Joseph Liba, Tianzhi Wu, Marta Taulet

19 April 2018

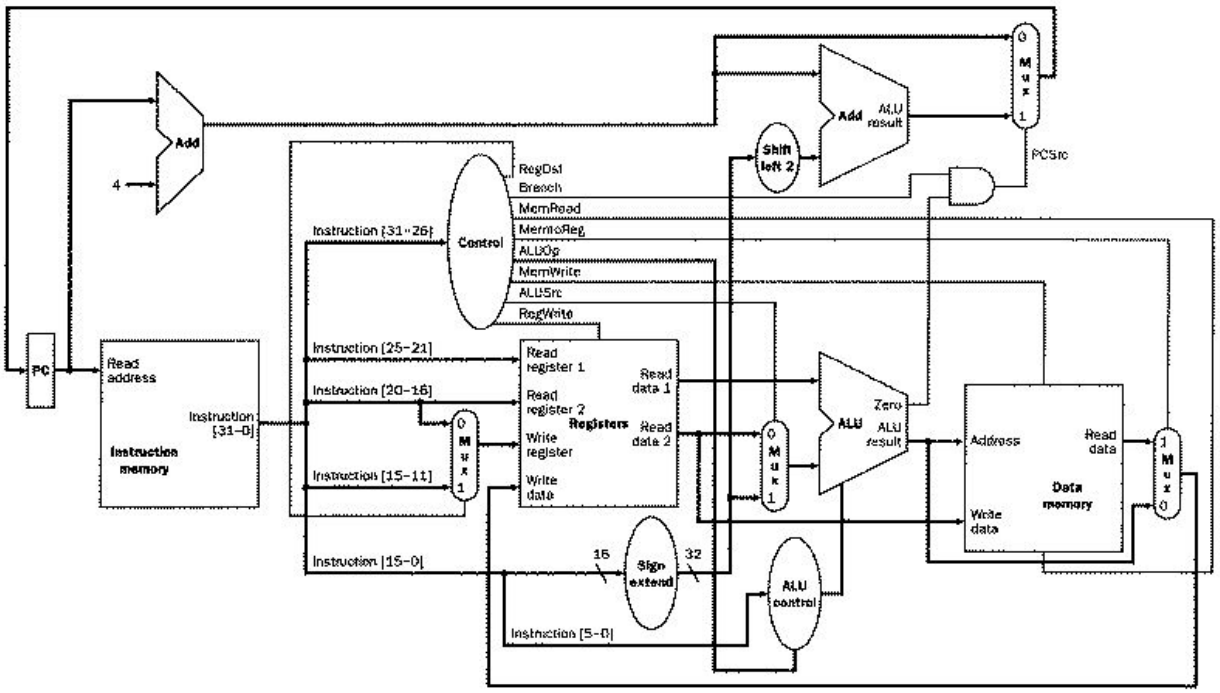
Mid-Project Report

Selection of the Number of Stages: Our team settled on having 5 stages in the pipeline for our superscalar architecture. This is because a 5 stage architecture is the most documented architecture. This makes implementation much easier. Furthermore, the verilog modules are neatly divided into 5 stages: fetch_unit, decode_unit, execute_unit, memory_unit, and writeback_unit.

One small challenge occurred when determining the number of pipeline stages: Should the control_unit be considered a stage on its own? Where should we put this in the control flow? We decided to have the decode and control_unit occur simultaneously. Two reasons:

1) In most single stage diagrams, both the control unit and decode stage receive inputs from the fetch stage. Therefore, it intuitively makes sense that they would occur

simultaneously.



2) Control_unit and decode_unit are very interdependent. For example, control_requires the opcode as an input. The opcode is an output of the decode stage. Normally this would logically order control after the decode stage. However, the decode stage also takes as input some of the outputs of the decode stage. Both of these stages therefore need to happen simultaneously within a single clock cycle.

One final challenge with deciding the number of stages in the pipeline was deciding whether writeback should have its own buffer. Then on the next cycle, the writeback would occur. We discussed with another team. Their implementation was that the writeback has no buffer, since at that point, it can just immediately send the result to the register file. We decided to follow their implementation since it would be unclear what the data would need to wait for when held in the

writeback stage. However, this decision could potentially be one of the sources of our challenges with writing data.

Dividing the tasks: These divisions are just general guidelines since the expectation is that we would have some overlap and be able to help each other in the following tasks:

Tianzhi Wu - 5 stage pipelining

Joseph Liba - Designing the superscalar architecture

Marta Taulet - Cache design

5 stage pipeline (Joseph's report): Although I was tasked with starting the superscalar architecture, I realized very early on that I would need a functioning pipelined architecture to build on top of before I could reasonably get work done with the module's. I sent out several messages to the group, particularly Tianzhi, asking how the progress with the pipelining was going. I would have liked at minimum the multicycle design working, since I can still build on that. However, there were no responses in the group chat. I also did not see any group members on April 13th (class time) to work on the project. I sent out a request to the group on April 15th but no reply.

I decided to do Tianzhi's part for him since I was not sure if he was not working on it. My first objective was to get multicycle pipelining working, without any bypass. The testbench I used was the default testbench with gcd.mem. To achieve this, for each over the modules (fetch, decode, control_unit, execute, memory, writeback), I created buffer modules. The buffer modules receive as inputs the outputs of the previous stage, as well as any other outputs from previous stages that might need to get pipelined to stages further down the pipeline. The buffer stores register outputs for each input. On the positive edge of a clock cycle, the register output

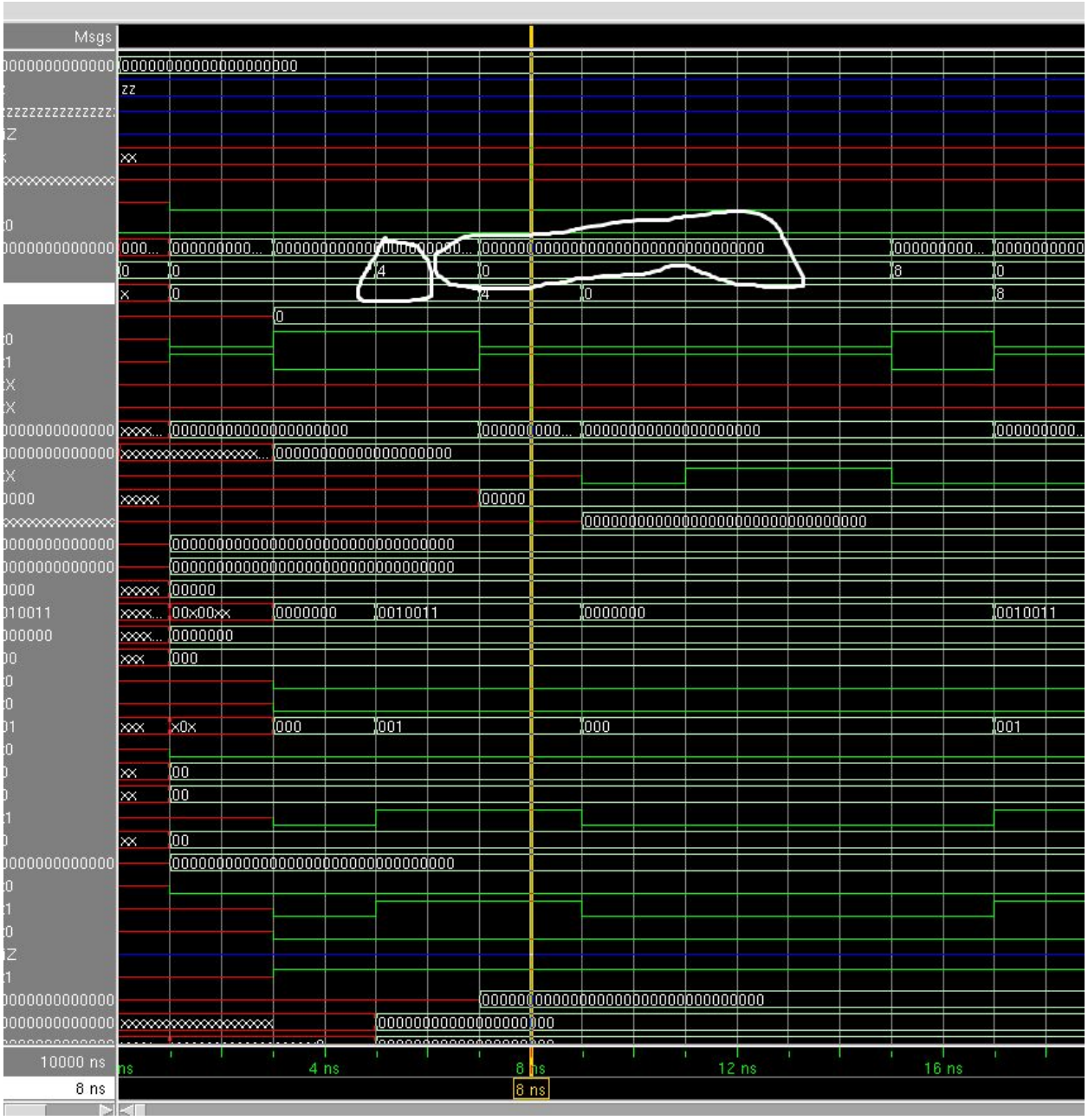
updates to the value of the current input. Because this update occurs only on the positive edge of the clock, each stage remains distinct. Here is an example of one of the modules:

```
module fetch_buffer #(parameter CORE = 0, DATA_WIDTH = 32, INDEX_BITS = 6,
                      OFFSET_BITS = 3, ADDRESS_BITS = 20)(
        clock,
        instruction,
        inst_PC,
        valid,
        ready,
        reg_instruction,
        reg_inst_PC,
        reg_valid,
        reg_ready
);

input [DATA_WIDTH-1:0]   instruction;
input [ADDRESS_BITS-1:0] inst_PC;
input valid;
input ready;
input clock;

output reg [DATA_WIDTH-1:0]   reg_instruction;
output reg [ADDRESS_BITS-1:0] reg_inst_PC;
output reg reg_valid;
output reg reg_ready;

always @ (posedge clock) begin

     reg_instruction <= instruction;
     reg_inst_PC <= inst_PC;
     reg_valid <= valid;
     reg_ready <= ready;

end



endmodule
```

Furthermore, here is an example of the module for RISC-V processor, indicating how the modules connect to their appropriate buffers:

```
reg  [31:0]  to_peripheral_data;
reg          to_peripheral_valid;

wire [ADDRESS_BITS-1: 0] e_JALR_target;
wire [ADDRESS_BITS-1:0] e_branch_target;
wire [1:0] e_next_PC_sel;
wire [ADDRESS_BITS-1: 0] e_JAL_target;
wire e_branch;

fetch_unit #(CORE, DATA_WIDTH, INDEX_BITS, OFFSET_BITS, ADDRESS_BITS) IF (
        .clock(clock),
        .reset(reset),
        .start(start),

        .PC_select(e_next_PC_sel),
        .program_address(prog_address),
        .JAL_target(e_JAL_target),
        .JALR_target(e_JALR_target),
        .branch(e_branch),
        .branch_target(e_branch_target),

        .instruction(instruction),
        .inst_PC(inst_PC),
        .valid(i_valid),
        .ready(i_ready),

        .report(report)
);

wire [DATA_WIDTH-1:0]  f_instruction;
wire [ADDRESS_BITS-1: 0] f_inst_PC;
wire f_i_valid, f_i_ready;

fetch_buffer #(CORE, DATA_WIDTH, INDEX_BITS, OFFSET_BITS, ADDRESS_BITS) IFB (
        .clock(clock),
        .instruction(instruction),
        .inst_PC(inst_PC),
        .valid(i_valid),
        .ready(i_ready),
        .reg_instruction(f_instruction),
        .reg_inst_PC(f_inst_PC),
        .reg_valid(f_i_valid),
        .reg_ready(f_i_ready)
);

decode_unit #(CORE, ADDRESS_BITS) ID (
        .clock(clock),
```

The following is a waveform to indicate that this multicycle stall implementation works to some degree:



Here, the PC starts at 4, then is set to 0 for 4 cycles, then goes to 8. Thus, it stalls for the entirety of the instruction.

For testing purposes, I created a very rudimentary stall implementation. This stall is even worse than the usual stall: After every instruction fetch, the next instruction fetch stalls until the current instruction is done. Here is the logic to this:

Applications Places System

fetch.v

File Edit Search Options Help

```
mem_interface #(CORE, DATA_WIDTH, INDEX_BITS, OFFSET_BITS, ADDRESS_BITS)
                    i_mem_interface (
                     .clock(clock),
                     .reset(reset),
                     .read(fetch),
                     .write(1'b0),
                     .address(inst_addr),
                     .in_data(0),
                     .out_addr(out_addr),
                     .out_data(instruction),
                     .valid(valid),
                     .ready(ready),
                     .report(report)
);

reg [31: 0] cycles;
reg [31: 0] cycleStart;

always @ (posedge clock) begin
      if (reset) begin
        fetch        <= 0;
        PC_reg       <= 0;
        old_PC       <= 0;
      end
      else begin
        if (start) begin
            fetch        <= 1;
            PC_reg       <= program_address;
            old_PC       <= 0;
        end
        else begin
            if (cycles-cycleStart!=5 && !branch) begin
            fetch        <= 0;
            old_PC       <= PC_reg;
            end
            else begin
            fetch        <= 1;
            PC_reg       <= (PC_select == 2'b10)?  JAL_target:
                            (PC_select == 2'b11)?  JALR_target:
                            ((PC_select == 2'b01)& branch)?  branch_target : PC_plus4;
            old_PC       <= PC_reg;
            cycleStart = cycles;
            end
        end
      end
end


always @ (posedge clock) begin
    cycles <= reset? 0 : cycles + 1;
```

In essence, I am really just forcing the processor to stall for 5 cycles. If I can get the processor to work when stalling every time, I can get the processor to work when deriving the appropriate stall signal.

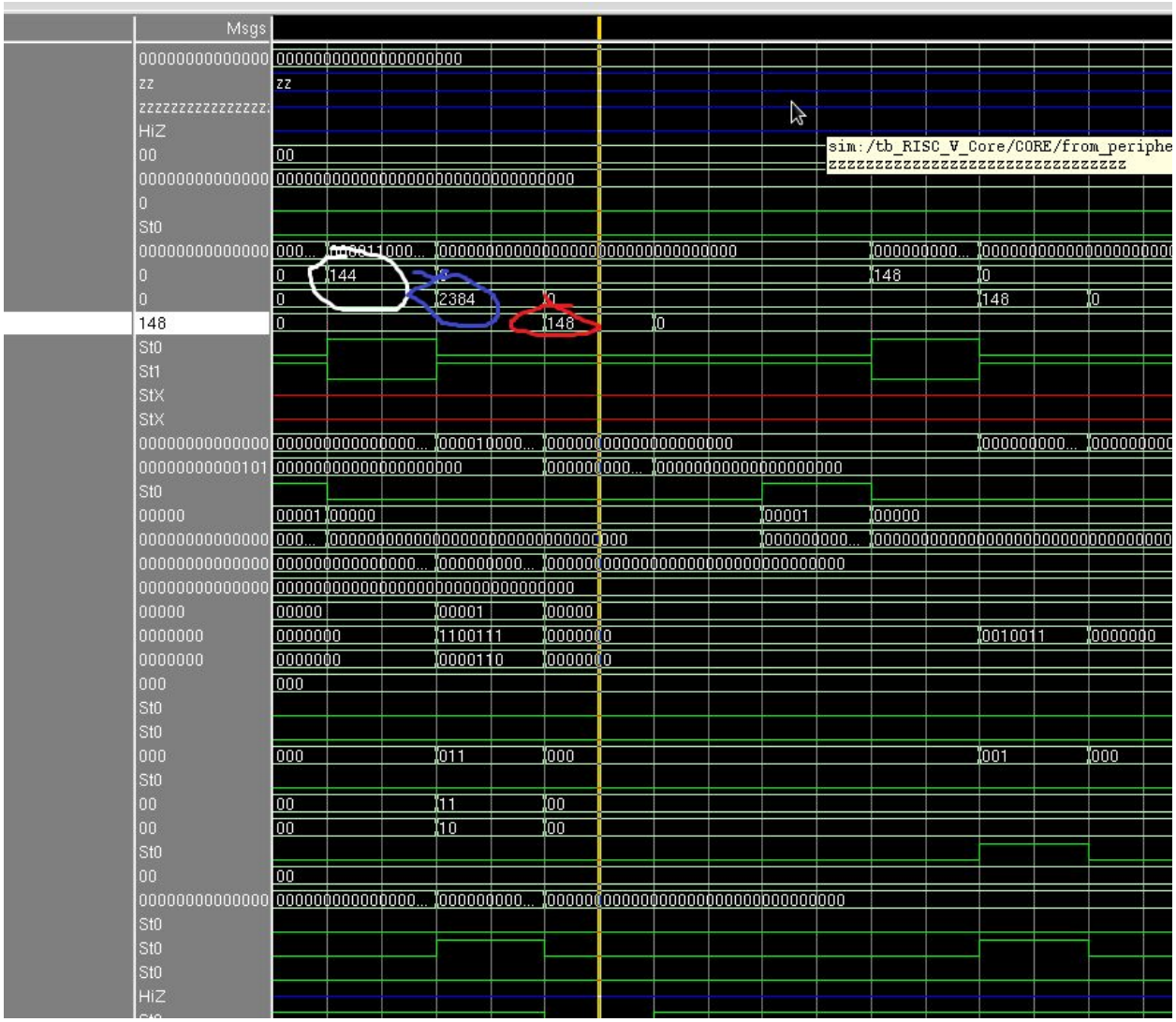
Here is the stall showing the stages to the pipeline.

24 indicates the program counter (indicated in the white circle). Notice that it takes 2 cycles to calculate the ALU result to be 1536 (indicated in the red circle). Therefore, it properly pipelines the stages from fetch to execute.

Here is the gcd.mem instruction that it is calling:

```
        addi    zero,zero,0
        addi    zero,zero,0
        addi    zero,zero,0
        addi    zero,zero,0
kernel:
        addi    zero,zero,0
        addi    ra,zero,0
        addi    sp,zero,1536
        addi    gp,zero,0
        addi    tp,zero,0
        addi    t0,zero,0
        addi    t1,zero,0
        addi    t2,zero,0
        addi    s0,zero,0
        addi    a0,zero,0
        addi    a1,zero,0
        addi    a2,zero,0
        addi    s3,zero,0
        addi    a4,zero,0
        addi    a5,zero,0
        addi    a6,zero,0
        addi    a7,zero,0
        addi    s2,zero,0
        addi    s3,zero,0
        addi    s4,zero,0
        addi    s5,zero,0
        addi    s6,zero,0
        addi    s7,zero,0
        addi    s8,zero,0
        addi    s9,zero,0
        addi    s10,zero,0
        addi    s11,zero,0
        addi    t3,zero,0
        addi    t4,zero,0
        addi    t5,zero,0
        addi    t6,zero,0
        call    main
        addi    zero,zero,0
        mv          s1,a0
```

The processor still does not work. At the following location in the waveform, the processor is supposed to branch to the main function. The ALU result is successfully calculated. However, no matter what we try to do with the code, the branch never successfully occurs. I tried pipelining the branch signal and target from the previous instruction to the fetch of the new instruction, but it still does not work. This is what occurs when we try to branch:



The white circle indicates the program counter for “call main”. However, the blue circle, which is branch target, is very odd, and the ALU result is just PC + 4. Therefore it never branches. I was not able to get branching to work correctly.

Although I would have liked to complete a successfully working pipeline, the short notice that we are still far behind made it difficult to complete this task in time. Had I taken the role earlier of pipelining instead of working on the superscalar research, I could have had more time

to ask other teams their implementations and get a working model of the superscalar implementation. Since it is unlikely to work, and we need to get superscalar implementation working for our final presentation, I would like to request that we receive a functioning bypassed 5 stage pipelined RISC-V architecture to start off with. That way, we will be able to appropriately devote our time towards developing the superscalar architecture and learning about its details and nuances.

(Note: The cache files are located in /hardware/cache-src. We did not have time to combine the pipelined CPU and the caching.

Multicache System (Marta's Report):

The memory system I implemented consists on the following modules. The mem_interface is the main module that combines and synchronizes the memory system. The second main module is the main_memory module, which acts as a RAM. The third one is the cache module. I chose to have L1 and L2 caches because having more levels would just mean instantiating more cache modules in mem_interface, since the design is very modular. This would not add quality of design but just increase the probability of errors since we would have more wires in the design. The cache is synchronous and gets the Read, Write, Address and Data inputs from the memory interface. If the data requested is found in the cache, it asserts the Hit signal. The logic for checking if the data is in the cache was not difficult to implement since we had already done a lab in caches before.

One of the challenging parts about the memory system was implementing when the Enable signals had to be asserted and by what module. At first, I thought about having a

cache_controller module to which all the caches and RAM were connected to. This only added a lot of extra complexity since it had to enable a cache, wait for its hit signal, and decide whether or not to enable the following ones. It would also have to keep track of what caches it has checked so far. I opted for a more intuitive implementation instead. When the memory interface gets a data request, it enables the L1. The L1 checks for the data inside the cache and if it is found then it emits the Hit signal and it is done. If there is a miss in L1, it Enables L2 and because it has all necessary data from the mem_interface inputs in can process the request. If it is a hit, it emits the Hit signal and it is done and if it is a miss, it enables main memory. Because with main memory there is always a hit, the success signal is called ready. In this way, the individual caches are responsible for enabling the following ones and issuing stalls until there is a hit.

The following challenge was deciding what to do when the data we want to read is not in the lower cache and we want to bring it there for following requests (satisfying the purpose of having caches in the first place). When we find the data in other caches or memory, these modules issue an update signal to the smaller caches and forward the data for them to be updated with the latest request.

The write policy I implemented was the following: Whenever we want to write, if the desired address is in the cache, then we make that piece of data's valid bit 0 (dirty bit) and enable the following caches to do the same until we reach main memory, in which we finally write it. Even though this method would cause cache misses in the future because we are not updating the caches with the newest information, it was simple to implement and did not interfere with the Enable signals' logic.

The stall signal is derived as follows. Whenever we are reading, we are not done until we make sure the data is present in L1, so we stall until then. When we write, we stall until the new data reaches main memory.

The only thing left in the implementation was the instruction cache. I had trouble implementing the one-memory architecture as opposed to the separate instruction and data memories which we have most commonly seen in class.

Overall, implementing a multilevel cache hierarchy was more difficult that I had anticipated and if I had managed my time better, as well as getting more help from classmates, I could have achieved a better much result. I will try my best to make it work for the superscalar architecture in Phase II.