RISC-V Atom

Release v1.2

Saurabh Singh

Apr 08, 2023

OVERVIEW


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


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 based on the open-source loyality-free RISC-V ISA. It is complete hardware prototyping and software development environment based around the *Atom* CPU. RISC-V Atom is a customizable processor platform which is easy to learn, use and tinker even for a begineer. It also provides a wide variety of software examples for testing, a rich documentation and a comprehensive guide for getting stated.

1.1 Key Highlights

1. Based around *Atom*: A 32-bit 2-stage pipelined RISC-V CPU.
2. Provides multiple SoC configurations.
3. Provides an interactive feature-rich RTL simulation frontend called **AtomSim**.
4. Features a python based processor verification framework called *SCAR*.
5. RISC-V GCC is used as default toolchain (prebuilt toolchains are also provided)
6. A rich documentation & getting started guide.
7. An array of software examples to run & test.
8. A rich software framework with a C library **libcatom** for all SoC peripherals.
9. Customizable, Easy to learn and tinker.
10. Open-Sources under MIT License.

Note: Although significant efforts have been and will be made towards optimizing the RTL for LUT consumption & timing on FPGAs, however, the **codebase is not recommended for any production use as of now**.

To get started, Check out the getting started guide *Prerequisites*.

CHAPTER

TWO

COMPONENTS

This page gives a high-level overview of various components of the RISC-V Atom project.

2.1 RISC-V Atom Core

Atom is a 32-bit embedded-class softcore processor written in Verilog HDL. It is designed to cater to embedded class applications. It is fully compliant with the open-source RISC-V Instruction Set Architecture (RV32I) and passes all official RISC-V compliance tests. Atom is based on a two-stage pipelined architecture inspired by the ARM cortex m0+ processor. It is aimed towards implementation on FPGAs.

Click here for in-depth documentation

2.2 SoC Targets

- **AtomBones**: It is a stub-target that consists of a single atom core only. The instruction memory, data memory, and serial port are simulated in C++. It is should be used for simulation and debugging purposes only.
- **HydrogenSoC**: It is a basic SoC implementation that contains a single Atom core along with instruction memory, data memory, serial ports, GPIO pins, etc. All the peripherals are connected with the CPU using a Wishbone-B4 bus.

Click here for in-depth documentation

2.3 AtomSim

AtomSim is a Spike-like simulator written in C++. In contrast to Spike, AtomSim Simulates the actual RTL in the backend. AtomSim provides a hardware-software co-simulation environment for developing applications and extensions on the Atom platform. It uses the Verilator tool to compile the RTL into a shared object which then gets linked with the C++ based simulator frontend to simulate the system. AtomSim also features a command-line interface to control various aspects of the simulation like start, stop, step, run indefinitely, run for a specified number of cycles, enable/disable vcd tracing, etc. It also supports run-time instruction disassembly with the help of the RISC-V objdump tool.

Click here for in-depth documentation

2.4 SCAR

SCAR (Search, Compile Assert, and Run) is a processor verification framework written in python. It performs a set of assembly-level tests to verify the processor implementation. Each assembly test usually checks for one particular function of the processor. SCAR does this by examining a state dump after the processor is done with executing a test code. This state dump is 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 RISC-V ISA compliance in this project.

Click here for in-depth documentation

2.5 ConvELF

ConvELF is a python script that is used to convert an ELF executable file to verilog friendly memory initialization files. These files can be in either *hex* or *bin* format and can be used to initialize a verilog memory with the help of *$readmemh* & *readmemb* functions. ConvELF is a flexible program configured by a memory map specified as a python dictionary. It can even fragment an elf file into multiple memory initialization files depending on the memory map provided.

Click here for in-depth documentation

CHAPTER

THREE

DIRECTORY STRUCTURE

```
riscv-atom          : root directory
    |
    ├─ docs             : RISCV-Atom documentation & user manual
    |   ├─ diagrams         : executable binaries
    |   └─ pages            : reStructuredText sources
    |
    ├─ rtl              : RISCV-Atom Verilog Sources
    |   ├─ core             : RISCV-Atom core components
    |   ├─ uncore           : RISCV-Atom non-core components (SoC peripherals)
    |   └─ dpi              : SystemVerilog DPI sources
    |
    ├─ scripts         : scripts for commonly used commands
    |
    ├─ sim              : Atomsim source code
    |   ├─ build            : AtomSim build files (autogenerated)
    |   ├─ docs             : AtomSim Source Documentation (Doxygen)
    |   ├─ include          : Third party Libraries for AtomSim
    |   └─ run              : Atomsim run logs, dumps and VCD traces (autogenerated)
    |
    ├─ sw               : Atomsim source code
    |   ├─ examples         : example programs
    |   └─ lib              : libc for RISCV-Atom (libcatom)
    |       ├─ include          : libcatom headers
    |       ├─ libcatom         : libcatom sources
    |       └─ link             : Linker scripts
    |
    ├─ synth            : RISC-V Atom Synthesis
    |   ├─ xilinx           : Synthesis project for xilinx FPGAs
    |   └─ yosys            : Yosys synthesis scripts
    |
    ├─ test             : RISCV Atom tests
    |   ├─ riscv-target     : Official RISC-V compliance test files
    |   └─ scar             : SCAR tests directory
    |
    └─ tools            : utility tools
        └─ elfdump          : elfdump utility
```

CHAPTER

FOUR

PERFORMANCE STATISTICS

4.1 Dhrystone

TBA

CHAPTER

FIVE

FPGA IMPLEMENTATION RESULTS

5.1 Xilinx Spartan Series

CHAPTER

SIX

PREREQUISITES

This page discusses how to set up your system in order to get riscv-atom up and running.

6.1 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 version of ubuntu with no or few additional packages.

6.1.1 Run apt update

This step is needed to make sure the apt package list is up to date.

```
$ sudo apt update
```

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

GNU C/C++ compilers and Make and other essential build tools are conveniently packaged as `build-essential` meta package.

```
$ sudo apt install git python3 build-essential
```

6.1.3 Install Verilator

Verilator will be used By Atomsim to *Verilate* Verilog RTL into C++. We recommend installing latest stable verilator version using git quick install method

6.1.4 Install GTK Wave

GTKwave is a GUI tool to view waveforms stored as Value Change Dump (VCD) files.

```
$ sudo apt install gtkwave
```

6.1.5 Install Screen

Screen is a command line utility that can be used to connect to serial ports on linux. It will be used to estabilish a two-way serial communication with the AtomSim.

```
$ sudo apt install screen
```

6.1.6 Install RISC-V GNU Toolchain

We will be installing the RV64-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
```

6.2 Optional Packages

Note: The following packages are optional and are only required for generating documentation using doxygen & sphinx

6.2.1 Install Doxygen

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

```
$ sudo apt install doxygen
```

6.2.2 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
```

6.2.3 Install sphinx & other python dependencies

Sphinx is used to generate the RISC-V Atom Documentation and User-Manual in PDF & HTML.

```
$ cd docs/ && pip install -r requirements.txt
```

CHAPTER

SEVEN

BUILDING RISC-V ATOM

7.1 Clone the repository

```
$ 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`.

7.2 Edit Config.mk

Edit Config.mk file and provide paths appropriately

7.3 RISC-V Atom environment variables

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 everytime you work with the project. This can be done as follows:

```
$ source sourceme
```

With this method, everytime 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 everytime you open a new terminl.

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

In the above command replace `rvatom-path` with the path to your RISC-V atom directory.

7.4 Building the Simulator

Let's build AtomSim simulator for `atombones` target.

```
$ make soctarget=atombones
```

This will create `RVATOM/sim/build` and `RVATOM/sim/run` directories for Atomsim build files and runtime files respectively. You can find the Atomsim executable in the former directory.

Assuming you've sourced the `RVATOM/sourceme` file, try the following command to verify the build.

```
$ atomsim --help
AtomSim v_._
Interactive RTL Simulator for Atom based systems [ atombones ]
Usage:
  atomsim [OPTION...] input
...
```

CHAPTER

EIGHT

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!

8.1 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 compile
```

The above command should generate a `hello.elf` file in the `hello-asm` directory. Now fire up atomsim and provide the generated elf file as argument.

```
$ atomsim hello-asm/hello.elf
Hello World!
  -- from Assembly
```

Alternatively, use make run to run the example.

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

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

8.2 Banner Example

```
$ make target=atombones ex=banner compile
$ atomsim banner/banner.elf
```

Output:

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

ROM size: xxxxx bytes   (xx KB)
RAM size: xxxxx bytes   (xx KB)
exiting...
```

8.3 The Runexamples Script

Instead of testing all examples one-by-one, we can use the provided `atomsim-runexamples` script to automatically compile and simulate all examples.

Simply invoke the script as following

```
$ atomsim-runexamples
```

atomsim-runexamples script internally uses the same makefile that we used earlier, with the run-all target. Therefore we can alternatively get the same result by using the make command (in `RVATOM/sw/examples` directory).

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

8.4 Using Atomsim Vuart

By default 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. Alternatively, Atomsim can estabilish a two-way communication with AtomSim through a linux serial port. This functionality is provided by the Vuart module in Atomsim.

8.4.1 Generating virtual serial ports

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

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

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

8.4.2 Interacting with Stdout and Stdin over virtual ports

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

```
$ screen $RVATOM/userport 9600
```

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

```
$ atomsim hello-asm/hello.s --vuart=$RVATOM/simport
```

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

To close the screen command press `ctrl+a`, type `:quit` and press `enter`.

8.4.3 Adding New Examples

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

```
$ mkdir newexample
```

Next, put your source files under this directory.

```
$ cat newexample.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 = newexample.c
executable = newexample.elf
```

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

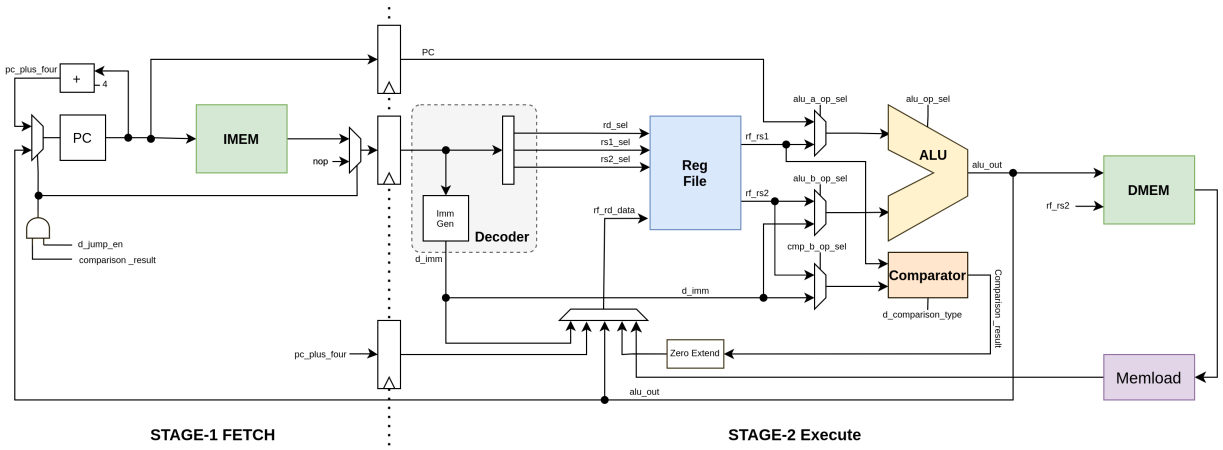
CHAPTER

NINE

RISC-V ATOM (CORE)

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 (RV32I), 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.



9.1 Processor Pipeline Stages

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

9.1.1 Stage-1: Fetch

Fetch unit is responsible for fetching instructions from instruction memory through the IBUS. It uses a 32-bit register called “Program counter” to keep track of the address of the instruction being fetched. After the instruction is successfully fetched, Program counter is incremented by 4.

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

In this stage, the instruction is decoded, all the signal are assigned in order to configure data-path to execute the instruction. & registers are fetched. A 32 bit immediate is generated by the ImmGen unit. ALU then execute the instruction which is followed by write-back into the register file. Branch calculation also happens in this stage and if branch is taken, the pipeline is flushed. Comparator module in this stage is used for all the instructions that involve comparison like slt, slti, beq, bltu etc.

9.2 Processor Interface

RISCV-Atom module is defined in the file `RVATOM/rtl/core/atomRV.v`. It has has two independent ports which it uses to access memory.

1. Instruction port &
2. Data port

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 Wishbobne. These wrappers are specified in the following files.

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

9.2.1 RISC-V Atom RTL

RISC-V Atom is written in Verilog. Its RTL is divided into 2 categories, core and uncore, both of which reside in the *rtl* directory in *core* and *uncore* subdirectories respectively.

Rtl directory

The top level verilog modules (atombones & hydrogensoc) are present in the *rtl* directory. Each of these top level modules are configured by their respective configure headers (<name>_Config.vh file). These configuration headers contain the macros used in the top-module definitions to control the generation of various sub-components and their parameters.

Core directory

core directory 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* defines various signal eunumerations and other parameters internal to the processor. *Utils.vh* defines some useful utility macros.

Uncore directory

The *uncore* subdirectory contains all the peripheral components such as uart, gpio, ram, rom etc. This also includes the wishbone wrappers of some non-wishbone components. SoC implementations of the Atom processor usually instantiate these hardware modules in their implementations.

RTL Features

DPI Logger

DPI Logger is a SystemVerilog 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 topmodule or in the Makefile VFLAGS variable 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 funciton. Example of logging the jump 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
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 containt all the dumped information.

CHAPTER

TEN

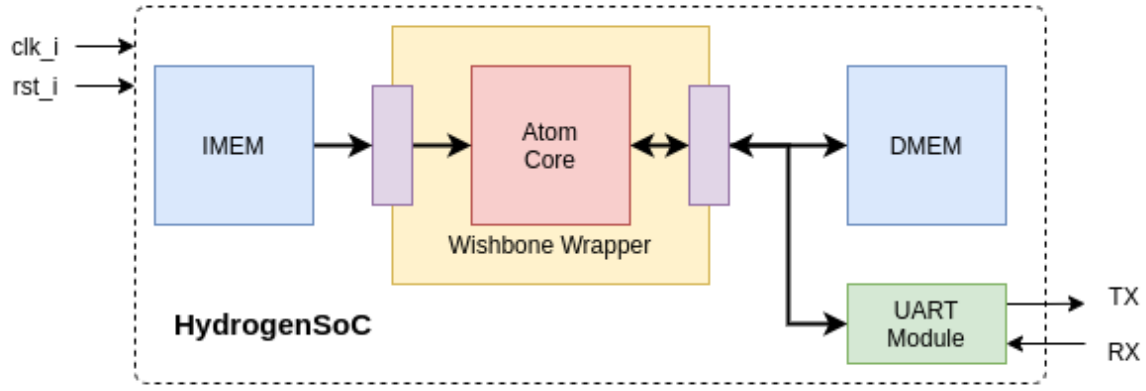
RISC-V ATOM SOC TARGETS

SoC Targets are systems that use the RISC-V atom core alongside different peripherals. 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 fully-fledged SOCs.

10.1 AtomBones

../diagrams/AtomBones.png

10.2 HydrogenSoC



CHAPTER

ELEVEN

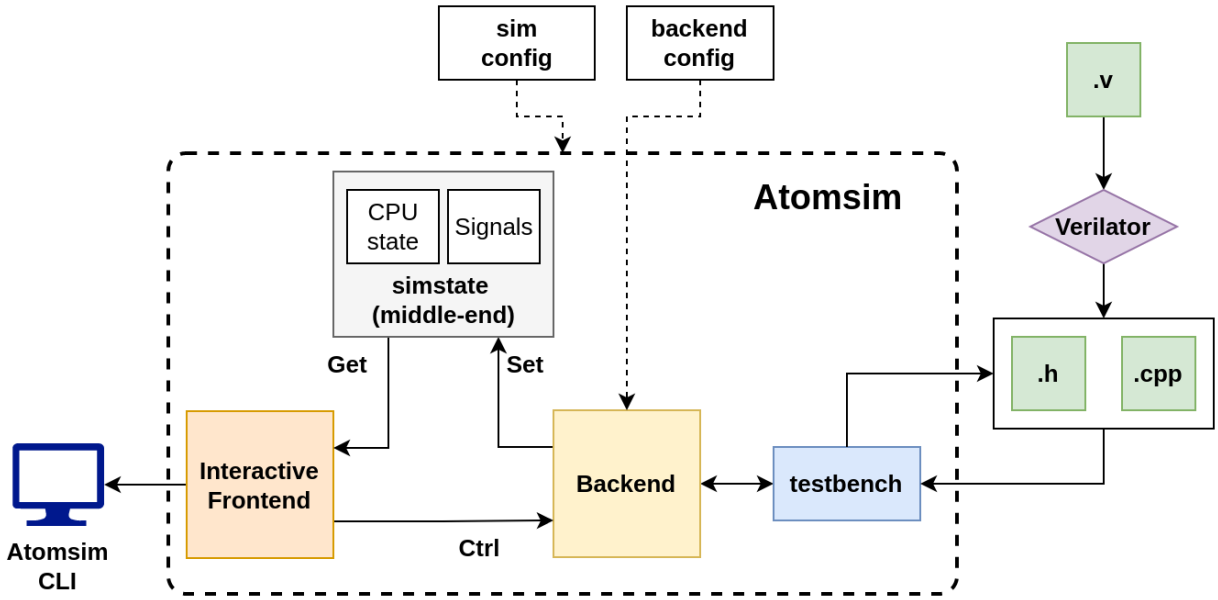
ATOMSIM: A SIMULATION TOOL FOR ATOM BASED SOCS

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 modular and flexible simulaion solution based on Verilator, due to which it can achieve a very high simulation rate. 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.

The following figure depicts the architecture of atomsim.



See *Building RISC-V Atom* for info on how to build atomsim.

To view available command line options, use:

```
$ atomsim --help
```

11.1 Modes of Operation

Atomsim supports two modes of operation:

11.1.1 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
 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.

11.1.2 Normal Mode

In this mode of simulation, no debug information is printed. Only serial data recieved from the system 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
```

Redierecting AtomSim Output

All verbose information is printed on **stderror** stream while the output of the program is printed on the **stdout**. Therefore one can easily redirect them to two different files if needed. This can be done as follows:

```
$ atomsim sw/examples/banner/banner.elf -v 1> output.log 2> sim.log
```

Reverting to Debug Mode from Normal Mode

During a program's execution on atomsim, if user presses `ctrl + c` atomsim quits by default. This behaviour can be changed by invoking AtomSim with `___to_be_added____` options. With this option, anytime during the simulation, it is possible to revert to debug mode from normal mode by pressing `ctrl + c`. To quit AtomSim, press `ctrl + c` twice or press `ctrl + c` once and then use the *quit* command on debug console.

11.2 AtomSim Topics

11.2.1 AtomSim CLI Argument Reference

The following are the arguments that may be passed to the AtomSim executable.

```
Usage:
$ atomsim [OPTION...] input_file

Config options:
   --maxitr arg      Specify maximum simulation iterations (default: 10000000)
   --vuart arg       use provided virtual uart port (default: Null)
   --vuart-baud arg  Specify virtual uart port baudrate (default: 9600)

Debug options:
-v, --verbose        Turn on verbose output
-d, --debug          Start in debug mode
-t, --trace          Enable VCD tracing
    --trace-dir arg  Specify trace directory (default: build/trace)
    --dump-dir arg   Specify dump directory (default: build/trace)
    --ebreak-dump    Enable processor state dump at hault
    --signature arg  Enable signature sump at hault (Used for riscv
                     compliance tests) (default: "")

General options:
-h, --help           Show this message
    --version        Show version information
    --simtarget      Show current AtomSim Target
-i, --input arg      Specify an input file
```

11.2.2 AtomSim Code Structure

This is a Placeholder

11.2.3 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

INDICES AND TABLES

- genindex
- modindex
- search