Universal Verification Methodology

Capstone Project

Project Manual

10x Engineers (Pvt.) Ltd.

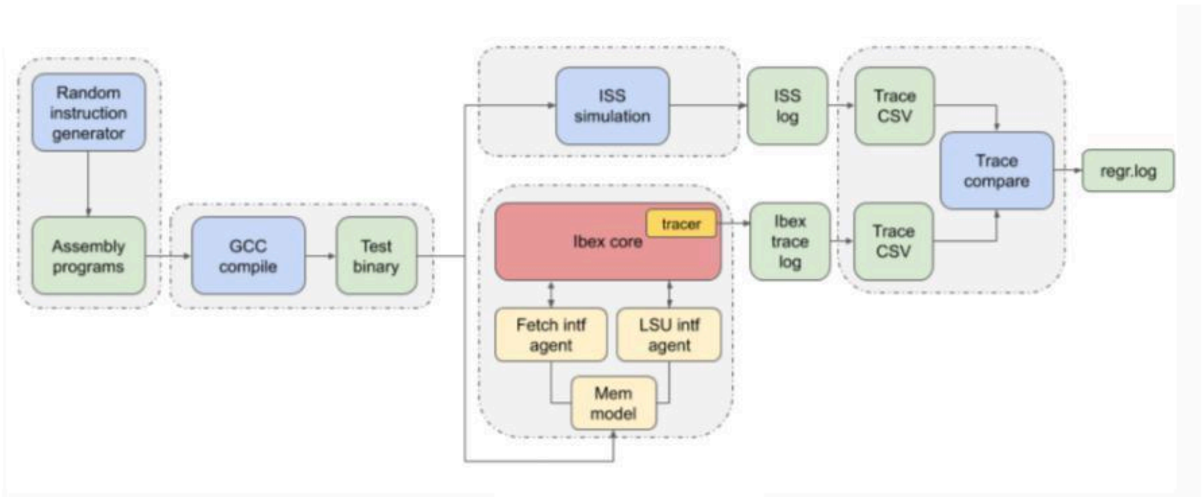
Table of Contents

Project **Design Verification Environment for Core Verification**

Objective: **Create a complete UVM based verification environment for core verification**

You will be creating a UVM based verification environment for the verification of the Ibex core. At a high level, this testbench should use the open source RISCV-DV random instruction generator to generate the tests and compile instruction binaries, loads them into a simple memory model, stimulates the Ibex core to run this program in memory, and then compares the core trace log against a golden model ISS (SPIKE) trace log to check for correctness of execution.

Block Diagram



Ibex Core

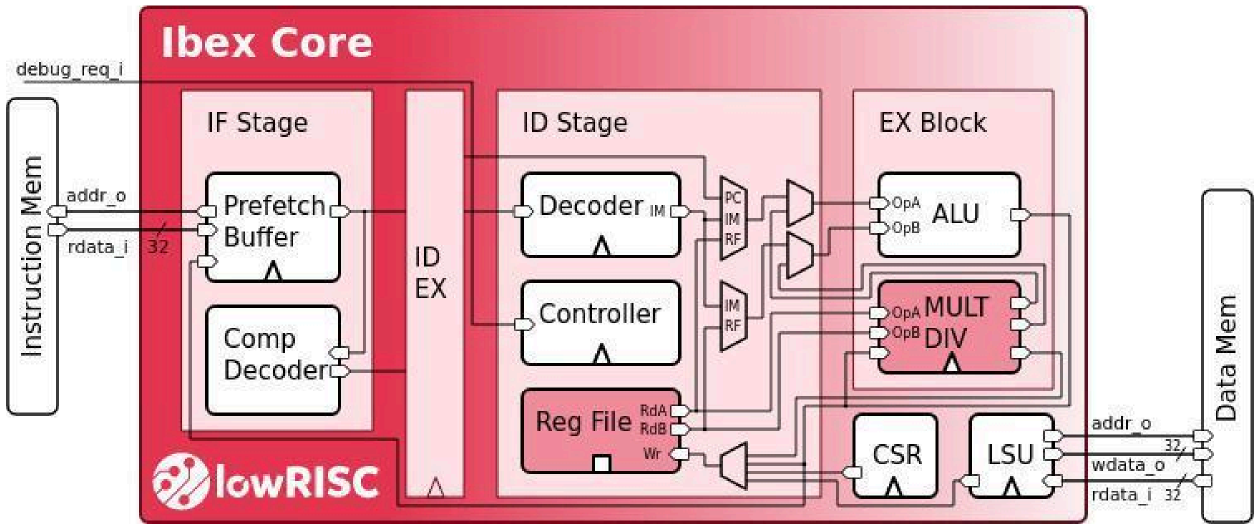
Ibex is a 32-bit RISC-V CPU core. The CPU core is heavily parameterizable and well suited for embedded control applications. Ibex supports the Integer (I) or Embedded (E), Integer Multiplication and Division (M), Compressed (C), and B (Bit Manipulation) extensions.

Ibex offers several configuration parameters to meet the needs of various application scenarios. The options include different choices for the architecture of the multiplier unit, as well as a range of performance and security features.

We will use the *small* parameterization which is:

- Two-stage pipeline without additional branch target ALU and 3 cycle multiplier (4 cycles for mulh), resulting in 2 stall cycles for mul (3 for mulh)

The block diagram below shows the *small* parametrization with a 2-stage pipeline.



UVM Testbench Architecture

Memory Model

The code is from OpenTitan and can be found in the *vendor/lowrisc_ip/dv/sv/mem_model* directory. The testbench instantiates a single instance of this memory model that loads the compiled assembly test program into the memory at the beginning of each test. This acts as a unified instruction/data memory that serves all requests from the memory interface agents.

Interfaces

Get the signals from design files and create the following interfaces:

- Memory interface
  - Interface to monitor memory requests
  - One is Instruction interface and other is LSU interface
- DUT probe interface
  - Interface to probe DUT internal signals

One can make other interfaces as well, depending upon the need and strategy.

Memory Agents

There are two approaches for creating memory interface agents:

- Slave Sequence Method
- Monitor-Sequence Analysis port method (explained below)

First, develop an Instruction agent using Approach 2, which is the Monitor-Sequence Analysis port method. Just start executing instructions from the boot address after loading the instructions. When ecall instruction is encountered by the core, detect it and stop the test by dropping the raised objection in your test class.

In the case of LSU agent use the salve sequence method.

Base test

Your base test should coordinate the entire flow for a single test. It starts from loading the compiled assembly binary program into the testbench memory model and ends at encountering the end condition.

For Instruction agent:

- Load binary file
- Start the core to execute instructions
- Wait for the end condition

For LSU agent:

- Load binary file
- Start the core to execute instructions
- Start base sequences
- Wait for the end condition

You can make functions for all the steps shown above and call these functions in the correct order in the respective UVM phases of the base test.

Sequence, Sequencer, and Driver class for Active Agent

For Instruction agent :

In sequencer class, make a FIFO that will aid the communication between sequence, driver and monitor. This FIFO can be used by the *p_sequencer* handle.

As mentioned above, sequences will act as slave sequences, i.e., they will respond to the memory requests (read/write) from the core.

In your monitor class, keep monitoring the requests on the interface from DUT and update the sequencer FIFO, in your sequence class, read the sequencer FIFO and generate the sequence serving the memory requests and then drive them in the driver class.

In simple scenarios (without interrupts and error injection), sequences read memory and write it back into the memory while the instructions are being executed in the core. For simplicity, interrupts and error injections are not being encountered for now.

Class skeletons for sequence, sequencer, driver, and monitor are shown below,

Sequence class:

```
.
.
Obj = Read from sequencer fifo
start_item(Obj); finish-item(Obj);
.
.
```

Sequencer class:

```
.
.
.
instantiate a FIFO
.
.
.
```

Driver class:

.
.
.

Check the status of the request from the memory interface and set the grant signal accordingly

```
get_next_item(<sequence via sequencer>);
Drive onto the memory interface;
item_done;
```

.
.
.

Monitor class:

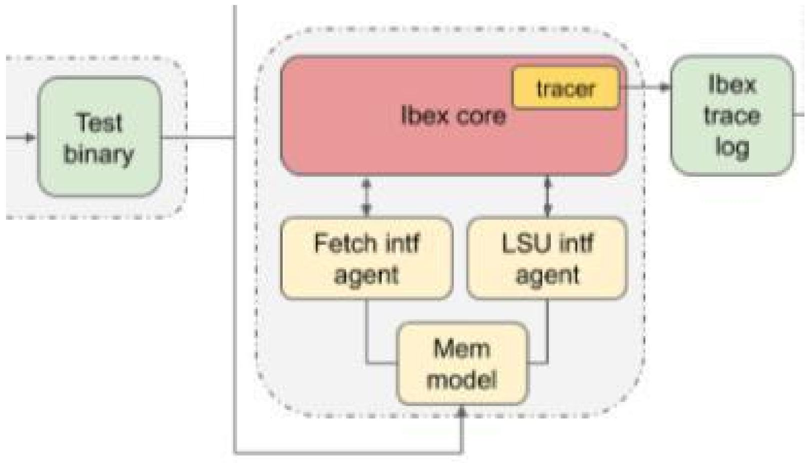
.
.
.

Monitor data from the memory interface

direct the data towards the sequencer FIFO

.
.
.

Part I: Create UVM based environment and run assembly tests



Work in the *project-uvm-coreverif/part-i* directory. The structure of the directory is as follows:

- `doc`: documentation related to core design
- `rtl`: actual rtl design files in system verilog
- `tests`: assembly language tests
- `verif`: verification environment files
- `Makefile`: work automation file to compilation and simulation (see Appendix A)
- `test.bin`: example binary file given for initial testing

Compiling/Elaborating the RTL design

Complete rtl design of the core is given in the `rtl` directory. Modify the `Makefile` as described below to compile the given rtl and verify that there are no compilation errors.

- Add the boot address of the core based on the documentation to the variable `BOOT_ADDR`

Once the `Makefile` is modified, open the terminal in your working directory i.e. `project-uvm-coreverif/part-i` and run the following command:

```
make rtl-compile
```

Now modify the `Makefile` again for elaboration of the design as given below:

- Change the `-compile` flag to `-elaborate`

Inform your instructor if you are facing any compilation or elaboration errors.

Creating the UVM environment

You will work in the `verif` directory and understand the testbench architecture given above to create all the files necessary for a complete uvm based verification environment.

Compiling the environment

Modify the `Makefile` as described below to compile the written testbench environment and verify that there are no compilation errors.

- Add your testbench file name to the variable `TB_TOP`
- Add a rule named `tb-compile` with the dependency of `gen-dirs`
- Add the `irun` command as used in the `rtl-compile` rule with a few changes
  - Include your testbench file using `TB_TOP`
  - Include verif directory using `-incdir` flag
  - Include uvm home path using `-uvmhome` flag

Once the `Makefile` is modified, open the terminal in your working directory i.e. `project-uvm-coreverif/part-i` and run the following command:

```
make tb-compile
```

Fix any compilation errors before moving forward.

Now modify the `Makefile` again for elaboration of the design as given below:

- Change the `-compile` flag to `-elaborate`

Fix any elaboration errors before moving forward.

Testing the environment

After completing the verification environment, modify the `Makefile` as described below to test the already given binary file (`test.bin`) to test your environment.

- Add uvmhome path to variable `UVMHOME` (as you did in `lab00-envsetup` of Module2)
- Add a rule named `sim`
- Add the `irun` command as used in the `tb-compile` rule with a few changes
  - Remove the `-compile` or `-elaborate` flag
  - Include the binary file using `+bin=<filename>`

Once the `Makefile` is modified, open the terminal in your working directory i.e. `project-uvm-coreverif/part-i` and run the following command:

```
make sim
```

You should see `UVM TEST PASSED` if your environment is working properly.

Compiling the assembly test

Once the environment is tested, compile and generate the binaries of the assembly tests given in the `tests` directory and run them using the environment.

- Add path where you downloaded riscv toolchain to the variable `RISCV_TOOLCHAIN`
- Set the `ASM_TEST` variable to a test name e.g. `init_regs`
- Modify the rule `sim` to use `BINARY` variable instead of the hardcoded binary file `test.bin`

Once the `Makefile` is modified, open the terminal in your working directory i.e. `project-uvm-coreverif/part-i` and run the following command:

```
make sim
```

You should see `UVM TEST PASSED` if your environment is working properly.

Note: `init_regs` test is just initializing all the registers in the register file to 0. Confirm this behaviour by adding the register file signals to the waveform.

You can write your own assembly code and run it. You can also just parse the test name to the makefile instead of modifying it every time you want to run a new test i.e.

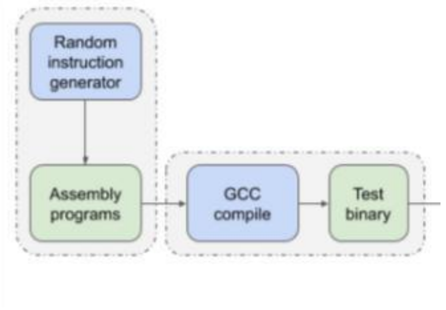
```
make sim ASM_TEST=<testname>
```

Complete all the steps given above before moving to Part II of the project.

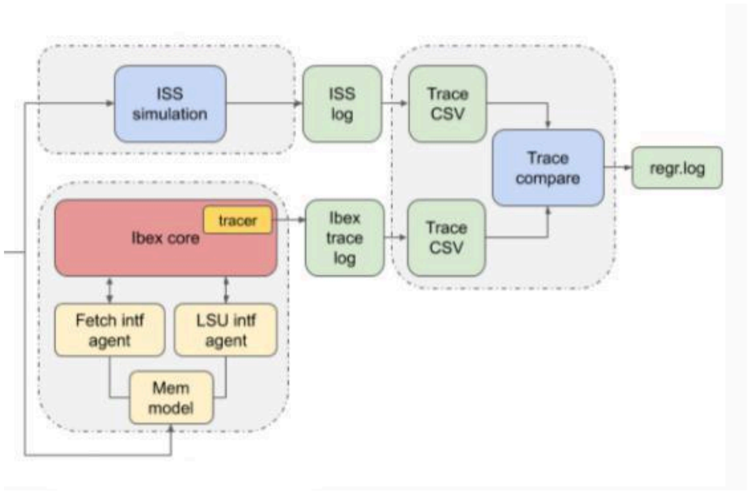
Next Steps (Optional)

Work with your project manager to complete the following parts of the project.

Part II: Integrate riscv-dv test generator



Part III: Integrate ISS model SPIKE for co-simulation checking



Part IV: Collect Code and Functional Coverage

The goal is to achieve maximum coverage for the ibex core. This includes testing all RV32IMCB instructions, privileged spec compliance, exception and interrupt testing, debug mode operations etc.