T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 2.1 Software Simulator of Patmos

Version 1.0
14 March 2012
Final

Public Distribution

Technical University of Denmark

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

Project Partner Contact Information

AbsInt Angewandte Informatik Christian Ferdinand Science Park 1 66123 Saarbrücken, Germany Tel: +49 681 383600 Fax: +49 681 3836020 E-mail: ferdinand@absint.com	**Eindhoven University of Technology** Kees Goossens Potentiaal PT 9.34 Den Dolech 2 5612 AZ Eindhoven, The Netherlands E-mail: k.g.w.goossens@tue.nl
GMVIS Skysoft 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

Document Control

Version	Status	Date
0.1	First draft	19 January 2012
0.2	Version for meeting review	8 March 2012
1.0	QA at Dresden meeting	14 March 2012

Executive Summary

This document describes the deliverable *D 2.1 Software Simulator of Patmos* of work package 2 of the T-CREST project, due 6 months after project start as stated in the Description of Work. This document presents the design and implementation of a cycle-accurate instruction set simulator for the Patmos processor.

1 Introduction

Simulation [6] is a popular and effective design aid during the development of a new processor and its instruction set architecture (ISA). A simulator allows to quickly evaluate trade-offs between different design alternatives, measure the impact of modifications to the processor, and evaluate the profitability of extensions to the instruction set. This document describes the ISA simulator for Patmos [5], the processor for the T-CREST platform.

Several choices are available for the implementation of an simulator, First of all, it has to be decided at which granularity the simulation models the underlying processor hardware. One option is to simulate the processor at the lowest-possible level, accounting for individual transistors, wires, and their electrical properties. Simulation at this level allows to gather very detailed information, sometimes including even power consumption and heat dissipation. However, the approach is overly complex and time consuming in practice, in particular in early phases of the design process. A more practical solution is called instruction set simulation (ISS), where merely the effects of executing individual instructions with regard to user-visible processor state, e.g., register files and memories, is modeled. Individual instructions can usually be simulated much more efficiently and also facilitate the development and modification of the processor's simulation model. On the downside, ISS only covers effects with regard to the visible processor state, internal processor state, such as the state of the processor pipeline, caches, etc., is not captured. The attainable accuracy of the simulation results may thus be considerably reduced. In the context of the T-CREST project we decided to use cycle-accurate ISS, which tries to balance between accuracy and simulation overhead by simulating individual instructions and all their impact with respect to timing on the processor pipeline, caches, and memories.

Several options are available to perform the actual simulation. A typical approach is interpretation [3], where the simulation proceeds by fetching and decoding the instruction that ought to be executed next and then invoking one or several simulation functions in order to simulate the behavior of that particular instruction. The main advantage of interpretation is that it is easy to realize, i.e., it is sufficient to provide a basic simulation framework for the main components of the processor (pipeline, registers, caches, memories) and simulation functions for the individual instructions. However, simulation speed is limited as the simulation performs the fetch and decode steps, which are usually rather expensive in terms of execution time, over and over again. Simulators thus sometimes build an abstract internal representation of the program being simulated and cache information on decoded instructions to improve simulation speed [1]. A very advanced way of implementing simulators is compiled simulation [4], also known as binary translation [2] or dynamic binary translation, allows to further improve speed by generating specialized machine code, for the simulation host machine, to simulate sequences of instructions (for instance, basic blocks, memory pages, frequently executed regions, et cetera). Compilation-based techniques have particular advantages when large, long running programs ought to be simulated – the observed improvements are then considerable. The Patmos simulator is intended to serve as a platform to experiment with different design alternatives, evaluate trade-offs, and explore new ideas. We thus opted for the implementation of a flexible and extensible interpreter that can quickly be adopted to whatever needs may arise in the course of the T-CREST project.

2 Patmos Simulator Design

The Patmos simulator is implemented in C++ and consists of classes and abstract interfaces representing the processor components that are designed to be adaptable and extensible. The simulator follows the working draft of the Patmos ISA as defined in the appendix, but is likely to be adapted to fulfill newly arising needs as the project evolves. Figure 1 shows the main classes of the Patmos simulator and the various relationships connecting them. The simulator's main interface is the class `simulator_t`, which can be used to set up and control a simulation session. An instance of the simulator is further connected to the processor's main components: the instruction decoder (`decoder_t`), the register files (`register_file_t`), the caches (`data_cache_t`, `method_cache_t`, `stack_cache_t`), and the memories (`memory_t`).

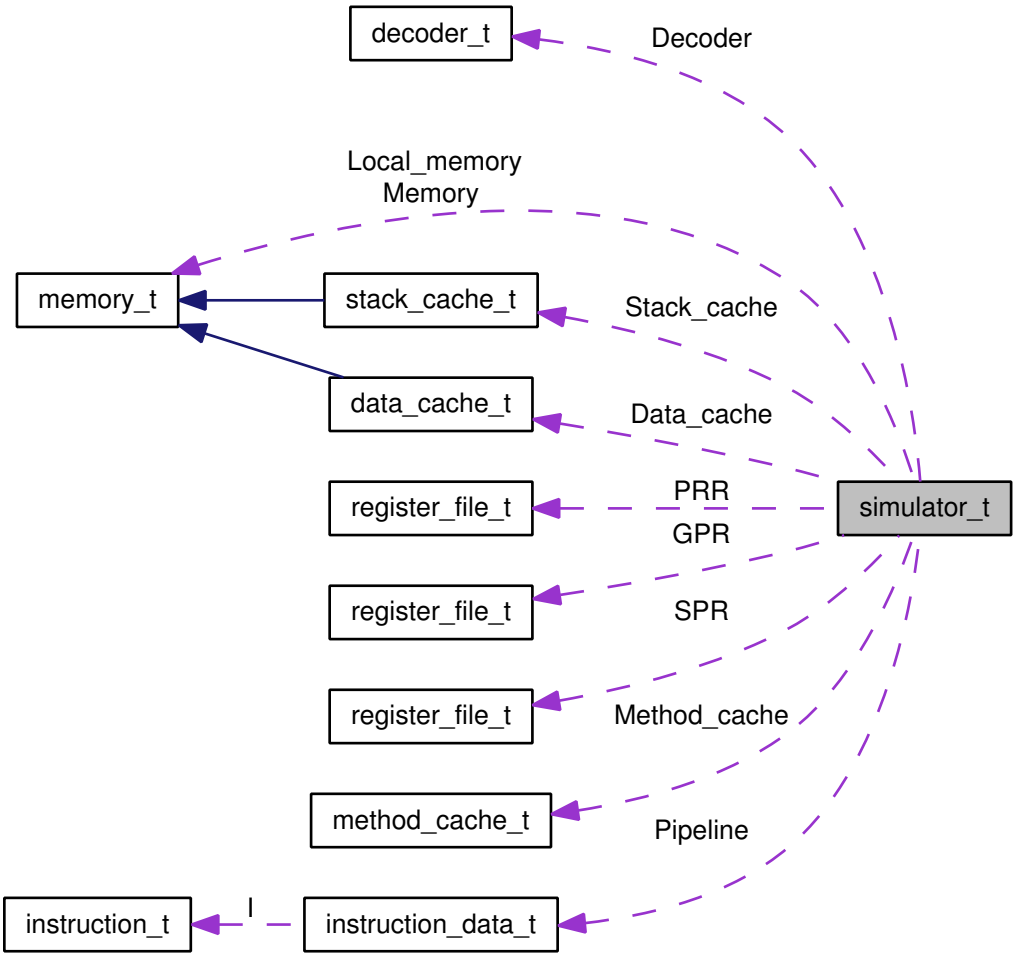


Figure 1: Overview of the Patmos simulator's main components and their collaboration relationships.

The simulation is controlled by the simulation main loop by the `simulator_t` class. On each iteration of the loop the simulated time advances by one cycle and the following simulation steps are performed:

1. Decode the next instruction bundle and increment the program counter accordingly – unless the fetch stage (`IF`) is stalled.
2. For every instruction present in any of the processor's pipeline stages, invoke an interpreter function that simulates the effect of the respective instruction on the processor state.
3. Advance the pipeline state by advancing all currently present instructions to the next pipeline stage. In the case of a pipeline stall, only certain instructions advance to the next stage and pipeline bubbles are injected into the corresponding pipeline stages.
4. Advance the internal state of memories and caches.

3 Instruction Simulation Functions

The Patmos instruction set is represented by a hierarchy of classes derived from the abstract interface `instruction_t`. Each instruction offers 4 interpreter functions corresponding to the pipeline stages of the Patmos processor (`IF`, `DR`, `EX`, and `MW`). The instruction hierarchy allows to reuse patterns common to groups of instructions and thus facilitate the simulator development. Figure 2 shows a large portion of the instruction classes available for simulation. The concrete instruction implementations are omitted due to space considerations, e.g., the `add` instruction of the `ALUr` instruction (see Appendix) format is implemented by the `i_add_t` class, which is derived from `i_alur_t` shown in the figure.

Each instruction is furthermore associated with a binary representation (provided by the `binary_format_t` interface) which allows the easy encoding and decoding of instructions to/from their representation as binary machine code.

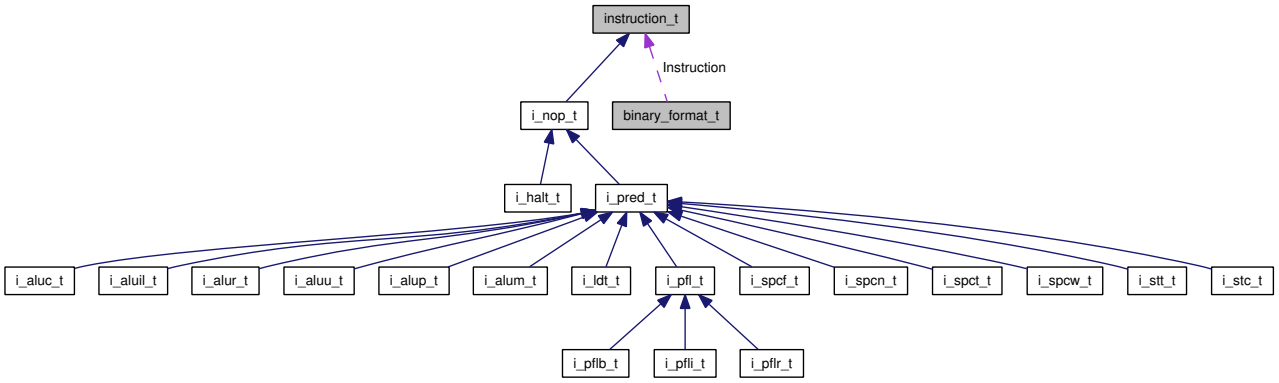


Figure 2: Overview of instruction classes supported by the Patmos simulator as well as their inheritance relations.

4 Memory and Cache Simulation

Memory and cache contents, as well as instructions, are accessed through well-defined interfaces that abstract from the concrete implementation of the underlying memory organization or caching strategy. The main interface, as shown by Figure 3, for memory accesses is provided by the class

memory_t, which is refined to model generic data caches (data_cache_t) as well as the Patmos specific stack cache (stack_cache_t). Instructions are fetched through the method cache (method_cache_t). Based on these concepts implementations of particular caching policies can easily be implemented – as shown, for instance, for the stack and method caches where multiple implementations are available. Since the Patmos design is anticipated to evolve, the main focus of the implementation was on providing an extensible framework to experiment with varying cache and memory configurations.

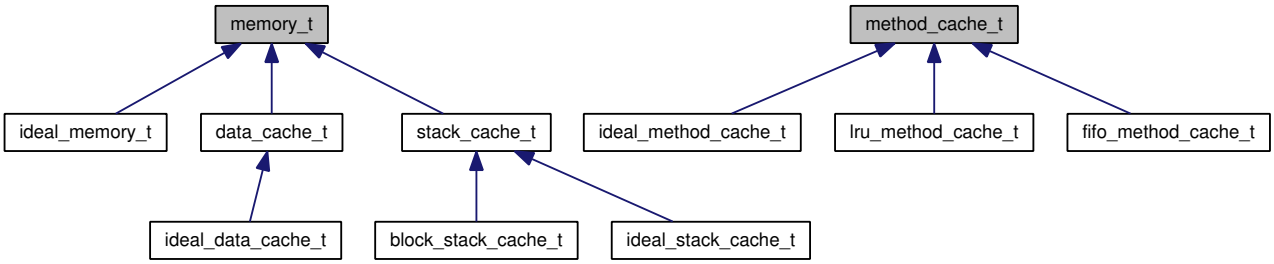


Figure 3: Overview of memory and cache classes supported by the Patmos simulator as well as their inheritance relations.

5 Using the Simulator

The simulator is available as open-source under the GNU General Public License (v.3)[1]. The source code is provided through the Patmos repository via the git source code management tool:

```
https://github.com/schoeberl/patmos
```

5.1 Requirements

In order to retrieve and build the simulator from source code several tools are required:

- git
  A source code management tool.
  http://git-scm.com/
- cmake, version 2.6 or above
  A tool to manage how software is built from source code independent from a particular platform.
  http://www.cmake.org/
- boost, version 1.46 or above
  A rich C++ library.
  http://www.boost.org/
  http://www.boostpro.com/download/ (Windows)
- A C++ compiler.
  http://gcc.gnu.org/
  http://www.microsoft.com/visualstudio/ (Windows)

[1] http://www.gnu.org/licenses/

5.2 Retrieving and Building the Source Code

The source code of Patmos and its simulator can be retrieved as follows. Before starting, make sure the required libraries and tools, listed above, are installed. The shown commands assume a Unix-like platform, other platforms might require slightly different commands.

- Retrieve the source code:

  ```
  git clone git://github.com/schoeberl/patmos.git
  ```

- Enter the simulator directory and create a build directory:

  ```
  cd simulator
  mkdir build
  ```

- Configure the build system:

  ```
  cd build
  cmake ..
  ```

- Build the source code:

  ```
  make
  ```

- Test the simulator:

  ```
  make test
  ```

5.3 Running the Simulator

After successfully building the source code, several tools are available that can be used as follows:

- `paasm`

  A minimal assembler intended for testing purposes. It accepts Patmos instructions, empty lines, and comments in the form of lines starting with a # symbol, i.e., no symbols or relocation are supported.

  usage:
  `paasm <input assembly> <binary stream>`

- `padasm`

  A minimal disassembler intended for testing purposes. It accepts a binary stream and prints the corresponding Patmos instructions.

  usage:
  `padasm <binary stream> <output assembly>`

- `pasim`

  The Patmos simulator. It accepts a binary stream as input, loads the entire content of the

stream into the simulator's main memory, and begins executing the program at main memory address zero. The simulator generates a trace of the program's execution.

usage:
`pasim <binary stream> <trace output>`

A The Architecture of Patmos

A.1 Pipeline

Figure 4 shows an overview of Patmos' pipeline. The pipeline consist of 5 stages: (1) instruction fetch (`IF`), (2) decode and register read (`DR`), (3) execute (`EX`), (4) memory access, 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 $\texttt{EX}_1, \ldots, \texttt{EX}_n$.

A.2 Register Files

The register files available in Patmos are depicted by Figure 5. 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.

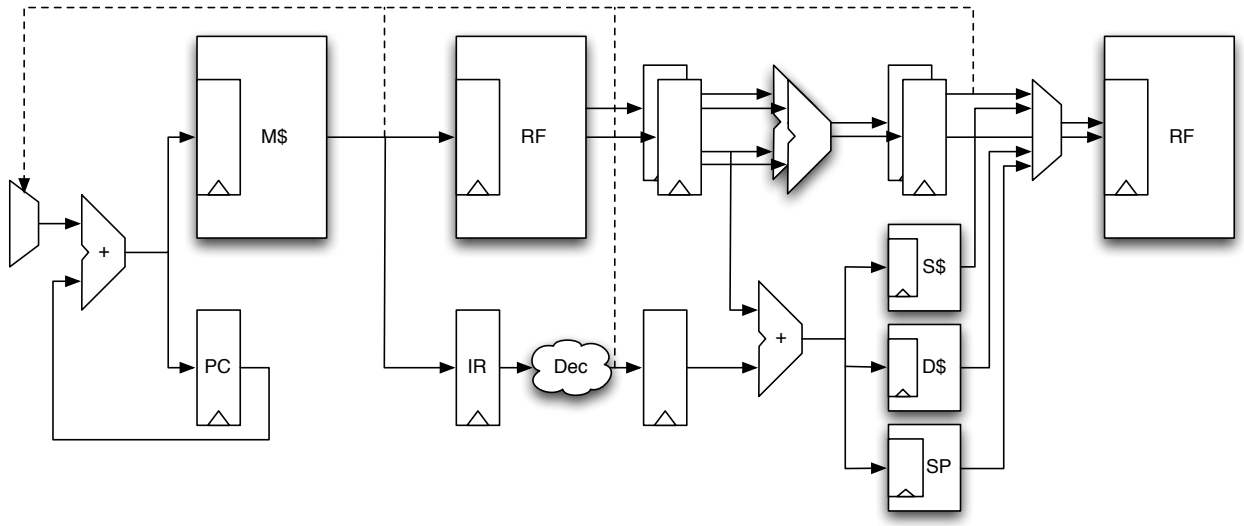


Figure 4: Pipeline of Patmos with fetch, decode, execute, memory, and write back stages.

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).

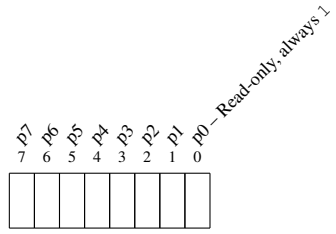
The predicate registers are usually encoded as 4-bit operands, where the most significant bit indicates that the value read from the predicate register file should be inverted before it is used. For predicate operands that are written, this additional bit is omitted.

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 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
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 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	
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)

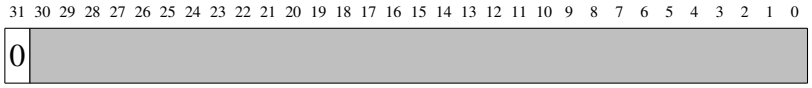
Figure 5: General-purpose register files, predicate registers, and special-purpose registers of Patmos.

B Bundle Formats

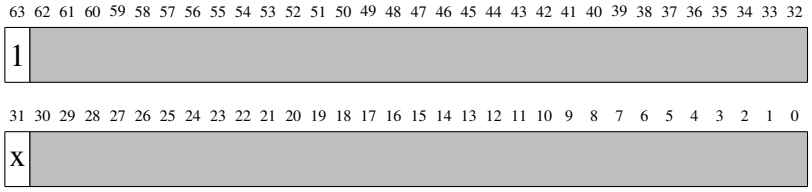
All Patmos instructions are 32 bits wide and are structured according to one of the instruction formats defined in the following section. Up to two instructions can be combined to form an instruction bundle; Patmos bundles are thus either 32 or 64 bits wide. The bundles sizes are recognized by the value of the most significant bit, where 0 indicates a short, 32-bit bundle and 1 a long, 64-bit bundle.

The following figures illustrate these two bundle variants:

- 32-bit bundle format



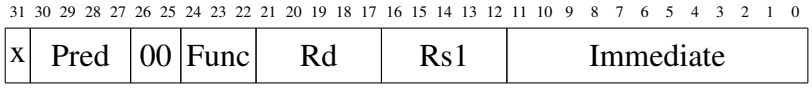
- 64-bit bundle format



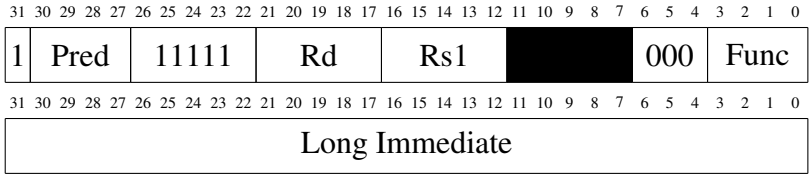
C Instruction Formats

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.

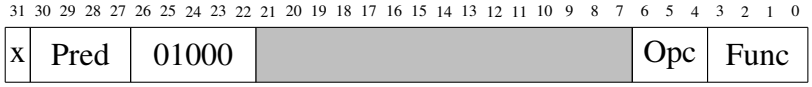
- ALUi – Arithmetic Immediate



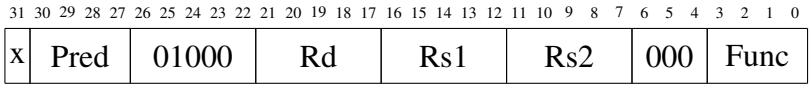
- ALUl – Long Immediate



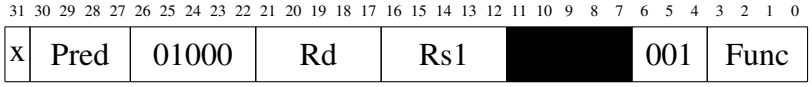
- ALU – Arithmetic



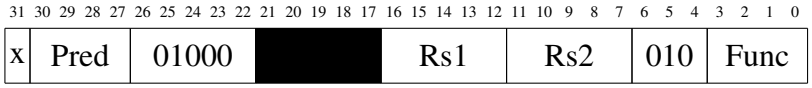
  - ALUr – Register



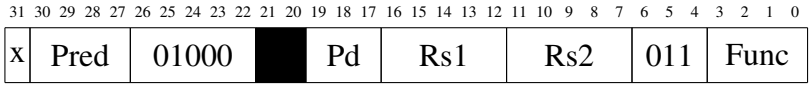
  - ALUu – Unary



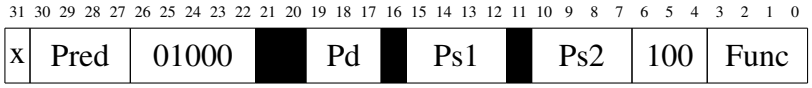
  - ALUm – Multiply



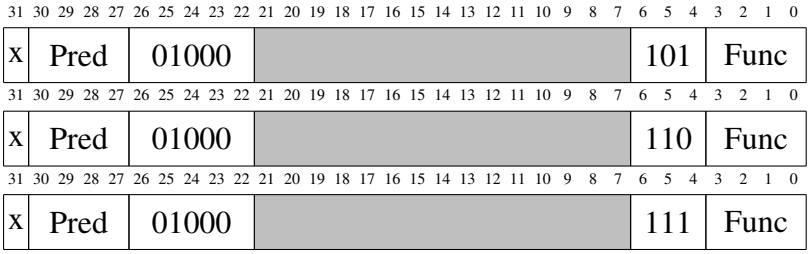
  - ALUc – Compare



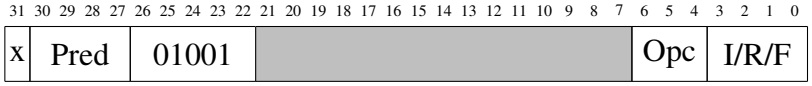
  - ALUp – Predicate



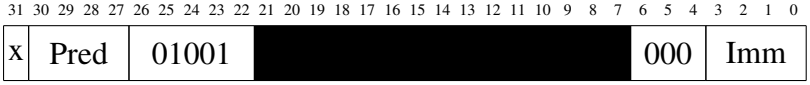
- Unused



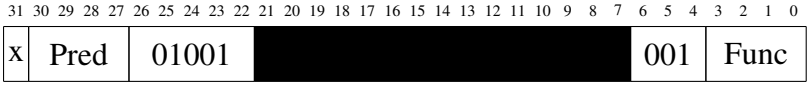
- SPC – Special



  - SPCn – NOP

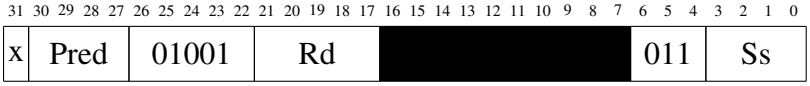


  - SPCw – Wait

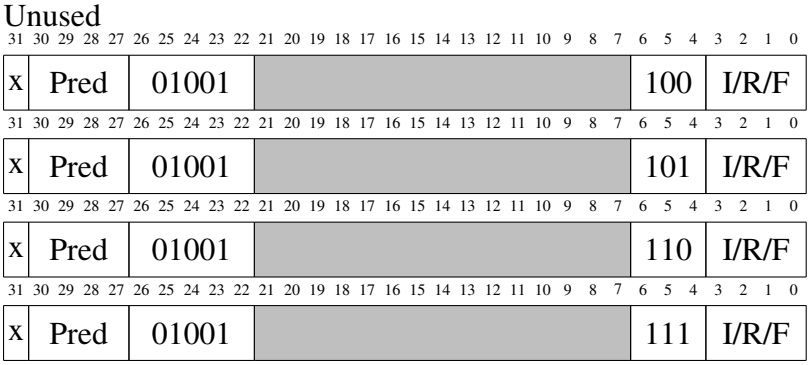


  - SPCt – Move To Special

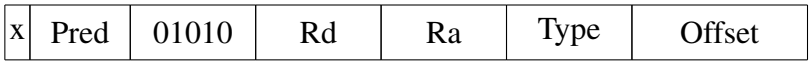
  - SPCf – Move From Special



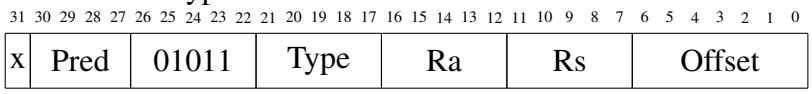
  - Unused



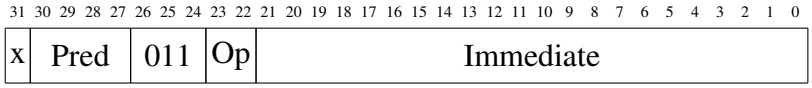
- LDT – Load Typed



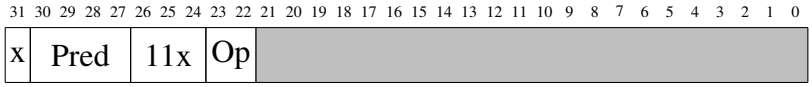
- STT – Store Typed



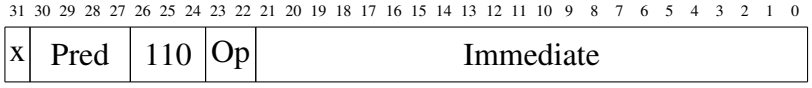
- STC – Stack Control



- PFL – Program Flow

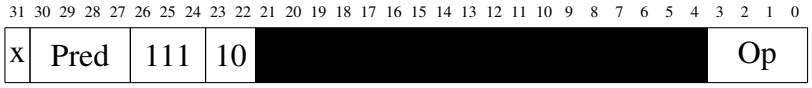


  - PFLb – Call / Branch

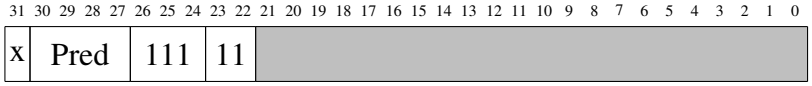


  - PFLi – Call / Branch Indirect

  - PFLr – Return



  - Reserved



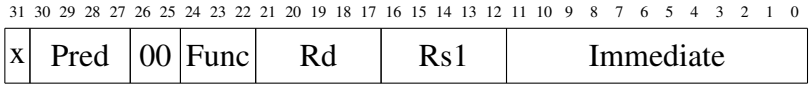
D Instruction Opcodes

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

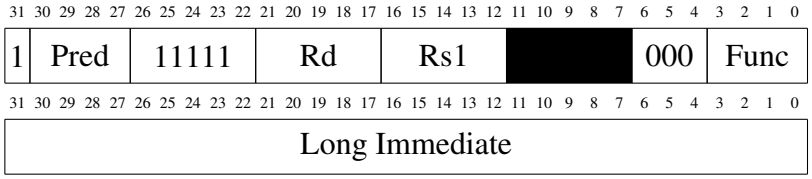
D.1 Binary Arithmetic

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.

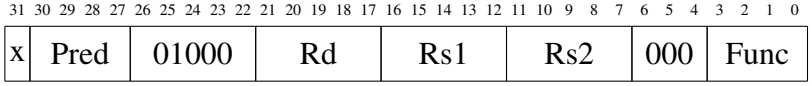
- ALUi – Arithmetic Immediate



- 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)`

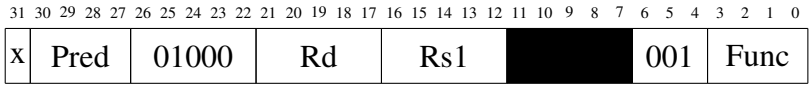
Behavior

`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.

D.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

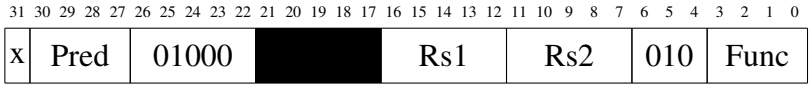
Behavior

`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.

D.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

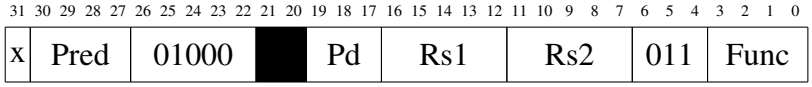
Multiplies may only be issued in the first slot of a bundle.

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

D.4 Compare

Applies to the ALUc format only.

- ALUc – Compare



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]`

Behavior

`IF` –

`DR` Read register operands `Pred`, `Rs1`, and `Rs2`.

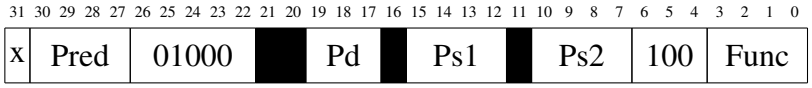
`EX` By-pass values for `Rs1` and `Rs2`, if needed.
Perform comparison (see above).
Derive write-enable signal for destination register `Pd` from predicate `Pred`.
Write to destination register `Pd`.

`MW` –

D.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`

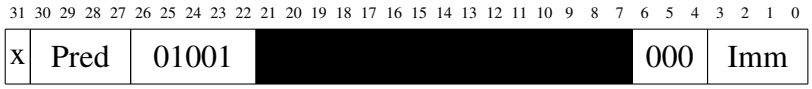
Behavior

`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` –

D.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



Behavior

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

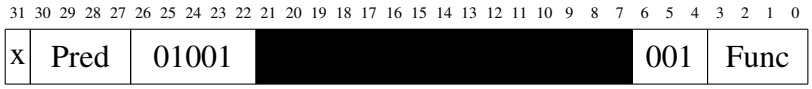
Note

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.

D.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	waitm	Wait for a memory access
0001	—	unused
...	...	...
1111	—	unused

Behavior

`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.

D.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`

Behavior

`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` –

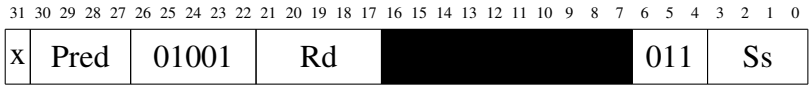
Note

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

D.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`

Behavior

`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.

D.10 Load Typed

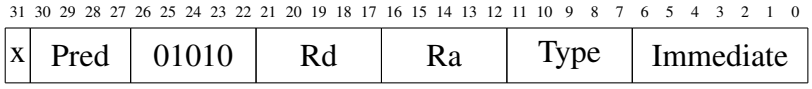
Applies to the LDT format only. Load from a memory or cache. In the table accesses to the stack cache are denoted by `sc`, to the local scratchpad memory by `lm`, to the date cache by `dc`, and to the global shared memory by `gm`. By default all load variants are considered with an implicit `wait` –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

Behavior – 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.

Behavior – 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.

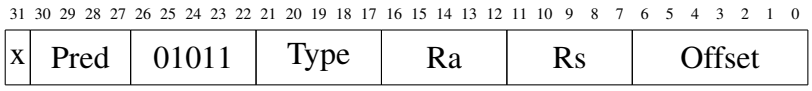
Two successive decoupled loads can be used in the following manner without the use of an additional `wait` instruction:

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

...

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

D.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



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

Behavior

`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

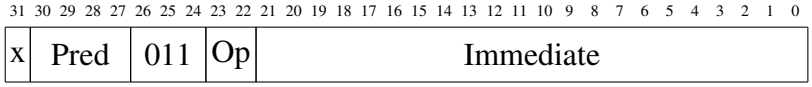
Stores to the stack cache can be issued, also concurrently, on both slots of an instruction bundle. All other stores can only be issued on the first slot.

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.

D.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

Behavior – 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.

Behavior – 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.

Behavior – Free

`IF` –
`DR` Read register operand `Pred` and internal stack-cache registers `head`.
`EX` Check predicate `Pred`.
Account for `head` − `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.

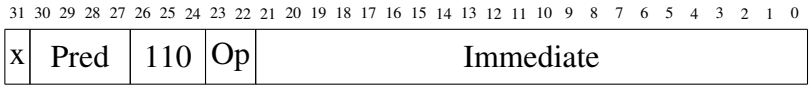
D.13 Call and Branch

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

Behavior – 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.

Behavior – 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.

D.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

Behavior – 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.

Behavior – 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.

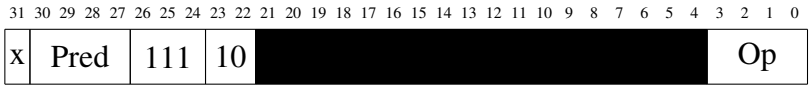
Offsets and addresses are interpreted in word size.

D.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

Behavior

`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.

References

[1] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.

[2] Florian Brandner. Precise simulation of interrupts using a rollback mechanism. In *SCOPES '09: Proceedings of the 12th International Workshop on Software and Compilers for Embedded Systems*, pages 71–80, 2009.

[3] Paul Klint. Interpretation techniques. *Software: Practice and Experience*, 11(9):963 – 973, 1981.

[4] Christopher Mills, Stanley C. Ahalt, and Jim Fowler. Compiled instruction set simulation. *Software: Practice and Experience*, 21(8):877–889, 1991.

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

[6] Joshua J. Yi and David J. Lilja. Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations. *IEEE Transactions on Computers*, 55(3):268–280, 2006.