Microarchitecture overview of RISC-V Processors (Free and Open-Source) from Bluespec, Inc.

at https://github.com/bluespec/Piccolo, Flute, Toooba, ...

Rishiyur S. Nikhil
Bluespec, Inc.

Version: January 20, 2020

Acknowledgements

... *To be written* ...

Contents

1 Introduction, and Simulation Top-level **1-1**

1.1 Introduction . . . 1-1

1.1.1 Purpose of this document/target audience(s) . . . 1-2

1.2 Common "system" for all three processors . . . 1-2

1.3 Common directory organization . . . 1-4

1.4 Pre-generated Verilogs, regenerating them, and generating new configurations . . . 1-5

1.5 Using just the core in your own designs . . . 1-5

1.6 Using SoC in your own designs, with these or other cores . . . 1-6

2 Common `mkCore` structure for Piccolo and Flute **2-1**

2.1 Optional Debug Module . . . 2-2

2.2 Optional Tandem Verification Trace Generation . . . 2-2

3 Piccolo **3-1**

4 Flute **4-1**

4.1 "Near Memory" . . . 4-2

4.2 Optional Debug Module interface . . . 4-2

4.3 Optional Tandem Verifier trace output interface . . . 4-3

5 mkCore structure for Toooba **5-1**

6 Toooba **6-1**

Chapter 1

Introduction, and Simulation Top-level

1.1 Introduction

Bluespec, Inc. provides three free, open-source RISC-V processors on GitHub (with more to come in the future).

- **Piccolo**: simple 3-stage, in-order.
  `https://github.com/bluespec/Piccolo`
- **Flute**: simple 5-stage, in-order, some control-flow speculation and branch prediction.
  `https://github.com/bluespec/Flute`
- **Toooba**: aggressive superscalar, out-of-order, agressive speculation and branch prediction.
  `https://github.com/bluespec/Toooba`

All three are Linux-capable (can be built for RV64IMAFD with Machine and Supervisor privilege levels; Sv39 virtual memory; and external, software and timer interrupts).

Piccolo and Flute are highly parameterized and can also be built with smaller capabilities (e.g., embedded, IoT). In particular, you can choose any of the following ISA options:

- RV32I or RV64I
- M (integer multiply/divide)
- A (atomics)
- F (single-precision floating point)
- D (double-precision floating point)
- C (compressed instructions)
- S (Supervisor privilege level), with Sv32 virtual memory for RV32 and Sv39 or Sv48 virtual memory for RV64
- U (User privilege level)

Piccolo and Flute have separate instruction and data memory channels, with L1 caches where you can choose cache sizes and organization.

Toooba is RV64IMAFDC with Sv39 virtual memory, and multicore. It has many parameter choices such as degree of superscalarity, L1 cache size and organization, L2 cache size and organization, reorder buffer size, number of arithmetic, floating point and memory pipelines, size of store buffers, memory model, number of cores, etc.

1.1.1 Purpose of this document/target audience(s)

This document is intended for people trying to use the pre-generated Verilogs in the repositories, and for people trying to understand the BSV source code in the repositories with a view to regenerating the Verilogs, perhaps with custom changes. It is intended to provide top-level context for the code and code structure so that you can navigate the directories and read the code and/or understand how to use the pre-generated Verilog.

Here are several possible use models for the code in these repos:

- Take the pre-generated Verilogs for the RISC-V CPUs, as-is, for use in the user's own SoC. The SoC provided in these repos allows developing and debugging software immediately even if the user's SoC is not yet ready.
- Choose different parameters (relative to the pre-generated Verilogs) for the CPUs, and regenerate the Verilogs, for use in the user's own SoC. E.g., choose a different set of RISC-V options (RV32/RV64, A, M, F, D, C, with/without Supervisor and virtual memory, ...)
- Modify the BSV code for the CPUs for the user's customizations, and re-generate the Verilogs. E.g., add new custom instructions.
- Use the system here as a "socket" to plug in the user's own RISC-V CPU implementation, giving the user a system that can run software out of the box with full debugging support, even if the user's SoC is not yet ready.
- Use the CPU and SoC here as is, just adding extra AXI4 ports on the system interconnect to connect the user's own memory-mapped accelerator. The user can develop software for the accelerator and test it immediately, with full debugger control.

This document does not attempt to explain the BSV High Level Hardware Design Language. For that, please refer to language manuals and tutorials at `https://github.com/BSVLang/Main/`.

This document does not attempt to do a detailed code explanation of the BSV code in the repositories. Rather, this document is only meant to provide you high level context and structure so that you can, on your own, easily navigate through the directories and files.

1.2 Common "system" for all three processors

The GitHub repositories for Piccolo, Flute and Toooba include a common "system" environment for the processors so that they can execute RISC-V binaries out of the box, using Verilog simulation. Most of this system is also synthesizable for FPGA or ASIC.

Fig. 1.1 illustrates the structure of the common top-level system.

Notation in figure: Nested rectangles indicate module hierarchy. Here, module `mkTop_HW_Side` instantiates module `mkSoC_Top` with instance-name `soc_top` and instantiates module `mkMem_Model`

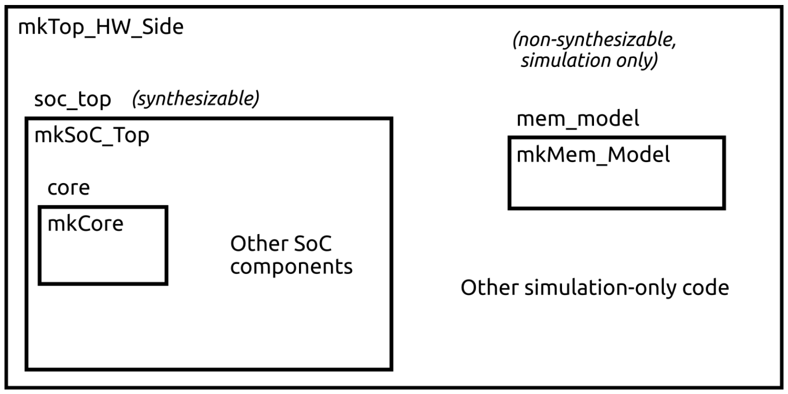

Figure 1.1: The top-level of the module hierarchy of the system.

with instance-name `mem_model`. Module `mkSoC_Top`, in turn, instantiates module `mkCore` with instance-name `core`.

Fig. 1.2 expands the `mkSoC_Top` module for more detail.

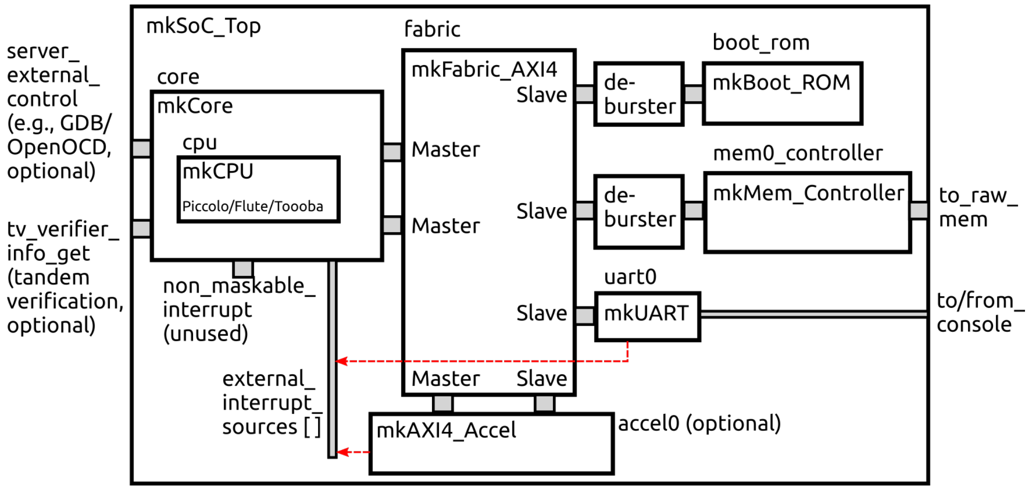

Figure 1.2: The common SoC structure shared in all repos.

At the center is an AXI4 fabric (interconnection network) that connects a parameterizable number of master ports to a parameterizable number of slave ports. Two of the master ports are used by `mkCore` inside which is the Piccolo/Flute/Toooba CPU. The slave ports are connected to a Boot ROM, a memory and a UART, and optionally to one or more memory-mapped accelerators.

The core can optionally be controlled by an external debugger (e.g., GDB and OpenOCD). It can optionally send out a detailed instruction-by-instruction trace for "Tandem Verification" with a golden reference model of a RISC-V CPU. It can accept external interrupts from I/O devices. The two master ports into the fabric are peers and can be used in different ways. In Piccolo and Flute, one master port is used for all instruction traffic and the other is used for all data (memory and I/O traffic). In Toooba, one master is used for all memory traffic (instruction and data) and the other is used for all I/O traffic.

The Boot ROM and Memory-Controller modules do not handle AXI4 bursts; the "deburster" modules convert burst requests from and responses to the fabric into individual request/responses to the modules behind them. The Memory Controller has an interface taken all the way out of `mkSoC_Top` to an external DRAM controller of 256-bits of 512-bits width.

The UART has an interface taken all the way out of `mkSoC_Top` streaming bytes in and out.

The core and the CPU have a "non-maskable interrupt" input interface that is not used in this SoC (in a real chip, they might be connected to sensors signaling power failure, temperature overheating, etc.).

There are two versions of `mkCore`, one shared by Piccolo and Flute, and the other for Toooba. The former is described in more detail in Ch. 2, and the latter in Ch. 5. In addition to the CPU pipelines themselves, the cores include:

- I- and D-caches;
- TLBs and virtual-memory management (including hardware page-table walkers);
- a set of "near-memory I/O" components such as a memory-mapped real-time timer, and memory-mapped software-interrupt location;
- a PLIC (Platform Level Interrupt Controller);
- an optional Debug Module;
- and an optional Tandem Verification output trace encoder.

The CPU pipelines themselves are described separately for Piccolo in Ch. 3, Flute in Ch. 4, and for Toooba in Ch. 6.

Optionally, one can have one or more memory-mapped accelerators connected to extra master and slave ports. Typically, a memory-mapped accelerator is programmed/configured and "started" by the processor using writes through a fabric slave port. The accelerator, once running, accesses memory directly using reads and writes through a fabric master port. Completion of the acceleration task is detected by the processor either by polling (reads) on the slave port, or by receiving an interrupt from the accelerator.

1.3 Common directory organization

In general, a module `mkFoo` can be found in a file `Foo.bsv`.

All the repositories also follow a common directory organization:

- The directory `src_Core/` contains `mkCore` (in `file Core.bsv`) and everything inside it, including the CPU pipelines, L1 caches, PLIC (Platform Level Interrupt Controller), Debug Module, and Tandem Verification trace generator.
- The directory `src_Testbench/SoC/` contains `mkSoC_Top` and other SoC components.
- The `src_bsc_lib_RTL` contains copies of a few Verilog library files used by the Bluespec *bsc* compiler, and can be treated here as black boxes.
- The `src_SSITH_P1/P2/P3` directories can be ignored; they contain different wrappers of `src_Core` intended for the DARPA SSITH project and are not otherwise relevant.

Linux/Unix users will be familiar with the standard 'find', which is useful to locate a file with a given name in the directory tree below the current directory:

```
   $ find . -name <filename>
```

Similarly, the standard program 'grep' is useful to locate a file containing a specific substring in the directory tree below the current directory:

```
   $ grep -Rn <string>  .               # Case sensiitive
   $ grep -iRn <string>  .              # Case insensitive
```

1.4 Pre-generated Verilogs, regenerating them, and generating new configurations

The repositories contain pre-generated Verilog for certain configurations as subdirectories of the `builds/` directory. For example, in the Piccolo repository, the directory

```
    builds/RV64ACFDIMSU_Piccolo_verilator
```

is for building Piccolo for RV64I with ISA options ACDFIMSU for verilator. The subdirectory `Verilog_RTL/` has the pre-generated Verilog for this.

If you have an installation of the Bluespec BSV *bsc* compiler, you can re-generate the Verilogs, or build a Bluesim simulation, or generate other configurations (see the README in the repo for how to compile and build).

1.5 Using just the core in your own designs

In Fig. 1.1, `mkCore` is the basic core: CPU pipeline, L1 caches, PLIC (Platform Level Interrupt Controller), and optional Debug Module and Tandem Verification trace generator.

For Piccolo and Flute, in the pre-generated Verilog directories (see Sec. 1.4), the code is in file `mkCore.v` and other Verilog files representing the module hierarchy below it.

For Toooba, in the pre-generated Verilog directories, the code is the file `mkCoreW.v` and other Verilog Verilog files representing the module hierarchy below it.

The interface of the core module is the same for all three CPUs:

- `CLK` and `RST_N` are the clock and reset inputs.
- The buses named `cpu_imem_master_*` are an AXI4 master interface.
- The buses named `cpu_dmem_master_*` are an AXI4 master interface.
- The following buses are useful in simulation for debugging:

- `set_verbosity_verbosity`
- `set_verbosity_logdelay`
- `EN_set_verbosity`
- `RDY_set_verbosity`

When the `RDY` output is high, the enviroment can assert the `EN` input and drive the other two inputs, setting the cycle delay after which the verbosity should change, and the verbosity value to which it should change. Verbosity of 1 will print an instruction trace; higher values will print more detail from the CPU pipeline.

- The 'reset' signals are for a 'soft' reset of the core to its initial state (the same state as after the electrical reset `RST_N`). They come in two groups representing a request/response protocol, since a full reset of the core can take multiple cycles.

  When output `RDY_cpu_reset_server_request_put` is 1, the environment can request a reset by asserting `EN_cpu_reset_server_request_put` and driving a 0 or 1 on `cpu_reset_server_request_put`. Driving a 1 specifies that, after the reset, the CPU should come up running (the normal case). Driving a 0 specifies that the CPU should come up halted in Debug Mode, waiting for commands from an external debugger.

  When output `RDY_cpu_reset_server_response_get` is 1, it indicates that the core has reset itself. The environment can read `cpu_reset_server_response_get` to see if the core is running (1) or halted (0). The environment can assert `EN_cpu_reset_server_response_get` to acknowledge to the core that it has observed this information.

- The core contains a standard RISC-V PLIC (Platform Level Interrupt Controller) supporting 16 external interrupts at the "m" (machine) privilege level. These are the 16 input signals of the form: `core_external_interrupt_sources_`*j*`_m_interrupt_req_set_not_clear`

- The core supports a *non-maskable interrupt* on the input signal `nmi_req_set_not_clear`, intended for very urgent interrupts such as power failures, thermal overload, etc.

1.6 Using SoC in your own designs, with these or other cores

In Fig. 1.1, `mkSoC_Top` is a basic SoC (System-on-a-chip).

In all the repositories, in the pre-generated Verilog directories (see Sec. 1.4), the code is in file `mkSoC_Top.v` and other Verilog files representing the module hierarchy below it. The SoC contains:

- The RISC-V core, described in the previous Sec. 1.5, is in file `mkCore.v`. You can use this `mkCore.v` or substitute your own core with the same Verilog interface.
- An AXI4 crossbar fabric (file `mkFabric_AXI4.v`).
- A boot ROM (file `mkBoot_Rom.v`) with an AXI4 interface (without burst support).
- A memory controller, front end for a DRAM controller (file `mkMem_Controller`) with and AXI4 interface (without burst support).
- An AXI4 to AXI4 slave adapter, instantiated twice, once for the Boot ROM and once for the memory controller (file `mkAXI4_Deburster_A.v`), so that they can support burst AXI4 requests.
- A synthesized model of an NS16550 UART (file `mkUART.v`).

Chapter 2

Common mkCore structure for Piccolo and Flute

The **mkCore** module for Piccolo and Flute in Fig. 1.2 is shown in more detail in Fig. 2.1.

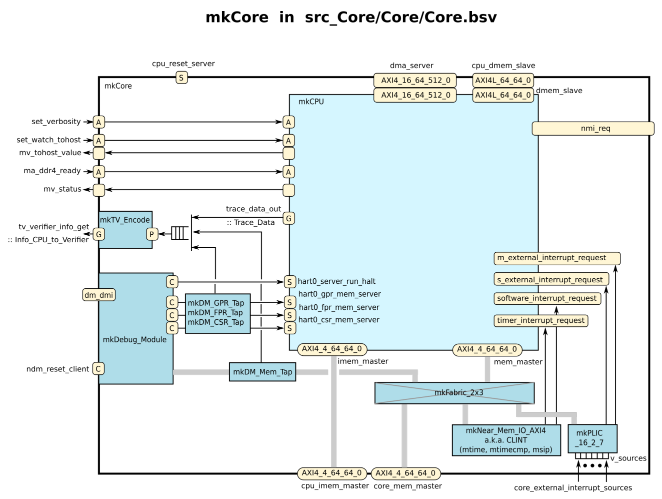

Figure 2.1: The common mkCore for Piccolo and Flute.

The **mkCPU** module is the Piccolo or Flute CPU with L1 caches and MMUs. The interface is identical for Piccolo and Flute. In the figure, on the right, it has two AXI4 interfaces. One of them goes straight out to the **mkCore** interface for instruction access as the **cpu_imem_master** interface; this is an AXI4 master. The other, for data memory and I/O access, is also an AXI4 master, but it connects to a 2x3 interconnect fabric. The 2x3 fabric has three AXI4 slaves:

- **mkNear_Mem_IO_AXI4** contains "nearby" memory-mapped locations, such as the RISC-V standard real-time timer (MTIME), timer-compare (MTIMECMP) and software-interrupt (MSIP). Interrupts generated from these, in turn, are connected back to mkCPU (red arrows).
- **mkPLIC**, a standard RISC-V Platform Level Interrupt Controller. It accepts a vector of external interrupt sources and arbitrates them, finally feeding the winning interrupt(s) into **mkCPU**.

- A direct connection out to the main system interconnect, `cpu_dmem_master` for all other data memory and I/O device access.

2.1 Optional Debug Module

An optional RISC-V standard Debug Module `mkDebug_Module` has connections into `mkCPU` for run-control (halt, resume) and access to GPRs, FPRs and CSRs. It is also a second master on the `mkFabric_2x3` so that it can access all memory and I/O devices. On the left of `mkDebug_Module` has a standard "DMI" (Debug Module Interface) which is a memory-like read/write interface by which an external debugger interacts with the Debug Module. A typical setup would be:

$$\text{GDB} \Longleftrightarrow \text{OpenOCD} \Longleftrightarrow \text{JTAG transport} \Longleftrightarrow \text{DMI}$$

Source code in the `src_SSITH_Pn/src_BSV/` directory is available for the JTAG connection to OpenOCD.

The Debug Module gives GDB full control over the RISC-V CPU, including ↑C (halt a running program); setting/removing breakpoints; reading and writing memory, GPRs, FPRs and CSRs (including the PC); single-stepping; resetting the CPU; and resetting the system.

2.2 Optional Tandem Verification Trace Generation

The optional `mkTV_Encode` module sends out an instruction-by-instruction trace to an external recorder/analyzer. Information associated with each instruction includes the PC, the instruction itself, updates to GPRs/FPRs/CSRs, updates to memory and the next PC. For memory instructions it also includes the effective address.

This output trace can be compared with an expected trace on a "golden reference model"; a divergence typically reveals a bug in the hardware implementation for some particular kind of instruction.

Chapter 3

Piccolo

... *To be written* ...

Chapter 4

Flute

Fig. 4.1 shows the major module structure inside `mkCPU` for Flute. At the heart is a 5-stage

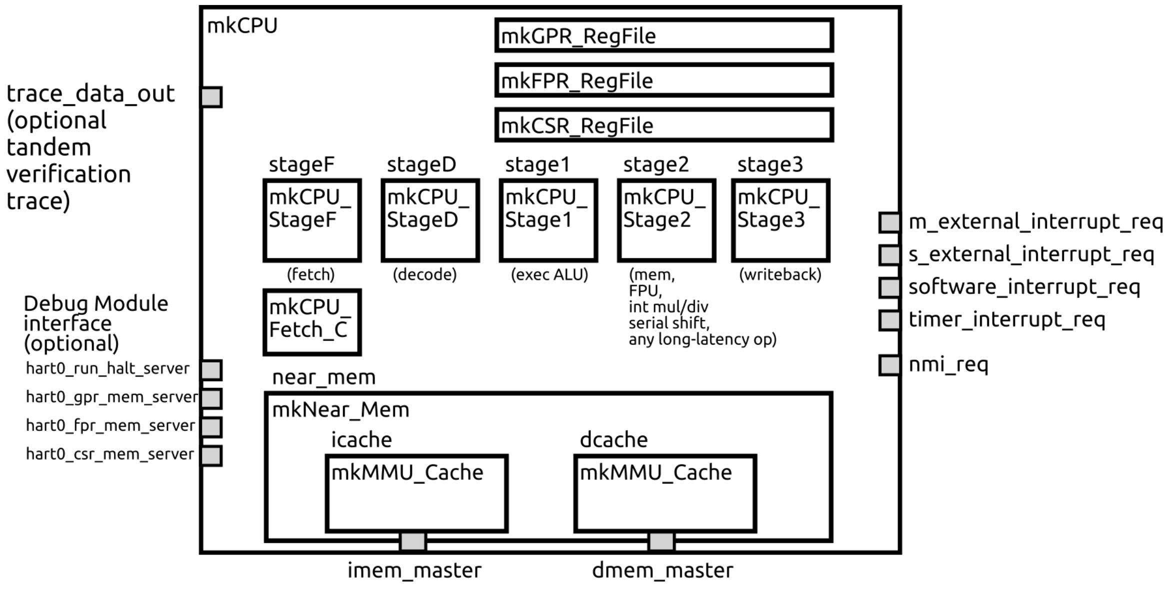

Figure 4.1: Flute module structure.

pipeline embodied in the following modules:

- `stageF`, an instantiation of `mkCPU_StageF`. This is the Fetch stage. The Fetch stage includes a branch predictor (branch target buffer and return-address stack). If the CPU is built with the RISC-V C extension (Compressed instructions), an additional module `mkCPU_Fetch_C` is interposed in front of the instruction-memory that decides whether the next instruction delivered is a C instruction or a normal RV32/RV64 instruction.
- `stageD`, an instantiation of `mkCPU_StageD`. This is the Decode stage. If the CPU is built with the RISC-V C extension (compressed instructions), the decode stage includes expansion of C instructions to their normal RV32/RV64 counterparts.
- `stage1`, an instantiation of `mkCPU_Stage1`. This is the Execute stage for all single-cycle ALU operations. Conditional branches and jumps are resolved here, and results are fed back to StageF to redirect on mispredictions and to train the Branch Predictor.

- `stage2`, an instantiation of `mkCPU_Stage2`. This stage has parallel paths for all potentially long-latency operations including memory access (load, store, atomics), floating point (RISC-V F and D extensions), integer multiply/divide (RISC-V M extension), optional serial shifter (instead of a barrel shifter in Stage 1), etc.
- `stage3`, an instantiation of `mkCPU_Stage3`, the writeback stage where final values are written back to the GPR and FPR register files.

The reason for the somewhat unusual naming scheme (instead of stages 1..5), is because of the code-sharing structure with Piccolo, a 3-stage pipeline. Piccolo's Stage 1 is essentially an expansion of Flute's Stage F, D and 1. Piccolo's Stage 2 and 3 are the same as Flute's Stage 2 and 3, respectively.

The GPR and FPR register files are read in Stage 1 and written back in Stage 3. There is bypass logic to feed back output values from Stage 2 and Stage 3 to the register-file read logic in Stage 1.

CSR instructions, instructions that trap, and interrupts are handled specially, outside normal pipeline flow. For CSR instructions, interrupts and traps in Stage 1, they wait in Stage 1 until the downstream stages (Stages 2 and 3) are empty, and are then handled in an FSM outside the pipeline. For instructions that trap in Stage 2 (e.g., memory access faults, protection faults, misaligned faults, unimplemented addresses), they wait until the downstream stage (Stage 3) is empty, and is then handled in an FSM outside the pipeline. In all these cases, the Fetch stage is redirected (restarted) after the instruction.

There are two external interrupt inputs, feeding the MEIP (external interrupt pending at machine privilege) and SEIP (external interrupt pending at supervisor privilege) bits of the RISC-V MIP CSR. There is one input for software interrupts (MSIP) and timer interrupts (MTIP). There is also one input for non-maskable interrupts.

4.1 "Near Memory"

Inside `mkCPU` we instantiate a "near memory" subsystem, `mkNear_Mem`. This is a separate module that implements L1 caches, MMUs and virtual memory support, but it can be replaced by an alternative implementation such as a fixed latency TCM (Tightly Coupled Memory implemented in SRAM).

Inside this module we have two instantiations of `mkMMU_Cache`, one for instruction memory (`imem`) and one for data memory and I/O accesses (`dmem`). The current instantiated module `mkMMU_Cache` is parameterized for size and associativity, and has a "write-through-no-allocate" (write-hits are performed in the cache and sent to memory; write-misses are only sent to memory).

4.2 Optional Debug Module interface

The optional Debug Module interfaces connect to a Debug Module outside `mkCPU`, allowing full GDB control of the CPU (see Sec. 2.1).

4.3 Optional Tandem Verifier trace output interface

The optional `trace_data_out` interface connects to Tandem Verifier Trace Encoder outside `mkCPU` (see Sec.2.2).

Chapter 5

mkCore structure for Toooba

... *To be written* ...

Chapter 6

Toooba

... *To be written* ...