ASIP Designer

TZSCALE

Processor Manual

L-2016.06

SYNOPSYS®

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
`www.synopsys.com`

Changes

Version	Date	Change
L-2016.06	July 2016	Initial Version of TZSCALE Documentation.
M-2017.03	July 2017	Add On Chip Debugging.

Contents

1 Introduction

The TZSCALE processor model is based on the TZSCALE instruction set architecture (ISA) [3] and is developed using Synopsys ASIP Designer tool. The architectural features of TZSCALE are listed below:

- 32 bit wide data path, with an ALU, shifter. instruction encoding.
- 16 or 32 field (configurable) central register file.
- load/store architecture, which supports 8, 16 and 32 bit memory transfers and an indexed addressing mode.
- 3 stage pipeline.
- single cycle multiplier and multicycle division/remainder unit.

The TZSCALE example processor model is complete, it supports C compilation, cycle accurate instruction set simulation and RTL generation. The C compiler comes with a C runtime library, emulation of 64 bit integer operations, a floating point emulation library and a math library. LLVM compiler support has also been incorporated with in the TZSCALE processor.

2 Register structure

TZSCALE has a 16 or 32 field (configurable) central register file, from which all instructions read operands and to which they write back their results. The register file has two read ports `r1` and `r2` and one write port `w1`.

Some register fields have a specific use:

- Any register ranging from `X1..Xn` (where n is the number of register fields in the architecture i.e. 16 or 32), can be used to read and write an operand value.
- Reading Register `X0` results in reading a constant value 0 and writing a value to this register field does not change its value. The nML code of the mode rule for an operand register is shown below:

```
trn div_wad<uint5>;

mode mR1(mR1_regs | mR1_dead);

mode mR1_regs(r: c5u) {
    read_value   : R[r] read(r1);
    write_value  : R[div_wad=r] write(w1);
    syntax       : "x" dec r;
    image        : r;
}

mode mR1_dead() {
    read_value   : r1;
    write_value  : w1;
    read_action  { r1 = 0;}
    write_action { w1_dead = w1;}
    syntax       : "x0";
    image        : "00000";
}
```

  An additional mode rule `mR1_dead` is created to ensure that reading and writing to register `R0` results in reading a 0 value and no value being written respectively. The w1_dead is a dead end transitory. More details on the dead end transitory can be found in the Chess modeling manual section 3.2.
- Register `X1` is used to save the return address before a subroutine call. It has the alias name `LR` (link register).
- Register `X14` is reserved for the stack pointer, with alias name `SP`.

Next to the central register file, there is a program counter `PC`.

3 Instruction formats

TZSCALE is built with the RV32I integer instruction set. Figure 3.1 shows the instruction set format for the RV32I TZSCALE. As the purpose of this document is to explain the TZSCALE architecture as modeled with the Synopsys ASIP Designer tool, so we shall not go into details of the ISA and for more details on this topic please refer to [3].

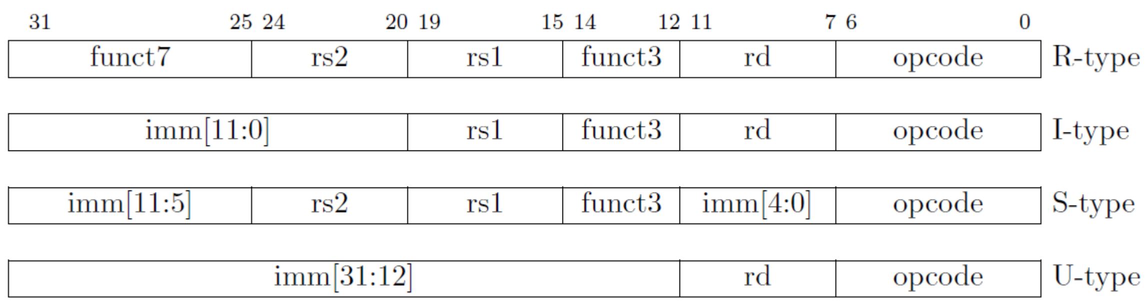

Figure 3.1: The TZSCALE Base instruction formats.

Note that the instruction encoding of the ASIP Designer model deviates from that of figure 3.1. Take as example the R-type instructions. This type can be modelled as follows in nML:

```
image  : func7::rs2::rs1::func3::rd::opcode;
```

The program memory `PM` is declared as a record alias of the byte memory `PMb`. In TZSCALE, it is specified that the memory is little endian. The `opcode` bits are numbered $6\ldots0$ and are located in the first byte of the instruction. As ASIP Designer only supports big endian program memory, the model deviates from the TZSCALE specification. In our model the opcode bits are numbered 25..31 and are stored in the fourth byte of the instruction.

A further complication arizes when we want to add the 16 bit compressed instructions. The 16 bit instructions are distinguised from 32 bit instructions through the `opcode` field.

	xxxxaaxxxxxxxxxx	16 bit, aane11
xxxxxxxxxxxxxxxx	xxxxxxxxxxxxxx11	32 bit

In ASIP Designer, there is a rule which states that the opcode bits differentiating between single word and multi-word instructions should be present in the most-significant instruction word. The solution we have chosen is to reverse the fields, as follows:

```
image  : opcode::rd::func3::rs1::rs2::funct7;
```

4 Pipeline

The TZSCALE ISA does not define the pipeline architecture. We modeled TZSCALE as a 3 stage pipeline, which is similar to the z-scale [3] and has three stages namely IF-DE-WB. Compared to a 5 stage RISC architecture, in TZSCALE the DE, EX are combined into a single stage, an ME stage is missing and memory load results are available on the bus in the WB stage.

Fetch stage (IF)

- A new instruction is fetched from program memory and is issued.

Decode/Execute stage (DE)

- The instruction is decoded and the operands are read from the register file. The target address is sent to program memory.
- This is the stage in which the ALU and shifter units execute their operation.
- The multiply unit executes in this stage.
- The multi-cycle iterative division is started in this stage and can take variable number of cycles to finish.
- For memory load operations, the effective address is computed and is sent to the memory. For store operations both address and data are sent to the memory. The load or store operation is started.
- The unconditional jump instruction executes in this stage:
- The conditional branch instructions execute in this stage.

Write back stage (WB)

- The result of memory load operations is available on the data bus.
- The result of ALU, shift, multiply, control and load operations is written to the destination field on the register file.

5 Data path

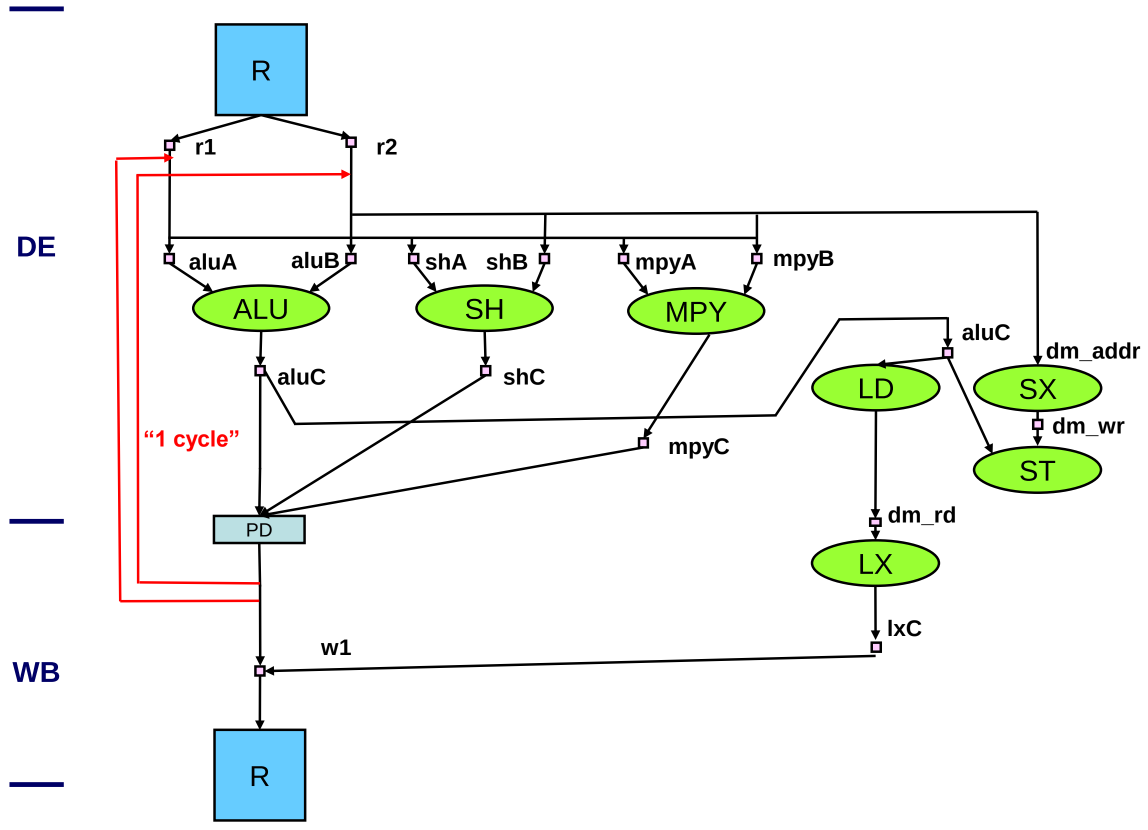

Figure 5.1: The TZSCALE pipelined data path.

5.1 Overview

Figure 5.1 shows the data path of the TZSCALE processor, with the names of the main transitories and pipeline registers.

The TZSCALE implementation has 3 stages namely IF-DE-WB. In the `DE` stage the operands are read from `R`, using ports `r1` and `r2`, the read ports `r1` and `r2` are connected to the inputs of the functional units, `aluA` and `aluB`, `shA` and `shB`, `mpyA` and `dm_wr`; depending on the instruction that is being executed. Operands can also be sent to the inputs `divA` and `divB` of the iterative divider, but these are not shown in figure 5.1.

The ALU, shifter and multiplier produce results at the end of `DE`. Their results are then stored in `PD` before being written to the register file in the `WB` stage.

The multiplier has a single cycle latency and hence it behaves in the same way as the `ALU` and the `SH` units.

The result of a load is also available in `WB`. It goes through a sign/zero extension unit LX, and is written to the register file.

In case of a store operation the relevant part of the data is extracted in the SX unit.

For both load and store operations, the effective address is computed on the ALU. The output `aluC` is copied to the address bus `dm_addr`.

5.2 Hazards

Operands are fetched from the register file in stage (`DE`) and written back one stage later (in `WB`). Consider a pair of data dependent instructions A and B: A produces a result in `X3`, and B uses this result (see figure 5.2). The B instruction can only read its operand if it is scheduled as the second instruction after A, as shown in figure 5.2. Therefore one independent instructions (possibly NOP instructions) must be scheduled between A and B.

			R[#] = PD	
A	IF	DE	WB	
B		IF	DE	WB
			... = PD	

Figure 5.2: (a) schedule for the 1 cycle bypass.

5.2.1 Bypasses

By adding register bypasses, it becomes possible to schedule A and B closer to each other. Register bypasses are connections that are added to the data path so that a result can be fed back to the operand pipeline registers as soon as it is available. When the A instruction is in the `WB` stage, the result is present on the `PD` pipe. It can be used by the B instruction in the same cycle. To add a bypass from `PD` to `r1` or `r2`, the following bypass rules are specified.

```
bypass 1 cycles () {
    stage WB: R[#] = #PD;
} -> {
    stage ID: #r1 = R[#];
}

bypass 1 cycles () {
    stage WB: R[#] = #PD;
} -> {
    stage ID: #r2 = R[#];
}
```

5.2.2 Stalls

Not all data hazards can be avoided by means of bypasses. The data that is loaded from memory is available only in the `WB` stage.

To deal with the data hazard for the case of a 1 cycle delay, hardware stalls are used. A hardware stall means that the issuing of the dependent instruction is postponed until the hazard is gone. The hardware stall for the load operation is defined as follows:

```
hw_stall 1 cycles () {
    stage WB: R[#] = lxC;
} -> {
    stage DE: ... = R[#];
}
```

The first part specifies the write path from `lxC` to `R`. The second part specifies the read path from `R`. Note that the load-specific transitory `lxC` is included in the path, so that the stall does not apply to other operations. More hardware stalls are described in chapter9.

6

Arithmetic instructions

The TZSCALE architecture contains three arithmetic units: the ALU, the shifter and the multiplier. Please refer to [3] for more details on the TZSCALE ISA.

Generation of constants

The addition with immediate is used to generate constants in the 12 bit range. This is achieved with a chess view, where the A input of the ALU is set to zero (the zero is generated by reading `R0`), so that the constant generated on `aluB` is copied to `aluC`.

```
chess_view() {
    aluC = bor(aluA=0,aluB);
} -> {
    aluC = aluB;
}
```

Register move

The bitwise OR instruction is also used to implement a register move. This is achieved with a chess view, where the B input of the ALU is set to zero.

```
chess_view() {
    aluC = bor(aluA,aluB=0);
} -> {
    aluC = aluA;
}
```

Conversions to short unsigned types

The TZSCALE processor has a 32 bit wide data memory DM, but also supports 8 and 16 bit transfers. When storing an 8 or 16 bit value, the data is extracted from the 32 bit register value before it is stored in the memory. This is modeled with `extract_w08()` and `extract_w16()` conversions on the DM store bus. These conversions can be used to implement an `unsigned char(int)` or `unsigned short(int)` conversion, but require that the result is stored to memory.

When the result must be kept in the register, the bitwise AND with immediate can be used to do the zero extension: Primitive `zero_extend_08()` is mapped using the following chess views:

chess_view() aluC=band(aluA,aluB=0xff); -> aluC=zero_extend_08(aluA); Both the `extract_w08()` and `zero_extend_08()` primitives can then be used to implement the type conversions. For example:

```
promotion operator unsigned char(int)  = { w32 zero_extend_08(w32),
                                           w08 extract_w08(w32) };
```

Shift instructions

The three shift operations: logical shift left, arithmetic shift right and logical shift right; are supported in the three register format (instructions `sll`, `sra` and `srl`); and in the immediate format with an unsigned immediate operand (instructions `slli`, `srai` and `srli`).

Compare instructions

Branch instructions are used in the TZSCALE to perform the comparison operations. Please refer to [3] for more details on the TZSCALE ISA.

Multiplication instructions

The multiplication instruction is not part of the base TZSCALE ISA and is implemented as an extension to the ISA. The basic multiply instruction `mul`, performs a 32x32 bit signed multiplication and stores the 32 bit result in any X filed. Other variants of multiply instructions are : MULH, MULHSU, MULHU. To enable resource sharing, all these instructions are mapped to the same multiplier unit.

Conversions to short signed types

Again, there are two ways to do a conversion to an 8 or 16 bit value. The first way is to store that value to the memory. This is modeled with `extract_w08()` and `extract_w16()` conversions on the DM store bus. These conversions are used to implement a `signed char(int)` or `signed short(int)` conversion, but require that the result is stored to memory. As the TZSCALE ISA does not include a sign extension instruction hence storing and loading back from memory is the only way to perform the mentioned conversion.

Load Upper Immediate instruction

The load upper immediate instruction initializes a 20 bit immediate in the high part of a registers and clears the 12 LSBs of that register. This instruction is used in combination with an OR-immediate instruction to generate a full 32 bit constant. This mechanism is specified via a `chess_routing_const` pattern.

```
namespace tzscale_primitive {
    inline void chess_convert(int20p msb, uint12 lsb, w32& L)
    {
        L = bor(lui(msb),lsb);
    }
};

chess_properties {
    convert_routing_const w32 : R;
}
```

When the LSB part of the constant is zero, the OR instruction is not needed. Such constants can be generated by defining the `lui` primitive as a conversion function with property `promotion_conversion_-alternate`, which does right padding of a dedicated 20 bit signed type.

```
class int20p property(20 bit signed /*padded*/);
...
w32 lui(int20p) property(right_padding_20 promotion_conversion_alternate);
```

7 Load/store instructions

The TZSCALE architecture supports 8, 16 and 32 bit wide load and store operations. In this section, we describe how the data memory and the load/store instructions are modeled. We also present the model of an IO interface, which implements the data memory using 2 x 4 byte banks. The IO interface also implements the unaligned memory access on byte, half words and words level.

7.1 Memory declaration

The data memory is modeled using the explicit declaration style of nML, with an access declaration. The root memory is of type `w08` and is called `DMb`. The root memory has two record aliases of types `w16` and `w32`. There is a common address transitory `dm_addr`, which is used for load and store, and for all data sizes. There are separate data transitories for reading and writing, and these are duplicated for all data sizes.

```
mem DMb [dm_size,1]<w08,addr> access {
    ld_dmb: dmb_rd ‘1‘ = DMb[dm_addr ‘0‘ ] ‘1‘;
    st_dmb: DMb[dm_addr] = dmb_wr;
};

mem DMh [dm_size-1,1]<w16,addr> alias DMb align 1 access {
    ld_dmh: dmh_rd ‘1‘ = DMh[dm_addr ‘0‘] ‘1‘;
    st_dmh: DMh[dm_addr] = dmh_wr;
};

mem DMw [dm_size-3,1]<w32,addr> alias DMb align 1 access {
    ld_dmw: dmw_rd ‘1‘ = DMw[dm_addr ‘0‘] ‘1‘;
    st_dmw: DMw[dm_addr] = dmw_wr;
};
```

Note that as the memory is a single byte aligned, so the DMb and the DMw memory aliases are annotated with the `align 1` keyword. Further implementation of the byte aligned memory accesses are present in the IO interface.

7.2 Load/store instructions

Five different load operations are possible:

- word load (`lw`),
- signed half word load (`lh`),
- unsigned half word load (`lhu`),

- signed byte load (`lb`), and
- unsigned byte load (`lbu`).

For the signed loads, the 8 or 16 bit value is sign extended and then stored in the destination register. For the unsigned loads, the 8 or 16 bit value is zero extended.

Three different store operations are possible: word store (`sw`), half word store (`sh`), and byte store (`sb`). For the 8 and 16 bit stores, the least significant bits of the source register are stored in memory.

7.3 Addressing modes

The load and store operations supports only indexed addressing mode. For the indexed addressing mode, `lw rd,i(rs1)` or `sw rs2,i(rs1)`, the 16 bit signed immediate `i` is added to the content of register `rs1`, to obtain the effective address. The compiler header file contains a property to favor indexed addressing whenever the offset fits in the 12 bit signed range:

```
pointer_index_type      : int12;
```

The indirect addressing mode is obtained by setting the index to zero. This is modeled with a chess view.

```
chess_view () {  // indirect addressing
    aguC = add(aguA,aguB=0);
} -> {
    aguC = aguA;
}
```

The direct addressing mode is obtained by setting the first input operand in the address generation to zero and hence using only the 12 bit signed immediate to generate the memory address. This is modeled with a chess view.

```
chess_view () {  // direct addressing
    aguC = add(aguA=0,aguB);
} -> {
    aguC = aguB;
}
```

The nML memory declaration of section 7.1 results in the following RTL interface.

- An address bus `dm_addr`.
- Three data read busses `dmb_rd`, `dmh_rd` and `dmw_rd` and three three data write busses `dmb_wr`, `dmh_wr` and `dmw_wr`.
- The following load and store enable signals: `ld_dmb`, `ld_dmh`, `ld_dmw`, `st_dmb`, `st_dmh` and `st_dmw`.

The record alias structure is typically implemented by means of two 4 byte wide banks. The glue logic between the above interface and the banks can be implemented outside of the processor, by means of an external RTL module. It is also possible to model this interface logic by means of an IO interface [4]. The IO interface is modelled in the file `tzscale_io.p`, which is incuded in `tzscale.p`.

7.4 IO Interface

In the IO interface, the two 4 byte banks are declared with their own address and data ports. The reason for implementing two 4 byte banks is to support the unaligned byte, half word and word memory accesses.

```
mem DM0[2**(DM_SIZE_NBIT-3),1]<v4uint8,daddr> access {
    ld_dm0: dm0_rd`1` = DM0[dm0_addr]`1`;
    st_dm0: DM0[dm0_addr] = dm0_wr;
};
```

```
mem DM1[2**(DM_SIZE_NBIT-3),1]<v4uint8,daddr> access {
    ld_dm1: dm1_rd`1` = DM1[dm1_addr]`1`;
    st_dm1: DM1[dm1_addr] = dm1_wr;
};
```

The `process_request()` function determines the type of memory request that comes from the processor, and forwards this to the appropriate banks with correct addresses and load/store enable signals.

- The address to the first or the second bank is calculated based on the `dm_addr`.
- The two load enable signals `ld_dm0` , `ld_dm1` and store enable signals `st_dm0`, `st_dm1` are computed, based on the type of load/store instruction and the memory address.
- Depending in the type of store, bit ranges of the word, half word and byte store busses are assigned to the four write busses `dm0_wr` and `dm1_wr`.

The `process_result()` function combines the load data that comes from the two banks.

When the IO interface file `tzscale_io.p` is included in the processor model, the RTL code will have the following memory interface signals:

- two address busses `dm0_addr` and `dm1_addr`;
- two data read busses `dm0_rd` and `dm1_rd` and two data write busses `dm0_wr` and `dm1_wr`;
- two load enable signals `ld_dm0` and `ld_dm1` and two store enable signals `st_dm0` and `st_dm1`.

When the IO interface file is not is included, the earlier RTL interface will be generated.

8

Control flow instructions

TZSCALE supports both conditional branches and unconditional jump instructions. The control flow instructions in the TZSCALE ISA do not have any architecturally visible delay slots.

8.1 Unconditional Branch

The unconditional jump instructions include the jump and link (JAL) and the jump and link indirect instruction (JALR). JAL instruction stores the next instruction address in the `rd` register. JAL instruction with (rd=x0) is used to model a J pseudo instruction. JALR instruction can be used to jump anywhere inside a 32 bit range. JALR instruction with immediate as 0 is used to model pseudo JR instruction which is a indirect register jump with no immediate offset.

```
// A JAL with destination register X0 is a plain jump instruction
chess_view () {
    w1_dead = lnk = jal(of21);
} -> {
    j(of21);
}

chess_view () {
    aluC = add(aluA,aluB = 0) @alu;
    w1_dead = lnk = jalr(trgt=aluC);
} -> {
    jr(trgt=aluA);
}
```

8.2 Conditional Branch

TZSCALE ISA provides conditional branch instructions i.e BEQ, BNE, BLT[U] and BGE[U]. These instructions perform comparison operation on the two source registers and then perform a PC relative branch based on a 12 bit immediate. To enable the TZSCALE processor to perform a conditional branch up to 20 bit range, the Chess compiler is directed to replace a far conditional branch with a complemented conditional branch and then a far jump. The complemented conditional branch is to jump over the next instruction which is a far unconditional jump. The code snippet below gives an example of this transformation.

```
JUMP LT <label>

to be replaced by following sequence:
```

```
JUMP GE 1     // complement the condition to jump over the next far jump instruction
JUMP <label>  // unconditional jump to far label
```

To be able to support the above described behavior the following tasks have been carried out which also include changes in the TZSCALE ISA:

- a new top-level opcode: ctrl_bnch_far (next to ctrl_bnch)
- For Chess compiler view there are two cond-branch instructions: ctrl_bnch doing a short branch and ctrl_bnch_far doing a far jump (concatenation of ctrl_bnch and far jal)
- For the other views, there is only the short branch being enabled for both opcodes (ctrl_bnch | ctrl_bnch_far).

For more details on the implementation of the far conditional branch please refer to the conditional branch nML code present in the control.n file.

9 Division

The TZSCALE processor has multi-cycle division and remainder instructions `div`, `divu`, `rem` and `remu` which are modeled partly in the nML model (in `div.n`), as a multi cycle functional unit (in `tzscale_div.p`), and by means of hazard rules (in `hazards.n`).

- `divs() divu() rems() remu()` primitives with the property `multicycle_32` are declared in the primitive header file.
- The `div rd, rs1, rs2`, `divu rd, rs1, rs2`, `rem rd, rs1, rs2` and `remu rd, rs1, rs2` instructions are modeled in `div.n`. These instructions read the dividend and divider from sources `rs1` and `rs2`. The results are stored in `rd` register.
- Control signals `vd_divs_divA_divB_wl_wh_divs_EX_sig` (in the same way for divu() rems() and remu()) are created and these signals will be asserted when the `divs` (or the corresponding)instruction is decoded.
- The multi cycle functional implements the iterative division algorithm.
- In the first cycle, it is checked if any of the control signal is high. When that is the case, first the numerator is normalized. The resulting operands are copied to local registers `PA` and `B`. Also, the counter is set to the number of division iterations that are still needed after normalization of the numerator.
- In the subsequent cycles, the `div_step` function is called and the counter is decremented.
- In the final cycle (when the counter equals 1), the results (quotient and remainder) are written to the destination register. The destination register index is passed to the MCFU in the first cycle and is stored in a local register and is used in the last cycle to store the divide or remainder result.

A number of hardware stall rules are defined, they can be found in the hazards.n file:

- To stall the pipeline when a subsequent instruction depends on the results of the `div` instruction.

```
 hw_stall trn () {
    trn(div_busy); address_trn(div_addr);
} -> {
    stage DE: ... = R[#];
}
```

- To avoid that a new `div or rem` instruction is issued while a previous instruction that uses the MCFU, is still iterating.

```
 hw_stall trn () {
    trn(div_cnt);
} -> {
    class(div);  // trn(div_new);
}
```

- Stall subsequent instructions that would write to the result registers in the same cycle as div and rem instructions of `div`.

```
hw_stall trn () {
    trn(div_busy); address_trn(div_addr);
} -> {
    stage WB: R[#] = ...;
}
```

- No new instruction should use the same register write port in the same cycle as the divide or remainder instructions.

```
 hw_stall trn () {
    trn(div_wnc);
} -> {
    stage WB: R[] = w1;
}
```

Note that, while an iterative `div` instruction is in progress, subsequent instructions will be issued and will complete, as long as they don't violate any of the hazards.

10

Controller model

The processor control unit (PCU) of TZSCALE is modeled in the file `tzscale_pcu.p`. The model is quite simple, it implements the following functionality.

10.1 In the `user_issue()` function

- Normally, every cycle a new instruction is read from the program memory read port, and issued into the pipeline for execution. Exceptions are: cycles where a hardware stall occurs (when `stall_sig()` is true) or when the issue_sig() function does not return a value 1 or if the processor is in the boot cycle or if a conditional jump is taken.

10.2 In the `user_next_pc()` function

A new program counter value is computed: by assigning the absolute target address `trgt`, by adding the offset `of16` or `of26` to the current `PC`, or by incrementing the `PC` with 4. Exception are:

- in the first cycle after a reset (the booting cycle), the `PC` is kept at zero,
- cycles where there is a hardware stall condition, the `PC` is not incremented.

11

Compiler model

11.1 The compiler header file

The mapping of C language constructs onto TZSCALE is specified in the `tzscale_chess.h` compiler header file.

- The C built in integer types are supported on TZSCALE with the following precision.

`char`, `signed char`, `unsigned char`	8 bit
`signed short`, `unsigned short`	16 bit
`int`, `unsigned`	32 bit
`long`, `unsigned long`	32 bit
`long long`, `unsigned long long`	64 bit

- The 8 and 16 bit types are directly promoted onto primitive types. For example:

```
promotion signed char  : strong  { w08, w16, w32 }
                         transitory { addr }
                         exclude { w32 extend_zero(w08),
                                   w32 extend_zero(w16) };
```

  - The promotion to `w08` is needed to store a `signed char` in the `DMb` memory.
  - The promotion to `w32` is needed to store a `signed char` in the register file.
  - The promotions to `w16` and `addr` are needed to obtain nil conversions.
  - When moving a `signed char` to a wider storage, paths that do a zero extension must be avoided, so these functional conversions are excluded.

- The C built-in operations on `int` and `unsigned` are mostly mapped directly onto primitive functions. The division and modulo operators, are mapped onto the iterative division instruction.
- The C built-in operations on `long long` and `unsigned long long` are implemented using dual precision arithmetic. An (`unsigned`) `long long` variable is represented as a pair of two `unsigned`s. Operations are then executed on these pairs, and are modeled by means of small inline functions. The division and modulo are implemented by means of subroutines.

11.2 Libraries

- The processor library `libtzscale`. This library contains the startup code and the subroutines that implement long division.

  The main purpose of the startup code in `tzscale_init.s` is the initialization of the stack pointer, based on the symbolic start address `_sp_start_value_DMb` that is generated by the linker. This address is a 32 bit value. Two instructions are needed to initialize the stack pointer register `r1`: `lui` is used to initialize the 20 MSBs, `ori` is used to initialize the 12 LSBs. In order to adjust the immediate parameters in the assembly code to the value of the symbolic start address, two relocators must be applied, as is shown below.

```
.text global 0 _main_init
.rela 4 _sp_start_value_DMb 0
        lhi r1, #0
.rela 6 _sp_start_value_DMb 0
        ori r1,r1,#0
        nop
.undef global data _sp_start_value_DMb
```

Note that it is advised to define the required relocators in the user-defined relocator file `tzscale.r`. In order to find out which relocators are needed to relocate the `lui` and `ori` instructions, the assembly code that is generated for the following C function can be used as example.

– Example C function.

```
int symbol = 0;
int* foo() { return &symbol; }
```

– Assembly code generated for this function.

```
        jr r15
.rela 4 symbol 0
        lui r2, #0
.rela 6 symbol 0
        ori r2,r2,#0
.undef global data symbol
```

- The C floating point types `float` and `double` are supported. Floating point operations are emulated in software. For this, the SoftFloat package is used (see `tzscale/lib/tzscale_float.h`, `tzscale/lib/tzscale_double.h` and `tzscale/lib/softfloat`).
- The C run time and math libraries are supported (see `tzscale/lib/runtime`, projects `libc.prx` and `libm.prx`).

For LLVM compiler, a source directory named libs need to be present in the same hierarchy level as the TZSCALE directory. It contains C and C++ libraries for LLVM compiler.

12

Overview of the processor model files

The TZSCALE processor model is located in the `tzscale/lib` directory. It consist of the following files:

Processor model

`tzscale.prx`	The processor project file, contains settings that are common to all projects.
`tzscale.h`	The primitive processor header file, contains declarations for the primitive data types and primitive functions of the TZSCALE core.
`tzscale.p`	The primitives definition file, contains the behavior models for the primitive functions.
`tzscale_div.p`	Definition of the multi cycle divide unit.
`tzscale_io.p`	Definition of the IO interface unit.
`tzscale.n`	The top level nML file, contains declarations for the storage elements of the processor, as well as the the top level instruction rules.
`regfile.n`	Contains the nML model of the register file.
`opcode.n`	Contains the nML code for the instruction op codes.
`alu.n`	Contains the nML model of the ALU instructions.
`control.n`	Contains the nML model of the control instructions.
`ldst.n`	Contains the nML model of the load and store instructions.
`div.n`	Contains the nML model of the instruction that support integer division.
`hazard.n`	Contains the hazard rules.
`tzscale_pcu.p`	Contains the behavioral model of the program control unit.

Compiler model

`tzscale_chess.h`	The top level compiler processor header file, specifies the mapping of C built-in types and operators to the primitive processor.
`tzscale_int.h`	Contains the application layer that maps the C built in types `int`, `unsigned` and associated operations on the TZSCALE primitives.
`tzscale_longlong.h`	Contains the application layer that maps the C built in type `long long` and associated operations on the TZSCALE primitives.
`tzscale_float.h`	Contains the application layer that maps the C built in types `float` associated operations. The floating point operations are emulated in software.
`tzscale_const.h`	Contains definitions for constant generation.
`tzscale_bitfield.h`	Contains definitions for bit-field manipulation functions.
`tzscale_mcpy.h`	Contains the definition of the Chess memory copy functions.
`tzscale_long_div.c`	Contains the C functions that implement the iterative division algorithm for `long` and `unsigned long`.
`tzscale_init.s`	Contains initialization code, which is executed before the `main()` function is entered.
`libtzscale.prx`	Project file, used to build the `libtzscale.a` archive, which contains the code of the above mentioned C and assembly files.

Miscellaneous files

`tzscale.l`	Simulator layout file.
`tzscale.r`	Relocator definition file.
`tzscale.bcf`	Default linker configuration file.

13

Some useful projects and commands

Projects to build the processor model

The following projects are available to build the processor model and processor specific software libraries.

1. To build the processor model.

   `lib/tzscale.prx`

2. To build the processor specific software library (this library is required to build the application program).

   `lib/libtzscale.prx`

3. To build the C runtime library (standard functions + hosted-IO).

   `lib/runtime/libc.prx`

4. To build the C math library.

   `lib/runtime/libm.prx`

5. To build the floating point emulation library.

   `lib/softfloat/softfloat.prx`

Project to build the instruction set simulator

The following project is available to build the ISS.

`iss/tzscale_ca.prx`

Project to build the RTL model

The following project is available to generate RTL code.

1. To build the Verilog model.

   `hdl/tzscale_vlog.prx`

Edit the `go_options.cfg` file to change the configuration options of the RTL generator.

Batch project file.

There is a batch project file, located in the `tzscale` directory.

`tzscale/model.prx`

This file allows to build the processor model; the software libraries `libtzscale`, `softfloat`, `libc` and `libm`; the ISSand HDL model.

Commands to build and simulate the sort example program

The following commands are executed in the `sort` directory.

1. To compile the sort program, open `sort.prx` in ChessDE and press the Make button.
2. To simulate the sort program, press the `Start debugging'` button.

Commands to generate test bench data for the RTL model

The following commands are executed in the `sort` directory.

1. Generate input data for the Verilog RTL test bench

   ```
   read_elf -e -G -f hath  -pPMb=8 -mDMb=8 test -o data
   ```

2. Copy the input data to the VHDL RTL directory.

   ```
   cp data.PMb ../hdl/tzscale_[vlog]_go
   cp data.DMb ../hdl/tzscale_[vlog]_go
   ```

On linux the regression suite can be run on TZSCALE with the following command:

```
cd regression;
tct\_tclsh run-test.tcl
```

14

Synthesis and Gate count

The gate count for TZSCALE is shown below with Clock Frequency at 200 and 500MHz and with register file depth of 16 and 32, using the 28nm TSMC synthesis library. In all configurations multiplier unit is non-pipelined. Area of a two input nand gate in the above mentioned library is 0.38475 um^2. The gate count for the resulting 4 configurations are shown below:

The synthesis gate count of the main units of TZSCALE with 32 registers and at 200MHz is shown below.

reg_R	7270	gates	2797	um^2	35.8%
mpy	5697	gates	2192	um^2	28.0%
div	2045	gates	787	um^2	10.1%
sh	684	gates	263	um^2	3.4%
alu	678	gates	261	um^2	3.3%
decoder	630	gates	242	um^2	3.1%
controller	564	gates	217	um^2	2.8%

The core area is 24802 um^2 and the final gate count is 20327 gates.

The synthesis gate count of the main units of TZSCALE with 32 registers and at 500MHz is shown below.

reg_R	8224	gates	3164	um^2	32.9%
mpy	6508	gates	2504	um^2	26.0%
div	2471	gates	951	um^2	9.9%
alu	1148	gates	442	um^2	4.5%
controller	1120	gates	431	um^2	4.4%
decoder	1057	gates	407	um^2	4.2%
sh	681	gates	262	um^2	2.7%

The core area is 18652 um^2 and the final gate count is 24990 gates.

The synthesis gate count of the main units of TZSCALE with 16 registers and at 200MHz is shown below.

mpy	5549	gates	2135	um^2	33.6%
reg_R	3556	gates	1368	um^2	21.6%
div	2038	gates	784	um^2	12.4%
sh	681	gates	262	um^2	4.1%
decoder	678	gates	261	um^2	4.1%
alu	678	gates	261	um^2	4.1%
controller	567	gates	218	um^2	3.4%

The core area is 22716 um^2 and the final gate count is 16496 gates.

The synthesis gate count of the main units of TZSCALE with 16 registers and at 500MHz is shown below.

```
mpy                       6342  gates   2440  um^2  31.6%
reg_R                     3961  gates   1524  um^2  19.7%
div                       2443  gates    940  um^2  12.2%
alu                       1099  gates    423  um^2   5.5%
controller                1042  gates    401  um^2   5.2%
decoder                   959   gates    369  um^2   4.8%
sh                        681   gates    262  um^2   3.4%
```

The core area is 15948 um^2 and the final gate count is 20090 gates.

15

Compiler Benchmark results and application software examples

TZSCALE model directory contains compiler benchmarks such as Dhrystone and Coremark v1.0 and examples such as JPEG encoder and sort algorithm. The performance numbers for these benchmarks on TZSCALE are:

15.1 Dhrystone

Two versions of Dhrystone benchmark are present in the benchmarks directory inside the model. Dhrystone1 is a single file version and Dhrystone2 is a two file version of Dhrystone benchmark.

15.1.1 Dhrystone1

- Chess compiler front end: 7123 cycles/10 iterations or 0,80 DMIPS/MHz
- LLVM compiler front end: 4056 cycles/10 iterations or 1,40 DMIPS/MHz

15.1.2 Dhrystone2

- Chess compiler front end: 7123 cycles/10 iterations or 0,80 DMIPS/MHz
- LLVM compiler front end: 5163 cycles/10 iterations or 1,10 DMIPS/MHz

15.2 Coremark v1.0

- Chess compiler front end: 1.85 coremarks/MHz.
- LLVM compiler front end: 2.17 coremarks/MHz.

16

On Chip Debugging

The provisioning of a core for on chip debugging (OCD) and the development of a debug client is well explained for the TMICRO exmaple core. The TZSCALE largely follows a same approach. Only some specific points are discussed below.

TZSCALE processor supports 2 kinds of ISAs, one ISA includes the 32 bits instructions and 16 bits compressed instructions, while the other one only has 32 bits the instructions, which controls by a macro `ISA_COMPRESSED` in `tzscale_define.h`.

The OCD supports both ISAs. We will introduce compresssed ISA OCD first, and then introduce the second ISA OCD's difference from the first.

16.1 Extensions to processor model

16.1.1 Extensions to nML model

- Program memory write port

  A write port is defined to allow OCD to access the program memory.

  ```
  mem PMb[pm_size] <uint8,addr> access {};

  mem PM[0..pm_size-4,2] <iword,addr> alias PMb align 2 access  {
      ifetch : pm_rd `1` = PM[pm_addr]`1`;
  #ifdef HAS_OCD
      istore : PM[pm_addr] = pm_wr;
  #endif
  };
  ```

- Software breakpoint support

  The TZSCALE on chip debugging solution provides two types of breakpoints: hardware breakpoints and software breakpoints.

  The hardware breakpoint logic is generated automatically by the GO tool when the on chip debugging option is specified. By default, GO creates 4 hardware breakpoint registers. Hardware breakpoints can be specified for instructions that are stored in ROM as well as for RAM based code.

  In order to support software breakpoints, a software break instruction must be added to the instruction set. For TZSCALE, this instruction is defined with the following nML rule.

  ```
  trn ocd_swbreak<uint1>;  hw_init ocd_swbreak = 0;

  opn swbrk_instr() {
      action { stage DE: ocd_swbreak = 1; }
  ```

```
    syntax : "swbrk";
    image  : "0000000000010000",cycles(2);
}
```

Software breakpoints is a mechanism in which the debug client replaces a program instruction with a software break instruction. When the core decodes the break instruction, the decoding logic places the core in debug mode. In order to resume normal execution or to single step:

The original program instruction must be placed back into the program memory, in place of the software break instruction; and the PC must be decremented so that it points to the address of the software break instruction. Software breakpoints require the presence of a software-break instruction in the instruction set of the processor.

- ocd_if.n file

  This file is well explained in the TMICRO example core, please find more details there.

```
reg ocd_addr<addr> read(ocd_addr_r) write(ocd_addr_w);
reg ocd_data<w08>;
reg ocd_instr<iword>;
trn ocd_instr_r<iword>;
property unconnected : ocd_instr_r;

trn ocd_ld_DMb<bool>; hw_init ocd_ld_DMb = 0;
trn ocd_st_DMb<bool>; hw_init ocd_st_DMb = 0;

trn ocd_ld_PMb<bool>; hw_init ocd_ld_PMb = 0;
trn ocd_st_PMb<bool>; hw_init ocd_st_PMb = 0;

fu ocd_addr_incr;

trn ocd_swbreak<uint1>;  hw_init ocd_swbreak = 0;

trn ocd_req<uint1>;  hw_init ocd_req = 0;
trn ocd_exe<uint1>;  hw_init ocd_exe = 0;

properties {
    ocd_address_register        : ocd_addr;
    ocd_data_register           : ocd_data;
    ocd_instruction_register    : ocd_instr;
    ocd_request                 : ocd_req;
    ocd_execute_instruction     : ocd_exe;
    ocd_swbreak                 : ocd_swbreak;
}

#if defined(__go__)
opn ocd_if()
{
    action{
     // DMb debug moves
        stage DE..WB:
        guard (ocd_ld_DMb‘DE‘){
        stage DE:
            ocd_addr = ocd_addr_w = incr1(ocd_addr_r=ocd_addr) @ocd_addr_incr;
        stage DE..WB:
            ocd_data‘WB‘ = dmb_rd = DMb[dm_addr‘DE‘=ocd_addr_r‘DE‘]‘WB‘;
        }
        stage DE:
        guard (ocd_st_DMb){
            ocd_addr = ocd_addr_w = incr1(ocd_addr_r=ocd_addr) @ocd_addr_incr;
            DMb[dm_addr=ocd_addr_r] = dmb_wr = ocd_data;
        }

     // PM debug moves
        stage DE..WB:
        guard (ocd_ld_PMb‘DE‘){
        stage DE:
            ocd_addr = ocd_addr_w = incr4(ocd_addr_r=ocd_addr) @ocd_addr_incr;
        stage DE..WB:
            ocd_instr ‘WB‘ = pm_rd‘WB‘ = PM[pm_addr‘DE‘=ocd_addr_r‘DE‘]‘WB‘;
        }
        stage DE:
        guard (ocd_st_PMb){
            ocd_addr = ocd_addr_w = incr4(ocd_addr_r=ocd_addr) @ocd_addr_incr;
            PM[pm_addr=ocd_addr_r] = pm_wr = ocd_instr;
        }
    }
}
#endif
```

16.1.2 PCU file Modification

- Instruction Issue Function

  Add the debug_mode definition, and the ocd_exe branch.

```
debug_mode = ocd_req && interruptible;
bool no_issue = reg_booting || debug_mode || reg_debug_mode;

if (ocd_exe) {
    ocd_instr_r = ocd_instr;
    issue_instr(0,0,ocd_instr_r[7:0],ocd_instr_r[15:8],ocd_instr_r[23:16],ocd_instr_r[31:24]);
}
else
if (issue_ins && !no_issue) {
    iword ii =  pm_rd;

    //32 bit instructions have 2 MSBs set to "11", step size is 4 bytes for 32 bit instructions
    //and 2 bytes for 16 bit instructions.

    pc_step = (ii[1:0] == 3) ? 4 : 2;

    //for 16 bit instructions, issue the second half as zero.
    if( ii[1:0] != 3) { issue_instr(pcr,1,ii[7:0],ii[15:8],0,        0        ); }
    else              { issue_instr(pcr,1,ii[7:0],ii[15:8],ii[23:16],ii[31:24]); }
}
else {
    pc_step = 0;
}
```

- Next PC Function

  Give jump instructions higher priority than debug_mode and reg_debug_mode to get the right next_pc when jump instructions and debug signals are high at the same time.

```
if (!hw_stall_sig()) {
    if (jump_of21_cd_sig)
        next_pc = pcr + of21_cd;
    else if (jump_trgt_sig)
        next_pc = trgt;
    else if (jump_of21_sig)
```

```
                next_pc = pcr + of21;
            else if (new_fetch && !(reg_booting || reg_debug_mode))
                next_pc = pcr + pc_step;
        }
```

16.2 The debug client

16.2.1 The CHECKERS interface class

Because the shortest instructions are 16 bits, the related functions need to use the 16 bits as the basic operations.

```
pdc_commands::instr_type tzscale_pdc_interface::get_instruction_at(unsigned pc)
{
    long long instr0, instr1;
    get_memory_at(cmd.pmem_name,pc+0,instr0);
    get_memory_at(cmd.pmem_name,pc+1,instr1);
    long long instr = (instr1 <<8) | instr0;
    return instr;
}

void tzscale_pdc_interface::put_instruction_at(unsigned pc,
                                           pdc_commands::instr_type instr)
{
    put_memory_at(cmd.pmem_name,pc+0,(instr >>0) & 0xff);
    put_memory_at(cmd.pmem_name,pc+1,(instr >>8) & 0xff);
}

unsigned tzscale_pdc_interface::next_instruction(unsigned pc) { return pc+2; }
unsigned tzscale_pdc_interface::prev_instruction(unsigned pc) { return pc-2; }
```

16.2.2 The PDC interface class

In TZSCALE, when we try to get a register value, we need to move it to DMb[0..3] first. TZSCALE has the property "little endian". Then we should ensure to put the register value to the DMb with little endian.

```
int pdc_commands::put_register(string name, long long value)
{
    if (!reg_put_code.count(name))
        return Checkers_debugger::failed;

    if (verb) cerr << "PDC put_register: " << name << ", " << value << '\n';
    // put value to memory (in 4 consecutive bytes)
    unsigned val = value;
    for (int i = 0; i < 4; i++) {  // LITTLE endian
        unsigned val = value >> i*8;
        put_memory_at(reg_access_via_mem.name,
                      reg_access_via_mem.addr+i,
                      val & 0xff);
    }
    // load register move instruction in instruction register
    cbl.add_dwrite(ocd_cable::inst,reg_put_code[name]);
    // execute move instruction (memory load)
    cbl.add_iwrite(execute_instr);
    return Checkers_debugger::ok;
}
```

```
void pdc_commands::received_next(string, pdc_caches::range,
                                 std::vector<long long>& vals)
{
    long long val = received_next();
    val &= 0xffffffffULL;
    vals[0] = (val >>  0) & 0xff;  // LITTLE endian
    vals[1] = (val >>  8) & 0xff;
    vals[2] = (val >> 16) & 0xff;
    vals[3] = (val >> 24) & 0xff;
}

int pdc_commands::put_memory_at(string nm, unsigned addr, pdc_caches::range,
                                const std::vector<long long>& vals)
{
    long long val = 0;
    val |= (vals[0] & 0xff) <<  0; // LITTLE endian
    val |= (vals[1] & 0xff) <<  8;
    val |= (vals[2] & 0xff) << 16;
    val |= (vals[3] & 0xff) << 24;
    return put_memory_at(nm,addr*4,val);
}
inline long long pdc_commands::received_next_reg(string)
//  a register read is done by doing 4 consecutive byte memory loads
{
    long long v = 0;
    for (int i = 0; i < 4; i++)   // LITTLE endian
        v |= (unsigned long long)received_next() << (i*8);
    return v;
}
```

16.3 The second ISA OCD's difference

16.3.1 sw_break instruction

In the second ISA, all the instructions are 32 bits, then sw_break instruction should be 32 bits too.

```
opn swbrk_instr() {
    action { stage DE: ocd_swbreak = 1; }
    syntax : "swbrk";
    image  : "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0",cycles(2);
}
```

16.3.2 The PCU file

All the instructions are 32 bits, then when issue the instruction, we set the pc_step always to 4;

```
    debug_mode = ocd_req && interruptible;
    bool no_issue = reg_booting || debug_mode || reg_debug_mode;

    if (ocd_exe) {
        ocd_instr_r = ocd_instr;
        issue_instr(0,0,ocd_instr_r[7:0],ocd_instr_r[15:8],ocd_instr_r[23:16],ocd_instr_r[31:24]);
    }
    else
    if (issue_ins && !no_issue) {
        iword ii =  pm_rd;
        pc_step = 4;
        issue_instr(pcr,1,ii[7:0],ii[15:8],ii[23:16],ii[31:24]);
    }
    else {
        pc_step = 0;
    }
```

16.3.3 The debug client

Because all the instructions are 32 bits, the related functions need to use the 32 bits as the basic operations.

```
pdc_commands::instr_type tzscale_pdc_interface::get_instruction_at(unsigned pc)
{
    long long instr0, instr1, instr2, instr3;
    get_memory_at(cmd.pmem_name,pc+0,instr0);
    get_memory_at(cmd.pmem_name,pc+1,instr1);
    get_memory_at(cmd.pmem_name,pc+2,instr2);
    get_memory_at(cmd.pmem_name,pc+3,instr3);
    long long instr = (instr3 <<24) | (instr2 <<16) | (instr1 <<8) | instr0;
    return instr;
}

void tzscale_pdc_interface::put_instruction_at(unsigned pc,
                                           pdc_commands::instr_type instr)
{
    put_memory_at(cmd.pmem_name,pc+0,(instr >> 0) & 0xff);
    put_memory_at(cmd.pmem_name,pc+1,(instr >> 8) & 0xff);
    put_memory_at(cmd.pmem_name,pc+2,(instr >>16) & 0xff);
    put_memory_at(cmd.pmem_name,pc+3,(instr >>24) & 0xff);
}

unsigned tzscale_pdc_interface::next_instruction(unsigned pc) { return pc+4; }
unsigned tzscale_pdc_interface::prev_instruction(unsigned pc) { return pc-4; }
```

Bibliography

[1] *Chess Compiler User manual*. Synopsys, September 2017. Version N-2017.09.

[2] *The nML Processor Description Language*. Synopsys, September 2017. Version N-2017.09.

[3] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Andrew Waterman and Krste Asanovic, 2017. Document Version 2.2.

[4] *Primitives Definition and Generation manual*. Synopsys, September 2017. Version N-2017.09.