RISC-V PROCESSOR

Ryan George
Individual Report – Team8

Table of Contents

Overview

In this project, I drew from past experiences in personal projects and my ARM internship, working mostly on design, some verification and general project maintenance/structure. I believe that when working in a group for a complex project, laying out the project into simpler components helps both the development and coding of the project, but also our communication and progress tracking.

Design Work

Base Control Unit

Since I was assigned the control unit from the start of lab 4, I carried on maintaining and adding features to the control unit.

```
case (opcode)
    7'b0010011: begin // I type
        alu_op     = 2'b10;
        reg_write  = 1'b1;
        imm_src    = 2'b00;
        alu_src    = 1'b1;
    end
    7'b1100011: begin // B type
        alu_op     = 2'b10;
        reg_write  = 1'b0;
        imm_src    = 2'b10;
        branch_e   = (funct3 == 3'b000);
        branch_ne = (funct3 == 3'b001);
```

Figure 1: Opcode switch statement

The control unit is split into two different parts, control logic for the CPU and ALU logic. This simple decomposition of the actual decoding of instructions means that if more instructions need to be added (to possibly implement different extensions) later they can easily be done by adding it to the switch statement

Pipelined Control Unit

After the main implementation of the control unit, the biggest change that happened after was for pipelining. In the original control unit, the decision to branch was all calculated within the control unit. This was only possible because in a single cycle CPU we already know the result of the ALU (being zero or not) so the next instruction we fetch will be the correct one.

However, in a pipelined CPU we don't know whether to take the branch or not in the decode stage so we need to move the branching logic to within the execute stage. In addition, the diagram provided has a major flaw: the branch signal which is sent along the pipeline would only work for branch equal instructions, as the logical condition whether to take the branch depends on both what type of branch it is (branch equal or branch not equal) and then the output from the ALU in the execute stage.

To allow for both instructions, I passed both signals branch equals and branch not equals so they can both still work and added the appropriate logic to determine if a jump should be taken.

```
85+ssign pc_src = (branch_e_e & alu_zero) | (branch_ne_e & ~alu_zero) | jump_e;
```

Figure 2: Branching logic based on which type of branch

```
3  module control_unit(
4      input  logic [6:0] opcode,
5      input  logic [2:0] funct3,
6      input  logic [6:0] funct7,
7-     input  logic       eq,
8
9-     output logic       pc_src,
10     output logic       result_src,
11     output logic       mem_write,
12     output logic [2:0] alu_control,
13     output logic       alu_src,
14     output logic [1:0] imm_src,
15     output logic       reg_write
16 );
```

```
3  module control_unit(
4      input  logic [6:0] opcode,
5      input  logic [2:0] funct3,
6      input  logic [6:0] funct7,
7
8+     output logic       branch_e,
9+     output logic       brance_ne,
10+    output logic       jump,
11     output logic       result_src,
12     output logic       mem_write,
13     output logic [2:0] alu_control,
14     output logic       alu_src,
15     output logic [1:0] imm_src,
16     output logic       reg_write
17 );
```

Figure 3: Git diff for the control unit's port changes for pipelining

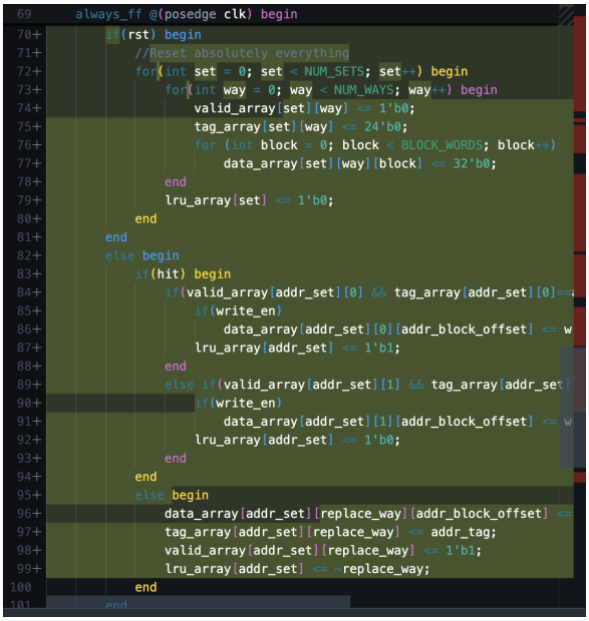
I eventually replaced this with a multibit wire called *branch*, with each bit indicating which type of branch instruction it is.

Decode Stage Register

We split the pipelining registers into four allowing each of us to tackle one stage in parallel. I simply implemented a module for the decode stage register, wired the execute stage to use the outputs from the decode register and then declared wires to connect the control signals to the memory stage register.

Cache Integration

Before I could integrate the cache, some modifications had to be made to it as I noticed a couple of subtle bugs with the cache's state machine:



```
69      always_ff @(posedge clk) begin
70+         if(rst) begin
71+             //Reset absolutely everything
72+             for(int set = 0; set < NUM_SETS; set++) begin
73+                 for(int way = 0; way < NUM_WAYS; way++) begin
74+                     valid_array[set][way] <= 1'b0;
75+                     tag_array[set][way] <= 24'b0;
76+                     for (int block = 0; block < BLOCK_WORDS; block++)
77+                         data_array[set][way][block] <= 32'b0;
78+                 end
79+                 lru_array[set] <= 1'b0;
80+             end
81+         end
82+         else begin
83+             if(hit) begin
84+                 if(valid_array[addr_set][0] && tag_array[addr_set][0]==
85+                     if(write_en)
86+                         data_array[addr_set][0][addr_block_offset] <= w
87+                     lru_array[addr_set] <= 1'b1;
88+                 end
89+                 else if(valid_array[addr_set][1] && tag_array[addr_set]
90+                     if(write_en)
91+                         data_array[addr_set][1][addr_block_offset] <= w
92+                     lru_array[addr_set] <= 1'b0;
93+                 end
94+             end
95+             else begin
96+                 data_array[addr_set][replace_way][addr_block_offset] <=
97+                 tag_array[addr_set][replace_way] <= addr_tag;
98+                 valid_array[addr_set][replace_way] <= 1'b1;
99+                 lru_array[addr_set] <= ~replace_way;
100             end
101         end
```

Figure 4: The updated cache state machine

This is the fixed cache state machine. It has three main parts and follows a very simple algorithm:

Reset:

- Loop through the entire cache and reset: the data array, the tags array and lru array.

If Hit (Determined by combinational logic):

- Determine which way we got the hit: Way0 or Way1
- Write the data if it is a sw instruction
- Switch the LRU (least recently used) to point to the other way as we have just either read or stored to it

If Miss (~Hit):

- Store the data by setting the required properties

Deep Pipelining

Deep pipelining involves splitting the 5 main stages into smaller, simpler substages. This has the effect of reducing the IPC of the CPU but has a major benefit: it simplifies the combinational logic which reduces critical path allowing for a faster clock speed. This helps offset the reduced IPC.

I researched what stages are typically deep pipelined, and most modern-day processors have about 10 stages. However, due to the time constraint involved with this assignment I chose to implement a simpler 7-stage pipeline by splitting the execute stage. The current execute stage performs two main functions, handles data forwarding and then the actual calculation of the data. We can therefore easily split it into two stages:

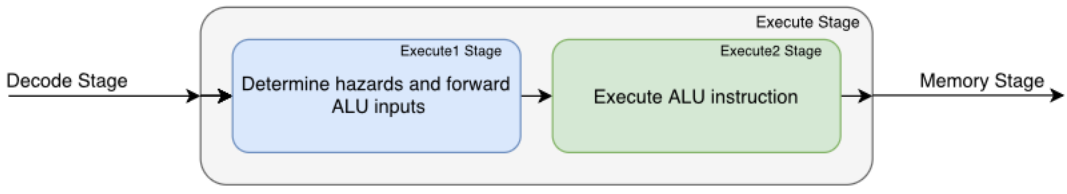


Figure 5: Decomposition of the execute stage

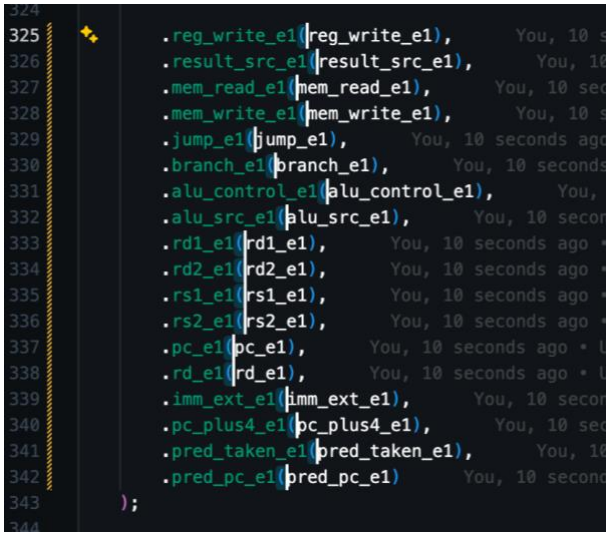
By inserting a pipeline register in between execute1 and execute 2 stages we can start the deep pipelining process. This starts by creating two new registers for each stage and then modifying variable names from *_e* to *_e1.*

```
module pipe_execute2(
    input logic clk,
    input logic rst,

    input logic        reg_write_e2,
    input logic [1:0]  result_src_e2,
    input logic        mem_write_e2,
    input logic [31:0] alu_result_e2,
    input logic [31:0] write_data_e2,
    input logic [4:0]  rd_e2,
    input logic [31:0] pc_plus4_e2,

    output logic        reg_write_m1,
    output logic [1:0]  result_src_m1,
    output logic        mem_write_m1,
    output logic [31:0] alu_result_m1,
    output logic [31:0] write_data_m1,
    output logic [4:0]  rd_m1,
    output logic [31:0] pc_plus4_m1
```

Figure 6: One of two new stage registers inserted



```
    .reg_write_e1(reg_write_e1),
    .result_src_e1(result_src_e1),
    .mem_read_e1(mem_read_e1),
    .mem_write_e1(mem_write_e1),
    .jump_e1(jump_e1),
    .branch_e1(branch_e1),
    .alu_control_e1(alu_control_e1),
    .alu_src_e1(alu_src_e1),
    .rd1_e1(rd1_e1),
    .rd2_e1(rd2_e1),
    .rs1_e1(rs1_e1),
    .rs2_e1(rs2_e1),
    .pc_e1(pc_e1),
    .rd_e1(rd_e1),
    .imm_ext_e1(imm_ext_e1),
    .pc_plus4_e1(pc_plus4_e1),
    .pred_taken_e1(pred_taken_e1),
    .pred_pc_e1(pred_pc_e1)
);
```

Figure 7: Editing variable names using VSCode multicursor feature

The remaining work now just includes updating the hazard modules and ALU inputs to match the new updated stages:

```
alu_src_mux alu_src_mux_i(
    .reg_op2(alu_input_b_e),
    .imm_ext(imm_ext_e),
    .alu_src(alu_src_e),
    .alu_op2(alu_op2_e)
);

alu alu_i(
    .alu_op1(alu_input_a_e),
    .alu_op2(alu_op2_e),
    .alu_ctrl(alu_control_e),
    .alu_out(alu_out),
    .eq(alu_zero)
);
```

```
alu_src_mux alu_src_mux_i(
    .reg_op2(rd2_e2),
    .imm_ext(imm_ext_e2),
    .alu_src(alu_src_e2),
    .alu_op2(alu_op2_e2)
);

alu alu_i(
    .alu_op1(rd1_e2),
    .alu_op2(alu_op2_e2),
    .alu_ctrl(alu_control_e2),
    .alu_out(alu_out),
    .eq(alu_zero)
);
```

Figure 8: Git diff for the updated module names

From there as an interesting experiment, I ran yosys (as part of the GowinSynthesis tools for hobbyist TangNano FPGAs) to calculate the critical path of the CPU to see whether my deep pipelining improved the CPU.

Regular Pipelined:

Resource	Usage
LUTs	9614
ALUs	1263
Registers	5470
Maximum Clock Frequency	7.104 MHz

Deep Pipelined:

Resource	Usage
LUTs	14380

ALUs	1248
Registers	8291
Maximum Clock Frequency	7.049 MHz

In our implementation and this specific synthesiser, we got a performance increase of 1MHz with a large increase in hardware utilisation. We got a very small performance upgrade and in the real world with more realistic hardware and better deep pipelining implementation it would be a larger increase.

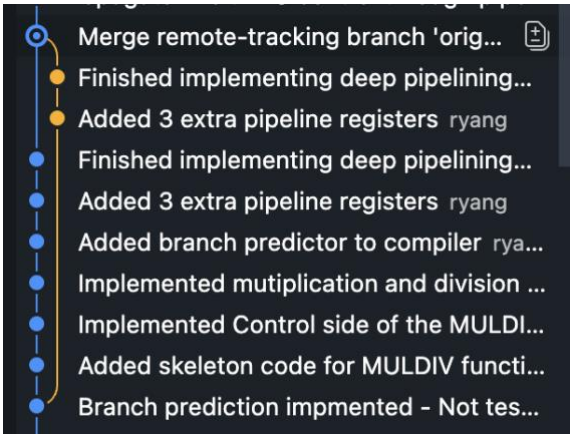


Figure 9: Screenshot of git history

This was all implemented near the end of the project and with other things left to develop. To make sure my commits and progress won't interfere I switched to a separate git branch so my teammates could work on the multiplication extension.

Verification Work

Hazard unit testbench

After we completed the hazard unit, I wrote a unit testbench to check its functionality:

```
TEST_F(HazardUnit_tb, NoHazardsNoForwarding)
{
    top->branch = 0;
    top->rs1_d = 4;
    top->rs2_d = 5;
    top->rs1_e = 1;
    top->rs2_e = 2;
    top->mem_read_e = 0;
    top->rd_e = 0;
    top->rd_m = 0;
    top->reg_write_m = 0;
    top->rd_w = 0;
    top->reg_write_w = 0;
    top->eval();

    EXPECT_EQ(top->flush, 0);
    EXPECT_EQ(top->stall, 0);
    EXPECT_EQ(top->forward_a_e, 0b00);
    EXPECT_EQ(top->forward_b_e, 0b00);
}
```

Figure 10: No data hazards should result in no data forwarding

```
TEST_F(HazardUnit_tb, DataForwarding)
{
    top->branch = 0;
    top->rs1_d = 0;
    top->rs2_d = 0;
    top->mem_read_e = 0;
    top->rd_e = 0;
    top->reg_write_m = 1;
    top->rd_m = 3;
    top->reg_write_w = 0;
    top->rd_w = 0;
    top->rs1_e = 3;
    top->rs2_e = 3;
    top->eval();

    EXPECT_EQ(top->flush, 0);       You, 2 weeks
    EXPECT_EQ(top->stall, 0);
    EXPECT_EQ(top->forward_a_e, 0b10);
    EXPECT_EQ(top->forward_b_e, 0b10);
}
```

Figure 11: Register 3 is being used in both execute and mem stage ∴ forward data

```
45  TEST_F(HazardUnit_tb, BranchFlushesPipeline)
46  {
47      top->branch = 1;
48      top->mem_read_e = 0;
49      top->rd_e = 0;
50      top->rs1_d = 0;
51      top->rs2_d = 0;
52      top->eval();
53
54      EXPECT_EQ(top->flush, 1);
55      EXPECT_EQ(top->stall, 0);
56  }
```

Figure 12: If we branch, we need to flush the pipeline to rid wrong instructions

```
TEST_F(HazardUnit_tb, WritebackForwarding)
{
    top->branch = 0;
    top->rs1_d = 0;
    top->rs2_d = 0;
    top->mem_read_e = 0;
    top->rd_e = 0;
    top->reg_write_m = 0;
    top->rd_m = 0;
    top->reg_write_w = 1;
    top->rd_w = 7;
    top->rs1_e = 7;
    top->rs2_e = 1;
    top->eval();

    EXPECT_EQ(top->stall, 0);
    EXPECT_EQ(top->forward_a_e, 0b01);
    EXPECT_EQ(top->forward_b_e, 0b00);
}
```

Figure 13: Register 7 is being used in both execute and write stage

```
TEST_F(HazardUnit_tb, LoadStalls)
{
    top->branch = 0;
    top->rs1_d = 2;
    top->rs2_d = 8;
    top->rs1_e = 0;
    top->rs2_e = 0;
    top->mem_read_e = 1;
    top->rd_e = 8;
    top->rd_m = 0;
    top->reg_write_m = 0;
    top->rd_w = 0;
    top->reg_write_w = 0;
    top->eval();

    EXPECT_EQ(top->flush, 0);
    EXPECT_EQ(top->stall, 1);
    EXPECT_EQ(top->forward_a_e, 0b00);
    EXPECT_EQ(top->forward_b_e, 0b00);
}
```

Figure 14: A load instruction (mem read) is present, so we stall

General Project Maintenance

Project Structure

As the project started, we implemented a very basic folder structure that worked for the simple Lab 4 assignment, but we recognised that as the project continued, it would very quickly become too complicated to find and run everything. To remediate this, I reorganised the folder structure recommended in the project brief.

```
(base) ryan@Ryans-MacBook-Pro Diversity8 % tree -d
.
├── rtl
└── tb
    ├── asm
    ├── c
    └── tests
        └── unit_tests
```

Figure 15: A list of the directories in our project

This was all completed on a separate git branch fix-project-layout so everyone else could work on the CPU without waiting for me to reorganise everything and update the script paths.

I then modified the demo repo provided so that unit tests (tests that assert the combinational logic of individual modules of the CPU) are separated from the top level *verify.cpp* file. This makes the tb easier to navigate and use as well, as now the top-level 'entire CPU' tests are separated from the individual module tests.

In addition to that, I configured git to ignore any build output and set line endings so that there was no need to run dos2unix.

Test Run Scripts

Since the team has members that both use MacOS and Windows (WSL) it was essential that the script could work on both platforms.

I modified the provided run script to allow for both MacOS and Windows usage, which required trial and error. I also changed the run script provided to prevent the testbench from continuing to run if the Verilator compile failed, which otherwise would just run the previous successful build giving the illusion that all was working, but in fact they weren't even compiling.

To fix this, I modified the run script to check the return code of Verilator and exit if the build failed making it explicit that this happened.

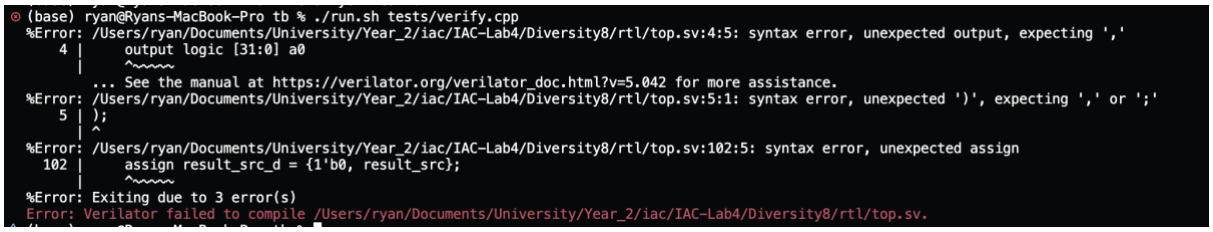


Figure 16: The script exits as soon as an error occurs

I also made a change to the system call for compiling, to ensure that the test would fail if the underlying assembly didn't compile.

Waveform Debugging Run Script

One thing we noticed quite early on in development was that running a demo program and checking the waves needed to be done a lot, especially when things weren't working. This was a bit awkward to do as the current C++ testbench was designed for assertions and verifying it was working rather than debugging.

To fix this, I wrote a very simple Verilog testbench that just loads the program.hex file, triggers a reset and then starts a clock for the CPU to run. I then created a run script that runs the simpler IVerilog compiler and opens GTKWave with the waveform loaded. The testbench also dumps the entire register file every clock cycle, making it even easier to see the changes happening to the register.

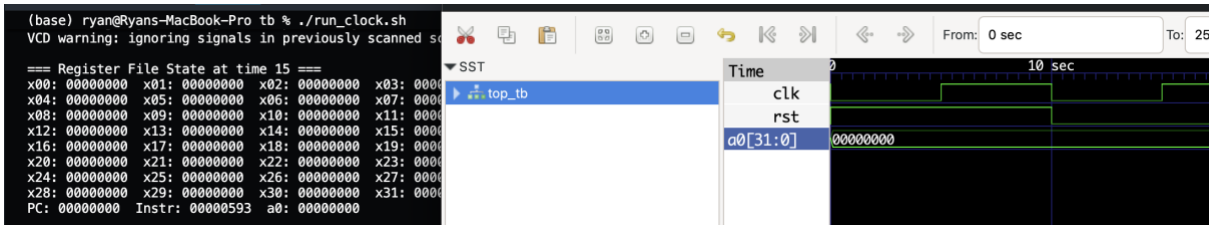
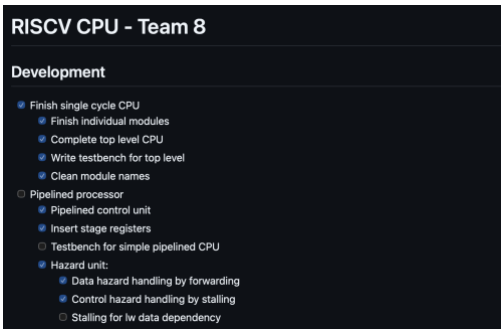


Figure 17: The result of the run_clock script

As above in the image, the script dumps the register file state and opens GTKWave, where we can confirm that a simple clock is running.

README



The README is the best place to put project related information and place documents in it. I created a simple README with a progress checklist and a 'how to' section.
The progress checklist was useful to monitor progress and remaining tasks, and the 'how to' section included example run cases to make it as clear as possible how to use the scripts.

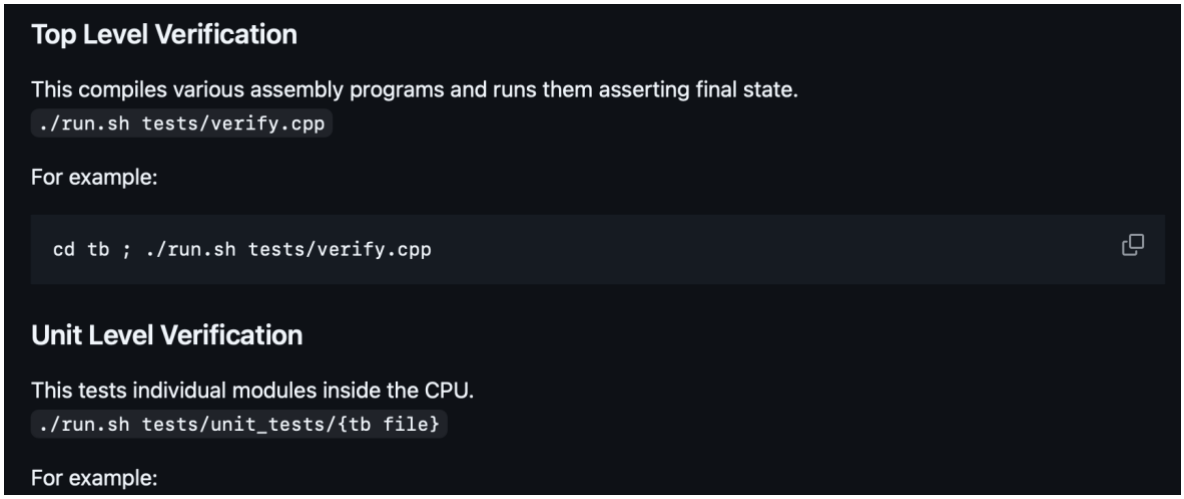


Figure 18: Screenshots from the README (as of 01/12/25)

Reflections

Looking back on the entire project there were a couple things I could have improved:

Improvement	Explanation
Test run scripts themselves	We had a couple issues with run scripts throughout the project, mainly verify.cpp outputting a PASS for a test that was clearly failing or sometimes not compiling at all.

	Putting more effort to fix this would have made the testing a lot easier.
Implemented deeper pipelining	I ran out of time to implement more deeper pipeline stages
CPU Memory realism	Memory is never single cycle access and technically there is no performance hit for accessing data memory instead of cache. Implemented delay cycles would have made much more realistic behaviour.