CprE 381 – Computer Organization and Assembly Level Programming, Fall 2018

Project B (Pipelined MIPS)

In this three-week, group lab assignment you will implement a 5-stage pipelined MIPS processor. The architecture shall support at least 6 MIPS standard instructions: ADD, ADDI, LW, SW, BEQ, J. This is the minimum set of instructions to support the full control/data path presented in lecture. Many of the components for the processor are provided including the components from Project-A and the pipeline registers. You will need to build your own hazard detection and forwarding units. It is highly suggested that you implement and successfully test the processor without hazard detection and forwarding before introducing those components.

0) Setup your project directory for Project-B. Here is a minimum schedule you should follow to complete this project:

- **Week 1:** Integrate the pipeline registers and run the test programs w/out dependencies **(2, 3)**
- **Week 2:** Implement branching into the pipeline **(4)**
- **Week 3:** Implement forwarding and hazard detection and verify all test programs **(5)**

1) There is an issue with the operation of our register file when we insert pipeline registers that has to do with a conflict between the ID and WB stage. This issue arises because of the timing of writing all registers into both the pipeline register and register file. Explain the conflict and how it relates to assumption that we will be making for forwarding and hazard detection (Hint: It has to do with the fact that we assume an instruction in the ID stage will get the data being written by the WB stage). Give an example sequence of instructions that the processor we are designing will not be able to properly handle without additional logic or another change. A quick fix that we can use that will not impact SIMULATION is to change the register file to a falling edge triggered register file.

Extra Credit (5 points) (This is not required, if complete please answer these questions at the end of the report):

- Explain the potential impact that the falling edge trigger register fill has on the critical path for the processor we are designing (2.5 points).
- Implement a fix (in VHDL) to the issue we have caused that will allow the register file to continue to write on the rising edge. Draw (or screenshot) the change that you have made and explain how it solves the potential issue.

2) A **5-stage MIPS Processor** is a simple RISC architecture that executes any arbitrary standard MIPS instruction with a latency of five clock cycles for most instructions. However, the throughput is still 1 instruction per cycle. This allows roughly 5x speedup in clock frequency because the critical path in the circuit is simply the longest stage of the pipeline. For this project, you will implement a 5-stage MIPS processor. You may combine the provided components (register file, ALU, memories, etc.) to build the architecture using Quartus or structural VHDL. For design reference, see the diagram below from the textbook of the 5-stage MIPS architecture.

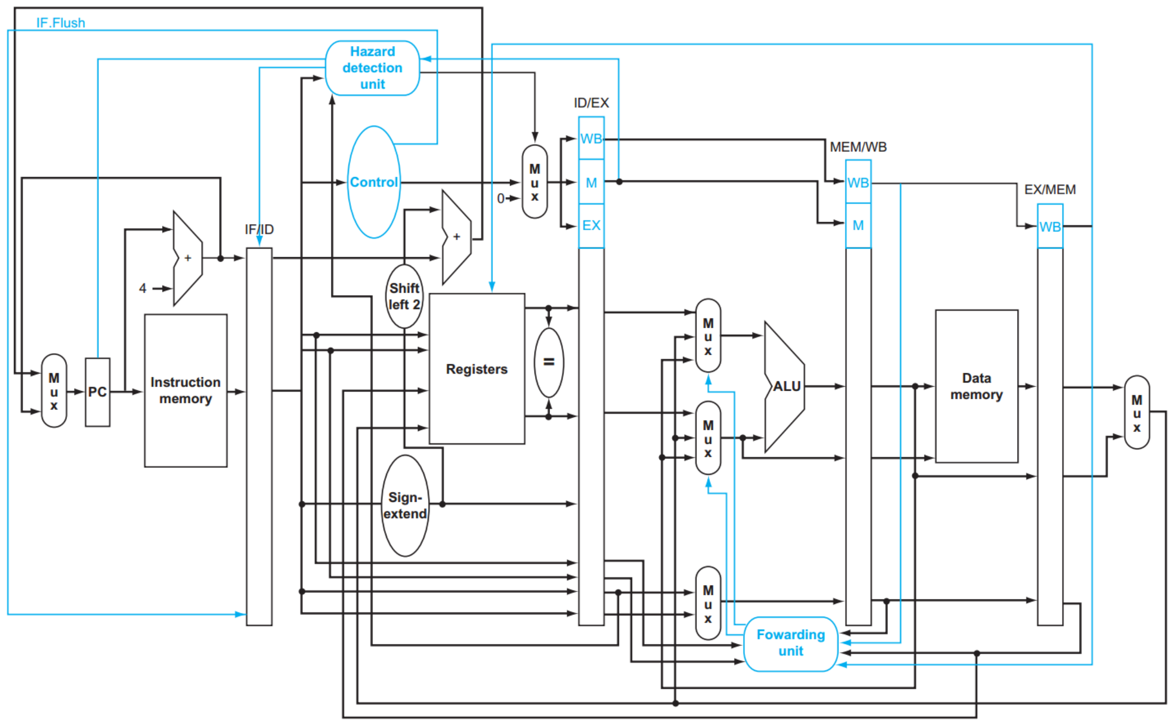


FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

The diagram above is a good starting point but it does not describe all parts of the 5-stage processor you are implementing. For example, it does not show the path of a "jump" instruction.

3) In this project, you will implement a version of the 5-stage MIPS processor. Your 5-stage processor will support the same instructions as the single-cycle from project-A (ADD, ADDI, LW, SW, BEQ, J). You may start from scratch or **make a copy** of Project-A and make changes accordingly. The first step is to insert the 4 pipeline registers: IF/ID, ID/EX, EX/MEM, and MEM/WB. The pipeline registers are provided for you in the *Components* folder under *pipeline registers*. Avoid inserting hazard detection/forwarding logic until after you've tested the processor w/pipeline registers (i.e. test it with instructions that don't cause data dependencies or just ignore data dependencies). The class textbook discusses an overview of pipelining in section 4.5 and talks about the MIPS 5-stage architecture starting in section 4.6.

Use Quartus or write your own VHDL to add the pipeline registers to your single-cycle processor from Project-A to make it a 5-stage pipeline.

4) Test the 5-stage processor without hazard detection/forwarding. Make sure your processor is working properly by testing instructions without data dependencies nor branches. Provide a description of a few test cases (at least one test case for each of the six instructions) and clear screenshots depicting your functioning test cases.

5) A simple 32-bit comparator has also been provided in projectB.zip. You will use this component to implement branch resolution in the ID stage, as opposed to using the ALU to resolve branches in the EX stage. The comparator file is labeled *branch_comparator.vhd* and it represents the "=" component in the diagram above. It takes RS data and RT data as inputs and outputs a single bit-flag which is '1' if RS data and RT data are equal, '0' otherwise. Implement ID stage branch resolution and provide a legible simulation-screenshot of a taken-branch instruction correctly executing.

6) You will now implement your own *hazard detection* and *forwarding* units.

a) The *forwarding* unit is used to pass dependent data between stages. For example:
ADD $r1, $2, $3;
ADD $r4, $1, $2;
This is an example of an ALU producer to ALU consumer data dependency. In this case the MEM stage will need to forward the value of $r1 to the EX stage for consumption. Implement a forwarding unit (using VHDL) to support the following data dependent cases. Give simulation screenshots of correct forwarding for each case:

i) ALU producer to ALU consumer at distance 1 (e.g. ADD $1, $2, $3; ADD $4, $1, $2)
ii) ALU producer to ALU consumer at distance 2 (e.g. ADD $1, $2, $3; <INST>; ADD $4, $1, $2)
iii) Load producer to ALU consumer distance 2 (e.g. LW $1, 0($10); <INST>; ADD $5, $1, $r2)
iv) ALU producer to BEQ consumer at distance 2 (e.g. ADD $1, $2, $3; <INST>; BEQ $1, $2, label)

Notes:

Your forwarding unit should prioritize i) over ii) if both cases are true at the same time.

Make sure you don't forward values that are trying to write to $0, e.g., ADD $0, $1, $3. The result of this addition should not be forwarded because $0 should always hold the constant 0.

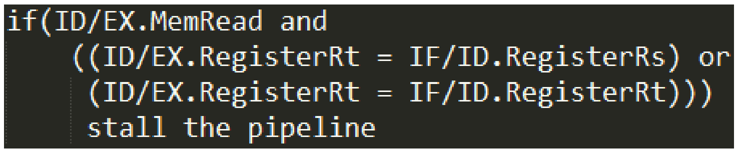
b) The *hazard detection* unit handles stalls and flushes in the pipeline. Implement a hazard detection unit (using VHDL) to support the following data dependent cases. Give simulation screenshots of correct forwarding for each case:

i) The jump instruction must trigger a flush for the instruction directly following. The hazard detection unit should check if a jump instruction is in the ID stage and set the flush signal to the IF/ID register if so.

ii) The BEQ instruction must trigger a flush for the instruction directly following **only if the branch is taken**. The hazard detection unit should check if a taken BEQ instruction is in the ID stage and set the flush signal to the IF/ID register if so.

iii) To assist our forwarding unit the hazard detection unit must create stalls if the pipeline sees certain sequences of instructions:

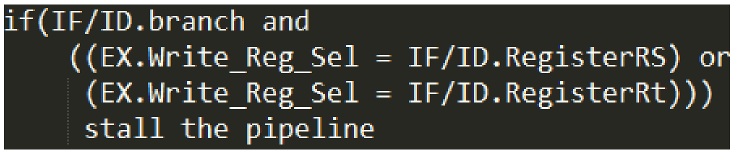
(1) Load producer to ALU consumer at distance 1 needs to introduce a stall of 1 cycle to load the data dependency for the ALU consumer. If the condition



```
if(ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
     stall the pipeline
```

is satisfied then we need to introduce a stall. “Stall the pipeline” in this case means stall PC, stall IF/ID, and flush ID/EX. *Note: The pc_reg.vhd has been updated in the “Components” folder to include a stall signal for this project.*

(2) ALU producer to BEQ consumer at distance 1 needs to introduce a stall of 1 cycle to compute the data dependency for the BEQ consumer. If the condition



```
if(IF/ID.branch and
    ((EX.Write_Reg_Sel = IF/ID.RegisterRS) or
     (EX.Write_Reg_Sel = IF/ID.RegisterRt)))
     stall the pipeline
```

is satisfied then we need to introduce a stall. “Stall the pipeline” in this case means stall PC, stall IF/ID, and flush ID/EX. *Note: The pc_reg.vhd has been updated in the “Components” folder to include a stall signal for this project.*

You may notice that for these two cases we turn a data dependency at distance 1, to distance 2. Therefore, your *forwarding* unit doesn’t need to support distance 1 for Load producer to ALU consumer and distance 1 ALU producer to BEQ consumer.

c) Connect your forwarding and hazard detection units to your pipelined processor and provide a simulation screenshot showing that your pipeline correctly executes the given test program. Be sure to include the result of the program’s execution.

SUBMISSION:

- Report the names of your group members in your report.
- Create a zip file *Project-B-submit.zip*, including the completed code and screenshots from the lab.
- A lab report that answers all highlighted sections from this document and screenshots.
- The file names in your submission should be self-explanatory.
- Submit the zip on Canvas under Project-B.