Patmos Reference Handbook

Martin Schoeberl, Florian Brandner, Stefan Hepp,
Wolfgang Puffitsch, Daniel Prokesch

October 7, 2013

Preface

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.

Acknowledgment

We would like to thank Tommy Thorn for the always intense and enjoyable discussions of the Patmos ISA and processor design in general. Jack Whitham offered his experience with RISC ISA design and trade-offs. Gernot Gebhard and Christoph Cullmann gave valuable feedback on the ISA relative to WCET analysis. Sahar Abbaspourseyedi is working on the stack cache and verifies the ideas and concepts presented here. We thank Rasmus Bo Sørensen for fixing some documentation errors.

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

Contents

List of Figures

List of Tables

1 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 [8, 3].

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.

1.1 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

1.2 Pending Changes

This is a live and temporary section that lists pending changes of the ISA. This changes shall live in a remote branch (to be specified) and have a date when to be merged into the master branch.

Currently we have following proposals for a change:

- bcopy instruction
- btest and bcopy with an immediate
- Automatic save of method base in a special register for a call
- Branch into subfunction
- Revised branch addressing modes
- Non-delayed branches
- Issue memory operations from any pipeline
- Interrupts

Remote branch: `isa_rev`
Merge date: 15 August 2013 (?)

1.2.1 `bcopy`/`btest`

Semantically, `bcopy` is the "inverse" to `btest`, such that it allows to copy the value of a predicate register to a position in a 32-bit register. This instruction should be used for

1. `mov Rd ← Ps ...  bcopy Rd ← Ps, r0`
   Currently, LLVM generates a two-instruction sequence `clr Rd; (Ps) li Rd ← 1` for for extending a 1 bit register to a 32 bit register.

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

Table 1.1: Revised branch addressing modes

2. Predicate register spilling.

To avoid the requirement for an additional temporary 32-bit register, particularly for register spill/restore, variants of bcopy and btest with a 5-bit immediate operand are beneficial.

1.2.2 Special Registers for Return Information

In order to support exceptions (see Section 8.1), it is necessary that the base address of the current function is available *at any time*. Otherwise, it could not be guaranteed that an interrupt handler is able to return to the correct function. Consequently, the `call` and `brcf` functions must store the base address in some hardware register. This makes the return-base information in register `r30` redundant and it could be replaced by an appropriate special return base register `sb` with minimal effort. One can then go even further and move the functionality of register `r31` to a special register `so` to free up another general-purpose register.

Costs and Benefits

For functions that call other functions, the change saves one instruction to set the base address, and adds two instructions to both the prologue and the epilogue. On the other hand, the change frees up two general-purpose registers, which can avoid spilling and lead to more efficient code.

1.2.3 Branch into Subfunction Instruction

Costs and Benefits

Much of the complexity and restrictions of the function splitter are eliminated (jump tables still have to be handled carefully, though we in theory we could even support having jump table entries anywhere in any region), without resorting to misuse `ret`. It works regardless of whether return information is stored in special registers or GPRs.

To make use of this feature, the function splitter needs a major rewrite though. Note that as long only `r0` is used for the offset, no changes to the compiler or the WCET analysis are required.

1.2.4 Revised Branch Addressing Modes

Rationale

With a method cache, there are two distinct views of the program counter: the program counter to address the cache memory (PC_{cache}), and the address of the respective instruction in memory (PC_{mem}). When loading instructions from memory (`call`, `brcf`), PC_{mem} is relevant, because it specifies from where data should be loaded. However, when execution remains within a function (`br`), only PC_{cache} is taken into account.

Listing 1.1 shows the pseudo-code for a call (using special registers `sb` and `so` for the return information). First, it stores the return information, and remembers the new base address. Then, it retrieves the offset into the cache for the function to be called and if necessary copies the instructions into the cache. Finally, it updates the internal program counter and continues execution from there.

Listing 1.2 shows the pseudo-code for a return. It first retrieves the return base and does the appropriate cache handling. It then adds the cache offset and the return offset and assigns the sum to the internal program counter, from which execution continues.

The pseudo-code for a PC-relative `brcf` is shown in Listing 1.3. It is similar to the call, but does not store any return information. However, before being able to do any cache handling, it has to compute the actual address,

Listing 1.1: Call

```
call(addr) {
  // Store return information
  sb = base;
  so = PC_cache;
  // Remember base address
  base = addr;
  // Cache look-up and load
  coff = offset(addr);
  if (!hit(base)) memcpy(cache[coff], mem[base], mem[base-4]);
  // Update PC
  PC_cache = coff;
}
```

Listing 1.2: Return

```
ret() {
  // Retrieve return base
  base = sb;
  // Cache look-up and load
  coff = offset(sb);
  if (!hit(base)) memcpy(cache[coff], mem[base], mem[base-4]);
  // Update PC
  PC_cache = coff+so;
}
```

given by `base-coff+`PC_{cache}`+off`. Even when precomputing `base-coff`, this includes two additions that increase the hardware overhead.

Listing 1.4 shows the pseudo-code for `brcf` with absolute addressing. It avoids the address computation of the PC-relative `brcf` and thus reduces the hardware overhead.

The argument for base-relative indirect branches is more subtle. In any case, the instruction requires an addition or subtraction (absolute: PC_{cache} `= addr-base+coff`, base-relative: PC_{cache} `= addr+coff`, PC-relative: PC_{cache} `=` PC_{cache}`+addr`). The address information for indirect branches only becomes available in the execute stage, because the value might require forwarding. Keeping this addition as cheap as possible avoids putting stress on the timing of the path from the execute stage to the fetch stage. Of the three variants, the addition `addr+coff` is "cheapest", because the lowest bits of `coff` are cleared due to the guaranteed alignment of functions to cache blocks in the method cache.

1.2.5 Non-delayed Branches

Delayed branches are a good means to hide the costs of branching. However, the compiler is not always able to fill the delay slots. In these cases, even a primitive strategy such as predict-not-taken provides better performance. Predict-not-taken has very little hardware overhead and can be modeled easily in an exact manner for WCET analysis. By introducing an *additional* instruction for non-delayed immediate branches, Patmos can gain from the advantages of both delayed and non-delayed branches.

Furthermore, having a non-delayed branch instructions enables research towards time-predictable branch-prediction schemes. Depending on our future findings, predict-not-taken could be replaced by predict-backward-taken or time-predictable forms of dynamic branch prediction to further improve both performance and WCET bounds.

Listing 1.3: PC-relative `brcf`

```
brcf(off) {
  // Compute actual address
  addr = base-coff+PC_cache+off;
  // Remember base address
  base = addr;
  // Cache look-up and load
  coff = offset(addr);
  if (!hit(base)) memcpy(cache[coff], mem[base], mem[base-4]);
  // Update PC
  PC_cache = coff;
}
```

Listing 1.4: Absolute `brcf`

```
brcf(addr) {
  // Cache look-up and load
  base = addr;
  coff = offset(addr);
  if (!hit(base)) memcpy(cache[coff], mem[base], mem[base-4]);
  // Update PC
  PC_cache = coff;
}
```

1.2.6 Issue Memory Operations in any Pipeline

Allow stack cache accesses, maybe even any memory access in the second pipeline, as long as there is no memory access to the same memory executed in the first pipeline.

```
  { (p1) sws [1] = $r1  ; (!p1) sws [3] = $r2 }   # allowed, only one predicate is true
  {      swc [r1] = $r4 ;       sws [2] = $r4 }   # allowed, different memories
  {      br  -4         ;       swl [$r1] = $r2 } # could also be allowed
  {      ret            ;       swc [$r1] = $r2 } # could also be allowed
  {      sws [4] = $r1  ;       sws [5] = $r2  }  # not allowed
```

Costs and Benefits

In contrast to the original plan, we do not need to have a double-clocked stack cache, but are still able to spill in the second pipeline while the first pipeline is either accessing main memory or doing control flow, and we can schedule if-converted code and single-path code more in parallel.

While this does not improve the situation for blocks with large sequences of spills, e.g. at prologues and epilogues, it might improve the situation for code with lots of control-flow or if-converted code, even in situations where a double-clocked stack cache access alone does not help.

1.2.7 Interrupts and Exceptions

See Section 8.1 for the proposed implementation and changes to the ISA to support interrupts and exceptions in Patmos.

2 The Architecture of Patmos

2.1 Pipeline

Figure 2.1 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 $\mathtt{EX}_1, \ldots, \mathtt{EX}_n$.

2.1.1 Fetch

Fetch one or two words of instruction from the ROM or method cache. Calculate next PC depending on the length of the instruction bundle.

2.1.2 Decode

Decode the instruction and generate control signals for the following stages. Read register operands. Sign or zero extend immediate operands.

2.1.3 Execute

Read predicate registers. Conditional execute (ALU) instructions. Write predicate register. Calculate effective address for memory operations.

2.1.4 Memory

Read or write memory. This is the only pipeline stage that might stall the pipeline.

2.1.5 Write Back

Write result into destination register.

2.2 Local Memories

Patmos contains several on-chip memories, as sketched in Figure 2.1. We apply the idea of split caches [10] to simplify and enhance the cache analysis. Instructions are fetched from the method cache that caches whole methods. Patmos also supports instruction and data scratchpad memories. Stack allocated data is cached in a stack cache and we envision also a normal data cache with LRU replacement. Accesses to data that are hard to analyze can bypass the data cache.

2.3 Register Files

The register files available in Patmos are depicted by Figure 2.2. In short, Patmos offers:

- 32, 32-bit general-purpose registers (R) : r0, ..., r31
  r0 is read-only, set to zero (0).

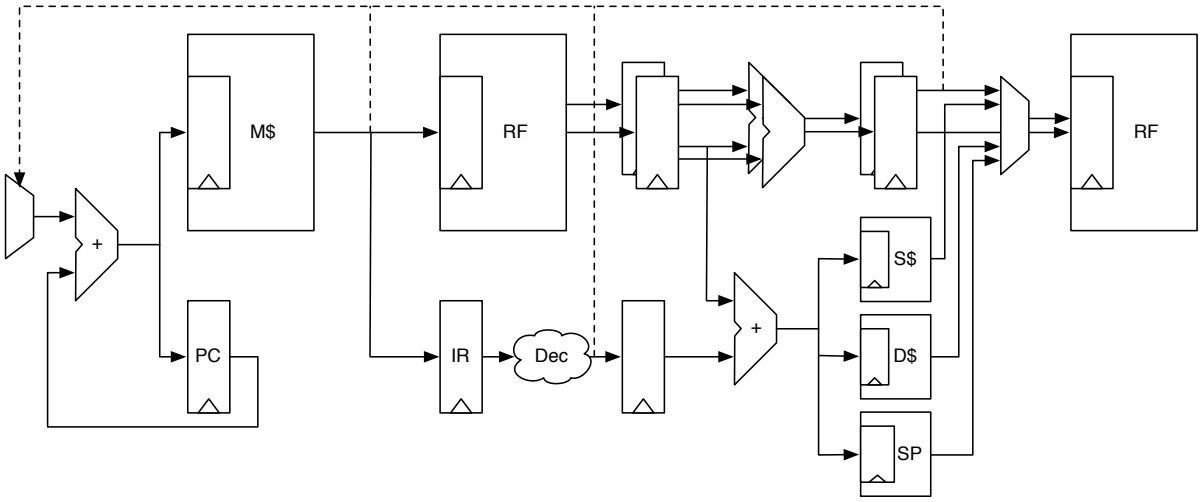


Figure 2.1: Pipeline of Patmos with fetch, decode, execute, memory, and write back stages.

- 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 old value of the register. Reads in subsequent cycles return the result most recently written to the register, i.e., the pipeline implements full forwarding.

When writing concurrently to the same register, the result is undefined. If two instructions of the current bundle have the same destination register, the result is only defined if the predicate of at most one instruction in the bundle evaluates to `true` (1).

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 also be accessed 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.

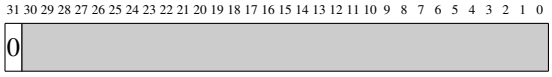
- `s6` can also be accessed through the name `st` and represents a pointer to the top-most element of the content of the stack cache. Updating `s6` does not change `s5` or spill the stack cache.

2.4 Bundle Formats

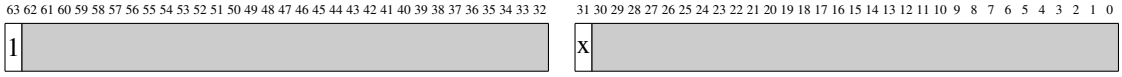
All Patmos instructions are 32 bits wide and are structured according to one of the instruction formats defined in the following section. Up to two instructions can be combined to form an instruction bundle; Patmos bundles are thus either 32 or 64 bits wide. The bundles sizes are recognized by the value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



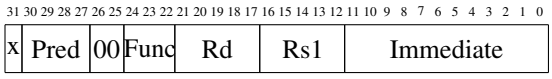
- 64-bit bundle format



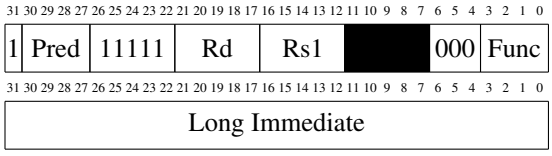
2.5 Instruction Formats

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

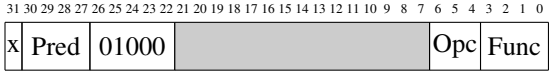
- ALUi – Arithmetic Immediate

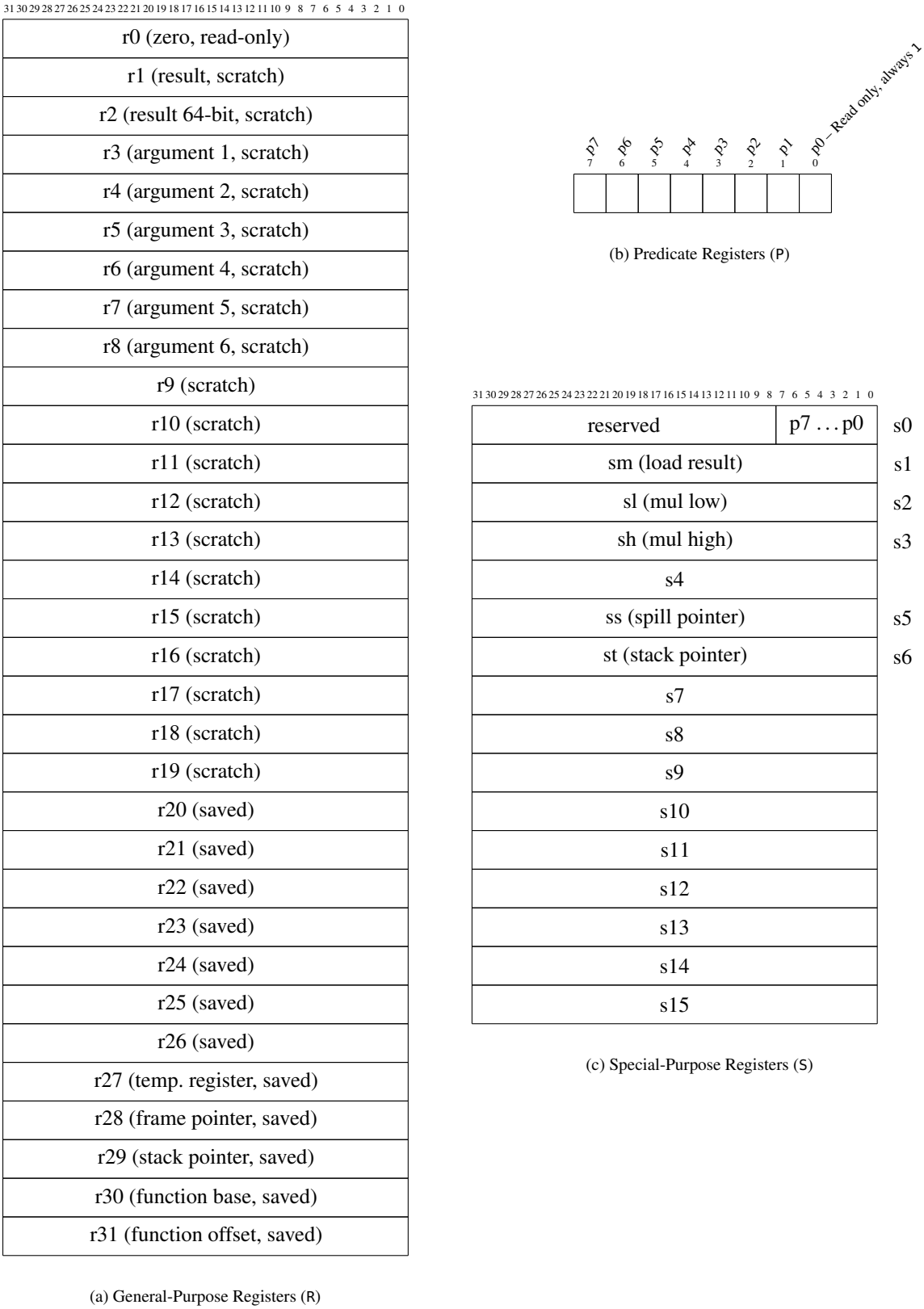


- ALUl – Long Immediate



- ALU – Arithmetic





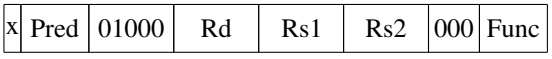
(a) General-Purpose Registers (R)

(b) Predicate Registers (P)

(c) Special-Purpose Registers (S)

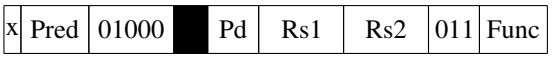
Figure 2.2: General-purpose register file, predicate registers, and special-purpose registers of Patmos.

ALUr – Register

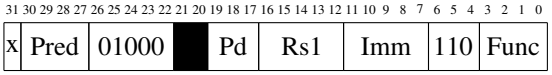


ALUm – Multiply

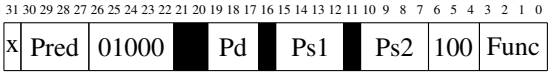
ALUc – Compare



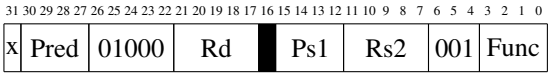
ALUci – Compare immediate



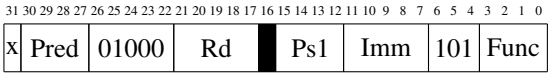
ALUp – Predicate



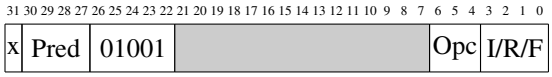
ALUb – Bitmove



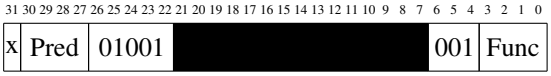
ALUbi – Bitmove immediate



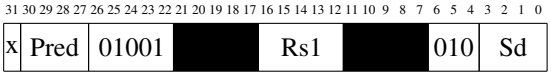
- SPC – Special



SPCw – Wait

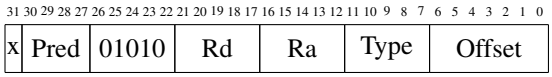


SPCt – Move To Special

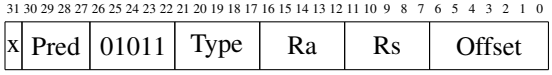


SPCf – Move From Special

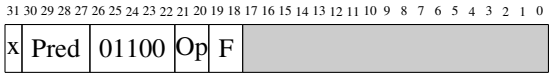
- LDT – Load Typed



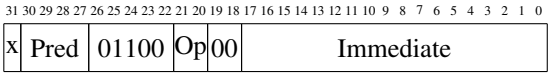
- STT – Store Typed



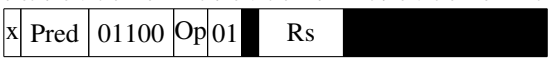
- STC – Stack Control



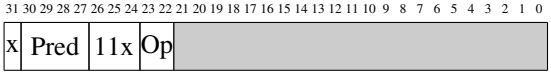
STCi – Stack Control Immediate



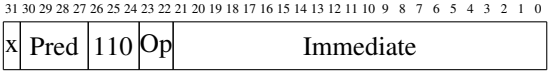
STCr – Stack Control Register



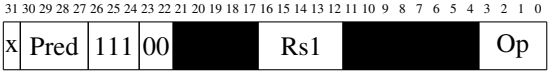
- CFL – Control Flow



CFLb – Call / Branch

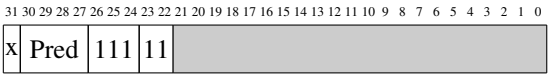


CFLi – Call / Branch Indirect



CFLr – Return

Reserved



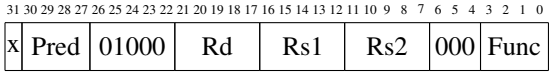
2.6 Instruction Opcodes

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.

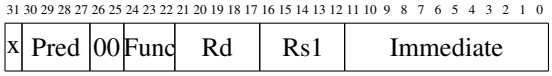
2.6.1 Binary Arithmetic

Applies to the ALUr, ALUi, and ALUl formats. Operand `Op2` denotes either the `Rs2`, or the `Immediate` operand, or the `Long Immediate`. The immediate operand is zero-extended. For shift and rotate operations, only the lower 5 bits of the operand are considered. Table 2.1 shows the encoding of the `func` field; for ALUi instructions, only functions in the upper half of that table are available.

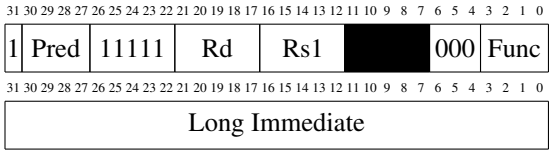
- ALUr – Register



- 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

Table 2.1: General ALU functions

Pseudo Instructions

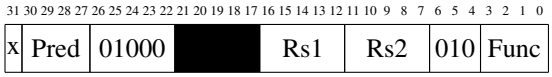
- `mov Rd = Rs` ... `add Rd = Rs + 0`
- `clr Rd` ... `add Rd = r0 + 0`
- `neg Rd = -Rs` ... `sub Rd = 0 - Rs`
- `not Rd = ~Rs` ... `nor Rd = ~(Rs | R0)`
- `li Rd = Immediate` ... `add Rd = r0 + Immediate`
- `li Rd = Immediate` ... `sub Rd = r0 - Immediate`
- `nop` ... `sub r0 = r0 - 0`

Note The use of `sub r0 = r0 - 0` to encode a `nop` pseudo-instruction results in a value of `0x00400000` in the binary instruction stream. This helps in distinguishing the execution of compiler-generated `nops` from executing instructions from memory that happens to be zero.

2.6.2 Multiply

Applies to the ALUm format only. Multiplications are executed in parallel with the regular pipeline and finish within a fixed number of cycles *TODO: how many?*. Table 2.2 shows the encoding of the `func` field for the ALUm instruction format.

- 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

Table 2.2: Multiplication functions

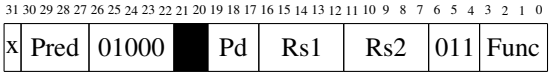
Behavior Perform multiplication in multiple cycles and write the result into destination registers `sl` and `sh`.

Note Multiplications are pipelined, it is thus possible to issue one multiplication on every cycles. Multiplications can only be issued in the first slot.

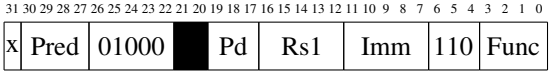
2.6.3 Compare

Applies to the ALUc and ALUci formats only. Tables 2.3 and 2.4 show the encoding of the `func` field for the ALUc and ALUci formats, respectively.

- ALUc – Compare



- ALUci – Compare immediate



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

Table 2.3: Compare functions

Func	Name	Semantics
0000	—	unused
...	...	...
0101	—	unused
0110	btest	`Pd = (Rs1 & (1 << Imm)) != 0`
0111	—	unused
...	...	...
1111	—	unused

Table 2.4: Compare immediate functions

Pseudo Instructions

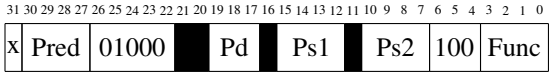
- `isodd Pd = Rs1` ... `btest Pd = Rs1[r0]`
- `mov Pd = Rs` ... `cmpneq Pd = Rs != r0`

Note The predicate register is read and written in the execute stage.

2.6.4 Predicate

Applies to the ALUp format only, the opcodes correspond to those of the ALU operations on general purpose registers. Table 2.5 shows the encoding of the `func` field for the ALUp format.

- 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
...	...	...
1111	—	unused

Table 2.5: Predicate functions

Pseudo Instructions

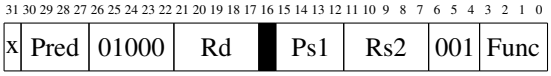
- `pmov Pd = Ps` ... `por Pd = Ps | Ps`
- `pnot Pd = ~Ps` ... `pxor Pd = (Ps ^ p0)`
- `pset Pd = 1` ... `por Pd = p0 | p0`
- `pclr Pd = 0` ... `pxor Pd = p0 ^ p0`

Note The predicate register is read and written in the execute stage. 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).

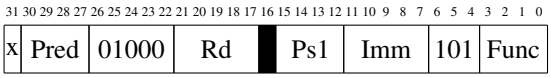
2.6.5 Bitmove

Applies to the ALUb, ALUbi formats only. Table 2.6 shows the encoding of the `func` field for the ALUb and ALUbi formats.

- ALUb – Bitmove



- ALUbi – Bitmove immediate



Func	Name	Semantics
0000	bcopy	`Rd[Rs2] = Ps1` `Rd[Imm] = Ps1`
0001	—	unused
...	...	...
1111	—	unused

Table 2.6: Bitmove functions

Pseudo Instructions

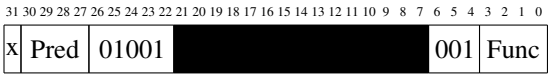
- `mov Rd = Ps` ... `bcopy Rd = Pd, r0`

Note `bcopy` sets a single bit in the destination register `Rd` to the value in `Ps1`. The position of the bit is specified by the lowest 5 bits of `Rs2` or by `Imm`, respectively. All other bits in `Rd` remain unchanged.

2.6.6 Wait

Applies to the SPCw format only. Wait for a multiplication or memory operation to complete by stalling the pipeline. Table 2.7 shows the encoding of the `func` field.

- SPCw – Wait



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

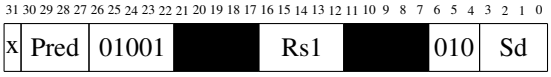
Table 2.7: Wait function encoding

Note A Wait can only be issued at the first position within a bundle. This instruction is not (yet) implemented in hardware as we do not (yet) support split loads.

2.6.7 Move To Special

Applies to the SPCt format only. Copy the value of a general-purpose register to a special-purpose register. The only instruction is `mts`, which stores the content of general-purpose register `Rs1` in special register `Sd`.

- SPCt – Move To Special

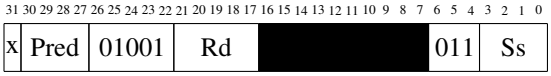


Note Special registers `sm`, `sl`, `sh` are read-only, writing to those registers may result in undefined behavior.

2.6.8 Move From Special

Applies to the SPCf format only. Copy the value of a special-purpose register to a general-purpose register. The only instruction is `mfs`, which loads the content of special register `Ss` to general-purpose register `Rd`.

- SPCf – Move From Special



2.6.9 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.

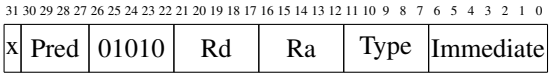
In addition, *decoupled* loads are provided that do *not wait* for the memory access to be completed. The result is then loaded into the special register `sm`. A dedicated `wait.mem` instruction can be used to stall the pipeline explicitly until the load is completed.

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



Note All 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 [$r1 + 5]
    ...
  { dlwc [$r2 + 7] ; mfs $r2 = $sm }
```

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

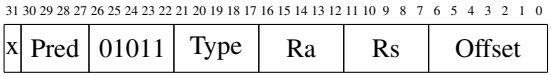
Table 2.8: Typed loads

2.6.10 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



Type	Name	Semantics
000 | 00	sws	$\texttt{sc[Ra+(Imm} \ll \texttt{2)]}_{32} = \texttt{Rs}$
000 | 01	swl	$\texttt{lm[Ra+(Imm} \ll \texttt{2)]}_{32} = \texttt{Rs}$
000 | 10	swc	$\texttt{dc[Ra+(Imm} \ll \texttt{2)]}_{32} = \texttt{Rs}$
000 | 11	swm	$\texttt{gm[Ra+(Imm} \ll \texttt{2)]}_{32} = \texttt{Rs}$
001 | 00	shs	$\texttt{sc[Ra+(Imm} \ll \texttt{1)]}_{16} = \texttt{Rs}_{[15:0]}$
001 | 01	shl	$\texttt{lm[Ra+(Imm} \ll \texttt{1)]}_{16} = \texttt{Rs}_{[15:0]}$
001 | 10	shc	$\texttt{dc[Ra+(Imm} \ll \texttt{1)]}_{16} = \texttt{Rs}_{[15:0]}$
001 | 11	shm	$\texttt{gm[Ra+(Imm} \ll \texttt{1)]}_{16} = \texttt{Rs}_{[15:0]}$
010 | 00	sbs	$\texttt{sc[Ra+Imm]}_{8} = \texttt{Rs}_{[7:0]}$
010 | 01	sbl	$\texttt{lm[Ra+Imm]}_{8} = \texttt{Rs}_{[7:0]}$
010 | 10	sbc	$\texttt{dc[Ra+Imm]}_{8} = \texttt{Rs}_{[7:0]}$
010 | 11	sbm	$\texttt{gm[Ra+Imm]}_{8} = \texttt{Rs}_{[7:0]}$
01100	—	unused
...	...	...
11111	—	unused

Table 2.9: Typed stores

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

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

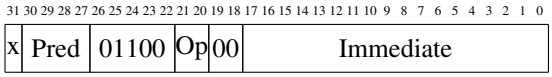
2.6.11 Stack Control

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 3.3. Table 2.10 shows the encoding of operations for STCi, while Table 2.11 shows the encoding for STCr.

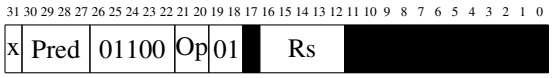
- STCi – Stack Control Immediate



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

Table 2.10: Stack control operations with immediates

- 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

Table 2.11: Stack control operations for registers

Behavior

`sres`: Check free space left in the stack cache. Update stack-cache registers. If needed, spill to global memory using `ss`.

`sense`: Check reserved space available in the stack cache. If needed, refill from global memory using `ss`.

`sfree`: Account for $\texttt{head} - \texttt{tail} < 0$, update `ss` and `st`. Update stack-cache register `head`.

`sspill`: Update `ss` and `st`. Update stack-cache register `tail`. Spill to global memory using `ss`.

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.

2.6.12 Control-Flow 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 7.3). `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.

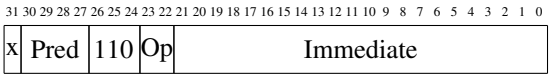
Instruction	Immediate	Indirect	Cache fill	Link
call	absolute	absolute	yes	yes
br	PC relative	absolute	no	no
brcf	PC relative	absolute	yes	no

`br` instructions are executed in the `EX` stage, while `brcf`, `call` and `ret` instructions are executed in the `MEM` stage. The instructions fetched in the meantime are *not* aborted. This corresponds 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 fill the delay slots.

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`, and `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 3.4.

- 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

Table 2.12: Control-flow operations

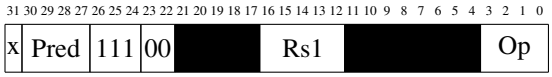
Behavior – call, brcf, and system call Store link information into R31. Method base is not stored to a visible register (this must be done by the caller, by convention using R30). Compute cache-relative program counter and update the PC.

Behavior – branch within cache Compute new, cache-relative program counter value. Update program counter.

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.

- 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

Table 2.13: Indirect control-flow operations

Behavior – call, branch, and system call 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.

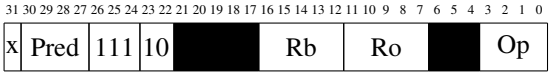
Behavior – branch within cache Compute new, cache-relative program counter value. Update program counter.

Note All branch/call instructions can only be issued on the first position within a bundle.
See also CFLi format notes.

2.6.13 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. Table 2.14 shows the encoding of the `func` field.

- CFLr – Return



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

Table 2.14: Return function encoding

Behavior 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 7.3). The return function offset `Ro` is provided by the `call` instruction in `r31`. Note that using `r30` and `r31` is a compiler convention. The unit and the addressing mode of the function offset is hardware implementation dependent.

2.7 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).

2.8 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 semi-colon `;` or a newline denotes the end of an instruction or operation. If an instruction contains two operations, the operations in the bundle must be enclosed by curly brackets. Bundles do not need to be separated by newlines or semi-colons. For bundles consisting of only one operation, the curly brackets are optional. 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 omitted, 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; nop # then wait 2 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:
       ...
```

2.8.1 Instruction Mnemonics

The LLVM assembler supports the instructions mnemonics as specified in this document, including all pseudo instructions.

The `paasm` assembler and the `pasim` simulator use the same basic instruction mnemonic, but a `i` or `l` suffix is appended for *immediate* and *long immediate* variants, while no suffix in general refers to the register indirect variant of the instructions. As exception, the control flow instructions use a `r` suffix for the register indirect variants and no suffix for the immediate instructions.

2.8.2 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));
```

3 Memory and I/O Subsystem

3.1 Local and Global Address Space

The typed loads of Patmos imply two address spaces: a local address space that is accessed through local loads and stores, and a global address space that is accessed when using other access types. All caches use memory that is mapped to the global address space as backing memory. For example, the data cache fetches data from global memory on a cache miss, and the stack cache uses global memory for spilling and filling. Consequently, there are two memory maps, one for the local address space and one for the global address space. Tables 3.1 and 3.2 show the respective address mappings. To simplify address decoding, the top four bits (A31–A28) are generally used to distinguish between different memory and I/O areas. The address range for I/O devices is divided further to distinguish the different devices, as discussed in Section 3.2.

As `call`, `ret`, and `brcf` do not include memory type information, the distinction between memory areas for these instructions is done solely through the address mapping. The boot instruction ROM and the instruction scratchpad memory are mapped to the lowest 128K of the global address space. Note that this applies only to these instructions; i.e., a `call` to address `0x00010100` executes code that is located in the instruction scratchpad, while non-local loads or stores to the same address access the external SRAM. Therefore, a binary that is loaded to external memory can use the lowest 128K of memory for data segments, but not for code segments.

The global address space also includes a ROM and a scratchpad for booting. The boot data ROM enables boot programs to use initialized data segments. By copying (parts of) the boot data ROM to the boot scratchpad, these programs can also use initialized data segments that require write access. Note that these memory areas can be accessed by loads and stores only; it is not possible to execute code located in them.

3.1.1 Boot Memories

By convention, the first four words of the boot data ROM contain information about data to be copied to the boot data scratchpad before starting actual execution. Table 3.3 shows the respective data fields. Upon start, the program should copy `src_size` bytes of data from `src_start` to `dst_start`. If `dst_size` is greater than `src_size`, the remaining bytes are filled with zeroes.

3.2 I/O Devices

Each processor contains a minimum set of standard I/O 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`.

Address	Memory area
`0x00000000–0x0000ffff`	Data Scratchpad Memory
`0x00010000–0x0001ffff`	Instruction Scratchpad Memory (write only)
`0xe0000000–0xe7ffffff`	NoC interface configuration registers
`0xe8000000–0xefffffff`	NoC communication memory
`0xf0000000–0xffffffff`	I/O devices

Table 3.1: Address mapping for local address space

Address	Memory area
0x00000000–0x0000ffff	Boot Instruction ROM (only for code)
0x00010000–0x0001ffff	Instruction Scratchpad Memory (only for code)
0x00000000–0x7fffffff	External SRAM
0x80000000–0x8000ffff	Boot Data ROM (only for data)
0x80010000–0x8001ffff	Boot Data Scratchpad Memory (only for data)

Table 3.2: Address mapping for global address space

Address	Name	Description
0x80000000	src_start	Start address of data to be copied
0x80000004	src_size	Size of data to be copied
0x80000008	dst_start	Destination for copying
0x8000000c	dst_size	Size of initialized data

Table 3.3: Boot data initialization information

Address	I/O Device	read	write
0xf0000000	cpuinfo	processor ID	–
0xf0000004	cpuinfo	clock frequency (Hz)	–
0xf0000200	timer	clock cycles (high word)	–
0xf0000204	timer	clock cycles (low word)	–
0xf0000208	timer	time in μs (high word)	–
0xf000020c	timer	time in μs (low word)	–
0xf0000800	UART	status	control
0xf0000804	UART	receive buffer	transmit buffer
0xf0000900	LED	–	output register

Table 3.4: I/O devices and registers

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 3.4 shows the I/O devices and the registers.

3.2.1 UART

The UART is a minimal IO device for `stdout` and `stdin`. It is also used for program download. Table 3.5 shows the bits of the control register.

The UART address for `pasim` is defined in `patmos/simulator/include/uart.h`. For the compiler/library the constant is in `llvm/tools/clang/lib/Driver/Tools.cpp`.

Proposal for CMP UART Sharing 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 by a marker from which core the data is coming. The marker byte may be precede each data byte or it may be

Bit	Status		Control	
0	TRE	TX Transmit ready	–	–
1	DAV	RX Data available	–	–

Table 3.5: UART status bits

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/demultiplex the different data streams.

3.3 The Stack Cache

The stack cache is a processor-local, on-chip memory [1]. The stack cache operates similar to a ring buffer. It can be seen as a stack-cache-sized window into the main memory address range. To manage the stack cache, we use three additional instructions: `reserve`, `ensure`, and `free`. Two hardware registers define which part of the stack area is currently in the stack cache.

3.3.1 Stack Cache Manipulation

We present the mechanics of the stack cache in C code for easier readability. However, the hardware implementation is a synchronous design and the algorithm is implemented by a state machine that handles the memory spill and fill operations. In the C code following data structures are used:

`mem` is an array representing the main memory,

`sc` is an array representing the stack cache,

`m_top` is the register pointing to the top of the saved stack content in the main memory, and

`sc_top` points to the top element in the stack cache.

The two pointers are full-length address registers. However, when addressing the stack cache, only the lower n bits are used for a stack cache of a size of 2^n words. The constant `SC_SIZE` represents the stack cache size and `SC_MASK` is the bit mask for the stack cache addressing. The stack cache is managed in 32-bit words.

At program start the stack cache is empty and both pointers, `m_top` and `sc_top`, point to the same address, the address that one higher as the stack area. `m_top` points to the last spilled word in main memory. Similar, `sc_top` points to the last slot in the stack frame (top of stack). Therefore, the number of currently valid elements in the stack cache is `m_top - sc_top`.

The compiler generates code to grow the stack downward, as it is common for many architectures. Growing the stack downwards has historical reasons. However, for multi-threaded systems each thread needs a reserved, fixed memory area for the stack and there is no benefit from growing the stack downwards.

Reserve The `reserve` instruction, as shown in Figure 3.1, reserves space in the stack cache. Typed load and store instructions use this reserved space. The reserve instruction may spill data to the main memory. This spilling happens when there are not enough free words in the stack cache to reserve the requested space.

The processor reads the number of words to be reserved (the immediate operand of the instruction) in the decode stage. The processor adjusts the `sc_top` register in the execution stage and also computes how many words need to be spilled in the execution stage. The processor spills to the main memory in the memory stage, as shown by the for loop in Figure 3.1.

Free The `free` instruction frees the reserved space on the stack. It does not fill previously spilled data back into the stack cache. It just changes the top of the stack pointer and may change the top of the memory pointer, as shown in Figure 3.2.

```
void reserve(int n) {

    int nspill, i;

    sc_top -= n;
    nspill = m_top - sc_top - SC_SIZE;
    for (i=0; i<nspill; ++i) {
       --m_top;
       mem[m_top] = sc[m_top & SC_MASK];
    }
}
```

Figure 3.1: The `reserve` instruction provides `n` free words in the stack cache. It may spill data into main memory.

```
void free(int n) {

    sc_top += n;
    if (sc_top > m_top) {
        m_top = sc_top;
    }
}
```

Figure 3.2: The `free` instruction drops `n` elements from the stack cache. It may change the top memory pointer `m_top`.

Ensure Returning into a function needs to ensure that the stack frame of this function is available in the stack cache. The `ensure` instruction, as shown in Figure 3.3, guarantees this condition. This instruction may need to fill back the stack cache with previously spilled data. This happens when the number of valid words in the stack cache is less than the number of words that need to be in the stack cache. Filling the stack cache is shown in the loop in Figure 3.3.

One processor register serves as stack pointer and points to the end of the stack frame. Load and store instructions use displacement addressing relative to this stack pointer to access the stack cache.

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 `m_top` - `sc_top` $>$ *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 `sc_top` 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.

```
void ensure(int n) {

    int nfill, i;

    nfill = n - (m_top - sc_top);
    for (i=0; i<nfill; ++i) {
        sc[m_top & SC_MASK] = mem[m_top];
        ++m_top;
    }
}
```

Figure 3.3: The `ensure` instruction ensures that at least `n` elements are valid in the stack cache. It may need to fill data from main memory.

```
// load one word from the stack cache
// addr is a main memory address (register value plus offset)

int load(int addr) {
    return sc[(sc_top + addr) & SC_MASK];
}

// store one word into the stack cache
// addr is a plain main memory address

void store(int addr, int val) {
    sc[(st_top + addr) & SC_MASK] = val;
}
```

Figure 3.4: Pseudo code for the load and store instructions.

- 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 7).

3.4 Method Cache

An overview of alternative design options with regard to the method cache can be found in Section 3.4.4. 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.

3.4.1 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.

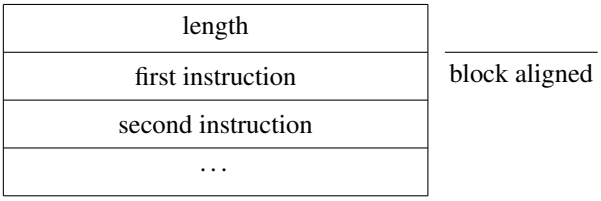


Figure 3.5: Layout of code sequences intended to be cached in the method cache.

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 3.5 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.

3.4.2 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.

The design of the method cache with FIFO replacement is based on the design for the Java processor JOP [7] .

3.4.3 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.

3.4.4 Method Cache Options

We have several options to implement the method cache and related instructions (call, return). Note, all replacement policies operate on the method/function level.

- Fetch on call, with FIFO replacement
  On a call it is checked whether the target method is in the cache or not. If yes, execution continues. Otherwise, the call instruction starts fetching the method into the cache, the pipeline is stalled.

- Fetch on call, with FILO replacement
  Same as above, only that the replacement strategy if FILO, i.e., like a stack.

- prefetch before call, with FILO replacement
  We define two instructions to perform a function call. A compiler-placed prefetch instructions ensure that the target method is either in the cache, or triggers the loading of the function. Calls merely check whether the method is completely loaded and transfer control. (FIFO replacement is not possible due to potential eviction of the current method executing the prefetch).

- prefetch before call, explicit unload, with FILO replacement
  Same as before, with an additional *unload* instruction to evict methods explicitly from the cache – i.e., to free space after as shown by the following example:

```
A()
// unload here to avoid mutual eviction
// in loop of B and C, assuming A+B, A+C,
// B+C fit in the cache, but not A+B+C
while(true) {
  B() //
  C();
}
```

- fetch on call, LRU replacement

- prefetch before call, LRU replacement
  This is everybody's darling: compiler-placed prefetching with stalling call instructions. Safe, clean, and expensive in hardware ;-)

3.5 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.

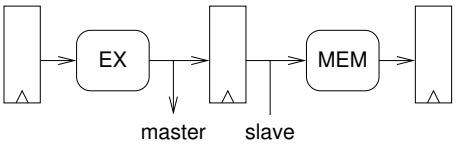


Figure 3.6: Localization of OCP signals in the pipeline

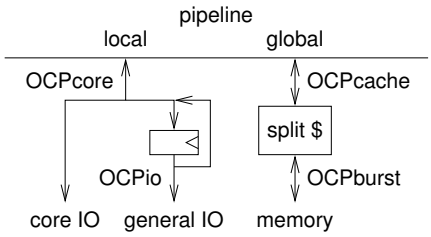


Figure 3.7: OCP levels in Patmos

3.6 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 [9, 4] where objects (heap allocated structures) are tracked via a fully associative cache. Furthermore, for arrays with low temporal locality a small set of prefetch buffers may benefit from spacial locality.

3.7 Hardware Interface

For the connection of Patmos to a memory controller, I/O devices, the core-to-core network on chip, and/or the memory arbiter an interface standard needs to be specified. Several standards are available. We decided to base the interface on OCP[1] [5] and subset the standard as we need it.

3.7.1 Description

Figure 3.6 shows the OCP signals in the Patmos pipeline. The master signals are generated in the execute stage, and the slave signals are captured in the memory stage.[2]

The different variants of the OCP protocol in the scope of the Patmos processor are shown in Figure 3.7.

`OCPcore`

The variant of OCP generated by the pipeline for the local address space. The respective signals are shown in Table 3.6. `OCPcore` is tailored to accesses to on-chip memories, which on FPGAs necessarily include an internal register. To enable sub-word transfers of data, the signal `MByteEn` is used. The following assumptions apply:

- Only reads (`RD`) and writes (`WR`) are supported. Writes require a response (`writeresp_enable=1`), such that every command must be followed by a response.
- No `SCmdAccept` or `MRespAccept`, flow control is done solely via `SResp`.

[1] `http://www.ocpip.org/`

[2] For clarity, the handling of both parts is implemented in the file `Memory.scala`.

Table 3.6: OCPcore signals

Signal	Description
MCmd	Command
MAddr	Address, byte-based, lowest two bits always 0
MData	Data for writes, 32 bits
MByteEn	Byte enables for sub-word accesses, 4 bits
SResp	Response
SData	Data for reads, 32 bits

- Slaves may generate responses earliest in the cycle after a command.
- The master may issue commands in the same cycle as the slave sends its response, i.e., basic support for pipelining is required.
- MByteEn is assumed to be properly aligned (force_aligned=1). The signal can be ignored for read accesses without side-effects.

OCPio

The OCPio level is derived from the OCPcore level by inserting a register in the master signals. It is slightly more flexible than OCPcore and appropriate for I/O devices that do not (or cannot) follow the semantics of OCPcore. Registering the master signals changes the protocol as follows:

- Slaves may generate responses in the same cycle as they receive a command.
- Commands are issued earliest in the cycle after a response (no pipelining).
- SCmdAccept is supported. It is sufficient to register the master signals only if the currently registered command is IDLE or SCmdAccept is high.
- In order to have symmetric handshaking for commands and responses and to facilitate clock-domain crossing, OCPio also includes a signal MRespAccept. An OCPio port that is derived directly from the pipeline's OCPcore port always accepts responses.

OCPcache

This OCP variant is generated for the global address space and is used for communication between the pipeline and the caches. It is the same as OCPcore, but includes an additional signal MAddrSpace to specify the cache that should serve the access.

OCPburst

The caches access the external memory through bursts only; Table 3.7 shows the signals of the OCPburst interface. The tie-off value for MBurstLength is 4, and MBurstSingleReq is tied off to 1. This means that the master supplies four data words for each write command, and the slave returns four words for each read command. All other burst-related signals are tied off to their default values. This entails that the only sequence for burst addresses is INCR. Bursts always stat at an address that is aligned to the burst size (burst_aligned=1). Furthermore, reqdata_together is set to 1, i.e., write commands and the respective first word are issued together. Instead of the signal MByteEn, OCPburst uses the signal MDataByteEn. This implies that partial write transfers are fully supported, but partial read commands are unsupported.

We assume that the master provides data for burst accesses in consecutive cycles and that slaves can accept all burst data words once they accept the first word. To enable handshaking for the acceptance of the first data word, the OCPburst variant includes the signals MDataValid and SDataAccept. As reqdata_together is set to

Table 3.7: OCPburst signals

Signal	Description
MCmd	Command
MAddr	Address, byte-based, lowest two bits always 0
MData	Data for writes, 32 bits
MDataByteEn	Byte enables for sub-word writes, 4 bits
MDataValid	Signal that data is valid, 1 bit
SResp	Response
SData	Data for reads, 32 bits
SCmdAccept	Signal that command is accepted, 1 bit
SDataAccept	Signal that data is accepted, 1 bit

1, delaying the acceptance of data also delays the acceptance of write commands. In order to do the same for read commands, OCPburst also includes the signal SCmdAccept. The signals SCmdAccept and SDataAccept can be generated by the same logic. For the acceptance of write commands they must be identical, otherwise at least one of the signals can have an undefined value. We assume that slaves return burst read data in consecutive cycles

The first response to a read command may be given in the cycle after the command. The response to a write command may be given earliest in the cycle after the last data word was sent. Commands may be issued earliest in the cycle after the last response from an earlier command is received.

3.7.2 Remarks

SCmdAccept is valid only while a command is unequal to IDLE. Consequently, SCmdAccept must be properly multiplexed to support multiple slaves. For handshaking via SResp, it is sufficient to combine the responses of different slaves with OR.

Note 4: Bursts are restricted to exactly four words (D1, D2, D3 and D4), or some other constant which is a power of 2. The address must be aligned. Burst data must be provided in four consecutive cycles. SResp is active during these cycles. For a burst write the master may have to provide D1 for two or more cycles if SResp is not active in the first cycle of the transaction.

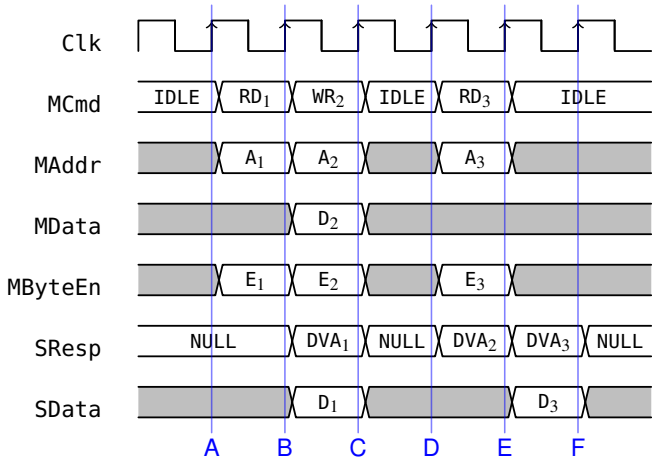


Figure 3.8: Timing diagram for OCPcore

3.7.3 Timing Diagrams

OCPcore

Figure 3.8 shows a sequence read/write/read in OCPcore, where the slave delays the response to the write by one cycle.

A: The master issues a read by setting MCmd to RD, MAddr to A_1 and MByteEn to E_1.

B: The slave responds to the read issued in cycle A by setting SResp to DVA and returning the appropriate data. The master can issue the next command in the same cycle as it receives the response and issues a write command WR. The master provides the byte enable value E_2 along with the data D_2 to specify which bytes should be actually written.

C: The slave does not respond immediately and the master is stalled. MCmd must be IDLE while the master is stalled.

D: The slave responds to the write issued in cycle B. The master issues a read in the same cycle.

E: The slave responds to the read issued in cycle D.

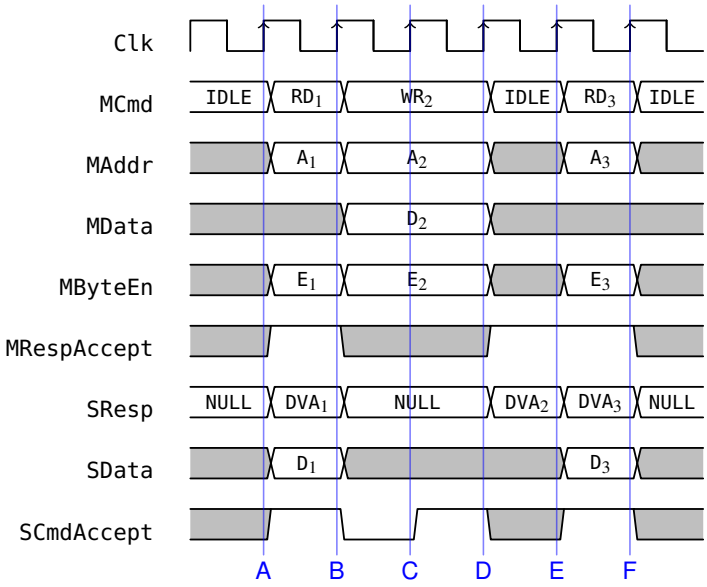


Figure 3.9: Timing diagram for `OCPio`

`OCPio`

Figure 3.9 shows a sequence read/write/read in `OCPio`, where the slave does not accept the write immediately and delays the response to the write by one cycle.

A: The master issues a read by setting `MCmd` to `RD`. The slave accepts the command by setting `SCmdAccept` to high and responds immediately by setting `SResp` to `DVA` and returning the appropriate data.

B: The master issues a write command `WR` with data D_2 and byte enables E_2. The slave signals that it does not accept the command by setting `SCmdAccept` to low.

C: As the slave did not accept the command in cycle B, the master still issues the command. The slave accepts the command by setting `SCmdAccept` to high.

D: The slave responds to the write it accepted in cycle C. Note that a) the master is not allowed to issue a new command immediately and b) `SCmdAccept` may take any value, because `MCmd` is `IDLE`.

E: The master issues a read to which the slave responds immediately.

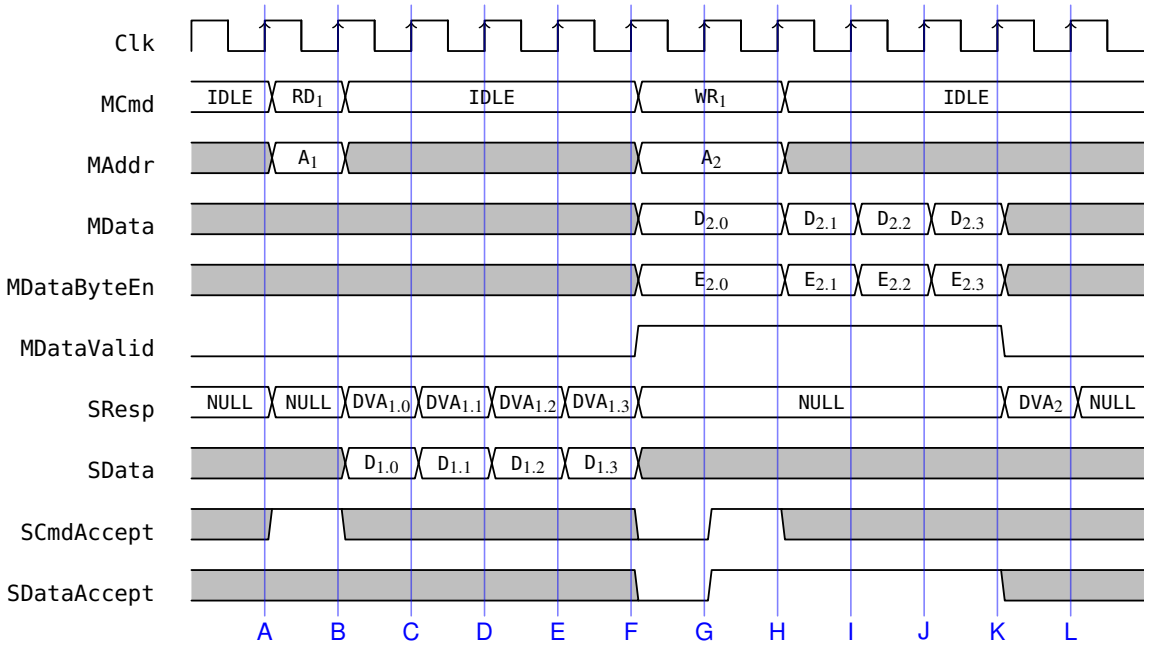


Figure 3.10: Timing diagram for `OCPburst`

`OCPburst`

Figure 3.10 shows a read followed by a write in `OCPburst`, were the slave does not accept the first data word immediately.

A: The master issues a read command by setting `MCmd` to `RD`. The slave accepts by asserting `SCmdAccept`.

B: The slave provides the first response $\texttt{DVA}_{1.0}$, with data from address $\texttt{A}_1$.

C: The slave provides the second response $\texttt{DVA}_{1.1}$, with data from address $\texttt{A}_1$+4.

D: The slave provides the third response $\texttt{DVA}_{1.2}$, with data from address $\texttt{A}_1$+8.

E: The slave provides the fourth and last response $\texttt{DVA}_{1.3}$, with data from address $\texttt{A}_1$+12.

F: The master issues a write command by setting `MCmd` to `WR` and provides the first data word $\texttt{D}_{2.0}$ width byte enables $\texttt{E}_{2.0}$ It signals that the data is valid by asserting `MDataValid`. The slave signals that it cannot accept the data by setting `SDataAccept` to low.

G: As the slave did not accept the data in cycle F, the master keeps issuing the command and providing the data word $\texttt{D}_{2.0}$. The slave now accepts the data by setting `SDataAccept` to high.

H: The master provides the second data word, $\texttt{D}_{2.1}$ with byte enables $\texttt{E}_{2.1}$.

I: The master provides the third data word, $\texttt{D}_{2.2}$ with byte enables $\texttt{E}_{2.2}$.

J: The master provides the fourth and last data word, $\texttt{D}_{2.3}$ with byte enables $\texttt{E}_{2.3}$.

K: The slave responds to the write burst by setting `SResp` to `DVA`.

4 Implementation

After a first implementation of Patmos in VHDL we did a cleanup and rewrite in a the hardware description language Chisel [2]. The following notes on the implementation of Patmos and implementation decisions is based on first design discussions within the VHDL version and concrete implementation experiments with Chisel. All size and frequency numbers are from the Chisel implementation. A comparison between VHDL and Chisel would be of great interest.

LoC, excluding the copyright header at 6.4.2013: Chisel: 996 VHDL: 3020

4.1 Component Organization and Pipeline Structure

The architecture of Patmos is structured around five components, each representing one pipeline stage. Each component contains the *left* pipeline register. E.g., the output of the `DEC` stage (decode signals, the two register values, and the immediate field) is combinatorial from the decode stage and registered in the `EX` stage. The motivation of this organization is that input registers of on-chip memory elements (e.g., instruction memory, register file, and data memory) are part of the pipeline register. They need to be fed unconditionally from the unregistered output of the former stage.

Each stage has exactly one pipeline register, which is placed at the begin of the component. The pipeline registers use an enable for stalling. Register that have no enable (input registers of on-chip memories) need a *shadow* register and a multiplexer for stalls.

The interface from the EX stage to the MEM stage might use one field for ALU results and the store data or individual fields. Individual fields might reduce the pressure on the ALU multiplexers.

4.2 Register File

There are two options to implement the register file (RF) in an FPGA: (1) use two on-chip memories to provide two read ports and one write port, or (2) use dedicated registers and larger multiplexer structures for the read ports. Usually one aims to use on-chip memory for the RF. However, in a design constraint largely by the available amount of on-chip memory, a RF built out of registers might be preferable.

For a dual issue pipeline we need 4 read ports and 2 write ports into the RF. We explored double clocking of a on-chip based RF in [11]. It is feasible, the resulting maximum frequency fits for the ALU path, but feels a little bit brittle. A RF from registers might give a more robust design for the two write ports. Furthermore, Chisel does not provide the possibility to use more than one clock.

The ideal solution would be to make it configurable if on-chip memory or LCs are used. The issue width should also be configurable.

4.3 Resource and Fmax Numbers

State 13.3.2013 with Chisel and DE2-70: A shared field (for EX to MEM?) results 3435 LCs and 81.7 MHz, two fields in 3499 LCs and 81.8 MHz. Looks like not a big deal, but just 64 more LCs. Where does this cost come from? A very inefficient enable on the pipeline register (MUX instead of an enable signal?).

4.4 ALU Discussion

The large multiplexers and the forwarding limit the maximum frequency. We have already removed the expensive rotate instructions. Maybe more can go (e.g. `abs`).

Current version (4.4.2013) with all ALU operations and test case ALU.s for the DE2-70 is: 3415 LCs, 85.44 MHz. Dropping `rsub` and all unary ALU operations: 3173 LCs, 91.91 MHz.

5 Build Instructions

In the following we present an example configuration on a Linux/Ubuntu system. Patmos and the compiler have also been successfully been installed on a Mac OSX system. The support of Windows is marginal, or basically not existent.

5.1 Setup On Ubuntu 13.04

After a plain Ubuntu installation several packages need to be installed. The following apt-get lists the packages that need to be installed:[1]

```
sudo apt-get install openjdk-7-jdk git gitk cmake make g++ texinfo flex bison \
  libelf-dev graphviz libboost-dev libboost-program-options-dev ruby1.9.1 \
  gtkwave
```

For the Quartus setup it is best to change the default shall to `/bin/bash`:

```
sudo rm /bin/sh
sudo ln -s /bin/bash /bin/sh
```

5.1.1 Compiler and Patmos

We assume that the T-CREST project will live in `$HOME/t-crest`. Patmos and the compiler can be checked out from GitHub and built as follows:

```
mkdir ~/t-crest
cd ~/t-crest
git clone https://github.com/t-crest/patmos-misc.git misc
./misc/build.sh
```

For developers with push permission generate an ssh key and upload it at GitHub (see `https://help.github.com/articles/generating-ssh-keys` for detailed instructions). The ssh based clone string for write access is then:

```
git clone git@github.com:t-crest/patmos-misc.git misc
./misc/build.sh
```

This will checkout several other repositories (the compiler, library, the Patmos source, and benchmarks) and build the compiler, the Patmos simulator, and the test benches. Therefore, take a cup of coffee and find some nice reading. After building the compiler add the path to the compiler executables into your `.profile`: [2]

```
export PATH="/home/martin/t-crest/local/bin:$PATH"
```

A complete logout from Ubuntu is needed to take effect (just closing a terminal window is not enough).
For correct correct signing of your changes set the username and email in git with:

```
git config --global user.name "Joe Someone"
git config --global user.email "joe.someone@domain.com"
```

[1] Some packages might be available in newer version when reading this document.
[2] The path needs to be absolute. LLVM cannot handle a path relative to the home folder ~, e.g., `~/t-crest/local/bin`.

5.1.2 Quartus

Download the free web edition of Quartus from Altera. The Linux version is installed as follows:[3]

```
tar xvf Quartus-web-xxx.tar
```

The software installation is started with a plain `setup.sh`. Then add the bin directory of Quartus to your $PATH.

5.2 Hello World

One might start with the standard, harmless looking Hello World:

```
main() {
    printf("Hello Patmos!\n");
}
```

With the compiler installed it can be compiled to a Patmos executable and run with the simulator as follows:

```
patmos-clang hell.c
pasim a.out
```

However, this innocent examples is quiet challenging for an embedded system: It needs a C compiler, an implementation of the standard C library, printf itself is a challenging function, the generated ELF file needs to be understood by a tool and the individual sections downloaded, and finally a terminal (often a serial line) needs to be available on the target, and your test PC needs to have a serial line as well and a terminal program needs to run.

Therefore, we might start with a minimal assembler program and execute that in the simulator and emulator.

5.3 Building Patmos

The whole build process of Patmos,[4] applications in assembler and in C, configuration of the FPGA, and downloading an application is `Makefile` based.

The complete design flow (including the LLVM based C compiler) can execute in a Linux machine. The flow without the C compiler should be able to execute in a Windows/Cygwin environment. Under Mac OS X all tools, except Quartus, are working (ModelSim under wine). For FPGA synthesis and configuration Windows XP within a VMWare virtual machine is a possible solution.

On a Linux box with the installed LLVM compiler in your PATH,[5] the complete build processes is kicked of by:

```
make BOOTAPP=bootable-bootloader APP=helloworld \
   tools comp gen synth config download
```

Here a list of the most important make targets:

tools build of all tools, including the Patmos software simulator

asm assemble source (from folder `asm`)

emulator build the Chisel based C++ emulator

csim execute the Patmos emulator

comp compile a C program as loadable ELF binary

bootcomp compile a C program as a bootable image

[3] `http://www.altera.com/literature/manual/quartus_install.pdf`

[4] Get the source from GitHub with: `git clone git@github.com:t-crest/patmos`

[5] Installation and build is a simple two step approach: `git clone git@github.com:t-crest/patmos-misc.git misc` and build with: `./misc/build.sh`

gen generate the Verilog code

synth synthesize for an FPGA

TODO: Test and describe the ELF download flow.

The `Makefile` use following variables to configure the build process: `BOOTAPP` is an application that ends in the on-chip ROM. This may be an assembler program or a simple C program that does not need any initialized data; most prominent the boot loader for ELF binaries. A C program that shall be compiled as ROM target needs to be prefixed with `bootable-`. `APP` is a C program resulting in an ELF binary that can be either loaded by the emulator or the boot loader when executing in an FPGA.

Here an example of the individual steps to build the blinking LED C hello world (on a different FPGA board):

```
make tools
make BOOTAPP=bootable-echo bootcomp gen
make BOOTAPP=bootable-echo QPROJ=bemicro synth
make QPROJ=bemicro BLASTER_TYPE=Arrow-USB-Blaster config
```

This split of the make commands is for demonstration. It is possible to merge all steps into a single make (on Linux systems) or two steps when using a two operating systems (for compilation and synthesis).

6 The Patmos Compiler

The LLVM compiler has been adapted to emit code for the Patmos ISA and support the special features, e.g., the stack cache [6].

7 Application Binary Interface

7.1 Data Representation

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

7.2 Register Usage Conventions

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

- `r0` is defined to be zero at all times.
- `r1` and `r2` are used to pass the return value on function calls.
- `r3` through `r8` are used to pass arguments on function calls.
- `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.

7.3 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).

7.4 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.

7.5 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.

7.6 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
....
```

8 Potential Extensions

8.1 Exceptions: Interrupts, Faults and Traps

In the following, we use *exception* to denote any kind of "abnormal" transfer of control. *Interrupts* are generated outside of the pipeline by I/O devices. *Faults* are triggered by the pipeline for instructions that cannot be executed as expected (accesses to unmapped memory, undecodable instructions, etc.). *Traps* are willfully generated exceptions, and are used to invoke operating system functions, which might require elevated privileges (see Section 8.10).

An *exception unit* that is mapped to the I/O space is responsible for managing exceptions. It includes the device registers shown in Table 8.1. The general principle of operation is that the exception unit requests the execution of an exception from the pipeline, and the pipeline returns an acknowledges when it starts the execution of the respective exception handler.

Exceptions are injected into the pipeline in the decode stage. The decode stage was chosen to do this because

1. Interrupts must not be triggered inside branch delay slots. The decode stage "knows" which instructions are executed and can easily keep track of branch delay slots.

2. Mechanisms for calls that are located in the execute stage can be shared with exception handling.

All exceptions that have their origin in the pipeline (faults and traps) are handled through signals from the memory stage to the exception unit. The memory stage is responsible for aborting the execution of any unfinished instructions (flushing the pipeline). The exception unit is responsible for demanding the injection of the correct exception from the decode stage.

To be able to serve exceptions generated by the pipeline when an interrupt already has been injected, the acknowledgment from the pipeline to the exception unit is generated in the memory unit. Only at this point it is clear if the interrupt will be served or if the injected instruction has been flushed from the pipeline.

The memory unit furthermore includes a signal to notify the exception unit of returns from exception handlers.

8.1.1 Changes to ISA

In order to support exceptions, it is necessary that the base address of the current function is available *at any time*. Otherwise, it could not be guaranteed that an interrupt handler is able to return to the correct function. Consequently, the `call` and `brcf` functions must store the base address in some hardware register. This makes the return-base information in register `r30` redundant and it could be replaced by an appropriate special return base register `sb` with minimal effort.

Address	Name	Description
`0xf?????00`	`status`	Interrupt-enable flag, possibly privilege bits. Left-shifted on start of exception handler, right-shifted when returning.
`0xf?????04`	`mask`	Mask of enabled interrupts
`0xf?????08`	`pend`	Pending flags for interrupts
`0xf??????0c`	`source`	Number of exception that is about to be served
`0xf?????20`	`vec<0>`	Address of exception handler 0
`0xf?????24`	`vec<7>`	Address of exception handler 1
...	...	...

Table 8.1: Exception unit device registers

Name	Stage	Description
`intr`	to Decode	Request injection of interrupt, must be ignored in delay slot
`exc`	to Decode	Request injection of other exception, cannot be ignored
`excaddr`	to Decode	Address of exception handler to be executed
`exccall`	from Memory	Acknowledge execution of exception handler
`excret`	from Memory	Notify exception unit of return from exception handler

Table 8.2: Exception unit signals to pipeline

The return information for exceptions must be stored in registers that are not used otherwise. Two special registers `sxb` and `sxo` provide the return base and return offset for exceptions.

Depending of on the exact implementation, delayed loads might require another special register to save the contents of the special register `sm` upon exceptions.

For traps, we introduce an instruction `trap <n>`, which triggers exception number *n*. In order to assimilate the handling of faults and traps, the `trap` instruction causes the memory unit to signal an exception to the exception unit, which then requests the injection of an exception from the decode unit. Consequently, traps do not have a delay slot despite altering the control flow.

To signal the return from an exception handler, we introduce an instruction `xret`, which causes the exception unit to update the status word appropriately.

8.1.2 Resuming Execution

Resuming execution after an interrupt is straight-forward. The interrupt handler simply returns to the bundle that was replaced by the injected interrupt instruction.

Resuming after a fault requires a bit more thought. In principle, execution could resume after the instruction that caused the fault, or retry execution of the instruction. Retrying execution would be suitable for faults like page faults, but would be inappropriate for faults like an accesses to an address that is not mapped to any memory or I/O device or undecodable instructions.

The issue is further complicated by instructions in the other slot of a bundle, whose effects must be made visible when resuming execution. Consider the scenario that the instruction in the first slot is faulty. The instruction in the second slot may use a register that is modified by the instruction in the first slot. Emulating the first instruction and simply resuming with the second instruction is therefore not correct. If the instruction in the first slot uses a register that is modified be the instruction in the second slot, letting the instruction in the second slot continue to the write back stage before executing the exception handle would incorrect.

As a consequence, resuming execution must either fix the cause of the fault and reexecute the whole bundle again, or emulate the effects of the whole bundle (including the triggering of further faults) and continue execution after the bundle. Due to the complexity of the second option, we consider faults where the respective instruction cannot be reexecuted *fatal*, and advise developers to terminate execution instead of trying to resume. For faults, the return address points to the bundle that triggered the fault.

Resuming after a trap in principle has to take into account the same considerations as faults. However, the content of the bundle is under the control of the compiler. We require that a trap instruction is the only instruction in a bundle. The return address for traps points to the bundle after the one that contains the trap instruction.

8.1.3 Older Comments

also delayed exceptions for speculative execution

- Enable/disable interrupts
- Return from interrupt
- Interrupt vector tables or something along these lines
- What happens to ongoing memory accesses through the method cache, stack cache or decoupled loads?

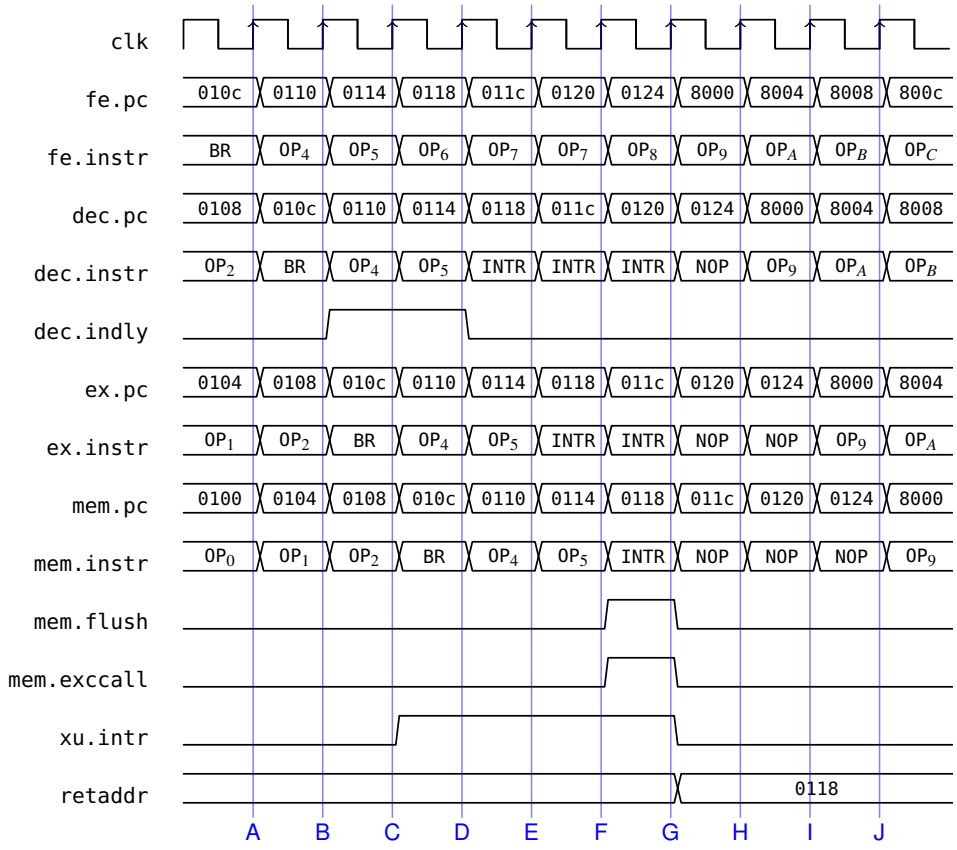


Figure 8.1: Interrupt, delayed by branch delay slot

8.1.4 Timing Diagrams

Interrupt

Figure 8.1 shows the triggering of an interrupt.

A) The decode stage decodes a branch instruction `BR`.

B) As the previous instruction was a branch and the current bundle is part of a branch delay slot, the signal `dec.indly` is high.

C) The exception unit requests an interrupt by asserting `xu.intr`. As `dec.indly` is still high, the request is ignored.

D) The signal `xu.intr` is still asserted, but the current instruction is not part of a branch-delay slot, so the decode stage generates an interrupt instruction `INTR`.

E) The interrupt has not been acknowledged, so signal `xu.intr` is still asserted and the decode stage generates another interrupt instruction. The previous interrupt instruction reaches the execute stage.

F) The interrupt instruction reaches the memory stage, which acknowledges the interrupt and flushes the pipeline.

G) The fetch stage fetches the first instruction of the interrupt handler, $\texttt{OP}_9$.

H) The first instruction of the interrupt handler reaches the decode stage.

I) The first instruction of the interrupt handler reaches the execute stage.

J) The first instruction of the interrupt handler reaches the memory stage.

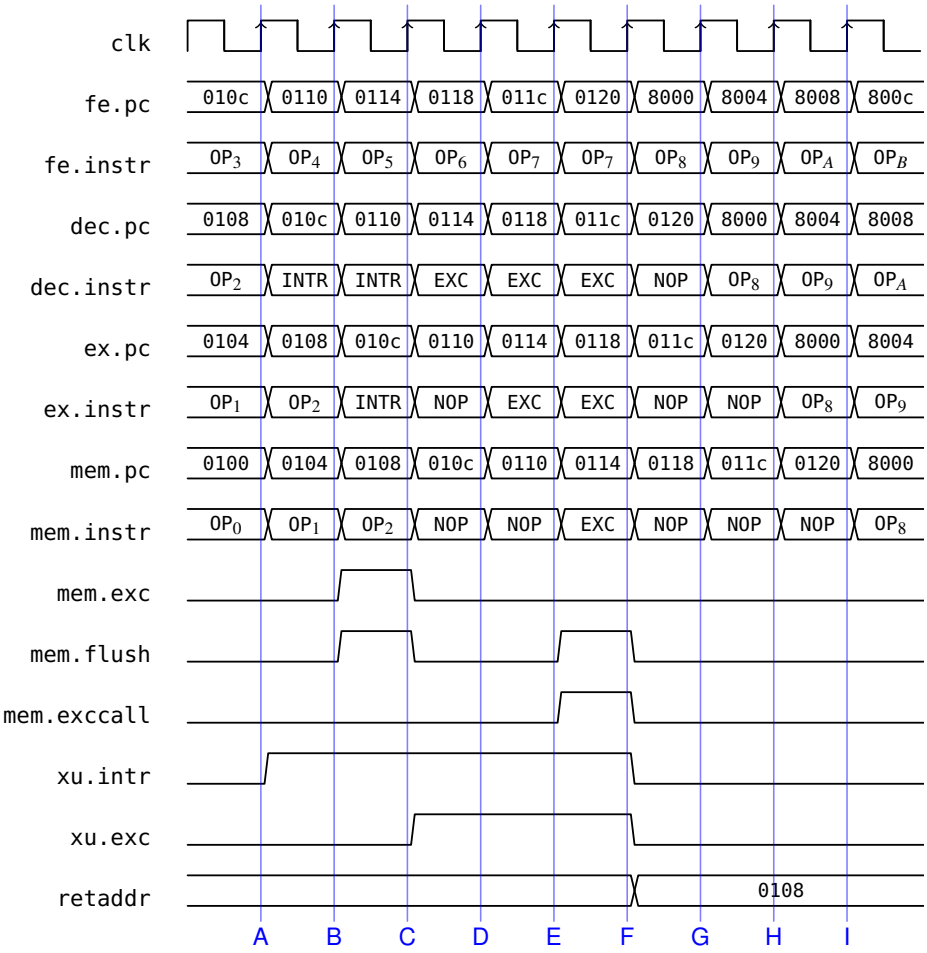


Figure 8.2: Fault while interrupt is requested

Fault

Figure 8.2 shows the triggering of a fault while an interrupt is requested.

A) The exception unit requests an interrupt and the decode stage generates an instruction `INTR`.

B) While the interrupt instruction is in the execute stage, instruction $\texttt{OP}_2$ in the memory stage causes a fault. The memory stage signals this to the exception unit and flushes the pipeline.

C) The exception unit now requests a (non-interrupt) exception and an interrupt. As (non-interrupt) exceptions take precedence, the decode stage generates an instruction `EXC`.

D) Instruction `EXC` reaches the execute stage.

E) Instruction `EXC` reaches the memory stage, which acknowledges the exception and flushes the pipeline again.

F) The fetch stage fetches the first instruction of the exception handler, $\texttt{OP}_8$.

G) The first instruction of the exception handler reaches the decode stage.

H) The first instruction of the exception handler reaches the execute stage.

I) The first instruction of the exception handler reaches the memory stage.

8.2 Decoupled Loads / Multiply / Wait / Move from Special

- Attaching a ready flag with all special registers.
- Specify destination special register with all decoupled operations; the operation sets/resets the ready flag accordingly.
- Wait operates on ready flags of special registers.
- Merged variant of Wait + Move from Special
- Wait with 16-bit mask to wait for multiple outstanding results.

This would be nice since it would allow to reload all special registers from memory without going through the general purpose registers. It would be a unified interface for decoupled operations and give more freedom to handle parallel decoupled operations (pipelined multiplies, loads). We could apply this also to the general purpose registers instead of the special registers.

8.3 Bypass load checks data cache

Let the bypass load use the data cache if the data is cached. If the data is not in the cache, load it from main memory, but do not update the data cache (in contrast to the normal load). Therefore the compiler could use bypass to load data that will not be used a second time or that might have a negative impact on the cache analysis, but we still take advantage of the cache if the data is already in the cache.

8.4 Merged Stack Cache Operations and Function Return

This might require an additional special-purpose register(?) to track the size of the last reserve instruction (this register might also be set explicitly). However, it would might reduce the number of ensure instructions needed.

Another option would be to merge the return and stack free operations. Both instructions belong to the same function and, due to the simpler semantics of the free, the combination would be easier to implement.

8.5 Non-Blocking Stack Control Instructions

Currently, all stack control operations, except `sfree`, are blocking. It might be useful to define non-blocking variants or define them to be non-blocking in all cases.

It is questionable whether this would actually buy us anything. Most `sres` instructions will be followed by a store to the stack cache (spill of saved registers). It might be more profitable for `sens` instructions.

8.6 Purge Cache Content

- dynamic code generation, self-modifying code, etc.
- help cache analysis?

8.7 Freeze Cache Content

- Bypass load can be used to avoid cache updates, but not on per-context basis (we cannot lock the cache and then call any function and assume the function does not update the cache. Instead we would need to generate function variants that only use bypass loads).
- Method cache freeze? Or should we just use a I-SPM for this if we want instruction cache locking?

8.8 Unified Memory Access

Instead of having typed loads per cache or SPM, maybe have types per “use-case scenario”, use local memory based on address

- Type for stack access (guaranteed hit, can be used in both slots)
- Type for guaranteed hit (any local SPM access, access to data cache must be always hit, else undefined result)
- Type for unknown data access (access SPM or data cache, or main memory and update data-cache)
- Type for bypass (access SPM or main memory, do not allocate in data cache)
- Maybe a type for no-allocate (access SPM, data cache or main memory, but do not allocate data in cache; could be useful to prevent single loads from thrashing the cache), or use some sort of cache-lock instruction instead (could be useful to prevent a code sequence or function call from thrashing the cache)

8.9 Memory Management Unit

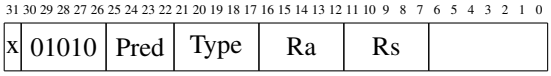
- Simple software managed TLB.
- Main idea is to provide protection and separation.
- Could be used for memory testing.
- No traps. Instead, reset or kill thread?

8.10 Supervisor Mode

- To restrict reconfiguration of critical components (e.g., TLB, Interrupt Tables).
- Transfer to supervisor mode, e.g., via `syscall`.
- Might be handy when running multiple threads or an OS.

8.11 DMA Interface

- Transfers between local and global memory
- Through special registers
- Alternatively, dedicated instructions `dmastart` and `dmalen`



Type	Mnemonic	Operation
00001	dma.sp	Start memory copy which contain the word
xx001	st.sc	Stack cache, no write through to main memory, never wait
xx010	st.sp	Scratchpad, no write through to

8.12 Data scratchpad

- Every core has its D-SPM with its own address range (all in the same address space), a core can write to the D-SPM of another core by writing to an address of the SPM of that core.
- Maybe have some sort of protection mechanism, to prevent cores from writing to any address in any remote SPM
- How do we handle writes to the same address by (remote) dma transfers and local writes (this might prevent local load and stores to the SPM from completing in a single cycle)?

8.13 Halt

8.14 Floating-Point Instructions

8.15 Prefetching

- For method cache
- For local memory

8.16 Data Caches

- Add a second (possibly larger), simple (direct mapped,..) data cache, to be used when the pointer address is known at compile time (i.e., a load does not destroy the whole cache state in the analysis), for array operations, ..
- We would need additional types in the load operation for that cache, but there are only two unused types left. Either use only blocking (?) loads and only word and byte (?) access, or replace some lesser used types, or even introduce a new opcode somehow..

8.17 Instruction scratchpad

- For instruction handlers or other code that should not destroy the method cache
- Could be used to store code that is executed at a call site (even if the method cache entry of the caller gets replaced)
- Replacement of code at runtime, statically scheduled or with some sort of software-controlled replacement strategy (maybe this could be used to prevent threads from destroying the I-cache of real-time tasks)
- Keep frequently used code on the I-SPM, can be used to do some sort of cache locking (instead of somehow locking the method cache).

8.18 Wired-AND/OR for predicates

Let `cmp` be some operation that sets a predicate `Pd` and is predicated by `Pred`. Then we could define the following variants:

```
cond:   if (Pred)           Pd = <cmp>
and:    if (Pred && !<cmp>) Pd = False
or:     if (Pred &&  <cmp>) Pd = True
uncond: Pd = Pred && <cmp>
```

Note that we do not need to read Pd, but the last variant uses Pred as input, not as write-enable signal. First variant is the normal (conditional) execution. The last variant forces Pd to false if Pred is false, thus saving the initialization of Pd or explicit and with Pred for code like

```
if (p1)  p2 = R1 < R2
if (p2)  ...
```

The other variants can be used to implement stuff like

```
if (a != 0 && b < 5) { .. }

p1  = cmpnez r1,    addi r3 = r0 + 5;
p1 &= cmplt r2, r3
```

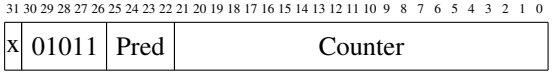
To implement `a != 0 && b > 1` we would need an additional bit that negates either the result of <cmp> or the value that is assigned to Pd (including the true and false assignments).

Note that if we do not need Pred, we can do and and or simply as

```
Pd &= <cmp>  ... if ( Pd)  Pd = <cmp>
Pd |= <cmp>  ... if (!Pd)  Pd = <cmp>
```

i.e., we use Pd as Pred.

8.19 Deadline instruction



Wait until the deadline counter reaches zero, then restart the counter with the given initial value.

9 Conclusion

Patmos is the next cool thing in the dry world of real-time systems.

Bibliography

[1] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: constructing hardware in a scala embedded language. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[3] 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.

[4] 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.

[5] OCP International Partnership. Open Core Protocol specification, release 3.0, 2009.

[6] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, pages 33–40, 2013.

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

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

[9] 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.

[10] M. Schoeberl, B. Huber, and W. Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, 2013.

[11] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.