Final Project Report for Lab 6

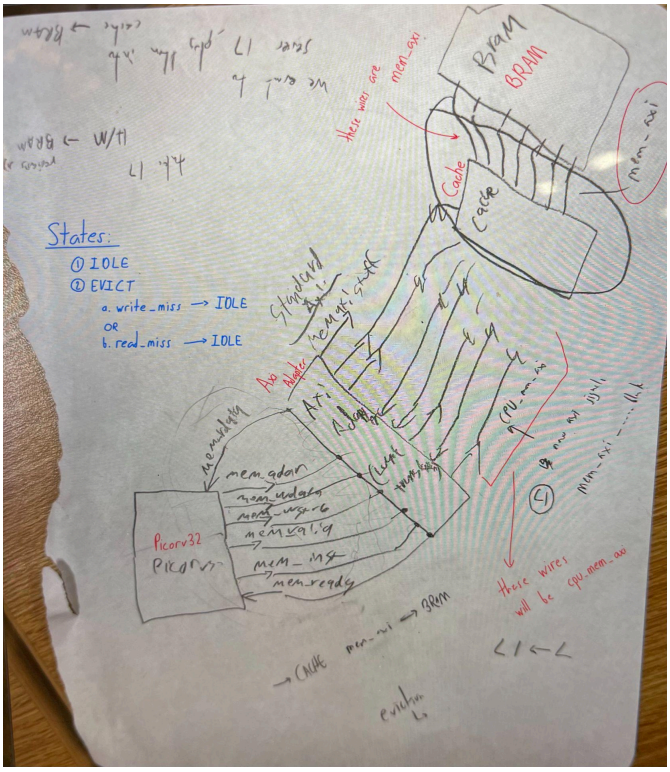
Source Code Link: https://github.com/imezabu/cs391r1-lab6.git
Slideshow Link:
https://docs.google.com/presentation/d/1hqesDeeYhcaregruN7IUlxvVxqowZqoApPfV1yJOrpk/edit?usp=sharing

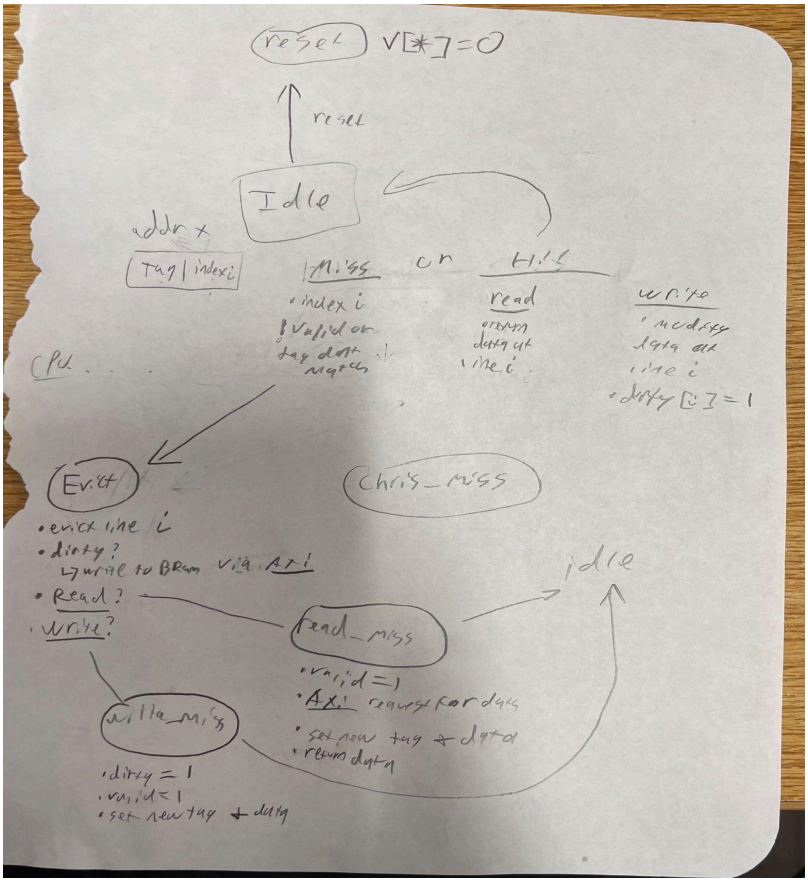
Part 2

Introduction

Goal: The main objective of our final project is to instantiate a single-entry, direct-mapped cache module utilizing write-back that sits between the picorv32 and the BRAM. This would require us to create our own cache module and then rework all the wiring and AXI control signals that are currently connecting the BRAM and CPU.

Approach





For our approach to modifying the picorv32 module, our group planned to implement a cache block that would connect the resulting output signal of the axi_adapter from the Pico to the BRAM. As shown above, we will build a cache module that takes cpu_mem_axi signals and outputs them as mem_axi signals into the BRAM module. For the cache logic, there will be 7 states that govern our Finite State Machine implementation. Our group designed the IDLE state to act as a resting location for the cache, waiting for the control signal data to check and see if we are doing a read or a write operation. If IDLE sees a read operation, it will then check for a hit or a miss. Under a hit, the state will be converted to READ_MEM, reading the data from the cache to the CPU and then returning to IDLE. On the other hand, under a miss, we check if there is a dirty bit or not. If the address of the cache has a dirty bit then we convert our state to EVICT, otherwise our state is set to REFILL. Under EVICT, we set the write valid signals for the BRAM to be high and check if the BRAM is ready and valid. If so, then we enter the EVICT_ACK state where, if we are in the read condition, we would go back to the REFILL state and otherwise we would go to the OVERWRITE state. Within the REFILL state, if the BRAM signals for arvalid and arready are on then we set our state to be REFILL_ACK and check to see if the BRAM control signals for rready and rvalid are on. If so, we would set the cache_data, set the dirty bit of the cache to be 0 and set what the cache_tag is and so on, finishing with setting the state to READ_MEM. Lastly, under the OVERWRITE state, we would set the cache_valid to be 1 and also the dirty bit as well, while setting the bvalid of the signal to be 1 as well. If the bvalid and bready of the cpu is high then our state is set back to IDLE and

we are no longer in a READ or WRITE operation. For the WRITE operation under IDLE, it is mostly the same template with just some of the signals replaced properly. Then afterwards we tried to instantiate the wiring between the Pico to cache and then to BRAM by reversing the inputs and outputs. We were able to use our same block design from the Getting Started Lab, but unfortunately it didn't work in the basic Lab3/4 test bench, so we weren't able to test it on the FPGA.

Code:

```
case(state)
    IDLE: begin
        cpu_arready <= 1;
        cpu_awready <= 1;
        cpu_wready  <= 1;

        //READS
            if(cpu_arready && cpu_arvalid) begin //handshake
                req_type<=READ;
                latched_addr<=cpu_araddr;
                tag<=cpu_araddr[31: INDEX_BITS]; //check range pleaseee
                index<=cpu_araddr[INDEX_BITS-1:0];

                cpu_arready<=0; //stop all transactions
                cpu_wready<=0;
                cpu_awready<=0;
                //HIT
                if(cache_valid[cpu_awaddr[INDEX_BITS-1:0]] && cache_tag[cpu_awaddr[INDEX_BITS-1:0]]==cpu_araddr[31: INDEX_BITS]) begin
                    state<=READ_MEM; //maybe adjust
                    /*cache efficiency
                    cpu_rvalid<=1;
                    cpu_rdata<=cache_data[index];*/
                end
                //MISS
                else begin
                    if(cache_valid[cpu_awaddr[INDEX_BITS-1:0]] && cache_dirty[cpu_awaddr[INDEX_BITS-1:0]]) begin //dirty
                        state<=EVICT;
                    end else begin //non-dirty
                        state<=REFILL;
                    end
                end

            end
```

```
//WRITES
else if((cpu_awvalid && cpu_awready)&&(cpu_wready&&cpu_wvalid)) begin //AW and W handshake
    req_type<=WRITE;


        latched_addr<=cpu_awaddr;
            tag<=cpu_awaddr[31: INDEX_BITS]; //check range pleaseee
            index<=cpu_awaddr[INDEX_BITS-1:0];
        latched_data<=cpu_wdata; //latch data

        cpu_arready<=0; //no new txns
        cpu_wready<=0;
        cpu_awready<=0;
        //HIT
        if(cache_valid[cpu_awaddr[INDEX_BITS-1:0]] && cache_tag[cpu_awaddr[INDEX_BITS-1:0]]==cpu_araddr[31: INDEX_BITS]) begin
            state<=OVERWRITE;
            /*efficiency? do this immediately on the clock?
            cpu_bvalid<=1;
            cache_data[index]<=latched_data;
            cache_tag[index]<=tag;
            cache_valid[index]<=1;
            cache_dirty[index]<=1;*/
        end
        //MISS
        else begin
            if(cache_valid[cpu_awaddr[INDEX_BITS-1:0]] && cache_dirty[cpu_awaddr[INDEX_BITS-1:0]]) begin //dirty
                state<=EVICT;
            end else begin //non-dirty
                state<=OVERWRITE; //maybe refill?
            end
        end

end
```

```
READ_MEM: begin
    cpu_rvalid<=1;
    cpu_rdata<=cache_data[index];
    if(cpu_rvalid && cpu_rready) begin
        cpu_rvalid<=0;
        req_type<=NONE;
        state<=IDLE;
    end
end
OVERWRITE: begin
    cpu_bvalid<=1;
    cache_data[index]<=latched_data;
    cache_tag[index]<=tag;
    cache_valid[index]<=1;
    cache_dirty[index]<=1;
    if(cpu_bvalid && cpu_bready) begin
        cpu_bvalid<=0;
        req_type<=NONE;
        state<=IDLE;
    end
end

REFILL: begin
    bram_arvalid<=1;
    bram_araddr<=latched_addr;
    if(bram_arvalid && bram_arready) begin
        bram_arvalid<=0;
        state<=REFILL_ACK;
    end
end
```

```
    REFILL_ACK: begin
        bram_rready<=1;
        if(bram_rready && bram_rvalid) begin
            bram_rready<=0;
            cache_data[index]<=bram_rdata;
            cache_valid[index]<=1;
            cache_dirty[index]<=0;
            cache_tag[index]<=tag;
            state<=READ_MEM; //if we implement clean write refill then we need logic here to change this depending
        end
    end
    EVICT: begin
        bram_awaddr<={cache_tag[index], index};
        bram_awvalid<=1;
        bram_wdata<=cache_data[index];
        bram_wvalid<=1;
        if((bram_awvalid&&bram_awready)&&(bram_wready&&bram_wvalid)) begin
            bram_awvalid<=0;
            bram_wvalid<=0;
            bram_bready<=1;
            cache_valid[index]<=0; //safety?
            state<=EVICT_ACK;
        end
    end
    EVICT_ACK: begin
        if(bram_bready&&bram_bvalid) begin
            bram_bready<=0;
            if(req_type==READ) begin
                state<=REFILL;
            end else begin
                state<=OVERWRITE;
            end
        end
    end
```

```
        // Adding in the cache!!!
        cache #(.INDEX_BITS(8), .TAG_BITS(24))
        cache_mod (
            .clk            (clk            ), //do we need to invert?
            .rst        (resetn         ), //do we need to invert?

            //CPU connections
            .cpu_awvalid(c_cpu_awvalid),
            .cpu_awready(c_cpu_awready),
            .cpu_awaddr (c_cpu_awaddr ),
            .cpu_awprot (c_cpu_awprot ),
            .cpu_wvalid (c_cpu_wvalid ),
            .cpu_wready (c_cpu_wready ),
            .cpu_wdata  (c_cpu_wdata  ),
            .cpu_wstrb  (c_cpu_wstrb  ),
            .cpu_bvalid (c_cpu_bvalid ),
            .cpu_bready (c_cpu_bready ),
            .cpu_arvalid(c_cpu_arvalid),
            .cpu_arready(c_cpu_arready),
            .cpu_araddr (c_cpu_araddr ),
            .cpu_arprot (c_cpu_arprot ),
            .cpu_rvalid (c_cpu_rvalid ),
            .cpu_rready (c_cpu_rready ),
            .cpu_rdata  (c_cpu_rdata  ),

            //BRAM connections
            .bram_awvalid(c_bram_awvalid),
            .bram_awready(c_bram_awready),
            .bram_awaddr (c_bram_awaddr ),
            .bram_awprot (c_bram_awprot ),
            .bram_wvalid (c_bram_wvalid ),
            .bram_wready (c_bram_wready ),
            .bram_wdata  (c_bram_wdata  ),
            .bram_wstrb  (c_bram_wstrb  ),
            .bram_bvalid (c_bram_bvalid ),
            .bram_bready (c_bram_bready ),
            .bram_arvalid(c_bram_arvalid),
            .bram_arready(c_bram_arready),
            .bram_araddr (c_bram_araddr ),
            .bram_arprot (c_bram_arprot ),
            .bram_rvalid (c_bram_rvalid ),
            .bram_rready (c_bram_rready ),
            .bram_rdata  (c_bram_rdata  ),
        );

        assign mem_axi_awvalid=c_bram_awvalid////input wire cpu_awvalid,
        assign c_bram_awready=mem_axi_awready;//output reg cpu_awready, //We changed this (previously awread)
        assign mem_axi_awaddr=c_bram_awaddr////input wire [31:0] cpu_awaddr,
        assign mem_axi_awprot=c_bram_awprot////input wire [2:0] cpu_awprot, //ignore for now

        assign mem_axi_wvalid=c_bram_wvalid////input wire cpu_wvalid,
        assign c_bram_wready=mem_axi_wready;//output reg cpu_wready,
        assign mem_axi_wdata=c_bram_wdata////input wire [31:0]  cpu_wdata,
        assign mem_axi_wstrb=c_bram_wstrb////input wire [3:0]  cpu_wstrb, //ignore for now

        assign c_bram_bvalid=mem_axi_bvalid;//output reg cpu_bvalid,
        assign mem_axi_bready=c_bram_bready////input wire cpu_bready,

        assign mem_axi_arvalid=c_bram_arvalid////input wire cpu_arvalid,
        assign c_bram_arready=mem_axi_arready;//output reg cpu_arready,
        assign mem_axi_araddr=c_bram_araddr////input wire [31:0] cpu_araddr,
        assign mem_axi_arprot=c_bram_arprot////input wire [2:0] cpu_arprot,// ignore for now

        assign c_bram_rvalid=mem_axi_rvalid;//output reg cpu_rvalid,
        assign mem_axi_rready=c_bram_rready////input wire cpu_rready,
        assign c_bram_rdata=mem_axi_rdata;//output reg [31:0] cpu_rdata,

        picorv32 #(
```

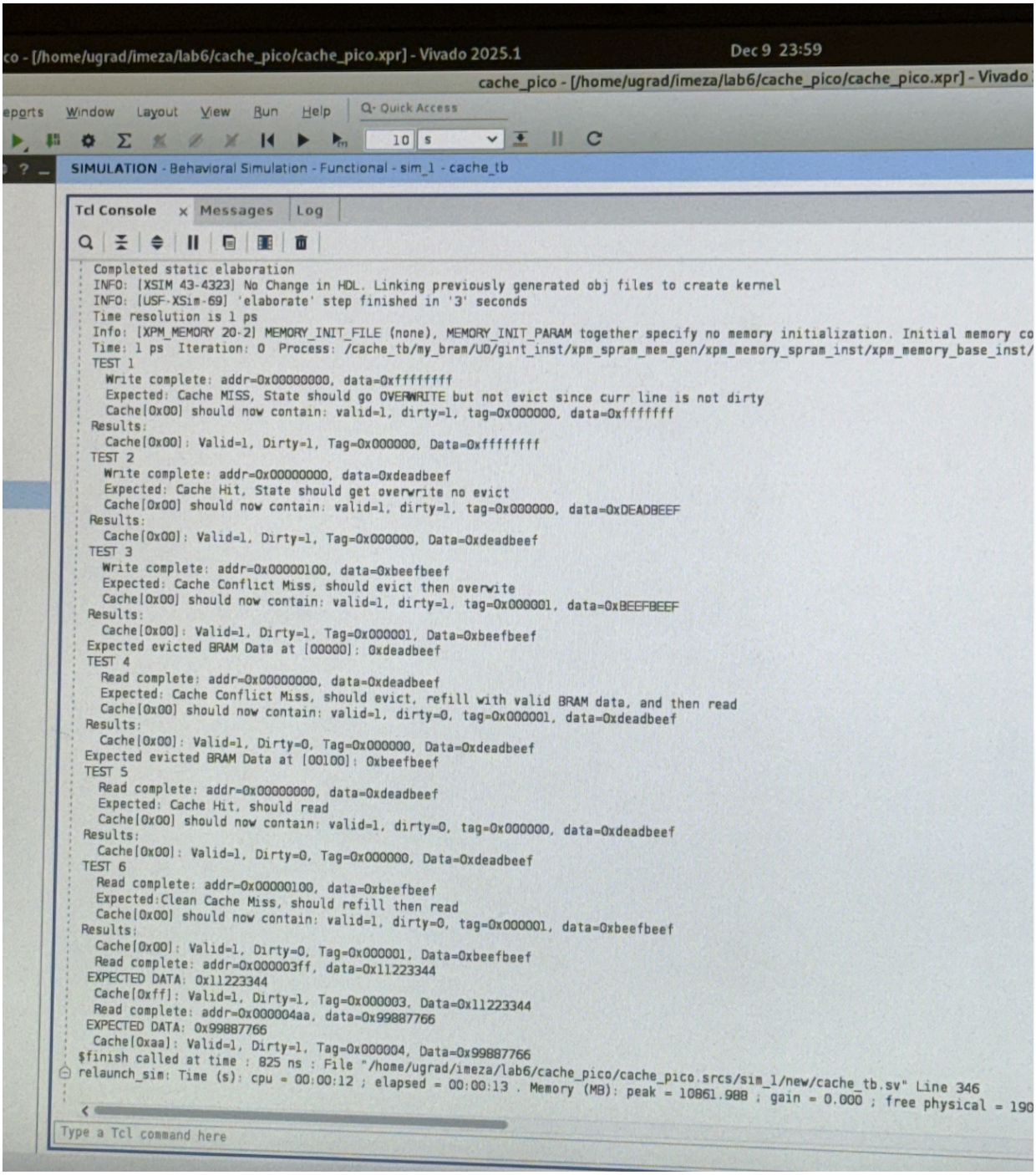
Testing:

Testbench Code:

```
initial begin
    cpu_awvalid=0;
    cpu_awaddr=0;
    cpu_wvalid=0;
    cpu_wdata=0;
    cpu_bready=0;
    cpu_arvalid=0;
    cpu_araddr=0;
    cpu_rready=0;rst=1; #100; rst=0;
    //TEST 1: BASIC WRITE MISS: Miss -> Overwrite
    //write to 0x000000_00 (8 bits index, the rest tag)
       $display("TEST 1");

    repeat(2) @(posedge clk);
   cpu_awvalid=1;
   cpu_awaddr={24'h000000, 8'h00}; //0x000000_00
   cpu_wvalid=1;
   cpu_wdata={32'hffffffff}; //write data
   wait(cpu_awready && cpu_wready&&cpu_wvalid&&cpu_awvalid);
   repeat (2) @(posedge clk);
   cpu_awvalid=0;
   cpu_wvalid=0;
   cpu_bready=1;
   wait(cpu_bvalid && cpu_bready);
   repeat (2) @(posedge clk);   cpu_bready=0;
   //Inspect cache address to make sure it has correct data
   //if this causes an eviction, investigate evicted address in bram to make sure it worked
   $display("  Write complete: addr=0x%h, data=0x%h", 32'h00000000, 32'hffffffff);
    $display("  Expected: Cache MISS, State should go OVERWRITE but not evict since curr line is not dirty");
    $display("  Cache[0x00] should now contain: valid=1, dirty=1, tag=0x000000, data=0xffffffff");
    $display("Results:"); print_cache_line(8'h00);
```

```
 //TEST 2: Basic overwrite on hit: Hit -> Overwrite
    $display("TEST 2");

repeat (2) @(posedge clk);
cpu_awvalid=1;
cpu_awaddr={24'h000000, 8'h00}; //0x000000_00
cpu_wvalid=1;
cpu_wdata={32'hdeadbeef}; //write data
wait(cpu_awready && cpu_wready);
repeat (2) @(posedge clk);
cpu_awvalid=0;
cpu_wvalid=0;
cpu_bready=1;
wait(cpu_bvalid);
repeat (2) @(posedge clk);   cpu_bready=0;
//Inspect cache address to make sure it has correct data
//if this causes an eviction, investigate evicted address in bram to make sure it worked
$display("  Write complete: addr=0x%h, data=0x%h", 32'h00000000, 32'hDEADBEEF);
 $display("  Expected: Cache Hit, State should get overwrite no evict");
 $display("  Cache[0x00] should now contain: valid=1, dirty=1, tag=0x000000, data=0xDEADBEEF");
 $display("Results:"); print_cache_line(8'h00);

 //TEST 3: Dirty Write Miss: Evict -> overwrite
    $display("TEST 3");

repeat (2) @(posedge clk);
cpu_awvalid=1;
cpu_awaddr={24'h000001, 8'h00}; //0x000000_00
cpu_wvalid=1;
cpu_wdata={32'hbeefbeef}; //write data
wait(cpu_awready && cpu_wready);
repeat (2) @(posedge clk);
cpu_awvalid=0;
cpu_wvalid=0;
cpu_bready=1;
wait(cpu_bvalid);
repeat (2) @(posedge clk);   cpu_bready=0;
//Inspect cache address to make sure it has correct data
//if this causes an eviction, investigate evicted address in bram to make sure it worked
$display("  Write complete: addr=0x%h, data=0x%h", 32'h00000100, 32'hbeefbeef);
 $display("  Expected: Cache Conflict Miss, should evict then overwite");
 $display("  Cache[0x00] should now contain: valid=1, dirty=1, tag=0x000001, data=0xBEEFBEEF");
 $display("Results:"); print_cache_line(8'h0);
$display("Expected evicted BRAM Data at [%h]: 0xdeadbeef",20'h00000);
```

```
//TEST 4 verify evict of 0x000000_00: DIRTY READ MISS
$display("TEST 4");
repeat (2) @(posedge clk);
cpu_arvalid=1;
cpu_araddr={24'h000000, 8'h00}; //0x000000_00
wait(cpu_arready);
repeat (2) @(posedge clk);   cpu_rready=1;cpu_arvalid=0;
wait(cpu_rvalid);
$display("  Read complete: addr=0x%h, data=0x%h", 32'h00000000, cpu_rdata);
$display("  Expected: Cache Conflict Miss, should evict, refill with valid BRAM data, and then read");
$display("  Cache[0x00] should now contain: valid=1, dirty=0, tag=0x000001, data=0xdeadbeef");
$display("Results:"); print_cache_line(8'h0);
$display("Expected evicted BRAM Data at [%h]: 0xbeefbeef",20'h00100);
repeat (2) @(posedge clk);cpu_rready=0;

//TEST 5: READ HIT
repeat (2) @(posedge clk);
cpu_arvalid=1;
cpu_araddr={24'h000000, 8'h00}; //0x000000_00
wait(cpu_arready);
repeat (2) @(posedge clk);   cpu_arvalid=0; cpu_rready=1;
wait(cpu_rvalid);
$display("TEST 5");
$display("  Read complete: addr=0x%h, data=0x%h", 32'h00000000, cpu_rdata);
$display("  Expected: Cache Hit, should read");
$display("  Cache[0x00] should now contain: valid=1, dirty=0, tag=0x000000, data=0xdeadbeef");
$display("Results:"); print_cache_line(8'h0);
repeat (2) @(posedge clk);cpu_rready=0;


//TEST 6: CLEAN READ MISS
repeat (2) @(posedge clk);
cpu_arvalid=1;
cpu_araddr={24'h000001, 8'h00}; //0x000001_00
wait(cpu_arready);
repeat (2) @(posedge clk);   cpu_arvalid=0;cpu_rready=1;
wait(cpu_rvalid);
$display("TEST 6");
$display("  Read complete: addr=0x%h, data=0x%h", 32'h00000100, cpu_rdata);
$display("  Expected:Clean Cache Miss, should refill then read");
$display("  Cache[0x00] should now contain: valid=1, dirty=0, tag=0x000001, data=0xbeefbeef");
$display("Results:"); print_cache_line(8'h0);
repeat (2) @(posedge clk);cpu_rready=0;
```



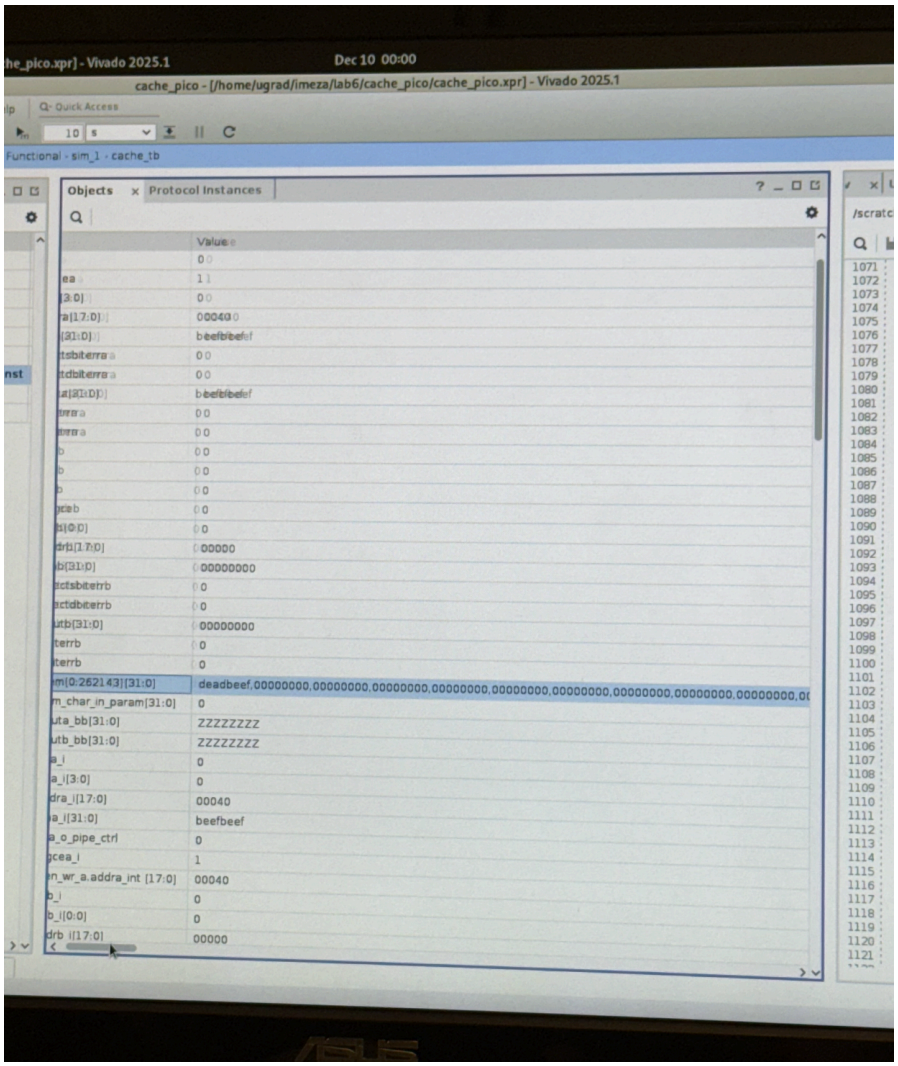
(Excerpt(s) of the testbench(es) for this problem.)

Testing Approach: For our testbench approach, our group wrote multiple unit tests that would check how our module handles different state conditions under the Read and Write operations. Primarily, we were trying to ensure that every condition of Reads and Writes was hit (hit, misses, dirty or not dirty bit, correct evicts), as this would be strong proof that our cache state logic was robust. In addition to the verification of all our different types of states, our group also

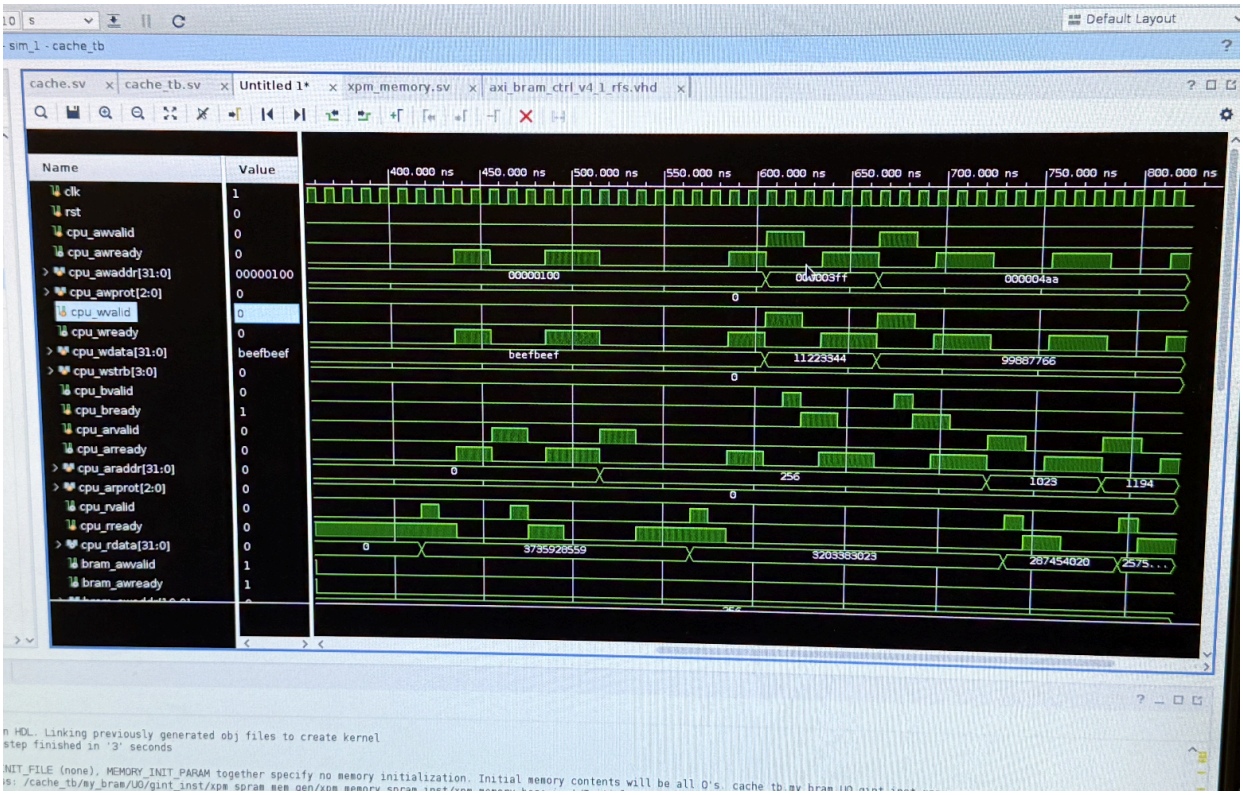
included some edge cases, such as writing to and reading from random addresses, and writing to and reading from the final index of the cache.

Results:

Outcomes: As shown in the waveforms and throughout the registers, our group was able to successfully write to the correct addresses with our newly edited module. However, we were unsuccessful in connecting our newly modified pico cpu to the FPGA board and generating a new bitstream at the very end.



Waveforms (Group): The waveform here shows different values being loaded into the cpu_wdata from our testbench waveform.



Reflections:

While working on this final project, our group encountered many obstacles along the way, thankfully overcoming most of them. Our group had many testbench problems, ensuring each state functioned correctly when provided data. Also, we had to make sure that our finite state machine logic was optimized using the correct clock cycles as it set the control signals properly at different rising/falling edges. One of the largest challenges we faced throughout this project was understanding the AXI Protocol and understanding how to properly handle AXI within the cache and to rewire and integrate our cache in-between the Pico and BRAM. We had a lot of trouble connecting all the wires and properly instantiating the cache. Ultimately, we were unable to process data successfully with our cache integrated with the Pico. However, in terms of the result, ideally our cache implementation would help speed up the efficiency of read-hits and write-hits. But the drawbacks of our read-misses and write-misses cause more delay than just the pico and BRAM itself. However, this can be workshopped and improved by adding more sets into our cache, increasing block size, and increasing cache lines, so the likelihood of misses would decrease.

Individual Contributions:

Beren - For our final project, I helped in brainstorming the pseudocode of our finite state machine for our group's cache module. After our pseudocode was completed, I began to translate our ideas into actual SystemVerilog code on Vivado starting with the IDLE state and going forward. Once my other group members took over coding, I also helped in the debugging process of the logic there. Additionally, I helped prepare the slides for our presentation working on the content for our instruction, approach, and testing content. Lastly, I helped write a lot of the final lab report here mainly contributing to the approach explanation, the photos of the code, the testing portion, and the introduction explanation and photos as well.

Ruby - For the final project, I helped by brainstorming the logistics of our project and it's implementation. I also helped with setting up the documents for all necessary documentation required and organized the general timeline of the project. When coming to the actual product, I helped with figuring out the logic and states for both the cache and the necessary AXI handoff protocols between our cache, pico, and the BRAM. I then helped with writing the cache code and the integration of our cache into the pico.v file and instantiating all the connecting wires between each different module/block and figuring out the proper input/outputs. I helped with making the final presentation, reformatting, finding templates, and putting all the content together with our pictures as well. And lastly, I helped with writing the goal and the reflection of our final project.

Michael - For the final project, I began by assisting in planning and theorizing our initial idea. After we came to an idea on the design of our state machine that defined our cache logic, it became time to gain a deep understanding of Axi protocol, knowing that our cache would have to handle both the manager and subordinate roles of this interaction. I assisted in the note taking and understanding of this information. After this, now that we had the components necessary to begin coding, I assisted in implementing and debugging our design in verilog. I also worked on editing the Pico code to accept our cache, configuring the Axi inputs and outputs of both modules, so that our block design from the Getting Started Guide could be used without changes in our final result.

Isaac - For this project, I helped come up with the idea of the cache, planning out the FSM, and implementing an outline. Since we needed to act as both manager and subordinate, me and Michael spent a lot of time reading the AMD AXILITE4 specification guide to help model out signal dependencies and timings and understand the protocols from both perspectives. I also helped code up the states for our cache, as well as debugging/testing.