The first step we took in the beginning of the project was to decide on which implementations we will focus. We used the official RISCV website to get a list of some of the implementations exist, and then we explored each of the implementations.

For each implementation, we checked few things: what is the basic ISA version used, does it support any extensions, in which HDL it is written, who is the developer, does it have any working compiler and/or debugger and if it possible to burn to FPGA. Then we summarized it all in a table: (you will see that some of the boxes are left empty, because we couldn’t find all the information online)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| name | ISA | extensions | HDL | developer | compiler | debugger | FPGA/chip |
| Rocket | RV64I | - | Chisel | SiFive | The GNU GCC cross-compiler | Spike | No, the repository is not updated |
| Pulpino  (RISCY) | RV32I | CMF | System verilog | ETH Zurich and Bologna university | official RISC-V toolchain | debug software via gdb |  |
| Pulpino  (zero-riscy) | RV32I | CM | System verilog | ETH Zurich and Bologna university | official RISC-V toolchain | debug software via gdb |  |
| Pulpissimo | RV32I | CMF | System verilog | ETH Zurich and Bologna university | official RISC-V toolchain | imporoved version of the advanced debug interface from OpenCores |  |
| Roa Logic RV12 | RV32E/ 32I/64I | M | System verilog | Roa Logic | GNU compiler collection | GNU debugger |  |
| Shakti | RV64I | MAFD | System verilog | IIT Madras | BSV compiler(2017 or above) | JTAG debugger – based Modelsim | Only C class is designed for FPGA |
| Hummingbird E200 | RV32I | MAC | verilog | SI-RISCV | - | - |  |
| SCR1 | RV32E/32I | MC | System verilog | Syntacore | Altera Quartus 13.0.1 Build 232" | GDB |  |
| ORCA | RV32I | M | VHDL | VectorBlox | Quartus/QSYS or Xilinx Vivado | Modelsim simulator |  |
| Mriscv | RV32I | M | System verilog | OnChipUIS | RISC-V GCC toolchain |  |  |
| VexRiscv | RV32I | M | Spinal HDL | Spinal HDL | Risc-5-GCC | eclipse debugging via an GDB >> openOCD >> JTAG connection |  |

**How to build the Rocket Chip repository:**

Before you begin, make sure you have sudo permissions in order to install needed packages.

All information is from <https://github.com/freechipsproject/rocket-chip> and from <https://github.com/ucb-bar/project-template>.

Mainly, there are two ways to install the rocket chip: the hard way, by doing all the steps by yourself, and the easy way, that gets you the basic installtion. Let us begin with the hard way:

First, you need to check out the code:

*$ git clone https://github.com/ucb-bar/rocket-chip.git*

*$ cd rocket-chip*

*$ git submodule update --init*

To build the rocket-chip repository, you must point the RISCV environment variable to your riscv-tools installation directory:

*$ export RISCV=/path/to/riscv/toolchain/installation*

For example, in our system:

*rotemshahar@asic2-serv3:~$ echo $RISCV*

*/home/rotemshahar/project-template/rocket-chip/riscv-tools*

The riscv-tools repository is already included in rocket-chip as a Git submodule. You must build this version of riscv-tools: (on linux, install **Newlib**)

*$ cd rocket-chip/riscv-tools*

*$ git submodule update --init --recursive*

*$ export RISCV=/path/to/install/riscv/toolchain*

*$ export MAKEFLAGS="$MAKEFLAGS -jN" # Assuming you have N cores on your host system*

*$ ./build.sh*

*$ ./build-rv32ima.sh (if you are using RV32).*

Ubuntu packages needed:

*$ sudo apt-get install autoconf automake autotools-dev curl device-tree-compiler libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev*

Install chisel3:

1. Install Java

*$ sudo apt-get install default-jdk*

1. Install sbt, which isn't available by default in the system package manager

<https://www.scala-sbt.org/release/docs/Installing-sbt-on-Linux.html>

*$ echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt.list*

*$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 642AC823*

*$ sudo apt-get update*

*$ sudo apt-get install sbt*

Install Verilator: (the recommended version is 3.904)

1. Install prerequisites (if not installed already):

*$ sudo apt-get install git make autoconf g++ flex bison*

1. Clone the Verilator repository:

*$ git clone* [*http://git.veripool.org/git/verilator*](http://git.veripool.org/git/verilator)

1. In the Verilator repository directory, check out a known good version:

*$ git pull*

*$ git checkout verilator\_3\_904*

1. In the Verilator repository directory, build and install:

*$ unset VERILATOR\_ROOT # For bash, unsetenv for csh*

*$ autoconf # Create ./configure script*

*$ ./configure*

*$ make*

*$ sudo make install*

Then you have to build the project: (follow the instructions on this link)

<https://github.com/freechipsproject/rocket-chip#building-the-project>

Now, there is another way to install and use the rocket chip, in this way all you have to do is follow those instructions (from here <https://github.com/ucb-bar/project-template>):

First steps, as usual, check out the code:

*$git clone https://github.com/ucb-bar/project-template.git*

*$cd project-template*

*$git submodule update --init --recursive*

Then export those variables:

*export RISCV=/path/to/install/dir*

*export PATH=$RISCV/bin:$PATH*

next step, build the riscv-tools:

cd rocket-chip/riscv-tools

./build.sh

And that’s it. You can then use this to create your own project:

make PROJECT=yourproject CONFIG=YourConfig

./simulator-yourproject-YourConfig ...

For example, run this test:

./simulator-example-DefaultExampleConfig $RISCV/riscv64-unknown-elf/share/riscv-tests/isa/rv64ui-p-simple

**How to work with the debugger:**

All the information is from:

<https://github.com/freechipsproject/rocket-chip#-debugging-with-gdb>

Step 1: Generating the Remote Bit-Bang (RBB) Emulator:

The objective of this section is to use GNU debugger to debug RISC-V programs running on the emulator in the same fashion as in Spike (https://github.com/riscv/riscv-isa-sim#debugging-with-gdb)

For that we need to add a Remote Bit-Bang client to the emulator. We can do so by extending our Config with JtagDTMSystem, which will add a DebugTransportModuleJTAG to the DUT and connect a SimJTAG module in the Test Harness. This will allow OpenOCD to interface with the emulator, and GDB can interface with OpenOCD.

The config file is locates at src/main/scala/system/Configs.scala. In the following example, we added this Config extension to the DefaultConfig: (add it at the end of the file)

*class DefaultConfigRBB extends Config(*

*new WithJtagDTMSystem ++ new WithNBigCores(1) ++ new BaseConfig)*

*class QuadCoreConfigRBB extends Config(*

*new WithJtagDTMSystem ++ new WithNBigCores(4) ++ new BaseConfig)*

To build the emulator with DefaultConfigRBB configuration we use the command:

*rocket-chip$ cd emulator*

*emulator$ CONFIG=DefaultConfigRBB make*

We can also build a debug version capable of generating VCD waveforms using the command:

*emulator$ CONFIG=DefaultConfigRBB make debug*

By default the emulator is generated under the name emulator-freechips.rocketchip.system-DefaultConfigRBB in the first case and emulator-freechips.rocketchip.system-DefaultConfigRBB-debug in the second.

Step 2: Compiling and executing a custom program using the emulator:

We suppose that helloworld is our program, you can use crt.S, syscalls.c and the linker script test.ld to construct your own program, check examples stated in riscv-tests (https://github.com/riscv/riscv-tests).

In our case we will use the following example:

*export PATH=$RISCV/bin:$PATH*

You can install the git riscv-tests (with the link that is in this section). If, while installing the riscv-tests, you get an error: "riscv64-unknown-elf-gcc command not found", just type: "*export PATH=$RISCV/bin:$PATH*" and then continue.

In the riscv-tests there are some files you can use to test the debugger, they are at riscv-tests/benchmarks and the c file is in the folder with the same name. I use rsort which is a quicksort example. The .riscv is the assembly file, so where they use hellowold, I used rsort.riscv.

First we can test if your program executes well in the simple version of emulator before moving to debugging in step 3:

*$ ./emulator-freechips.rocketchip.system-DefaultConfig helloworld*

Additional verbose information (clock cycle, pc, instruction being executed) can be printed using the following command:

*$ ./emulator-freechips.rocketchip.system-DefaultConfig +verbose helloworld 2>&1 | spike-dasm*

VCD output files can be obtained using the -debug version of the emulator and are specified using -v or --vcd=FILEarguments. A detailed log file of all executed instructions can also be obtained from the emulator, this is an example:

*$ ./emulator-freechips.rocketchip.system-DefaultConfig-debug +verbose -v output.vcd helloworld 2>&1 | spike-dasm > output.log*

Please note that generated VCD waveforms and execution log files can be very voluminous depending on the size of the .elf file (i.e. code size + debugging symbols).

Please note also that the time it takes the emulator to load your program depends on executable size. Stripping the .elf executable will unsurprisingly make it run faster. For this you can use $RISCV/bin/riscv64-unknown-elf-strip tool to reduce the size. This is good for accelerating your simulation but not for debugging. Keep in mind that the HTIF communication interface between our system and the emulator relies on tohost and fromhost symbols to communicate. This is why you may get the following error when you try to run a totally stripped executable on the emulator:

*$ ./emulator-freechips.rocketchip.system-DefaultConfig totally-stripped-helloworld*

This text will appear on screen:

*This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag\_rbb\_enable=1.*

*Listening on port 46529*

*warning: tohost and fromhost symbols not in ELF; can't communicate with target*

To resolve this, we need to strip all the .elf executable but keep tohost and fromhost symbols using the following command:

*$ riscv64-unknown-elf-strip -s -Kfromhost -Ktohost helloworld*

More details on the GNU strip tool can be found here: https://www.thegeekstuff.com/2012/09/strip-command-examples/

The interest of this step is to make sure your program executes well. To perform debugging you need the original unstripped version, as explained in step 3.

Step 3: Launch the emulator:

First, do not forget to compile your program with -g -Og flags to provide debugging support as explained here.

We can then launch the Remote Bit-Bang enabled emulator with:

*$ ./emulator-freechips.rocketchip.system-DefaultConfigRBB +jtag\_rbb\_enable=1 --rbb-port=9823 helloworld*

This text will appear on screen:

*This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag\_rbb\_enable=1.*

*Listening on port 9823*

*Attempting to accept client socket*

You can also use the emulator-freechips.rocketchip.system-DefaultConfigRBB-debug version instead if you would like to generate VCD waveforms.

Please note that if the argument --rbb-port is not passed, a default free TCP port on your computer will be chosen randomly.

Please note also that when debugging with GDB, the .elf file is not actually loaded by the FESVR. In contrast with Spike, it must be loaded from GDB as explained in step 5. So the helloworld argument may be replaced by any dummy name.

Step 4: Launch OpenOCD:

You will need a RISC-V Enabled OpenOCD binary. This is installed with riscv-tools in $(RISCV)/bin/openocd, or can be compiled manually from riscv-openocd. OpenOCD requires a configuration file, in which we define the RBB port we will use, which is in our case 9823.

*$ cat cemulator.cfg*

Go to $RISCV directory and type (to create a new file)

*$ nedit cemulator.cfg*

Then copy and paste the text down here and save

*interface remote\_bitbang*

*remote\_bitbang\_host localhost*

*remote\_bitbang\_port 9823*

*set \_CHIPNAME riscv*

*jtag newtap $\_CHIPNAME cpu -irlen 5*

*set \_TARGETNAME $\_CHIPNAME.cpu*

*target create $\_TARGETNAME riscv -chain-position $\_TARGETNAME*

*gdb\_report\_data\_abort enable*

*init*

*halt*

You may have to install OpenOCD first, go to the $RISCV/riscv-openocd directory (rocket-chip/riscv-tools/riscv-openocd) and type in this order:

*./onfigure*

*make*

*make install*

You may need sudo before the make install (sudo make install)

Then we launch OpenOCD in another terminal using the command

*$(RISCV)/bin/openocd -f ./cemulator.cfg*

This text will appear on screen (some of it will appear during the work with the GDB at the next step):

*Open On-Chip Debugger 0.10.0+dev-00112-g3c1c6e0 (2018-04-12-10:40)*

*Licensed under GNU GPL v2*

*For bug reports, read*

*http://openocd.org/doc/doxygen/bugs.html*

*Warn : Adapter driver 'remote\_bitbang' did not declare which transports it allows; assuming legacy JTAG-only*

*Info : only one transport option; autoselect 'jtag'*

*Info : Initializing remote\_bitbang driver*

*Info : Connecting to localhost:9823*

*Info : remote\_bitbang driver initialized*

*Info : This adapter doesn't support configurable speed*

*Info : JTAG tap: riscv.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0000, ver: 0x0)*

*Info : datacount=2 progbufsize=16*

*Info : Disabling abstract command reads from CSRs.*

*Info : Disabling abstract command writes to CSRs.*

*Info : [0] Found 1 triggers*

*Info : Examined RISC-V core; found 1 harts*

*Info : hart 0: XLEN=64, 1 triggers*

*Info : Listening on port 3333 for gdb connections*

*Info : Listening on port 6666 for tcl connections*

*Info : Listening on port 4444 for telnet connections*

A -d flag can be added to the command to show further debug information.

Step 5: Launch GDB:

In another terminal launch GDB and point to the elf file you would like to load then run it with the debugger (in this example, helloworld):

*$ ./riscv64-unknown-elf-gdb helloworld*

Is it fails run this: *export PATH=$RISCV/bin:$PATH* and try again:

This text will appear on screen:

*GNU gdb (GDB) 8.0.50.20170724-git*

*Copyright (C) 2017 Free Software Foundation, Inc.*

*License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>*

*This is free software: you are free to change and redistribute it.*

*There is NO WARRANTY, to the extent permitted by law. Type "show copying" and "show warranty" for details.*

*This GDB was configured as "--host=x86\_64-pc-linux-gnu --target=riscv64-unknown-elf".*

*Type "show configuration" for configuration details.*

*For bug reporting instructions, please see:*

[*http://www.gnu.org/software/gdb/bugs/*](http://www.gnu.org/software/gdb/bugs/)*.*

*Find the GDB manual and other documentation resources online at:*

[*http://www.gnu.org/software/gdb/documentation/*](http://www.gnu.org/software/gdb/documentation/)*.*

*For help, type "help".*

*Type "apropos word" to search for commands related to "word"...*

*Reading symbols from ./proj1.out...done.*

*(gdb)*

Compared to Spike, the C Emulator is very slow, so several problems may be encountered due to timeouts between issuing commands and response from the emulator. To solve this problem, we increase the timeout with the GDB set *remotetimeout* command.

Then we load our program by performing a load command. This automatically sets the $PC to the \_start symbol in our .elf file.

*(gdb) set remotetimeout 2000*

*(gdb) target remote localhost:3333*

*Remote debugging using localhost:3333*

*0x0000000000010050 in ?? ()*

*(gdb) load*

*Loading section .text.init, size 0x2cc lma 0x80000000*

*Loading section .tohost, size 0x48 lma 0x80001000*

*Loading section .text, size 0x98c lma 0x80001048*

*Loading section .rodata, size 0x158 lma 0x800019d4*

*Loading section .rodata.str1.8, size 0x20 lma 0x80001b30*

*Loading section .data, size 0x22 lma 0x80001b50*

*Loading section .sdata, size 0x4 lma 0x80001b74*

*Start address 0x80000000, load size 3646*

*Transfer rate: 40 bytes/sec, 520 bytes/write.*

*)gdb)*

Now we can proceed as with Spike, debugging works in a similar way:

*(gdb) print wait*

*$1 = 1*

*(gdb) print wait=0*

*$2 = 0*

*(gdb) print text*

*$3 = "Vafgehpgvba frgf jnag gb or serr!"*

*(gdb) c*

*Continuing.*

*^C*

*Program received signal SIGINT, Interrupt.*

*main (argc=0, argv=<optimized out>) at src/main.c:33*

*33 while (!wait)*

*(gdb) print wait*

*$4 = 0*

*(gdb) print text*

*$5 = "Instruction sets want to be free!"*

*(gdb)*

Further information about GDB debugging is available:

<https://sourceware.org/gdb/onlinedocs/gdb/>

<https://sourceware.org/gdb/onlinedocs/gdb/Remote-Debugging.html#Remote-Debugging>

**Chisel toturial:**

All of the tutorials are basically the same

1. Github tutorial: (the summary in the table below)

* <https://github.com/ucb-bar/chisel-tutorial/wiki/The-Basics>
* <https://github.com/ucb-bar/chisel-tutorial/wiki/basic-types-and-operations>
* <https://github.com/ucb-bar/chisel-tutorial/wiki/instantiating-modules>
* <https://github.com/ucb-bar/chisel-tutorial/wiki/writing-scala-testbenches>

Bottom line - chisel testbench works well when it simple, few iterations and no more. If you want more than that- use c++ emulator or Verilog test harness.

* <https://github.com/ucb-bar/chisel-tutorial/wiki/Conditional-Assignments-and-Memories>

1. Risc-V workshops

<https://riscv.org/wp-content/uploads/2015/01/riscv-chisel-tutorial-bootcamp-jan2015.pdf>

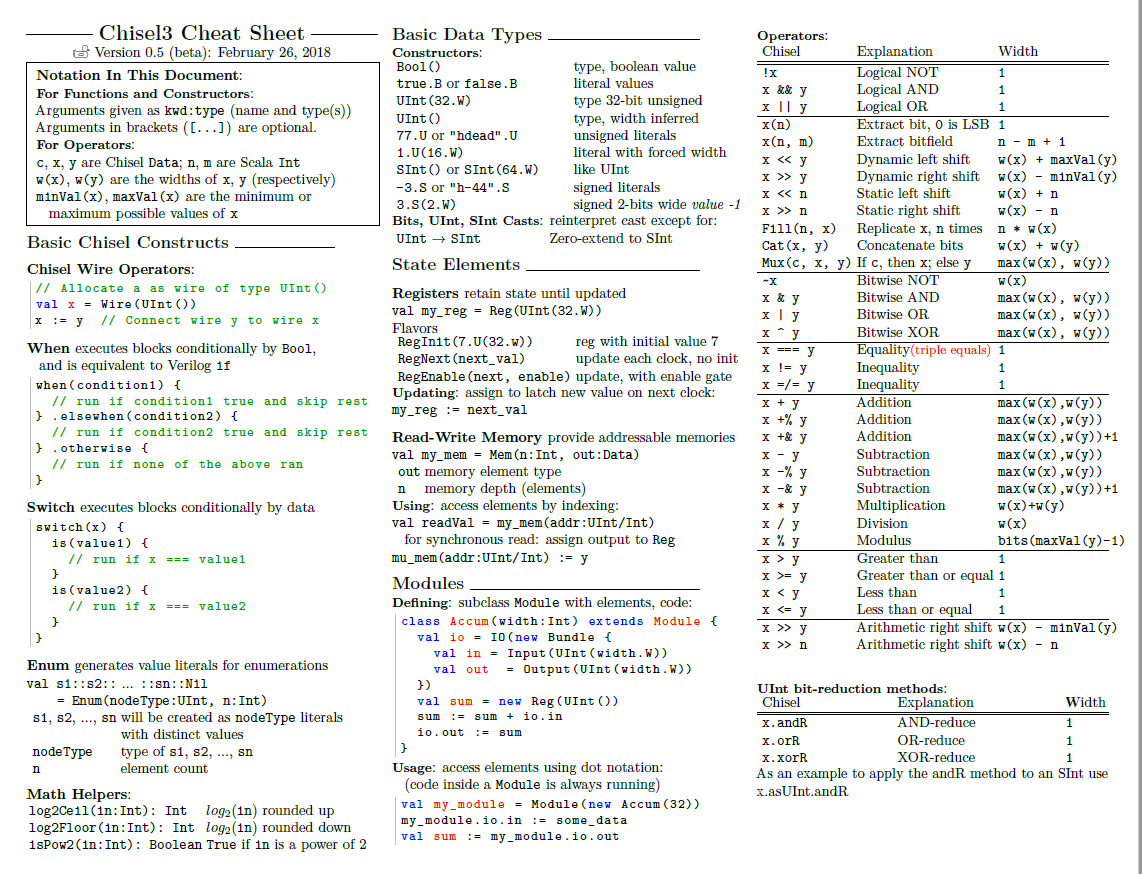
1. Berkeley's tutorial

<https://chisel.eecs.berkeley.edu/tutorial-20120618.pdf>

Chisel uses the Scala HDL as a platform.

|  |  |
| --- | --- |
| import chisel3.\_ | Use the chisel module- must |
| class GCD extends Module | Scala class definition for the Chisel component |
| val io = IO(new Bundle{   val a  = Input(UInt(16.W))      val b  = Input(UInt(16.W))      val e  = Input(Bool())      val z  = Output(UInt(16.W))      val v  = Output(Bool())  }) | Defines the inputs & outputs of the component |
| val y = io.x  x := w | The assignment is different whether we need to create it ("=") or reassign (":=").  => each value should assigned uses "=" for the first time, for the next time- use ":=" |
| UInt(16.W))    val true\_value  = true.B | The wire wide and type.  In this example – 16 bit- represents unsigned number    \*\*if we don’t declare the wire size- it will infare the bit width for us based on inputs |
| val x  = Reg(UInt())    val y  = Reg(UInt()) | Represent the registers, their type is Uint |
| when   (x > y) { x := x - y }      .elsewhen (x <= y) { y := y - x } | If condition- if true- do the command in {}  Else-  selecting the default assignment or keep a register value |
| io.z := x    io.v := y === 0.U | Implement the output's values |
| val y = io.x   val z = RegNext(y) | Positive edge triggered register  (z is the register's output, y is the register's input) |
| val x = Reg(UInt())  when (a > b) { x := y }  .elsewhen ( b > a) {x := z}  .otherwise { x := w}      when (<condition 1>) {<register update 1>}   .elsewhen (<condition 2>) {<register update 2>}   ...   .elsewhen (<condition N>) {<register update N>}  .otherwise {<default register update>} | Positive edge triggered register with conditions.  It isn't mandatory to write the "otherwise" if you want the default will be the old register's value          Another option- fiew conditions. The order matters! + default value |
| val r0 = RegInit(0.U(1.W)) | Allows to define reset value to zero(or any other value we choose) synchronously. |
| val x\_to\_y = z(x, y)  val x\_from\_z = z(x) | Mask for bits indexed (x, y) from variable z. or extract single bit.  X>y. |
| val A = UInt(32.W)   val B = UInt(32.W)   val bus = Cat(A, B) | Merge 2 values- concatenate  \*\*Need to import:  import chisel3.util.Cat  \*\*import all utills:  import chisel3.util.\_ |
| io.out := (in===0.U).asUInt    options:   * asUInt() * asSInt() * asBool() | casting |
| |  |  |  | | --- | --- | --- | | **Operand** | **Operation** | **Output Type** | | + | Add | UInt | | - | Subtract | UInt | | \* | Multiply | UInt | | / | UInt Divide | UInt | | % | Modulo | UInt | | ~ | Bitwise Negation | UInt | | ^ | Bitwise XOR | UInt | | & | Bitwise AND | UInt | | | | Bitwise OR | UInt | | === | Equal | Bool | | =/= | Not Equal | Bool | | > | Greater | Bool | | < | Less | Bool | | >= | Greater or Equal | Bool | | <= | Less or Equal | Bool | | Commonly used operations |
| val myVec = Vec(Seq.fill( <number of elements> ) { <data type> })    Example:  val ufix5\_vec10 := Vec(Seq.fill(10) { UInt(5.W) })  10 entry vector of 5 bit UInt values | Vector type  (we can create vector of registers) |
| reg\_vec32(0) := 0.U | Vector assignment    \*\*Vector of constants = read only memory |
| val myMem = Mem(128, UInt(32.W))    Memory - 128 units of 32-bit-Uint variables. | Memory instantiation |
| when (<read condition>) {     <read data 1> := <memory name>( <read address 1> )     ...     <read data N> := <memory name>( <read address N>)   } | Read memory ports |
| when (<write condition> ) {     <memory name>( <write address> ) := <write data>   } | Write memory ports |

**Cheesel cheat sheet**





**How to add a new instruction:**   
this is an example on how to change an existing command, and which files to look up to. For example, we want to change the instruction add\sub- increase the value by 1.

1. The RTL located in "rocket-chip/src/main/scala/rocket/" at .scala files.
2. Instructions.Scala- the file which contain the instruction object.
3. IDecode.Scala- contain the translation of the assembler command into parameters.
4. Constd.Scala- contain the opcode constants.
5. ALU.Scala- the implementation of the alu.
6. RVC.Scala- implementation of the immidiate instructions- including Addi lines- 74-79

*def q1 = {*

*def addi = inst(Cat(addiImm, rd, UInt(0,3), rd, UInt(0x13,7)), rd, rd, rs2p)*

….

}

Inst- translate the instruction to a struct:

*def inst(bits: UInt, rd: UInt = x(11,7), rs1: UInt = x(19,15), rs2: UInt = x(24,20), rs3: UInt = x(31,27)) = {*

*val res = Wire(new ExpandedInstruction)*

*res.bits := bits*

*res.rd := rd*

*res.rs1 := rs1*

*res.rs2 := rs2*

*res.rs3 := rs3*

*Res*

*}*

1. RocketCore.Scala- execute stage (line 272) ALU is implemented by:

|  |  |  |
| --- | --- | --- |
| *inputs:*     *From Intctrlsigs* | *ex\_op1 + ex\_op2 - mux lookup in core\_scala*  *Ex\_inn- value of immidiate data*    *Ex\_ctl.Dw- double word.*  *Ex\_ctl.fn* |  |
| *Outputs:* | *results* |  |

1. split to lsb + msb:

Chain of muxs that find key in inputs. The output is the operand- "val\_ex\_op1"

**How to add a new instruction:**

1. The RTL located in "rocket-chip/src/main/scala/rocket/" at .scala files.
2. Instructions.Scala- the file which contain the instruction object. We want to change the instruction add\sub- increase the value by 1.
3. IDecode.Scala- contain the translation of the assembler command into parameters.
4. Constd.Scala- contain the opcode constants.
5. ALU.Scala- the implementation of the alu.
6. RVC.Scala- implementation of the immidiate instructions- including Addi lines- 74-79

*def q1 = {*

*def addi = inst(Cat(addiImm, rd, UInt(0,3), rd, UInt(0x13,7)), rd, rd, rs2p)*

….

}

Inst- translate the instruction to a struct:

*def inst(bits: UInt, rd: UInt = x(11,7), rs1: UInt = x(19,15), rs2: UInt = x(24,20), rs3: UInt = x(31,27)) = {*

*val res = Wire(new ExpandedInstruction)*

*res.bits := bits*

*res.rd := rd*

*res.rs1 := rs1*

*res.rs2 := rs2*

*res.rs3 := rs3*

*Res*

*}*

1. RocketCore.Scala- execute stage (line 272) ALU is implemented by:

|  |  |  |
| --- | --- | --- |
| *inputs:*     *From Intctrlsigs* | *ex\_op1 + ex\_op2 - mux lookup in core\_scala*  *Ex\_inn- value of immidiate data*    *Ex\_ctl.Dw- double word.*  *Ex\_ctl.fn* |  |
| *Outputs:* | *results* |  |

1. split to lsb + msb:

Chain of muxs that find key in inputs. The output is the operand- "val\_ex\_op1"

**How to build the Pulpissimo repository:**

There are three repositories to build (in this order):

1. **PULP-SDK- (=software development kit)**
2. Pulp-Riscv-gnu-toolchain
3. Pulpissimo

**Start follow the instructions (pulp-SDK):**

<https://github.com/pulp-platform/pulp-sdk/blob/master/README.md>

*sudo apt install git python3-pip gawk texinfo libgmp-dev libmpfr-dev libmpc-dev swig3.0 libjpeg-dev lsb-core doxygen python-sphinx sox graphicsmagick-libmagick-dev-compat libsdl2-dev libswitch-perl libftdi1-dev cmake*

*sudo pip3 install artifactory twisted prettytable sqlalchemy pyelftools openpyxl xlsxwriter pyyaml numpy*

<https://github.com/pulp-platform/pulp-riscv-gnu-toolchain>

*git clone --recursive*[*https://github.com/pulp-platform/pulp-riscv-gnu-toolchain*](https://github.com/pulp-platform/pulp-riscv-gnu-toolchain)

*sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev*

*export PATH=$PATH:/opt/riscv/bin*

*cd pulp-riscv-gnu-toolchain/*

*./configure --prefix=/opt/riscv --with-arch=rv32imc --with-cmodel=medlow --enable-multilib* //This command chooses the ISA type you want to work with

*Sudo chmod 775 \**

In the lab, we need admin approval for few commands, so we use "sudo"

*Sudo make*

Back to <https://github.com/pulp-platform/pulp-sdk/blob/master/README.md#dependencies-setup>

*export PULP\_RISCV\_GCC\_TOOLCHAIN=/opt/riscv/*

*git clone --recursive*[*https://github.com/pulp-platform/pulpissimo.git*](https://github.com/pulp-platform/pulpissimo.git)

*export VSIM\_PATH=/home/rotemshahar/pulpissimo/sim*

*git clone --recursive*[*https://github.com/pulp-platform/pulp-sdk.git*](https://github.com/pulp-platform/pulp-sdk.git)*-b master*

*cd pulp-sdk*

*source configs/pulpissimo.sh* // choose the core type

*source configs/platform-rtl.sh*// options platform-board.sh  platform-fpga.sh   platform-gvsoc.sh  platform-rtl.sh

*Make all*

*source pkg/sdk/dev/sourceme.sh* // run every time-build things for our choice

[*https://github.com/pulp-platform/pulpissimo*](https://github.com/pulp-platform/pulpissimo)

*Cd ../pulpissimo*

*export VSIM\_PATH=/home/<your\_home\_dir>/pulpissimo/sim*

*\*\*in our repository:*

*export VSIM\_PATH=/home/rotemshahar/pulpissimo/sim*

*./update-ips*

*source setup/vsim.sh*

*cd sim/*

*export LM\_LICENSE\_FILE=5280@132.68.55.55* //modelsim license

*export PATH=$PATH:"/home/<your\_home\_dir>/<modelsim\_dir>/ bin/"*

*\*\*in our repository:*

*export PATH=$PATH:"/home/rotemshahar/new\_modelsim/modeltech/bin/"*

*make clean lib build opt*

*git clone https://github.com/pulp-platform/pulp-rt-examples.git*

**Quick Start from our repository:**

**We worked with SSH**: 132.68.59.10

*export PULP\_RISCV\_GCC\_TOOLCHAIN=/opt/riscv/*

*export VSIM\_PATH=/home/rotemshahar/pulpissimo/sim*

*cd pulp-sdk*

*source configs/pulpissimo.sh*

*source configs/platform-rtl.sh*

*source pkg/sdk/dev/sourceme.sh*

*cd ../pulpissimo*

*source setup/vsim.sh*

*cd sim/*

*export LM\_LICENSE\_FILE=5280@132.68.55.55*

*export PATH=$PATH:"/home/rotemshahar/new\_modelsim/modeltech/bin/"*

*make clean lib build opt*

*cd \*name of test folder\**

*make conf gui=1   // for modelsim gui*

*make clean all run*

**Pulpissimo simulation using Modelsim:**

Using the Modelsim GUI very recommended, it allows watching the different objects, waveforms, signals (list report), etc.

We used the example tests,and re-wrote the code as our own tests. The code written in C.

In order to activate the GUI, follow the instructions:

*cd <name of test folder>*

*make conf gui=1*

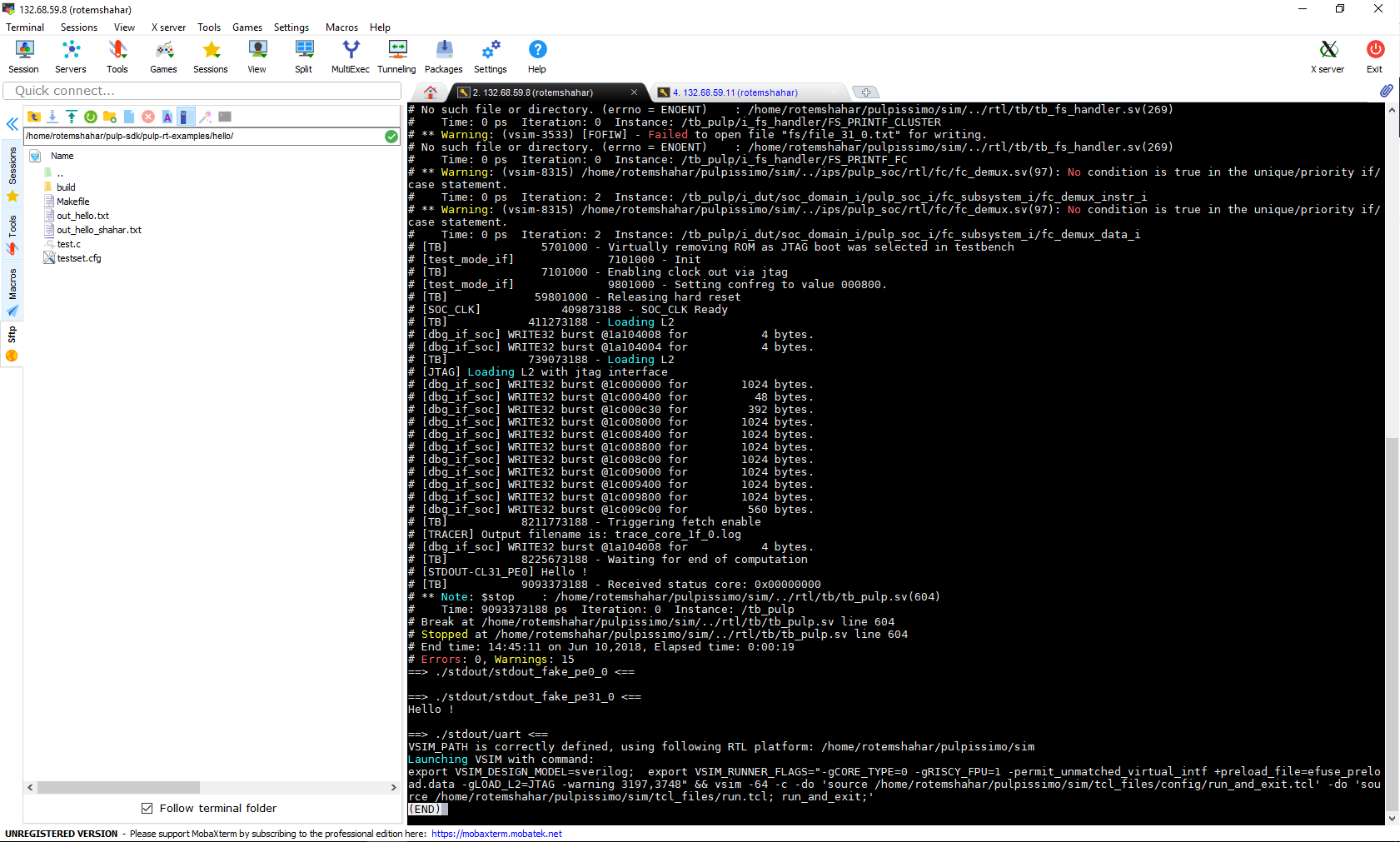
*make clean all run*

Results located in "build" repository.

Important log/results files:

1. "<test\_repository path>/build/pulpissimo/trace\_core\_1f\_0.log" – it contains the Assembly instructions, registers status and PC number.
2. The List output from the Modelsim simulation. It contain timestamp and buses values at the time.   
   The relevant buses which indicates the core's activity from the outside  are:
   * instr\_addr\_o
   * instr\_rdata\_I
   * data\_addr\_o
   * data\_wdata\_o
   * data\_rdata\_I

It is possible to convert the output to excel chart and analyze it.

**Output of "Hello" test, without Modelsim's gui:**

**Conclusions from Modelsim simulations:**

We tried to understand the memory behavior. We discovered it uses an algorithm in order to save memory usage. Therefore, the program we executed used the same memory cell for all variables. Moreover, the simulation uses JTAG to read new instruction (simulates the instruction cache)

#include <stdio.h>

#include "pulp.h"

int main**()**

**{**

int a**=**0xffff**;**

int b**=**0xeeee**;**

int c**=**0xdddd**;**

int d**=**0xcccc**;**

int e**=**0xbbbb**;**

int f**=**0xaaaa0000**;**

int sum\_a**,**sum\_b**,**sum\_c**,**sum\_d**,**sum\_e**,**sum\_f**,**sum\_g**=**0**;**

sum\_a**=**a**;**

sum\_b**=**b**;**

sum\_c**=**c**;**

sum\_d**=**d**;**

sum\_e**=**e**;**

sum\_f**=**f**+**d**;**

sum\_g**=**f**+**a**;**

printf**(**"Sum=%d\n"**,**sum\_a**);**

printf**(**"Sum=%d\n"**,**sum\_b**);**

printf**(**"Sum=%d\n"**,**sum\_c**);**

printf**(**"Sum=%d\n"**,**sum\_d**);**

printf**(**"Sum=%d\n"**,**sum\_e**);**

printf**(**"Sum=%d\n"**,**sum\_f**);**

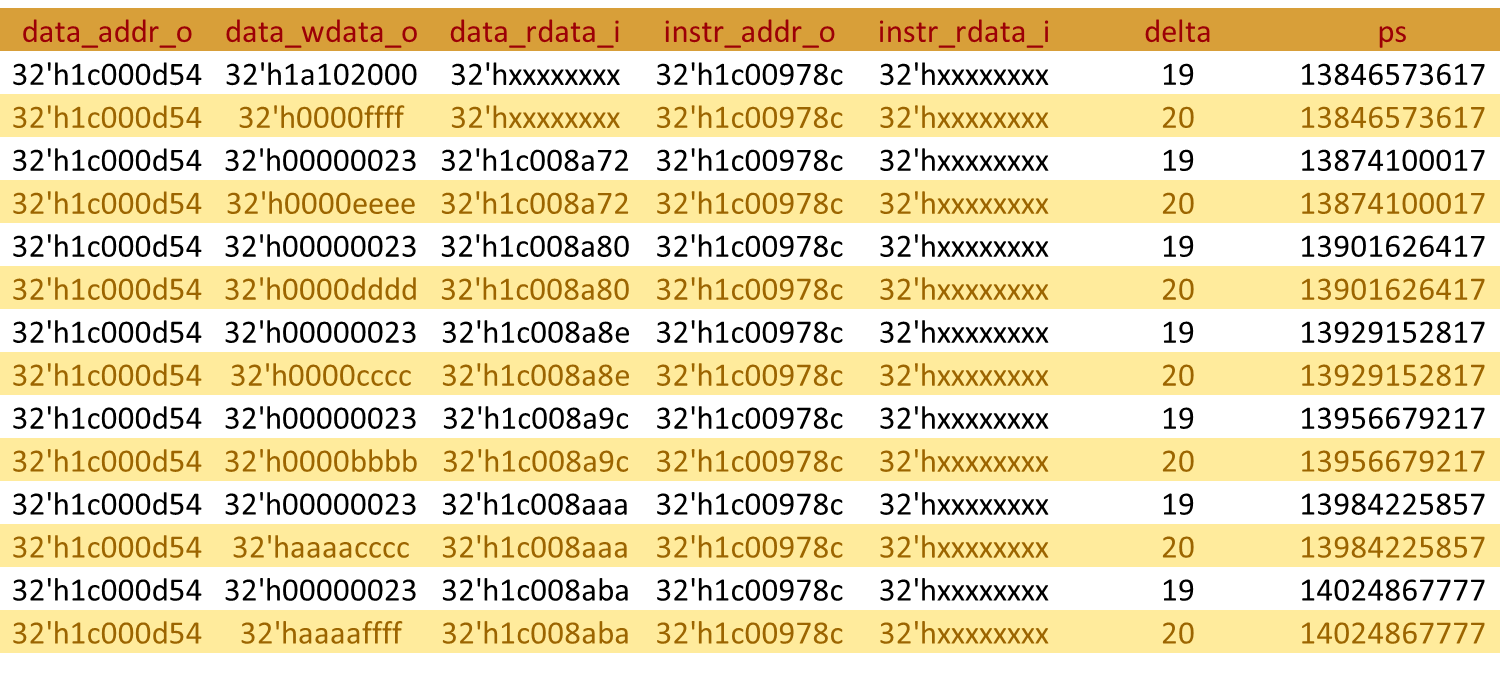
printf**(**"Sum=%d\n"**,**sum\_g**);**

printf**(**"Hello !\n"**);**

**return** 0**;**

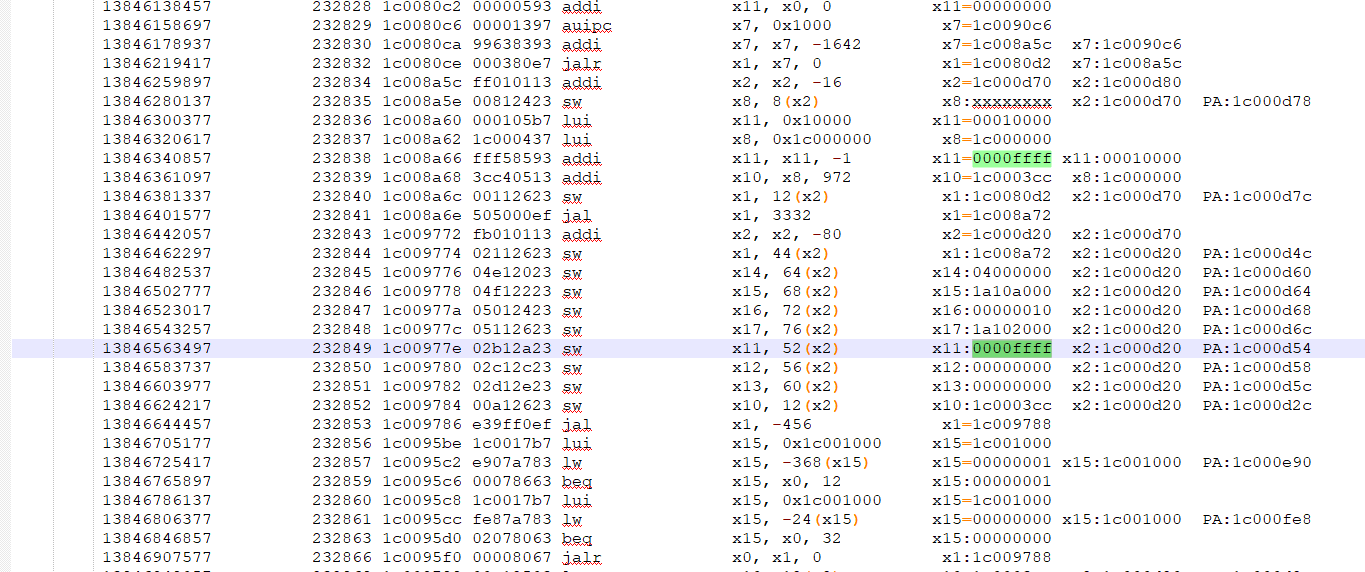
**}**

**©Varun Tandon**

The output list, converted to excel chart:

We can see the values stored in the same place

**General Conclusions:**

The ISA is not 4B-aligned, extensive explanations in the user manual.

The output above is part of the trace log file.   
From right to left: cycles, PC, instruction, assembly instruction, registers' status.

* User manuals of all PULP implementations:

<https://www.pulp-platform.org/implementation.html>