

RISC-V PROCESSOR

Ryan George

Individual Report – Team8

Table of Contents

Overview	2
Design Work	2
<i>Base Control Unit.....</i>	<i>2</i>
<i>Pipelined Control Unit</i>	<i>2</i>
<i>Decode Stage Register.....</i>	<i>3</i>
<i>Cache Integration</i>	<i>3</i>
<i>Deep Pipelining.....</i>	<i>4</i>
Verification Work	6
<i>Hazard unit testbench</i>	<i>6</i>
General Project Maintenance.....	8
<i>Project Structure.....</i>	<i>8</i>
<i>Test Run Scripts</i>	<i>9</i>
<i>Waveform Debugging Run Script.....</i>	<i>9</i>
<i>README</i>	<i>10</i>
Reflections	10

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+ assign pc_src = (branch_e_e & alu_zero) | (branch_ne_e & ~alu_zero) | jump_e;
86
```

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
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);
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

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] == 1'b1)
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][1] == 1'b1)
90+             if(write_en)
91+                 data_array[addr_set][1][addr_block_offset] <= w;
92+             lru_array[addr_set] <= 1'b1;
93+         end
94+     end
95+     else begin
96+         data_array[addr_set][replace_way][addr_block_offset] <= w;
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:

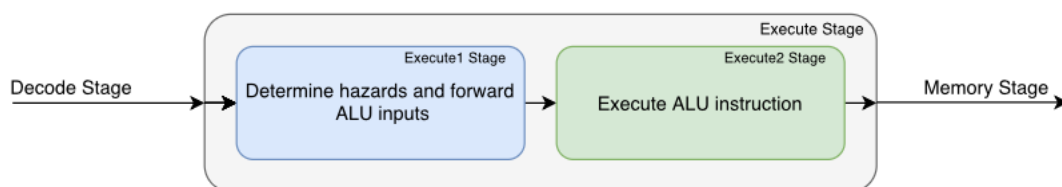


Figure 5: Decomposition of the execute stage

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`.

```

1  module pipe_execute2(
2      input logic clk,
3      input logic rst,
4
5      input logic      reg_write_e2,
6      input logic [1:0] result_src_e2,
7      input logic      mem_write_e2,
8      input logic [31:0] alu_result_e2,
9      input logic [31:0] write_data_e2,
10     input logic [4:0] rd_e2,
11     input logic [31:0] pc_plus4_e2,
12
13     output logic      reg_write_m1,
14     output logic [1:0] result_src_m1,
15     output logic      mem_write_m1,
16     output logic [31:0] alu_result_m1,
17     output logic [31:0] write_data_m1,
18     output logic [4:0] rd_m1,
19     output logic [31:0] pc_plus4_m1

```

Figure 6: One of two new stage registers inserted

```

324
325     .reg_write_e1(reg_write_e1),
326     .result_src_e1(result_src_e1),
327     .mem_read_e1(mem_read_e1),
328     .mem_write_e1(mem_write_e1),
329     .jump_e1(jump_e1),
330     .branch_e1(branch_e1),
331     .alu_control_e1(alu_control_e1),
332     .alu_src_e1(alu_src_e1),
333     .rd1_e1(rd1_e1),
334     .rd2_e1(rd2_e1),
335     .rs1_e1(rs1_e1),
336     .rs2_e1(rs2_e1),
337     .pc_e1(pc_e1),
338     .rd_e1(rd_e1),
339     .imm_ext_e1(imm_ext_e1),
340     .pc_plus4_e1(pc_plus4_e1),
341     .pred_taken_e1(pred_taken_e1),
342     .pred_pc_e1(pred_pc_e1)
343 );
344

```

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:

```

194     alu_src_mux alu_src_mux_i(
195         .reg_op2(alu_input_b_e),
196         .imm_ext(imm_ext_e),
197         .alu_src(alu_src_e),
198         .alu_op2(alu_op2_e)
199     );
200
201     alu alu_i(
202         .alu_op1(alu_input_a_e),
203         .alu_op2(alu_op2_e),
204         .alu_ctrl(alu_control_e),
205         .alu_out(alu_out),
206         .eq(alu_zero)
207     );

```

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.

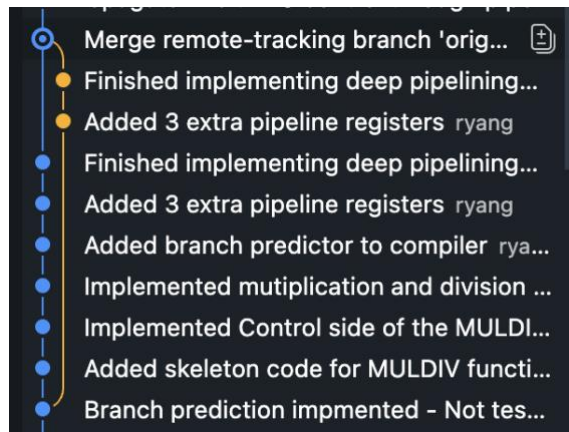


Figure 9: Screenshot of git history

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);
    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 \therefore 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.

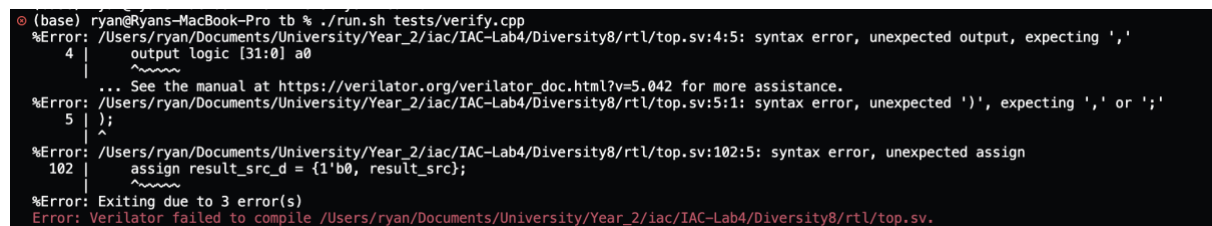
In addition to that, I configured git to ignore any build output and set line endings so that there was no need to run dos2unix.

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.

A terminal window showing the execution of a script. The prompt is (base) ryan@Ryans-MacBook-Pro tb %. The command is ./run.sh tests/verify.cpp. The output shows several Verilator errors. The first error is on line 4: output logic [31:0] a0. The second error is on line 5: ... See the manual at https://verilator.org/verilator_doc.html?v=5.042 for more assistance. The third error is on line 102: assign result_src_d = {1'b0, result_src};. The terminal ends with %Error: Exiting due to 3 error(s) and Error: Verilator failed to compile /Users/ryan/Documents/University/Year_2/iac/IAC-Lab4/Diversity8/rtl/top.sv.

```
(base) ryan@Ryans-MacBook-Pro tb % ./run.sh tests/verify.cpp
%Error: /Users/ryan/Documents/University/Year_2/iac/IAC-Lab4/Diversity8/rtl/top.sv:4:5: syntax error, unexpected output, expecting ','
4 | output logic [31:0] a0
  | ~~~~~
  | ... See the manual at https://verilator.org/verilator_doc.html?v=5.042 for more assistance.
%Error: /Users/ryan/Documents/University/Year_2/iac/IAC-Lab4/Diversity8/rtl/top.sv:5:1: syntax error, unexpected ')', expecting ',' or ';'
5 | );
  | ^
%Error: /Users/ryan/Documents/University/Year_2/iac/IAC-Lab4/Diversity8/rtl/top.sv:102:5: syntax error, unexpected assign
102 | assign result_src_d = {1'b0, result_src};
    | ~~~~~
%Error: Exiting due to 3 error(s)
Error: Verilator failed to compile /Users/ryan/Documents/University/Year_2/iac/IAC-Lab4/Diversity8/rtl/top.sv.
```

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.

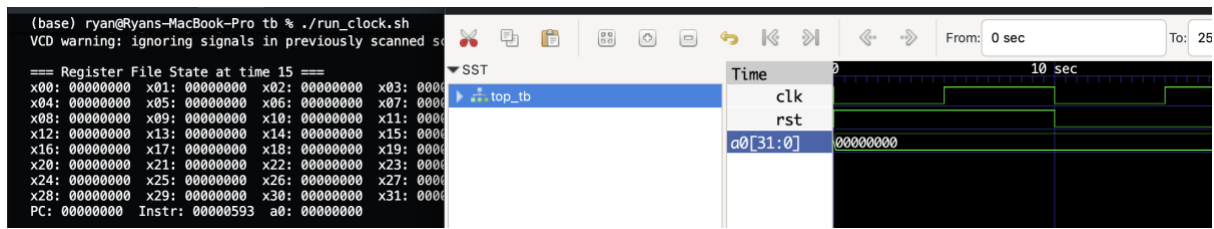


Figure 17: The result of the run_clock script

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.

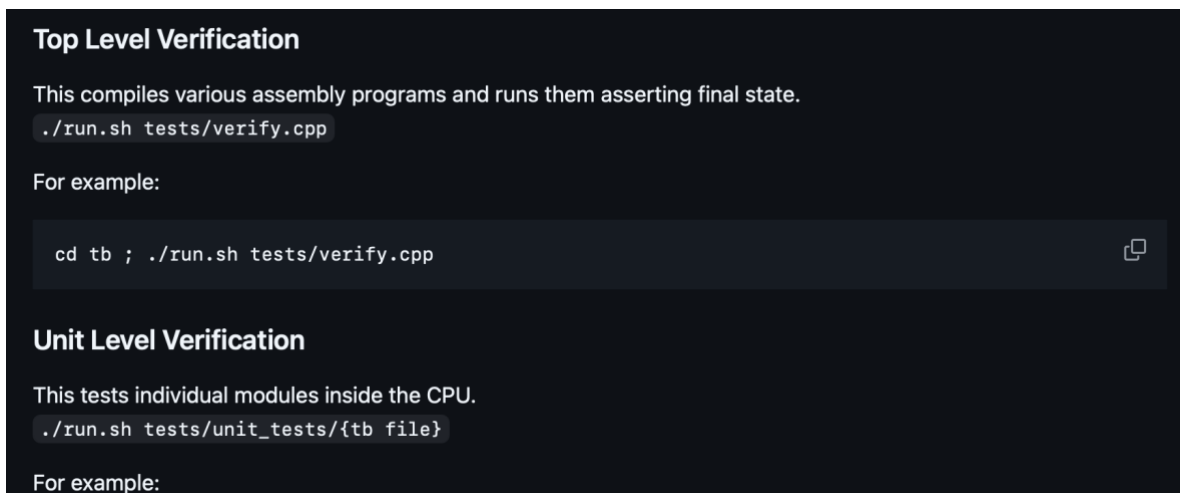


Figure 18: Screenshots from the README (as of 01/12/25)

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.