Prof. Salma Elmalaki
Winter 2022

Lab 4 (100 Points)

In this lab, we want you to design a pipeline Processor.

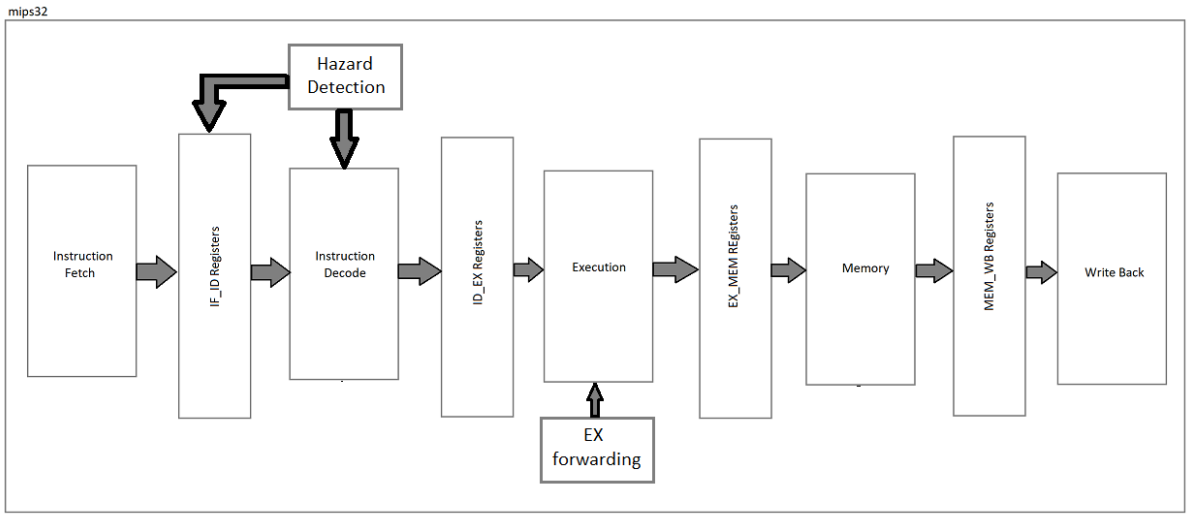
1 Pipeline Processor

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. In a single cycle design, every instruction takes exactly one cycle. While in a pipeline processor, the datapath is broken into separate independent stages. MIPS instructions classically take five steps:

- 1. Fetch the instruction from memory.
- 2. Read registers while decoding the instruction.
- 3. Execute the operation or calculate an address.
- 4. Access an operand in data memory.
- 5. Write the result into a register.

Hence, the MIPS pipeline has five stages, named IF (Instruction Fetch) , ID (Instruction Decode), EXE (Execution), MEM (Memory) and WB (Write Back). Figure 1 shows a pipeline processor. You see the pipeline registers are located between every two stages.

Figure 1: Pipeline processor



The pipeline improves performance by increasing instruction throughput. The idea behind pipelining is to keep all the stages busy at all times. As an example, when an instruction is using the ALU, the register file and the instruction memory are used by other instructions. Every instruction seems to have its datapath. All instructions advance during each cycle from one pipeline register to the next. In this case, all the required information of an instruction such as control signals and registers need to be stored in the pipeline registers.
There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types.

- Structural hazard. When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute
- Data hazard. Also called a pipeline data hazard. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available. Adding extra hardware to retrieve the missing item early from the internal resources is called forwarding or bypassing.
- Control hazard. Also called branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

To avoid delay in our pipeline when one of these hazards happen, we use forwarding.
To implement forwarding, we need to a) detect the data dependencies by adding the forwarding unit, and b) add the required paths to enable forwarding.
Forwarding cannot solve all the problems. There are some cases that we cannot avoid delay by forwarding in those cases the pipeline will stall for 1 or more clock cycles. We need to detect these cases with Hazard Detection unit.

Now we will go through each pipeline stage with more detail.

1.1 Instruction Fetch

Figure 2 shows the Instruction Fetch stage of the pipeline. In this stage instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. In case there is no brach or jump instructions, the PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. For branch and Jump instructions we need to send branch address, and jump address to the PC. So we need two multiplexers and the select signals. We also need to save the PC_plus4 in the IF/ID pipeline register in case it is needed later (to calculate the branch address in the next stage) for instruction, such as beq. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

Here is sample code for Instrcuction Fetch stage. Please use this for the module part of the IF_pipe_stage

Code 1: IF_pipe_stage.

```
module IF_pipe_stage(
    input  clk, reset,
    input  en,
    input  [9:0] branch_address,
    input  [9:0] jump_address,
    input  branch_taken,
    input  jump,
    output [9:0] pc_plus4,
    output [31:0] instr
    );
```

1.2 IF/ID Registers

we use pipeline registers to store the data that we may need in the next stages. They are flip_flops.

you may use this code sample for IF/ID pipeline registers. Other pipeline registers don't need enable and flush input. So this one is a little bit different.

Figure 2: Instruction Fetch.

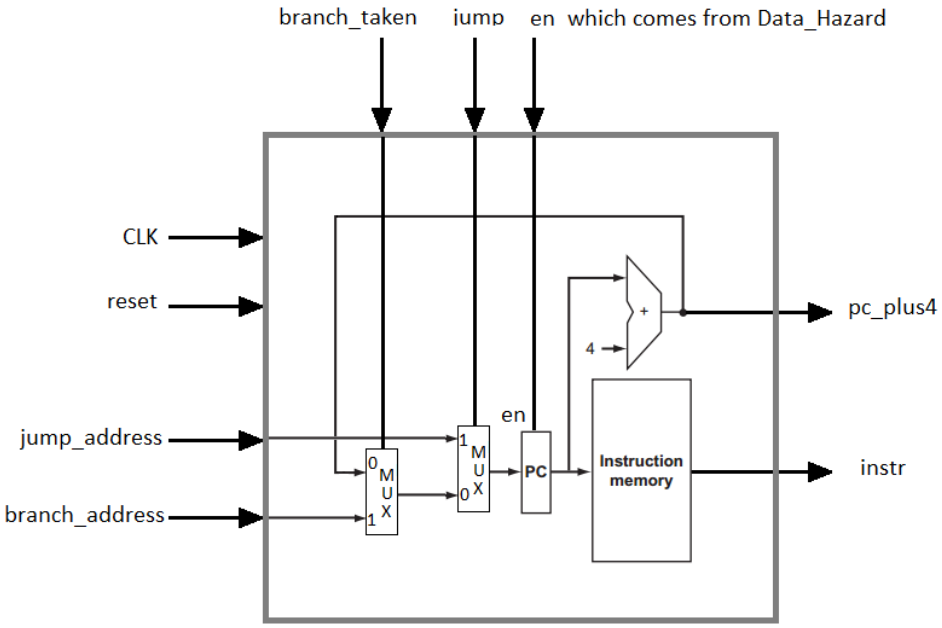
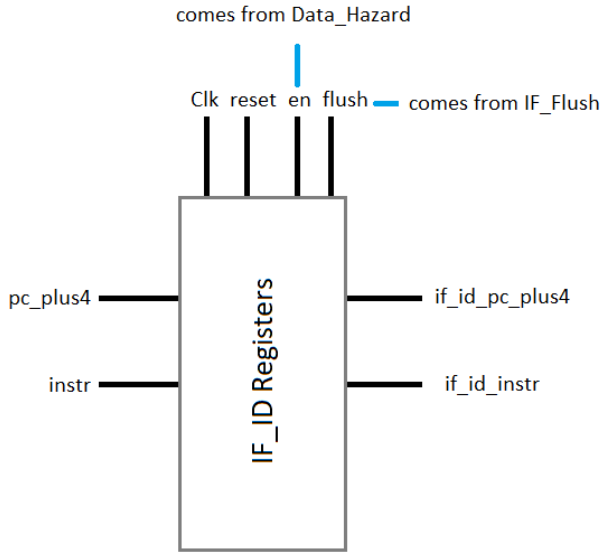


Figure 3: IF/ID Registers.



Code 2: IF/ID Pipeline Register.

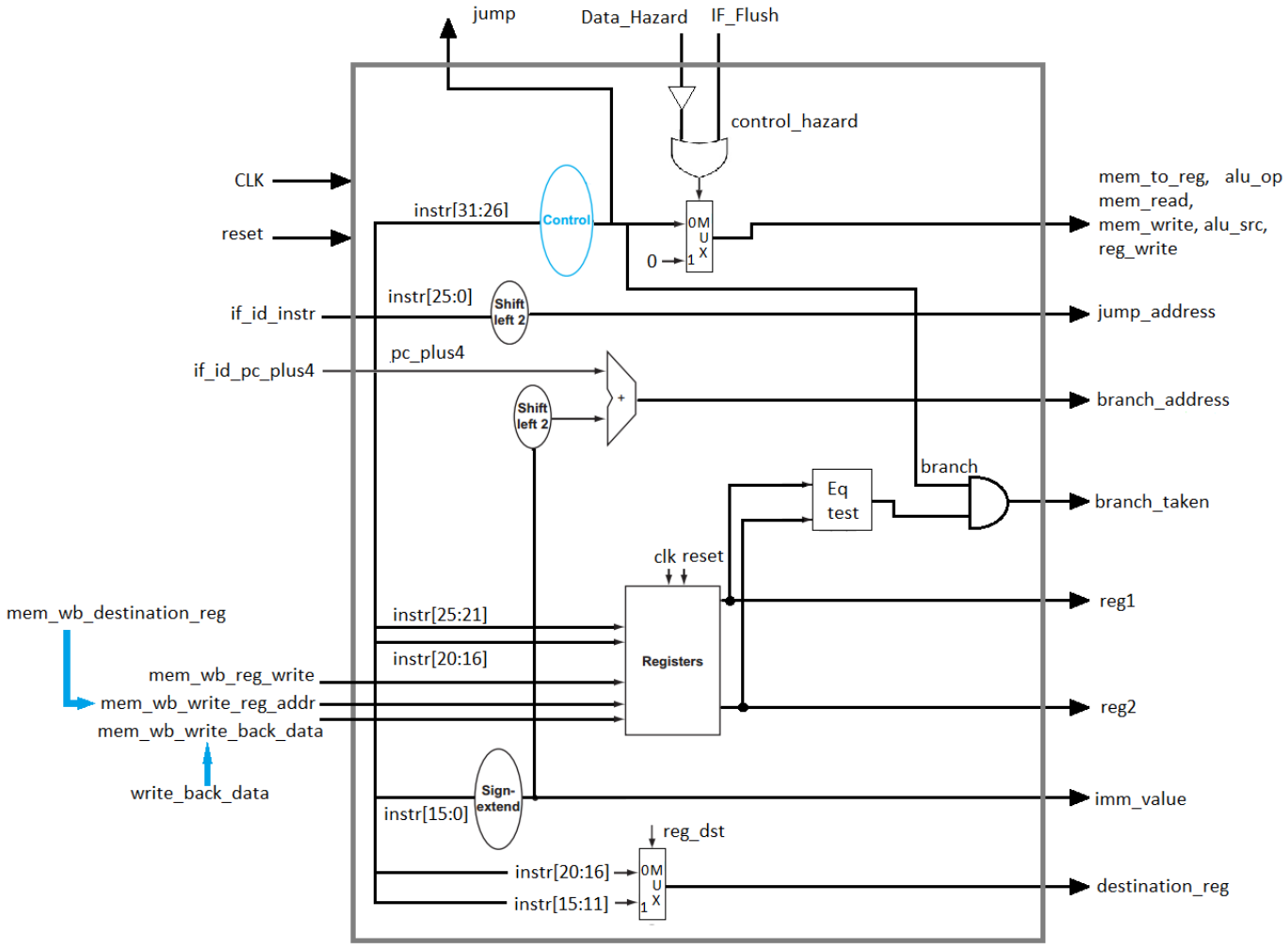
```
module pipe_reg_en #(parameter WIDTH = 8)(
    input clk, reset,
    input en, flush,
    input [WIDTH-1:0] d,
    output reg [WIDTH-1:0] q
    );

    always @(posedge clk or posedge reset)
    begin
        if(reset)
```

```
            q <= 0;
        else if (flush)
            q <= 0;
        else if (en)
            q <= d;
    end
endmodule
```

1.3 Instruction Decode

Instruction decode and register file read. Figure 3 shows the Instruction decode stage. The instruction portion of the IF/ID pipeline register supplying the 16 bit immediate field, which is sign extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the control signals from the Control unit. You see here that we also calculate the branch and jump addresses. We also have an equality test unit which helps us to find out if the branch is taken or not.

Figure 4: Instruction Decode.



Use this sample code for the module part of the ID_pipe_stage

Code 3: ID_pipe_stage.

```
module ID_pipe_stage(
    input  clk, reset,
    input  [9:0] pc_plus4,
    input  [31:0] instr,
    input  mem_wb_reg_write,
    input  [4:0] mem_wb_write_reg_addr,
    input  [31:0] mem_wb_write_back_data,
    input  Data_Hazard,
    input  Control_Hazard,
    output [31:0] reg1, reg2,
    output [31:0] imm_value,
    output [9:0] branch_address,
    output [9:0] jump_address,
    output branch_taken,
    output [4:0] destination_reg,
    output mem_to_reg,
    output [1:0] alu_op,
    output mem_read,
    output mem_write,
    output alu_src,
    output reg_write,
    output jump
    );
```

Note1: We need to make a small change in our register file code.

Code 4: Change writing clk edge.

```
always @ (negedge clk ) begin
```

The reason for this change is that both Instruction decode and Write back stage are using register file. With this change we are dividing the register file use to two half clk cycles. in the first half clk cycle we read in the second half clk cycle we write.

Note2: The equality test unit takes two registers as inputs and tells us if they are equal or not. In the single cycle processor we used the ALU to subtract two registers to find out if they are equal or not. Here we can use XOR gate which is faster and less complicated than subtraction.

Code 5: equality test.

```
((reg1 ^ reg2)==32'd0) ? 1'b1: 1'b0;
```

1.4 ID/EX Registers

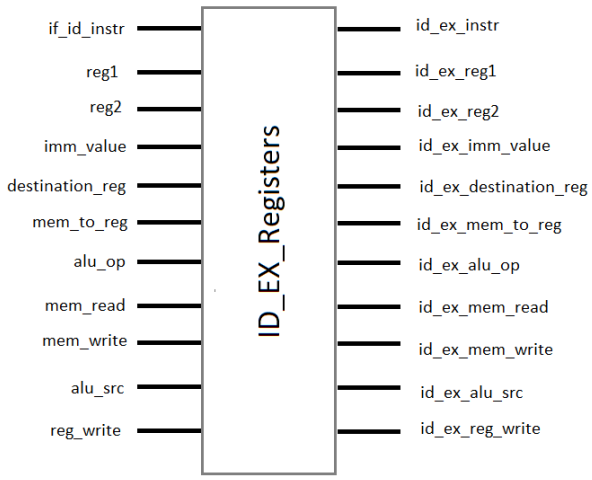
Same as Instruction Fetch stage, here also we need to send some signals to the next stages and we will use ID/EX pipeline registers for this.

Here is a sample code for pipeline registers(ID/EX, EX/MEM, and MEM/WB)

Code 6: Pipeline Register.

```
module pipe_reg #(parameter WIDTH = 8)(
    input clk, reset,
    input [WIDTH-1:0] d,
    output reg [WIDTH-1:0] q
    );

    always @(posedge clk or posedge reset)
    begin
        if(reset)
            q <= 0;
        else
            q <= d;
    end
endmodule
```

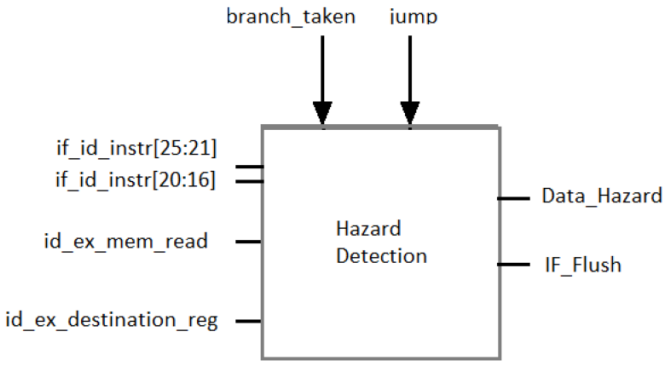
Figure 5: ID/EX Registers.



1.5 Hazard Detection

There are some cases that we need to stall the pipeline. For example, when a branch instruction is in the Decode stage the next instruction after that would be in the Fetch stage. If the branch is taken then we need to go to a new address and get rid of the instruction that is already in the IF stage. The same thing happens with jump instructions. We need a module to detect these situations and insert NOP in the pipeline. Below you see the Hazard detection module.

Figure 6: Hazard Detection.



Sample code for the Hazard detection unit:

Code 7: Hazard_detection.

```
module Hazard_detection(
    input id_ex_mem_read,
    input [4:0]id_ex_destination_reg,
    input [4:0] if_id_rs, if_id_rt,
    input branch_taken, jump,
    output reg Data_Hazard,
    output reg IF_Flush
    );

    always @(*)
    begin

        if ((id_ex_mem_read == 1'b1) &
            ((id_ex_destination_reg == if_id_rs) | (id_ex_destination_reg == if_id_rt)) )
```

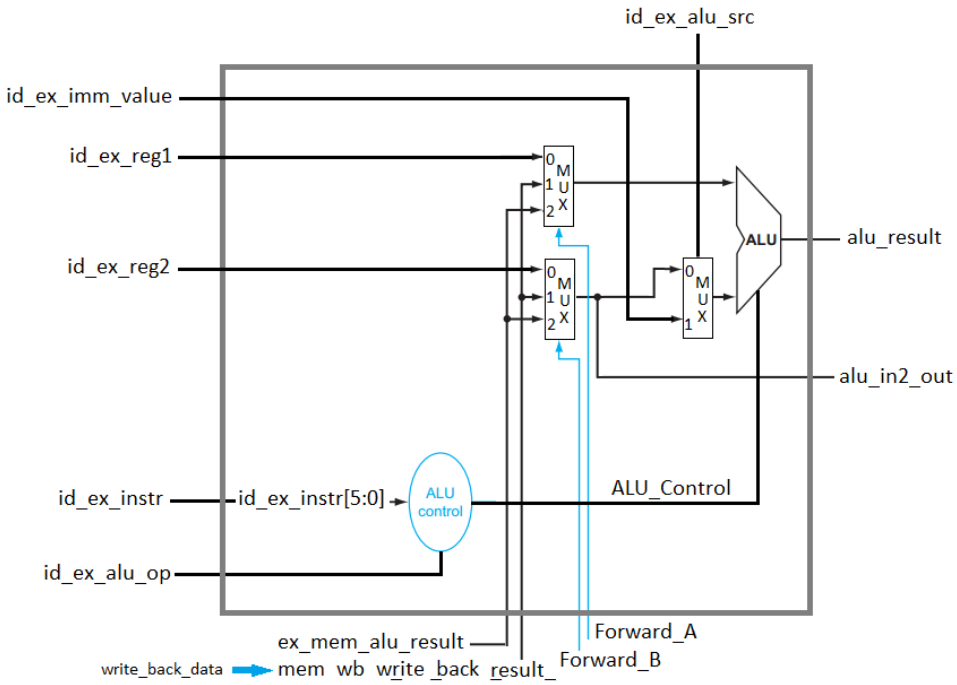
```
            Data_Hazard = 1'b0;
        else
            Data_Hazard = 1'b1;

        IF_Flush = branch_taken | jump;
    end
```

Data_Hazard output is active low and makes the pipeline to stall for 1 clock cycle. IF_Flush makes the IF/ID registers zero.

1.6 Execution

Execute or address calculation. In this stage we use the ALU to execute some logical or arithmetic operations. For load and store instructions we read the contents of register 1 and the sign extended immediate from theID/EX pipeline register and adds them using the ALU to find the memory address. That result is placed in the EX/MEM pipeline register. The ALU Control unit is also in this stage and takes the 2 bit alu_op from ID/EX pipeline register and calculate the 4 bit ALU_Control signal for the ALU.

Figure 7: Execution.



The sample code for the EX_pipe_stage:

Code 8: EX_pipe_stage.

```
module EX_pipe_stage(
    input [31:0] id_ex_instr,
    input [31:0] reg1, reg2,
    input [31:0] id_ex_imm_value,
    input [31:0] ex_mem_alu_result,
    input [31:0] mem_wb_write_back_result,
    input id_ex_alu_src,
    input [1:0] id_ex_alu_op,
    input [1:0] Forward_A, Forward_B,
    output [31:0] alu_in2_out,
    output [31:0] alu_result
    );
```

We need a forwarding unit here in the execution stage. Here is the sample code for the module part of the Forwarding_unit:

Code 9: Forwarding_unit.

```
module Forwarding_unit(
    input ex_mem_reg_write,
    input [4:0] ex_mem_write_reg_addr,
    input [4:0] id_ex_instr_rs,
    input [4:0] id_ex_instr_rt,
    input mem_wb_reg_write,
    input [4:0] mem_wb_write_reg_addr,
    output reg [1:0] Forward_A,
    output reg [1:0] Forward_B
    );
```

Under these conditions, we may need to forward data from EX/MEM registers or MEM/WB registers back to the Execution stage. If these conditions are not true the default value for Forward_A and Forward_B is 00.

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠  0)
       and (EX/MEM.RegisterRd =  ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠  0)
       and (EX/MEM.RegisterRd =  ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

1.7 EX/MEM Registers

We send some signals to the next stages using EX/MEM pipeline registers.

1.8 Memory

In the Memory stage, we will write to the memory (store word), read from memory (load word), or bypass the data memory (for all other instructions). in this stage, we only have the data memory. So, there is no need to create a new module.

1.9 MEM/WB Registers

We send some signals to the next stages using MEM/WB pipeline registers.

Figure 8: EX/MEM Registers.

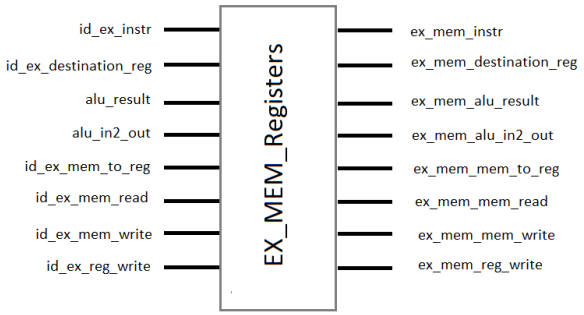


Figure 9: Memory.

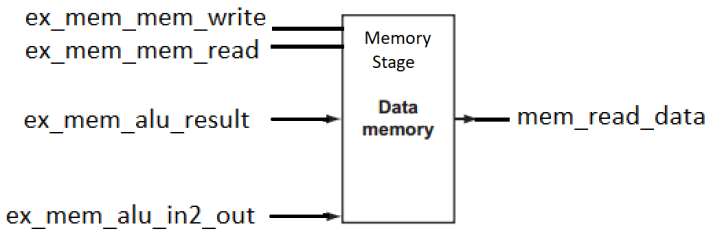
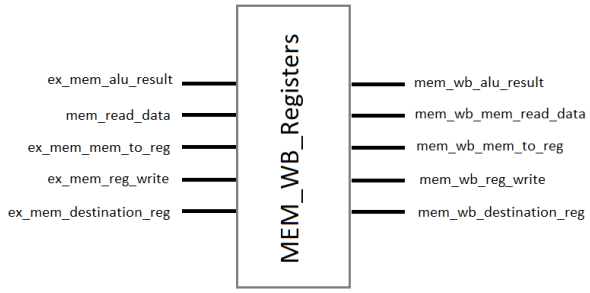


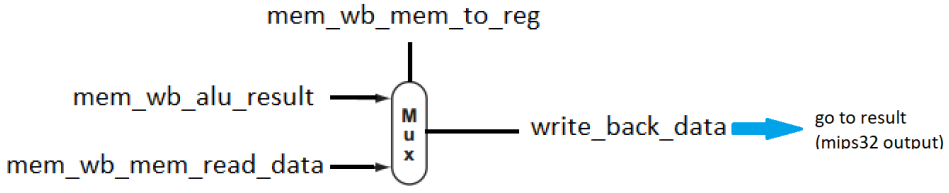
Figure 10: MEM/WB Registers.



1.10 Write Back

in the final stage we will read the data from the MEM/WB pipeline register and writing it back into the register file.

Figure 11: Write Back.



since it is only a mux you can implement it in the mips32.v file. No need to create a new module.

2 Test Pipeline Processor

To test your pipeline you need some instructions in the instruction memory. You can use some of the instructions in the Instruction_mem.v, that we gave you for the single cycle processor. Test your pipeline with a small set of instructions. This will help you with debugging. Use the waveform to track signals through pipeline stages.

If you are testing your design using the testcases we provide in Lab 3, you may see that your design doesn't pass some beq and jump instructions. The reason is that there is another type of hazard that can occur which is Load then Beq or R-type instruction then Beq. Since we are making the equality comparison in the decode stage for the beq, a hazard can occur. That's why some of the tescases in Lab 3 won't pass. You are not asked to handle this hazard or handle the forward for this case. Please follow the diagram shown in the lab manual (there is no hazard detection for this case and no forward unit for this case). The testcases we will use to test your design will not test for this hazard.

Debugging insights:

1. When you see Z in your waveform, check if that signal has any connections or not (sometimes you defined a wire without connecting it).
2. When you see X in the waveform, check if you are connected more than one source to a signal.
3. Double-check the number of bits. For example, when you assign two wires, they should have the same number of bits.
4. Make sure that the "address" and "data" lines of the data memory are coming from correct sources.

3 Assignment Deliverables

Your submission should include the following:

- Block designs. (mips_32.v, pipe_reg_en.v, pipe_reg.v, IF_pipe_stage.v, mux.v, ID_pipe_stage.v, register_file.v, sign_extend.v, mux4.v, control.v, Hazard_detection.v, EX_pipe_stage.v, ALU.v , ALUControl.v, EX_Forwarding_unit.v, data_memory.v)
- A report in **pdf** format. Follow the rules in the "sample report" under "additional resources" in the Canvas.

Note1: Compress all files (17 files : 16 .v files + report) into zip and upload to the **CANVAS** before deadline.

Note2: Use the code samples that are given. **The module names and the port names should exactly look like the code sample otherwise you lose points**.

Note3: we are going to test the content of data memory so please make sure that you are writing your results back to the register file (for R_type instructions) or in the data memory (for SW instructions).