Patmos Reference Handbook
Technical Report

We shall have authors and acknowledgement for contributors
June 3, 2013

***Abstract*—This TR shall evolve to the documentation of Patmos. In the mean time it is intended to collect design notes and discussions. Especially ISA design notes now.**

I. Introduction

Real-time systems need a time-predictable execution platform so that the worst-case execution time (WCET) can be statically estimated. It has been argued that we have to rethink computer architecture for real-time systems instead of trying to catch up with new processors in the WCET analysis tools [3], [1].

We present the time-predictable processor Patmos as one approach to attack the complexity issue of WCET analysis. Patmos is a static scheduled, dual-issue RISC processor that is optimized for real-time systems.

A. TODO

This report shall converge towards a real manual. At the moment it serves discussion well, but we shall keep this in mind. Here a starting list of TODOs:

- Send an email to all and ask about cleanup of some discussion points
- Convert some discussion text into readable sections and argue why we did what we did
- Get a nice introduction and a good architecture section written

II. The Architecture of Patmos

A. Pipeline

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

B. Register Files

The register files available in Patmos are depicted by Figure 2. In short, Patmos offers:

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

The general-purpose registers `R` are read in the `DEC` stage and written in the `WB` stage. Full forwarding makes them available in the `EX` stage before written into the register file.

The predicate registers are single bits that are set and read in the `EX` stage.

The special registers `S` is just a collection of various 'special' processor registers. These registers might be used by different units/stages in the pipeline and are not physically collected in a 'register file'. The pipeline stage where those registers are read and written by the `mfs` and `mts` are dependent on the type of the special register.

So all-in-all the recoverable process state is: general-purpose registers `R`, the predicates `P`, and a collection of various processor registers mapped to the 'special' register files `S`.

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, it must be guaranteed that the predicate of at least one instruction in the bundle 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 register file should be inverted before it is used. For 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. Setting the reserved bits has no effect.
- `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.
- `s5` can alse be access through the name `ss` and represents the register pointing to the top of the saved stack content in the main memory (i.e., the current stack spill pointer). Updating `s5` does not change `s6` or spill the stack cache.
- `s6` can also be accessed through the name `st` and represents a pointer to the top-most element of the content

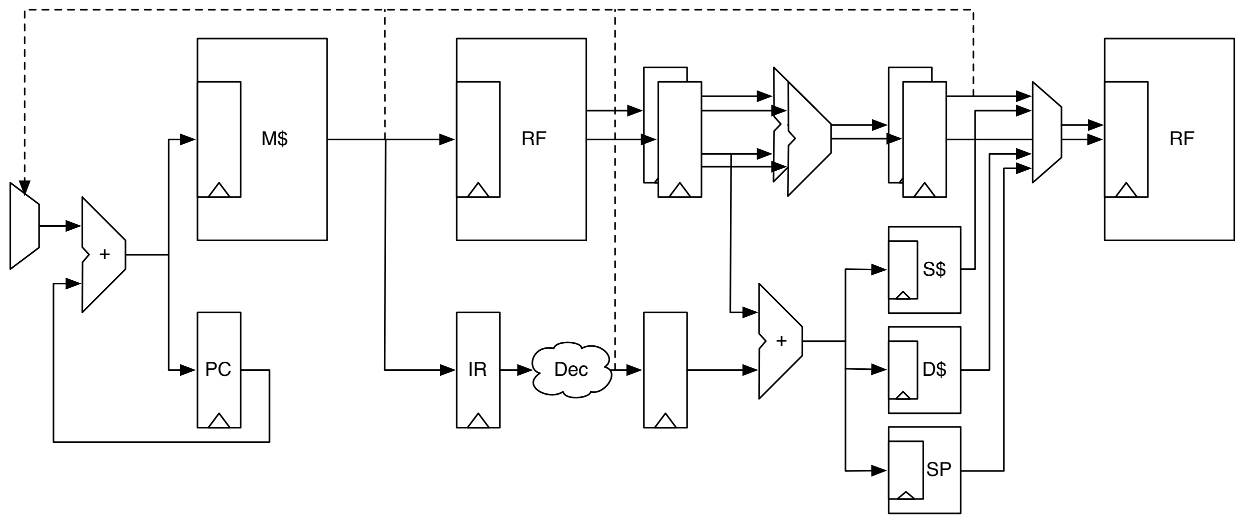


Fig. 1. Pipeline of Patmos with fetch, decode, execute, memory, and write back stages.

of the stack cache. Updating `s6` does not change `s5` or spill the stack cache.

III. 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

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

0	▒

- 64-bit bundle format

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

1	▒

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	▒

IV. 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

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	00	Func	Rd	Rs1	Immediate

- ALUl – Long Immediate

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

1	Pred	11111	Rd	Rs1	■	000	Func

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

Long Immediate

- ALU – Arithmetic

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	▒	Opc	Func

  - ALUr – Register

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	Rd	Rs1	Rs2	000	Func

  - ALUm – Multiply

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	■	Rs1	Rs2	010	Func

  - 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

  - ALUp – Predicate

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	■	Ps1	■	Ps2	100	Func

- SPC – Special

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	01001	▒	Opc	I/R/F

  - SPCw – Wait

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

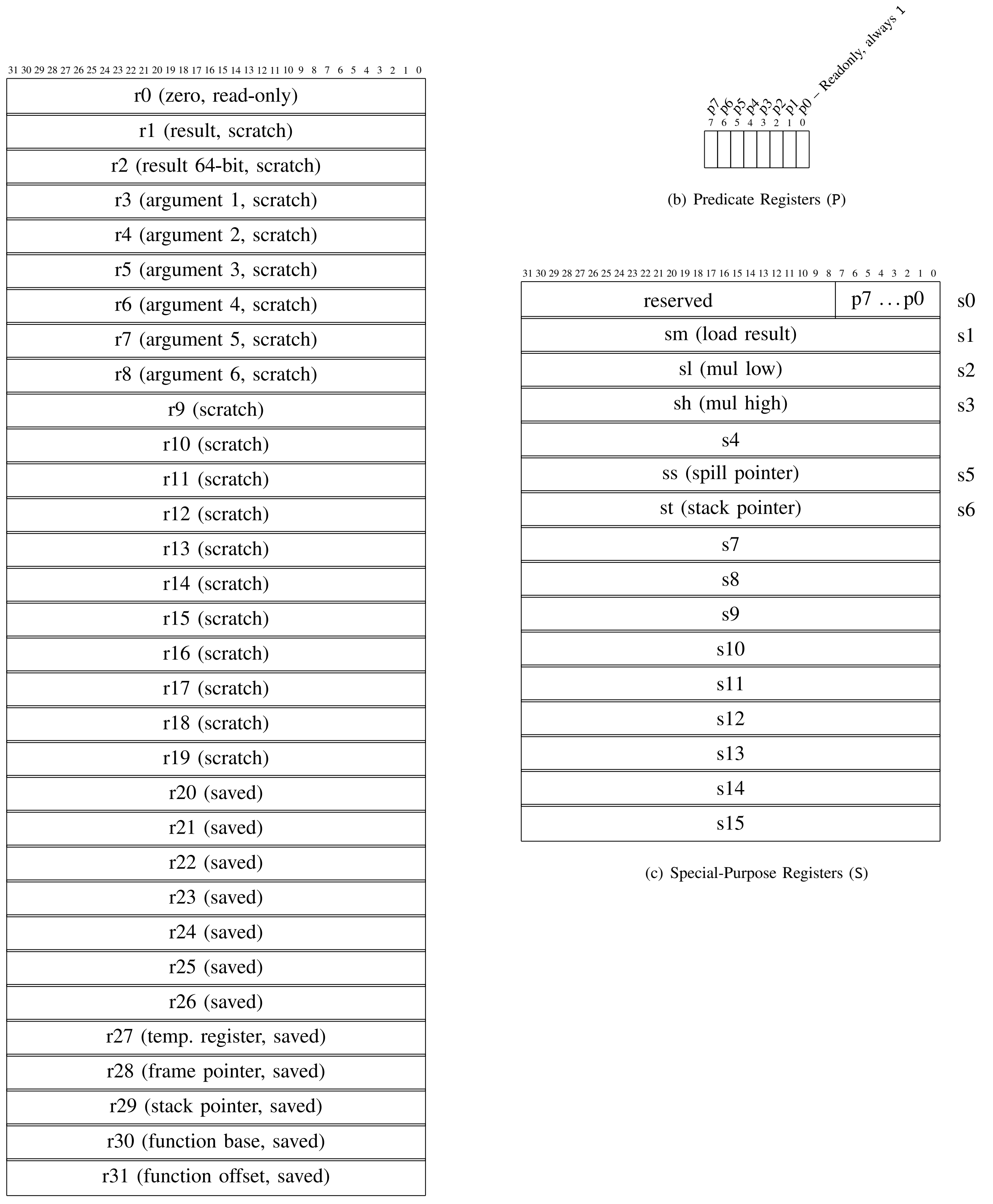
x	Pred	01001	■	001	Func

  - SPCt – Move To Special

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	01001	■	Rs1	■	010	Sd

  - SPCf – Move From Special

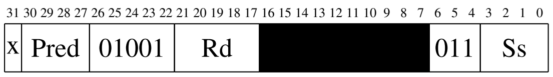


(b) Predicate Registers (P)

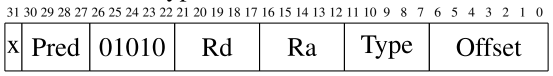
(c) Special-Purpose Registers (S)

(a) General-Purpose Registers (R)

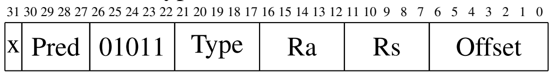
Fig. 2. General-purpose register files, predicate registers, and special-purpose registers of Patmos.



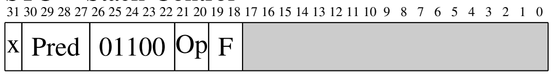
- LDT – Load Typed



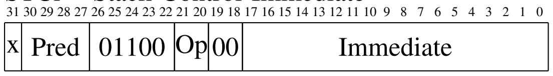
- STT – Store Typed



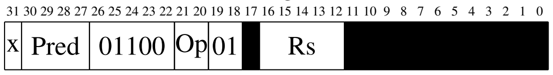
- STC – Stack Control



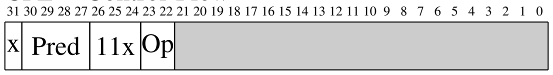
  - STCi – Stack Control Immediate



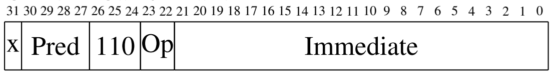
  - STCr – Stack Control Register



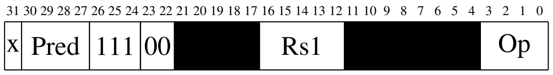
- CFL – Control Flow



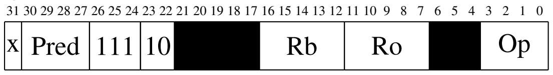
  - CFLb – Call / Branch



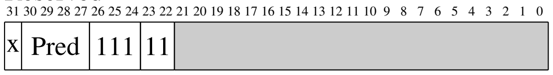
  - CFLi – Call / Branch Indirect



  - CFLr – Return



  - Reserved



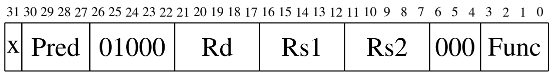
V. Instruction Opcodes

This section defines the instruction set architecture, the instruction opcodes, and the behavior of the respective instructions of Patmos. This section should be less used for discussions and should slowly converge to a final definition of the instruction set.

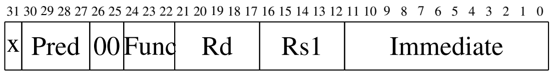
A. Binary Arithmetic

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

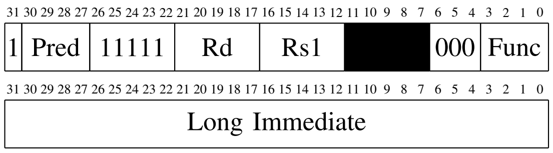
- ALUr – Register



- ALUi – Arithmetic Immediate



- ALUl – Long Immediate



Func	Name	Semantics
0000	add	`Rd = Rs1 + Op2`
0001	sub	`Rd = Rs1 - Op2`
0010	xor	`Rd = Rs1 ^ Op2`
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	—	unused
1001	—	unused
1010	—	unused
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`
- `clr Rd` ... `add Rd = r0 + 0`
- `neg Rd = -Rs` ... `sub Rd = 0 - Rs`
- `not Rd = ~Rs` ... `nor Rd = ~(Rs | R0)`
- `li Rd = Immediate` ... `add Rd = r0 + Immediate`
- `li Rd = Immediate` ... `sub Rd = r0 - Immediate`
- `nop` ... `sub r0 = r0 - 0`

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.

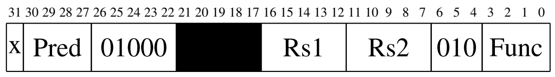
Note Function codes `1000` and `1001` have been used for rotate left and rotate right, they have been removed.

```
lwc r0 = [addr] # speculative load to cache
nop             # (sub r0 = r0 - 0): r0 != 0
li  r1 = imm    # (add r1 = r0 + imm ):
                #   r0 != 0 => r1 != imm
```

B. Multiply

Applies to the ALUm format only. Multiplications are executed in parallel with the regular pipeline and finish within a fixed number of cycles (3-5 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

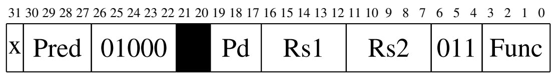
Behavior

`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).
$\mathtt{EX}_1$ Perform multiplication.
...
$\mathtt{EX}_n$ Perform multiplication.
Write to destination registers `sl` and `sh`.

C. 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

- `isodd Pd = Rs1` ... `btest Pd = Rs1[r0]`
- `mov Pd = Rs` ... `cmpneq Pd = Rs != r0`

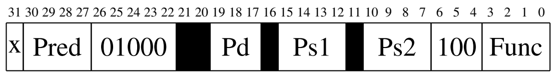
Behavior

`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. 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	por	`Pd = Ps1 | Ps2`
0111	pand	`Pd = Ps1 & Ps2`
1000	—	unused
1001	—	unused
1010	pxor	`Pd = Ps1 ^ Ps2`
1011	—	unused
1100	—	unused
...	...	...
1111	—	unused

Pseudo Instructions

- `pmov Pd = Ps` ... `por Pd = Ps | Ps`
- `pnot Pd = ~Ps` ... `pxor Pd = (Ps ^ p0)`
- `pset Pd = 1` ... `por Pd = p0 | p0`
- `pclr Pd = 0` ... `pxor 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 –

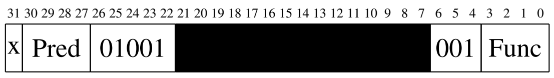
Note

All predicate combine instruction mnemonics (including pseudo instructions) are prefixed with `p`, all other instructions involving predicates are not prefixed (e.g., moving from register to predicate).

E. Wait

Applies to the SPCw format only. Wait for a multiplication or 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

Behavior

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

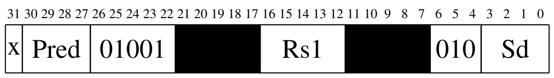
Note

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

F. 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 –

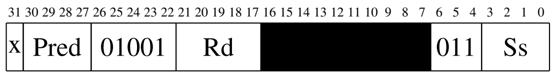
Note

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

G. 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.

H. 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.

Regular loads 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.

The displacement value (Imm) value is interpreted unsigned.

- LDT – Load 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	01010	Rd	Ra	Type	Immediate

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

All loads can only be issued on the first slot. An optimization to allow loads in both slots of an instruction bundle for the stack cache is open for evaluation.

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
```

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.

The displacement value (Imm) value is interpreted unsigned. Stores can only be issued on the first slot.

- 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$_{[15:0]}$
001 | 01	shl	lm[Ra+(Imm << 1)]$_{16}$ = Rs$_{[15:0]}$
001 | 10	shc	dc[Ra+(Imm << 1)]$_{16}$ = Rs$_{[15:0]}$
001 | 11	shm	gm[Ra+(Imm << 1)]$_{16}$ = Rs$_{[15:0]}$
010 | 00	sbs	sc[Ra+Imm]$_{8}$ = Rs$_{[7:0]}$
010 | 01	sbl	lm[Ra+Imm]$_{8}$ = Rs$_{[7:0]}$
010 | 10	sbc	dc[Ra+Imm]$_{8}$ = Rs$_{[7:0]}$
010 | 11	sbm	gm[Ra+Imm]$_{8}$ = Rs$_{[7:0]}$
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.

Store operations do not stall. 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.

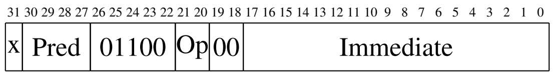
J. Stack Control

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). `sspill` writes the tail of the stack cache to main memory and updates the spill pointer.

All immediate stack control operations are carried out assuming word size, i.e., the immediate operand is multiplied by four. All register operands and stack pointer addresses in special registers are in units of bytes.

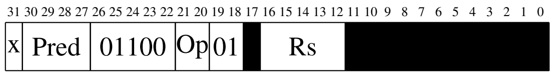
A more detailed description of the stack cache is given in Section VII.

- STCi – Stack Control Immediate



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	sspill	Spill tail of the stack cache to memory

- STCr – Stack Control Register



Op	Name	Semantics
00	—	unused
01	sens	Ensure stack space (with refill)
10	—	unused
11	sspill	Spill tail of the stack cache to memory

Behavior – Reserve

IF –
DR Read register operand `Pred`, `ss`, `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 `ss` and stall.

Behavior – Ensure

IF –
DR Read register operand `Pred`, `ss`, `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 `ss` and stall.

Behavior – 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 `ss` and `st`.
Update stack-cache register `head`.
MW –

Behavior – Spill

IF –
DR Read register operand `Pred`, `ss`, `st` and internal stack-cache registers `head`.
EX Check predicate `Pred`.
Update `ss` and `st`.
Update stack-cache register `tail`.
MW Spill to global memory using `ss` and stall.

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.

K. Control Instructions

Applies to CFLb and CFLi format only. Transfer control to another function or perform function-local branches. `br` performs a function-local, relative branch within the method cache. `call` performs a function call, storing the program counter (or function offset) of the instruction to be fetched after returning in `r31`. The function base of the caller is not stored implicitly (see Section XII-C). `brcf` behaves like `call`, except that it does not write return information to `r31`.

`call` and `brfc` may cause a cache miss and a subsequent cache refill to load the target code; they expect the size of the code block fetched to the cache in number of bytes at *<base>-4*. `br` is assumed to be a cache hit.

Immediate call and branch instructions interpret the operand as *unsigned* for function calls, and as *signed* for PC-relative branches (`br`, `brcf`). The target address of PC-relative branches is computed relative to the address of the branch instruction. All immediate values are interpreted in *word size*.

Indirect call and branch instructions interpret the operand as *unsigned* absolute addresses in *byte size*.

The link/return information provided by `call` in `r31` should only be passed to `ret`. The unit and addressing mode (absolute or function relative) of the returned value is implementation dependent (see description of `ret`).

The following table gives an overview of the addressing modes of the available call and branch instructions.

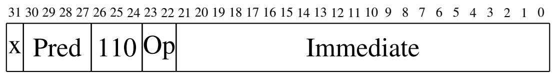
Instruction	Immediate	Indirect	Cache fill	Link
call	absolute	absolute	yes	yes
br	PC relative	absolute	no	no
brcf	PC relative	absolute	yes	no

`br` instructions are executed in the `EX` stage, while `brcf`, `call` and `ret` instructions are executed in the `MW` stage. The instructions fetched in the meantime are *not* aborted. This corresponds to to a branch delay of 2 instructions for `br` and 3 instructions for `call`, `brcf` and `ret`, which has to be respected by the compiler or assembly programmer. If no other instructions are available, two single-cycle NOPs can be used to stall the processor explicitly.

Size of the delay slots: The `call` instructions must have exactly one single-word instruction in each delay slot and it must not be bundled with another instruction. All other control flow instructions (`br`, `brcf`, `ret`) can have any valid bundle in their delay slots and can be bundled.

More details on the organization of the method cache is given in Section VIII.

- CFLb – Call / Branch



Op	Name	Semantics
00	call	function call (absolute, with cache fill)
01	br	local branch (PC relative, always hit)
10	brcf	local branch (PC relative, with cache fill)
11	—	unused

Behavior – call, brcf, and system call

`IF` –
`DR` Read register operand `Pred`.
`EX` Check predicate `Pred`.
Store link information into R31. Method base is not stored to a visible register (this must be done by the caller, by convention using R30).
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. Update program counter.
`MW` –

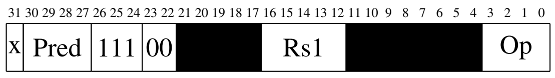
Implementation Note

The method cache keeps track of the base address of the current function, i.e., the target/base address of the last `call`, `brcf` or `ret` instruction. `call` calculates the return offset as $\mathtt{nextPC}_{\mathtt{EX}} - \mathtt{base}_{\mathtt{MC}}$. However, the application code must not rely on this.

Note

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

- CFLi – Call / Branch Indirect



Op	Name	Semantics
0000	call	function call (indirect, with cache fill)
0001	br	local branch (indirect, always hit)
0010	brcf	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 link information into R31. Method base is not stored to a visible register (this must be done by the caller, by convention using R30).
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` and `Rs1`.
`EX` By-pass value for `Rs1`.
Check predicate `Pred`.
Assert on method cache.
Compute new, cache-relative program counter value.
Update program counter.
`MW` –

Note

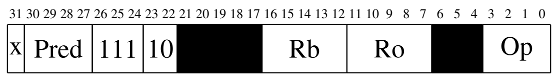
All branch/call instructions can only be issued on the first position within a bundle.

See also CFLi format notes.

L. Return

Applies to CFLr format only. Transfer control to the function specified by function base and offset. `ret` may cause a cache miss and a subsequent cache refill to load the target code.

- CFLr – Return



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

Behavior

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

The return function base address `Rb` is an absolute address in bytes. The return function base will typically be provided by the caller in `r30` (see Section XII-C). The return function offset `Ro` is provided by the `call` instruction in `r31`. The unit and the addressing mode of the function offset is hardware implementation dependent.

VI. Dual Issue Instructions

Not all instructions can be executed in both pipelines. In general, the first pipeline implements all instructions, the second pipeline only a subset. All memory operations are only executed in the first pipeline.

What other instructions can be executed in both pipelines is still open for discussion and evaluation with benchmarks. A minimal approach, as first step for the hardware implementation, is to have only ALU instructions available in the second pipelines (excluding predicate manipulation instructions).

VII. 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 `head` − `tail` > *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 `ss` is accordingly incremented.

The stack cache is organized in blocks of fixed size, e.g. 32 bytes. All spill and fill operations are performed on the block level, while reserve, free and ensure operations are in words.

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 register `ss`. `st` contains the address the top of the stack cache would get if the stack cache would be fully spilled to memory.

The organization 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 realized 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 XII).

In order to allow two parallel load/store operations the memory of the stack cache might be operated with a doubled clock frequency.

VIII. Method Cache

An overview of alternative design options with regard to the method cache can be found in Section **??**. It is uncertain which of those options is best, however, two candidates appear very promising and should be evaluated: fetch on call with FIFO replacement and fetch on call with LRU replacement. Compiler managed prefetching can still be added at a later stage.

A. Common Features

The cache is organized in blocks of a fixed size, e.g., 32 bytes.

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 the entire function at once. Code transfers between the respective junks of the original function can be performed using the `brcf` instruction.

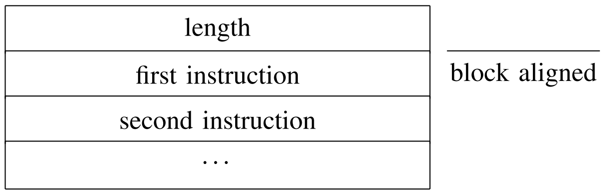


Fig. 3. Layout of code sequences intended to be cached in the method cache.

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 in units of bytes right in front of the first instruction of a code sequence that is intended for caching. Figure 3 illustrates this convention.

The organization 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.

B. 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.

C. LRU replacement

The LRU cache configuration is more complex. Code is *not* kept in contiguous blocks and might be scattered according to the LRU time stamps of the blocks in the cache. Every block is thus tagged with the address of the block in global memory and an LRU time stamp. The address part of the tag is need to rediscover the block during instruction fetch. The time stamp is required to implement the LRU policy.

In addition, the length of every code sequence currently in the cache has to be stored.

Time Stamps: On every access to the method cache, i.e., for every call or non-cache-relative branch, the LRU time stamps of the blocks (possibly all) in the cache has to be done. It is important to note that *all* blocks of a code sequence share the *same* LRU time stamp at all times.

Instruction Fetch: In order to fetch an instruction from the method cache the address of the instruction is compared with the address tag of every block in the cache (excluding some of the least significant bits depending on the cache's block size). If a matching tag is found, i.e., a cache hit, the respective word in the block is fetched.

If no block with a matched tag exists, a cache miss occurs and the target block has to be transferred from the global memory into the cache.

Purging Blocks: When a code sequence is to be loaded into the cache, it has to be ensured that enough space is available to hold all the blocks of that code sequence by purging some blocks currently in use. Note, again, only entire code sequences are purged from the cache, i.e., all its blocks at once. Code sequences are repeatedly purged until enough space becomes available in the cache.

Cache Fill: Once enough space is available, the code sequence is transferred block-wise from global memory to the cache, the tag and LRU time stamp are set accordingly for every fetched block.

IX. Instruction Cache

This section will cover a configuration of Patmos with a standard instruction cache design, i.e., without a method cache.

We would like to see a Patmos implementation with a 2-way set-associative instruction cache governed under the least-recently used (LRU) replacement policy. This allows for a "real" comparison between a FIFO method cache and a standard instruction cache on the same architecture.

X. Data Cache

This section will cover a configuration of Patmos with a standard data cache design, i.e., without a stack cache (and shadow stack).

We would like to see a Patmos implementation with a 2-way or 4-way set-associative data cache governed under the least-recently used (LRU) replacement policy. The data cache should be write-through.

Again this allows for a "real" comparison between the stack cache and a standard data cache on the same architecture.

It might also be interesting to explore the object cache idea [4], [2] where objects (heap allocated structures) are tracked via a fully associative cache. Furthermore, for arrays with low temporal localty a small set of prefetch buffers may benefit from spacial locality.

XI. IO Devices

Each processor contains a minimum set of standard IO devices, such as: processor ID, cycle counter, timer, and interrupt controller. For a minimum communication with the outside world a processor shall be attached to a serial port (UART). The UART represents `stdout`.

On a multicore system only one processor can be directly connected to the UART. However, for debugging it would be convenient to attach several (or all) cores to the UART. Sharing the UART can be achieved by an arbitration device that has *n* input ports and a simple protocol that precedes UART data

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

TABLE I
ADDRESS MAPPING

Bit	Status		Control	
0	TRE	TX Transmit ready	–	–
1	DAV	RX Data available	–	–
2	PAE	RX Parity error (or EOF)	–	–
3	–	–	TFL	TX Flush

TABLE II
MEANING OF BITS IN UART CONTROL WORD

by a marker from which core the data is coming. The marker byte may be precede each data byte or it may be distinguished by setting Bit 8. This mechanism can also be used to represent several UARTs per core (e.g., stdout, stderr, user for SLIP,...).

On the PC side a small program is needed to dispatch/de-multiplex the different data streams.

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

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

In the initial prototype of Patmos we have 3 I/O devices: a system device that contains cycle and microsecond counters, a UART for basic communication, and LEDs on the FPGA board. One counter ticks with the clock frequency and the second counter ticks with 1 MHz for clock frequency independent time measurements. The counters are 64-bit values and readout of the lower 32 bits also latches the upper 32 bits. Table III shows the I/O devices and the registers.

The UART address for `pasim` is defined in

Address	I/O Device	read	write
`0xf0000000`	ID	processor ID	–
`0xf0000010`	counter	clock cycles (high word)	–
`0xf0000014`	counter	clock cycles (low word)	–
`0xf0000018`	counter	time in μs (high word)	–
`0xf000001c`	counter	time in μs (low word)	–
`0xf0000100`	UART	status	control
`0xf0000104`	UART	receive buffer	transmit buffer
`0xf0000200`	LED	–	output register
`0xf0000300`	SDRAM	not used	

TABLE III
I/O DEVICES AND REGISTERS

`patmos/simulator/include/uart.h`.

For the compiler/library the constant is in `llvm/tools/clang/lib/Driver/Tools.cpp`.

XII. APPLICATION BINARY INTERFACE

A. Data Representation

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

B. 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.
- `r27` is used as temp register.
- `r28` is defined as the frame pointer and `r29` is defined as the stack pointer for the *shadow* stack in global memory. The use of a frame pointer is optional, the register can freely be used otherwise. `r29` is guaranteed to always hold the current stack pointer and is not used otherwise by the compiler.
- `r30` and `r31` are defined as the return function base and the return function offset. Usually, they are passed as operands to the `ret` instruction.
- `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 stack cache control registers `ss` and `st` are callee-saved *saved* register.
- `s0`, representing the predicate registers, is a callee-saved *saved* register.
- All other special registers are caller-saved scratch registers and should not be used across function calls.

C. Function Calls

Function calls have to be executed using the `call` instruction that automatically prefetches the target function to the method cache and stores the return information to the general-purpose register `r31`. At a function call, the callers base address has to be in `r30`. The callee is responsible to store/restore the callers function base and pass it as first operand to the return instruction. The `call` and `brcf` instructions neither use nor modify `r30`, the return function base is only used by `ret`.

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 return function base `r30` and the return offset `r31` needs 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. Note that in contrast to `br` and `brcf` the return offset refers to the next instruction after the *delay slot* of the corresponding `call` and can be implementation dependent (cf. the description of the `call` and `ret` instructions).

D. Sub-Functions

A function can be split into several sub-functions. The program is only allowed to use `br` to jump within the same sub-function. To enter a different sub-function, `brcf` must be used. It can only be used to jump to the first instruction of a sub-function.

In contrast to `call`, `brcf` does not provide link information. Executing `ret` in a sub-function will therefore return to the last `call`, not to the last `brcf`. Function offsets however are relative to the *sub-function* base, not to the surrounding function. The function base register `r30` must therefore be set to the base address of the current *sub-function* for calls inside sub-functions.

A sub-function must be aligned and must be prefixed with a word containing the size of the sub-function, like for a regular function. If a function is split into sub-functions, the first sub-function must also be prefixed with the size of the first sub-function, not with the size of the whole function.

There are no calling conventions for jumps between sub-functions, for the compiler this behaves just like a regular jump, except that the base register `r30` must be updated if the sub-function contains calls.

E. 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.

F. Interrupts and Context Switching

Interrupt handlers may use the shadow stack pointer `r29` to spill registers to the shadow stack. Interrupts must ensure that all special registers that might be in use when the interrupt occurs are saved and restored.

Here is a simple example of storing and restoring the context for context switching.

```
sub $r29 = $r29, 40
sws [$r29 + 0] = $r31
sws [$r29 + 1] = $r30
sws [$r29 + 2] = $r22  // free some registers
sws [$r29 + 3] = $r23
mfs $r22 = $s2  // by now any mul should be finished
mfs $r23 = $s3
sws [$r29 + 4] = $r22
sws [$r29 + 5] = $r23
mfs $r22 = $r5  // read out cache pointers, spill
mfs $r23 = $s6
sub $r22 = $r23, $r22
sspill $r22   // spill the memory, s5 == s6 now
sws [$r29 + 6] = $r22  // store the stack pointer
sws [$r29 + 7] = $r23  // store stack size
...
// TODO store return base and offset

// restore
lws $r23 = [$r29 + 7]
lws $r22 = [$r29 + 6]
mts $s5 = $r22   // restore the stack
mts $s6 = $r22
sens $r23
lws $r23 = [$r29 + 5]
lws $r22 = [$r29 + 4]
mts $s2 = $r22
mts $s3 = $r23
....
```

References

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

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

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

[4] M. Schoeberl. A time-predictable object cache. In *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, pages 99–105, Newport Beach, CA, USA, March 2011. IEEE Computer Society.