RISC-V — Architecture and Interfaces
The RocketChip

Moritz Nöltner-Augustin
University of Heidelberg, ZITI

***Abstract*—This paper gives a short overview of the RocketChip system-on-chip generator and how to use it.**

***Index Terms*—RISC-V, RocketChip, Boom, SoC generator.**

I. Introduction

COMPUTER technology has seen the rise and fall of many instruction set architectures (ISAs) over time. Proprietary ISAs not only lead to fragmentation of the CPU market as companies cannot easily use the already existing ISAs of their competitors, but are also vulnerable to extinction when their proprietor companies run into financial troubles. Along with design complexity and licensing issues, these considerations led Krste Asanović et al. in 2010 to decide upon creating a new and free ISA for their next round of research projects at the University of California Berkeley (UCB). This fifth reduced instruction set ISA developed at UCB, called RISC-V[1], is – unlike its predecessors – not only meant for teaching but also actual implementation. With three supported word-widths (32, 64 and 128 bits[2, Time: 17:06]), RISC-V is aimed at all possible computational environments ranging from small embedded systems up to full scale supercomputers. To facilitate widespread adoption, the ISA is licensed permissively, allowing use for academic and commercial use in open- and closed-source designs free of charge and now, roughly 6 years after its inception, RISC-V is used in a number of roles:

- The LowRISC project aims to become the "linux of the hardware word"[3]
- SiFive[4] and OnChip are creating custom silicon products[5]
- ETH Zurich and Università di Bologna cooperate to create a state-of-the-art low-power parallel processor (PULP) and a single-core microcontroller derived from that processor (PULPino)[6]
- India has decided that RISC-V is the new "natinal ISA" including defense systems[2, Time: 46:55]), IIT-Madras is developing a range of processors for that[7]
- NVIDIA will use the RISC-V ISA for the replacement of their Falcon processor[8]
- A number of vendors create commercial soft and hard implementations[9]
- UCB uses RISC-V processors for research[10] and teaching purposes[11]

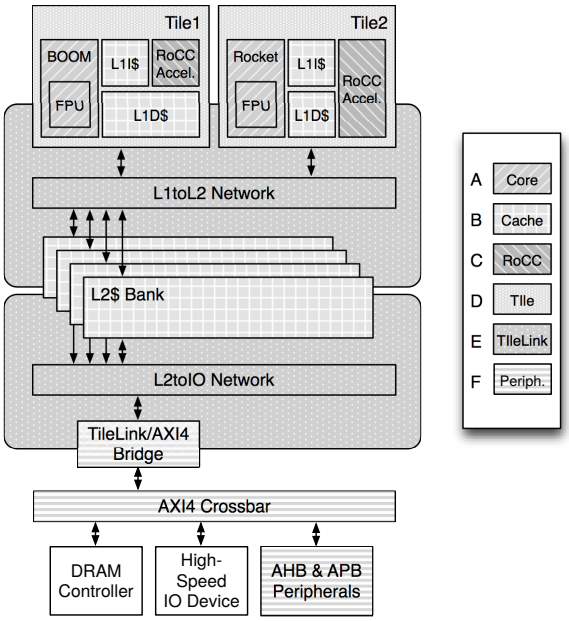

Figure 1. Overview of the components of RocketChip. Source: Image taken from [12].

II. RocketChip

The UCB currently develops three lines of 64-bit[1] RISC-V processors, Sodor[11], a collection of simple processors for teaching, Rocket[12, p.4], a Scalar in-order processor, and Boom[13], a superscalar out-of-order processor for high-performance applications. All three are written in Chisel (Creating Hardware in a Scala Embedded Language), a hardware description language – as the name implies – embedded in Scala. Chisel allows compilation into three distinct target formats: a cycle accurate C++ model for fast simulation and software development and verilog code aimed at implementation either in an FPGA or ASIC. For Rocket and Boom there is a generator framework that can create a whole parametrised system around the processor cores, called RocketChip[12]. It will create one or more Rocket, Boom or custom cores, surround it with caches, optionally an FPU and accelerators to form one or more [compute-]tiles. This assembly is then completed with an interconnect and other components to form a complete CPU. Figure 1 shows a schematic of the generated system.

[1]The RV64I instruction `ADDIW`[1, p.28] is present in the decoding tables[14].

Listing I
BUILD STEPS FOR ROCKETCHIP.
SOURCE: COLLECTED FROM [15] AND [16].

```
# $RC is some place in the file system
cd $RC
git clone \
https://github.com/ucb-bar/rocket-chip.git
cd rocket-chip
git checkout boom # Only to build Boom
git submodule update --init --recursive
```

(a) Downloading and initialising.

```
cd $RC/riscv-tools
export RISCV=/where/to/install/toolchain
export PATH="${PATH}:$RISCV/bin"
./build.sh # Takes about 45 min
```

(b) Building the RSCV-toolchain.

```
# Substitute ExampleSmallConfig with
# any available configuration.
# Use e.g BOOMConfig when building a Boom
cd $RC/emulator # Build the C++ simulator
make run CONFIG=ExampleSmallConfig
cd $RC/vsim     # Create the verilog code
make -jN CONFIG=ExampleSmallConfig
```

(c) Building the SOC and a simulator.

A. General Usage

The RocketChip is maintained in a Git repository[15]. To build the default-configuration RocketChip, one only has to clone the repository, build the included GCC toolchain for RISC-V, and run "`make`" as shown in listing Ia. For building a Boom-chip, the steps are identical except that one also has to checkout the "`boom`" branch of the same repository which then includes the Booms' repo[16] as a submodule.

If the environment variable `$RISCV` is not set to point to a sane installation of the toolchain, the following build steps will fail, so the next part is building the RISC-V-toolchain as shown in listing Ib.

B. Configurations

1) Building an existing Configuration: The makefile has a variable named "`CONFIG`" which is set to the name of a configuration. A configuration is a set of configuration options that define the generated system. Available configuration options can be found in `$(RC)/src/main/scala/coreplex/Configs.scala`, and include for example:

- Which core[s] (Rocket | BOOM | Custom)
- #Tiles/Cores
- Which Cache configuration (Associativity, ECC, replacement policy)
- Which Debug hardware (#Breakpoints, #Performance Counters)
- Which floating point unit
- Which multiplication and division logic
- Implement atomic instructions
- Which Bootrom
- Which TileLink configuration

These options are grouped into different build configurations which can be found in `$RC/src/main/scala/rocketchip/Configs.scala`. Building one of the pre-defined configurations is as simple as setting the `CONFIG` variable while `make`-ing the project. Listing Ic shows the steps to build the emulator and generate the verilog code for a certain RocketChip configuration.

The generated verilog code is placed into a single file of around 11MB (for `DefaultConfig`) and includes around 300 verilog modules. However, the generated code does not only not use any generics like parameterised modules but also involves numerous verbatim copies of the same modules with only the name differing e.g. AsyncResetRegVec_1..57 (also in `DefaultConfig`). Furthermore, the verilog code uses distictive wires for every connection. While this point-to-point connection scheme probably simplifies the generator code and allows to immediately know which output is driving a certain signal, it renders the code very hard to read as signals can go by many different names although they are electrically connected either directly or through instantiated submodules[2].

2) Creating a new Configuration: RocketChip configurations are written as Scala code and inherit from the `Config` class defined in `$RC/src/main/scala/config/Configs.scala`, which sets up some functionality including the '++' operator that is used to add the configuration options, or one of its child classes which already have some options set. To evaluate the propagation of different configuration options into the generated verilog code, two new configurations differing in cache associativity and size were created as shown in listing IIb using the configurations options listed in listing IIa.

3) Analysis of configuration propagation: The instruction cache parameters were chosen for their simple observability. Unlike for example the TileLink configuration options, the cache parameters only affect one verilog module, ICache_icache, and are relatively simple to check by examining the SRAM memories that are instantiated. For the 2-way configuration, the core of the cache can be expected to be two 32x21 bit tag SRAMs and two 32x512 bit (32x64 bytes) data SRAMs[3]. Likewise one would expect four 64x21 bit tag SRAMs and four 64*256 bit (64*32 bytes) data SRAMs[3] for the 4-way configuration.

Looking at the generated code shown in listing III, it can be verified that SRAMS of the expected capacities are

[2]e.g. the clock signal in `ExampleRocketTop` which is input to the module by the name `clock`, is connected to (and on another place assigned to) the wire textttsocBus_clock, and assigned to `AXI4Fragmenter_1_clock`, `AXI4ToTL_1_clock`, `AsyncQueueSink_1_1_clock`, `AsyncQueueSource_1_1_clock`, `TLMonitor_40..52_clock`, `TLToAXI4_2..3_clock`, `TLWidthWidget_2_clock`, `bootrom_TLAtomicAutomata_clock`, `bootrom_TLFragmenter_clock`, `bootrom_TLWidthWidget_clock`, `bootrom_clock`, `coreplex_clock`, `extInterruptXing_clock`, `intBus_clock`, `l1tol2_TLSourceShrinker_clock`, `l1tol2_TLWidthWidget_clock`, `l2_clock` and `peripheryBus_clock`.

[3]Or, of course a configuration of smaller width but the same capacity

Listing II
OVERVIEW OF THE CONFIGURATION SYSTEM.

```
class BaseCoreplexConfig
        extends Config ((site, here, up) => {
  case CacheBlockBytes => 64
  case CacheName("L1D") => CacheConfig(
  nSets= 64,
  nWays= 4,
  rowBits= site(L1toL2Config).beatBytes*8,
  nTLBEntries= 8,
  cacheIdBits= 0,
  splitMetadata = false)
  ...
})
...
class WithL1ICacheSets(sets: Int)
    extends Config((site, here, up) => {
  case CacheName("L1I") =>
    up(CacheName("L1I"), site).copy(nSets = sets)
})

class WithCacheBlockBytes(linesize: Int)
    extends Config((site, here, up) => {
  case CacheBlockBytes => linesize
})
```

(a) Extract of some of the configuration options found in `$RC/src/main/scala/coreplex/Configs.scala`.

```
// Config with 2-way, 32 sets x 64 bytes/block cache
// tag: 21 bits, index: 5 bits, WS: 4 bits, BS: 2 bits
class DualCoreConfig2way extends Config(
  new WithNCores(2) ++ new WithL1ICacheWays(2)
                    ++ new WithL1ICacheSets(32)
                    ++ new WithCacheBlockBytes(64)
                    ++ new WithL2Cache
                    ++ new BaseConfig)

// Config with 4-way, 64 sets x 32 byts/block cache
// tag: 21 bits, index: 6 bits, WS: 3 bits, BS: 2 bits
class DualCoreConfig4way extends Config(
  new WithNCores(2) ++ new WithL1ICacheWays(4)
                    ++ new WithL1ICacheSets(64)
                    ++ new WithCacheBlockBytes(32)
                    ++ new WithL2Cache
                    ++ new BaseConfig)
```

(b) Example of new RocketChip configurations defined in `$RC/src/main/scala/rocketchip/Configs.scala`.

indeed created. However, the two and four tag SRAMs are merged into a big single two- and four-port SRAM respectively. The data SRAMs use a narrower but deeper storage array, providing word-wise access but retaining the correct capacity. The required capacity per cache is the same for both configurations so the same SRAM gets instantiated. The used SRAM verilog modules get created by the script `$RC/vsim/vlsi_mem_gen`, which per default, generates behavioural descriptions of the requested SRAMs. For an actual implementation, this script should be adapted to generate verilog wrappers for actually available SRAM blocks. Since such an implementation will be forced to always use the next bigger (wider and/or deeper) available SRAMs, it might be beneficial to dimension the cache accordingly to maximise the cache parameters within the used SRAMs.

Looking at the different iterations of generated code,

Listing III
EXTRACTS FROM THE GENERATED ICACHE_ICACHE MODULE.

```
module ICache_icache(...);
  ...
  module tag_array( // Instantiated once
    input  [4:0] RW0_addr,
    input   RW0_en,
    input   RW0_clk,
    input   RW0_wmode,
    input  [20:0] RW0_wdata_0,
    input  [20:0] RW0_wdata_1,
    output [20:0] RW0_rdata_0,
    output [20:0] RW0_rdata_1,
    input   RW0_wmask_0,
    input   RW0_wmask_1
  );
  reg [41:0] ram [31:0];
  ...
  module _T_772( // Instantiated twice
    input  [7:0] RW0_addr,
    input   RW0_en,
    input   RW0_clk,
    input   RW0_wmode,
    input  [63:0] RW0_wdata,
    output [63:0] RW0_rdata
  );
  reg [63:0] ram [255:0];
  ...
endmodule
```

(a) The instantiated SRAMs for the 2-way configuration.

```
module ICache_icache(...);
  ...
  module tag_array( // Instantiated once
    input  [5:0] RW0_addr,
    input   RW0_en,
    input   RW0_clk,
    input   RW0_wmode,
    input  [20:0] RW0_wdata_0,
    input  [20:0] RW0_wdata_1,
    input  [20:0] RW0_wdata_2,
    input  [20:0] RW0_wdata_3,
    output [20:0] RW0_rdata_0,
    output [20:0] RW0_rdata_1,
    output [20:0] RW0_rdata_2,
    output [20:0] RW0_rdata_3,
    input   RW0_wmask_0,
    input   RW0_wmask_1,
    input   RW0_wmask_2,
    input   RW0_wmask_3
  );
  reg [83:0] ram [63:0];
  ...
  module _T_850( // Instantiated four times
    input  [7:0] RW0_addr,
    input   RW0_en,
    input   RW0_clk,
    input   RW0_wmode,
    input  [63:0] RW0_wdata,
    output [63:0] RW0_rdata
  );
  reg [63:0] ram [255:0];
  ...
endmodule
```

(b) The instantiated SRAMs for the 4-way configuration.

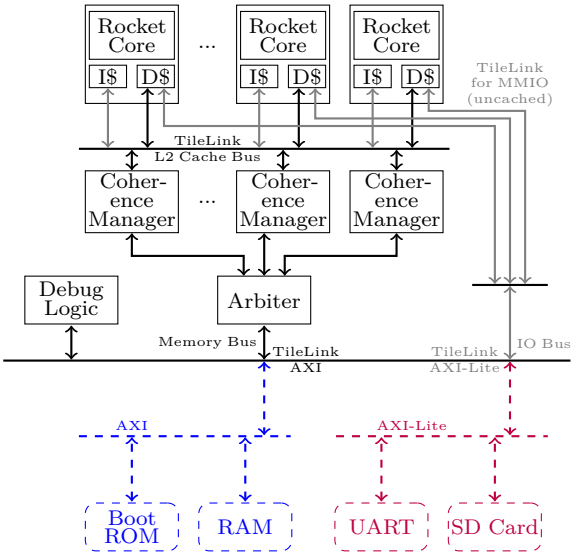

Figure 2. The principal busses in the RocketChip. The dashed parts are possible periphery modules and not part of RocketChip.

some key modules like the tiles and caches do not seem to change much. For the rest of the modules, instantiations change considerably when changing the parameters, so hand-writing code to interface to anything within the core except specified interfaces would seem rather unwise.

C. Interfaces

While the internals of the generated RocketChip vary strongly depending on the configuration options, the external interface is more static. Save the clock, the reset and the interrupt lines, it consists only of the debug interface and some high speed data busses.

Currently available open-sourced versions of RocketChip are tethered[17], meaning they will need a host environment to start up. In this process, the debug interface is essential, for loading a program to be run. To support untetherd operation, some form of non-volatile program memory needs to be added and the bootloader has to be modified to load executable code from that memory.

1) TileLink: The internal bus system uses TileLink[18], which is specific to RocketChip and connects the tiles, caches and other core components. Using the TileLink and few ancilliary signals as the only interface for each tile allows easy exchanging of tiles, which is one of the key features allowing the BOOM project to use the RocketChip to create a full processor around their core. See [19, p.23 ff] for an overview on how to implement a new tile.

The physical bus sytem used within the processor is divided into 4 blocks, each with independent synchronisation, data, address, source, size, mask, parameter and opcode signals, implementing two full-duplex paths. The widths of these signals vary in different parts of the processor. For the Rocket tile in default configuration, there are two TileLink interfaces (io_cached and io_uncached) with a width of around 440 each.

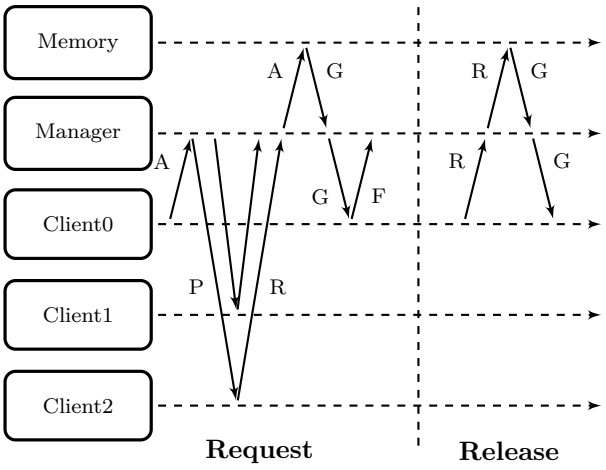

Figure 3. An example of TileLink request and release interactions.

TileLink is the cache coherence protocol on top of the physical bus system that connects different agents who participate in a coherent memory model. From TileLinks' point of view, each agent is either a

`Client`	which manipulates data in cache blocks, or a
`Manager`	which tracks and controls the access and data flow of the cache blocks.

An agent can also fit into both categories. For example, the L2 cache will appear as a client to caches farther out (or the memory), while managing its clients the L1 caches. TileLink specifies 5 basic operations, called channels:

`Acquire`	For getting permissions on a cache block and writing back data
`Probe`	To find out if a client holds a certain cache block
`Release`	Response to a probe. Returns permissions for a cache block along with its changed data
`Grant`	Provide permissions and data for a cache block or acknowledge a release
`Finish`	Used as final acknowledgement after receiving a Grant to allow ordering transactions.

A number of behavioural constraints have to be implemented in the agents to ensure consistent, deadlock-free operation:

- If the different channels share the same physical link, preemption rules must be followed to avoid a deadlock situation: Each channel listed in II-C1 must have priority over the channels listed above it (So that `Finish` has highest priority and is always sent first.
- A `Manager` must wait for transactions on a block to be finished before accepting new requests on that block unless it can merge the results of the different transactions.
- A `Client` cannot release and respond to a `Probe` if there is still an outstanding voluntary `Release` on that block.

Figure 3 gives an example of TileLink operations: Several `Clients` are managed by a higher-level `Manager` which is

itself a client to the memory[4]. In the first shown transaction `Client0`, which may be a DMA engine or a processor core, requests a cache block. As a result, the `Manager` sends `Probe` messages to all `Clients` under its control. At this point, there are two possibilities:

- No `Client` holds the requested cache block.
  The `Clients` indicate this in their `Release` response and after receiving all negative responses, the `Manager` requests the block from `Memory`. When receiving the `Grant` with the cache data, it forwards it to `Client0`. On reception, `Client0` sends a `Finish` message[5].
- One of the other `Clients` holds the requested cache block.
  It will release its hold of the cache block and return clobbered data with its `Release` message. The `Manager` now has the data and permissions for the cache block, so it can directly issue a `Grant` to `Client0` without contacting `Memory`. Like in the first case, `Client0` replies with a `Finish` message.[5]

If `Manager` implemented a directory, it would know if/which one of its `Clients` holds a requested cache block and therefore would not need to send the `Probes` to the other `Clients`, thereby saving bandwidth and minimising response time.

The second shown transaction is a voluntary release. `Client0` decides that it no longer needs to hold a cache block under its control and releases it by sending a `Release` message with the clobbered data to the `Manager`, which in turn sends a `Release` to the `Memory`. The `Memory` acknowledges the voluntary release by issuing a `Grant` which the `Memory` passes to `Client0`. For a simple write to memory, the diagram would look the same except that instead of `Release`, `Client0` would send an `Acquire` message with the data and parameters indicating that it does not need to acquire hold of the affected cache block.

2) Advanced eXtensible Interface: Earlier versions of RocketChip used a custom bus interface named Not A Standard Interface (NASTI and NASTI-Lite) for the external busses, but current versions use the Advanced eXtensible Interface (AXI) specified by ARM[20]. AMBA and AXI are well known open standards not only with many peripherals readily available as IP[21] but also good support for new development[22].

AXI uses a multi-channel interface where each channel is unidirectional. It supports different topologies, including multilayered point-to-point connections, bus topologies and mixtures with multiple data– but a shared address channel. The protocol is transaction-oriented and all transfers are done as bursts (although it is possible to have bursts of only one beat). Each channel has handshake signals (`Valid`, `Ready` and `Last` for the two data channels) and a two-bit response field and operates independent from the other channels. Here, "Independent" means, that there are relatively few constraints on relative timing between the diffent channels. While for example a read will (obviously) need to have the address to be read transmitted first, there is no such constraint for a write[6]. Each channel is equipped with four ID bits[7], which allow matching transactions between the different channels. Transactions from different masters or with different ID values may be reordered[8]. Using independent unidirectional channels allows to insert register stages at arbitrary points in the physical interconnect, allowing tradeoffs between required baud rate, latency and routing complexity. E.g. for a component farther out a register stage may be added at the cost of an additional bus cycle of latency to avoid degrading the bus frequency. AXI can use different power-of-two widths of the data paths up to 1024 bits, the default configuration of RocketChip uses 64 bits for both directions. Signals pertaining to a channel are marked by a common prefix[9]. These prefixes are:

`…_aw_…`	Write Address Channel
`…_w_…`	Write Data Channel
`…_b_…`	Write Response
`…_ar_…`	Read Address Channel
`…_r_…`	Read Data Channel
`…_c_…`	(Low Power Interface)

RocketChip does not use the Low Power Interface. Although also being an intra-IC interconnect, AXI is narrower than TileLink, consisting of around 280 signals per interface with the RocketChip default configuration.

To support the need of real-world systems with multiple processing units for synchronisation, AXI has an exclusive access feature: A master can issue an exclusive read. The addressed slave will provide the data like for a normal read. If it supports exclusive access it sets the response to `EXOKAY` and store the address for watching. When the master – some time after receiving the `EXOKAY` response – tries to exclusive write to the location with the same ID value, the slave will check if there was another write to the location. In case there was no other write, the slave will store the new data and respond `EXOKAY`. In case there was a concurrent write the slave will respond `OKAY` and not change the stored data. The slave will only watch one address for each transaction ID. A successive exclusive read with the same transaction ID without the exclusive write first, will result in the slave updating the watch-address. Detection of this capability is possible because a slave that does not support exclusive access will simply respond with `OKAY` (instead of `EXOKAY`).

Likewise, atomic access is supported as a concept. Devices in an atomicity group are designed to support

[4]There may be other managers – with other clients – that are siblings to the shown `Manager`, if the `Memory` is a Manager instead.

[5]The `Manager` does not need to pass the `Finish` to a higher-level Manager if the used network guarantees ordering of the sent packages.

[6]However, if there is some sort of switch in the interconnect, this switch will have to buffer the data until it knows the target address so that it can transfer the data only to the intended device.

[7]For RocketChip. The specification only **suggests** using 4 bits for masters and 8 bits of ID for slaves.

[8]While the specification states that transactions with different IDs or from different masters have no ordering restrictions, it imposes certain restrictions for sequences of transactions[20, 5.3] without defining what constitutes a sequence of transactions in contrast to some transactions a master sends over time.

[9]In the port list of the RocketChip Verilog module, this is actually in the middle of the name, as the different AXI interfaces are also indicated with a prefix.

the same maximum size of atomic memory updates. A write to one such device will update up to the bus width of bytes and address alignment atomically. E.g. an 8-bit aligned write on a 64 bit bus will result in an atomic write, while a write that is only aligned to 4 bits will only have 4 bits updated atomically. There does not seem to be a defined way to query the atomicity size of a device.

AXI version 3 did support locked transactions where a master could request the interconnect to be locked. Since this was a seldomly required use case that not only added significant complexity to the interconnect but also degraded the quality of service guarantees, this capability was removed from AXI4. If an AXI3 device actually issues a locked transaction in an AXI4 environment, the transaction will be treated as a normal transaction by the AXI4 devices, meaning that software relying on this feature will need to be changed.

AXI supports some other, more advanced features like specifying the bufferability of a memory and peripheral devices that are only required to allow certain methods of access. Describing these would go beyond the scope of this paper.

For lower performance peripherals like simple IO ports or UARTs, there is AMBA AXI4-Lite[20, B1] a simplified version of AXI4. It supports neither bursts, nor transfers smaller than the bus size, nor exclusive access, the bus width is fixed to either 32 or 64-bits and – as there is no ID field – all accesses have to be in order.

Interfacing between AXI and AXI-Lite only requires some logic to create the correct transaction IDs on the AXI side if the AXI part of the system can be designed to use only transactions that conform to the AXI-Lite subset when communicating with the AXI-Lite subsystem. For the other cases there is a Defined conversion mechanism[20, B1.3] to allow interfacing.

III. Conclusion

The RISC-V instruction set architecture is an interesting new player in the field of general purpose ISAs. It has sparked interest by a broad spectrum of players spanning from research facilities over commercial chip vendors to state-owned defense projects with several implementations in the process of reaching or having reached market-ready status and intensive use in the research field.

Rocket and Boom are open-source implementations of RISC-V processor cores developed at the University of Berkeley, as is RocketChip, a framework written in Chisel that can create processors using Rocket-, Boom- or custom cores.

In this paper, RocketChip was examined, the steps for creating new configurations were laid out, the propagation of the configuration options in Chisel into the generated Verilog code was tested and the relevant internal and external bus interfaces were described.

References

Links were last checked on April 20, 2017.

[1] Specification for RISC-V:
https://riscv.org/specifications/

[2] Krste Asanović presenting RISC-V at Stanford University in 2014
https://youtu.be/vB0FC1DZqUM

[3] The lowRISC project:
http://www.lowrisc.org/

[4] The SiFive company, founded by the creators of RISC-V:
https://www.sifive.com/

[5] Open-V, the Microcontroller developed by OnChip:
http://hackaday.com/2016/10/10/the-journey-toward-a-completely-open-microcontroller/
https://www.crowdsupply.com/onchip/open-v

[6] The Pulp processor project by ETH Zurich and UNIBO:
http://www.pulp-platform.org
http://iis-projects.ee.ethz.ch/index.php/PULP

[7] The Shakti processor project at IIT Madras:
http://rise.cse.iitm.ac.in/shakti.html

[8] Nvidias' anouncement to switch to RISC-V:
https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf

[9] List of companies presenting commercial implementations of RISC-V at Embedded World 2017
https://riscv.org/2017/03/risc-v-mainstream-ew2017/

[10] Example of RISC-V used in research project at UC Berkeley:
https://www.youtube.com/watch?v=WJndUQssFBg&t=1539s

[11] The Sodor processor collection:
https://github.com/ucb-bar/riscv-sodor

[12] Asanović, Krste/Avizienis, Rimas/Bachrach, Jonathan et al.
The Rocket Chip Generator. (2016)
Technical Report No. UCB/EECS–2016–17, EECS Department, University of California, Berkeley.
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf

[13] Celio, Christopher and Patterson, David A. and Asanović, Krste
The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor
Technical Report No. Celio:EECS-2015-167, EECS Department, University of California, Berkeley.
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.pdf

[14] The decode tables for the three UCB developed RISC-v processors:
https://github.com/ucb-bar/riscv-sodor/blob/master/src/common/instructions.scala#L39
https://github.com/ucb-bar/rocket-chip/blob/master/src/main/scala/rocket/IDecode.scala#L149
https://github.com/ucb-bar/riscv-boom/blob/master/src/main/scala/decode.scala#L123

[15] The RocketChip github pages
https://github.com/ucb-bar/rocket-chip
https://github.com/ucb-bar/project-template

[16] The Berkeley Out Of Order Machine github page
https://github.com/ucb-bar/riscv-boom

[17] Github issue about untethered operation of RocketChip
https://github.com/ucb-bar/rocket-chip/issues/434

[18] The TileLink Specification, Version 0.3.3
https://docs.google.com/document/d/1Iczcjigc-LUi8QmDPwnAu1kH4Rrt6Kqi1_EUaCrfrk8/pub

[19] Tutorial for the Rocket Chip build system.
https://riscv.org/wp-content/uploads/2015/01/riscv-rocket-chip-tutorial-bootcamp-jan2015.pdf

[20] AMBA® AXI™ and ACE™ Protocol Specification, ARM 2011
https://www.arm.com/products/system-ip/amba-specifications
Specification, see (A1.1) (requires registration):
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html
Specification (download from third party without registration):
http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf

[21] Exemplary listing of available IP cores:
https://www.design-reuse.com/sip/amba-axi-c-288/

[22] AMBA AXI Accelerated Verification IP (VIP) by Cadence:
https://ip.cadence.com/ipportfolio/verification-ip/accelerated-vip/arm-amba-2/amba-axi-accelerated-vip
AXI Interconnect support by Xilinx:
https://www.xilinx.com/products/intellectual-property/axi_interconnect.html