Spring 2022 ECE 552 Extra Credit Descriptions
Group #1: Justin Qiao and Xin Su

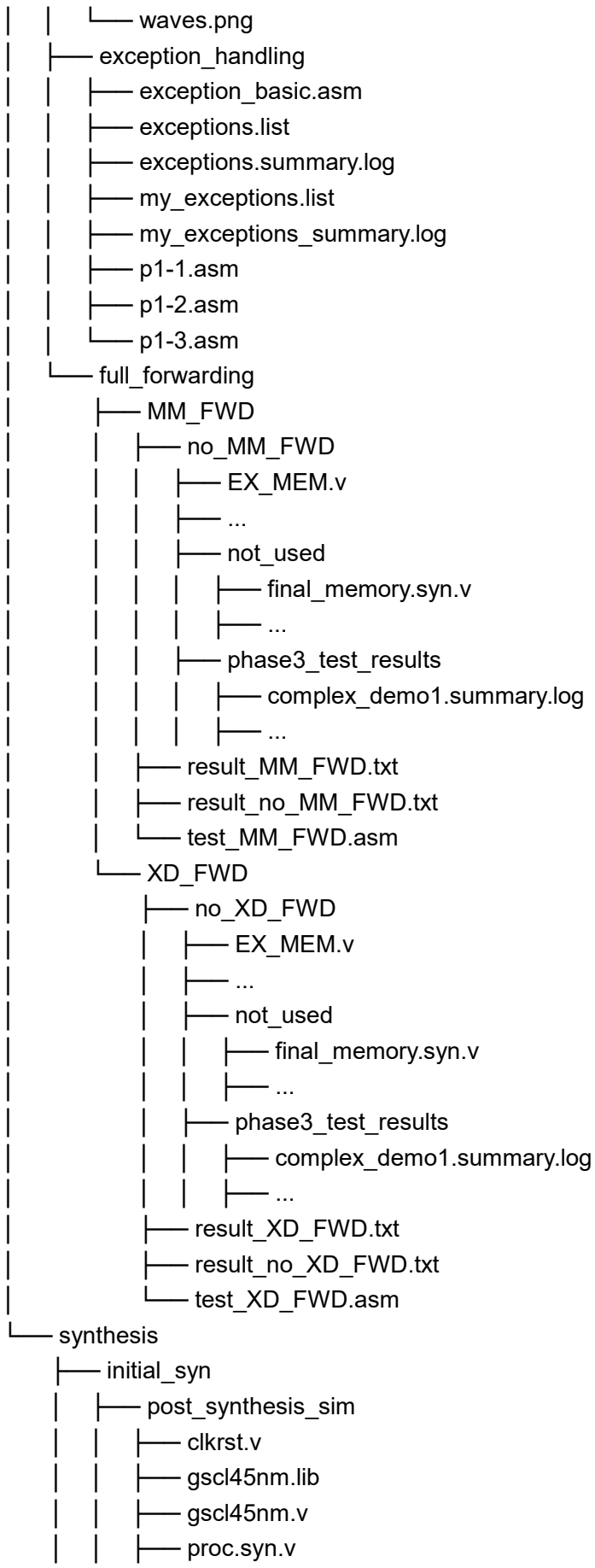
Summary

We attempted Branch in Decode, Full Forwarding, LRU policy (both caches), Exception Handling, Synthesis, and Critical Word First reads (both caches, based on LRU policy). We are aware that these worth 14 points, but we attempt them anyways to make sure we could get as many points as possible. All optimizations except Critical Word First reads are described in this document based on the source files in no_CWF_src. We got stuck on synthesis, so we only documented all steps we've taken, and went to develop CWF LRU caches. We have developed specific test cases for each optimization we have, as explained below, and we've re-run all Phase 3 tests serval times through the process to show overall improvements. We noticed that on Canvas, we have a geometric mean CPI of 5.286744 for Phase 3. However, when we run "run-final-all.sh" and calculate the CPI by ourselves, we get 5.3048 for our original Phase 3 submission (which does not have CWF LRU cache and M->M forwarding). In comparison, with all of these optimizations, we get a final CPI of 5.1499.

Submission ZIP Structure

```
Extra_Credit
├── DC_CWF_src
│   ├── EX_MEM.v
│   ├── ...
│   ├── not_used
│   │   ├── cache_FSM.v
│   │   ├── final_memory.syn.v
│   │   ├── mem_system.v
│   │   ├── memc.syn.v
│   │   ├── memory2c.syn.v
│   │   ├── memory2c.v
│   │   ├── memory2c_align.syn.v
│   │   ├── memory2c_align.v
│   │   ├── memv.syn.v
│   │   ├── stallmem.syn.v
│   │   └── stallmem.v
│   ├── phase3_test_results
│   │   ├── complex_demo1.summary.log
│   │   ├── ...
│   ├── vcheck
│   │   ├── EX_MEM.vcheck.out
│   │   ├── ...
├── extra_credit.docx
├── final_src
│   ├── EX_MEM.v
```

```
│   ├── ...
│   ├── not_used
│   │   ├── cache_FSM.v
│   │   ├── ...
│   ├── phase3_test_results
│   │   ├── complex_demo1.summary.log
│   │   ├── ...
│   ├── vcheck
│   │   ├── EX_MEM.vcheck.out
│   │   ├── ...
├── no_CWF_src
│   ├── EX_MEM.v
│   ├── ...
│   ├── not_used
│   │   ├── final_memory.syn.v
│   │   ├── ...
│   ├── phase3_test_results
│   │   ├── complex_demo1.summary.log
│   │   ├── ...
│   ├── vcheck
│   │   ├── EX_MEM.vcheck.out
│   │   ├── ...
├── our_tests
│   ├── CWF_LRU_cache
│   │   ├── CWF_1.asm
│   │   ├── CWF_1.txt
│   │   ├── CWF_2.asm
│   │   ├── CWF_2.txt
│   │   ├── CWF_3.asm
│   │   ├── CWF_3.txt
│   │   ├── no_CWF_1.txt
│   │   ├── no_CWF_2.txt
│   │   └── no_CWF_3.txt
│   ├── LRU_cache
│   │   ├── LRU.addr
│   │   ├── LRU_cachesim.txt
│   │   ├── LRU_randtest_result.txt
│   │   ├── LRU_result.txt
│   │   └── verilog
│   │       ├── cache.v
│   │       ├── ...
│   ├── branch_in_decode
│   │   ├── branch_in_decode.asm
│   │   ├── test_result.txt
```


waves.png
exception_handling
exception_basic.asm
exceptions.list
exceptions.summary.log
my_exceptions.list
my_exceptions_summary.log
p1-1.asm
p1-2.asm
p1-3.asm
full_forwarding
MM_FWD
no_MM_FWD
EX_MEM.v
...
not_used
final_memory.syn.v
...
phase3_test_results
complex_demo1.summary.log
...
result_MM_FWD.txt
result_no_MM_FWD.txt
test_MM_FWD.asm
XD_FWD
no_XD_FWD
EX_MEM.v
...
not_used
final_memory.syn.v
...
phase3_test_results
complex_demo1.summary.log
...
result_XD_FWD.txt
result_no_XD_FWD.txt
test_XD_FWD.asm
synthesis
initial_syn
post_synthesis_sim
clkrst.v
gscl45nm.lib
gscl45nm.v
proc.syn.v

```
│   │   ├── proc_hier.v
│   │   ├── proc_hier_pbench.v
│   │   └── testing.log
│   ├── synth
│   │   ├── area_report.txt
│   │   ├── cell_report.txt
│   │   ├── power_report.txt
│   │   ├── proc.ddc
│   │   ├── proc.syn.v
│   │   ├── reference_report.txt
│   │   ├── synth.log
│   │   ├── synth.tcl
│   │   └── timing_report.txt
│   └── verilog
│       ├── EX_MEM.v
│       ├── ...
│       ├── not_used
│       │   ├── clkrst.v
│       │   ├── ...
├── list.txt
└── syn_pbench
    ├── pbench_rerun
    │   ├── EX_MEM.v
    │   ├── ...
    │   ├── not_used
    │   │   ├── final_memory.syn.v
    │   │   ├── ...
    │   ├── phase3_test_results
    │   │   ├── complex_demo1.summary.log
    │   │   ├── ...
    ├── post_synthesis_sim
    │   ├── clkrst.v
    │   ├── ...
    ├── synth
    │   ├── area_report.txt
    │   ├── ...
    └── verilog
        ├── EX_MEM.v
        ├── ...
        ├── not_used
        │   ├── clkrst.v
        │   ├── ...
```

41 directories, 702 files

1. Branch Decisions in Decode (1 points)

We decided to solve branches, J, and JAL in decode since phase 1. As shown in branchJumpD.v, we called these operations together as “branchJumpD”, and we have branchJumpDTaken and branchJumpDTarget in decode.v to manage the decisions and targets of these instructions, respectively. These two signals have been passed to the fetch stage from proc.v, and some muxes are added to select the proper next PC value. We do not have data about the overall average CPI changes due to this optimization, because we never managed these 6 instructions in stages other than decode.

The test program branch_in_decode.asm demonstrates **the benefit with branch decision resolved in decode stage without considering instruction or data cache stall**. With branch predict not taken, we will continue fetching the instructions following beqz r1, .label. **If branch resolved in execute stage and the branch is actually taken, we will need to flush two instructions** following the branch instruction, resulting in total of 10 cycles running this program. **But if branch resolved in decode stage and branch is actually taken, we will only need to flush one instruction** following the branch instruction and the correct next instruction will also be fetched earlier, resulting in total of 9 cycles running this program.

Branch resolved in Execute stage(if branch actually taken):

	1	2	3	4	5	6	7	8	9	10
lbi r1, 0	F	D	X	M	W					
beqz r1, .label		F	D	X	M	W				
addi r2, r1, 1			F	D	=					
addi r2, r2, 1				F	=					
addi r2, r1, 1					F	D	X	M	W	
halt						F	D	X	M	W

Branch resolved in Decode stage(if branch actually taken):

	1	2	3	4	5	6	7	8	9
lbi r1, 0	F	D	X	M	W				
beqz r1, .label		F	D	X	M	W			
addi r2, r1, 1			F	=					
addi r2, r1, 1				F	D	X	M	W	
halt					F	D	X	M	W

“=”: Flush this instruction

The major overhead of this optimization is an additional 16-bit adder in the decode stage to calculate the branch or J/JAL target, since we can’t use the ALU in EX for this purpose. In our implementation, even if the instruction in decode is not the 6 mentioned above, we still need to keep the adder working. Which can be a waste of power.

2. Additional Forwarding Paths (1 points)

Overall, adding these extra forwarding involved adding “enable” logic to decided when we perform a forwarding and when we do not, adding muxes that selecting between the original value and the forwarded value based on the “enable” signals, and modifying the hazard detection unit to stall less when some kind of forwarding is available to prevent

a stall. **The overhead of these extra forwarding is mainly the additional muxes and logic to form their select input signals.**

(1) M->M forwarding (0.5 out of 1 point)

This optimization was done in forwarding.v using a single line of code:

assign MEMMEM_fwd = MemRead_WB & MemWrite_MEM & (Write_register_WB == read2RegSel_MEM) & (Write_register_WB != read1RegSel_MEM);

In short, this means we enable M->M forwarding when we have a LD in WB, a ST or STU in MEM, and Rd of the LD instruction is the same as the Rd of the later ST or STU. Note that Rs in the later memory instructions could not be forwarded by M->M, because the address calculations happen in EX, not MEM stage.

In proc.v, MEMMEM_fwd was used as a mux select input, selecting between the MemOut_WB and read2Data_MEM as input to the WriteData port of the memory stage.

Note that M->M forwarding happens so rarely because of cache stalls, and as mentioned in piazzia post @737, M->M forwarding messed up the way pbench is counting ICOUNT since we keep the pipeline running when possible at cache stalls, thus M->M forwarding actually increased our average CPI across the phase 3 tests.

Thus, we disabled M->M forwarding in our Phase 3 submission.

We also compared the average CPI across all Phase 3 tests with and without X->M forwarding. As shown in final_src/phase3_test_results, our overall geometric mean of CPI across Phase 3 tests with all optimizations is 5.3048. However, if we take out M->M forwarding, we could result in a slight smaller geometric mean CPI (5.3036), as shown in the txt files at:

our_tests/full_forwarding/MM_FWD/no_MM_FWD/phase3_test_results

Thus, because of the ICOUNT issue, we decide to measure our improvements here in another way. Without M->M forwarding, the total number of cycles to pass all Phase 3 tests is 680186. However, with M->M forwarding, we only need 680104 cycles.

In all, M->M forwarding is slightly beneficial to our processor.

(2) X->D and M->D forwarding (0.5 out of 1 point)

This is mainly supported by this line in hazard_detection.v, we did not put this in forwarding.v just to reuse some logic we already have:

assign XD_fwd = (OpCode_ID[4:2] == 3'b011) & RegWrite_MEM & ~MemRead_MEM & (Rs_ID == Write_register_MEM) & ~((Write_register_EX == Rs_ID) & RegWrite_EX);

In short, this line of code means we only do X->D forwarding, if there is a branch instruction in Decode, and the instruction just enter the MEM stage is not a load and writing to the same register that branch decision is based on, and the instruction in the middle at EX is not going to modify the same register.

This XD_fwd signal goes into the decode stage and serves as a mux select input selecting between XOut_MEM and read1Data of the RF as the Rs of the branchJumpD's input.

The test program test_XD_FWD.asm demonstrates the benefit with X->D

forwarding. Without X->D forwarding, the program's CPI is 6.5, and use total of 39 cycles to finish. In contrast, with X->D forwarding, the program's CPI is 6.3, and use total of 38 cycles to finish. Since without X->D forwarding, we will need to stall the instruction beqz r2, .label in decode stage for 1 cycle, waiting R2 to be ready for register file bypassing from instruction addi r2, r1, 1. But with X->D forwarding, we will be able to resolve branch in 1 cycle, and go on to fetch the next instruction.

We also compared the average CPI across all Phase 3 tests with and without X->D forwarding. As shown mentioned above, our overall geometric mean of CPI across Phase 3 tests with all optimizations is 5.3048. However, if we take out the X->D forwarding, as the code in our_tests/full_forwarding/XD_FWD/no_XD_FWD, we will get the tests results across the same Phase 3 tests in our_tests/full_forwarding/XD_FWD/no_XD_FWD/phase3_test_results at a higher geometric mean of 5.3491.

In all, X->D forwarding is beneficial to our processor.

Note that M->D forwarding is not implemented, since with RF bypassing, although the data path is different, we could see the same result as if M->D forwarding is implemented for branches in decode (see piazzia post @738).

3. Cache Replacement Policy (Both I$ and D$, 3 points in total)

We have implemented LRU caches for both the instruction and the data memory in Phase 3. However, **we finally removed that from the Phase 3 submissions** because it increases the average CPI of the Phase 3 tests, as mentioned in piazzia post @747 and @749.

The changes from pseudo random to LRU only happens in mem_system.v. We first removed the single victimway dff, and added 256 dffs called "LRU", one for each valid index of the caches. One crucial change is when both the lines corresponding to the same index in c0 and c1 are valid and the cache misses, instead of using ~victimway as the victim, we are using the lru_bit of at that index as the victim:

assign victim = (c0_valid & c1_valid) ? lru_bit : c0_valid;

It was tricky to choose the correct dff from the 256 possible LRU dffs, because we are not allowed to use << with a variable amount. Thus, we implemented this 8:256 decoder in a multistage fashion. Initially, lru_base is set to 256'h1, then we used 8 stages, from lru_mask_0 to lru_mask_7, to shift that 1 to the left according to each bit of the given index. With this decoder, we could convert the 8 bit input index (inside the 16 bit Addr) to a one hot vector indicating which line is active, and then it was easy to re-use the other logic, like muxes and buses from our original pseudo random logic.

In the following two lines of code, lru_bit is a one bit signal saying whether cache 0 or cache 1 is LRU at the corresponding index. Lru_in is a 256-bit vector used to update the 256 LRU dffs.

assign lru_bit = |(lru_out & lru_mask_7);

assign lru_in = Done ? (activeway ? ((~lru_mask_7) & lru_out) : (lru_mask_7 | lru_out)) : lru_out;

With these changes, we are able to get **6775 hits** in the original randbench for phase 2.3, as shown in:

our_tests/LRU_cache/LRU_randtest_result.txt

This result is the same as cachesim gave us, without the "pseudorandom" argument. However, although we see **6 more hits than pseudorandom cache**, after we integrate the LRU cache into our pipeline, the overall average CPI across all phase 3 tests was actually increased by 0.0039 (piazzia @749). **So, we would just say this optimization have small benefits sometimes, but can make performance worse at some other time.**

The overhead of this mainly the extra time needed to go through the levels of the 8:256 decoder. This design also occupies a lot of extra area.

4. Exception Handling (2 points)

Exception handling was added in Phase 3 of our project.

We treated SIIC and RTI as special J instruction. We added an EPC register in proc.v, and we added muxes to the branchJumpDTarget input to the fetch stage:

.branchJumpDTarget(siic ? 16'h0002 : (rti ? EPC_out : branchJumpDTarget_ID)),

The siic and rti signals are one bit control signals newly added to our control unit, the JumpD control signal is also raised when our control unit sees an opcode for SIIC or RTI, so that the stall, flush, and other related logic already implemented for solving branches in decode can be shared for exception handling.

After making these changes, we passed all tests in /u/s/i/sinclair/public/html/courses/cs552/spring2022/handouts/testprograms/public/exceptions.list. We also wrote exception_basic.asm, in which we only load 0xBADD to r7, as required in the extra credit description document. Moreover, since Justin wrote siic tests for HW4, his tests are also copied here, and formed my_exceptions.list. The test results are in the two txt files in the same folder as the tests are.

The benefit is straightforward, we get support of the two additional instructions. The overhead here is not significant, because all we need is to add one more option to the mux selecting the next PC, and an additional register holding EPC.

5. Synthesizing the Design (2 point)

In the beginning, we substituted memc.v, memv.v, and final_memory.v by their .syn.v versions, and then we performed our initial synthesis according to instructions on: https://pages.cs.wisc.edu/~sinclair/courses/cs552/spring2022/synthesis.html

Note that we used the command in section 4.4 all the time:

synth.pl --list=list.txt --type=proc --cmd=synth --top=proc --f=fetch --d=decode --e=execute --m=memory --wb=write_back --opt

We then took the netlist (proc.syn.v) and tried to run the test with wsrun-synth.pl. However, there were a lot of warnings and errors because of the signals we were monitoring in the pbench was optimized away by Design Compiler, as shown in:

synthesis/initial_syn/post_synthesis_sim/testing.log

Thus, we modified our design to make sure all signals we were monitoring are accessible as output ports of proc.v. Then we modified proc_hier.v and the pbench accordingly, as shown in the folder: synthesis/syn_pbench/verilog.

Before synthesizing the design again, **we re-run all Phase 3 tests with the modified pbench and proc to make sure they still pass the pre-synthesis tests.** The results are at:

synthesis/syn_pbench/pbench_rerun/phase3_test_results

We then synthesized the design again, and we are able to find the expected signal names in proc.syn.v, but somehow we can't access any multi-bit port, so **we modified the pbench again to concatenate the one bit outputs from proc.syn.v as the signals we are finally monitoring**. This way, **at least we were able to run the tests smoothly, see:**

synthesis/syn_pbench/post_synthesis_sim/testing.log

Then it comes the next problem, that we are not passing any tests (but it looks we reset the processor properly), which could because of our max delay problem. As we discussed in office hours and on piazzia @797, our design could not meet timing no matter the frequency was set to 1GHz or 100MHz. **We could see the synthesis log is changed to 100MHz with your modification, but somehow we got exactly the same critical path, with the same amount of max delay violation.**

Putting that aside, our critical path is from PC through the Instruction Cache FSM, going into the write back logic at the RF in Decode, and then coming back to PC from the branch detection unit. We had a max delay slake at -0.75 on this path. Moreover, even if we do not consider this, a similar path starting from EX/MEM going through the Data Cache FSM, coming back to decode and the branch detection unit, and eventually the PC, would also result in a large negative slack. This can be a consequence of the 8 level 8:256 decoder for the LRU caches. Nevertheless, even if we do not consider either paths with the cache system, we still could not make timing, because the path from ID/EX through ALU and ending at EX/MEM would also cause a negative slack at around -0.5.

In all, our takeaway from synthesis is: a) our logic is fully synthesizable; b) we were able to move one step further from what was described in piazzia @789 that we could run any Phase 3 tests with our modified pbench and proc; c) our current design has a Total Cell Area of 265543.075879.

We did NOT move forward with synthesis from this point as instructed.

*******Optimizations mentioned above are based on the source code in the folder "no_CWF_src", the Critical Word First (CWF) LRU Cache designs mentioned in the next section have all above optimizations, but we had no time to go through synthesis again with our final design.*******

6. "Critical Word First" Data Reads from Memory to (LRU) Cache (Both I$ and D$, 5 points in total)

(1) Critical Word First Reads LRU Data Cache (3 out of the 5 points)

We started with duplicating our working LRU cache (mem_system.v and cache_FSM.v), and rename their copies as critical_mem_system.v and critical_cache_FSM.v. In this stage, we used the old LRU cache for the Instruction memory, and used the CWF LRU cache for the Data memory.

The **critical changes from LRU to CWF LRU** for the Data Cache are as follows:

A) For the alloc states in the cache controller, we no longer load the cache from the memory in the order of 000, 010, 100, 110 as the offset. Instead, the offset of the critical word, which is known by Addr[2:0] is loaded first, and the other three values are loaded in a wrapped around order. For instance, if we have a cache read miss and the critical word we are reading has an offset of 100,

then we load the corresponding cache line in this order by offset: 100, 110, 000, 010.

B) Next important change is to assert Done early for CWF read misses. In cache_FSM, this occurs in the ALLOC2 state, and read misses no longer assert Done in the cycle before returning to IDLE (from state RDDONE). To inform the upper level when the read miss really end, we have a signal called “ending” that would only go high in place of the original Done in state RDDONE. Moreover, an additional one bit dff called CRI was used to keep track of if we are in the states during a CWF read. Which means we could potentially go on with the next instructions in the pipeline while we are still loading the cache due to a previous read miss.

C) The above changes mainly happens in critical_cache_FSM.v, but we also need to modify critical_mem_system.v to compensate these changes. To continue the pipeline under a read miss, the inputs to the Data Cache system must be latched, until the cache line of the earlier read miss is fully updated. A bunch of dffs were added in critical_mem_systems.v to support this.

D) Similarly, we also need to modify memory.v to latch the data we read out earlier than before because of our CWF optimization. In case we need to stall the pipeline due to the next instruction is accessing the same cache line or some other reasons that we have to stall, if we do not have these latches, we will lose the CWF data. The additional array of dffs (critical_data) was used to handle this in memory.v

E) Finally, We updated the control signal, DC_Stall, in memory.v and proc.v to make our pipeline continue under Data Cache read misses when possible.

The benefit of this optimization is we could save some cycles when we have a Data Cache read miss. To illustrate the benefits, we re-run all Phase 3 tests and got a geometric mean of CPI at 5.2615 (with original LRU cache this number was 5.3048), as shown in the summary.log files at:

DC_CWF_src/phase3_test_results

There are not too many **overheads** in terms of timing due to this optimization. Only a few muxes are added on to the combinational paths, and the majority of the changes are additional flops in parallel. But yes, those additional flops would increase the total area of the design significantly, and there are two copies of them for the two memories.

(2) Critical Word First Reads LRU Data and Instruction Cache (the other 2 points)

Our final design in folder final_src used CWF LRU Caches for both the Data Cache and the Instruction Cache. With the working design in the previous sub-section, we modified fetch.v in a similar fashion, removed our original LRU cache files (mem_system.v and cache_FSM.v), and **we got a further reduced geometric mean CPI of 5.1499, smaller than what we get with only CWF LRU D$ (5.2615)**.

More specifically, **we have three tests in this folder to show the benefits**:

our_tests/CWF_LRU_cache

The txt files with a “no_” prefix are tests results from no_CWF_src, where both

caches are our original LRU caches. The other 3 txt files with the same name as the tests are corresponding test results when both the instruction and the data cache has CWF features. From CWF_1.asm, we could see CWF can save us 4 cycles because the later addi instructions could proceed when the load miss is still being handled. From CWF_2.asm, we can see that CWF LRU at least do no worse than our original LRU design when it have to stall for consecutive memory accesses to the same cache line. From CWF_3.asm, we arbitrarily picked some store and load instructions, and it shows our new design could manage stores as usual, and bring us a reduction of 11 cycles to execute this program.

Some key difference between the I$ and D$ includes that we are only reading the I$, so we could create a different copy of cache system with all Write supports removed so that the **overheads** should be reduced. Unfortunately, we did not have enough time to do so.