A working MU0 plus test-benches

The purpose of this section is to give a concrete example of what it might mean to develop and test a CPU, in this case a MU0. You've already done parts of this in first year, and some of you may have already done some or all of this (depending how far you got with ARMish). Your role here is not specifically to do anything, but more to look at how problems have been solved. They may work for you, or you might prefer to do things differently, but you should recognise what the problems being solved are.

The reason for giving you something big is to avoid the normal situation where we give you fragments of systems, or toy examples. This is a reasonably complete implementation and test system for a synthesisable MU0, so as a consequence it is relatively complex.

Particular problems this code tries to solve are:

- Testing multiple implementations of the same CPU
- Constructing machine-readable test binaries from assembly test cases
- Using a single simple test-bench to run multiple complex test-cases
- Producing "golden" reference outputs to check against the simulated output
- Making testing automated and consistent

Note that there are a number of things this code doesn't do, which it probably should:

- Test individual sub-modules, rather than just the top-level CPU
- Support on-demand and partial re-builds, using a system like make

The MU0 ISA being implemented is an extension of the classic MU0 ISA, as it includes an extra instruction "OUT" (opcode=8) which prints the accumulator. This is a bit artificial, as in most case any input or output would go via memory, but makes this code-base simpler.

Conceptual organisation

The code is organised around two "axes":

- "variants" : These are different organisations of the CPU. In this case there are currently two variants:
  - `delay0` : Assumes a RAM with combinatorial read path.
  - `delay1` : Assumes a RAM with a single-cycle register read path.

–"testcases" : These are specific test inputs, in the form of programs that should be executed on each CPU. At the moment there are three of them:

```
- `countdown` : Counts down from 10 to 0, outputting each value.
- `multiply` : Multiplies together two numbers stored in memory, then outputs the product.
```

- `fibonacci` : Calculates and outputs the first 10 values from the Fibonacci sequence.

Each of the testcases should work on any CPU variant; conversely, each variant should get every test case correct. So in total we have $O(VT)$ simulations to run if we want to test everything, where T is the number of test-cases, and V is the number of variants.

MU0 is a simple ISA, but for a real ISA you might want to have at least one test-case per instruction, along with multiple integration tests that combine instructions together. Typically students use 50-200 test-cases for MIPS1. If you have two or three variants, this quickly builds up to a large number of (T, V) pairs - each pair is quick to test, but the whole thing might take minutes.

Directory structure

The project is organised into multiple directories, which contain the different types of artefact used in the project:

- `src` : Contains the Verilog sources used to describe the circuits and test-benches. Arguably this could have been split further, for example into `src/synth` and `src/test`.
- `test` : Contains files related to testing, organised into different phases. The first of these is written by humans, while the following phases are performed automatically.
  - `test/0-assembly` : This contains assembly files defining particular MU0 programs.
  - `test/1-binary` : MU0 binaries (stored as hex), assembled from the programs in `test/0-assembly`.
  - `test/2-simulator` : Compiled Verilog simulations, each one specialised to load a particular binary and run a particular CPU variant
  - `test/3-output` : The output printed by each Verilog simulation when it is executed.
  - `test/4-reference` : The expected output, as printed by a software simulator.
- `utils` : Other software utilities needed to build or test the circuits. In this case they are MU0 assembly and simulation routines.
- `bin` : binary programmes built during testing, for example the MU0 assembler and simulator
- `./*.sh` : A number of scripts used to actually make things happen.

Note that everything assumes paths relative to this root directory, so scripts will only work if they are executed as `./SCRIPT.sh`. This is also true of the simulations, which will contain hard-coded relative paths to input files.

Scripts

There are four main scripts:

- `run_all.sh` : Does "everything". Builds the utilities, then runs all known V, T pairs.
- `run_all_testcases.sh VARIANT` : Runs all test-cases for a given variant, passed as a command-line argument. For example, you could do `./run_all_testcases.sh delay0` to test the `delay0` variant.
- `run_one_testcase.sh VARIANT TESTCASE` : Runs a specific V, T pair. For example, you could do `./run_all_testcases.sh delay0 countdown` to test the `delay0` variant against the `countdown` testcase.
- `build_utils.sh` : This just builds the MU0 assembler and simulator.

Assuming you have Icarus Verilog installed (in particular the version supporting `always_comb` and `always_ff`) then you can run the whole thing as:

```
$ ./run_all.sh
```

This should produce output similar to:

```
Building MU0 utils
  done
Test CPU variant delay0 using test-case countdown
  1 - Assembling input file
 2 - Compiling test-bench
  3 - Running test-bench
    Extracting result of OUT instructions
  4 - Running reference simulator
  b - Comparing output
  delay0, countdown, pass
Test CPU variant delay0 using test-case fibonacci
  1 - Assembling input file
 2 - Compiling test-bench
  3 - Running test-bench
    Extracting result of OUT instructions
  4 - Running reference simulator
  b - Comparing output
  delay0, fibonacci, pass
  ...
```

If you want to just see the results of the test, you can redirect the information printed to `stderr` to `/dev/null`:

```
$ ./run_all.sh 2>/dev/null
  delay0, countdown, pass
  delay0, fibonacci, pass
```

```
delay0, multiply, pass
delay1, countdown, pass
delay1, fibonacci, pass
delay1, multiply, pass
```

Note : *If you get an error message saying that a .sh script is not executable, you may need to change the file permissions. This can be done using* `chmod` *to add the* `x` *(executable) permission. For example*:

```
$ chmod u+x run_all.sh
```

The scripts are all Shell scripts, so they are scripts that run within a command-line shell. If you open one of them, for example build_utils.sh you can see it is a just a text file. Some of the lines should look familiar, for example, the call to `g++` is a command you could type on the command-line yourself. At it's core a shell script is just a way of collecting lots of commands that you might type into a script, and having all those commands execute in sequence as part of a script.

There are also some other parts of note:

- The line `#!/bin/bash` at the top is called a Shebang, and tells the shell how to execute this file. In this case the shebang gives the path `/bin/bash` which identifies the `bash` shell. So if you type in:

  ```
  $ ./run_all.sh
  ```

  then behind the scenes the shell will actually call:

  ```
  $ /bin/bash run_all.sh
  ```

- `set -eou pipefail` is a trick used to make scripts more robust. If any of the programs in the script indicates a failure code (for example by using `exit(EXIT_FAILURE)`), then the whole script will immediatly stop executing.

- `MU0_SRCS="utils/mu0_assembly.cpp utils/mu0_disassembly.cpp utils/mu0_simulate.cpp"` Is storing three filenames in a variable called `MU0_SRCS`. Later on in the script we can retrieve the value of the vairable using `${MU0_SRCS}`.

- `echo "Building MU0 utils" > /dev/stderr` is using the `echo` program to print `"Building MU0 utils"`, and then redirecting it to `stderr`. This is very similar to using `std::cerr<<"Building MU0 utils<<"\n";` in a C++ program - it prints status information that is not part of the true output of the program.

Understanding the code and the infrastructure

At this point it is up to you to look at the code and try to develop understanding. You can do this by:

- Tracing code execution
- Reading the source files
- Reading the comments
- Breaking the Verilog, and seeing what happens
- Breaking the testcases, and seeing what happens
- Looking up unknown commands and constructs to see how they work
- Rewriting or simplifying parts of the code, while making sure the tests still work

Questions you may want to ask yourself are:

- Each variant has a single test-bench. How does the test-bench know which test-case to load?
- Each test-case is written in assembly, but must be loaded as a hex binary. At what point does the conversion happen?
- If you add new test-case assembly files, they will be automatically included. How does that happen?

Concrete modifications you might like to try are:

- There is a lot of shared code between delay0 and delay1, particularly in the instruction execute logic. Can this logic be refactored out into a shared module?
- The delay1 variant always takes 4 cycles per instruction. Can you create a new variant which uses the minimal number of cycles for each instruction?
- In the delay1 variant, the instruction fetch always takes one cycle before the execution cycle. Is it possible to fetch the next instruction while executing the previous instruction? If so, for which instructions is it possible? (Does this seem like a good idea, from a critical path point of view?)
- The three test-cases don't actually test all instructions. Try adding a test-case that tests any missed instructions.
- Think through edge-cases, and try to add test-cases. The MIPS presented here contains one known edge-case failure, and possibly other unknowns. Traditional problem edge cases include:
  - Sign extension
  - Greater-than versus greater-or-equal
  - Wrap-around of addresses

  The purpose of this type of infrastructure is to make it very easy to add testcases, and once you've added and resolved them they remain there forever in case you accidentally break them.

Thinking about more advanced test scenarios

MU0 is quite simple, with few instructions or possible interactions between instructions. In comparison MIPS1 has ~40 instructions which can interact in quite complex ways. So it is worth thinking about how you could/should test a MIPS.

- MIPS has no equivalent to the OUT instruction for IO. How can you get output from a MIPS test-case?
- CPUs execute sequences of instructions, but in terms of development and debugging it is useful to be able to test each instruction individually. How might you attempt to test individual instructions in MIPS CPU?
- You can compile relatively complex C functions into MIPS assembly, but how do you know what the exact (bit-accurate) expected output should be?
- How does splitting a CPU design into separate internal modules help with testing?
- What sort of testing might you apply to an ALU?
- What sorts of edge-cases would be worth testing in instructions like `andi` or `ori`?
- What sorts of edge-cases would be worth testing in `j`?
- If your MIPS CPU goes into an infinite loop for a particular sequence, how would you try to isolate the problem?