RISC-V Atom

Release v1.2

Saurabh Singh

Jan 16, 2024

OVERVIEW

Welcome to RISC-V Atom Documentation and User Manual! Please follow the *getting started guide* to setup an environment to build and test the RISC-V Atom project. Please feel free file a bug report in github.

RISC-V Atom
A simple 32-bit embedded class RISC-V processor

CHAPTER

ONE

INTRODUCTION

RISC-V Atom is an open-source soft-core processor platform targeted for FPGAs. It is complete hardware prototyping and software development environment based around **Atom**, which is a 32-bit embedded-class processor based on the RISC-V Instruction Set Architecture (ISA).

1.1 Key Highlights

Key highlights of the RISC-V Atom projects are are listed below:

1. Atom implements `RV32IC_Zicsr` ISA as defined in the RISC-V unprivileged ISA manual.
2. Simple 2-stage pipelined architecture, ideal for smaller FPGAs.
3. Optional support for RISC-V exceptions and interrupts.
4. Wishbone ready CPU interface.
5. Interactive RTL simulator - *AtomSim*.
6. In-house verification framework - *SCAR*.
7. Multiple SoC configurations.
8. Tiny libc like standard library - *Libcatom*.
9. Wide range of example programs.
10. Open source under MIT License.

Tip: To get started, Check out the *getting started guide*.

1.2 Components

Following is list of various components of the RISC-V Atom project.

RISC-V Atom CPU A simple 32-bit RISC-V processor.

SoC Targets RISC-V Atom project provides several configurable SoC targets that can be built around the Atom CPU.

AtomSim AtomSim is the interactive RTL simulator for RISC-V Atom SoCs.

SCAR SCAR (Search, Compile Assert, and Run) is an in-house processor verification framework written in python.

CHAPTER

TWO

DIRECTORY STRUCTURE

```
riscv-atom          : root directory
    |
    ├─ docs             : RISCV-Atom documentation & user manual
    |
    ├─ rtl              : RISCV-Atom Verilog Sources
    |   ├─ common           : Common headers
    |   ├─ config           : SoC target config files (JSON)
    |   ├─ core             : RISCV-Atom core components
    |   ├─ dpi              : SystemVerilog DPI sources
    |   ├─ soc              : SoC RTL files
    |   ├─ tb               : Verilog testbenches
    |   └─ uncore           : RISCV-Atom non-core components (SoC peripherals)
    |
    ├─ scripts         : Commonly used python and bash scripts
    |
    ├─ sim              : Atomsim source code
    |   ├─ build            : AtomSim build files (autogenerated)
    |   ├─ docs             : AtomSim Source Documentation (Doxygen)
    |   └─ include          : Third party Libraries for AtomSim
    |
    ├─ sw               : Software sources
    |   ├─ bootloader       : RISC-V Atom bootloader
    |   ├─ examples         : Example programs
    |   └─ lib              : Libc for RISCV-Atom (libcatom)
    |       ├─ include          : Libcatom headers
    |       ├─ libcatom         : Libcatom sources
    |       └─ link             : Linker scripts
    |
    ├─ synth            : RISC-V Atom Synthesis
    |   ├─ altera           : Synthesis project for Altera FPGAs
    |   ├─ xilinx           : Synthesis project for Xilinx FPGAs
    |   └─ yosys            : Synthesis project for Yosys
    |
    ├─ test             : RISCV Atom tests
    |   ├─ riscv-target     : Official RISC-V compliance test files
    |   └─ scar             : SCAR tests directory
    |
    └─ tools            : Utility tools
        └─ elfdump          : Elfdump utility
```

CHAPTER

THREE

PREREQUISITES

This page discusses how to set up your system in order to get RISC-V Atom up and running.

3.1 Required Packages

Note: RISC-V Atom project has been developed and tested on **Ubuntu 20.04**. However, It should work just fine on any other linux based distribution with relevant packages.

3.1.1 Install git, make, python3, gcc & other tools

RISC-V Atom uses *Make* for all builds. *GNU C/C++ compilers*, *Make* and other essential build tools are conveniently packaged as `build-essential` meta package in Ubuntu. RISC-V Atom uses the GNU Readline library to implement the interactive console in AtomSim. We also want to install GTKWave to view VCD waveforms and screen to connect to serial ports.

```
$ sudo apt-get update
$ sudo apt-get install git python3 python3-pip build-essential libreadline8 libreadline-
↪dev socat
$ sudo apt-get install gtkwave screen
$ pip install -r requirements.txt
```

3.1.2 Install Verilator

Verilator will be used By AtomSim to *Verilate* Verilog RTL into C++. We recommend installing the latest stable Verilator version (>=5.006) using git quick install method.

3.1.3 Install RISC-V GNU Toolchain

We will be installing the **RISC-V 64-bit Multilib** toolchain Further install instructions can be found here. We recommend using the provided `install_toolchain.sh` script to install the proper toolchain.

```
$ chmod +x install_toolchain.sh
$ ./install_toolchain.sh
```

3.1.4 Allow user to access serial ports

To allow current linux user to access serial ports and usb devices (such as JTAG), the user must be added to the `dialout` and `plugdev` groups respectively.

```
$ sudo usermod -aG dialout $USER
$ sudo usermod -aG plugdev $USER
```

Note: This takes effect after user logs out and logs back in.

3.1.5 OpenFPGAloader

We use openFPGAloader to load bitstreams on FPGA. you are free to use vendor tools instead. To install openFPGAloader follow this guide.

3.2 Optional Packages

Note: The following packages are optional and are only required for generating documentation using Doxygen & Sphinx

3.2.1 Install Doxygen

Doxygen a tool is used to generate C++ source code documentation from comments inside the C++ source files.

```
$ sudo apt-get install doxygen
```

3.2.2 Install Latex Related packages

These packages are essential for generating Latex documentation using Doxygen.

```
$ sudo apt -y install texlive-latex-recommended texlive-pictures texlive-latex-extra␣
↪latexmk
```

3.2.3 Install sphinx & other python dependencies

Sphinx is used to generate the RISC-V Atom Documentation and User-Manual in PDF & HTML. To install the packages to generate sphinx documentation, run the following command in `docs` directory under riscv-atom repository.

```
$ pip install -r docs/requirements.txt
$ sudo apt install graphviz             # Graphviz is for flow diagrams in sphinx␣
↪documentation
```

CHAPTER

FOUR

BUILDING RISC-V ATOM

4.1 Clone the repository

First let's clone the repository as follows.

```
$ git clone https://github.com/saursin/riscv-atom.git
$ cd riscv-atom     # switch to riscv-atom directory
```

Note: All the commands are executed from the root directory unless explicitly mentioned. We'll refer to this root directory as `RVATOM`.

4.2 Setting up the environment

1. `RVATOM` environment variable must point to root of riscv-atom directory for the tools & scripts to work properly.
2. `RVATOM_LIB` environment variable must point to the `RVATOM/sw/lib` folder. This variable is used by the compile scripts to locate *libcatom*.

For convenience, `RVATOM/sourceme` script is provided that you can source as follows:

```
$ source sourceme
```

Tip: With this method, every time you open a new terminal, you have to source the `sourceme` file. You can optionally append the aforementioned to your `.bashrc` to source it automatically every time you open a new terminal.

```
$ echo "source <rvatom-path>/sourceme" >> ~/.bashrc
```

Replace `rvatom-path` with the path to your RISC-V atom directory.

4.3 Building AtomSim

AtomSim is the interactive RTL simulator for RISC-V Atom. Let's build AtomSim for *AtomBones* target.

```
$ make soctarget=atombones sim=1
```

Optionally, to speed up builds, you can specify the number of parallel jobs to run using `-j <njobs>` flag in the above command. This will create `RVATOM/sim/build` directory for AtomSim build files. You can find the Atomsim executable in `RVATOM/sim/build/bin` directory.

Assuming you've sourced the `RVATOM/sourceme` file, try the following command to check if the build was successful.

```
$ atomsim --version
v2.2 [ atombones ]
...
```

CHAPTER

FIVE

RISC-V ATOM DEVELOPMENT IN DOCKER

Alternative to previous approach, you can also use the provided Dockerfile to build a Docker image containing all the necessary tools to checkout the RISC-V Atom project. As a prerequisite, you must have Docker installed on your system. You can install Docker by following the official Docker guide.

Once you have installed Docker, you can clone the RISC-V Atom repository and build the Docker image as follows.

```
$ git clone https://github.com/saursin/riscv-atom.git
$ cd riscv-atom                          # switch to riscv-atom directory
$ docker build . -t rvatom-dev           # we'll name this image rvatom-dev
```

Once the build is finished you should be able to see the image using the following command:

```
$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
rvatom-dev   latest    a9cab48034fc   24 hours ago   475MB
```

To run an instance of this docker image (also called container), you can run the following command.

```
$ docker run -it -v .:/home/riscv-atom rvatom-dev
> Setting environment variables...
*** Welcome to the riscv-atom container! ***
root@7110d3ddecd7:/home/riscv-atom#
```

The above command should launch the container and attach to it, and you should have a familiar linux prompt! It will also mount the riscv-atom directory on host machine to `/home/riscv-atom` directory in the container, and automatically set-up the environment variables for RISC-V Atom development.

Tip: Checkout this cheatsheet to learn more about Docker CLI syntax.

CHAPTER

SIX

RUNNING EXAMPLES ON ATOMSIM

The RISC-V Atom project consists of a wide range of examples programs out-of-the-box to test. These examples programs reside in *RVATOM/sw/examples* directory.

Switch to examples directory

```
$ cd sw/examples
```

Lets run the classical "hello World!" example first!

6.1 Hello World Example

The source code for the *hello-world* example resides in the `hello-asm` directory. You can have a look at the source code. First we need to compile the hello world example with our RISC-V gcc cross-compiler. For this purpose, use the provided makefile as following.

```
$ make soctarget=atombones ex=hello-asm sim=1 compile
```

The above command should generate a `hello.elf` file in the `hello-asm` directory.

Tip: `soctarget=atombones` in the above command can be skipped if AtomSim is already built. The SoC target will be automatically detected from the AtomSim executable.

Now fire up AtomSim and provide the generated elf file as argument.

```
$ atomsim -u hello-asm/hello.elf
   ___  __              _____
  / _ |/ /____  __ _  / __(_)_ _
 / __ / __/ _ \/  ' \_\ \/ /  ' \
/_/ |_\__/\___/_/_/_/___/_/_/_/_/  v2.2
----------8<-----------8<-----------8<-----------8<---------

**** RISC-V Atom Bootloader ****
bootmode: 0x1
Jumping to RAM
--------------------------------
Hello World!
      -- from Assembly
```

(continues on next page)

(continued from previous page)

```
EBreak hit at 0x2000014a
Exiting..
```

You should see *Hello world* message on the screen.

Note: Make sure to use `-u` flag to direct UART output from SoC to stdout. You can also use `-t` flag to generate a VCD trace of the simulation.

Alternatively, use `make run` to run the example as follows

```
$ make soctarget=atombones ex=hello-asm run
```

We can compile other examples also in the similar fashion by using the following syntax:

```
$ make soctarget=<TARGET> ex=<EXAMPLE> sim=1 compile
$ make soctarget=<TARGET> ex=<EXAMPLE> run
```

Note: Run `$ make help` to get more information about supported targets and examples.

6.2 Banner Example

```
$ make target=atombones ex=banner compile
```

```
$ atomsim -u banner/banner.elf
   ___  __              _____
  / _ |/ /____  __ _  / __(_)_ _
 / __ / __/ _ \/  ' \_\ \/ /  ' \
/_/ |_\__/\___/_/_/_/___/_/_/_/_/  v2.2
----------8<-----------8<-----------8<-----------8<---------

**** RISC-V Atom Bootloader ****
bootmode: 0x1
Jumping to RAM
--------------------------------

                 .';,.           ....;;;.
                .ll,:o,                ':c,.
                .dd;co'                  .cl,
               .:o:;,.                     'o:
               co.                          .oc
              ,o'          .coddoc.          'd,
              lc         .lXMMMMMMXl.         ll
             .o:         ;KMMMMMMMMK,         :o.
             .o:         'OMMMMMMMMO.         :o.
              co.         .o0XNNX0o.          .oc
              .o:           ..''..            :o.
               'o:                           :o'
```

(continues on next page)

(continued from previous page)

```
                .lc.                    .ll.
                  ,lc'                'cl,
                    'cc:,..       ..,:c:'
                       .;::::;;;;::::;.
                             ....
     ____  _________ _______    __         __
    / __ \/  _/ ___// ____/ |  / /  ____ _/ /_____  ____ ___
   / /_/ // / \__ \/ /    | | / /  / __ `/ __/ __ \/ __ `__ \
  / _, _// / ___/ / /___  | |/ /  / /_/ / /_/ /_/ / / / / / /
 /_/ |_/___//____/\____/  |___/   \__,_/\__/\____/_/ /_/ /_/
/=========By: Saurabh Singh (saurabh.s99100@gmail.com)====/

CPU      : RISC-V Atom @ 50000000 Hz
Arch     : RV32IC - little endian
CODE RAM : 0x20000000 (40960 bytes)
DATA RAM : 0x2000a000 (8192 bytes)
Exiting...
EBreak hit at 0x2000007c
Exiting..
```

6.3 How to compile and run all examples?

Instead of testing all examples one-by-one, we can compile and run all examples as follows.

```
$ make soctarget=atombones run-all
```

6.4 Using Atomsim Vuart

When using `-u` flag, AtomSim relays the output of the running application on stdout. But, in this mode of operation, user cannot provide any input to the running program. AtomSim provides Virtual UART to work around this problem. Virtual UART is an inbuilt class in AtomSim that can attach the stdin, stdout streams of the simulation to a linux serial port.

6.4.1 Generating Pseudo Serial Ports

A pair of connected pseudo serial ports can be generated by using the provided `atomsim-gen-vports` script as following.

```
$ atomsim-gen-vports
```

This will generate a pair of new pseudo serial ports in `/dev/pts` and links them together using the `socat` linux command. This means that whatever is sent to port-1 is received at port-2 and vice-versa. Further, this script also generates symlinks to these generated ports in the `RVATOM` directory as `simport` and `userport`.

6.4.2 Interacting with Stdout and Stdin over Pseudo Serial Ports

Open a new terminal (say terminal-2) and run the screen command as following

```
$ screen $RVATOM/userport 115200
```

And on the other terminal (terminal-1) run atomsim as following

```
$ atomsim hello-asm/hello.s -p $RVATOM/simport -b 115200
```

You should now be able to see the output on the terminal-2.

Tip: To close *screen* first press `ctrl + a`, then press `k` followed by `y`. To clear the *screen* window, press `ctrl + a`, then press `shift + c`.

6.4.3 Adding New Examples

To add a new example to the existing framework, simply create a directory under the `RVATOM/sw/examples` directory.

```
$ mkdir myexample
```

Next, put your source files under this directory.

```
$ cat myexample.c
#include <stdio.h>
void main()
{
    char hello[] = "New Example\n";
    printf(hello);
    return;
}
```

Finally add a new file named `Makefile.include` in the same directory which defines the name of the source files and executable file as follows.

```
$ cat Makefile.include
src_files = myexample.c
executable = myexample.elf
```

That's it! Now you can use the same compile and run commands as discussed earlier to run this example.

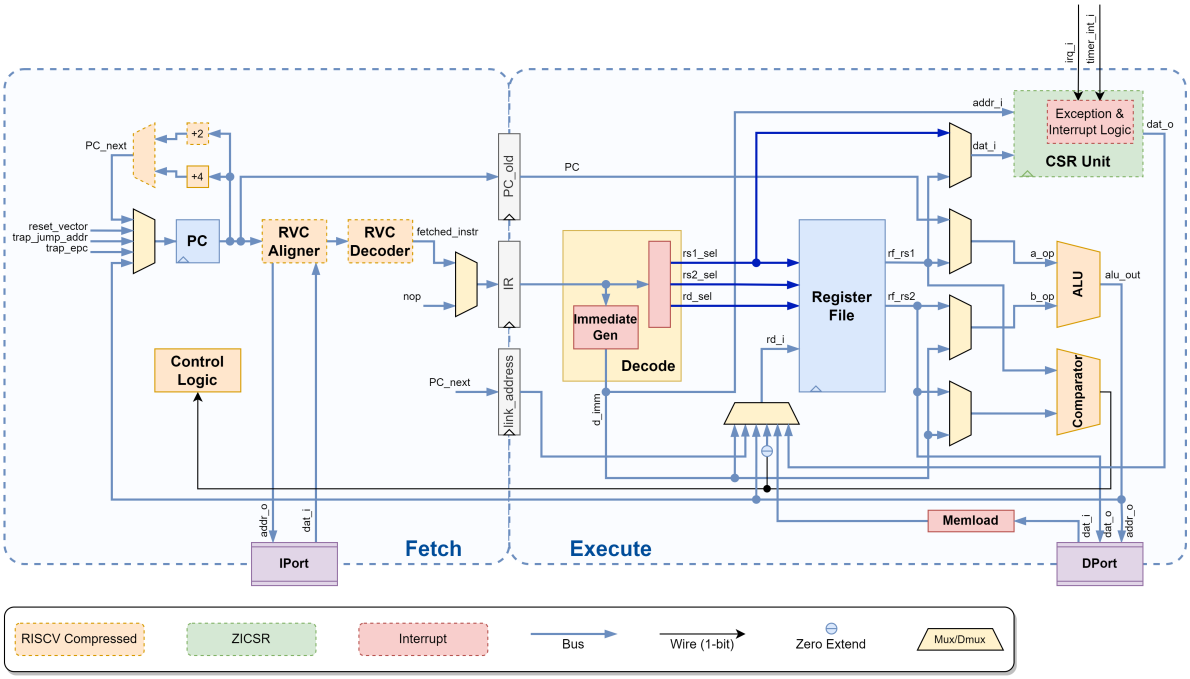
CHAPTER

SEVEN

RISC-V ATOM CPU

Atom is an open-source 32-bit soft-core processor written in Verilog. It is an embedded class processor architecture that implements the open-source RISC-V instruction set architecture (RV32IC), as described in the RISC-V unprivileged spec. Atom contains a two stage pipeline inspired from arm cortex m0+.

The following diagram showcases the architecture of RISC-V Atom core.



7.1 Atom Pipeline Stages

The pipeline is divided into two stages. These are explained below.

7.1.1 Stage-1: Fetch

Fetch unit is responsible for fetching instructions from instruction memory through the IPort. It uses a 32-bit Program Counter (PC) to keep track of the address of the instruction being fetched. After the instruction is successfully fetched, Program counter is incremented by either 4 or 2 (in case of compressed instruction). Fetch stage also includes pipeline control logic which controls pipeline stalls and flushes. If compressed extension in enables, Fetch stage includes RISC-V Compressed Aligner which aligns all the memory requests to 4 byte boundary. It also includes RISC-V Compressed instruction decoder, which decodes 16-bit compressed instructions to their 32-bit equivalents.

7.1.2 Stage-2: Decode, Execute & Write-back

In this stage, the instruction from Instruction Register (IR) is decoded and executed. First, the decode unit decodes the instruction and sets all the control signals in order to configure the data-path to execute the instruction. Parallelly, operand registers are fetched and 32-bit immediate value is generated. Next, ALU/Comparator does the necessary computation/comparison and the results are written back to register file. In case current instruction invokes a memory request, stage-2 is stalled until response is received. Memload module is used extract the correct data from the received memory response. If CSR extension is enabled, CSR Unit is included in this stage. CSR Unit provides Control and status registers which perform various special functions. SR Unit also includes exceptions and interrupt handling logic if Exceptions and interrupts are enabled. Branch calculation also happens in this stage and if branch is taken, a signal is sent to the pipeline control logic to flush the pipeline.

7.2 Atom Interface

Atom module is defined in the file `RVATOM/rtl/core/atomRV.v`. It has has two independent ports (IPort & DPort) which it uses to access memory. Both the ports use a generic ready-valid handshaking protocol to transfer data. We also provide wrappers to the core to convert the generic handshaking protocol to standard bus protocols such as Wishbone. These wrappers are specified in the following files.

1. Wishbone-B4 Wrapper with separate instruction and data port: `RVATOM/rtl/core/atomRV_wb.v`

7.3 Atom Configuration operations

Macro	Function
`EN_RVC`	Enables support for RISC-V Compressed Extension
`EN_RVZICSR`	Enables Control and Status Registers (CSRs)
`EN_EXCEPT`	Enables support for RISC-V interrupts and exceptions
`DPI_LOGGER`	Enable DPI Logger

7.4 RISC-V Atom RTL

RISC-V Atom is written in Verilog. The RTL specification for Atom is divided into 3 categories, *core*, *uncore* and *soc*, all of which reside in the `rtl` directory.

7.4.1 Core Directory

The `rtl/core` subdirectory contains the the *core* components of the CPU such as register file, ALU, decode unit etc. It also contains Verilog header files like `Defs.vh` and `Utils.vh`. `Defs.vh` contain various signal enumerations and other parameters internal to the processor. `Utils.vh` defines some useful utility macros.

7.4.2 Uncore Directory

The `rtl/uncore` subdirectory contains the SoC peripheral components such as UART, GPIO, memories, etc. It also contains the wishbone wrappers of some non-wishbone components. SoC implementations of the Atom processor usually instantiate these hardware modules in their implementations.

7.4.3 SoC directory

The top level Verilog modules for SoCs (such as AtomBones & HydrogenSoC) are present in the `rtl/soc` directory in the corresponding subdirectories. Each of these top level modules are configured by their respective configuration headers (<name>_Config.vh file). These configuration headers contain the macros used in the top-module definitions to control the generation of SoC, various sub-components and their parameters.

7.5 RTL Features

7.5.1 DPI Logger

DPI Logger is a System Verilog DPI based logging mechanism provided with the RTL. It can be used to dump useful run-time debug information such as PC values, Jump addresses, Loads and Stores, etc. into a log file. This module is present in the `rtl/dpi` subdirectory.

To enable DPI Logger simply define `DPI_LOGGER` macro in the top-module or in the CLI as `-DDPI_LOGGER`. This will trigger the inclusion of the `rtl/dpi/util_dpi.vh` header in `rtl/core/Utils.vh`. User is free include the `rtl/core/Utils.vh` header file in any Verilog file that needs to be debugged.

To log information, user first needs to call the `dpi_trace_start()` function somewhere in the rtl as following.

```
initial begin
  dpi_trace_start();
end
```

Then the `dpi_trace()` function can be used to dump information. Its syntax is exactly same as the Verilog `$display` system function. Example of logging the jumps during code execution is provided in the `AtomRV.v` file and is also shown below.

```
`ifdef DPI_LOGGER
    initial begin
        dpi_logger_start();     // begin logging
    end
`endif

`ifdef LOG_RVATOM_JUMP
always @(posedge clk_i) begin
    if(jump_decision)   // on some trigger condition
        dpi_logger("Jump  address=0x%x\n", {alu_out[31:1], 1'b0});      // dump␣
↪information
```

(continues on next page)

(continued from previous page)

```
end
`endif
```

For logging the Jumps, user must also define the `LOG_ATOMRV_JUMP` macro in a similar way. This will generate a `run.log` file in the current directory containing all the dumped information.

CHAPTER

EIGHT

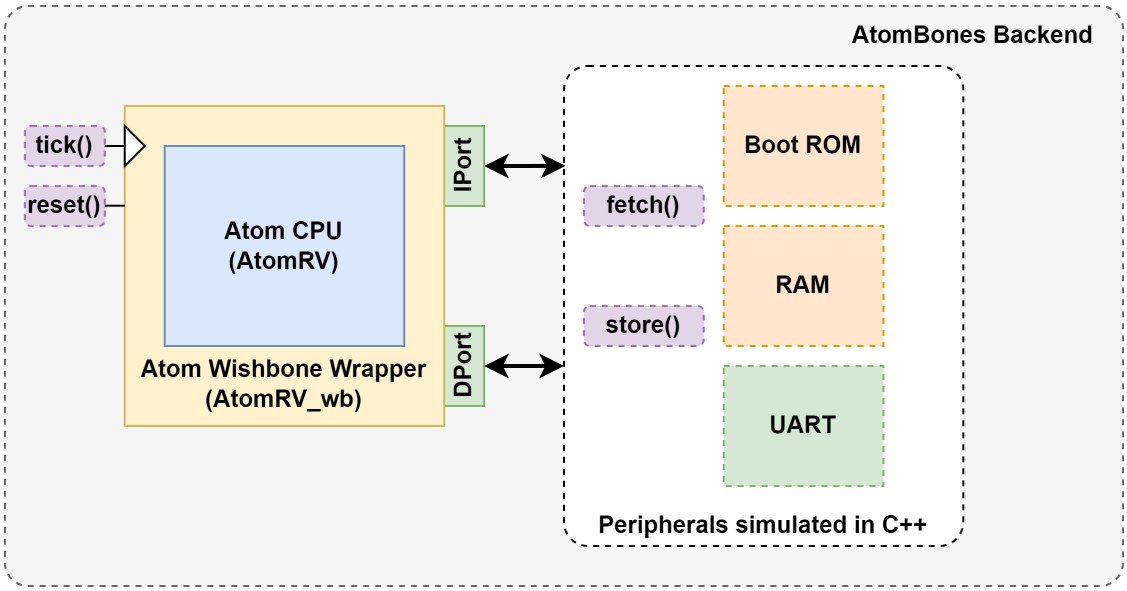
SOC TARGETS

SoC Targets are RISC-V Atom based SoCs. These contain one or more Atom cores, peripheral IPs, crossbars memories etc. From a complexity and functionality point-of-view, SoC Targets can be as simple as a wrapper to the core interface (e.g. AtomBones) and as complex as multi-core SOCs.

8.1 AtomBones

AtomBones is a stub-target that consists of a single atom core only. All other peripherals like memories and serial port are simulated in C++. It is meant to be used for simulation and debugging of the core because of its faster simulation rate. Unlike FPGA SoC targets, AtomBones provides larger memories for experimenting with the programs without being constrained by physical memory size.

The following figure shows the architecture of AtomBones.

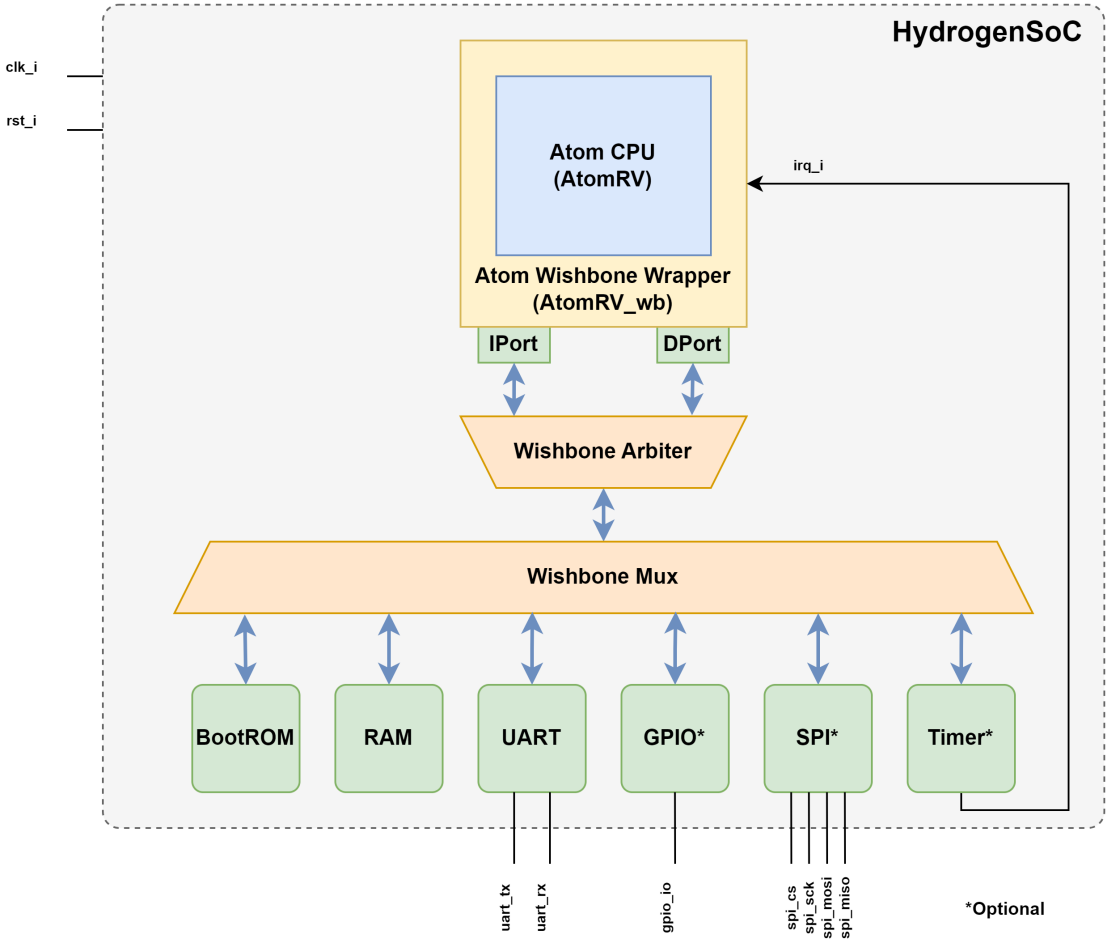


AtomBones backend implements the C++ simulation of SoC peripherals. In each cycle, the backend listens to IPort and DPort for any requests and responds to them by bit-banging. The backend provides the `fetch()` and `store()` API that are used to access these memory mapped peripherals.

8.2 HydrogenSoC

HydrogenSoC is a full SoC implementation that contains a single Atom core along with Memories, and peripheral IPs like UART, GPIOs, timers, etc. All the peripherals are connected to the CPU using a Wishbone-B4 bus. Users have the flexibility to disable/enable core features, add/remove IPs, and set memory maps through a config JSON file. HydrogenSoC is proven on multiple FPGA platforms.

The following diagram shows the architecture of HydrogenSoC.



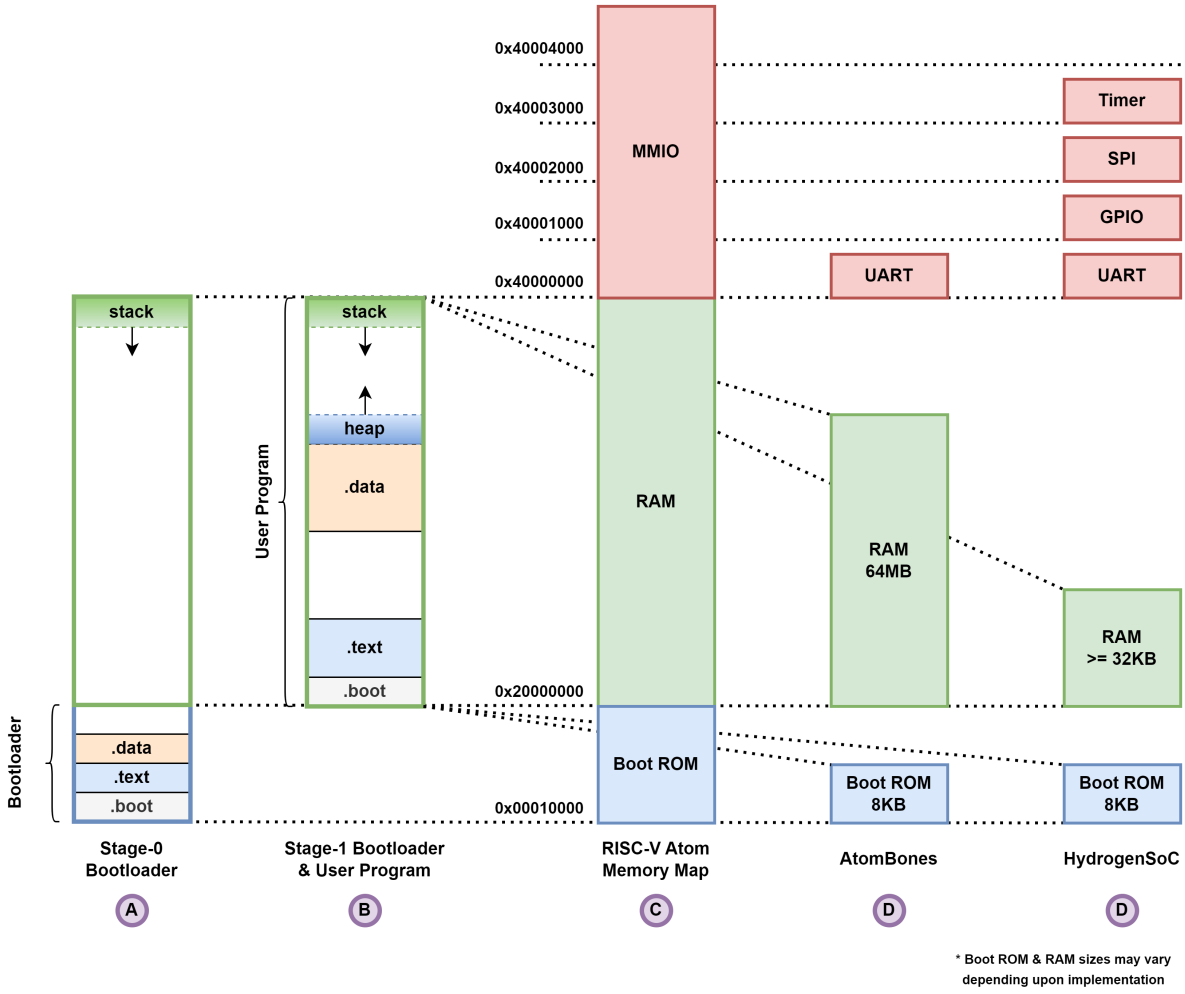
CHAPTER

NINE

MEMORY MAP AND BOOT FLOW

9.1 Memory Map

The following image shows the memory map of Atom-based SoCs.



The column **C** shows the memory map template for all RISC-V Atom based SoCs. Column **D** and **E** show the memory map for AtomBones and HydrogenSoC respectively.

9.2 Boot Flow

Upon reset, the core jumps to the reset vector (default: `0x10000`) which points to the start of BootROM. RISC-V Atom BootROM includes a **stage-0 bootloader** that is automatically built and included when building the AtomSim or running FPGA builds. The stage-0 bootloader uses RAM for stack. It initializes the platform, loads the user program in RAM and finally transfers control to the user program. The user program often includes a **stage-1 bootloader** which sets up the runtime environment and standard library before executing the application.

To know more about the stage-0 bootloader, see *this* page.

CHAPTER

TEN

RISC-V ATOM BOOTLOADER

The RISC-V Atom bootloader is a **stage-0** bootloader. It is loaded in the BootROM and executable as the first thing after reset. The source code for Atom bootloader is located in `RVATOM/sw/bootloader` subdirectory. The bootloader can be built manually using the following *make* command.

```
$ cd sw/bootloader
$ make soctarget=hydrogensoc
```

During run-time, the bootloader uses RAM to implement stack. it includes code for platform initialization, loading executable and control transfer. Bootloader provides various *bootmodes* to the user allowing the user to control the boot process.

Following is a list of actions that are performed step-by-step during the boot process:

1. Upon reset the control is transferred to reset vector, which by default points to start of BootROM.
2. **CPU starts executing the stage-0 bootloader.**
    1. Bootloader performs common initialization.
    2. Bootloader then performs any platform specific initializations.
    3. Bootloader reads the *bootmode* pins and decides the bootmode.
    4. Actions are performed according to bootmode, which includes loading the executable into RAM from correct source.
    5. Finally, the control is transferred to User program.

10.1 Bootmodes

RISC-V Atom bootloader mainly supports 4 boot modes:

Pinval	Bootmode	Function
`0b00`	FLASHBoot	Loads a binary image into RAM from FLASH memory using SPI
`0b01`	Jump to RAM	Jumps to RAM, without memory initialization
`0b10`	UARTBoot	Loads a binary image into RAM from UART stream using XMODEM protocol
`0b11`	Infinite Loop	Executes an infinite loop

10.1.1 FLASHBoot

In this bootmode, user must write the binary image for the program directly in FLASH memory at a predefined offset value. During boot, this binary image will be copied as it is to the RAM and will be executed. FLASHBoot requires the SPI IP to communicate with FLASH memory and will give an boot-panic error if its missing.

10.1.2 UARTBoot

In this mode, the bootloader obtains the program binary image from UART using the XMODEM protocol. It supports XMODEM CRC16 with default packet size of 128 bytes.

Transferring files over UART

RISC-V Atom project provides the `xmsend.py` python script that can be used to transmit binary files over UART. *xmsend* can be used as follows.

```
$ xmsend.py -b 115200 -p /dev/ttyUSB0 firmware.bin
```

Tip: Make sure that no other process (like screen) is using the serial port before invoking the xmsend script.

CHAPTER

ELEVEN

ATOMSIM: A SIMULATION TOOL FOR ATOM BASED SOCS

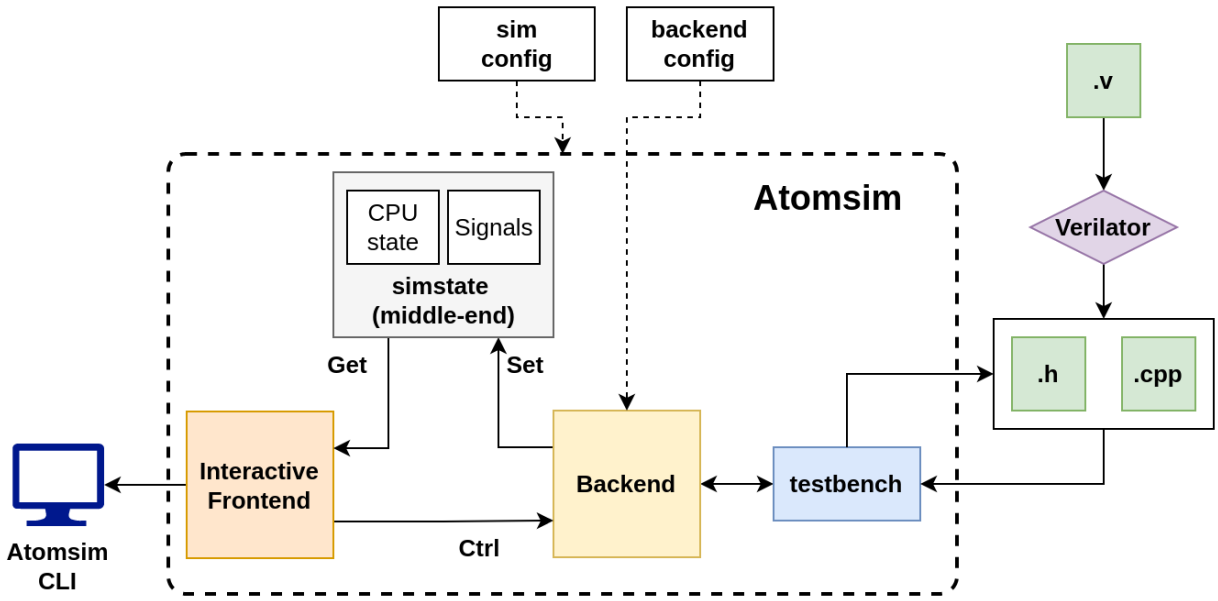
AtomSim is an interactive RTL simulator for Atom based SoCs. It provides an interface which is similar to the RISC-V Spike simulator, but simulates the actual RTL in the backend. AtomSim is a feature rich tool which makes it very powerful for debugging code on the Atom CPU.

Key Features of AtomSim are listed below:

1. Achieves a high simulation rate due to use of Verilator.
2. Target Configurable, can be easily extended for new SoC designs.
3. In-built debug mode similar to spike.
4. External Debug Support using OpenOCD & GDB **[TODO]**.
5. Supports VCD trace generation.
6. Supports memory dumps.
7. Compatible with RISC-V compliance tests framework.
8. Compatible with SCAR framework.

Tip: See *Building RISC-V Atom* for info on how to build AtomSim.

The following figure depicts the architecture of AtomSim.



11.1 AtomSim Architecture

AtomSim is designed in a modular fashion with a clear API between the layers. There are 3 main layers in AtomSim, 1) frontend, 2) middle-end, and 3) backend.

11.1.1 Frontend

Frontend is the interactive part of the simulator. User can interact with AtomSim in the following two modes of operation.

1. *Normal Mode*
2. *Debug Mode*

11.1.2 Middle-end

Middle-end contains references to signals and states of the CPU. These states and signal values are set by the backend and read by the frontend to display information.

11.1.3 Backend

Backend is the part which probes the signal values and CPU state from the RTL. All backends extend from the `Backend` class.

To view available command line options, use:

```
$ atomsim --help
```

11.2 AtomSim Topics

11.2.1 AtomSim CLI Argument Reference

Following are the arguments that may be passed to the AtomSim executable.

Short Option	Long Option	Function	Default value
General Options			
-h	–help	Show this message	
	–version	Show version information	
	–soctarget	Show current AtomSim SoC target	
	–no-color	Don't show colored output	
	–no-banner	Don't show banner	
-i	–input arg	Specify an input file	
Debugging Options			
-v	–verbose	Turn on verbose output	
-d	–debug	Start in debug mode	
-t	–trace	Enable VCD tracing	
	–trace-file arg	Specify trace file	trace.vcd
	–dump-file arg	Specify dump file	dump.txt
	–ebreak-dump	Enable processor state dump at hault	
	–signature arg	Enable signature dump at hault (Used for riscv compliance tests)	""
Sim Config Options			
	–maxitr arg	Specify maximum simulation iterations	1000000
Backend Config Options (Common)			
-u	–enable-uart-dump	Enable dumping UART data (from soc) to std-out	
-p	–vuart-port arg	serial port for virtual UART	""
-b	–vuart-baud arg	serial baud rate for virtual UART	115200
Backend Config Options (AtomBones)			
	–bootrom-size arg	Specify size of bootrom to simulate (in KB)	8
	–bootrom-image arg	Specify bootrom hex image	${RVATOM}/sw/bootloader/bootloader.hex
	–ram-size arg	Specify size of RAM memory to simulate (in KB)	81920
Backend Config Options (HydrogenSoC)			
	–bootmode arg	Specify bootmode signal	1

Note: For most up-to-date information, run `atomsim --help` command.

11.2.2 AtomSim Simulation Modes

Normal Mode

In this mode of simulation, no debug information is printed. Only serial data recieved from the soc is printed to the stdout. Using `--verbose / -v` flag shows additional useful information.

```
$ atomsim sw/examples/banner/banner.elf -v
Input File: hello-asm/hello.elf
Resetting..
Relaying uart-rx to stdout (Note: This mode does not support uart-tx)
Initialization complete!
Hello World!
      -- from Assembly

Haulting @ tick 931
```

Debug/Interactive Mode

In this mode of simulation, Contents of Program counter (in both stages), Instruction register, instruction disassembly and contents of registers (if verbosity is set) are printed to stdout. A console with symbol `:` is also displayed at the bottom if screen for user to enter various commands to control the simulation. To step through one clock cycle, user can simply press `enter` key (without entering anything in console).

To invoke interactive debug mode, invoke atomsim with *-d & -v* flag:

```
$ ./build/bin/atomsim hello.elf -d -v
Segments found : 2
Loading Segment 0 @ 0x00000000 --- done
Loading Segment 1 @ 0x00010000 --- done
Entry point : 0x00000000
Initialization complete!
:
-< 1 >-------------------------------------------
F-STAGE  |  pc : 0x00000034  (+4) ()
E-STAGE  V  pc : 0x00000000   ir : 0x00010517   [addi x1, 0x33f]
--------------------------------------------------
 x0  (zero) : 0x00000000   x16 (a6)   : 0x00000000
 x1  (ra)   : 0x00000000   x17 (a7)   : 0x00000000
 x2  (sp)   : 0x00000000   x18 (s2)   : 0x00000000
 x3  (gp)   : 0x00000000   x19 (s3)   : 0x00000000
 x4  (tp)   : 0x00000000   x20 (s4)   : 0x00000000
 x5  (t0)   : 0x00000000   x21 (s5)   : 0x00000000
 x6  (t1)   : 0x00033000   x22 (s6)   : 0x00000400
 x7  (t2)   : 0x00000000   x23 (s7)   : 0x00000000
 x8  (s0/fp): 0x00000000   x24 (s8)   : 0x00000000
 x9  (s1)   : 0x00000000   x25 (s9)   : 0x00000000
 x10 (a0)   : 0x00000000   x26 (s10)  : 0x00000000
 x11 (a1)   : 0x00000000   x27 (s11)  : 0x00000000
 x12 (a2)   : 0x00000000   x28 (t3)   : 0x00000000
 x13 (a3)   : 0x00000000   x29 (t4)   : 0x00000000
 x14 (a4)   : 0x00000000   x30 (t5)   : 0x00000000
```

(continues on next page)

(continued from previous page)

```
 x15 (a5)   : 0x00000000   x31 (t6)   : 0x00000000
:
```

Interacting With Debug Console

Displaying contents of a register

Contents of register can be displayed simply typing its name (abi names are also supported) on the console. ex:

```
: reg x0
x0 = 0x000045cf
: reg ra
ra = 0x0000301e
```

Use ':' to display a range of registers. ex:

```
: x0 : x1
```

Displaying Contents of a memory location

```
: m <address> <sizetag>
```

Address can be specified in hex or decimal. Use sizetag to specify the size of data to be fetched, b for byte, h for half-word and w for word (default is word).

```
: m 0x30 b
mem[0x30] = 01
```

Use ':' to display contents of memory in a range. ex:

```
: m 0x32:0x38 w
mem[0x30] = 01 30 cf 21
mem[0x38] = 11 70 ab cf
```

Generating VCD traces

Tracing can be enabled by:

```
: trace out.vcd
Trace enabled : "./out.vcd" opened for output.
```

or by passing –trace <file> option while invoking atomsim.

Tracing can be disabled by:

```
:notrace
Trace disabled
```

Controlling execution

You can advance the simulation by one clock cycle by pressing the enter-key. You can also execute until a desired equality is reached:

1. until value of a register <reg> becomes <value>

```
: until <reg> <value>
```

2. until value of a memory address <address> becomes <value>

```
: until <address> <value>
```

3. while <condition> is true

```
: while <condition>
```

4. Execute for specified number of ticks

```
: for <ticks>
```

5. You can continue execution indefinitely by:

```
: r
```

6. To end the simulation from the debug prompt:

```
: q
```

or

```
: quit
```

> **Note:** At any point during execution (even without -d), you can enter the interactive debug mode with `ctrl + c`.

7. Miscellaneous verbose-on & verbose off commands can be used to turn on /off verbosity.

11.2.3 AtomSim Code Structure

This is a Placeholder

11.2.4 Adding a New Target to AtomSim

This is a placeholder

CHAPTER

TWELVE

SCAR: SEARCH COMPILE ASSERT RUN

SCAR is a processor verification framework in python. SCAR performs a set of assembly level tests to verify the processor implementation. Each assembly test checks for one particular functionality of the processor. SCAR does this by examining a state dump file after the processor is done with executing a test code. This state dump file then checked assuming a set of assertions in the form of expected register values. These assertions are provided in the assembly file itself. SCAR is also used to verify the ISA-compliance.

12.1 SCAR Workflow

1. **Search:** SCAR searches for all the available assembly level test in the directory and makes a list
2. **Compile:** It then compiles all the found tests with a user-defined linker script.
3. **Execute:** In this step, The elf files are executed on the target simulator which creates a state dump file after execution.
4. **Verify:** Finally, Assertions are read from the assembly file containing the test. These are then used to check for mismatches in the generated state dump file.

12.2 Assembly test format

The assembly file must satisfy the following criteria:

1. File must have a `_start` label before the start of code.
2. File must have a `ebreak` instraction after the end of code.
3. File must have an assertion section at the bottom with the following format.

12.2.1 Assertion Section Format

The assembly file must contain a set of assertions at the bottom in the following format:

```
 .global _start
_start:

li t0, 0x00d01010
li t1, 0x1ddc1044
li t2, 0xdeadbeef
li t3, 0x22101301
```

(continues on next page)

(continued from previous page)

```
li t4, 0xfaf01569
li t5, 0x078b102a
li t6, 0xdae013c0

add a0, t0, t1
add a1, t1, t2
add a2, t2, t3
add a3, t3, t4
add a4, t4, t5
add a5, t5, t6

nop
nop
ebreak

# $-ASSERTIONS-$
# eq a0 0x1eac2054
# eq a1 0xfc89cf33
# eq a2 0x00bdd1f0
# eq a3 0x1d00286a
# eq a4 0x027b2593
# eq a5 0xe26b23ea
```

12.2.2 State-Dump file format

[TODO]

CHAPTER

THIRTEEN

CONVELF: A UTILITY TOOL FOR ELF CONVERSION

ConvELF is a flexible tool written in python to convert ELF executable files to Verilog friendly $readmemh/$readmemb compatible files.It is a target-configurable tool which can be configured by specifying a memory map. *convelf.py* can be either imported into user programs as a module or user may directly use it as a script.

13.1 convelf.py as a Python Script

convelf.py can be invoked as a script by passing elf file as a command-line argument.

```
$ python3 convelf.py file.elf
```

13.2 convelf.py as a Python Module

Using ConvELF is easy, just import it into your code and instantiate the ConvELF class.

```
import convelf as CE

# Create a convELF class object
ce = CE.ConvELF(ELF_file, Memory_Map)
```

Now, Generate $readmemh compatible files, using:

```
# generate output hex files for $readmemh.
ce.convert('hex')
```

Alternatively to generate $readmemb compatible files, using:

```
# generate output bin files for $readmemh.
ce.convert('hex')
```

ConvELF provides the following object fields to further configure the tool

```
# RISCV Toolchain Prefix
ce.RVPREFIX = 'riscv64-unknown-elf-'

# Sections to load into hex files
ce.INCLUDE_SECTIONS = ['.text', '.rodata', '.sdata', '.data']
```

(continues on next page)

(continued from previous page)

```
# Name of temporary hex file
ce.TEMP_FILE = 'temp.hex'

# Echo shell commands during execution
ce.ECHO_CMD = False

# Delete temp file after conversion
ce.DELETE_TEMPFILE = True
```

13.2.1 Memory Map Format

Memory map of the target should be specified in python dictionary format. one key-value pair (element) in the dictionary must represent one memory block for which a initialization file should be generated.

- The **key** should be a string which should represent the name of the memory block.
- The **value** should be a python list containing atleast 3 elements.
  1. **base address** of the memory block.
  2. **size** of the memory block.
  3. **name of the initialization file** to be generated.

13.2.2 Example

Let's assume a target that has 3 different memories.

Table 1: Example Memory Map

Memory	Base Address	Size	Init file
ROM	0x00000000	0x00010000	boot.hex
FLASH	0x01000000	0x00010000	code.hex
EEPROM	0x02000000	0x00010000	data.hex

This memory map can be specified in python as:

```
HydrogenSoC_MemMap = {
    'ROM'   : [0x00000000, 0x00001000, 'boot.hex'],
    'FLASH' : [0x01000000, 0x00001000, 'code.hex'],
    'EEPROM': [0x02000000, 0x00001000, 'data.hex']
}
```

CHAPTER

FOURTEEN

LIBCATOM: C STANDARD LIBRARY FOR RISC-V ATOM

Libcatom is a minimal C standard library for RISCV-Atom. It consists of startup code, basic stdio library, heap allocator, soc-specific libraries such as drivers for peripherals such as GPIO, UART, SPI etc. and linker scripts. The soc-specific sources (such as drivers) are located inside the corresponding folder. `platform.h` header defines most of the platform-specific macros for libcatom. Libcatom does not define the software multiply and soft floating point operations and therefore uses the standard library provided with the compiler.

14.1 Building Libcatom

Libcatom can be built as follows.

```
$ make soctarget=hydrogensoc sim=1
```

Tip: You don't need to build Libcatom separately in most cases as it will be built automatically (with correct soc target) while building AtomSim.

Note: You must use `sim=1` option to build the library for simulation. If you see framing errors in UART output during simulation, most likely you haven't built the library and anything that uses it with `sim=1`.

CHAPTER

FIFTEEN

RISC-V ATOM BUILD FLOW

TBA

CHAPTER

SIXTEEN

PERFORMANCE STATISTICS

16.1 Dhrystone

TBA

CHAPTER

SEVENTEEN

FPGA IMPLEMENTATION RESULTS

17.1 Xilinx Spartan Series

CHAPTER

EIGHTEEN

INDICES AND TABLES

- genindex
- modindex
- search