Contents

Overview

As the "verification" engineer, I worked on the testbench, bash scripts and top level RISC-V processor, that I found to be a very engaging and informative role, as I could obtain a solid understanding of each of the major components in the processor.

I also managed our repository, making edits to comments, naming styles and branch management on Git. This is valuable experience I am glad to have learnt on this coursework, as it is especially applicable to my future career.

Testbenches

In the following section, I detail the key technical work I did, writing testbenches for all the verisons of our RISC-V processor, running various programs and looking at waveforms/the behaviour displayed by the Vbuddy where possible.

I took the decision to write overall testbenches at first, having already written an admittedly basic Control Unit testbench, followed by, time allowing, module-wise testbenches.

As I progressed to develop testbenches and alter the `pdf.s` of the reference program, I decided to be more thorough with my testing, using the FSM on the pipelined and cached CPUs respectively.

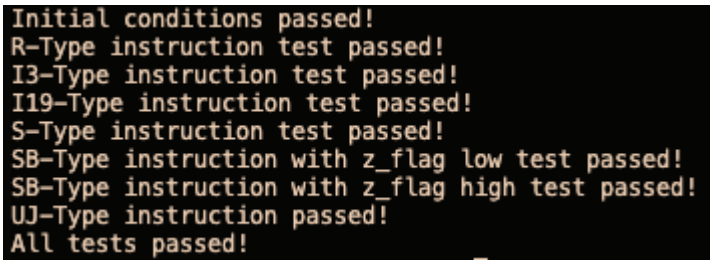
When developing testbenches, I found that the edited Python Script I was using to validate our processor through a GitHub Action did not allow me to view console outputs and debug the testbenches I was writing for the reference programs. See here.

CU

The final point of the Testbenches section applied mainly to the testbench I developed for the control unit, which used assertions, a feature found in many programming languages that we learnt about in Discrete Maths. I chose to **functionalise** the assertion testing, where each test case would input the *DUT (Device Under Test), instruction word in hexadecimal, the state of the zero flag (to test behaviour of branch

instructions) as well as a vector containing the expected results. See the relevant commit here and the relevant function below.

```
void test_case(Vcontrol_unit *dut, u_int32_t hex_code, int z_flag,
std::vector<int> results)
{

    dut->instr = hex_code;
    dut->z_flag = z_flag;

    dut->eval();

    assert(int(dut->alu_opcode) == results[0]);
    assert(int(dut->pc_src) == results[1]);
    assert(int(dut->reg_write_en) == results[2]);
    assert(int(dut->rs1) == results[3]);
    assert(int(dut->rs2) == results[4]);
    assert(int(dut->rd) == results[5]);
    assert(int(dut->wd3_src) == results[6]);
    assert(int(dut->data_write_en) == results[7]);
    assert(int(dut->pc_target_op) == results[8]);
    assert(int(dut->alu_op1_src) == results[9]);
    assert(int(dut->alu_op2_src) == results[10]);
}
```



See the successful control unit testbench message above.

I then realised that the initial test was not working as intended as 0 is not a valid 32-bit instruction for RISC-V, so it was deprecated.

F1-FSM

This incorporated most of the testbench previously used in Lab 3, with a few changes made myself, due to the nature of the input signals provided to our RISC-V processor.

```
    top->clk = 1;
    top->rst = 1;

    for (simcyc = 0; simcyc < MAX_SIM_CYC; simcyc++)
    {
        if (simcyc > 2)
        {
```

```
            top->rst = 0;
        }
        for (tick = 0; tick < 2; tick++)
        {
            tfp->dump(2 * simcyc + tick);
            top->clk = !top->clk;
            top->eval();
        }

        vbdHex(1, top->a0 & 0xF); // changed to a0 from dout
        vbdBar(top->a0 & 0xFF);
        vbdCycle(simcyc);

        if ((Verilated::gotFinish() || (vbdGetkey() == 'q')))
            exit(0);
    }
```

Reference Program

To run the reference program, I used `make reference` whilst in the `reference` directory as instructed, then modified the `riscv_tb.cpp` in the `pipelining` branch as required. I tried various different testbenches, then employed the help of groupmates to pair program a solution that functioned how I wanted it to.

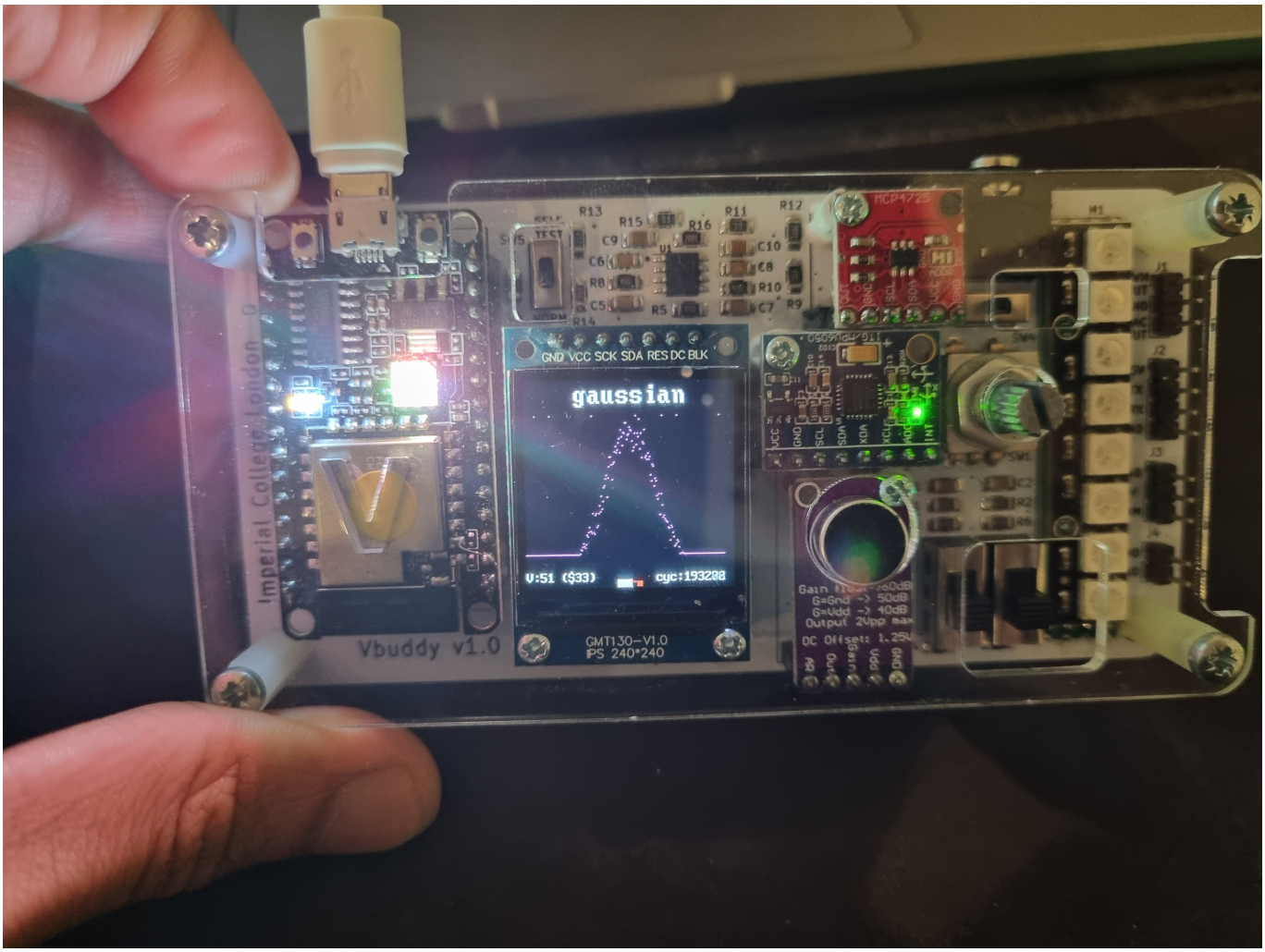
Me and Oskar found that the .hex file were the wrong endian - they were big-endian instead of little-endian, so we needed to generate new hex files from the assembly provided. Oskar used his local RISC-V GNU toolchain to compile the assembly to hex. For some reason, the `triangle.mem` and `noisy.mem` files would give a zero value for a0, and this cannot be the fault of the testbench as it worked for Gaussian and Sine without issue, so they've not been included here.

Gaussian

```
int main(int argc, char **argv, char **env)
{
    int simcyc;
    int tick;
    bool a0_is_not_zero = false;
    int counter = 0;
    std::ofstream csv_file("data.csv");

    Verilated::commandArgs(argc, argv);
    // initialises the top verilog instance
    Vriscv *top = new Vriscv;
    Verilated::traceEverOn(true);
    VerilatedVcdC *tfp = new VerilatedVcdC;
    top->trace(tfp, 99);
    tfp->open("./testbench/vcd/riscv.vcd");

    if (vbdOpen() != 1)
```

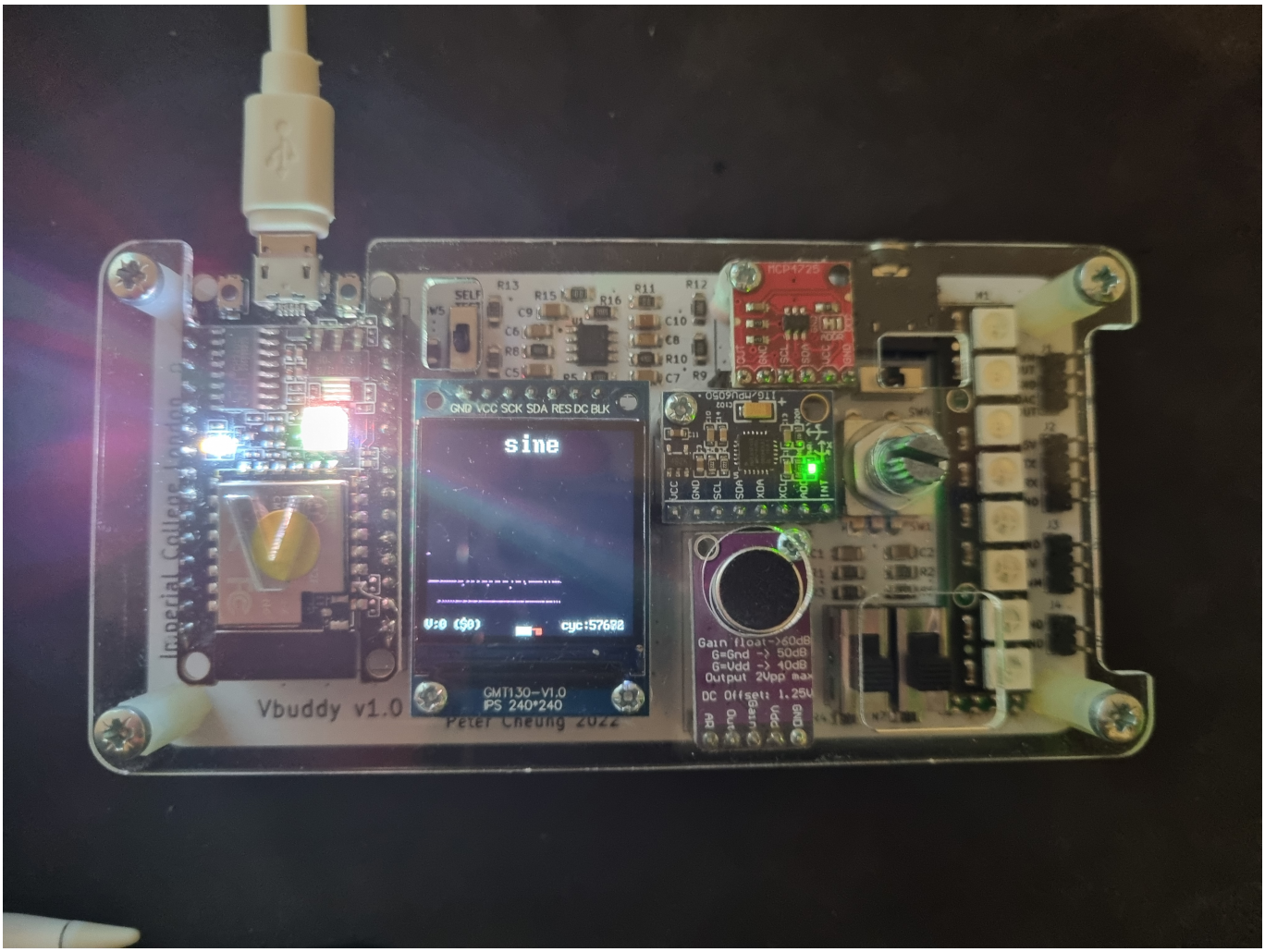
```
        return (-1);
    vbdHeader("gaussian");
    std::cout << "Starting riscv test" << std::endl;
    for (simcyc = 0; simcyc < MAX_SIM_CYC; simcyc++)
    {
        if (simcyc > 2)
        {
            top->rst = 0;
        }
        for (tick = 0; tick < 2; tick++)
        {
            tfp->dump(2 * simcyc + tick);
            top->clk = !top->clk;
            top->eval();
        }

        if (int(top->a0) != 0)
            a0_is_not_zero = true;

        if (a0_is_not_zero && counter < 2500)
        {
            if (simcyc % 8 == 0)
            {
                vbdPlot(int(top->a0), 10, 190);
                vbdCycle(simcyc);
            }
            csv_file << simcyc << "," << int(top->a0) << std::endl;
            counter++;
        }
        else
        {
            if (counter > 2500)
                exit(0);
        }

        if ((Verilated::gotFinish()) || (vbdGetkey() == 'q'))
            exit(0);
    }

    csv_file.close();
    vbdClose(); // ++++
    tfp->close();
    exit(0);
}
```

The above code was used to generate the csv files as well as the plots on the Vbuddy. See the plot for the Gaussian PDF below.

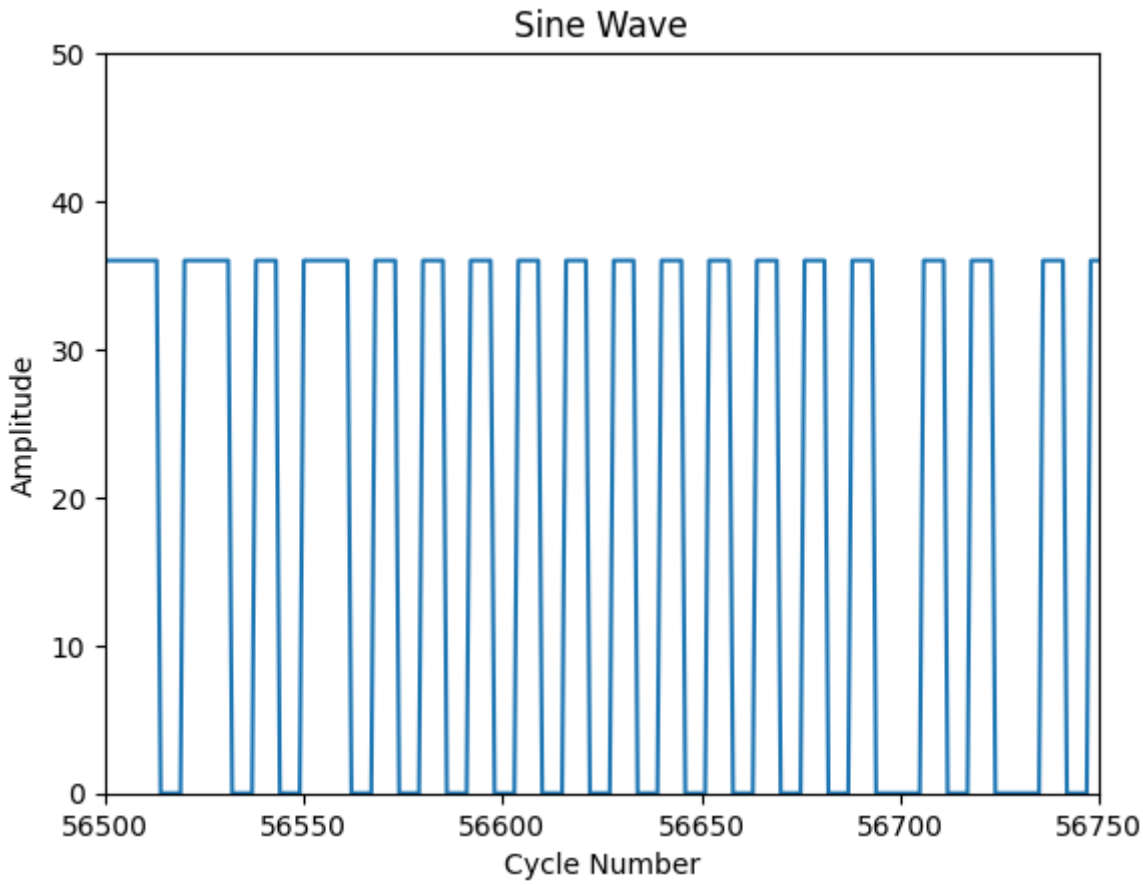


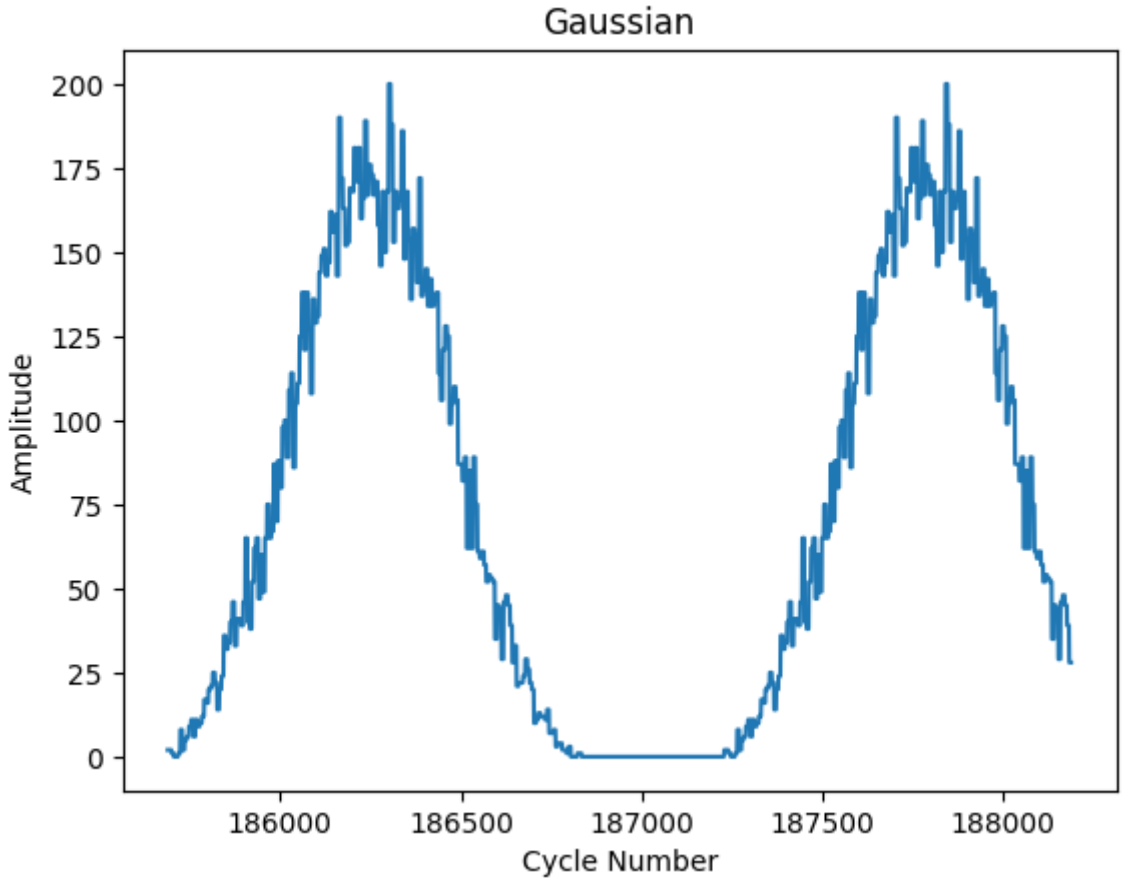
Sine

The plot for the Sine PDF below.



See below for the plots generated from the csv files, so we can observe them with axes and different scales, find the notebook for this in the `pipelining` branch.

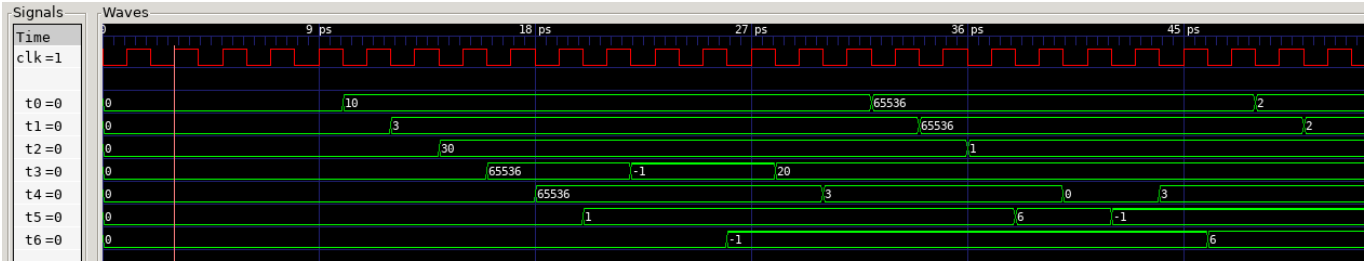




M-Extension

Following Oskar's extension to the processor, enabling for multiplication, division and remainder operations, I tested them against the following assembly program, and obtained the wavetrace.

```
.section .text
.globl main
# Program to test the M Extension to the RV32I(M)
# OB
main:
    # Test MUL
    li t0, 10           # Load 10 into t0
    li t1, 3            # Load 3 into t1
    mul t2, t0, t1      # t2 = t0 * t1 = (10 * 3) = 30

    # Test MULH
    li t3, 0x10000      # Load high value into t3
    li t4, 0x10000      # Load high value into t4
    mulh t5, t3, t4     # t5 = high bits of t3 * t4 = 0x1

    # Test MULHSU
    li t3, -1           # Load -1 (signed) into t3
    li t4, 0x10000      # Load high value into t4 (unsigned)
    mulhsu t6, t3, t4   # t6 = high bits of signed t3 * unsigned t4 = -1

    # Reset t3, t4 for next tests
    li t3, 20           # Load 20 into t3
    li t4, 3            # Load 3 into t4

    # Test MULHU
    li t0, 0x10000      # Load high value into t0 (unsigned)
    li t1, 0x10000      # Load high value into t1 (unsigned)
    mulhu t2, t0, t1    # t2 = high bits of t0 * t1 (both unsigned) = 0x1

    # Test DIV
    div t5, t3, t4      # t5 = t3 / t4 = (20 / 3) = 6

    li t4, 0

    # Test divide by zero output is as expected (-1)
    div t5, t3, t4      # t5 = t3 / t4 = (20 / 0) = 0xFFFFFFFF (-1)

    li t4, 3

    # Test DIVU
    divu t6, t3, t4     # t6 = t3 / t4 = (20 / 3) = 6

    # Test REM
    rem t0, t3, t4      # t0 = t3 % t4 = (20 % 3) = 2

    # Test REMU
    remu t1, t3, t4     # t1 = t3 % t4 = (20 % 3)= 2

    # End of test! Yay, hopefully passed?
end_loop:
    j end_loop
```



The program was written by Oskar and I repurposed the RISC-V testbench to run and validate the code.

Mistakes I Made

When developing the top level module for the formative Lab, which directly lent itself to this coursework, I assumed that everyone was done with their commits, so I failed to pull the remote repo prior to starting the work. This came up the next day when I debugged the processor using the testbench and it did not display the behaviour as expected. I took this onboard for the actual coursework.

When developing the testbench to demonstrate that it was working as intended, I realised that I needed to write device agnostic code, a technique I have learnt whilst machine learning. The issue I was coming up against was that the testbench for the overall processor was trying to use the Vbuddy in the GitHub action as well, which was causing an error, so I need to add some conditions to let the testbench know to avoid the Vbuddy relevant code when it was not connected. See commit e7ef215 for full details.

Key Design Decisions

As I was working on the testbench, a lot of the key design decisions lied with myself. As this was the case, I took the decision that we should construct several mid-level modules that grouped together relevant modules in order to allow for us to replace major components simply, rather than having to unpick all the connections to replace basic modules such as a MUX. It also allowed use to narrow our search area when correcting bugs with the overall processor.

As requested in the Project Statement, I also decided a coding style for us to use in our modules, which is listed in the `project_overview.md` file. See the commit.

I also decided to suffix files with "_deprecated" and then proceeded to move these files into a folder, in case their contents may be required later on.

I had to decide which modules made sense to be grouped together, along with their relevant designers, and this also worked in reverse, for example, choosing to break up the regfile from the alu as they were together in the formative lab. See the following commits:

Choosing to group PC with relevant mux and reg.

Separated the data_mem module and the 4-1, 2-bit select multiplexer, deprecating the `full_data_mem` module. This was done to ease the implementation of pipelining in future. See the commit here.

Corrected the muxes being labelled wrong and serving the wrong purposes. See here.

After completion of the single-cycle CPU, I wrote testbenches for the smaller components that we didn't see it as necessary for originally, for the sake of cohesive testing, and then furthered the functionality of Oskar's Python script to allow for some CI/CD like functionality, utilising GitHub Actions. This took

significant experimentation and learning, as well as waiting for Verilator to build several times. I had upwards of 30 commits on this section alone to test build on push!

I learnt about **caching and artifacting**, both methods with which I thought I could carry the Verilator install/binaries across workflow runs, but both of these ideas fell flat and resulted in what is an even more of an industry-standard approach; I then ended up using the **Docker image of Verilator** to build and run the code in a closed environment, ideal for guaranteeing thorough testing. See the actions demonstrating bug highlighting failures and successful running.

Reflection

Ensure to co-ordinate a coding style from an early stage, otherwise a lot of monotonous reformatting is required - changing from UpperCamelCase to lower_snake_case takes a lot of effort.

Construct the Continuous Integration workflow using GitHub Actions from earlier in the project, as I only made it after we finished the single-cycle design, so we couldn't use it's full capability from the beginning. Write a more cohesive Python Script and maybe consider experimenting with CocoTB.

More specifically to myself, I rushed myself along dependent on my teammates progressing quickly, which led to make a few more errors when constructing the top file like such. This could be prevented by making the schema earlier in the process, aka prior to starting coding it out. See this commit for the error I made.

Overall, I very much enjoyed this coursework and project, as even more so than last year's group project, I felt as if we were free to do with the processor as we pleased, which allowed me to explore outside the brief, such as with my development of YAML script and integrating that into a GitHub Action, providing us with a quick and easy method with which to test our processor and it's functionality, ensuring that the existing processor would continue to work even as we developed more modules for it.