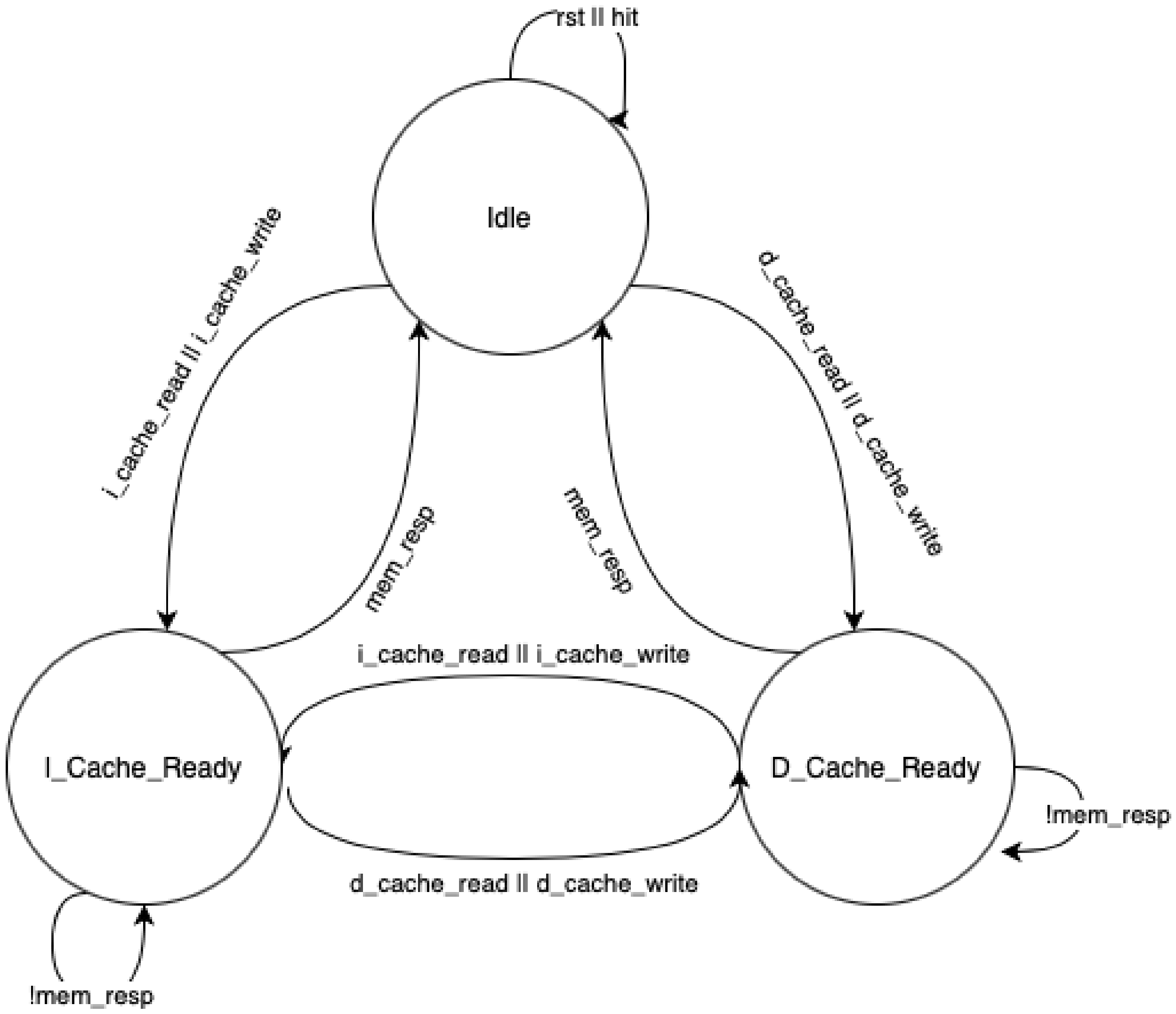
Arbiter, Hazard Detection, and Forwarding Design

Arbiter:



rst || hit
Idle
i_cache_read || i_cache_write
d_cache_read || d_cache_write
mem_resp
mem_resp
i_cache_read || i_cache_write
I_Cache_Ready
D_Cache_Ready
!mem_resp
d_cache_read || d_cache_write
!mem_resp

!mem_resp takes precedence over cache reads and writes.
Further, the data cache will be served first in the event that both caches miss at the same time.

At State:

Idle: All signals set to 0

I_Cache_Ready:
i_cache data = data from cache adapter
i_cache resp = mem_resp
read_out = read signal from i_cache
write_out = write signal from i_cache
addr_out = address from the i_cache

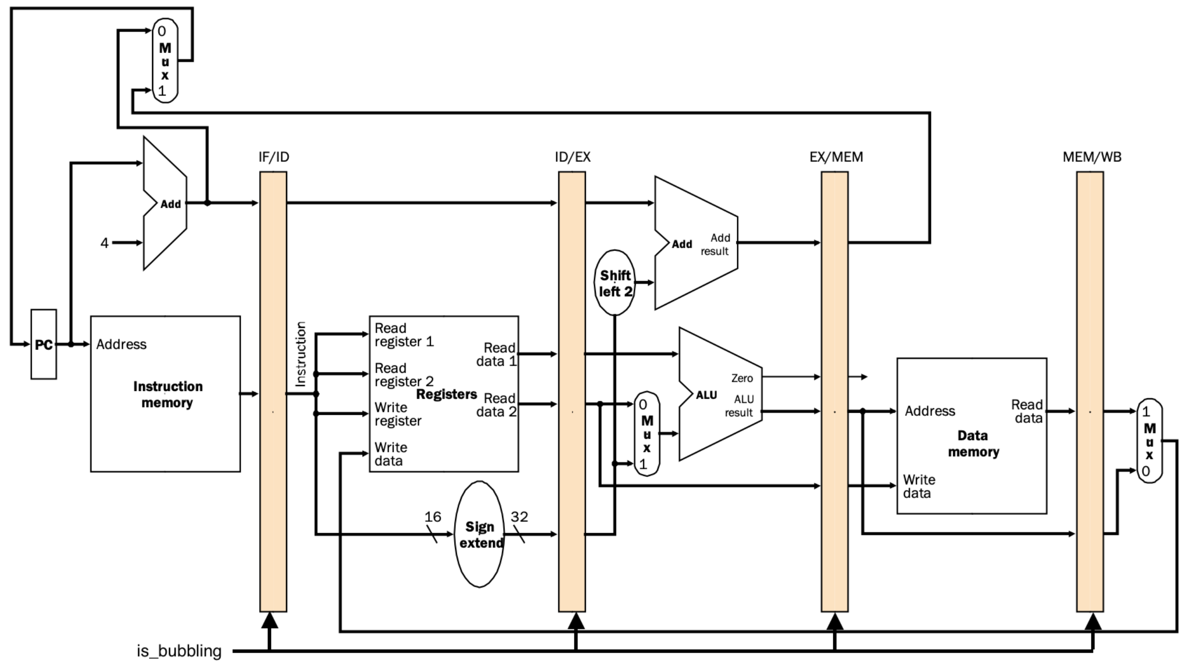
D_Cache_Ready:
d_cache_data = data from cache adapter
d_cache_resp = mem_resp
read_out = read signal from d_cache
write_out = write signal from d_cache
addr_out = address from the d_cache

Hazard Detection:

As mentioned in lecture, there are three categories of hazards to address:

- Structure Hazards: When one of the connected devices is busy. This will occur on a cache miss.
- Data Hazard: Occurs when an instruction needs the result from previous instructions to complete.
- Control Hazards: Whenever a control operation occurs, the next instruction to fetch will depend on the control operation's output.

Before discussing hazards, it is worth talking about **stalls**. In order to stall a stage in the pipeline, we need to disable the output signals to any of our memory devices. We already have a baseline for this in our "is_bubbling" signal. We also need to disable all register load signals in that stage, so that the state of the stage does not change. This is done by also using our "is_bubbling" signal to prevent registers from loading



Control Hazards:

Detecting Control Hazards (occurs during branch instruction, when deciding whether to branch in the MEM stage)

1. We can always stall as soon as we see a branch/jump. (Not very good for timing as it takes 3 cycles per branch)

2. We can assume the branch is not taken, but if the branch is actually taken, we proceed to edit the following/flush the instruction by:
    a. IF: zero out instruction field in IF/ID
    b. ID: zero out control lines
    c. EX: zero out alu_out, rs1, rs2

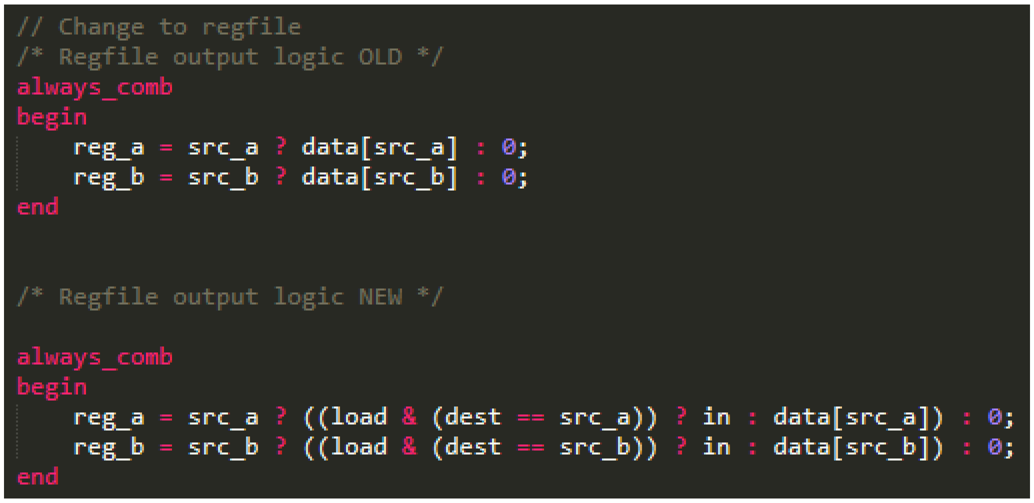
    We are effectively turning these three instructions into nop's.
3. Finally, we can create a history table to predict whether or not a branch will be taken. DIfferent history tables for loops and jumps in order to predict the branch most accurately. If we are wrong, proceed to flush the next instructions like in method 2.

    The approach that we are using for CP2 is item #2, assuming the branch is not taken.

Data Hazards:

Detecting Data Hazards
One way to address hazards is stalling. This is not ideal, but explained here.
Between instruction i and i + 1:
ID/EX.write_reg == (IF/ID.rs1 || IF/ID.rs2) incorporate 3 bubbles
Between instruction i and i + 2:
EX/MEM.write_reg == (IF/ID.rs1 || IF/ID.rs2) incorporate 2 bubbles
Between instruction i and i + 3:
MEM/WB.write_reg == (IF/ID.rs1 || IF/ID.rs2) incorporate 1 bubbles
Bubbles/stalls stop instructions in the ID stage, which means we must stop fetching new instructions otherwise the PC and IF/ID register will be clobbered.

The better way to address data hazards is by using forwarding. It will allow the instructions to get the outputs of previous instructions before they are written to the regfile. This will allow us to remove many of the stalls that we would need above. The hazard has a reach of up to three instructions. We can effectively reduce this to 2 if we change the regfile to the 'transparent design', where the data that we read is the same as the data that we write on the event that we are writing into the regfile and rd==rs1/rs2. This is shown in the following change to the regfile code.



```
// Change to regfile
/* Regfile output logic OLD */
always_comb
begin
    reg_a = src_a ? data[src_a] : 0;
    reg_b = src_b ? data[src_b] : 0;
end


/* Regfile output logic NEW */

always_comb
begin
    reg_a = src_a ? ((load & (dest == src_a)) ? in : data[src_a]) : 0;
    reg_b = src_b ? ((load & (dest == src_b)) ? in : data[src_b]) : 0;
end
```

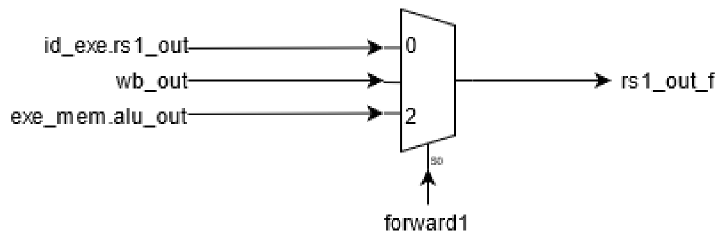
Data hazards are explained in more detail in the forwarding section.

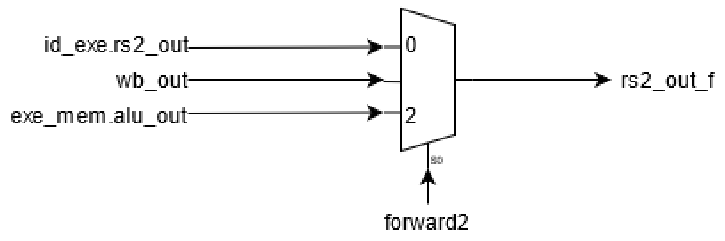
Structure Hazards:

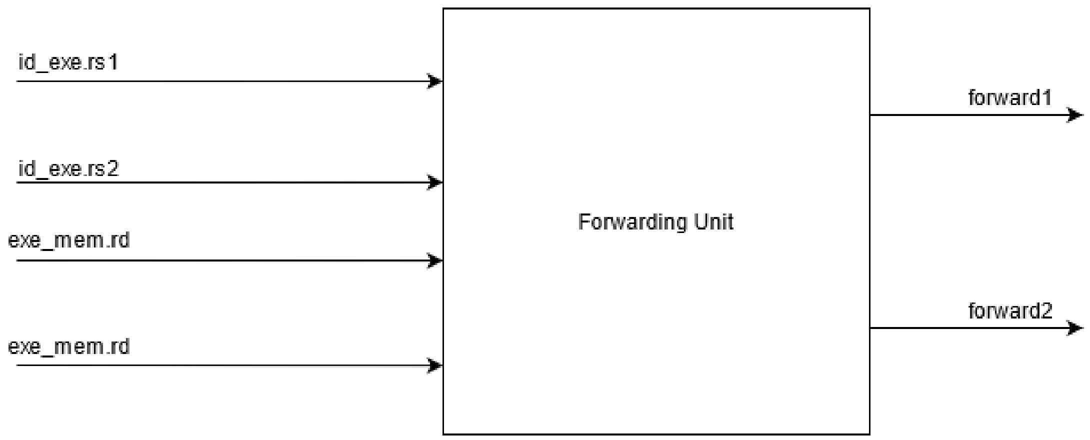
Structure hazards occur on cache misses. This will cause a very large delay on either the IF or MEM stage. In order to address this, all we need to do is stall the previous stages until the mem_resp is high. The way we stall is explained above.

If both caches miss at the same time, the data cache is served first. This will stall the previous stages (except IF). After MEM is served, the instructions in ID, EXE, and MEM will be allowed to complete while IF is stalled for the data read. Then the stages will again stall as they wait for new instructions.

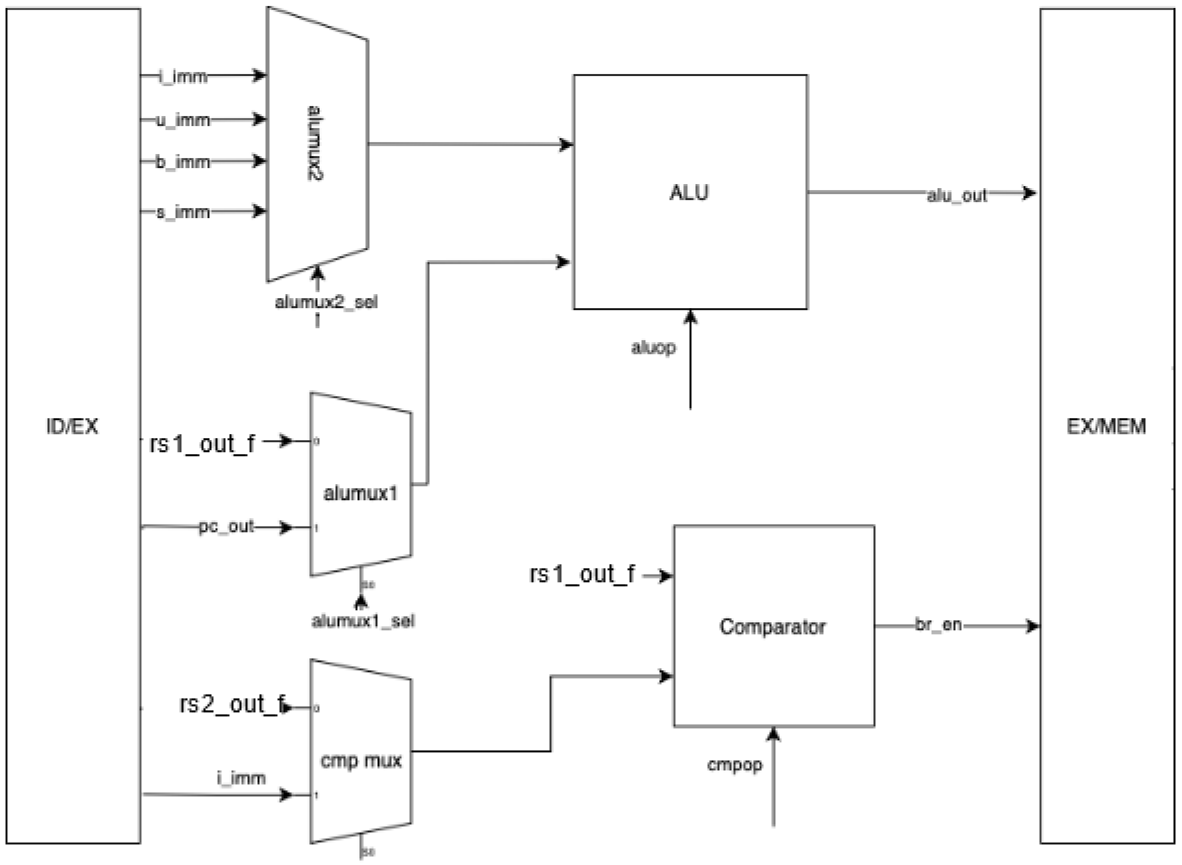
Forwarding Design:



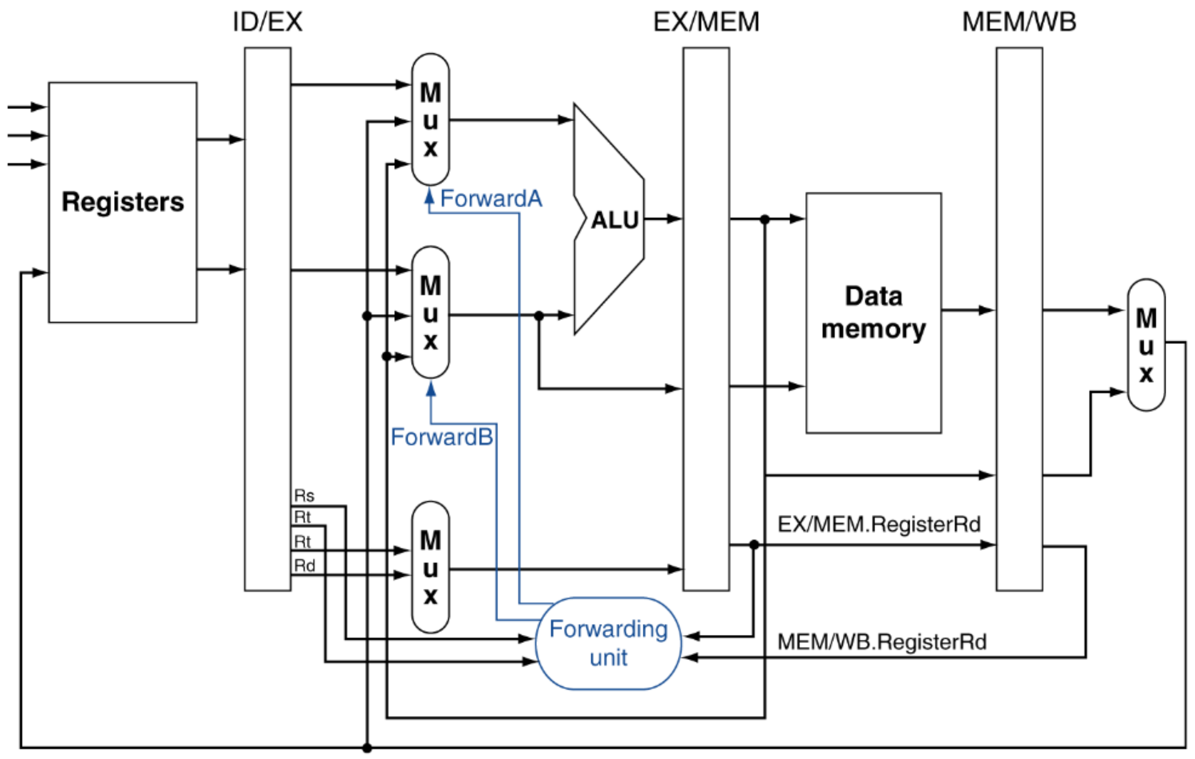




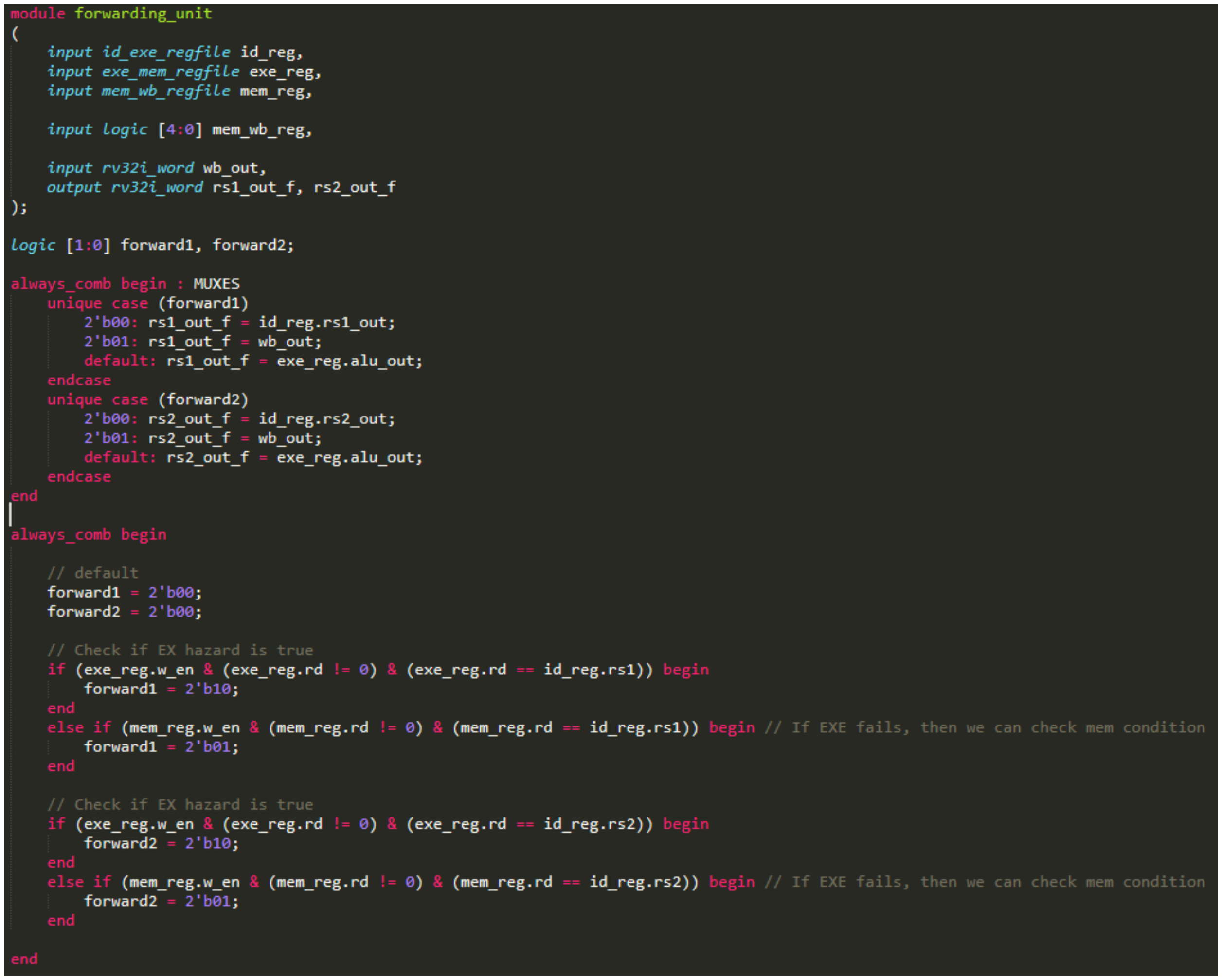
Above is the forwarding design. The signals are labeled with the name of the regfile that they come from. The outputs are ‘rs1_out_f’ and ‘rs2_out_f’. These signals permanently replace rs1_out and rs2_out in the MEM and WB stages.



The above image shows the changes to the EXE stage in the datapath to include forwarding.



Above image is taken from lecture 5. While this diagram does not match our design exactly, it very clearly shows the connections into the forwarding unit. These are the same connections made in the previous two images.



```
module forwarding_unit
(
    input id_exe_regfile id_reg,
    input exe_mem_regfile exe_reg,
    input mem_wb_regfile mem_reg,

    input logic [4:0] mem_wb_reg,

    input rv32i_word wb_out,
    output rv32i_word rs1_out_f, rs2_out_f
);

logic [1:0] forward1, forward2;

always_comb begin : MUXES
    unique case (forward1)
        2'b00: rs1_out_f = id_reg.rs1_out;
        2'b01: rs1_out_f = wb_out;
        default: rs1_out_f = exe_reg.alu_out;
    endcase
    unique case (forward2)
        2'b00: rs2_out_f = id_reg.rs2_out;
        2'b01: rs2_out_f = wb_out;
        default: rs2_out_f = exe_reg.alu_out;
    endcase
end

always_comb begin

    // default
    forward1 = 2'b00;
    forward2 = 2'b00;

    // Check if EX hazard is true
    if (exe_reg.w_en & (exe_reg.rd != 0) & (exe_reg.rd == id_reg.rs1)) begin
        forward1 = 2'b10;
    end
    else if (mem_reg.w_en & (mem_reg.rd != 0) & (mem_reg.rd == id_reg.rs1)) begin // If EXE fails, then we can check mem condition
        forward1 = 2'b01;
    end

    // Check if EX hazard is true
    if (exe_reg.w_en & (exe_reg.rd != 0) & (exe_reg.rd == id_reg.rs2)) begin
        forward2 = 2'b10;
    end
    else if (mem_reg.w_en & (mem_reg.rd != 0) & (mem_reg.rd == id_reg.rs2)) begin // If EXE fails, then we can check mem condition
        forward2 = 2'b01;
    end

end
```

The above diagram and control address most of the concerns for data hazards. Forwarding allows us to execute consecutive operations with almost no issues. The updated data will be sent to a previous stage before it updates the regfile. One issue that this does not address perfectly is a load operation followed by an operation with the loaded data. In this case, we need to stall or bubble for one cycle before the arithmetic operation reaches the EXE stage. This will of course be longer on a cache miss.