ASIP Designer

Trv (RISC-V ISA) Models

Processor Manual

Q-2020.03-SP1

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at :
http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

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

Changes

Version	Date	Change
Q-2020.03	March 2020	Initial Version of TRV Documentation.
Q-2020.03-SP1	April 2020	Minor updates.

Contents

1

Introduction

The TRV models are a family of ASIP Designer example processor cores based on the RISC-V ISA [3]. Eight models are part of the example cores collection:

Model	ISA	Description
TRV32P3	RV32IM	3-stage pipeline
TRV32P5	RV32IM	5-stage pipeline
TRV64P3	RV64IM	3-stage pipeline
TRV64P5	RV64IM	5-stage pipeline
TRV32P3X	RV32IMC	3-stage pipeline, hardware loops, post-modify addressing
TRV32P5X	RV32IMC	5-stage pipeline, hardware loops, post-modify addressing
TRV64P3X	RV64IMC	3-stage pipeline, hardware loops, post-modify addressing
TRV64P5X	RV64IMC	5-stage pipeline, hardware loops, post-modify addressing

RV32 is the 32-bit RISC-V ISA. The TRV32 models have a 32-bit wide data path.

RV64 is the 64-bit RISC-V ISA. The TRV64 models have a 64-bit wide data path.

RVXXI is the basic instruction set. M adds multiplication, division and remainder instructions. C is the ISA extension for compressed instructions.

There are base and extended models. The extended models, ending with X, feature non-standard extensions. These have hardware support for two nested levels of zero-overhead loops. Their address generation unit supports post-modify addressing.

The P3 models have a 3-stage pipeline. The pipeline stages are fetch (IF), decode (ID), and execute (EX). Results are committed in the EX stage. Instructions can be stalled in the IF and the ID stage.

The P5 models have a 5-stage pipeline. The pipeline stages are fetch (IF), decode (ID), execute (EX), memory access (ME) and write-back (WB). The 5-stage models have a pipeline 2-stage multiplier. Results are committed in the WB stages. Instructions can be stalled in the IF and the ID stage.

All models are in-order single-issue. They have one ALU that implements integer arithmetic, bit-wise logical operations, shifts and comparisons. There is one multiplier. The load-store unit supports byte, half-word, word, and double-word (for RV64) memory accesses. All models have an out-of-pipe multi-cycle divider unit.

The central register file contains 32 fields, of which one field is hardwired to zero. The P3 models use three read ports and one write port. The P5 models use two read ports and one write port. The P3X models use three read ports and two write ports. The P5X models use two read ports and two write ports.

Both the instruction and data memory port use distinct address spaces. Both ports support only aligned addressing.

The models attach IO interfaces to both instruction and data ports, which translate byte addresses to word addresses. The data memory interface has a write-back buffer, eliminating dead bus cycles between reads and writes. The data memory interface uses a byte write mask.

The processor control unit uses a fetch buffer of 8 or 10 bytes. Jumps to misaligned instruction addresses cause a one-cycle buffer stall.

All TRV example processor models are complete. The models support C and C++ compilation. The models include cycle-accurate (CA) and instruction-accurate (IA) instruction-set simulators (ISS). The models have examples for *SystemC* integration. The compiler comes with a C runtime library, a floating-point emulation library. and a math library. For the 32-bit models, there is a library for the emulation fo 64-bit integer operations. Both ASIP Designer compiler front-ends (CHESS and LLVM) are fully supported. The LLVM front-end supports C++ compilation and provides a light-weight C++ standard library.

2

Overview of the Processor Model Files

The processor models are located in the `lib` directories. The TRVXXPY part of filenames reflects the model name: TRV32P3, TRV32P3X, TRV32P5, TRV32P5X, TRV64P3, TRV64P3X, TRV64P5, and TRV64P5X.

The models consist of the following files:

Processor Model	
`trvXXpY.prx`	The processor project file, contains settings that are common to all projects.
`trvXXpY.h`	The primitive processor header file, contains declarations for the primitive data types and primitive functions of the cores.
`trvXXpY_rvc.h`	Additional primitive processor header file for the compressed instructions. Only present in the extended models.
`trvXXpY_zlp.h`	Additional primitive processor header file for the zero-overhead loops. Only present in the extended models.
`trvXXpY.n`	The top level nML file, contains declarations for the storage elements of the processor, as well as the top level instruction rules.
`trvXXpY_define.h`	Contains preprocessor defines for compile-time configuration options like the memory sizes.
`opc.n`	Contains the enumeration types for major and function opcodes.
`alu.n`	Contains the nML model of the ALU instructions: integer arithmetic like addition and subtraction, bit-wise logical operations, shift operations and comparisons.
`mpy.n`	Contains the nML model of multiplier.
`ldst.n`	Contains the nML model of the load-store instructions.
`ctrl.n`	Contains the nML model of the control instructions.
`div.n`	Contains the nML model of the division and remainder instructions.
`rvc.n`	Contains the nML model of the compressed instructions. Only present in the extended models.

`zlp.n`	Contains the nML model of the zero-overhead loop instructions. Only present in the extended models.
`hzrd.n`	Contains the hazard rules for hardware stalls and bypasses.
`cv.n`	Contains the chess-view rules that provide alternative views on parts of the data path to the compiler.
`trvXXpY.p`	Contains the behavioral models of the primitive functions.
`div.p`	Contains the behavioral model of the out-of-pipe multi-cycle divider unit.
`dm.p`	Contains the behavioral model of the data memory interface.
`pm.p`	Contains the behavioral model of the program memory interface.
`trvXXpY_pcu.p`	Contains the behavioral model of the processor control unit (PCU).

Compiler Model

`trvXXpY_chess.h`	The top level compiler processor header file, specifies the mapping of C built-in types, defines the ABI and sets compiler-related processor properties.
`trvXXpY_chess_rvc.h`	Additional compiler processor header file for the compressed instructions. Only present in the extended models.
`trvXXpY_chess_zlp.h`	Additional compiler processor header file for the zero-overhead loops. Only present in the extended models.
`trvXXpY_chess_agu.h`	Additional compiler processor header file for the extended AGU, to setup post-modify addressing. Only present in the extended models.
`trvXXpY_int.h`	Contains the application layer that maps the C built-in integer types and associated operations on the primitives.
`trvXXpY_setcmp.h`	Defines set-compare intrinsics mapping onto the respective RISC-V ISA instructions.
`trvXXpY_const.h`	Defines patterns for constructing complex constants with multiple instructions.
`trvXXpY_chess_mem.h`	Contains the chess_memory_copy and chess_memory_set definitions.
`trvXXpY_bitfield.h`	Contains definitions for bit-field manipulation functions.
`trvXXpY_softfloat.h`	Filtered version of the main header file of the SOFTFLOAT library, which is used for the floating-point emulation by the Chess compiler front-end.
`trvXXpY_float.h`	Contains the application layer for the emulation of single-precision floating-point types and associated operations.
`trvXXpY_double.h`	Contains the application layer for the emulation of double-precision floating-point types and associated operations.

`trvXXpY_rewrite.h`	Contains Chess rewrite rules.
`trvXXpY.r`	Relocator definition file.
`trvXXpY.bcf`	Default linker configuration file.
`trvXXpY_llvm.h`	Main header file for the LLVM frontend.
`trvXXpY_native.h`	Header file for native compilation.

Processor Support Libraries

`libtrvXXpY.prx`	Project file, used to build the `libtrvXXpY.a` archive, providing processor-specific functions like boot code, startup handler and called (no-inlined) emulation functions (if any).
`trvXXpY_init.s`	Contains the startup assembly code that initializes the stack pointer.
`trvXXpY_basic.c`	Contains the driver that calls the main application function.
`trvXXpY_div_longlong.c`	Contains the C functions that implement the iterative division algorithm for `long long` and `unsigned long long`.
`libnative.prx`	Project file, used to build the `libnative.a` archive, providing implementations of processor-specific functions for native compilation.
`compiler-rt/compiler_rt.prx`	Project file, used to build the `libcompiler_rt.a` archive, providing an emulation layer used by the LLVM front-end. The LLVM front-end uses this library among others for the emulation of floating-point types and their associated operators.

Lightweight Stack

`runtime/libc.prx`	Project file for a lightweight default C library (see [1]).
`runtime/libm.prx`	Project file for a lightweight math library (see [1]).
`libcxx-lite/libcxx-lite.prx`	Project file for a lightweight C++ standard library (see [1]).

Instruction-Set Simulators

`iss/trvXXpY_*.prx`	Project files to build instruction-set simulators for the processor models, in various configurations.
`*_ca_dbg.prx`	Cycle-accurate ISS with debug and profiling options enabled.
`*_ca_fst_nojit.prx`	Fast CA ISS with most debug and profiling options disabled.

`*_ca_fst_jit.prx`	Fast CA ISS with most debug and profiling options disabled. Uses just-in-time compilation.
`*_ni_dbg.prx`	Cycle-accurate ISS, without IO interfaces for the instruction and data memory ports, with debug and profiling options enabled.
`*_ni_fst_nojit.prx`	Fast CA ISS, without IO interfaces, with most debug and profiling options disabled.
`*_ni_fst_jit.prx`	Fast CA ISS, without IO interfaces, with most debug and profiling options disabled. Uses just-in-time compilation.
`*_ia_dbg.prx`	Instruction-accurate ISS with debug and profiling options enabled.
`*_ia_fst_nojit.prx`	Fast IA ISS with most debug and profiling options disabled.
`*_ia_fst_jit.prx`	Fast IA ISS with most debug and profiling options disabled. Uses instruction based just-in-time compilation.
`*_ia_fst_blockjit.prx`	Fast CA ISS with most debug and profiling options disabled. Uses block based just-in-time compilation.
`*_sc_dbg.prx`	Standalone *SystemC* cycle-accurate ISS with debug and profiling options enabled.
`*_sc_ia.prx`	Standalone *SystemC* instruction-accurate ISS with debug and profiling options enabled.
`iss/systemc/*.cpp`	Example models for CA and IA ISS *SystemC* integration.

HDL Generation

`hdl/trvXXpY_*.prx`	Project files to build HDL models.
`*_vlog.prx`	Project file to generate *Verilog* model. Includes register logging in the model for ISS to RTL comparisons.
`*_vlog_noreglog.prx`	Project file to generate *Verilog* model. No register logging.
`*_vlog_isacov.prx`	Project file to generate *Verilog* model with auto-generated SystemVerilog covergroups to sample issued instructions.
`*_vhdl.prx`	Project file to generate *VHDL* model.
`hdl/go_options.cfg`	Configuration file for ASIP DESIGNER RTL generation tool GO.

Miscellaneous

`model.prx`	Batch project file that builds core, ISS, HDL, and runtime libraries for the CHESS front-end.
`model_llvm.prx`	Batch project file that builds core, ISS, HDL, and runtime libraries for the LLVM front-end.

Compile Configurations

`Release`	Default optimizations. Uses CHESS front-end.
`Debug`	No optimizations, applies debug-specific transformations to ease debugging. Uses CHESS front-end.
`Release_LLVM`	Default optimizations. Uses LLVM front-end.
`Debug_LLVM`	No optimizations, applies debug-specific transformations to ease debugging. Uses LLVM front-end.
`Native`	Compiles the application for native execution on the host.

Run/debug Configurations

`default`	Default configuration, suitable for most tasks.
`rcd`	Configures ISS and HDL simulators to generate register change dump files (RCD) for comparing ISS against HDL models. The ISS then runs the `bin/iss_rcd.tcl` script. The HDL simulation uses the `rcd` Makefile target (see generated HDL directory). This `rcd` target executes the `gen_rcd.tcl` script in VCS, which runs the simulation for a specified cycle count.
`rtlsim`	Configures the HDL model to use the `sim` Makefile target, which uses the `gen_rcd_exit.tcl` script in VCS. This runs the HDL simulation until the program counter value matches a specified exit address.
`saif`	Configures the HDL model to use the `saif` Makefile target, which runs `gen_saif.tcl` in VCS. This collects switching activity information for use with Synopsys Design Compiler synthesis.
`dve`	Configures the HDL model to use the `dve` Makefile target, which opens the Synopsys VCS DVE debugger GUI and loads the currently specified application project into the testbench memories.

3

Getting Started

Steps to build the processor model

The following steps build the processor model and processor-specific software libraries.

You can use the top-level batch project file:

- To build the processor model, processor support libraries, C libraries, the cycle-accurate ISS with debug options, the Verilog HDL model and the support library for native compilation. Compile configuration `Release` for CHESS front-end.

  ```
  chessmk model.prx -m
  ```

- To build the processor model, processor support libraries, C libraries, C++ libraries, the cycle-accurate ISS with debug options, the Verilog HDL model and the support library for native compilation. Compile configuration `Release_LLVM` for LLVM front-end.

  ```
  chessmk model_llvm.prx -m
  ```

You can also execute the required steps individually:

1. To build the processor model.

   ```
   chessmk lib/{trvXXpY}.prx
   ```

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

   ```
   chessmk lib/lib{trvXXpY}.prx
   ```

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

   ```
   chessmk lib/runtime/libc.prx
   ```

4. To build the C math library.

   ```
   chessmk lib/runtime/libm.prx
   ```

5. To build the floating-point emulation library.

   ```
   chessmk lib/softfloat/softfloat.prx
   ```

6. To build the cycle-accurate ISS with debug and profiling options enabled.

   ```
   chessmk iss/{trvXXpY}_ca_dbg.prx
   ```

7. To generate the Verilog HDL model, with register logging enabled for HDL-to-ISS comparisons.

   ```
   chessmk hdl/{trvXXpY}_vlog.prx
   ```

8. To compile the HDL model e.g. with Synopsys VCS.

   ```
   chessmk hdl/{trvXXpY}_vlog.prx +e
   ```

Commands to build and simulate the sort example program

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

Using the graphical user interface CHESSDE:

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.

On the command line:

1. To compile the sort program.
   ```
   chessmk sort.prx
   ```
2. To simulate the sort program.
   ```
   chessmk sort.prx -S
   ```

Commands to simulate the HDL model and compare with ISS

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

1. Make sure that the HDL model is already compiled with your HDL simulator.
2. Compile the sort program. Disable the use of hosted IO (printf), since this is an ISS only feature.
   ```
   chessmk sort.prx -DNO_HOSTEDIO
   ```
3. Simulate the sort program in the ISS with run/debug configuration `rcd`, to obtain a cycle count for the application.
   ```
   chessmk sort.prx -DNO_HOSTEDIO -S +C rcd
   ```
4. Simulate the sort program with your HDL simulator, e.g. Synopsys VCS.
   ```
   chessmk sort.prx -DNO_HOSTEDIO -H
   ```
5. Compare the register traces from ISS and HDL simulations.
   ```
   rcd_compare Release/sort.{trvXXpY}_ca_dbg.rcd Release/sort.{trvXXpY}_vlog.rcd
   ```

Running the regression suite

On Linux, you can run the regression suite on the processor model with the following command:

```
cd regression/
./domake diff
```

Renaming the model

There is a bash script `rename_model.sh` in the directory `bin/`. This script first renames all files, then replaces all occurrences of the original core name inside files.

```
./bin/rename_model.sh mynewmodel
```

Comparing base to extended models

You might want to select only some features from the extended models.

1. Rename the extended model, e.g. TRV32P5X to TRV32P5
   ```
   cd trv32p5x/
   ./bin/rename_model.sh trv32p5
   ```
2. Do a text based comparison e.g. using the *meld* tool, or *Eclipse*, or *vimdiff*. Selectively merge changes from the renamed extended model into the base model.

Running a single regression test

The following commands are executed in the directory of the test that you want to run, e.g. `regression/C00_simple_report`.

The following sequence of steps compares the execution of the test on the ISS against its execution on the host and a golden reference file.

1. To compile the test with the default compile configuration
   ```
   make -f ../Makefile.test chess
   ```
2. To run the test on the default ISS
   ```
   make -f ../Makefile.test run
   ```
3. To compile and run the test in `Native` mode, i.e., on the host
   ```
   make -f ../Makefile.test native
   ```
4. To compare the golden reference file `test.ok` to the ISS output `test.mem` and the `Native` output `test.gcc`
   ```
   make -f ../Makefile.test diff
   ```

The `Makefile.test` offers multiple targets:

`native`	Compile and run in native mode
`ok`	Use output from native execution as new golden reference
`chess`	Compile for target core
`run`	Run test program on ISS
`diff`	Compare ISS vs Native vs Reference
`rtlrun`	Run test program on HDL simulator, stop after N cycles (N taken from ISS run)
`rtldiff`	Compare ISS vs HDL
`syscrun`	Run test program on the *SystemC* ISS
`syscdiff`	Compare *SystemC* ISS vs Native vs Reference
`ocdrun`	Run test program via On Chip Debugging interface on HDL simulator OCD is currently not present in the models.
`ocddiff`	Compare OCD vs Native vs Reference OCD is currently not present in the models.
`rtlsim`	Run test program on HDL simulator, stop when exit address is hit
`saifrun`	Run test program on HDL simulator, generate switching activity file
`hex`	Generate HDL testbench memory init files
`rtlgui`	Open VCS DVE debugger GUI, load test program into testbench memories
`rtlfree`	Start a free-running VCS simulation, no program loaded (use to connect via OCD, for example)

`gdbrun`	Like run, but uses GDB interface
`gdbdiff`	Like diff, but uses GDB interface
`gdbsyscrun`	Like syscrun, but uses GDB interface
`gdbsyscdiff`	Like syscdiff, but uses GDB interface
`gdbocdrun`	Like ocdrun, but uses GDB interface OCD is currently not present in the models.
`gdbocddiff`	Like ocddiff, but uses GDB interface OCD is currently not present in the models.
`clean`	Cleanup
`virgin`	Cleanup

The `Makefile.test` offers multiple configuration parameters:

`CFG`	Select compile configuration. Default: `Release`
`ISS`	Select ISS. Default: `{trvXXpY}_ca_dbg`
`HDL`	Select HDL model `{trvXXpY}_${HDL}.prx`. Default: `vlog`
`SCISS`	Select *SystemC* ISS. Default: `{trvXXpY}_sc_dbg`

Example. To run a test in the compile configuration `Release_LLVM` on the instruction-accurate ISS with debug/profiling capabilities:

```
cd regression/C00_simple_report
make -f ../Makefile.test diff CFG=Release_LLVM ISS={trvXXpY}_ia_dbg
```

The top-level `domake` script can pass parameters to `Makefile.test`.

Example. To run all tests in the compile configuration `Release_LLVM` on the instruction-accurate ISS with debug/profiling capabilities:

```
cd regression
./domake 'diff CFG=Release_LLVM ISS={trvXXpY}_ia_dbg'
```

Please note that the target and the arguments are packed inside a single-quoted string. The target needs to come first. There should be only one target, with the following exception. You can place the `clean` target in front of the target, e.g. `./domake 'clean diff [..]'`.

4

Primitive Layer

4.1 Primitive Types

`w{08,16,32,64}`	Main data types (`w64` is only present in 64-bit models). These are used for input and outputs of primitive functions, and for both static (registers, memories) and transitory (buses) storages. The types are signed and have a word length matching their name, e.g., `w08` is a signed 8-bit type.
`addr`	Main address type use for the both the data and program memory. It is an unsigned 32-bit or 64-bit type, for the RV32 and RV64 models respectively.
`iword`	Instruction word type: 32-bit unsigned.
`t[0-9][us]`	Primitive types for transitories and immediates. The word length is part of the name. The last letter indicates signedness: either unsigned or signed. For example, `t12s` is a signed 12-bit type.
`t5unz`	Unsigned 5-bit non-zero primitive type, representing value range 1 to 31. This type is used by the multi-cycle divider unit in `div.p`.
`t{13,21}s_s2`	Signed 13/21-bit step-2 primitive types. The type `t13s_s2` represents the values set {-4096, -4094, ..., 0, 2, ..., 4094}. The type `t21s_s2` represents the values set {-1048576, -1048574, ..., 0, 2, ..., 1048574}. Both types have a step size of two. Due to the step size of two, the LSB is always zero and these types have 12 and 20 effective bits, respectively. They are used for the relative jump offsets (branches and jumps).
`t20s_rp12`	Signed 20-bit primitive type with 12-bit right padding. The conversion from this type to the main word type `w32` or `w64` is annotated with the property `right_padding_12`, which inserts 12 zeroes on the right side. This is used for the `lui` and `auipc` instructions. `w32(t20s_rp12) property(right_padding_12);` This is similar to a signed 32-bit type with step size 4096 and 20 effective bits.
`v[48]u[18]`	These primitive vector types are used in the IO interfaces for the `DM` and `PM` nML memories. They represent vectors of four or eight 1-/8-bit unsigned elements. The IO interfaces use these for the byte write enables and the byte lanes of the memory data buses.

`_s{2,4,16}, _rp12` For the compressed instructions (only present in extended models), there are various extra types with different step sizes, and there is an additional `right_padding_12` (see above) type with limited value range.

4.2 Primitive Functions

`w64 add(w64,w64)` Regular addition. Input/output type is `w32` for RV32 models, `w64` for RV64 models.

`w64 sub(w64,w64)` Regular subtraction. Input/output type is `w32` for RV32 models, `w64` for RV64 models.

`w64 slt(w64,w64)` Set-less-than comparison: `a < b ? 1 : 0` Input/output type is `w32` for RV32 models, `w64` for RV64 models.

`w64 sltu(w64,w64)` Set-less-than-unsigned comparison: `a <u b ? 1 : 0` Input/output type is `w32` for RV32 models, `w64` for RV64 models.

`w64 band(w64,w64)` Bit wise logical AND operation Input/output type is `w32` for RV32 models, `w64` for RV64 models.

`w64 bor(w64,w64)` Bit wise logical OR operation Input/output type is `w32` for RV32 models, `w64` for RV64 models.

`w64 bxor(w64,w64)` Bit wise logical XOR operation Input/output type is `w32` for RV32 models, `w64` for RV64 models.

`w64 sll(w64,w64)` Logical left shift. Input/output type is `w32` for RV32 models, `w64` for RV64 models. For RV32, the shift amount is taken from the five LSBs of the second operand. For RV64, the shift amount is taken from the six LSBs of the second operand. The other bits of the second operand are ignored.

`w64 srl(w64,w64)` Logical right shift. Input/output type is `w32` for RV32 models, `w64` for RV64 models. For RV32, the shift amount is taken from the five LSBs of the second operand. For RV64, the shift amount is taken from the six LSBs of the second operand. The other bits of the second operand are ignored.

`w64 sra(w64,w64)` Arithmetic right shift. Input/output type is `w32` for RV32 models, `w64` for RV64 models. For RV32, the shift amount is taken from the five LSBs of the second operand. For RV64, the shift amount is taken from the six LSBs of the second operand. The other bits of the second operand are ignored.

`void mul_hw(w64,w64,t2u,w64,w64)` Regular multiplication, returns low and high product bits. Input/output type is `w32` for RV32 models, `w64` for RV64 models. The third input, with type `t2u`, controls the signedness of the two input operands.

`w64 divs(w64,w64)` Regular division, returns the quotient. Input operands are treated as signed numbers. Input/output type is `w32` for RV32 models, `w64` for RV64 models. This is a multi-cycle primitive mapped onto the multi-cycle functional unit `div`.

`w64 divu(w64,w64)`	Regular division, returns the quotient. Input operands are treated as unsigned numbers. Input/output type is `w32` for RV32 models, `w64` for RV64 models. This is a multi-cycle primitive mapped onto the multi-cycle functional unit `div`.
`w64 rems(w64,w64)`	Regular division, returns the remainder. Input operands are treated as signed numbers. Input/output type is `w32` for RV32 models, `w64` for RV64 models. This is a multi-cycle primitive mapped onto the multi-cycle functional unit `div`.
`w64 remu(w64,w64)`	Regular division, returns the remainder. Input operands are treated as unsigned numbers. Input/output type is `w32` for RV32 models, `w64` for RV64 models. This is a multi-cycle primitive mapped onto the multi-cycle functional unit `div`.
`bool eq(w64,w64)`	Compare `equal`. Input type is `w32` for RV32 models, `w64` for RV64 models.
`bool ne(w64,w64)`	Compare `not-equal`. Input type is `w32` for RV32 models, `w64` for RV64 models.
`bool lt(w64,w64)`	Compare `less-than`. Input type is `w32` for RV32 models, `w64` for RV64 models.
`bool ge(w64,w64)`	Compare `greater-or-equal`. Input type is `w32` for RV32 models, `w64` for RV64 models.
`bool ltu(w64,w64)`	Compare `less-than`. Input operands are treated as unsigned numbers. Input type is `w32` for RV32 models, `w64` for RV64 models.
`bool ge(w64,w64)`	Compare `greater-or-equal`. Input operands are treated as unsigned numbers. Input type is `w32` for RV32 models, `w64` for RV64 models.
`w64 jal(t21_s2)`	Relative jump-and-link (relative call) control primitive Output type is `w32` for RV32 models, `w64` for RV64 models.
`w64 jalr(w64)`	Absolute jump-and-link (absolute call) control primitive. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
`w64 br(t13s_s2)`	Relative branch (relative conditional jump) control primitive. Output type is `w32` for RV32 models, `w64` for RV64 models.
`w64 [sz]ext(w{08,16,32})`	Zero or sign extension primitives for load results. Output type is `w32` for RV32 models, `w64` for RV64 models. The `w32` variant are Only present in the RV64 models.
`w64 addw(w64,w64)`	32-bit addition. The 32-bit result is sign extended to 64 bits. Only present in the RV64 models.
`w64 subw(w64,w64)`	32-bit subtraction. The 32-bit result is sign extended to 64 bits. Only present in the RV64 models.
`w64 sllw(w64,w64)`	Logical left shift. The first operand is interpreted as unsigned 32-bit number. The shift amount is taken from the five LSBs of the second operand. The other bits of the second operand are ignored. The 32-bit result is sign extended to 64 bits. Only present in the RV64 models.

`w64 srlw(w64,w64)` Logical right shift. The first operand is interpreted as unsigned 32-bit number. The shift amount is taken from the five LSBs of the second operand. The other bits of the second operand are ignored. The 32-bit result is sign extended to 64 bits. Only present in the RV64 models.

`w64 sraw(w64,w64)` Arithmetic right shift. The first operand is interpreted as signed 32-bit number. The shift amount is taken from the five LSBs of the second operand. The other bits of the second operand are ignored. The 32-bit result is sign extended to 64 bits. Only present in the RV64 models.

`w64 divsw(w64,w64)` Regular division, returns the quotient. Input operands are treated as signed 32-bit numbers. The 32-bit result is sign extended to 64 bits. This is a multi-cycle primitive mapped onto the multi-cycle functional unit `div`. Only present in the RV64 models.

`w64 divuw(w64,w64)` Regular division, returns the quotient. Input operands are treated as unsigned 32-bit numbers. The 32-bit result is sign extended to 64 bits. This is a multi-cycle primitive mapped onto the multi-cycle functional unit `div`. Only present in the RV64 models.

`w64 remsw(w64,w64)` Regular division, returns the remainder. Input operands are treated as signed 32-bit numbers. The 32-bit result is sign extended to 64 bits. This is a multi-cycle primitive mapped onto the multi-cycle functional unit `div`. Only present in the RV64 models.

`w64 remuw(w64,w64)` Regular division, returns the remainder. Input operands are treated as unsigned 32-bit numbers. The 32-bit result is sign extended to 64 bits. This is a multi-cycle primitive mapped onto the multi-cycle functional unit `div`. Only present in the RV64 models.

`void hwdo(w64,t13u_s2)` Zero-overhead loop setup instructions (relative doloop). The first operand is the loop count. The second operand is the relative loop end offset. Only present in the extended models.

`w64 mul(w64,w64)` Regular multiplication, returns low product bits. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by a functional mode of the `mul_hw` primitive.

`w64 mulh(w64,w64)` Regular multiplication, returns high product bits. Input operands are treated as signed numbers. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by a functional mode of the `mul_hw` primitive.

`w64 mulhu(w64,w64)` Regular multiplication, returns high product bits. Input operands are treated as unsigned numbers. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by a functional mode of the `mul_hw` primitive.

`w64 mulhsu(w64,w64)` Regular multiplication, returns high product bits. Left input operand is treated as signed number. Right input operand is treated as unsigned number. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by a functional mode of the `mul_hw` primitive.

`w64 mulw(w64,w64)` 32-bit multiplication, return the low product bits. The 32-bit result is sign extended to 64 bits. Only present in the RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by a functional mode of the `mul_hw` primitive.

`w64 seq0(w64)` Set-equal-zero comparison: `a == 0 ? 1 : 0`. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by supplying the immediate value 1 as the second operand to the `sltu` primitive.

`w64 sne0(w64)` Set-not-equal-zero comparison: `a != 0 ? 1 : 0`. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by choosing the `X[0]` field as the first operand to the `slt` primitive.

`w64 zext_08(w64)` Zero extend byte. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by supplying the immediate value `0xFF` as the second operand to the `band` primitive. This is used for in-place zero extensions, e.g. for unsigned char upcasts.

`w64 sext_32(w64)` Sign extend word. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by supplying the immediate value 0 as the second operand to the `addw` primitive. Only present in the RV64 models. This is used for in-place sign extensions, e.g. for int upcasts.

`void nop()` No operation.
This is a `programmers_view` primitive. The corresponding functionality is implemented with the `add` primitive: all operands are zero, the result is discarded (written to `X[0]`).

`w64 j(t21_s2)` Relative jump control primitive Output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by discarding the result of the `jal` primitive (write to `X[0]`);

`w64 jr(w64)` Absolute jump control primitive. Input/output type is `w32` for RV32 models, `w64` for RV64 models.
This is a `programmers_view` primitive. The corresponding functionality is implemented by discarding the result of the `jalr` primitive (write to `X[0]`);

5

Structural Skeleton

5.1 Static Storage

Memories and controllable registers fall in the class of static storage elements (see [2]).

5.1.1 Memories

The TRV models have multiple nML memories, including memory range aliases. There are IO interfaces attached to those ports (in `dm.p` and `pm.p`), which translate the internal ports to two external ports for instruction and data memory access.

The program memory is modeled as two nML memories:

```
mem PMb [pm_size] <t8u,addr> access {};

// if compressed instructions are present
mem PM  [pm_size-2,2] <iword,addr> alias PMb align 2 access {
  ld_pm‘0‘ : pm_rd‘1‘ = PM[pm_addr‘0‘]‘1‘;
};

// otherwise
mem PM  [pm_size,4] <iword,addr> alias PMb access {
  ld_pm‘0‘ : pm_rd‘1‘ = PM[pm_addr‘0‘]‘1‘;
};
```

The nML memory `PMb` is the root program memory. It stores instruction bytes and uses byte addresses, as per RISC-V ISA specification. The memory alias `PM` is used by the processor control unit (PCU) to access one instruction at a time (4 bytes). In the presence of compressed instructions, which are only 2 bytes long, the `PM` alias has an address step of 2 and an alignment of 2 in its parent `PMb`. The PCU however ensures that only 4-aligned fetches are issued.

The data memory is modeled with multiple nML memories, representing byte, half-word, word and double-word accesses.

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

mem DMh [dm_size,2] <w16,addr> alias DMb access {
  ld_dmh‘0‘ : dmh_rd‘1‘          = DMh[dm_addr‘0‘]‘1‘;
  st_dmh‘0‘ : DMh[dm_addr‘0‘]‘1‘ = dmh_wr‘1‘;
};

mem DMw [dm_size,4] <w32,addr> alias DMb access {
  ld_dmw‘0‘ : dmw_rd‘1‘          = DMw[dm_addr‘0‘]‘1‘;
  st_dmw‘0‘ : DMw[dm_addr‘0‘]‘1‘ = dmw_wr‘1‘;
};

// only for 64-bit models
mem DMd [dm_size,8] <w64,addr> alias DMb access {
  ld_dmd‘0‘ : dmd_rd‘1‘          = DMd[dm_addr‘0‘]‘1‘;
  st_dmd‘0‘ : DMd[dm_addr‘0‘]‘1‘ = dmd_wr‘1‘;
};
```

The IO interface `dm_merge` in the file `dm.p` merges the three (or four) differently sized accesses into a single word (or double-word) external memory interface.

Control signals and effective address are sent to the data memory in the first cycle. The memory places the load result on the data bus in the next cycle. The core places store data on the data bus also in the next cycle. With this timing, for both loads and stores, the uses of the address bus are in the same stage, and the uses of the data bus reside in the same stage. Resource conflicts are avoided. There is no delay when switching between reads and writes. The IO interface module `dm_wbb` in the file `dm.p` translates this timing to a timing appropriate for most SRAM hard macros. For store operations, both the address and data are sent to the memory in the same cycle. This module has a write-back buffer to handle the load-after-store hazard.

There is a nML memory range aliases for the data memory, specifying the first 2048 bytes. This address range is directly addressable with load and store instructions. A direct address in the range 0..2047 can be generated by using the RISC-V ISA indexed addressing mode, which has a 12 bit signed offset, and by specifying the hardwired zero register `X[0]` as base register.

```
def dm_stat = 2**11;
mem DMb_stat [dm_stat,1] <w08,addr> alias DMb[0] access {};
mem DMh_stat [dm_stat,2] <w16,addr> alias DMb_stat access {};
mem DMw_stat [dm_stat,4] <w32,addr> alias DMb_stat access {};
// only for 64-bit models
mem DMd_stat [dm_stat,8] <w64,addr> alias DMb_stat access {};
```

These memory range aliases are designated as `small_static_memory` in the compiler header file. The C compiler allocates static small (scalar) objects, e.g. 'static int' variables, to these ranges.

For the 64-bit models, there are additional memory range aliases covering the lower half of the full 64-bit address space. A 32-bit address is sufficient for these ranges. This models a similar behavior like GCC's `medlow` code model.

```
def dm_low = 2**(DM_SIZE_NBIT);
// if DM_SIZE_NBIT > 31, set to dm_low to 2**31

mem DMb_low [dm_low,1] <w08,addr> alias DMb[0] access {};
mem DMh_low [dm_low,2] <w16,addr> alias DMb_low access {};
mem DMw_low [dm_low,4] <w32,addr> alias DMb_low access {};
mem DMd_low [dm_low,8] <w64,addr> alias DMb_low access {};
```

These memory range aliases are designated as `static_memory` in the compiler header file. The C compiler allocates static objects to these ranges.

5.1.2 Central Register File

The TRV models have a 32-field central register file, from which all instructions read operands and to which they write back their results. The register file has two or three read ports and one or two write ports (base versus extended models). The fields are 32-bit or 64-bit wide, for the TRV32 and TRV64 models respectively.

```
enum eX { x0, x1, ..., x31 };
reg X[32] <w64,t5u> syntax (eX)
    read(  r1   // operand 1
           r2   // operand 2
           r3   // for AGU, only present in extended models
       )
    write( w1   // alu, mpy, div, load, ctrl
           w2   // for AGU, only present in extended models
         );
hw_init X = others => 0;
```

Some register fields have a specific use:

1. Field `X[0]` is hardwired to zero. Reading this field always returns zero. Writes to this field are ignored. The register alias `zero` refers to this field.
2. Field `X[1]` is used as link register, to save the return address before a subroutine call. The register alias `LR` refers to this field.
3. Field `X[2]` is used as stack pointer. The register alias `SP` refers to this field.

The special functionality of the field `X[0]` is implemented as follows. The field is marked as reserved in the compiler header file:

```
reserved : x0;
```

The compiler will not use `X[0]` to store variables.

There is a chess-view rule in `cv.n` that provides an alternative view onto reading this field:

```
chess_view() { r1 = zero; }
          -> { r1 = 0;    }
```

The code selection phase of the compiler is then aware that reading `X[0]` is equivalent to writing the value `0` to the read port transitory `r1`.

Writes to the `X[0]` field are modeled as assignments to a dead-end transitory:

```
// writeback to x0 via w1 write port
trn w1_dead <w64>;
property dead_end : w1_dead;

opn alu_rrr_ar_instr(op: funct10_rrr, rd: eX, rs1: eX, rs2: eX)
{
  action {
[...]
  stage EX:           pex_D1 = tex_D1 = aluR;
  stage ME:           pme_D1 = tme_D1 = pex_D1;
  stage WB:
    if (rd: x0)       w1_dead = w1 = pme_D1;
    else              X[rd] = w1 = pme_D1;
[...]
```

The RTL generator tool GO is capable of determining that there are no writes to field `X[0]`. To leverage this knowledge, we instruct GO to generate write accesses for each individual field, by setting the following option the GO configurations file:

```
register_write_per_field;
```

Additional remark: For each register file access in nML code, the C compiler assumes that the full range is accessible, i.e., that it can access all fields of a register (alias). For the above nML rule, there is no write access to the field `X[0]`. However, the C compiler interprets the write to `X[rd]` as full-range write access, including access to `X[0]`. This is still fine, since the field `X[0]` is explicitly marked as `reserved` in the compiler header file. The C compiler will never try to write to this reserved field.

5.1.3 Additional Registers

The program counter is a separate register. It is 32-bit or 64-bit wide, for the TRV32 and TRV64 models respectively.

```
reg PC <w64> read(pcr) write(pcw); hw_init PC = 0;
```

The program address for the instruction that are currently in the decode (`ID`) or execute (`EX`) stage is tracked with two additional nML registers:

```
reg  pif_PC <w64> read(trPC_ID) write(tif_PC); // PM addr. of instr @ ID
reg  pid_PC <w64> read(trPC_EX) write(tid_PC); // PM addr. of instr @ EX
```

The processor control unit (PCU) writes to `pif_PC` when issuing an instruction into the decode stage. Instructions that need to know their program address in the `EX` stage forward the value from `pif_PC` to `pid_PC` appropriately. Mainly the control instructions use this mechanism to allow instruction-relative addressing.

```
opn br_instr(op: funct3_bnch, i: c13s_s2, rs1: eX, rs2: eX)
{
  action {
#ifndef __programmers_view__
  stage ID:             pid_PC = tid_PC = trPC_ID = pif_PC;
#endif
[...]
```

For example in case of a branch instruction, the target is specified as offset, relative to the branch instruction. The PCU, which implements the branch action, uses the value of `pid_PC` to determine the address of the branch instruction in the `EX` stage.

For the hardware loop support (only in extended models), there are additional registers for the loop status, loop counts, loop start addresses, and loop end addresses.

```
reg LF        <t2u> read(lfr) write(lfw);       // loop flag
reg LS[2]<addr,t2u> read(lsr) write(lsw);       // loop start address
reg LE[2]<addr,t2u> read(ler) write(lew);       // loop end address
reg LC[2]<w64 ,t2u> read(lcr) write(lcw);       // loop count
```

5.2 Functional Units

`alu` This functional unit implements arithmetic, bit-wise logical, shift and comparison operations.

`mpy` This functional unit implements the multiplication operations. For RV32, the multiplier inputs are sign or zero extended to 33 bits and the product is 66 bits. For RV64, the multiplier inputs are sign or zero extended to 65 bits and the product is 130 bits. A multiplexer at the output selects the low or high product bits.

`agu` This functional unit computes the effective address. It only supports addition. The second operand is an immediate value that always comes from the instruction register.

`lx` This is the load-extension unit. It is basically a multiplexer, which selects between zero or sign extended versions of the byte, half-word, word and double-word inputs, which come from the data memory.

`div` This is a multi-cycle function unit, implemented in the file `div.p`, which executes a multi-cycle iterative division algorithm (restoring division) to execute division and remainder operations.

`dlp` This units computes start and end addresses for hardware loops. Only present in the extended models.

5.3 Transitory Storage

5.3.1 Transitories (Buses, Nets, Wires)

Most transitories are input and output nets for functional units.

`alu[ABRF]` Functional unit `alu`: arithmetic and logic unit, includes shifter. Input operands are `aluA` and `aluB`. Outputs are `aluR` for arithmetic, shift and logical operations, and `aluF` for comparisons.

`mpy[ABRLHW]`, `mpyMD` Functional unit `mpy`: multiplier. Input operands are `mpyA` and `mpyB`. The multiplier returns the low product bits via `mpyL` and the high bits via `mpyH`. The signedness of the input operands `mpyA` and `mpyB` is determined with the 2-bit mode selection input `mpyMD`. The `mpyR` is a multiplexer for `mpyL` or `mpyH`. For the 64-bit cores, there is an additional 32-bit wide transitory `mpyW`, which is used to perform sign extension on `mpyL`.

`agu[ABR]` Functional unit `agu`: address generation unit. Input operands are `aguA` and `aguB`. The compute address is provided on `aguR`.

`div[ABR]`	Functional unit `div`: multi-cycle divider unit. Input operands are `divA` and `divB`. The `divR` models the output path for the compiler only.
`div_{bsy,w3c,adr,wad}`	Functional unit `div`. Control transitories from/to `div` related to hazard control.
`lx[BHWR]`	Functional unit `lx`: load-extension unit. Input operands are `lxB` and `lxH`, and `lxW` for 64-bit models. For 64-bit models, the unit multiplexes onto `lxR` the sign- or zero-extended byte `lxB`, half-word `lxH`, word `lxW` or double-word `dmd_rd` memory load results. For 32-bit models, the unit multiplexes onto `lxR` the sign- or zero-extended byte `lxB`, half-word `lxH`, or word `dmw_rd` memory load results.
`dlp[AB],dlpL[SE]`	Functional unit `dlp`: hardware loop setup. This unit computes loop start and end addresses. Only present in the extended models.
`dlp_le_offs, dlp_lc`	Functional unit `dlp`. Conveys loop end offset and loop count values from core to PCU. Only present in the extended models.
`lnk_id, lnk_ex`	For subroutine calls, the PCU places the return addresses for instructions in the `ID` or `EX` stages on these transitories.
`cnd`	Condition flag for branch, from core to PCU.
`of21, of13_cd, trgt`	Jump offsets or absolute targets, from core to PCU.
`*_dead`	Dead-end transitories, for modeling writes to field `X[0]`.
`t{ex,me}_D[12]`	Dedicated write transitories for the respective nML pipe registers `pex_D1`, `pme_D1`, `pex_D2` and `pex_D2`, used as sources for bypasses. `D1` is only present for 5-stage models. `D2` is only present for extended 5-stage models.

The names of transitories that are designated for write access to pipeline registers follow the scheme below. The first letter is `t`, for `trn`. The next two letters indicate the pipeline stage in which the write occurs. The transitory's name closely matches the pipeline register's name. For example, `tex_D1` is the write transitory for `pex_D1` (see the subsection on pipeline registers).

5.3.2 Processor Ports

The TRV models currently have no nML processor ports (keywords `inport` and `outport`).

5.3.3 Pipeline Registers

Pipeline register names follow the scheme below. The first letter is `p`, for `pipe`. The next two letters indicate the pipeline stage in which the write occurs. For example `pid` is a pipeline register that is written in the `ID` stage. Pipeline registers need to be read in the next cycle, unless the stage is stalled.

`pid_D1`	Forwards return address from relative call instruction `jal` from `ID` to `EX` for write back. Only present in 3-stage models.
`pid_D2`	Forwards AGU result in case of memory instructions with post-modify addressing, from `ID` to `EX` for write back. Only present in extended 3-stage models.

`pid_S[12]` These pipeline registers hold the first (second) source operand value, fetched from the register file in the `ID` stage, and forward the values to the `EX` stage, where the operations are executed.
Only present in 5-stage models.

`p{ex,me}_D[12]` Write back paths through the pipeline from `EX` to `ME` to `WB`. Operations place their result into these pipeline registers, to forward it to the write back in stage `WB`. Most operations place the result in `EX`. The load result and the multiplier result are inserted in `ME`. The `D1` path is used for most units. The `D2` path is used only for the AGU write back, in case of memory instructions with post-modify addressing (only extended models). There are explicit write transitories for each of these pipeline registers. These transitories are start points for the bypasses.
Only present in 5-stage models.

6

Models With 3-Stage Pipeline

6.1 Stages

Instruction Fetch (IF)

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

Instruction Decode (ID)

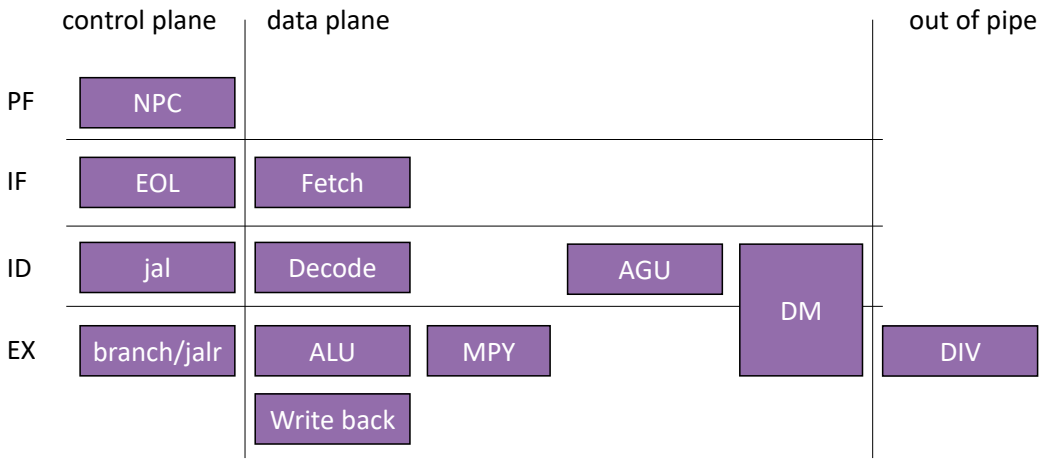
- The instruction that was fetched in the previous cycle is decoded.
- Address modifications and standalone address computations on the address generation unit are executed. Operands for the AGU operation are read in this stage.
- For all memory operations, the effective address is computed and is sent to the memory. The load or store operation is started.
- The unconditional relative jump instruction (jal) executes in this stage.

Instruction Execute (EX)

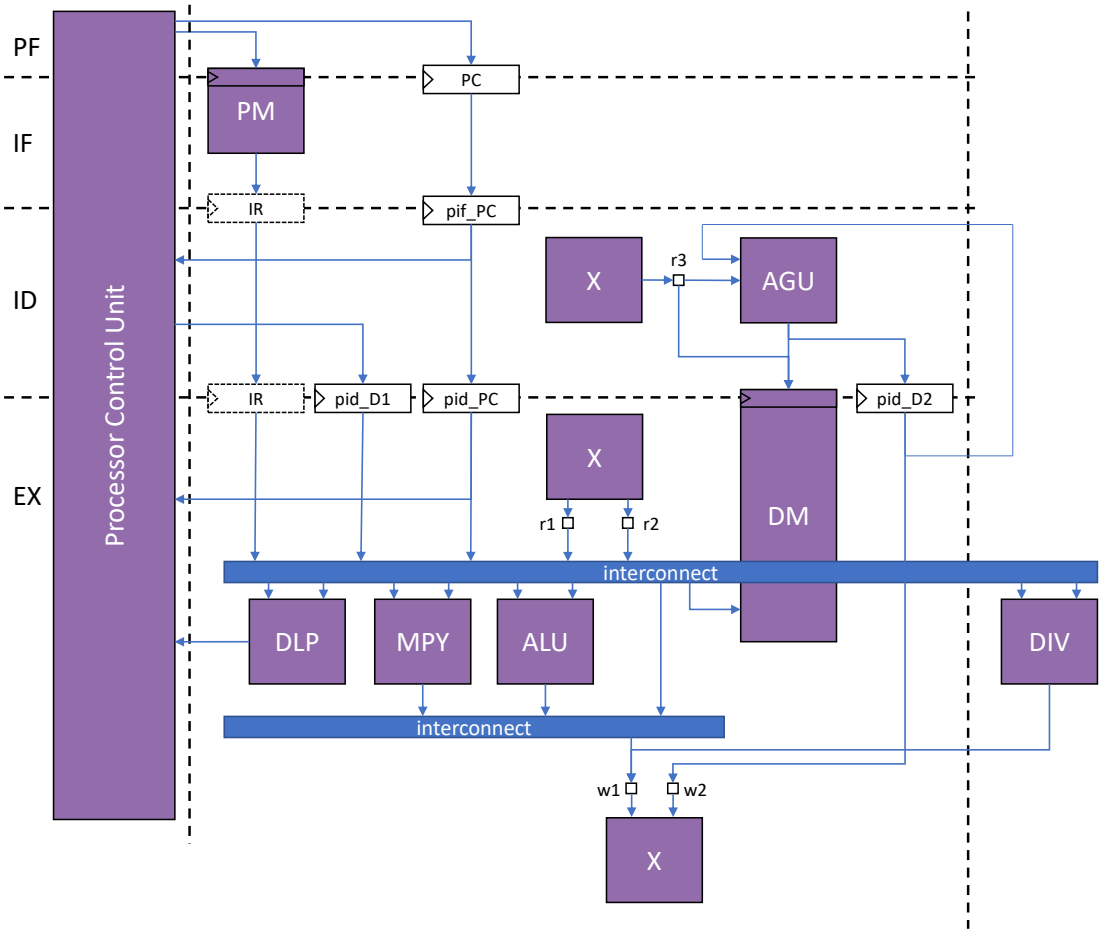
- Operands for ALU, multiplier, divider and control operations are read in this stage.
- This is the stage in which the ALU executes its operation.
- The multiplier unit executes in this stage.
- The multi-cycle iterative division is started in this stage and can take a variable number of cycles to finish.
- The result of memory load operations is available on the data bus.
- For store operations, the data is sent to the memory.
- The result of ALU, shift, multiply, AGU, control and load operations is written to the destination field in the register file.
- The conditional branch instructions execute in this stage.
- The unconditional absolute jump instruction (jalr) executes in this stage.
- The hardware loop registers are setup in this stage.

6.2 Overview

The `next-PC` computation as part of the controller is visualized as prefetch stage `PF`.



6.3 Structural Skeleton



6.4 Bypasses

The base models (p3) have no bypasses.

The extended models (p3x) support post-modify addressing modes on the AGU. There is a bypass from the AGU write back to the AGU read port.

```
// cycle            0  1  2  3  4  5  6  7  8
// ld x3,4(x5!)     IF ID EX
// ld x4,4(x5!)        IF ID EX
//                        ^^ bypass x5

bypass 1 cycles class(agu_to_agu) () {
  stage EX: X$[#] = #pid_D2;
} -> {
  stage ID: #r3 = X$[#];
}
```

6.5 Hardware Stalls

The controller inserts one stall cycle when the AGU reads a result that is produced by the preceding instruction on a unit other than the AGU. For the extended models, there is an additional bypass from AGU to AGU, allowing back-to-back post-modify address computations (see section on bypasses).

```
// Hazard condition
// cycle            0  1  2  3  4  5  6  7  8
// mv x5, x3        IF ID EX
//                        ^^ write x5
// ld x4,4(x5!)        IF ID EX
//                        ^^ read x5

hw_stall 1 cycles class(agu_read_after_write) () {
  stage EX: X$[#] = w1;
} -> {
  stage ID: r3 = X$[#];
}
```

Multiple hardware stall rules coordinate the interaction between the core pipeline and the multi-cycle divider unit.

- Stall when an instruction reads the result while unit is still busy (RAW).

```
hw_stall trn class(read_after_div) () {
  trn(div_bsy); address_trn(div_adr);
} -> {
  ... = X$[#];
}
```

- Do not write to destination register before div result is written (WAW).

```
hw_stall trn class(write_after_div) () {
  trn(div_bsy); address_trn(div_adr);
} -> {
  X$[#] = ...;
}
```

- No new division may be started while the previous is still busy.

```
hw_stall trn class(div_busy) () {
  trn(div_bsy);
} -> {
  class(div);
}
```

- No instruction may use register write port when result is written. The divider sets the transitory `div_wnc` when it writes in the next cycles, i.e., will use write port `w1` in the next cycle.

```
hw_stall trn class(div_wp) () {
  trn(div_wnc);
} -> {
  stage EX: X$[] = w1;
}
```

Additionally, the division and remainder instructions are annotated with `cycles(2)`, which means that they occupy the decode stage for two cycles. The transitories from the divider to the core which are used for hardware stalls are only valid in the cycle after the divider setup. The divider implementation is a basic Moore machine. The output transitories are computed based on the current state of the divider, and are not based on the divider state **and** the divider input. Since hardware stall rules only stall instructions in

the decode stage, we use the `cycles(2)` annotation to delay subsequent possibly dependent instructions until the divider setup is ready. The divider is activated in the `EX` stage, right after the decode stage `ID`. It is sufficient to keep `ID` empty while the division instruction is in `EX`. The `cycles(2)` annotation achieves exactly this.

The control flow instructions are annotated in nML with the `cycles` property. For example, an instruction with property `cycles(2)` occupies the decode stage for two cycles. This causes one stall cycle at the fetch stage.

- The jump-and-link instruction `jal` with a relative jump offset encoded in the instruction bits, executes in the decode stage. It requires two cycles.
- The jump-and-link-register instruction `jalr`, which reads the jump target from a register field and adds a relative offset, executes in the execute stage. It requires three cycles.
- The branch-and-compare instructions, which read two register fields and compare their value, execute in the execute stage. If the branch is not taken, there is no branch penalty, so the branch instruction takes only one cycle. If the branch is taken, there is penalty of two cycles, so the branch instruction takes in total three cycles. The PCU speculatively continues fetching and issuing, as if the branch is not taken. For branches, default assumption is fall-through. The PCU kills the speculatively fetched and issued instructions when the branch is taken.
- Hardware loop setup instructions occupy the decode stage for three cycles.

6.6 Software Stalls

In the presence of hardware loop support (only present in the extended models), some hazard conditions are avoided in software (i.e., at compile time by the C compiler). The end-of-loop check is performed in the fetch stage `IF`, when an instruction is about to be submitted to the decode stage `ID`. Control flow changes by other control instructions, like conditional and unconditional jumps, however occur in stages later than `IF`.

A minimum offset is required from a control instruction to the end-of-loop, such that the control instruction and the instructions that follow cannot trigger an end-of-loop check before the jump condition is known.

The `special(end_of_loop)` statement is a predefined notation corresponding to the end point of the last instruction inside a loop implemented by a hardware do-loop, that is, it refers to the start point of the next instruction.

- A jump may not be the end-of-loop instruction, as the end-of-loop check at `IF` stage would then come earlier than the jump action at `ID` or `EX`. We assume that `jal` and `jalr` are annotated with appropriate `cycles(N)` properties such that there are no delay slots. Hence, an offset of 1 instruction is sufficient for the hazard condition. The instruction after the `jal` or `jalr` is fetched but not issued, thus cannot trigger the end-of-loop check.

```
// Hazard condition
// cycle            0  1  2  3  4  5  6  7  8
// jal              IF ID ..
//                     ^^ jump action
// next instr          IF
//                     ^^ end-of-loop check @ PCU

// Hazard condition
// cycle            0  1  2  3  4  5  6  7  8
// jalr             IF ID EX ..
//                        ^^ jump action
// next instr          IF IF
//                     ^^ stall due to cycles(3)
//                        ^^ end-of-loop check @ PCU

sw_stall 1 instructions class(ctrl_before_EOL) () {
  class(ctrl);
} -> {
  special(end_of_loop);
}
```

- The branch instruction has the annotation `cycles(3|1)`. It requires three cycles if the branch is taken, and one cycle if the branch is not taken. The PCU fetches the next two instructions, and already issues the next instruction, which it kills if the branch is taken. The speculatively issued instruction, the one right after the branch, could trigger the end-of-loop check, but then might be killed. The branch and this instruction cannot be the end of the loop.

```
// Hazard condition - speculatively issue instr. following branch in cycle 1,
//                    kill that instruction in cycle 2, when branch is taken
// cycle            0  1  2  3  4  5  6  7  8
// branch           IF ID EX ..
//                        ^^ jump action
// next instr          IF **killed
//                     ^^ end-of-loop check @ PCU

sw_stall 1..2 instructions class(branch_before_EOL) () {
  class(branch);
} -> {
  special(end_of_loop);
}
```

- Two (nested) hardware loops may not end at the same address. This is a limitation of the simple end-of-loop check in the PCU. To allow two nested hardware loops to end on the same instruction, a more sophisticated end-of-loop check can be added to the PCU.

```
sw_stall 0 instructions class(between_loop_ends) () {
  special(end_of_loop);
} -> {
  special(end_of_loop);
}
```

- The last instruction of a hardware loop should not be a misaligned jump target. This simplifies the end-of-loop check in the PCU.

```
sw_stall 1 instructions class(aligned_EOL) () {
  special(jump_target);
} -> {
  special(end_of_loop);
}
```

- The last instruction of a hardware loop should not be a misaligned return address, i.e., jump target. This simplifies the end-of-loop check in the PCU. To achieve this, we forbid calls as the last two instructions in a hardware loop.

```
sw_stall 1..2 instructions class(aligned_EOL) () {
  LR = jal(of21);
} -> {
  special(end_of_loop);
}

sw_stall 1..2 instructions class(aligned_EOL) () {
  LR = jalr(trgt);
} -> {
  special(end_of_loop);
}
```

7

Models With 5-Stage Pipeline

7.1 Stages

Instruction Fetch (IF)

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

Instruction Decode (ID)

- The instruction that was fetched in the previous cycle is decoded.
- Operands for ALU, AGU, multiplier, divider and control operations are read in this stage. They are latched in pipeline registers.
- The unconditional relative jump instruction (jal) executes in this stage.

Instruction Execute (EX)

- This is the stage in which the ALU executes its operation.
- The pipelined 2-stage multiplier unit starts execution in this stage.
- The multi-cycle iterative division starts in this stage and can take a variable number of cycles to finish.
- Address modifications and standalone address computations on the address generation unit are executed.
- For all memory operations, the effective address is computed and is sent to the memory. The load or store operation is started.
- The conditional branch instructions execute in this stage.
- The unconditional absolute jump instruction (jalr) executes in this stage.
- The hardware loop registers are setup in this stage.

Memory Access (ME)

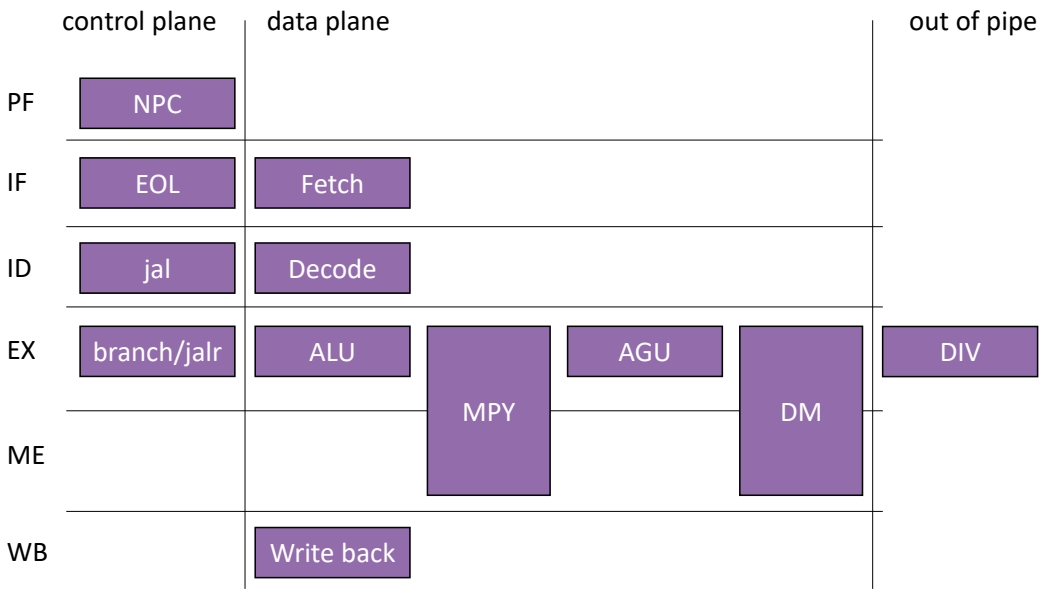
- The result of memory load operations is available on the data bus.
- For store operations, the data is sent to the memory.
- The pipelined 2-stage multiplier unit finishes execution in this stage.

Write Back (WB)

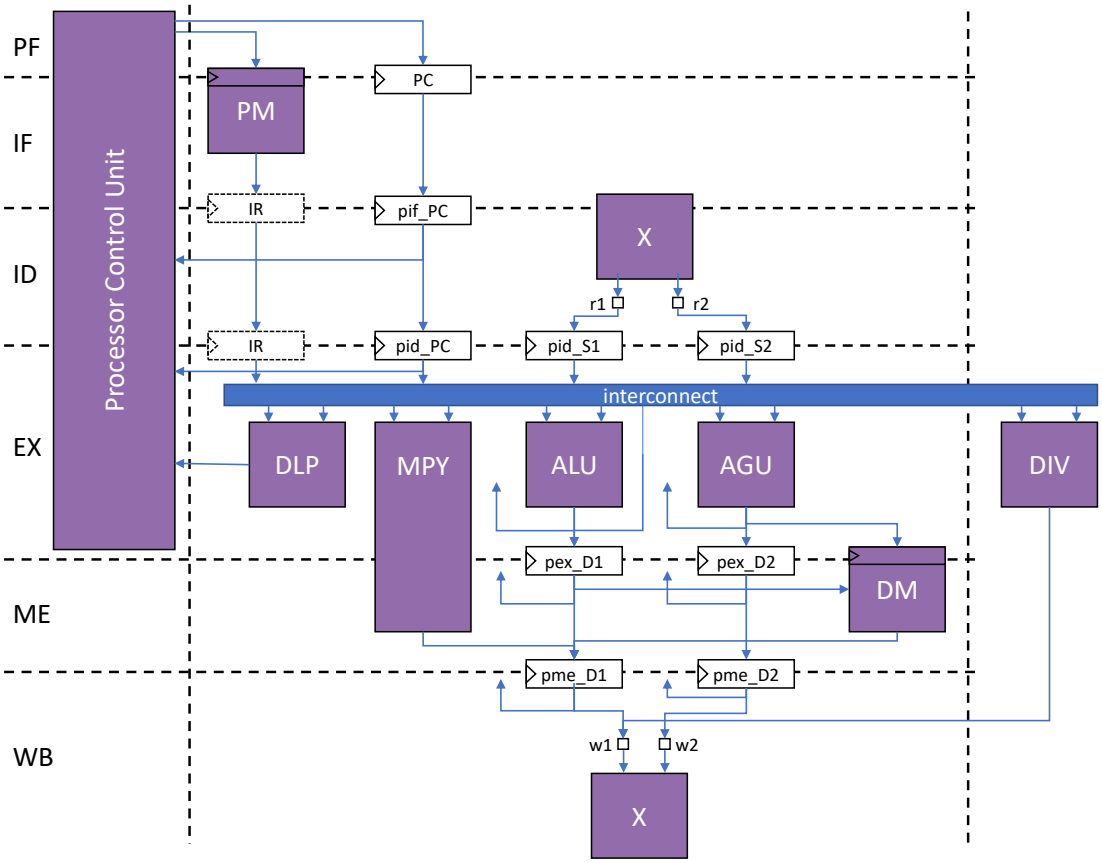
- The result of ALU, shift, multiply, AGU, control and load operations is written to the destination field in the register file.

7.2 Overview

The `next-PC` computation as part of the controller is visualized as prefetch stage `PF`.



7.3 Structural Skeleton



7.4 Bypasses

There are bypasses from each stage `EX`, `ME`, `WB` to the decode stage `ID`. For the base models, there are six bypasses. For the extended models, there are twelve bypasses. Bypass destinations are the two read ports `r1` and `r2`. Bypass sources are the write transitories of the destination pipeline registers (one for base, two for extended models).

For example, this is one `EX` to `ID` bypass, from `D1` to `r1`:

```
// cycle            0  1  2  3  4  5  6  7  8
// x4 = 1           IF ID EX ME WB
// x5 = x4 + 1         IF ID EX ME WB
//                        ^^ bypass

bypass 1 cycles class(bypass_from_EX) () {
  stage EX..WB: X$[#]‘WB‘ = wX‘WB‘ = #tex_D1‘EX‘;
} -> {
  stage ID: #r1 = X$[#];
}
```

This is one `WB` to `ID` bypass, from `D2` to `r2` (Only present in the extended models):

```
// cycle            0  1  2  3  4  5  6  7  8
// x4 = 1           IF ID EX ME WB
// xxx                 IF ID EX ME WB
// yyy                    IF ID EX ME WB
// x5 = x4 + 1               IF ID EX ME WB
//                              ^^ bypass

bypass 3 cycles class(bypass_from_WB) () {
  stage WB: X$[#] = wX = #pme_D2;
} -> {
  stage ID: #r2 = X$[#];
}
```

7.5 Hardware Stalls

The load-to-use latency is two cycles. The controller inserts one stall cycle if the load result is used by the instruction immediately succeeding the load.

```
// Hazard condition
// cycle            0  1  2  3  4  5  6  7  8
// ld x3,4(x5)      IF ID EX ME WB
//                        ^^ DM request
//                           ^^ DM response
// mv x6,x3            IF ID EX ME WN
//                        ^^ read x3 before load result is available

hw_stall 1 cycles class(read_after_load) () {
  stage ME..WB: X$[#]‘WB‘ = w1‘WB‘ = lxR‘ME‘;
} -> {
  stage ID: ... = X$[#];
}
```

The multiply-to-use latency is two cycles, since we have a two-stage pipelined multiplier. The controller inserts one stall cycle if the multiplication result is used by the instruction immediately succeeding the multiplication.

```
// Hazard condition
// cycle            0  1  2  3  4  5  6  7  8
// mul x3,x4,x5     IF ID EX ME WB
//                        ^^ start
//                           ^^ finish
// mv x6,x3            IF ID EX ME WN
//                        ^^ read x3 before multiplication is ready

hw_stall 1 cycles class(read_after_mpy) () {
  stage ME: pme_D1 = mpyR;
  stage WB: X$[#] = pme_D1;
} -> {
  stage ID: ... = X$[#];
}
```

Multiple hardware stall rules coordinate the interaction between the core pipeline and the out-of-pipe multi-cycle divider unit.

- Stall when an instruction reads the result while unit is still busy (RAW).

```
hw_stall trn class(read_after_div) () {
  trn(div_bsy); address_trn(div_adr);
} -> {
  ... = X$[#];
}
```

- Do not write to destination register before div result is written (WAW).

```
hw_stall trn class(write_after_div) () {
  trn(div_bsy); address_trn(div_adr);
} -> {
  X$[#] = ...;
}
```

- No new division may be started while the previous is still busy.

```
hw_stall trn class(div_busy) () {
  trn(div_bsy);
} -> {
  class(div);
}
```

- No instruction may use register write port when result is written. The divider sets the transitory `div_w3c` when it writes in the three cycles, i.e., will use write port `w1` in three cycles. This forecast is required because `hw_stall` rules are stalling instructions only in the decode stage. So we predict that we will use the write port `w1` in three cycles, and stall an instruction in the decode stage if it will also use the write port `w1` in three cycles.

```
hw_stall trn class(div_wp) () {
  trn(div_w3c);
} -> {
  stage WB: X$[] = w1;
}
```

Additionally, the division and remainder instructions are annotated with `cycles(2)`, which means that they occupy the decode stage for two cycles.

The control flow instructions are annotated in nML with the `cycles` property. For example, an instruction with property `cycles(2)` occupies the decode stage for two cycles. This causes one stall cycle at the fetch stage.

- The jump-and-link instruction `jal` with a relative jump offset encoded in the instruction bits, executes in the decode stage. It requires two cycles.
- The jump-and-link-register instruction `jalr`, which reads the jump target from a register field and adds a relative offset, executes in the execute stage. It requires three cycles.
- The branch-and-compare instructions, which read two register fields and compare their value, execute in the execute stage. If the branch is not taken, there is no branch penalty, so the branch instruction takes only one cycle. If the branch is taken, there is penalty of two cycles, so the branch instruction takes in total three cycles. The PCU speculatively continues fetching and issuing, as if the branch is not taken. For branches, default assumption is fall-through. The PCU kills the speculatively fetched and issued instructions when the branch is taken.
- Hardware loop setup instructions occupy the decode stage for three cycles.

7.6 Software Stalls

Please see the section on software stalls for models with a 3-stage pipeline. For software stalls, there is currently no difference between 3-stage and 5-stage pipelines.

8

Instruction Set

8.1 TRV32P3 and TRV32P5

The two models TRV32P3 and TRV32P5 implement the RV32IM ISA specification [3]. The list below shows the assembly syntax of all instructions. Some instructions have multiple alternative syntax, which the RISC-V ISA specification calls pseudoinstructions (for example: `beqz X, imm` is `beq x0, X, imm`).

```
addi X, X, imm
add X, X, X
andi X, X, imm
and X, X, X
auipc X, imm
beq X, X, imm
beqz X, imm
bgeu X, X, imm
bge X, X, imm
bgez X, imm
bgtz X, imm
blez X, imm
bltu X, X, imm
blt X, X, imm
bltz X, imm
bne X, X, imm
bnez X, imm
call imm
call X
divu X, X, X
div X, X, X
jalr X, X, imm
jal X, imm
j imm
jr X
lbu X, imm(X)
lb X, imm(X)
lhu X, imm(X)
lh X, imm(X)
li X, imm
lui X, imm
lw X, imm(X)
mulhsu X, X, X
mulhu X, X, X
mulh X, X, X
mul X, X, X
mv X, X
neg X, X
nop
not X, X
oriu X, X, imm
ori X, X, imm
or X, X, X
remu X, X, X
rem X, X, X
ret
sb X, imm(X)
seqz X, X
sgtz X, X
sh X, imm(X)
slli X, X, imm
sll X, X, X
sltiu X, X, imm
slti X, X, imm
sltu X, X, X
slt X, X, X
sltz X, X
snez X, X
srai X, X, imm
sra X, X, X
srli X, X, imm
srl X, X, X
sub X, X, X
sw X, imm(X)
xori X, X, imm
xor X, X, X
```

The `oriu` instruction is non-standard extension. It is a bitwise logical `OR` instruction with an unsigned 12-bit immediate. The complex constant generation mechanism uses `lui` and `oriu` to generate 32-bit constants:

```
lui   x5, (value>>12)
oriu  x5, x5, (value & 0xfff)
```

8.2 TRV32P3X and TRV32P5X

The two models TRV32P3X and TRV32P5X implement the RV32IMC ISA specification and include a few custom extensions.

The list below shows the assembly syntax of all instructions that are additional to the base models (see previous section). Some instructions have multiple alternative syntax, which the RISC-V ISA specification calls pseudoinstructions (for example: `beqz X, imm` is `beq x0, X, imm`).

```
c.addi16sp imm          c.lui X, imm            do X, imm
c.addi4spn X,X,imm      c.lwsp X, imm(X)        lbu X, imm(X!)
c.addi4spn X,X, imm     c.lw X, imm(X)          lbu X, imm(X!)
c.addi X, imm           c.mv X, X               lb X, imm(X!)
c.add X, X              c.nop                   lb X, imm(X!)
c.andi X, imm           c.or X, X               lhu X, imm(X!)
c.and X, X              c.ret                   lhu X, imm(X!)
c.beqz X, imm           c.slli X, imm           lh X, imm(X!)
c.bnez X, imm           c.srai X, imm           lh X, imm(X!)
c.jal imm               c.srli X, imm           lw X, imm(X!)
c.jalr X                c.sub X, X              lw X, imm(X!)
c.j imm                 c.swsp X, imm(X)        sb X, imm(X!)
c.jr X                  c.sw X, imm(X)          sh X, imm(X!)
c.li X, imm             c.xor X, X              sw X, imm(X!)
```

The load and store instructions support a post-modify addressing mode: `imm(X!)`. In this mode, the effective address for the memory access is the value stored in the specified base register. The address is post-modified by adding the immediate value. The modified address is written back to the base register. Note that the unmodified address, before the update, is sent to the memory.

For example, the load-byte-unsigned instruction with post-modify addressing has the following syntax:

```
lbu rd, imm(rs1!)
```

It executes the following operation:

```
rd   = zext(DMb[rs1])
rs1 += imm
```

The `do` instruction performs the setup for zero-overhead hardware loops. In a loop nest, up to two loop levels can be mapped onto hardware loops. The remaining loop levels are implemented using regular branch-compare instructions.

The syntax for the `do` instructions is as follows:

```
do rs1, imm
```

The `do` instruction has two arguments. The first argument is the loop count, which is retrieved from the central register file. The second argument is the relative offset to the loop end. The offset is relative to the address of the `do` instruction.

Hardware loops have a few constraints. The compiler is aware of these via software stall rules. These rules are documented in Section 6.6. Here is a summary:

- An unconditional jump cannot be the last instruction of the loop.
- A conditional jump cannot be the last and previous-to-last instruction.
- Two nested hardware loops cannot end on the same address.
- The last instruction cannot be a misaligned jump or return target.

The following listing is an example for a simple single-instruction loop. The `addi` instruction is executed ten times. The loop count register is `x5`. The loop end offset, relative to the `do` instruction, is four bytes.

```
li   x4,  0
li   x5,  10
do   x5,  4
addi x3,  x3, 1
mv   x10, x3
```

An instruction tracing of the execution of the above snippet would like this:

```
li   x4,  0
li   x5,  10
do   x5,  4
[issue bubble]
[issue bubble]
addi x3,  x3, 1
addi x3,  x3, 1
addi x3,  x3, 1
addi x3,  x3, 1
addi x3,  x3, 1
addi x3,  x3, 1
addi x3,  x3, 1
addi x3,  x3, 1
addi x3,  x3, 1
addi x3,  x3, 1
mv   x10, x3
```

The loop setup takes three cycles in total. The `do` instruction copies the value of `x5` into the loop count register `LC`. The register `x5` can be reused inside the loop.

Up to two hardware loops can be nested, but they cannot end on the same instruction. The listing below shows a simple example. The inner loop contains only a single instruction. The outer loop is three instructions long. There is a `nop` instruction, because the two loops cannot end on the same address.

```
li   x4,  0
li   x5,  10
li   x6,  10
do   x5,  12
do   x6,  4
addi x3,  x3, 1
nop
mv   x10, x3
```

8.3 TRV64P3 and TRV64P5

The two models TRV64P3 and TRV64P5 implement the RV64IM ISA specification [3].

The list below shows the assembly syntax of all instructions. Some instructions have multiple alternative syntax, which the RISC-V ISA specification calls pseudoinstructions (for example: `beqz X, imm` is `beq x0, X, imm`).

```
addiw X, X, imm
addi X, X, imm
addw X, X, X
add X, X, X
andi X, X, imm
and X, X, X
auipc X, imm
beq X, X, imm
beqz X, imm
bgeu X, X, imm
bge X, X, imm
bgez X, imm
bgtz X, imm
blez X, imm
bltu X, X, imm
blt X, X, imm
bltz X, imm
bne X, X, imm
bnez X, imm
call imm
call X
divuw X, X, X
divu X, X, X
divw X, X, X
div X, X, X
jalr X, X, imm
jal X, imm
j imm
jr X
lbu X, imm(X)
lb X, imm(X)
ld X, imm(X)
lhu X, imm(X)
lh X, imm(X)
li X, imm
lui X, imm
lwu X, imm(X)
lw X, imm(X)
mulhsu X, X, X
mulhu X, X, X
mulh X, X, X
mulw X, X, X
mul X, X, X
mv X, X
negw X, X
neg X, X
nop
not X, X
oriu X, X, imm
ori X, X, imm
or X, X, X
remuw X, X, X
remu X, X, X
remw X, X, X
rem X, X, X
ret
sb X, imm(X)
sd X, imm(X)
seqz X, X
sext.w X, X
sgtz X, X
sh X, imm(X)
slliw X, X, imm
slli X, X, imm
sllw X, X, X
sll X, X, X
sltiu X, X, imm
slti X, X, imm
sltu X, X, X
slt X, X, X
sltz X, X
snez X, X
sraiw X, X, imm
srai X, X, imm
sraw X, X, X
sra X, X, X
srliw X, X, imm
srli X, X, imm
srlw X, X, X
srl X, X, X
subw X, X, X
sub X, X, X
sw X, imm(X)
xori X, X, imm
xor X, X, X
```

The `oriu` instruction is non-standard extension. It is a bitwise logical `OR` instruction with an unsigned 12-bit immediate. The complex constant generation mechanism uses `lui` and `oriu` to generate 32-bit constants.

8.4 TRV64P3X and TRV64P5X

The two models TRV64P3X and TRV64P5X implement the RV64IMC ISA specification and include a few custom extensions.

The list below shows the assembly syntax of all instructions that are additional to the base models (see previous section). Some instructions have multiple alternative syntax, which the RISC-V ISA specification calls pseudoinstructions (for example: `beqz X, imm` is `beq x0, X, imm`).

```
c.addi16sp imm          c.lwsp X, imm(X)        lbu X, imm(X!)
c.addi4spn X,X,imm      c.lw X, imm(X)          lbu X, imm(X!)
c.addi4spn X,X, imm     c.mv X, X               lb X, imm(X!)
c.addiw X, imm          c.nop                   lb X, imm(X!)
c.addi X, imm           c.or X, X               ld X, imm(X!)
c.addw X, X             c.ret                   ld X, imm(X!)
c.add X, X              c.sdsp X, imm(X)        lhu X, imm(X!)
c.andi X, imm           c.sd X, imm(X)          lhu X, imm(X!)
c.and X, X              c.sextw X               lh X, imm(X!)
c.beqz X, imm           c.slli X, imm           lh X, imm(X!)
c.bnez X, imm           c.srai X, imm           lwu X, imm(X!)
c.jalr X                c.srli X, imm           lwu X, imm(X!)
c.j imm                 c.subw X, X             lw X, imm(X!)
c.jr X                  c.sub X, X              lw X, imm(X!)
c.ldsp X, imm(X)        c.swsp X, imm(X)        sb X, imm(X!)
c.ld X, imm(X)          c.sw X, imm(X)          sd X, imm(X!)
c.li X, imm             c.xor X, X              sh X, imm(X!)
c.lui X, imm            do X, imm               sw X, imm(X!)
```

Please see Section 8.2 for an introduction to the extra instructions (load/store with postmodify addressing, and hardware loops).

9

nML Model

9.1 Opcode Enums

Enumeration types for the opcodes are specified in the file `opc.n`.

The enum `opc32` groups the major opcode formats for the 32-bit instructions:

```
enum opc32 {
  OP          =  0b0110011,
  OP_IMM      =  0b0010011,
[...]
  SEVEN       =  0b1111111
};
```

There are enumeration types for format-specific function opcodes:

```
enum funct10_rrr {
  add  = 0b0000000000,
  sub  = 0b0100000000,
[...]
  TEN  = 0b1111111111
};
```

For the extensions (compressed instructions, hardware loops), the associated enumeration types are defined the in the respective nML files, i.e., `rvc.n` and `dlp.n`.

9.2 nML Rules

The top level nML rule of all models looks like this:

```
start trv64p5x;
opn trv64p5x (
    bit32_ifmt
  | bit16_ifmt    // only present in extended models
#ifdef __chess__
  | br_far_pinstr // pseudo instr.
#endif
);
```

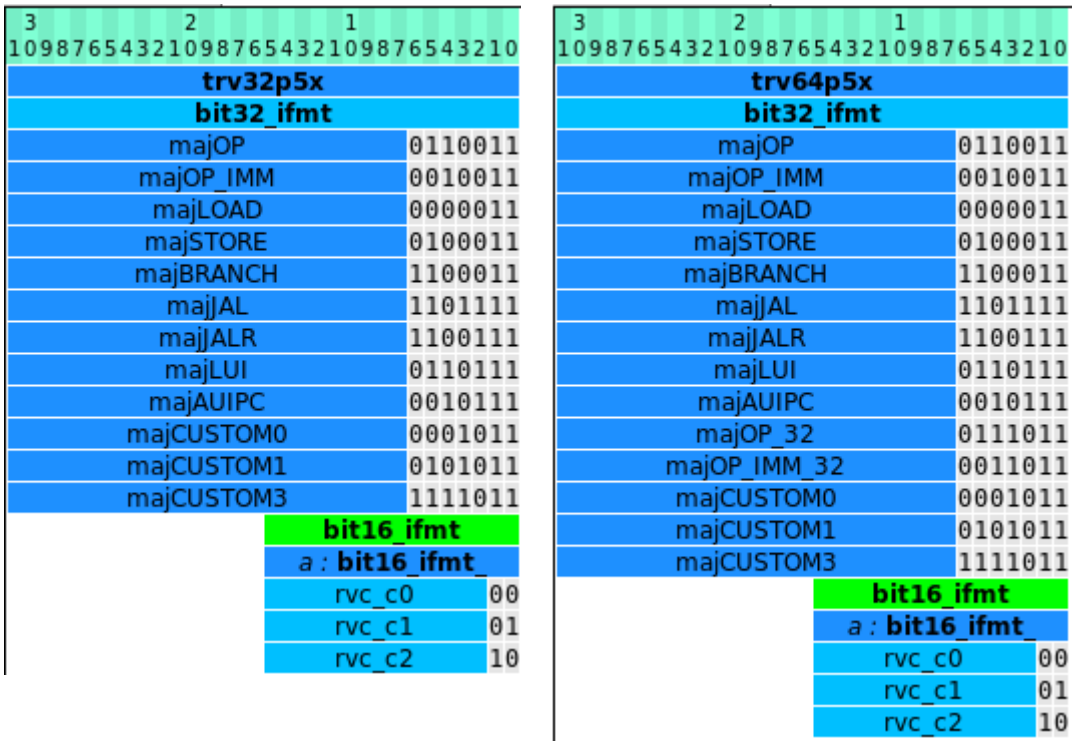
The `bit32_ifmt` rule is for the regular instructions that are 32-bit wide. The `bit16_ifmt` rule is for the compressed instructions that are 16-bit wide. The models have one pseudo instruction `br_far_pinstr` (explained later, for far branches), which is visible only to the C compiler. The ending `pinstr` means *pseudo instruction*.

9.3 32-bit Instruction Format

The top-level nML rule for the 32-bit instruction format is an nML `OR` rule that combines the major opcode formats. Per major opcode, there is one rule which lists the instructions belonging to that class. These rules are annotated with `complete_image` to simplify the decoding logic.

```
opn bit32_ifmt(
    majOP
  | majOP_IMM
[...]
)
{
  image : majOP      :: opc32.OP
        | majOP_IMM  :: opc32.OP_IMM
[...]
}

// One rule per major opcode
opn majOP      (alu_rrr_ar_instr | mpy_rrr_instr | div_instr) complete_image;
opn majOP_IMM  (alu_rris_ar_instr | alu_rri_sh_instr)         complete_image;
[...]
```

For example, this is an overview for the TRV32P5X and TRV64P5X models:



Instruction nML rules end with `_instr`. Each of these describes one or more instructions. The arguments typically are the function opcode, and the enumeration types for the register fields. This is an example for 3-register instructions on the ALU:

```
// ~~~ ALU
// <op> rd, rs1, rs2
// add, sub, xor, or, and, slt, sltu, sll, srl, sra

opn alu_rrr_ar_instr(op: funct10_rrr, rd: eX, rs1: eX, rs2: eX)
{
  action {
  stage ID:             pid_S1 = r1 = X[rs1];
                        pid_S2 = r2 = X[rs2];
  stage EX:             aluA = pid_S1;
                        aluB = pid_S2;
  // ---
    switch (op) {
      case add  :       aluR = add  (aluA,aluB) @alu;
[...]
    }
  // ---
  stage EX:             pex_D1 = tex_D1 = aluR;
  stage ME:             pme_D1 = tme_D1 = pex_D1;
  stage WB:
    if (rd: x0)         w1_dead = w1 = pme_D1;
    else                X[rd] = w1 = pme_D1;
  }
  syntax : "neg"  PADMNM " " rd "," PADOP1 rs2  op<<sub>>  rs1<<x0>>
             // sub  rd, x0,  rs2
[...]
         | op     PADMNM " " rd "," PADOP1 rs1 "," PADOP2 rs2;
  image  : op[9..3]::rs2::rs1::op[2..0]::rd, class(alu_rrr);
}
```

The instruction rules follow the following scheme.

- The header documents the instruction format and the instructions.

  ```
  // ~~~ ALU
  // <op> rd, rs1, rs2
  // add, sub, xor, or, and, slt, sltu, sll, srl, sra
  ```

- The `action` statement is split into three parts.

  1. The first part fetches the operands.

     ```
     stage ID:             pid_S1 = r1 = X[rs1];
                           pid_S2 = r2 = X[rs2];
     stage EX:             aluA = pid_S1;
                           aluB = pid_S2;
     ```

  2. The second part executes the operation.

     ```
       switch (op) {
         case add  :       aluR = add  (aluA,aluB) @alu;
     [...]
       }
     ```

  3. The third part writes back the results.

```
stage EX:             pex_D1 = tex_D1 = aluR;
stage ME:             pme_D1 = tme_D1 = pex_D1;
stage WB:
  if (rd: x0)         w1_dead = w1 = pme_D1;
  else                X[rd] = w1 = pme_D1;
```

- The syntax attribute defines the assembly syntax. The models use alternative syntax with matching conditions to simplify the assembly where possible.

```
syntax : "neg"  PADMNM " " rd "," PADOP1 rs2  op<<sub>>  rs1<<x0>>
           // sub  rd, x0,  rs2
[...]
         | op     PADMNM " " rd "," PADOP1 rs1 "," PADOP2 rs2;
```

  The regular syntax is `op rd, rs1, rs2`. If the opcode is `sub` and the first operand register is `x0`, then an alternative assembly syntax is `neg rd, rs2`. The diassembler shows the first matching syntax. The assembler accepts both `neg rd, rs2` and `sub rd, x0, rs2`.

- There are padding specifiers in the syntax attribute to align operands in the disassembly. These are defined in the top nML file like this:

```
#define PADMNM _pad_7
#define PADOP1 _pad_6
#define PADOP2 _pad_5
```

- The image attribute specifies the instruction encoding. The last seven bits, representing the major opcode, are added by the major opcode rules explained above.

```
image  : op[9..3]::rs2::rs1::op[2..0]::rd, class(alu_rrr);
```

  This rule is additionally annotated with `class(alu_rrr)`, which shows up in the profiling statistics.

9.4 16-bit Instruction Format

The top-level nML rule for the 16-bit instruction format is `bit16_ifmt`, specified in the file `rvc.n`. It is only present in the extended models. The rule `bit16_ifmt` only adds the `class(rvc)` annotation to all compressed instructions. This allows to 1) see the use of compressed instructions in profiling statistics, and 2) disable the instruction class e.g. on a per-function basis.

The nML rule `bit16_ifmt_` is an nML `OR` rule that lists the three quadrants of the 16-bit encoding space. Each of the three quadrants, labels `C0`, `C1` and `C2` in the RISC-V ISA specification, are described by the respectively named nML rules `rvc_c0`, `rvc_c1`, and `rvc_c2`. These three rules are annotated with `complete_image` to simplify decoding logic.

```
opn bit16_ifmt (a:bit16_ifmt_) { action: a; syntax: a; image: a, class(rvc); }

opn bit16_ifmt_(rvc_c0|rvc_c1|rvc_c2)
{
  image : rvc_c0 :: opc16.c0
        | rvc_c1 :: opc16.c1
        | rvc_c2 :: opc16.c2;
}

opn rvc_c0(
    addi4spn_instr16
  | lw_instr16
[...]
) complete_image;

opn rvc_c1(
[...]
```

The nML rules for compressed instructions end with `_instr16`. These rules follow the same scheme like the 32-bit instruction nML rules described above. There is a header. There are three parts of the `action` attribute: for reading, execution and write back.

```
// c.addi4spn rd, x2, imm
// -> addi rd, x2, imm
//
// i==0: reserved

opn addi4spn_instr16(rd: eXC, i: c10u_s4)
{
  action {
  stage ID:             pid_S1 = r1 = SP;
                        pid_S2 = i;
  stage EX:             aluA = pid_S1;
                        aluB = pid_S2;
  // ---
                        aluR = add (aluA,aluB) @alu;
  // ---
  stage EX:             pex_D1 = tex_D1 = aluR;
  stage ME:             pme_D1 = tme_D1 = pex_D1;
  stage WB:             XC[rd] = w1 = pme_D1;
  }
  syntax : "c.addi4spn" PADMNM " " rd "," PADOP1 "x2" "," PADOP2 i;
  image  : "000"::i[5..4]::i[9..6]::i[2 zero]::i[3]::rd, class(alu_rri);
}
```

9.5 RTL-Optimized Multiplier

The RISC-V ISA specification defines four (five for RV64) different multiplication instructions. Next to the regular multiply `mul`, there are instructions `mulh*` that return the high product bits. For these, the signedness of the operands is important. There are three variants:

- Both inputs are signed numbers: `mulh`.
- Both inputs are unsigned numbers: `mulhu`.
- One input is a signed, the other one is an unsigned number: `mulhsu`.

The RV64 ISA additionally defines a 32-bit multiply instruction `mulw`.

To ease the gate-level synthesis process, we use an RTL-optimized primitive function that implements all functional modes. This primitive makes the desired resource sharing between the different multiplication modes explicit. Otherwise, the gate-level synthesis tools have to identify possible resource sharing over and over again at every synthesis run. From our experience, this works well in most cases. For very tight timing constraints however, we have seen that the synthesis tools prefer to implement three distinct multipliers (signed, unsigned, mixed). While this can make sense for timing closure, it is usually undesired, hence we guide the synthesis tool in the resource sharing process, by modeling explicitly the intended result.

The relevant part of the `action` attribute of the nML rule for the multiplication instructions is shown below.

```
case mul    : mul_hw (mpyA‘EX‘,mpyB‘EX‘,mpyMD‘EX‘=0b11‘EX‘,mpyL‘ME‘,mpyH‘ME‘) @ mpy;
case mulh   : mul_hw (mpyA‘EX‘,mpyB‘EX‘,mpyMD‘EX‘=0b11‘EX‘,mpyL‘ME‘,mpyH‘ME‘) @ mpy;
case mulhsu : mul_hw (mpyA‘EX‘,mpyB‘EX‘,mpyMD‘EX‘=0b10‘EX‘,mpyL‘ME‘,mpyH‘ME‘) @ mpy;
case mulhu  : mul_hw (mpyA‘EX‘,mpyB‘EX‘,mpyMD‘EX‘=0b00‘EX‘,mpyL‘ME‘,mpyH‘ME‘) @ mpy;

// low product
case mul :
              mpyR‘ME‘ = mpyL‘ME‘;
// high product
case mulh|mulhsu|mulhu:
              mpyR‘ME‘ = mpyH‘ME‘;
```

This example shows a pipelined 2-stage multiplier. The multiplication starts in the `EX` stage. The product is available in the `ME` stage. The multiplier is pipelined, hence, a new multiplication can start in every cycle.

The 5-stage models use a pipelined 2-stage multiplier. The 3-stage models use a single-stage multiplier.

The `mul_hw` primitive always returns both low and high product bits, `mpyL` and `mpyH` respectively.

There are different ways to inform the C compiler about the usage of the `mul_hw` primitive. The preferred approach is to add a `chess_view` rule, which maps the different functional modes of `mul_hw` onto four (five for RV64) distinct primitives for each mode: `mul`, `mulh`, `mulhu` and `mulhsu`.

This `chess_view` rule in thr file `cv.n` provides an abstract view on the `mul` functional mode of the `mul_hw` primitive:

```
chess_view () {
  mul_hw (mpyA,mpyB,mpyMD=0b11,mpyL,mpyH);
  mpyR = mpyL;
} -> {
  mpyR = mul(mpyA,mpyB);
}
```

The rule for `mulhsu` looks like this:

```
chess_view () {
  mul_hw (mpyA,mpyB,mpyMD=0b10,mpyL,mpyH);
  mpyR = mpyH;
} -> {
  mpyR = mulhsu(mpyA,mpyB);
}
```

Outlook In the compiler header file `{trvXXpY}_int.h`, which defines the application layer for built-in integer arithmetic, the C built-in multiplication operator is promoted onto the `mul` primitive:

```
promotion int operator * (int,int) = w32 mul (w32,w32);
```

9.6 Pseudo Far Branch

The branch instruction offers only a limited target range. The offset immediate, pointing at the target relative to the branch instruction, has a limited value range. In rare cases, larger branch offsets are needed.

The TRV models have a pseudo instruction for far branches. This instruction rule is visibly only to the C compiler. It is a combination of a branch, with complemented condition, and of a jump, which offers a larger target range. To the C compiler, this appears as single wide instruction. All other tools see two instructions.

```
#ifdef __chess__

// pseudo instr.: conditional far branch
// b<!cmp> rs1, rs2, +4   ; jump over jal if negated condition is true
// jal x0, imm            ; jump to far target

opn br_far_pinstr(op: funct3_bnch, i: c21s_s2, rs1: eX, rs2: eX)
{
  action {
  stage ID:             pid_S1 = r1 = X[rs1];
                        pid_S2 = r2 = X[rs2];
  stage EX:             aluA = pid_S1;
                        aluB = pid_S2;
  // ---
    switch (op) {
      // NOTE: conditions complemented
      case bne  :       aluF = eq  (aluA,aluB) @alu;
[...]
    }
                        br(cnd=aluF,of13_cd=i);
  }
  syntax : op " " rs1 "," rs2 "," i "chess_only";
  image  :
          // jal_instr writing to X[0]
          i[20]::i[10..1 zero]::i[11]::i[19..12]::"00000"::opc32.JAL

          // branch + 4 (jumping over 32-bit jal_instr)
          ::"0"::"000000"::rs2::rs1::op::"0100"::"0"::opc32.BRANCH

          // artificial prefix, removed by Chess, use 64b opcode space
          ::"0000000000111111",

          cycles(4|2), class(ctrl), class(pbranch), chess_pc_offset(6),
             class(chess_artificial_prefix_word);
}

#endif
```

The pseudo instruction is ten bytes wide. Four bytes belong to the branch instruction. Another four bytes belong to the jump instruction. The remaining two bytes are a so called artificial prefix. The CHESS

compiler removes this prefix at code emission. All other tools only see the eight bytes of the branch and jump instructions. See [1] for details.

The size of the artificial prefix is specified in the comiler header file `{trvXXpY}_chess.h`:

```
artificial_prefix_bits : 16;
```

It is needed to avoid warnings about overlapping instruction encodings.

Note that the condition codes are complemented: the functional opcode `bne` is implemented with the primitive `eq` for equal comparisons.

The `chess_pc_offset` attribute is six bytes (regular branch: zero). The offset needs to be relative to the jump instruction, which starts at the sixth byte of this pseudo instruction (skip the four bytes of the branch and the two bytes of the artificial prefix).

Note that instruction words are little endian, hence the order of the bytes seems reversed. The instruction is ten bytes wide, i.e., 80 bits. Bits 7 to 0 correspond to byte 0, bits 15 to 8 to byte 1. In the memory, due to the little endianess, byte 0 comes first. In the instruction image, MSBs are on the left, LSBs are on the right, i.e., the image has the bit order 79..0.

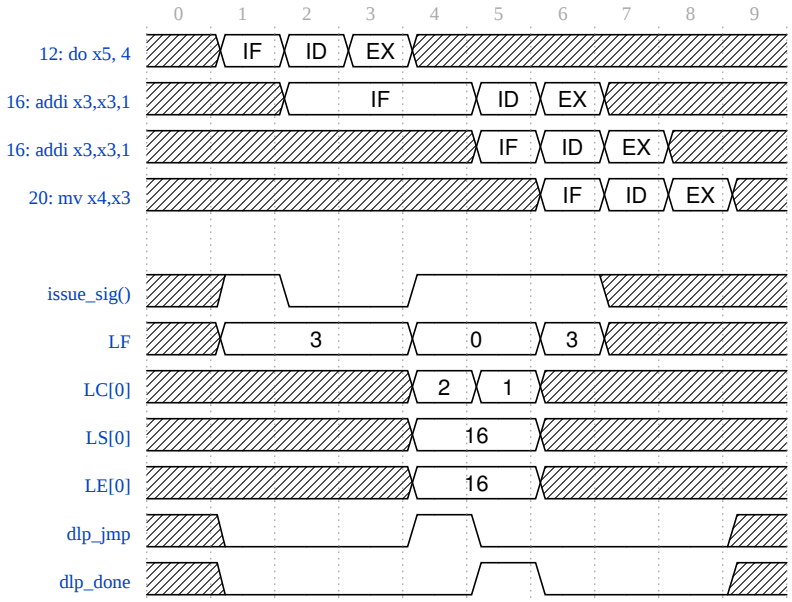
9.7 Zero Overhead Loops

The C compiler sees a primitive labeled `hwdo`, which is annotated with `relative doloop`. It uses this primitive when it wants to setup a zero overhead loop. The `hwdo` primitive is enabled by the `do` instruction.

```
enum eDLP {
  do     = 0b000,

  THREE  = 0b111
};

// ~~~ Do-loop setup instruction
// do rs1, imm
//
// rs1: loop count
// imm: loop end offset, relative to instruction

opn do_instr(op: eDLP, rs1: eX, i: c13u_s2)
{
  action {
#ifndef __programmers_view__
  stage ID:             pid_PC = tid_PC = trPC_ID = pif_PC;
#endif
  stage EX:             dlp_lc = r1 = X[rs1];
                        if (op: do) { hwdo(dlp_lc,dlp_le_offs=i); }
#ifndef __programmers_view__
                        dlpA  = trPC_EX = pid_PC;
                        dlpB  = dlp_le_offs;
                        dlpLE = add  (dlpA,dlpB)   @dlp;
                        dlpLS = inc4 (dlpA)        @dlp; // our addr + 4
                        LF = lfw = inc1 (lfr=LF)   @dlp;
                        LE[lfw] = lew = dlpLE;
                        LS[lfw] = lsw = dlpLS;
                        LC[lfw] = lcw = dlp_lc;
#endif
  }
  syntax : op PADMNM " " rs1 "," PADOP1 i;
  image  : i[12..1 zero]::rs1::op::"00000",
           cycles(3), class(ctrl), class(doloop);
}
```

Parts of the `action` attribute are visible only to ISS and HDL models. These parts compute the loop start and end addresses. The addresses, alongside with the loop count, are written into the `LS`, `LE` and `LC` register files. All computations are performed on the functional unit `dlp`. The instruction is annotated with `cycles(3)`. The setup is done in the `EX` stage. The end-of-loop check is done for instructions which are currently in the `IF` stage and will be issued. The registers writes need to be finished before the end-of-loop check is performed.

The timing diagram below shows a simple single-instruction loop.



9.8 Special Register Field `X[0]`

The register field `X[0]` is hardwired to zero. Reading `X[0]` always returns zero. Writing to `X[0]` should be ignored. These writes to the dead field should not be considered in the read-after-write and write-after-write hazard checks.

In order to model this, we redirect writes to this field to dead-end transitories. This is an example for the `alu`:

```
trn w1_dead <w64>;
property dead_end : w1_dead;

stage EX:              pex_D1 = tex_D1 = aluR;
stage ME:              pme_D1 = tme_D1 = pex_D1;
stage WB:
  if (rd: x0)          w1_dead = w1 = pme_D1;
  else                 X[rd] = w1 = pme_D1;
```

If the destination field is `X[0]`, here represented by the `x0` value of the enumeration type `eX`, then the output `aluR` of the `alu` unit is redirected to `w1_dead`. This `w1_dead` transitory is marked as `dead_end`, which is required to avoid errors about incomplete register transfers (checked by the tools).

Here, the writes are terminated in the `EX` stage. When the destination is `X[0]`, then the `alu` will be enabled, but the writes to the subsequent pipeline registers are disabled.

Reading from `X[0]` always returns zero. The register field is initialized to zero at reset:

```
hw_init X = others => 0;
```

It can not be updated, hence always keeps the value zero. A `chess_view` rule tells the compiler how to use `X[0]`:

```
chess_view() { r1 = zero; }
          -> { r1 = 0;    }
```

Section 5.1.2 provides more details.

9.9 Chess View Rules

The models contain several `chess_view` rules, which explain useful functional modes to the compiler.

- To load a signed 12-bit immediate into a register, use the `add` primitive and select `X[0]` as first operand register. This is the `addi` instruction. Due to the argument type of the `chess_view` statement, this matches only to the `addi` instruction's nML code.

```
chess_view (i:c12s) { aluR = add (aluA=zero,aluB=i); }
                  -> { aluR = i; }
```

- To load a signed 20-bit immediate into a register, right padded by 12 zeros, use the `add` primitive and select `X[0]` as first operand register. This is the `lui` instruction. Due to the argument type of the `chess_view` statement, this matches only to the `lui` instruction's nML code.

```
chess_view (i:c20s_rp12) { aluR = add (aluA=zero,aluB=i); }
                       -> { aluR = i; }
```

- We define a `seq0` primitive in terms of `sltu`:

```
chess_view () { aluR = sltu (aluA,aluB=1); }
           -> { aluR = seq0 (aluA); }
```

  Later, an intrinsic is promoted to this primitive, such that we can explicitly use this functionality in applications via a function call.

- We define a `sne0` primitive in terms of `sltu`:

```
chess_view () { aluR = sltu (aluA=zero,aluB); }
           -> { aluR = sne0 (aluB); }
```

- A plain unconditional relative jump is a relative call (jump-and-link) with discarded return address (write to `X[0]`:

```
chess_view () { w1_dead = jal (of21); }
           -> { j (of21); }
```

- A plain unconditional absolute jump is an absolute call (jump-and-link) with discarded return address (write to `X[0]`:

```
chess_view () { w1_dead = jalr (trgt); }
           -> { jr (trgt); }
```

  There is an `add` in front of the `jalr` primitive. We need to explain that a zero immediate offset for the `jalr` instruction can be used to pass through the register value:

```
chess_view () { trgt = add (aluA,aluB=0); }
           -> { trgt = aluA; }
```

- A nop (no-operation) is an `add` with zero inputs, and discarded output:

```
chess_view () { w1_dead = add (aluA=zero,aluB=0); class(alu_rri); }
           -> { nop(); }
```

  As per RISC-V ISA, the `addi` is preferred. The `class(alu_rri` restricts the pattern to the `addi` instruction.

- A register move can be done with the `add` primitive and choosing a zero as second operand:

```
chess_view () { aluR = add (aluA,aluB=0); class(alu_rri); }
           -> { aluR = aluA; }
```

  The `class(alu_rri)` specifier restricts the pattern to the `addi` instruction.

- An in-register zero extension of bytes is possible with `andi rd, rs1, 0xff`:

```
chess_view () { aluR = band (aluA,aluB=0xff); }
           -> { aluR = zext_08 (aluA); }
```

The `zext_08` primitive is used in the promotion rules for `unsigned char` upconversions.

- These rules explain direct and indirect addressing on the AGU:

```
chess_view () { aguR = add (aguA,aguB=0); }
           -> { aguR = aguA; }

chess_view () { aguR = add (aguA=zero,aguB); }
           -> { aguR = aguB; }
```

- These four rules explain the functional modes of the multiplier primitive to the compiler:

```
chess_view () { mpyR = mpyL; mul_hw (mpyA,mpyB,mpyMD=0b11,mpyL,mpyH); }
           -> { mpyR = mul    (mpyA,mpyB); }

chess_view () { mpyR = mpyH; mul_hw (mpyA,mpyB,mpyMD=0b11,mpyL,mpyH); }
           -> { mpyR = mulh   (mpyA,mpyB); }

chess_view () { mpyR = mpyH; mul_hw (mpyA,mpyB,mpyMD=0b10,mpyL,mpyH); }
           -> { mpyR = mulhsu (mpyA,mpyB); }

chess_view () { mpyR = mpyH; mul_hw (mpyA,mpyB,mpyMD=0b00,mpyL,mpyH); }
           -> { mpyR = mulhu  (mpyA,mpyB); }
```

- For the RV64 models, there is an additional rule that explains the 32-bit multiply:

```
chess_view () { mul_hw (mpyA,mpyB,mpyMD=0b11,mpyL,mpyH);
                mpyR = mpyW; mpyW = mpyL; }
           -> { mpyR = mulw   (mpyA,mpyB); }
```

- For the RV64 models, an in-register sign extension of a 32-bit word can use the `addw` primitive:

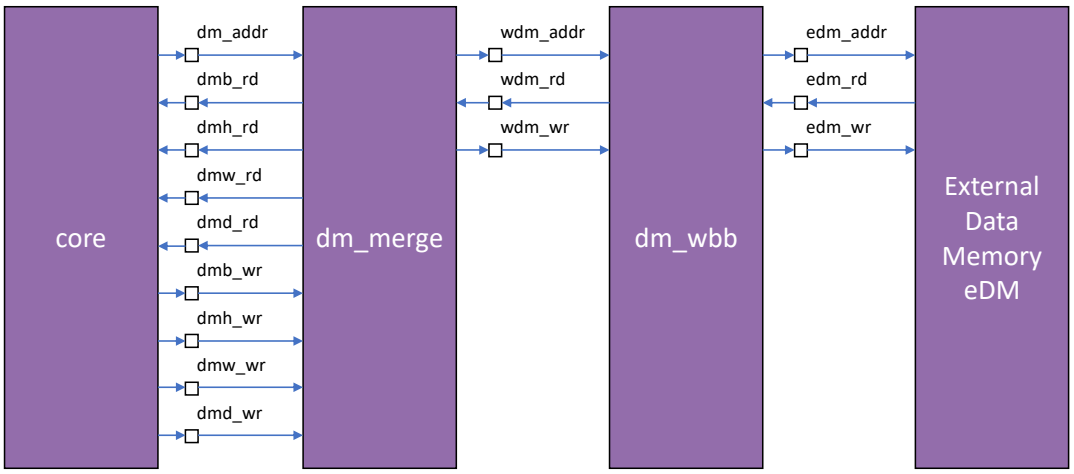
```
chess_view () { aluR = addw (aluA,aluB=0); class(alu_rri); }
           -> { aluR = sext_32 (aluA); }
```

10

IO Interfaces

The TRV models use three IO interfaces. Two are attached to the data memory interface. One is attached to the program memory interface.

This diagram shows the data memory interface chain:



The unit `dm_merge` in the file `dm.p` merges the byte, half-word, word and double-word (for RV64) accesses into a single (double-)word interface labeled `wDM`. It supports only aligned accesses. For accesses that are smaller than the maximum data width (32-bit for RV32 models, and 64-bit for RV64 models), tt translates these smaller accesses to the full-width interface. For stores, it sets byte write enable flags appropriately. Both the load result, and the store data, is appropriately shifted when necessary.

The unit `dm_wbb` aligns the memory store timing of the core to the store timing used by most SRAM macros.

The core uses a store timing identical to the load timing. The memory access, either load or store, is initiated by setting the load or store enable and driving the address bus. For stores, the core places the store data on the data bus in the next cycle. For loads, the memory places the load data on the data bus in the next cycle.

```
mem wDM [2**(DM_SIZE_NBIT-3)] <v8u8,addr> access {
  ld_wdm‘0‘: wdm_rd‘1‘           = wDM[wdm_addr‘0‘]‘1‘;
  st_wdm‘0‘: wDM[wdm_addr‘0‘]‘0‘ = wdm_wr‘0‘;
};
```

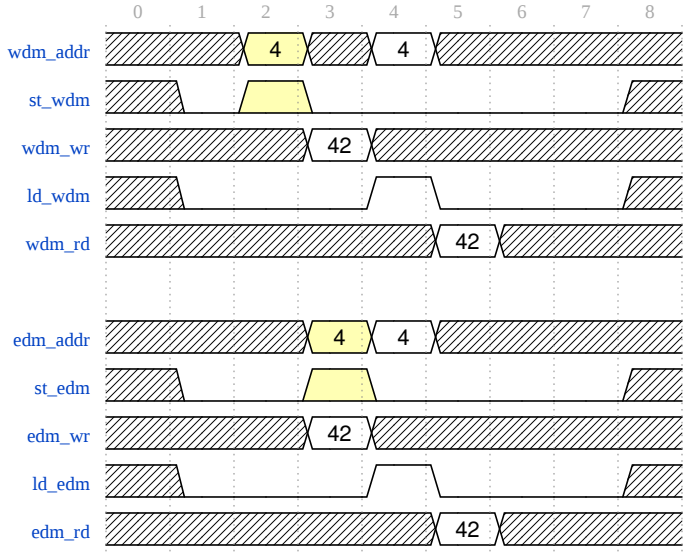
SRAMs typically use a different store timing. For stores, the core has to place the store data on the data bus in the same cycle as the address bus is driven, For loads, the memory still places the load data on the

data bus in the next cycle, after receiving the address.

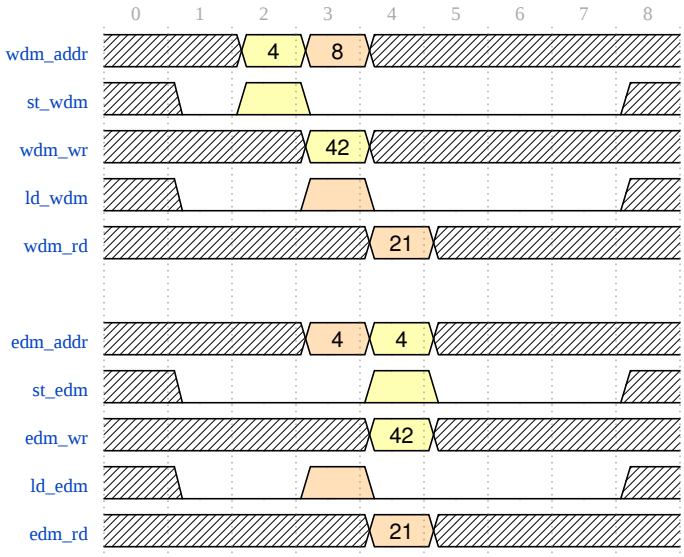
```
mem eDM [2**(DM_SIZE_NBIT-3)] <v8u8,addr> access {
  ld_edm‘0‘: edm_rd‘1‘           = eDM[edm_addr‘0‘]‘1‘;
  st_edm‘0‘: eDM[edm_addr‘0‘]‘0‘ = edm_wr‘0‘;
};
```

The dm_wbb unit appropriately realigns stores. With the help of a write back buffer, it synchronizes loads that follow stores, which would result in a resource conflict (data bus used in the same cycle). When a load follows a store, the pending store is delayed. If the load's address matches the pending store's address, then load is served from the write back buffer. Pending stores are issued to the memory in the next free cycle.

The timing diagram below shows a simple situation where a load follows a store with two-cycles distance. On the wDM side, both load and store exhibit the same timing as explained. On the eDM side, the store timing is changed. The store address and the store enable are delayed for one cycle, such that store data, address and enable are sent together in the same cycle to the memory.



The next diagram shows a store-load sequence with a one-cycle distance, but different addresses. The store request on the wDM side is buffered until the eDM interface is available. On the eDM side, the load and store order is swapped. The interface delays the store address and store enable by one cycle, as explained before. In cycle 3, the load request on eDM takes precedence over the delayed store request from cycle 2. The store data and address are placed into a write-back buffer. In cycle 4, the eDM side is free to send the next request. The pending store from cycle is send to the memory. If the addresses of the store and load request are matching, then the load request would be partially or completely served from the write-back buffer.



The program memory interface chain is shown in the diagram below. The unit `pm` in the file `pm.p` simply translates the byte addresses of the core to word addresses for the memory. It cuts off the LSBs.

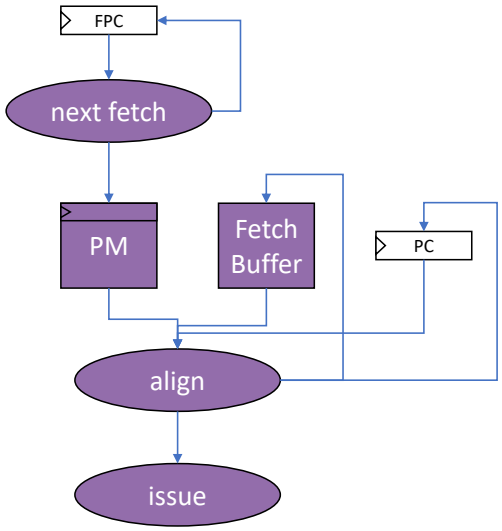
11

Processor Control Unit

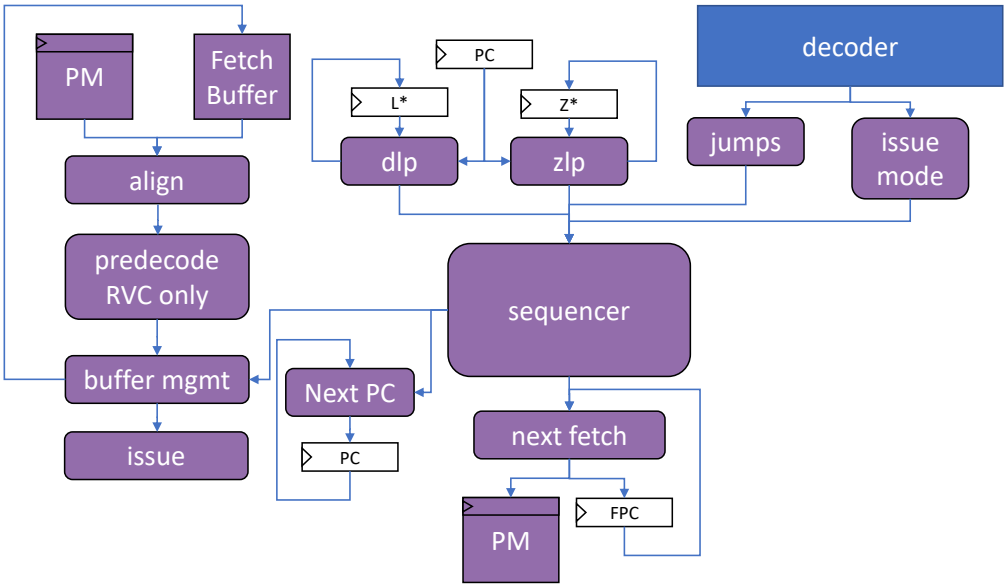
11.1 Overview

This unit coordinates fetching and issuing. It uses a fetch buffer. The buffer decouples the issue logic from the fetch logic, e.g., the length of the recently fetched instruction does not affect the next fetch address.

A general overview is shown in the diagram below. The fetch part determines the next fetch address. The align part uses the PM load result and the fetch buffer content to assemble the next instruction for issue. There are separate registers for the fetch program counter (`FPC`) and the issue program counter (`PC`).



More details are shown in the next diagram. It shows the various process and their interaction. Note that currently, all functionality is implemented in only two PCU functions: `user_next_pc()` and `user_issue()`. The split into functional parts is for visual purposes only.



On the top left, the *alignment* selects four bytes from the concatenation of the *PM* load result and the *fetch buffer* content. If compressed instructions are present in the model, a *predecode* function determines the length of the selected instruction - either two bytes or four bytes. The *buffer management* updates the fetch buffer. The selected instruction is then considered for *issue*.

From the *decoder* on the top right, the PCU obtains various signals indicating jumps, stalls etc. The *issue mode* part determines if the PCU can issue, or if it is stalled. The *jumps* part processes the signals that come from control flow instructions in the pipeline. It determines whether a control flow change is indicated.

The *dlp* and *zlp* parts handle zero-overhead loops with `do` instruction (*dlp*) and without `do` instruction (*zlp*). They have access to the respective nML registers for loop count, start, end addresses etc. The loop end addresses are compared to the `PC` value.

The *sequencer* in the middle coordinates all units. It determines if and what should be issued into the pipeline. This decision is communicated to the *buffer management*, the *next-PC* logic and the *next-fetch* part. The *buffer management* consumes the selected instruction if it is issued. The *next-PC* part advances the `PC`. The *next-fetch* part determines the next fetch address. There can be a control flow change (non-linear update due to jumps), or a linear advance in case of a sequential control flow. For the linear progress, the process always fetches at `FPC+4`, independent of the length of the issued instruction.

12

Compiler Model

The compiler header file `{trvXXpY}_chess.h` describes the application layer of the TRV models.

12.1 Type System

The RV32 models use the ILP32 C language data model. The RV64 models use the LP64 C language data model.

Built-in Type	RV32	RV64
`char`	8 bit	8 bit
`short`	16 bit	16 bit
`int`	32 bit	32 bit
`long`	32 bit	64 bit
`long long`	64 bit	64 bit
`pointer`	32 bit	64 bit

The models use the following approach for the type system:

- Signed C built-in types are represented by signed primitive types of same size.
- The types `unsigned short` and `unsigned char` are promoted to signed primitive types of same size.
- The type `unsigned int` is represented by a signed 32-bit primitive type.
- For RV32, `unsigned long long` is emulated by a dual-int struct type.
  For RV64, `unsigned long long` is represented by a signed 64-bit primitive type.
- For RV32, `(unsigned) long` is represented by `(unsigned) int`.
  For RV64, `(unsigned) long` is represented by `(unsigned) long long`.
- The `char` type is signed.
- When placed in a wider storage, `char` and `short` are sign extended.
- When placed in a wider storage, `unsigned char` and `unsigned short` are zero extended.
- For RV64, when placed in a wider storage, both `int` and `unsigned int` are sign extended.
- Smaller primitive types are promoted to larger primitive types. Sign extension is used (all types are signed), paths with zero extension are excluded where necessary.
- Pointers are represented by the largest signed primitive type.

The floating-point types `float`, `double` and `long double` are fully supported.

1. The single-precision type `float` is 32 bit wide and follows the IEEE754 standard.
2. The double-precision type `double` is 64 bit wide and follows the IEEE754 standard.

3. The extended-precision type `long double` is represented by `double`.

All floating-point types are emulated. Floating-point operations are mapped on calls to the `SoftFloat` library, for the CHESS front-end, or use LLVM's own emulation layer, which is part of the `compiler-rt` library.

Bibliography

[1] *Chess Compiler User manual*. Synopsys, March 2020. Version Q-2020.03.

[2] *The nML Processor Description Language*. Synopsys, March 2020. Version Q-2020.03.

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