Design and Implementation of a 3-stage RISC-V CPU

Jack Hughes, Elias Lehman

May 9, 2024

Abstract

In this project we implement a 3-stage pipeline and one-way set associative cache to process the base RISC-V instruction set architecture (ISA). We validate the performance of our core logic using a provided assembly test suite, achieving a minimum clock period of 18ns (55.5 MHz). Although our cache design passes tests on most RISC-V instructions, it fails to correctly store and subsequently load data. We perform place and route (PAR) free of design rule violations using Cadence Innovus despite the cache setback. The most up-to-date commit (which should be used for grading) in on branch **jack-cache**.

1 Design

We chose to design a 3-stage pipeline as depicted in Figure 1. In this section, we will go through the major components of our design from left (fetch/decode) to right (memory/writeback) then discuss the cache in the proceeding section. For most wires and regs, D, E, or M prefixes indicate which stage of the pipeline the wire is tied to; Decode, Execute, or Memory, respectively.

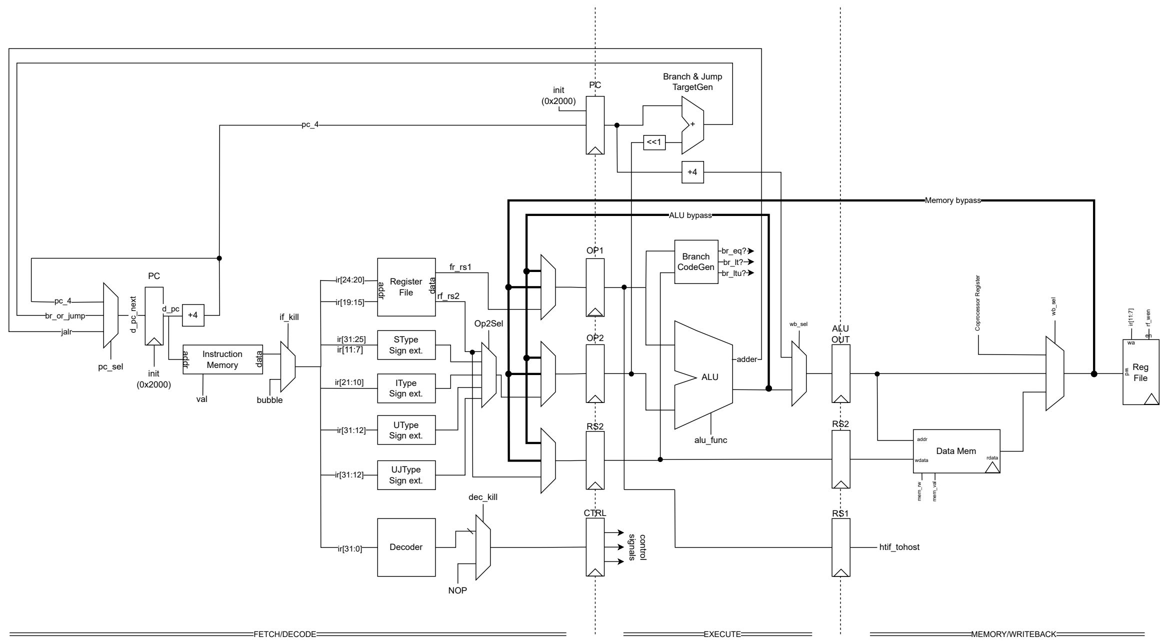


Figure 1: Block diagram of the implemented core.

1.1 Pipeline Design

The first major set of design questions relates to how we fetch instructions and how we increment the PC. When we increment the PC, we have three options: 1) PC + 4 to fetch the next instruction from the icache, 2) a signal call br_or_jmp which is the branch or jump target that is generated in the execute stage by adding the instructions PC to either an immediate offset or a register offset, or 3) no change to the PC. We continuously assign the D_PC_Next value to one of these three options depending on the instruction and other control flow parameters (i.e. are we stalling or bubbling?). The D_PC0 register pushes D_PC_Next to D_PC on each clock cycle. Another thing to note is that for conditional branches, we predict branch not taken. For jumps, we don't care what the next PC will be because we will insert a bubble in the pipeline on the next instruction.

Now we encounter the instruction cache which will be discussed later. Some key things to note are that on a read hit, the cache will return the data after 2 cycles. On a read miss, there is a 4-cycle penalty to read from the main memory into the cache, so 6 cycles will be required before the pipeline can resume. Lastly, when the icache or dcache is stalled (it cannot accept new requests because it is busy), the pipeline will stall. This is accomplished by tying the mem_stall signal to the clock enable of each pipeline register.

Next, we come across a MUX that represents our logic for needing to kill the current instruction in the decode/fetch stage. This is needed for when we mispredict a conditional branch. If the instruction in the execute stage is a branch, and if the branch is taken, since we predicted not taken we insert a NOP in the fetch/decode stage, thereby killing the false instruction.

Now we're at the register file and immediate parsing blocks. For the register file, we make sure the read enable signal is pulled high, then we parse the instruction that was fetched from the icache to provide the two read addresses. This data is returned asynchronously and gets passed to a series of MUX's which decide which signals are passed on to the execute stage. The immediate parsing is handled in a file called "imm_parse.v", and the output from those computations are fed into the Op1Sel MUX along with register file data specified from the address d_rs2 from the instruction. The logic for the Op1Sel MUX is handled in a file called "op_1_sel_mux.v" which uses the current instruction to decide which signals should be moved through the pipeline. Below this, you will also notice a box labeled "decoder" and another kill MUX. This is here to signify our bubble insertion/NOP logic. There are cases when we'd need to keep the current instruction in the decode/fetch stage for additional cycles (such as when there's a WAR hazard). In the case of a load instruction immediately preceding an instruction that uses the load's destination register as an operand, we insert a NOP into the pipeline here such that there's a NOP between the two instructions. This is necessary due to the fact that we don't have a memory to execute bypass path (which could have been a good optimization to add).

Next, we see a series of three MUXs called XX_bp_sel. The selection logic for these MUXs is contained in the file named "bp_sel_mux.v". This file handles our bypass logic which is how we are able to avoid almost all data hazards. These optimizations limit the need to insert bubbles in the pipeline, thus decreasing our CPI. At a high level, the logic works as follows: it first checks the instruction opcode in the decode stage to determine if it could have a data dependency. If so, it then checks the operand values (rs1 and rs2) against all of the destination register (rd) values from the instructions in the execute and decode stage. If there is a match, it sets the correct signal for each of the three MUXs which is determined where the dependency was found, and which instructions they are between.

At this point, most of the heavy lifting is done. The execute stage contains a few combinational logic paths which feed data forward to the memory/writeback stage, or backwards again to the fetch/decode stage. On the next clock cycle, the instruction that was previously in the fetch/decode stage, is now at the output of the E_INST_REG_1 register (assuming a bubble wasn't inserted and the pipeline isn't stalled). The ALU performs an operation on the values in registers E_OP1 and E_OP2. The operation it performs is determined by the instruction that is executing in the execute stage, and the logic can be found in the "ALU.v" and "ALUdec.v" files. The BranchCodeGen block compares the data in the E_OP1 and E_RS2 registers and pulls the wire br_out high if certain conditions are met. This signal is used in the next PC logic to determine whether or not a conditional branch in the execute stage resolved to taken or not taken. If the instruction in execute isn't a branch, this signal is ignored. The branch and target generation block above the ALU takes the value from E_OP2 and adds it to the instructions PC value for branches and jal instruction, or to the value in E_OP1 for jump and link instructions. The signal br_or_jump is then used to set the next PC value if the instruction is a taken branch or a jump. The output of the ALU, branch target generator, and of the E_PC (+4) register are fed to a MUX. Depending on the instruction, this MUX selects the appropriate signal to send to the memory stage.

Next, we look at the memory/writeback stage of the pipeline. If the instruction requires a dcache write operation, then the M_RS2 register will hold that data. The output of the M_ALU_OUT register will be a memory address or writeback data. Let's take a look at how the pipeline sets the appropriate dcache signals. If the instruction is a load, we set read enable high, write enable low, and capture the output. Since the cache only returns data on word boundaries, and since we need to handle read byte, half-word, and full word load operations, we need to parse the output of the dcache before sending it to the register file or to a bypass path. This is handled in a case statement. For store instructions, we use a similar case statement. Depending on the lower two bits of dcache_addr and on which type of store it is, we set the write enable mask. Lastly, depending on the type of instruction, we select either the M_ALU_OUT or m_dmem_out to send to the bypass path and as the data to the register file. If the instruction is a branch, a store, or if the register destination is x0, the register file write enable signal will be low; otherwise, a writeback will occur so it will be pulled high. That concludes a high-level description of our pipeline, so now lets take a look at the cache.

1.2 Cache Design

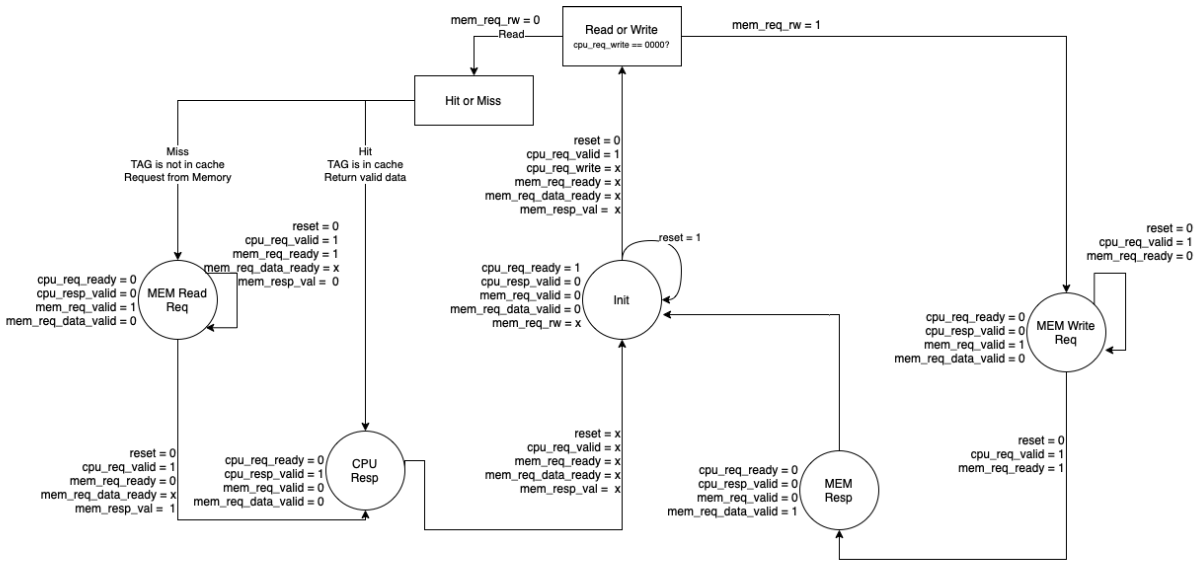


Figure 2: Cache finite state machine (FSM) diagram.

Our cache is set up as an FSM with a controller. We implemented a direct mapped cache with write-through and no write allocate policies. For the SRAM units, we used four of the sram22_256x32m4w8 macros. The cache line size is 64 bytes and words are 4 bytes. This results in a cache that has 64 lines. This configuration calls for the following TIO designations: the offset is the lowest 4 bits of the cpu_req_addr, the index is the middle [9:4] bits of the address, and the tag is designated the remaining bits ([29:10]). To handle the cache's valid bits, we have a 64-bit register that gets initialized to 64'b0 upon reset. With our scheme, we only need to modify these bits when we pull up a new cache line from memory on a cache read miss by setting the bit at the index of the cache line to 1. We also use a 64x32 SRAM macro to store the tags that correspond to each cache line. Our cache can read and write in two cycles, and there is a 4 cycle read miss penalty. At one point this was optimized to deliver one cycle reads about 75

For the sake of readability and clarity, our cache has more states than are needed (some of them could be combined into one). You will also notice some redundant variables, logic, and comments. This is leftover from our debugging efforts and should be ignored (I'm hoping to come back and fix our final bug later). The initial state that is set on reset is the INIT state. This state takes on most of the logic needed to make the cache work.

Signals that are set continuously to handle control: cpu_request_ready is high when the FSM is in the INIT state. The tag, index, and offset wires are parsed from cpu_req_addr as described above. The sram_line is 4 x index which is used to indicate the actual line in the SRAMs where the given cache line starts. sram_block indicates which of the 4 SRAM macros holds the specific word (or bytes) being looked for on a read. The SRAM write masks always stay at 1111 (since we handle the masking of data in the pipeline). For all four SRAMs, the data-in and address values are continuously assigned to some offset of sram_line and the mem_resp_data based on the current state. A mem_data_in_local reg is assigned based on where the word that needs to be written should fall in a 128-bit line (the other bits are set to zero, and the mem write mask is used in order to maintain previous data). This is used when writing to main memory. We assign a "hit" signal which checks whether the tag for the cache line currently in the cache matches the tag of the address that we're trying to read or write from. It also checks that the cache line is valid. Additionally, we continuously assign our mem_req_valid signal and pull it high whenever we need to read from main memory (when we have a read miss and cpu_req_valid).

The INIT state: We first rest all relevant signals (i.e. write enable and valid signals). Then we check if cpu_req_valid - if it's not valid (the CPU is stalled due to the other cache), we set next_state = 'INIT and end. If it is valid, we check for four scenarios: 1) read hit 2) read miss 3) write hit 4) write miss. This is handled by if statements, and within each case, the relevant signals are set. On a read hit, you set cpu_req_valid low and set the next state to 'READ_CACHE. In the READ_CACHE state, we only need to set cpu_resp_valid high and return to the init state. Like some of the other states, READ_CACHE acts as a delay state to give the SRAM time to read. As described above, the specific SRAM that is being read from and the line in that SRAM are set continuously. On a read miss, there's a bit more to do. Firstly, you must check that mem_req_ready. Then, if the memory is ready to receive a read, set the SRAM write enable signals high and set the next state to 'MEM_READ_1. In MEM_READ_1, we check that mem_resp_valid, keep the SRAM write enable signals held high, write a valid bit to the correct index, set the metadata SRAM write enable signal high, then go to MEM_READ_2. These three mem_read states are about the same. In the second one, however, you need to pull the metadata write enable signal low again. This is because you already updated the tag and valid bit after the first mem_read state. These states act as delays while the data is read from the memory into the SRAMs. On the final mem_read state, you set the next state to MEM_READ where the write enable signals are pulled low, and the cpu_resp_valid signal is set high. On a write hit, we write through to the SRAM and main memory. To do this, in INIT, we set the correct SRAM block write enable signal high, set mem_req_rw high, set the mem data mask and data bits as described earlier, then set the next state to CACHE_MEM_WRITE. The CACHE_MEM_WRITE state resets the appropriate signals, pulls cpu_rest_valid high, and sets the next state to INIT. The write miss case works in much the same way, except without writing to the SRAMs.

2 Simulation

Our pipeline and cache implementation together pass all but three of the instruction assembly tests. Namely, it fails SW, SH, and SB. Because of this, we are unable to test our full design implementation on benchmark tests. The cycles required for each assembly test is included in Table 1 below. Without the cache implementation, however, we pass all of the assembly and benchmark tests. The cycles required to complete each assembly and benchmark test is included in Tables 2 and 3 below.

3 Implementation

3.1 Synthesis

3.1.1 Timing

Our synthesized design passes setup and hold timing constraints with a maximum frequency of **55.5 MHz** and critical path slack of **1.368 ns**. The critical path is that of our data cache last SRAM setup. The path travels from the pipeline control module's memory register, **M_INST_REG_1**, to the write-enable input of the data cache's fourth SRAM module, **sram4_we_reg**.

addi	483	*lb*	575	*slti*	473
add	987	*lbu*	575	*sltiu*	473
andi	383	*lh*	603	*slt*	971
and	1023	*lhu*	617	*sltu*	971
auipc	73	*lui*	85	*srai*	515
beq	625	*lw*	616	*sra*	1093
bge	687	*ori*	401	*srli*	503
bgeu	737	*or*	1029	*srl*	1077
blt	625	*sb*		*sub*	967
bltu	679	*sh*		*sw*	
bne	629	*simple*	27	*xori*	405
jal	65	*slli*	481	*xor*	1027
jalr	217	*sll*	1047		

Table 1: Reported cycles for assembly tests with cache.

addi	223	*lb*	253	*slti*	218
add	455	*lbu*	253	*sltiu*	218
andi	179	*lh*	265	*slt*	449
and	475	*lhu*	272	*sltu*	449
auipc	36	*lui*	40	*srai*	237
beq	292	*lw*	275	*sra*	502
bge	319	*ori*	186	*srli*	231
bgeu	344	*or*	478	*srl*	496
blt	292	*sb*	486	*sub*	447
bltu	317	*sh*	540	*sw*	545
bne	294	*simple*	15	*xori*	188
jal	32	*slli*	222	*xor*	477
jalr	102	*sll*	483		

Table 2: Reported cycles for assembly tests without cache.

cachetest	3407778
final	7358
fib	6286
sum	22096966
replace	22096993

Table 3: Reported cycles for benchmark tests without cache.

In our diagram, this is a relatively short path. However, there is significant combinational logic between when the M stage gets the next instruction and when the final SRAM write-enable is set to in the INIT cache stage. First, the instruction gets parsed. Then it goes through a large case statement to determine what the write-enable and read-enable signals and input data of the dcache should be. Then, there is required logic to initiate the CPU's ready-valid exchange, which is used to determine whether the operation is a read or write in the cache before setting the SRAM write-enable signal. Further, the fourth SRAM input signals are the last to be set.

3.2 Place and Route

3.2.1 Floorplan

Floorplanning requires an iterative process to balance design check violations while optimizing the clock tree. The final floorplan, seen in Figure 3, was designed to give the CPU clear paths to both the clock input (located on the bottom of the die), and the cache SRAM macros. The CPU was constrained to be placed in a rectangular area, such that the smaller metadata marcos could be aligned on the top, while the larger instruction and data cache SRAMs could be aligned on the left and right sides, respectively.

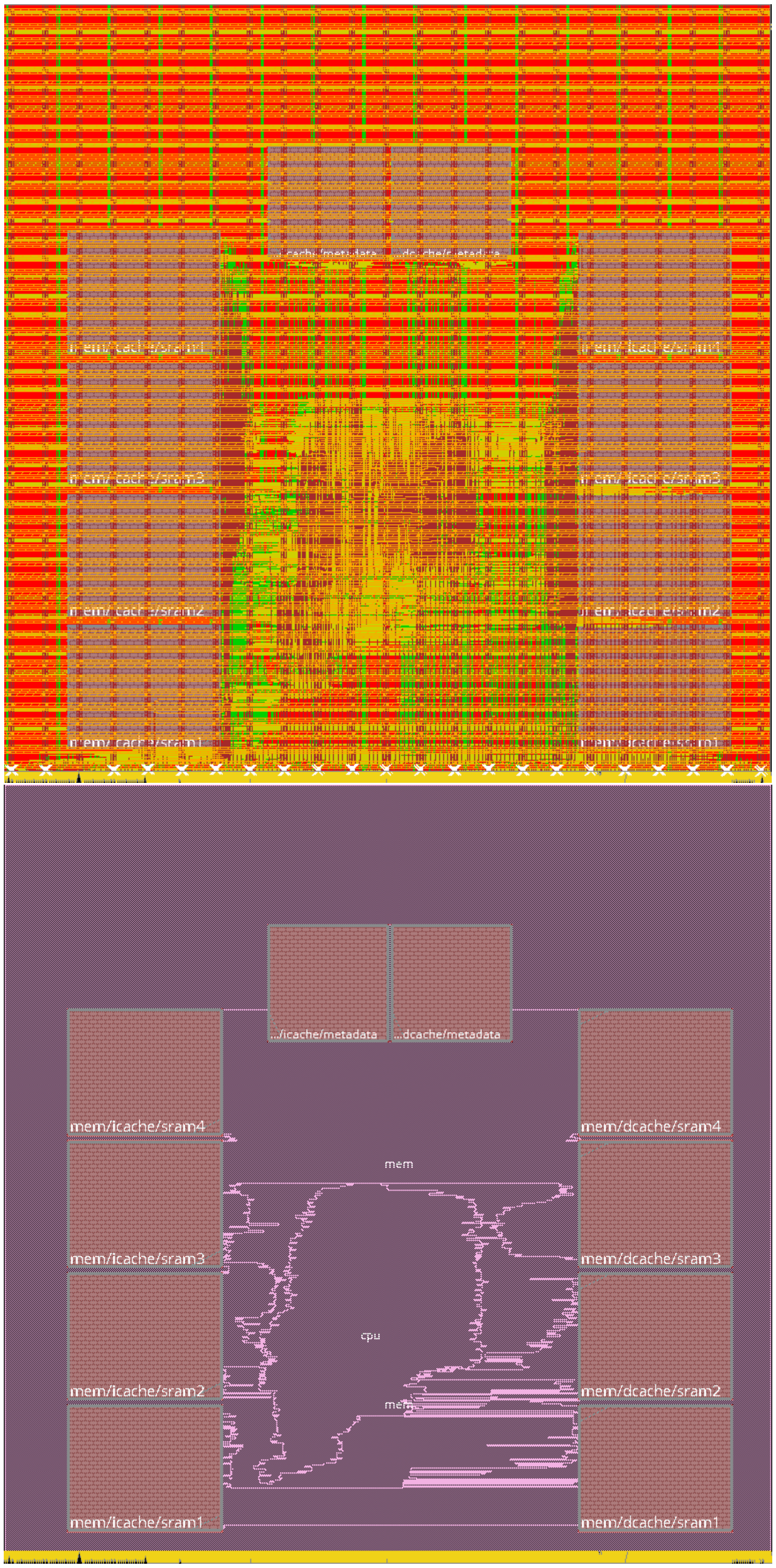


Figure 3: Screenshots of die floorplan from Innovus.

3.2.2 Timing

The final design gave the clock tree visible in Figure 4, with a maximum frequency of **55.5 MHz**. Binned slack values and their respective occurrences are visualized in 5. The critical path slack is **0.003 ns**, as specified by the Innovus timing report found in Figure 6.

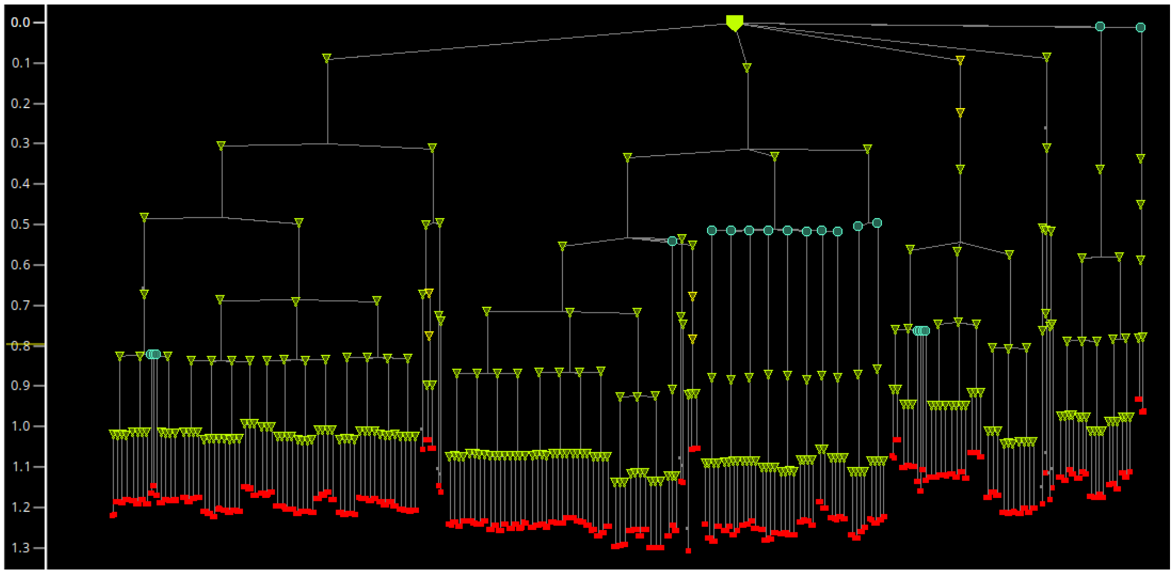


Figure 4: Clock tree diagram.

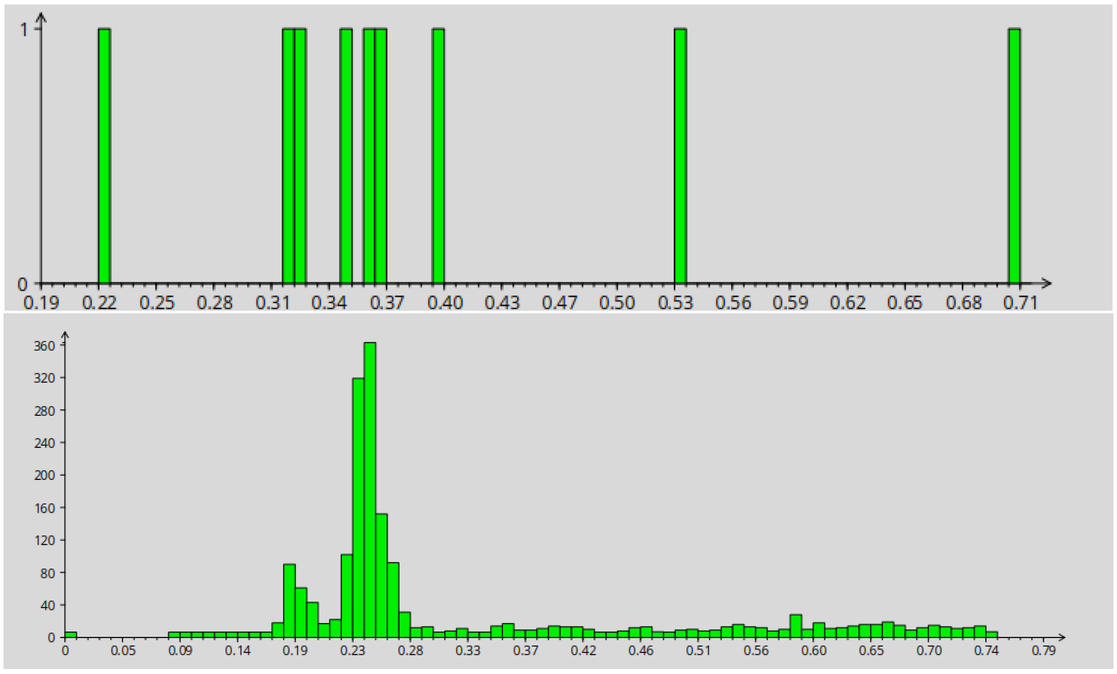
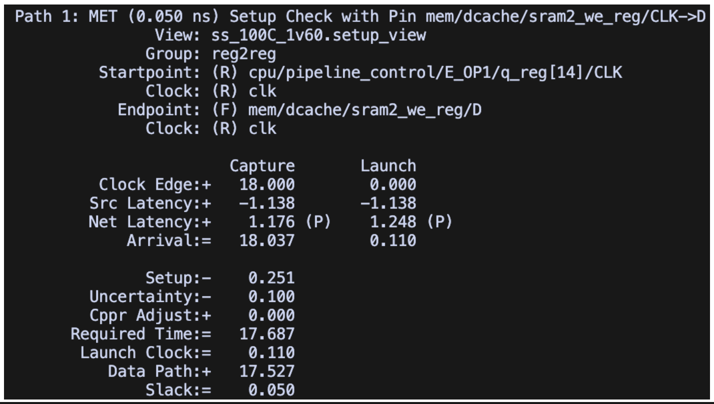


Figure 5: Clock tree debugging for setup times (top) and hold time (bottom). The y-axis is number of occurrences and the x-axis is delay in nanoseconds.

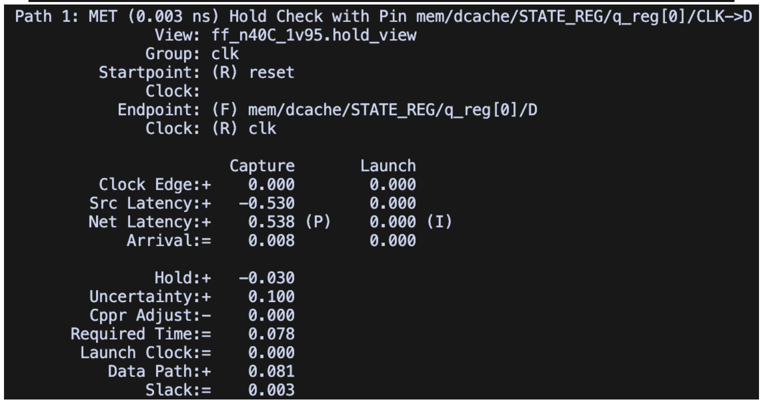
The hold clock tree debugging window helps us visualize how many paths are limiting the frequnecy. There are a few critical paths with slack on the order of a few picoseconds that if resolved would allow slightly better performance. Otherwise, there seems to be a bimodal distribution of paths around 19 ps and 24 ps which are likely contributed by the bulk of the memory and CPU logic.



```
Path 1: MET (0.050 ns) Setup Check with Pin mem/dcache/sram2_we_reg/CLK->D
               View: ss_100C_1v60.setup_view
              Group: reg2reg
         Startpoint: (R) cpu/pipeline_control/E_OP1/q_reg[14]/CLK
              Clock: (R) clk
           Endpoint: (F) mem/dcache/sram2_we_reg/D
              Clock: (R) clk

                       Capture       Launch
         Clock Edge:+   18.000        0.000
        Src Latency:+   -1.138       -1.138
        Net Latency:+    1.176 (P)    1.248 (P)
            Arrival:=   18.037        0.110

              Setup:-    0.251
        Uncertainty:-    0.100
        Cppr Adjust:+    0.000
      Required Time:=   17.687
       Launch Clock:=    0.110
          Data Path:+   17.527
              Slack:=    0.050
```



```
Path 1: MET (0.003 ns) Hold Check with Pin mem/dcache/STATE_REG/q_reg[0]/CLK->D
               View: ff_n40C_1v95.hold_view
              Group: clk
         Startpoint: (R) reset
              Clock:
           Endpoint: (F) mem/dcache/STATE_REG/q_reg[0]/D
              Clock: (R) clk

                       Capture       Launch
         Clock Edge:+    0.000        0.000
        Src Latency:+   -0.530        0.000
        Net Latency:+    0.538 (P)    0.000 (I)
            Arrival:=    0.008        0.000

               Hold:+   -0.030
        Uncertainty:+    0.100
        Cppr Adjust:-    0.000
      Required Time:=    0.078
       Launch Clock:=    0.000
          Data Path:+    0.081
              Slack:=    0.003
```

Figure 6: Innovus timing report critical path output for setup (top) and hold (bottom).

The post-place-and-route critical path travels through the ALU, from the output of execute instruction register in the pipeline control module, **E_INST_REG_1**, to the ALU output register, **M_ALUOut**. This makes sense as the ALU contains much of the combinations logic required to operate the CPU including all arithmetic instructions, while the registers also contribute delay. It should be noted that this path is unlike the post-synthesis critical path. We think the paths differ because the synthesis procedure is able to abstract wire delay, which may contribute larger portions of the delay post-PAR. On the other hand, PAR is required to constrain the wires to fit the area provided, so many additional turns and therefore more length may be needed, contributing the additional delay.

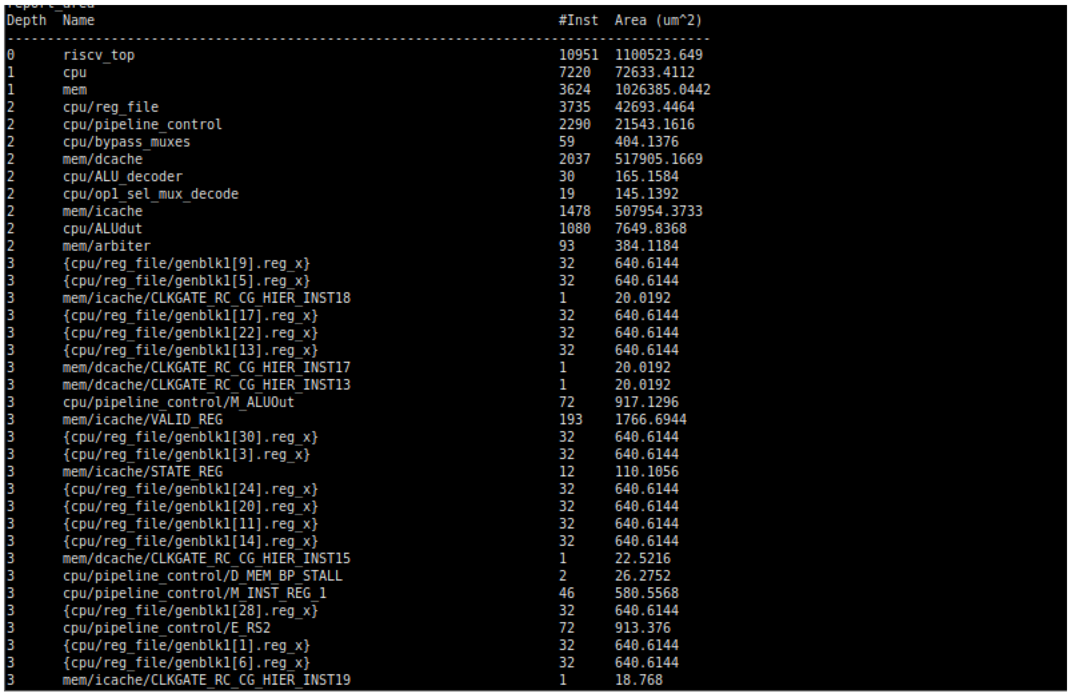
3.2.3 Area

The component using the most area our design the register file belonging to the CPU(Figure 7). This file needs to be large to store all incoming instructions in the case of a live user, as well as support the test suites.

4 Conclusion

4.1 Known Bugs

We still have a bug related to the cache. The pipeline passes all benchmark and assembly tests, but the cache is failing the store assembly tests while passing the rest. We have spent countless hours trying to fix this issue. To summarize. We are having trouble correctly timing everything. In a sense, the cache itself is working perfectly fine (it is correctly reading and writing to and from memory and the SRAM blocks). The issue arises due to some strange interaction between the icache, our pipeline, and the dcache. Essentially, when a store instruction is followed by a load instruction, elements of the several ready valid interfaces and stall signals interact and cause timing issues. One of two things happens depending on some parameters that we modify: 1) there's an infinite stall that occurs due to the timing of the states in the dcache and icache when the dcache finishes executing a write or



```
Depth  Name                                                            #Inst  Area (um^2)
-----------------------------------------------------------------------------------------
0      riscv_top                                                       10951  1100523.649
1      cpu                                                             7220   72633.4112
1      mem                                                             3624   1026385.0442
2      cpu/reg_file                                                    3735   42693.4464
2      cpu/pipeline_control                                            2290   21543.1616
2      cpu/bypass_muxes                                                59     404.1376
2      mem/dcache                                                      2037   517905.1669
2      cpu/ALU_decoder                                                 30     165.1584
2      cpu/op1_sel_mux_decode                                          19     145.1392
2      mem/icache                                                      1478   507954.3733
2      cpu/ALUdut                                                      1080   7649.8368
2      mem/arbiter                                                     93     384.1184
3      {cpu/reg_file/genblk1[9].reg_x}                                 32     640.6144
3      {cpu/reg_file/genblk1[5].reg_x}                                 32     640.6144
3      mem/icache/CLKGATE_RC_CG_HIER_INST18                            1      20.0192
3      {cpu/reg_file/genblk1[17].reg_x}                                32     640.6144
3      {cpu/reg_file/genblk1[22].reg_x}                                32     640.6144
3      {cpu/reg_file/genblk1[13].reg_x}                                32     640.6144
3      mem/dcache/CLKGATE_RC_CG_HIER_INST17                            1      20.0192
3      mem/dcache/CLKGATE_RC_CG_HIER_INST13                            1      20.0192
3      cpu/pipeline_control/M_ALUOut                                   72     917.1296
3      mem/icache/VALID_REG                                            193    1766.6944
3      {cpu/reg_file/genblk1[30].reg_x}                                32     640.6144
3      {cpu/reg_file/genblk1[3].reg_x}                                 32     640.6144
3      mem/icache/STATE_REG                                            12     110.1056
3      {cpu/reg_file/genblk1[24].reg_x}                                32     640.6144
3      {cpu/reg_file/genblk1[20].reg_x}                                32     640.6144
3      {cpu/reg_file/genblk1[11].reg_x}                                32     640.6144
3      {cpu/reg_file/genblk1[14].reg_x}                                32     640.6144
3      mem/dcache/CLKGATE_RC_CG_HIER_INST15                            1      22.5216
3      cpu/pipeline_control/D_MEM_BP_STALL                             2      26.2752
3      cpu/pipeline_control/M_INST_REG_1                               46     580.5568
3      {cpu/reg_file/genblk1[28].reg_x}                                32     640.6144
3      cpu/pipeline_control/E_RS2                                      72     913.376
3      {cpu/reg_file/genblk1[1].reg_x}                                 32     640.6144
3      {cpu/reg_file/genblk1[6].reg_x}                                 32     640.6144
3      mem/icache/CLKGATE_RC_CG_HIER_INST19                            1      18.768
```

Figure 7: Area by module output by the Innovus report_area command.

2) an unavoidable stall causes the icache to pass on a garbage value which propagates through the pipeline. It's hard to explain without looking at the waveforms and code, and we know this sounds pretty solvable, but we have tried many fixes to no avail. We have modified the states (added more and set different signals in different ones), we have modified the logic for the dcache and icache read and write enable signals in the pipeline (which set CPU req valid from Memory151), we have changed the timing characteristics of the cache by moving signals from non-continuous to continuous and visa versa, we have spent many hours stepping through every signal in the waveforms (for both caches, the pipeline, and the external memory), we have stepped through working tests, gone to office hours, etc. It is frustrating and deeply unsatisfying that we'll be submitting the cache with this little bug, but we have spent so much time on this single that we think my next step would have to be refactoring the code to figure out what might be going wrong.

Update: Kevin H. recently pointed out some bugs. Some of them we have tinkered with in the past, but there are a few things that need changing. We will request an extension on the lab's eda machines and hopefully sort this out.

4.2 Optimizations

We have two bypass paths that run from the execute and memory stages back to the fetch/decode stage. This prevents the need to bubble our pipeline to wait for data hazards to resolve and lowers our CPI. We mindfully designed our three stages such that the critical path is not too long. One example of this is our placement of the branch target generator in the execute stage instead of on the longest path of the fetch/decode stage. This was a trade off for a slightly higher CPI (due to the need to bubble on jump instructions) in favor of a shorter critical path and higher clock frequency.

We implemented a pretty neat cache optimization that we later had to roll back to simplify the design for debugging. Essentially, since we set the SRAM addresses continuously, we realized that as long as we knew that the last requested address (for a read hit) was one less than the current address and on the same line, we didn't need to jump to the CACHE READ state since the output was already ready. This decreased the memory delay for a series of 16 consecutive memory addresses (the first of which was a miss) from 36 to 24.

4.3 Contributors

We want to give thanks to the supportive course staff of EECS 151 at UC Berkeley who wrote the test scripts, and to the UC Berkeley EECS Department for allowing access to the tools needed to make this project possible.