Patmos: A Time-Predictable Dual-Issue Microprocessor

Technical Report

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 [2], [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.

II. The Architecture of Patmos

A. Pipeline

Figure 1 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 $\texttt{EX}_1$, ..., $\texttt{EX}_n$.

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

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

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

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

The predicate registers are usually encoded as 4-bit operands, where the most significant bit indicates that the value read from the 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:

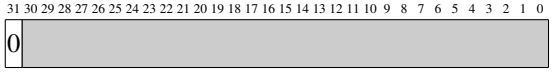
- The lower 8 bits of `s0` can be used to save/restore *all* predicate registers at once. The other bits of that register are currently reserved, but not used.
- `s1` can also be accessed through the name `sm` and represents the result of a decoupled load operation. The value is already sign-/zero-extended according to the load instruction. This register is read-only.
- `s2` and `s3` can also be accessed through the names `sl` and `sh` and represent the lower and upper 32-bits a multiplication. These registers are read-only.
- `s5` Represents the register pointing to the top of the saved stack content in the main memory.
- `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. This register is read-only.

III. Bundle Formats

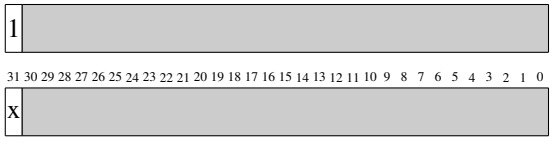
All Patmos instructions are 32 bits wide and are structured according to one of the instruction formats defined in the following section. Up to two instructions can be combined to form an instruction bundle; Patmos bundles are thus either 32 or 64 bits wide. The bundles sizes are recognized by the value 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



- 64-bit bundle format



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.

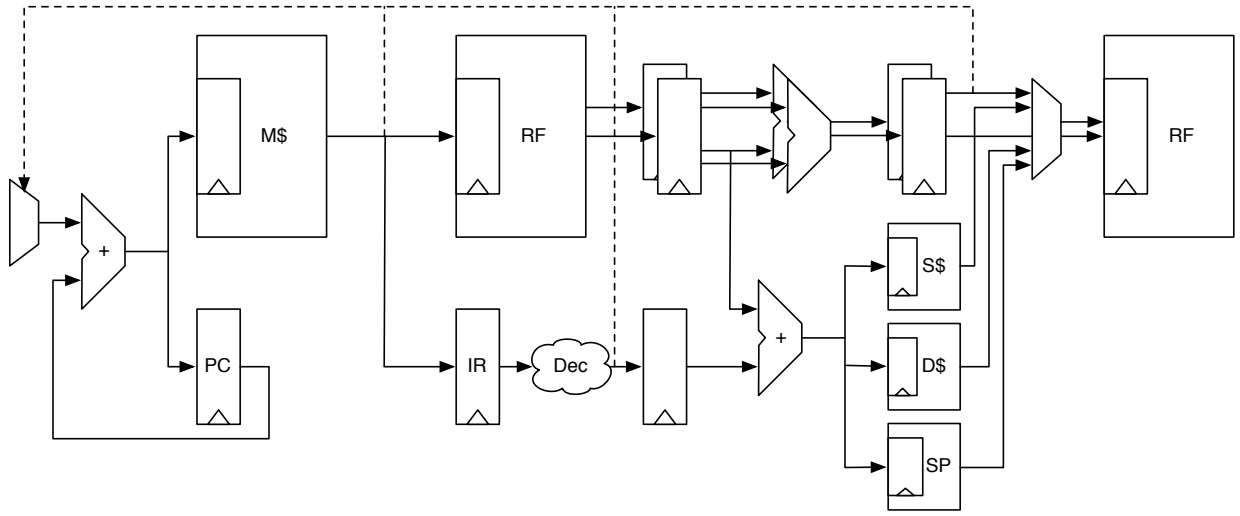
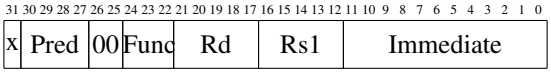
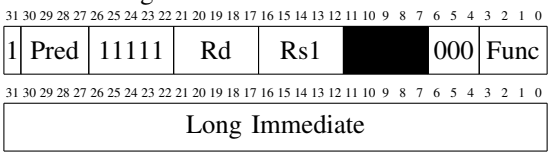


Fig. 1. Pipeline of Patmos with fetch, decode, execute, memory, and write back stages.

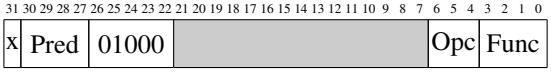
- ALUi – Arithmetic Immediate



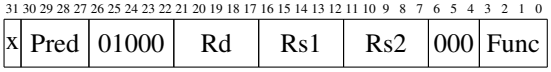
- ALUl – Long Immediate



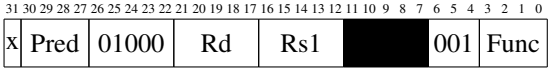
- ALU – Arithmetic



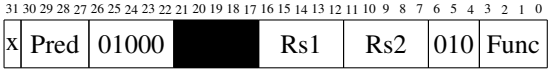
  - ALUr – Register



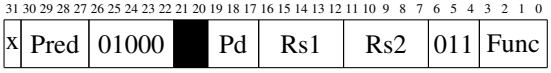
  - ALUu – Unary



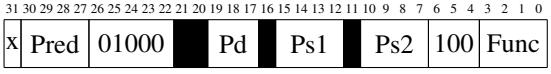
  - ALUm – Multiply



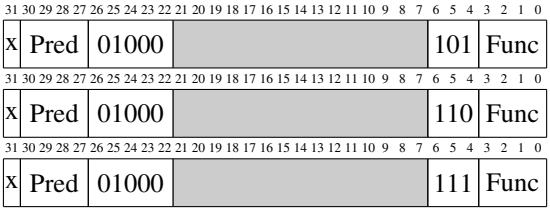
  - ALUc – Compare



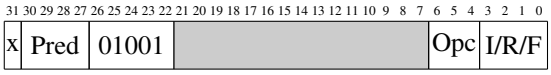
  - ALUp – Predicate



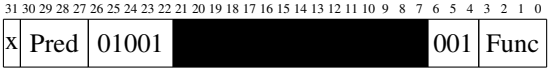
  - Unused



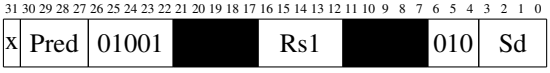
- SPC – Special



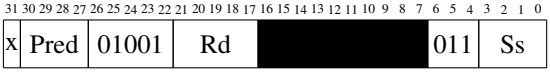
  - SPCw – Wait



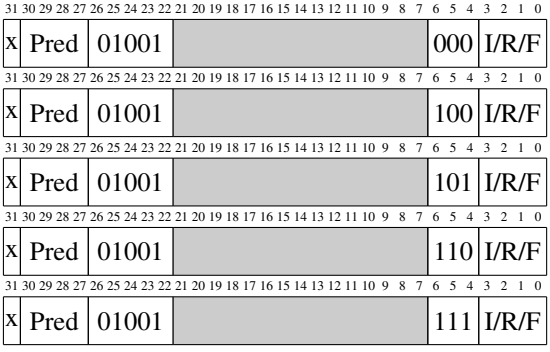
  - SPCt – Move To Special



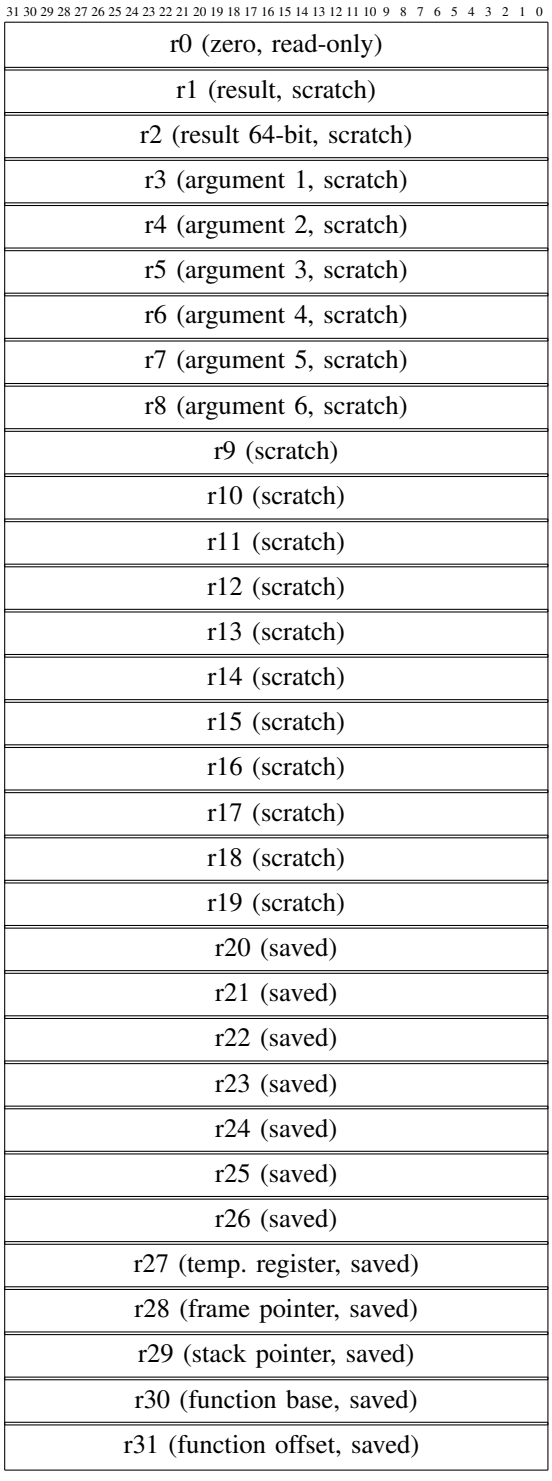
  - SPCf – Move From Special



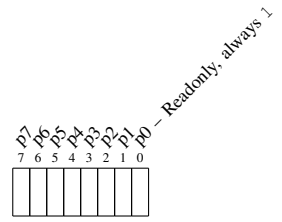
  - Unused



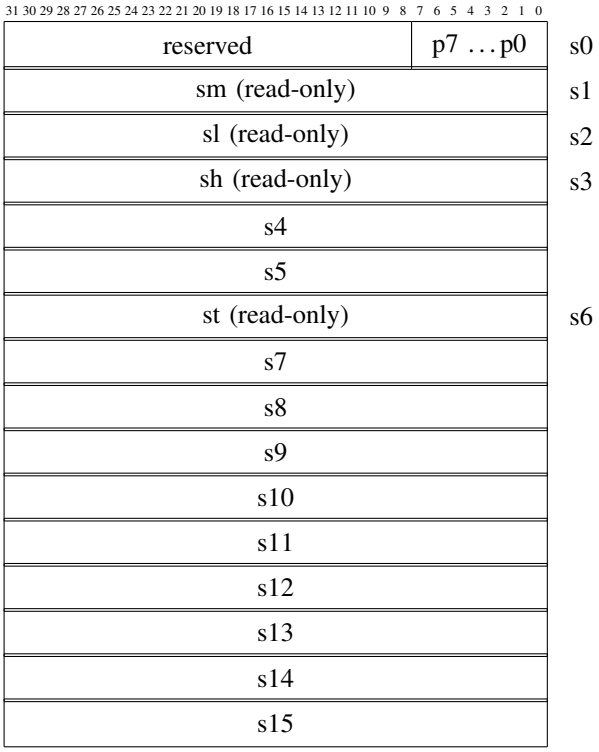
- LDT – Load Typed



(a) General-Purpose Registers (R)

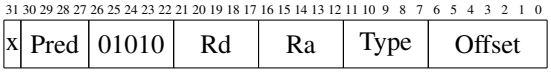


(b) Predicate Registers (P)

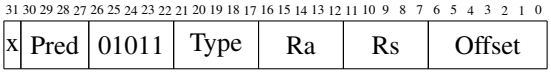


(c) Special-Purpose Registers (S)

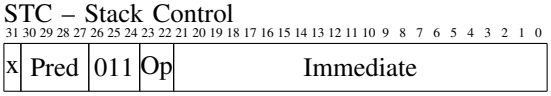
Fig. 2. General-purpose register files, predicate registers, and special-purpose registers of Patmos.



- STT – Store Typed

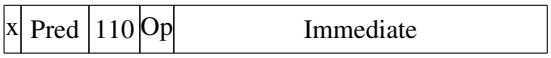


- STC – Stack Control

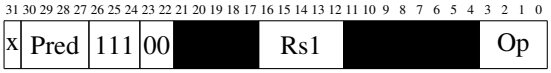


- CLF – Control Flow

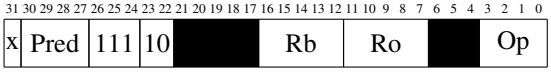
  - CLFb – Call / Branch



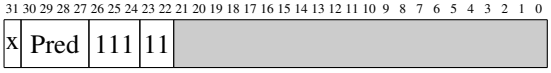
  - CLFi – Call / Branch Indirect



  - CLFr – Return



  - Reserved



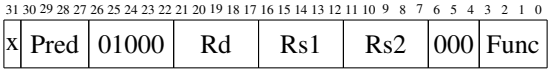
V. Instruction Opcodes

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.

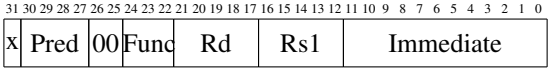
A. Binary Arithmetic

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.

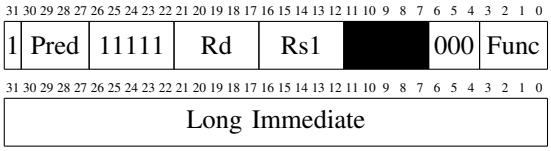
- ALUr – Register



- ALUi – Arithmetic Immediate



- ALUl – Long Immediate



Func	Name	Semantics
0000	add	`Rd = Rs1 + Op2`
0001	sub	`Rd = Rs1 - Op2`
0010	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` ... `sub Rd = 0 - Rs`
- `not Rd = ~Rs` ... `nor Rd = ~(Rs | R0)`
- `zext8 Rd = (uint8_t)Rs` ... `and Rd = Rs & 0xFF`
- `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.

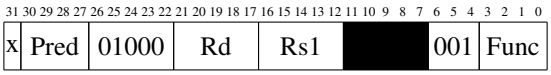
`MW` Write to destination register `Rd`.
Supply result value for by-passing to `EX` stage.

```
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. 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

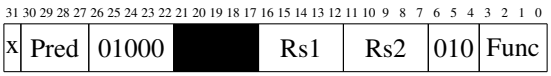
Behavior

`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.

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

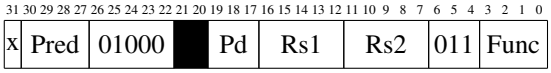
Note

Multiplies are pipelined, it is thus possible to issue one multiplication on every cycles.

D. 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`

Pseudo Instructions (not supported by LLVM assembler)

- `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`

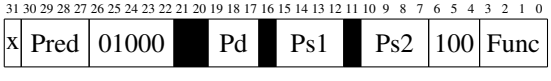
Behavior

`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` –

E. 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` –

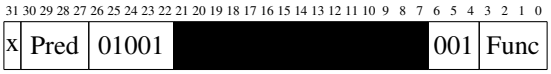
F. NOP

Multicycle NOP has been removed from the ISA and replaced by a single cycle pseudo-instruction NOP (see ALU section).

G. 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` –

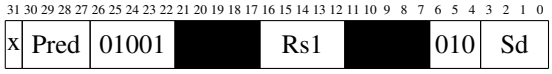
Note

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

H. 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` –

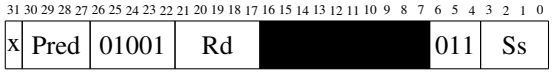
Note

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

I. 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.

J. Load Typed

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

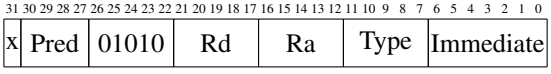
In addition, *decoupled* loads are provided that do *not wait* for the memory access to be completed. The result is then loaded into the special register `sm`. A dedicated `wait.mem` instruction can be used to stall the pipeline explicitly until the load is completed.

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

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

The displacement value (Imm) value is interpreted signed.

- 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:

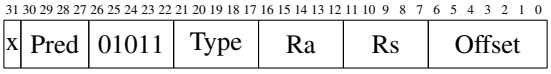
```
dlwc $sm = [$r1 + 5] ;;
...
dlwc $sm = [$r2 + 7] ; mfs $r2 = $sm
```

K. 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 signed.

- 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.

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.

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.

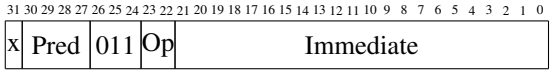
Two parallel stores within the same bundle writing to the same memory such that the accessed memory cells overlap are not permitted. The content of the respective memory cells is undefined in this case.

L. 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.

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

- 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.

M. Call and Branch

Applies to CLFb and CLFi format only. Transfer control to another function or perform function-local branches. br performs a function-local 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 IX-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).

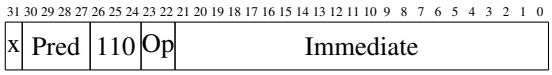
The following table gives an overview of the addressing modes of the available call and branch instructions.

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

Branch and call instructions are effectively executed in the EX stage. The instructions fetched in the meantime are *not* aborted. This corresponds to to a branch delay of 2 instructions that 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.

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

- CLFb – 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.
If needed, fill method cache and stall.
Update program counter.
MW –

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 –

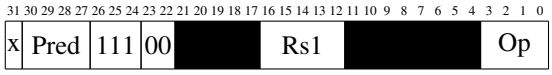
Implementation Note

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 $\texttt{nextPC}_{\texttt{EX}} - \texttt{base}_{\texttt{MC}}$. 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.

- CLFi – 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.
If needed, fill method cache and stall.
Update program counter.
MW –

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

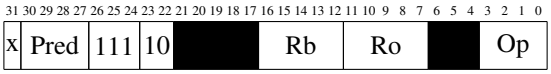
All branch/call instructions can only be issued on the first position within a bundle.

See also CLFi format notes.

N. Return

Applies to CLFr 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.

- CLFr – 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. If needed, fill method cache and stall.
Update program counter.
MW –

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 IX-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 st 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 registers st.

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.

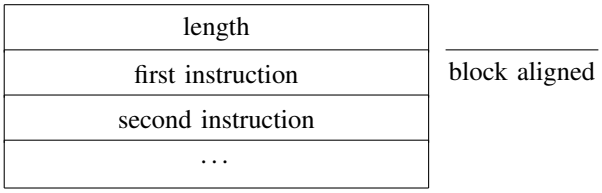


Fig. 3. Layout of code sequences intended to be cached in the method cache.

- 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 IX).

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.

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.

IX. 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` and `r29` are defined as the frame pointer and stack pointer for the *shadow* stack in global memory. The use of a frame pointer is optional, the register can freely be used otherwise.
- `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 top-of-stack of the stack cache `st` is a callee-saved *saved* register.
- The lower 8 bits of `s0`, representing the predicate registers, are are callee-saved *saved* registers. The other bits of the register are reserved and should not be modified.
- All other special registers are scratch registers and should not be used across function calls.

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. Instruction Usage Conventions

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

X. Assembly format

A VLIW instruction consists of one or two operations that are issued in the first or both pipelines. Each operation is predicated, the predicate register is specified before the operation in parentheses `()`. If the predicate register is prefixed by a `!`, its negation is considered. If omitted, it defaults to `(p0)`, i.e. always true.

A double semi-colon `;;` or a newline denotes the end of an instruction. If an instruction contains two operations, the operations must be separated by a single semi-colon or a single-semicolon followed by either a newline or a comment. Note that since newline is an instruction separator, the operation separator must always appear on the same line as the operation for the first slot. Labels that are prefixed by `.L` are local labels.

All register names must be prefixed by `$`. We use destination before source in the instructions, between destination and source a `=` character must be used instead of a comma. Immediate values are not prefixed for decimal notation, the usual 0 and 0x formats are accepted for octal and hexadecimal immediates. Comments start with the hash symbol `#` and are considered to the end of the line. For memory operations, the syntax is `[$register + offset]`. Register or offset can be ommited, in that case the zero register `r0` or an offset of 0 is used.

Example:

```
      # add 42 to contents of r2
      # and store result in r1 (first slot)
      add   $r1 = $r2, $42;
      # if r3 equals 50, set p1 to true
      cmpeq $p1, $r3, 50
      # if p1 is true, jump to label_1
($p1) br label_1 ;; nop 3   # then wait 3 cycles
      # Load the address of a symbol into r2
      li $r2 = .L.str2 ;;
      # perform a memory store and a pred op
      swc [$r31 + 2] = $r3 ; or $p1 = !$p2, $p3
      ...
label_1:
      ...
```

A. Inline Assembly

Inline assembly syntax is similar to GCC inline assembly. It uses `%0`, `%1`, ... as placeholders for operands. Accepted register constraints are: `r` or `R` for any general purpose register, or {`<registername>`} to use a specific register.

Example:

```
int i, j, k;
asm("mov  $r31 = %1  # copy i into r31\n\t"
    "add  %0 = $r5, %2"
    : "=r" (j)
    : "r" (i), "{r10}" (k));
```

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