs2`, if needed.
Perform computation (see above).
Derive write-enable signal for destination register `Rd` from predicate `Pred`.
Supply result value for by-passing to `EX` stage.
MW Write to destination register `Rd`.
Supply result value for by-passing to `EX` stage.

A.4.2 Unary Arithmetic

Applies to the ALUu format only.

- ALUu – Unary



Func	Name	Semantics
0000	sext8	`Rd = (int8_t)Rs1`
0001	sext16	`Rd = (int16_t)Rs1`
0010	zext16	`Rd = (uint16_t)Rs1`
0011	abs	`Rd = abs(Rs1)`
0100	—	unused
...	...	...
1111	—	unused

Behaviour

IF –
DR Read register operands `Pred` and `Rs1`.
EX By-pass value for `Rs1`.
Perform computation (see above).
Derive write-enable signal for destination register `Rd` from predicate `Pred`.
Supply result value for by-passing to `EX` stage.
MW Write to destination register `Rd`.
Supply result value for by-passing to `EX` stage.

A.4.3 Multiply

Applies to the ALUm format only. Multiplications are executed in parallel with the regular pipeline and finish within a fixed number of cycles.

- ALUm – Multiply

Func	Name	Semantics
0000	mul	`sl = Rs1 * Rs2;`
		`sh = (Rs1 * Rs2) >>> 32`
0001	mulu	`sl = (uint32_t)Rs1 *`
		`(uint32_t)Rs2;`
		`sh = ((uint32_t)Rs1 *`
		`((uint32_t)Rs2) >>> 32`
0010	—	unused
...	...	...
1111	—	unused

Behavior

`IF`	–
`DR`	Read register operands `Pred`, `Rs1`, `Rs2`.
`EX`	By-pass values for `Rs1` and `Rs2`.
	Derive write-enable signal for destination registers `sl` and `sh` from predicate `Pred`.
	Perform multiplication (see above).
$\texttt{EX}_1$	Perform multiplication.
...	
$\texttt{EX}_n$	Perform multiplication.
	Write to destination registers `sl` and `sh`.

Note

Multiplies may only be issued in the first slot of a bundle.

Multiplies are pipelined, it is thus possible to issue a multiplication on every cycle.

A.4.4 Compare

Applies to the ALUc format only.

- ALUc – Compare

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

x	Pred	01000	■	Pd	Rs1	Rs2	011	Func

Func	Name	Semantics
0000	cmpeq	`Pd = Rs1 == Rs2`
0001	cmpneq	`Pd = Rs1 != Rs2`
0010	cmplt	`Pd = Rs1 < Rs2`
0011	cmple	`Pd = Rs1 <= Rs2`
0100	cmpult	`Pd = Rs1 < Rs2, unsigned`
0101	cmpule	`Pd = Rs1 <= Rs2, unsigned`
0110	btest	`Pd = (Rs1 & (1 << Rs2)) != 0`
0111	—	unused
...	...	...
1111	—	unused

Pseudo Instructions

- `mov Pd = Rs` ... `cmpneq Pd = Rs != r0`
- `cmpz Pd = Rs == 0` ... `cmpeq Pd = Rs == r0`
- `cmpnz Pd = Rs == 0` ... `cmpneq Pd = Rs != r0`
- `cmpgt Pd = Rs1 > Rs2` ... `cmplt Pd = Rs2 < Rs1`
- `cmpge Pd = Rs1 >= Rs2` ... `cmple Pd = Rs2 <= Rs1`
- `cmpugt Pd = Rs1 > Rs2` ... `cmpult Pd = Rs2 < Rs1`
- `cmpuge Pd = Rs1 >= Rs2`..`cmpule Pd = Rs2 <= Rs1`
- `isodd Pd = Rs1` ... `btest Pd = Rs1[r0]`

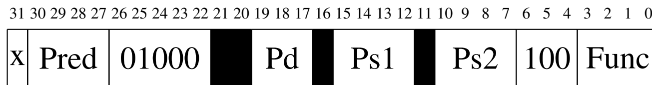
Behaviour

`IF` –
`DR` Read register operands `Pred`, `Rs1`, and `Rs2`.
`EX` By-pass values for `Rs1` and `Rs2`, if needed.
Perform comparison (see above).
Derive write-enable signal for destination register `Pd` from predicate `Pred`.
Write to destination register `Pd`.
`MW` –

A.4.5 Predicate

Applies to the ALUp format only, the opcodes correspond to those of the ALU operations on general purpose registers.

- ALUp – Predicate



Func	Name	Semantics
0000	—	unused
...	...	...
0101	—	unused
0110	or	`Pd = Ps1	Ps2`
0111	and	`Pd = Ps1 & Ps2`
1000	—	unused
1001	—	unused
1010	xor	`Pd = Ps1 ^ Ps2`
1011	nor	`Pd = ~(Ps1 | Ps2)`
1100	—	unused
...	...	...
1111	—	unused

Pseudo Instructions

- `mov Pd = Ps` ... `or Pd = Ps | Ps`
- `not Pd = ~Ps` ... `nor Pd = ~(Ps | Ps)`
- `set Pd = 1` ... `or Pd = p0 | p0`
- `clr Pd = 0` ... `xor Pd = p0 ^ p0`

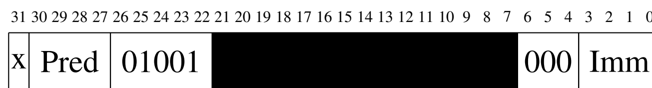
Behaviour

IF –
DR Read register operands `Pred`, `Ps1`, and `Ps2`.
EX By-pass values for `Ps1` and `Ps2`, if needed.
Perform predicate computation (see above).
Derive write-enable signal for destination register `Pd` from predicate `Pred`.
Write to destination register `Pd`.
MW –

A.4.6 NOP

Applies to the SPCn format only. Issue `Imm` NOP operations to the pipeline. The main idea is to save code size, however, the multi-cycle NOP might also be used to enforce certain timing constraints, e.g., wait for a certain amount of time, ensure a minimal execution time, et cetera.

- SPCn – NOP



Behaviour

`IF` –
`DR` Read register operands `Pred`.
`tmp` = `Pred` ? `Imm` : 0.
while (`tmp`-- != 0) { stall `DR`; next cycle; }
`EX` –
`MW` –

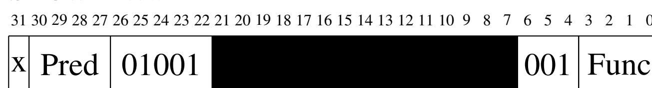
Note

A NOP can only be issued at the first position within a bundle. This does in no way restrict the use of the NOP. It can always be scheduled in the next/previous bundle, while incrementing/decrementing the number of NOP cycles accordingly; the code size is in both cases not increased.

A.4.7 Wait

Applies to the SPCw format only. Wait for a memory operation to complete by stalling the pipeline.

- SPCw – Wait



Func	Name	Semantics
0000	wait.mem	Wait for a memory access
0001	—	unused
...	...	...
1111	—	unused

Behaviour

IF –
DR Read register operands `Pred`.
while (~Pred & ~finished) { stall `DR`; next cycle; }
EX –
MW –

Note

A Wait can only be issued at the first position within a bundle.

A.4.8 Move To Special

Applies to the SPCt format only. Copy the value of a general-purpose register to a special-purpose register.

- SPCt – Move To Special

Name	Semantics
mts	`Sd = Rs1`

Behaviour

IF –
DR Read register operands `Pred` and `Rs1`.
EX By-pass value for `Rs1`.
Derive write-enable signal for destination register `Sd` from predicate `Pred`.
Write to destination register `Sd`.

MW –

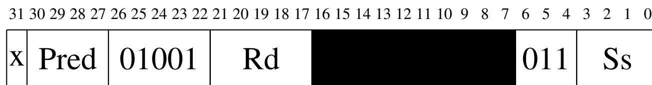
Note

Special registers `sm`, `sl`, `sh` are intended to be only read, writing to those registers may result in undefined behaviour.

A.4.9 Move From Special

Applies to the SPCf format only. Copy the value of a special-purpose register to a general-purpose register.

- SPCf – Move From Special



Name	Semantics
mfs	`Rd = Ss`

Behaviour

`IF` –
`DR` Read register operands `Pred` and `Ss`.
`EX` Derive write-enable signal for destination register `Rd` from predicate `Pred`.
Supply result value for by-passing to `EX` stage.
`MW` Write to destination register `Rd`.
Supply result value for by-passing to `EX` stage.

A.4.10 Load Typed

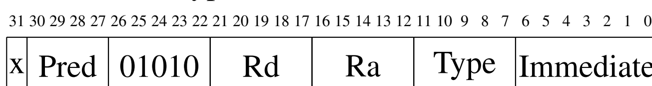
Applies to the LDT format only. Load from a memory or cache. In the table accesses to the stack cache are denoted by `sc`, to the local scratchpad memory by `lm`, to the date cache by `dc`, and to the global shared memory by `gm`. By default all load variants are considered with an implicit `wait` – which causes the load to stall until the memory access is completed.

In addition, *decoupled* loads are provided that do *not wait* for the memory access to be completed. The result is then loaded into the special register `sm`. A dedicated `wait.mem` instruction can be used to stall the pipeline explicitly until the load is completed.

If a decoupled load is executed while another decoupled load is still in progress, the pipeline will be stalled implicitly until the already running load is completed before the next memory access is issued. The value of the previous load can then be read from `sm` for (at least) one cycle.

All loads, decoupled and regular, incur a one cycle load-to-use latency that has to be respected by the compiler/programmer. The destination register of the load is guaranteed to be unmodified, i.e., a one-cycle load delay slot.

- LDT – Load Typed



Type	Name	Semantics
000 ∣ 00	lws	`Rd=sc[Ra+Imm << 2]`$_{32}$
000 ∣ 01	lwl	`Rd=lm[Ra+Imm << 2]`$_{32}$
000 ∣ 10	lwc	`Rd=dc[Ra+Imm << 2]`$_{32}$
000 ∣ 11	lwm	`Rd=gm[Ra+Imm << 2]`$_{32}$
001 ∣ 00	lhs	`Rd=(int32_t)sc[Ra+Imm << 1]`$_{16}$
001 ∣ 01	lhl	`Rd=(int32_t)lm[Ra+Imm << 1]`$_{16}$
001 ∣ 10	lhc	`Rd=(int32_t)dc[Ra+Imm << 1]`$_{16}$
001 ∣ 11	lhm	`Rd=(int32_t)gm[Ra+Imm << 1]`$_{16}$
010 ∣ 00	lbs	`Rd=(int32_t)sc[Ra+Imm]`$_{8}$
010 ∣ 01	lbl	`Rd=(int32_t)lm[Ra+Imm]`$_{8}$
010 ∣ 10	lbc	`Rd=(int32_t)dc[Ra+Imm]`$_{8}$
010 ∣ 11	lbm	`Rd=(int32_t)gm[Ra+Imm]`$_{8}$
011 ∣ 00	lhus	`Rd=(uint32_t)sc[Ra+Imm << 1]`$_{16}$
011 ∣ 01	lhul	`Rd=(uint32_t)lm[Ra+Imm << 1]`$_{16}$
011 ∣ 10	lhuc	`Rd=(uint32_t)dc[Ra+Imm << 1]`$_{16}$
011 ∣ 11	lhum	`Rd=(uint32_t)gm[Ra+Imm << 1]`$_{16}$
100 ∣ 00	lbus	`Rd=(uint32_t)sc[Ra+Imm]`$_{8}$
100 ∣ 01	lbul	`Rd=(uint32_t)lm[Ra+Imm]`$_{8}$
100 ∣ 10	lbuc	`Rd=(uint32_t)dc[Ra+Imm]`$_{8}$
100 ∣ 11	lbum	`Rd=(uint32_t)gm[Ra+Imm]`$_{8}$
1010 ∣ 0	dlwc	`sm=dc[Ra+Imm << 2]`$_{32}$
1010 ∣ 1	dlwm	`sm=gm[Ra+Imm << 2]`$_{32}$
1011 ∣ 0	dlhc	`sm=(int32_t)dc[Ra+Imm << 1]`$_{16}$
1011 ∣ 1	dlhm	`sm=(int32_t)gm[Ra+Imm << 1]`$_{16}$
1100 ∣ 0	dlbc	`sm=(int32_t)dc[Ra+Imm]`$_{8}$
1100 ∣ 1	dlbm	`sm=(int32_t)gm[Ra+Imm]`$_{8}$
1101 ∣ 0	dlhuc	`sm=(uint32_t)dc[Ra+Imm << 1]`$_{16}$
1101 ∣ 1	dlhum	`sm=(uint32_t)gm[Ra+Imm << 1]`$_{16}$
1110 ∣ 0	dlbuc	`sm=(uint32_t)dc[Ra+Imm]`$_{8}$
1110 ∣ 1	dlbum	`sm=(uint32_t)gm[Ra+Imm]`$_{8}$
11110	—	unused
11111	—	unused

Behaviour – regular Load

`IF` –
`DR` Read register operands `Pred` and `Ra`.
`EX` By-pass value for `Ra`.
Derive write-enable signal for destination register `Rd` from predicate `Pred`.
Begin memory access.
`MW` Finish memory access.
while (~Pred & ~finished) { stall `MW`; next cycle; }
Write to destination register `Rd`.
Supply result value for by-passing to `EX` stage.

Behaviour – decoupled Load

`IF`	–
`DR`	Read register operands `Pred` and `Ra`. while (~finished) { stall `DR`; next cycle; }
`EX`	By-pass value for `Ra`. Derive write-enable signal for destination register `sm` from predicate `Pred`. Begin memory access.
`MW`	Finish memory access. Write to destination register `sm`.

Note

Loads to the stack cache can be issued, also concurrently, on both slots of an instruction bundle. All other loads can only be issued on the first slot.

Two successive decoupled loads can be used in the following manner without the use of an additional `wait` instruction:

```
dlwc sm = [r1 + 5];

...

dlwc sm = [r2 + 7] || mfs r2 = sm;
```

A.4.11 Store Typed

Applies to the STT format only. Store to a memory or cache. In the table accesses to the stack cache are denoted by `sc`, to the local scratchpad memory by `lm`, to the date cache by `dc`, and to the global shared memory by `gm`.

- STT – Store Typed

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

x	Pred	01011	Type	Ra	Rs	Offset

Type	Name	Semantics
000 | 00	sws	`sc[Ra+Imm << 2]`$_{32}$ `= Rs`
000 | 01	swl	`lm[Ra+Imm << 2]`$_{32}$ `= Rs`
000 | 10	swc	`dc[Ra+Imm << 2]`$_{32}$ `= Rs`
000 | 11	swm	`gm[Ra+Imm << 2]`$_{32}$ `= Rs`
001 | 00	shs	`sc[Ra+Imm << 1]`$_{16}$ `= Rs`$_{[0:16]}$
001 | 01	shl	`lm[Ra+Imm << 1]`$_{16}$ `= Rs`$_{[0:16]}$
001 | 10	shc	`dc[Ra+Imm << 1]`$_{16}$ `= Rs`$_{[0:16]}$
001 | 11	shm	`gm[Ra+Imm << 1]`$_{16}$ `= Rs`$_{[0:16]}$
010 | 00	sbs	`sc[Ra+Imm]`$_{8}$ `= Rs`$_{[0:8]}$
010 | 01	sbl	`lm[Ra+Imm]`$_{8}$ `= Rs`$_{[0:8]}$
010 | 10	sbc	`dc[Ra+Imm]`$_{8}$ `= Rs`$_{[0:8]}$
010 | 11	sbm	`gm[Ra+Imm]`$_{8}$ `= Rs`$_{[0:8]}$
01100	—	unused
...	...	...
11111	—	unused

Behaviour

`IF` –
`DR` Read register operands `Pred`, `Ra`, and `Rs`.
`EX` By-pass values for `Ra` and `Rs`.
Check predicate `Pred`.
Begin memory access.
`MW` Finish memory access.

Note - Global Memory / Data Cache

With regard the data cache, stores are performed using a *write-through* strategy without *write-allocation*. Data that is not available in the cache will not be loaded by stores; but will be updated if it is available in the cache.

Consistency between loads and other stores is assumed to be guaranteed by the memory interface, i.e., memory accesses are handled in-order with respect to a specific processor. This has implications on the bus, Network-on-Chip connection between the processor and the global memory.

Note - Stack Cache

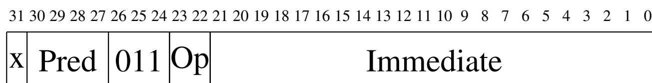
Stores to the stack cache can be issued, also concurrently, on both slots of an instruction bundle. All other stores can only be issued on the first slot.

Two parallel stores within the same bundle writing to the same memory such that the accessed memory cells overlap are not permitted. The content of the respective memory cells is undefined in this case.

A.4.12 Stack Control

Applies to the STC format only. Manipulate the stack frame in the stack cache. `sres` reserves space on the stack, potentially spilling other stack frames to main memory. `sens` ensures that a stack frame is entirely loaded to the stack cache, or otherwise refills the stack cache as needed. `sfree` frees space on the stack frame (without any other side effect, i.e., no spill/fill is executed). All stack control operations are carried out assuming word size, i.e., the immediate operand is multiplied by four.

- STC – Stack Control



Op	Name	Semantics
00	sres	Reserve space on the stack (with spill)
01	sens	Ensure stack space (with refill)
10	sfree	Free stack space.
11	—	unused

Behaviour – Reserve

`IF` –
`DR` Read register operand `Pred`, `st`, and internal stack-cache registers `head` and `tail`.
`EX` Check predicate `Pred`.
Check free space left in the stack cache.
Update stack-cache registers.
`MW` If needed, spill to global memory using `st` and stall.

Behaviour – Ensure

`IF` –
`DR` Read register operand `Pred`, `st`, and internal stack-cache registers `head` and `tail`.
`EX` Check predicate `Pred`.
Check reserved space available in the stack cache.
`MW` If needed, refill from global memory using `st` and stall.

Behaviour – Free

`IF` –
`DR` Read register operand `Pred` and internal stack-cache registers `head`.
`EX` Check predicate `Pred`.
Account for $\texttt{head} - \texttt{tail} < 0$, update `st`.
Update stack-cache register `head`.

MW –

Note

Stack control instructions can only be issued on the first position within a bundle.

It is permissible to use several reserve, ensure, and free operations within the same function.

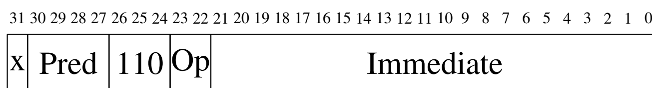
A.4.13 Call and Branch

Applies to PFLb format only. Transfer control to another function or perform function-local branches. `bc` performs a function-local branch relative to the base address of the current function within the method cache. `b` performs a branch relative to the base address of the current function in memory. `bs` performs a function call. `b` and `bs` may cause a cache miss and a subsequent cache refill to load the target code. `bc` is guaranteed to be a cache hit. `bs` and `b` are very similar, the main difference is that `bs` stores return information to implement function calls, while `b` does not.

Branch and call instructions are effectively executed in the `EX` stage. The instructions fetched in the meantime are *not* aborted. This corresponds to a branch delay of 2 that has to be respected by the compiler or assembly programmer.

The immediate operand is interpreted as unsigned for function calls (`bs`) and cache-relative branches (`bc`); for relative branches (`b`) the immediate is interpreted as signed. The target address of branches are computed from the base address of the current code sequence executed in the cache.

- PFLb – Call / Branch



Op	Name	Semantics
00	bs	function call (absolute, with cache fill)
01	bc	local branch (cache relative, always hit)
10	b	local branch (relative, with cache fill)
11	—	unused

Behaviour – call, branch, and system call

IF –
DR Read register operand `Pred`.
EX Check predicate `Pred`.
Store method base and program counter to `sb` and `so`.
Check method cache.
Compute cache-relative program counter.

`MW` If needed, fill method cache and stall.
Update program counter.

Behaviour – branch within cache

`IF` –
`DR` Read register operand `Pred`.
`EX` Check predicate `Pred`.
Assert on method cache.
Compute new, cache-relative program counter value.
`MW` Update program counter.

Note

All branch/call instructions can only be issued on the first position within a bundle.

Offsets and addresses are interpreted in word size.

A.4.14 Call and Branch Indirect

Applies to PFLi format only. Transfer control to another function or within a function. `bsr` and `br` may cause a cache miss and a subsequent cache refill to load the target code; while `bcr` is assumed to be a cache hit.

- PFLi – Call / Branch Indirect

Op	Name	Semantics
0000	bsr	function call (indirect, with cache fill)
0001	bcr	local branch (cache indirect, always hit)
0010	br	local branch (indirect, with cache fill)
0011	—	unused
...	...	...
1111	—	unused

Behaviour – call, branch, and system call

`IF` –
`DR` Read register operand `Pred` and `Rs1`.

EX By-pass value for Rs1.
Check predicate Pred.
Store method base and program counter to sb and so.
Check method cache.
Compute cache-relative program counter value.
MW If needed, fill method cache and stall.
Update program counter.

Behaviour – branch within cache

IF –
DR Read register operand Pred and Rs1.
EX By-pass value for Rs1.
Check predicate Pred.
Assert on method cache.
Compute new, cache-relative program counter value.
MW Update program counter.

Note

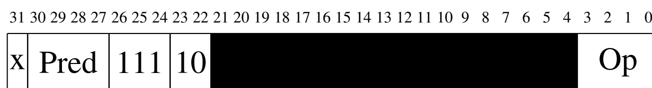
All branch/call instructions can only be issued on the first position within a bundle.

Offsets and addresses are interpreted in word size.

A.4.15 Return

Applies to PFLr format only. Transfer control back to the calling function or return from an interrupt. ret may cause a cache miss and a subsequent cache refill to load the target code.

- PFLr – Return



Op	Name	Semantics
0000	ret	Return from a function (w. cache fill)
0001	—	unused
...	...	...
1111	—	unused

Behaviour

`IF` –
`DR` Read register operand `Pred`, `sb` and `so`.
`EX` Check predicate `Pred`.
Check method cache.
Compute cache-relative program counter value.
`MW` If needed, fill method cache and stall.
Update program counter.

A.5 Stack Cache

The stack cache of Patmos essentially consists of a fast, small, local memory `head` and `tail` pointers into the local memory, and a top-of-stack pointer into the global memory. The structure has some similarities with a ring buffer, reserving and freeing space on the stack moves the `head` pointer, spilling and filling moves the `tail`.

As with regular ring buffers, when the size of the stack cache is not sufficient in order to reserve additional space requested, it needs to spill some data so far kept in the stack cache to the global memory, i.e., whenever $\texttt{head} - \texttt{tail} > \textit{stack cache size}$. A major difference, however, is that freeing space does *not* imply the reloading of data from the global memory. When a free operation frees all stack space currently held in the cache (or more), the special register `st` is accordingly incremented.

Addresses for load and store operations from/to the stack cache are relative to the `head` pointer.

The base address for fill and spill operations of the stack cache is kept in special registers `st`.

The organisation of the stack cache implies some limitations:

- The maximum size of stack data accessible at any moment is limited to the size of the cache. The stack frame can be split, such that at any moment only a subset of the entire stack frame has to be resident in the stack cache, or a *shadow* stack frame in global memory can be allocated.
- When passing pointers to data on the stack cache to other functions it has to be ensured that: (1) the data will be available in the cache, (2) the pointer is only used with load and store operations of the stack cache, and (3) the relative displacement due to reserve and free operations on the stack is known. Alternatively, aliased stack data can be kept on a *shadow* stack in the global memory without restrictions.
- The stack control operations only allow allocating constant-sized junks. Computed array sizes (C 90) and `alloca` with a computed allocation size have to be realised using a *shadow* stack in global memory.
- The calling conventions for functions having a large number of arguments have to be adapted to account for the limitation of the stack cache size (see Section A.7).

A.6 Method Cache

The cache is organised in blocks of a fixed size.

Contiguous sequences of code are cached. These code sequences will often correspond to entire functions. However, functions can be split into smaller junks in order to reduce to overhead of loading

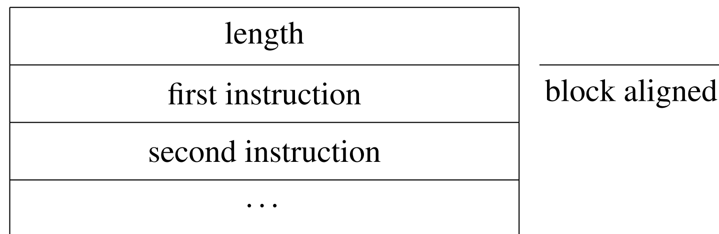


Figure 4: Layout of code sequences intended to be cached in the method cache.

the entire function at once. Code transfers between the respective junks of the original function can be performed using the `b` and `br` instructions.

A code sequence is either kept entirely in the method cache, or is entirely purged from the cache. It is not possible to keep code sequences partially in the cache.

Code intended for caching has to be aligned in the global memory according to the cache's block size. Call and branch instructions do not encode the size of the target code sequence. The size is thus encoded right in front of the first instruction of a code sequence that is intended for caching. Figure 4 illustrates this convention.

The organisation of the method cache implies some limitations:

- The size of a code sequence intended for caching is limited to the size of the method cache. Splitting the function is possible.
- Compiler managed prefetching, if supported, has to ensure that the currently executed code is not purged.

A.6.1 FIFO Replacement

The method cache with FIFO replacement allocates a single junk of contiguous space for a cached code sequence. Every block in the cache is associated with a tag, that corresponds to the base addresses of cached code sequences. However, the tag is only set for the first block of a code sequence. The tags of all other blocks are cleared. This simplifies the purging of cache content when other code is fetched into the cache.

Code is fetched into the cache according to a `FIFO-base` pointer, which points to the first block of the method cache where the code will be placed. After the fetching the code from global memory has completed this pointer is advanced to point to the block immediately following the least recently fetched block.

A.7 Application Binary Interface

A.7.1 Data Representation

Data words in memories are stored using the big-endian data representation, this also includes the instruction representation.

A.7.2 Register Usage Conventions

The register usage conventions for the general purpose registers are as follows:

- `r0` is defined to be zero at all times.
- `r1` and `r2` are used to pass the return value on function calls.
- `r3` through `r8` are used to pass arguments on function calls.
- `r30` and `r31` are defined as the frame pointer and stack pointer for the *shadow* stack in global memory. The use of a frame pointer is optional, the register can freely be used otherwise.
- `r1` through `r19` are caller-saved *scratch* registers.
- `r20` through `r31` are callee-saved *saved* registers.

The usage conventions of the predicate registers are as follows:

- All predicate registers are callee-saved *saved* registers.

The usage conventions of the special registers are as follows:

- The top-of-stack of the stack cache `st` is a callee-saved *saved* register.
- The method base address and offset of the method cache `sb` and `so` are callee-saved *saved* registers.
- The lower 8 bits of `s0`, representing the predicate registers, are are callee-saved *saved* registers. The other bits of the register are reserved and should not be modified.
- All other special registers are scratch registers and should not be used across function calls.

A.7.3 Function Calls

Function calls have to be executed using the `bs` or `bsr` instructions that automatically prefetch the target function to the method cache and store the current method base address and offset to the respective special-purpose registers (`sb` and `so`).

The register usage conventions of the previous section indicate which registers are preserved across function calls.

The first 6 arguments of integral data type are passed in registers, where 64-bit integer and floating point types occupy two registers. All other arguments are passed on the *shadow* stack via the global memory.

When the `sb` and `so` need to be saved to the stack, they have to be saved as the first elements of the function's stack frame, i.e., right after the stack frame of the calling function.

A.7.4 Stack Layout

All stack data in the global memory, either managed by the stack cache or using a frame/stack pointer, grows from top-to-bottom. The use of a frame pointer is optional.

Unwinding of the call stack is done on the stack-cache managed stack frame, following the conventions declared in the previous subsection on function calls.

A.7.5 Instruction Usage Conventions

To simplify the reconstruction of the program's control flow from binary code, the use of multi-cycle `NOP` instructions within branch delay slots should be avoided.

References

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 177–189. ACM, 1983.

[2] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM.

[3] 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.

[4] Joachim Biskup. *Security in Computing Systems - Challenges, Approaches and Solutions*. Springer, 2009.

[5] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, May 2010.

[6] Jean-Francois Deverge and Isabelle Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.

[7] 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.

[8] Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile time decided instruction cache locking using worst-case execution paths. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS*, pages 143–148, Salzburg/Austria, sep 2007.

[9] Gernot Gebhard, Christoph Cullmann, and Reinhold Heckmann. Software structure and wcet predictability. In Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[10] 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.

[11] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proc. Real Time Systems Symposium*, pages 229–237, Santa Monica, CA, USA, Dec. 1989.

[12] Raimund Kirner and Peter Puschner. Obstacles in worst-case execution time analysis. In *Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 333–339, Orlando, Florida, May 2008.

[13] Raimund Kirner and Peter Puschner. Time-predictable computing. In *Proc. 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 23–34, 2010.

[14] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 1997.

[15] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[16] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 12–21, Washington, DC, USA, 1999. IEEE Computer Society.

[17] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 138–150. ACM, 1995.

[18] The Motor Industry Software Reliability Association (MISRA). *Guidelines for the use of the C language in critical systems*. The MISRA Consortium, October 2004.

[19] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE 2007)*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[20] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[21] Peter Puschner. The single-path approach towards WCET-analysable software. In *Proc. IEEE International Conference on Industrial Technology*, pages 699–704, Dec. 2003.

[22] 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.

[23] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Peter Puschner, Raimund Kirner, and Robert G. Pettit. Towards composable timing for real-time software. In *Proc. 1st International Workshop on Software Technologies for Future Dependable Distributed Systems*, pages 1–5, 2009.

[25] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

[26] Peter Puschner and Martin Schoeberl. On composable system timing, task timing, and WCET analysis. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 91–101, Prague, Czech Republic, July 2008. OCG.

[27] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007.

[28] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany.

[29] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the Real-Time Systems Symposium (RTSS 2007)*, pages 49–60, Dec. 2007.

[30] Michel P. Schellekens. MOQA; unlocking the potential of compositional static average-case analysis. *Journal of Logic and Algebraic Programming*, 79(1):61–83, Jan. 2010.

[31] 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.

[32] 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.

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

[34] Martin Schoeberl. Time-predictable chip-multiprocessor design. In *Asilomar Conference on Signals, Systems, and Computers*, Asilomar, CA, November 2010.

[35] 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.

[36] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[37] 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)*, Grenoble, France, March 2011.

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

[39] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *Trans. on Embedded Computing Sys.*, 7(1):1–38, 2007.

[40] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Proceedings of the Fifth International Conference on Quality Software (QSIC2005)*, pages 295–306. IEEE Computer Society, 2005.

[41] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):966 –978, july 2009.

[42] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.