ECEN 651: Microprogrammed Control of Digital Systems Department of Electrical and Computer Engineering Texas A&M University

Prof. Mi Lu
TA: Younggyun Cho

Laboratory Exercise #6

Pipelined MIPS Processor

Objective

The objective of lab this week is to pipeline the single-cycle MIPS processor built in the last lab. The pre-laboratory exercise introduced the various challenges associated with creating a five stage MIPS pipeline. We will expand on those concepts by implement data forwarding in our pipeline. Control hazards this week will be resolved by simply stalling portions of the pipeline. Next week we will improve the design with speculative execution.

Background

Figure 1 shows the block diagram of the MIPS processor with data forwarding. This diagram differs from that shown in the pre-laboratory with that addition of a forwarding unit, a hazard detection unit, and addi-tional bypass logic (i.e. multiplexers). In lecture, it was shown that Read After Write (RAW) data hazards involving R-type instructions can be completely resolved without stalls in the classic five-stage processor pipeline. In Figure 1, the ALU source multiplexers have been expanded to support four inputs, two of which are in the datapath from last lab. The two additional inputs are from the Memory and Write-back stages ahead of the Execute stage. The idea is that the result for an R-type instruction, in the case of a RAW with another R-type instruction, already exists in the pipeline but has not yet been committed to the register file. The job of the forwarding unit is to detect resolvable data hazards and produce the control signals for the bypass logic appropriately. When a load instruction produces a value that is consumed by the instruction

immediately following in the pipeline, a stall is unavoidable. This is due to the fact that the value from the load is produced in the Memory stage rather than the Execute stage. Thus, a hazard detection unit is required to identify and resolve data hazards that cannot be completely fixed with bypass logic. In the case of a load as a producer in a RAW hazard, the hazard unit stalls the fetch and decode stages for one clock cycle. The aforementioned technique for resolving data hazards is commonly referred to as pipeline interlocking. The Microprocessor without Interlocked Pipeline Stages (MIPS) was design to minimize interlocking. Two additional multiplexers are found in the diagram of our MIPS processor for forwarding register values to the data memory. In the case of a RAW between a store following a load, the bypass logic is able to completely avoid a stall. Naturally, this is a result of the fact that the store does not need the register value until the beginning of the Memory stage.

In addition to resolving data hazards, the hazard detection unit is responsible for orchestrating pipeline stalls due to control hazards (i.e. jumps and branches). Control hazards due to branching come from the fact that the processor fetches in the first stage, while it does not resolve the branch until the third stage (second or fourth in other designs). If the branch logic is moved to the second (Decode) stage, the penalty for a branch is only one clock cycle. However, bypass logic must also be created for the conditional evaluation, which is why Figure 1 shows that the branch logic in the third (Execute) stage.

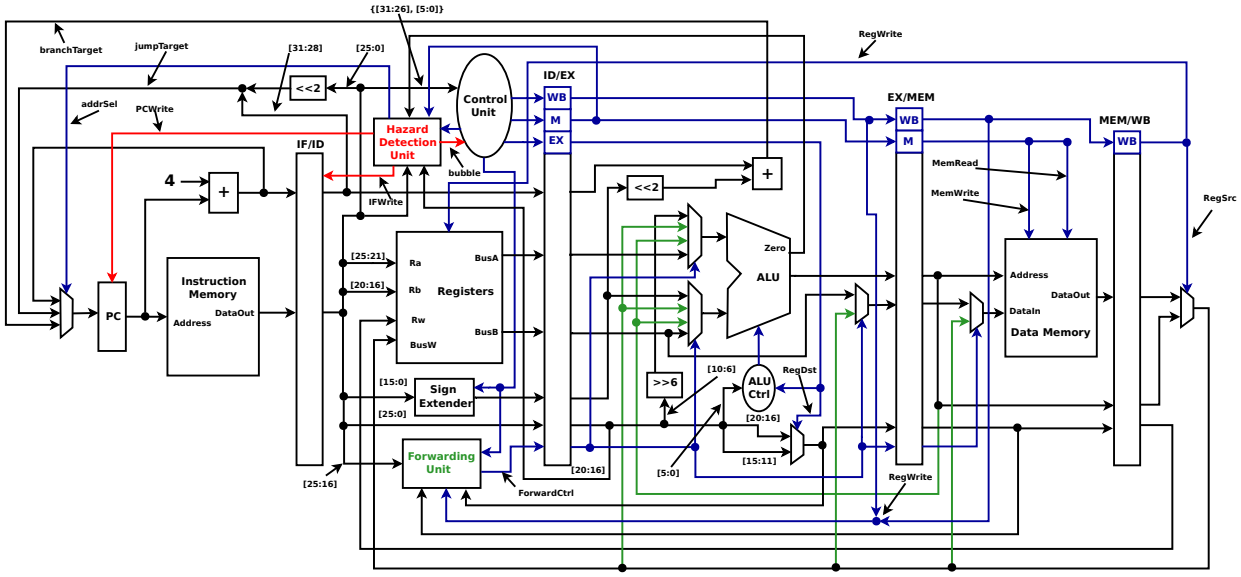

Figure 1: MIPS Block Diagram

Procedure

1. **Design the Forwarding Unit.** The forwarding unit shown in Figure 1 sits in the Decode stage of the pipeline and generates the forwarding control signals, which propagate through the pipeline with the appropriate data. The following bullets will help you understand how the forwarding unit must operate.

    - In order to detect a RAW data hazard, the forwarding unit needs to know the values of Rs and Rt of the instruction currently in the Decode stage along with the values of Rw (Rd or Rt depending on the instruction) for the previous (in time that is) instructions, which lie ahead of the current instruction. You will notice that in Figure 1 only the Rw values from the Execute and Memory stage are directed to the forwarding unit. The assumption here is that if the producing instruction is in the Write-back stage, the actual latching of data into the register file takes place slightly before the data is read from the register file in the Decode stage. In our design, we will ensure this happens by modifying the register file slightly such that data is written on the positive edge of the clock.
    - The forwarding unit must prioritize that the most recent result in the pipeline. This means that if the two previous instructions modify a particular register that is used in the current instruction, the bypass logic must forward the result from the stage closest to the Execute (i.e. from the Mem stage).
    - Writes to register zero must be ignored. The programmer must be allowed to use the zero register as the destination register in order to create NOPs. Thus, the forwarding logic must not attempt to forward the result of an instruction that is destine for the zero register.
    - In addition to the register addresses, the forwarding unit needs to know if the corresponding Rw registers are indeed being changed. Therefore, we must also route in the RegWrite signals from the Execute and Memory stage into the forwarding unit.
    - In the case of an immediate type instruction or a shift instruction that makes use of the shamt field, we need not forward any data. Thus, the *UseShamt* and *UseImmed* signals from the control unit must be routed into the forwarding unit. In either of those cases, we simply select the appropriate field from the current instruction.
    - The outputs of the forwarding unit include the AluOpCtrlA and AluOpCtrlB for controlling the ALU source operand multiplexers for A and B respectively. Also part of the output port list are the DataMemForwardingCtrl_EX and DataMemForwardingCtrl_MEM for controlling the Data Memory input multiplexers in the Execute and Memory stages respectively. One thing to note for Data memory forwarding is that the multiplexing logic must be broken up such that the result in the Write-back stage is forwarded before it is committed to the register file and "drained" from the pipeline. Likewise, we want to forward the result for the instruction just ahead of the consuming store in the Write-back stage so that we can allow a store to immediately follow a load.

(a) Draw a block diagram which reflects the above logic. You may assume equality blocks are available for use. The symbol for such a device is a square with an equal sign in it. Your block diagram should be detailed enough that, from it, someone could create the necessary boolean equations for the forwarding unit. Also take note of the helpful hints below.

(b) Write a Verilog module to implement the Forwarding Unit discussed above. Use the following module interface:

```
module ForwardingUnit(UseShamt, UseImmed, ID_Rs, ID_Rt, EX_Rw, MEM_Rw,
EX_RegWrite, MEM_RegWrite, AluOpCtrlA, AluOpCtrlB, DataMemForwardCtrl_EX,
DataMemForwardCtrl_MEM);

  input UseShamt, UseImmed;
  input [4:0] ID_Rs, ID_Rt, EX_Rw, MEM_Rw;
  input EX_RegWrite, MEM_RegWrite;
  output reg [1:0] AluOpCtrlA, AluOpCtrlB;
  output reg DataMemForwardCtrl_EX, DataMemForwardCtrl_MEM;
```

Helpful Hints:

- The forwarding unit does not have to worry about whether or not the producing instruction is a load. The hazard unit that you build in the next section will detect this and resolve it.
- Your forwarding unit should be made up of purely combinational logic. Notice that the clock and reset signals are *not* routed into this module. Unsure your Verilog code does not create latches.
- The *UseShamt* and *UseImmed* signals are essentially the select lines of the ALU source multiplexers from the single cycle design, which were generated in the ALU Control block and the Control unit respectively. Since the ALU control module is in the Execute stage, logic for generating the UseShamt field must be moved into the main control unit. This of course implies that the Function field of the current instruction must also be routed into the Control Unit.
- You may want to create a test bench for your logic. If you are working on this project with a friend, have he or she create your test bench and you create his or her test bench. This will help flush out any flaws in your logic. Please note that all forwarding units should behave in exactly the same manner.

(c) Synthesize your design using the Device properties provided in Lab 3. Fix **all** warnings and errors and ensure the compiled hardware contains no latches or flip-flops. Provide the summary results of the synthesis process in your lab write-up.

2. **Design the Hazard Detection Unit.** When a load is the producing instruction in a RAW data hazard and the consuming instruction immediately follows the load, the pipeline must be stalled. Likewise, when a control hazard is detected, the pipeline must be stalled. The hazard unit in the Decode stage takes care of the pipeline stalling. The following bullets will help you understand how the hazard unit should operate.

- In order to detect a RAW hazard that requires a pipeline stall, the Rt field from the Execute stage along with Rs Rt, UseShamt, and UseImmed from the Decode stage must be routed into the hazard detection unit. Likewise, we must input the ReadMem signal from the Execute stage to allow our logic to determine if the producing instruction is a load.
- Stall are executed with the appropriate use of the IF_write, PC_write, and bubble output ports. When HIGH, the IF_write causes the IF/ID pipeline registers to latch on the falling edge of the clock. When the IF_write signal is LOW, the IF/ID pipeline registers will simply hold the previously stored values. The PC_write signal provides the same functionality for the PC register. The bubble signal causes the control signals of the ID/EX pipeline registers to reset (i.e. store 0's), which essentially inserts NOPs or bubbles in the pipeline starting at the Decode stage.
- During normal execution, the PC_write and IF_write signals are HIGH, while the bubble signal is LOW.
- When a RAW hazard requiring a pipeline stall is detected, the hazard unit must cause IF_write and PC_write to go LOW to stall the IF and ID stage and set bubble to HIGH to release NOPs into the remainder of the pipeline.
- To detect the presence of a jump or a branch, we must route the Jump and Branch signals from the Control unit into the Hazard Detection Unit. Likewise, we must output an address select signal, addrSel, to control the PC source multiplexer. During normal execution, addrSel should be set to select the PC+4 bus.
- When a Jump is detected, the hazard unit must stall the ID stage, while fetching a new instruction starting at the jumpTarget. This has the effect of squashing the PC+4 instruction that was fetched immediately after fetching the jump.
- When a branch is detected, the IF stage must be stalled for one clock cycle, while the branch instruction is allowed to continue down the pipeline. Since the branch is resolved in the Execute stage, in the following clock cycle, the hazard detection logic can use the ALUZero signal from the Execute stage to determine what to do next. If the branch was Not Taken, then the PC+4 instruction in the IF stage is correct and should then be allowed to move into the Decode stage. Then execution resumes as normal. If the branch was Taken, the fetch must be redone starting at the branchTarget. As with the jump, this operation squashes the PC+4 instruction. Once this happens, the pipeline will return to normal execution.

(a) Create a block diagram for the logic required to detect a data hazard that requires stalling. Assume your logic creates an internal signal called *LdHazard* which is HIGH when the hazard exists. Use the above description as a guide.

(b) Create the state diagram that describes the workings of the Hazard unit. Assume that the inputs to this FSM are Jump, LdHazard, Branch, and ALUZero and the outputs are PC_write, IF_write, bubble, and addrSel.

Hints:

- The FSM should be a Mealy machine such that outputs are defined for each of the state transitions (i.e. the output are dependent on inputs as well as state.
- The FSM should have one state for normal operation (NoHazard_state). You will also need one state for jump (Jump_state) and two for branch (Branch0_state and Branch1_state).
- Provide a block diagram for the FSM.

(c) Describe the Hazard Detection Unit in Verilog using the following module interface:

```
module HazardUnit(IF_write, PC_write, bubble, addrSel, Jump, Branch, ALUZero,
memReadEX, currRs, currRt, prevRt, UseShamt, UseImmed, Clk, Rst);

output reg IF_write, PC_write, bubble;
output reg [1:0] addrSel;
input Jump, Branch, ALUZero, memReadEX, Clk, Rst;
input UseShamt, UseImmed;
input [4:0] currRs, currRt, prevRt;
```

Helpful Hints:

- Implement the *LdHazard* generation logic and the FSM logic in the same modules but in separate section of code.
- Use proper design techniques when describing the FSM. This involves separating out the sequential and combinational logic as would be reflected in your block diagram of the FSM. Again, keep everything in contained in the same module.
- The combinational portion of the FSM can be implemented easily with a case based on the current state.
- As with the Forwarding Unit, it may be a good idea to create a test bench for your Hazard Detection Unit.

(d) Synthesize your design using the same Device properties from Lab 3. Fix **all** warnings and errors. Provide the summary results of the synthesis process in your lab write-up.

3. **Make slight modifications to the Register file, Control Unit, and ALU Control Module.** In this section of the procedure, you will modify existing components to fit within our pipelined microarchitecture.

(a) The Register file must be modified to latch data on the positive edge of the clock rather than the negative edge. As mentioned earlier this is essentially a method of automatic forwarding that allows us to write and read from the same registers in the same clock cycle. Although this will make the timing requirements on the register file tighter, it is already fast since it is small and will not be part of the critical path.

(b) Modify the Control unit to include the UseShamt signal. Essentially this signal should already be in you datapath, but for review, it must be HIGH when an R-type instruction with a Function code specifying one of the three supported shift operations (SLL, SRL, and SRA) is detected. This will mean that you must input the Function field into the Control Unit.

(c) For the ALU Control Module, you must remove the UseShamt logic as it is now generated in the main control unit.

(d) Re-synthesize the above modules and ensure once again that no warnings or errors exist and that the ALU control and Control Unit do not have latches or flip-flops.

4. **Design the remainder of the MIPS pipelined datapath.** Up to this point, we have created all the major subcomponents of the MIPS pipelined processor. It is now time to describe the remainder of the datapath in Verilog using Figure 1 as a guide.

   (a) Describe the pipelined MIPS datapath in Verilog using the following module interface:

   ```
   module PipelinedProc(CLK, Reset_L, startPC, dMemOut);
   ```

   Helpful Hints:

   - Download the updated instruction memory Verilog file on the laboratory website and incorporate it into your ISE project.
   - For 3:1 and 4:1 multiplexers, use *always* blocks with the *case* statement. Do not forget to include a default case to avoid creating extraneous latches.
   - The **PC** should be sensitive to the negative edge of the clock and change state only when the *PC_write* signal is active. Similarly, the Reset signal is active low and should synchronously reset the **PC**.
   - Keep your signal names as consistent with Figure 1 as possible and try to minimize the number of intermediate signals.
   - The pipelined version of the MIPS processor holds state between each stage in the pipeline. For each stage, choose the name of the pipelined register wisely and try to stick to a convention. For example, the ALU result is stored between the Execute and Memory stage and also between the Memory stage and the Write-Back stage. Thus, appropriate register names would be *ID_MEM_ALUResult* and *MEM_WB_ALUResult*.
   - Similarly, the control signals are registered between pipeline stages such that they flow with the corresponding data. It may be easier to group the control signals according to where they are used. For example, the *RegSrc* and *RegWrite* signals are used in the Write-Back stage so an appropriate name for the first set of pipelined registers carrying control signals for the Write-Back stage might be *ID_EX_ctrl_WB*. Note that this register is 2-bits wide.
   - The pipeline registers holding control signals must have a synchronous reset signal. The registers holding data or address signals do not need to be reset. Additionally, the control portion of ID/EX pipeline registers should also reset when *bubble* is active.

- It might be helpful to periodically synthesize the design while debugging since the synthesis tool is much more strict and throws many more warnings than the simulation tool does. Thus, it has the potential of catching more bugs.

(b) Use the test bench provided on the laboratory website to test the functionality of your overall design. Paste the output of the simulator into your lab write-up.
Note: The test bench for the pipelined processor is different than the test bench for the single-cycle processor!

(c) Once your design passes all of the tests provided by the test bench, synthesize your design and ensure no errors or warnings exist. Provide the summary results of the synthesis process in your lab write-up.

1 Deliverables

1. Submit a lab report that captures your efforts in lab.

2. Include all Verilog source files with comments.

   Note: Code submitted without adequate comments will not be graded!

3. Answer the following review questions:

   (a) Instead of stalling the pipeline while waiting for a branch to be evaluated, we could simply continue to fetch from PC+4. What name is commonly given to this technique? What changes would have to be made to our existing design to accommodate this improvement? Be sure to include modifications to the Hazard Detection Unit and pipeline registers.

   (b) What clock rate did the synthesis process estimate your overall design would run at? Look through the synthesis report and locate the section where the design timing is estimated. From that, determine which components are in the critical path. What changes could you make to the design to improve the clock rate without adding additional pipeline stages?

   (c) Determine the CPI for each of the three programs executing on your processor. Compare those results to the CPI of the single-cycle processor. Also, provide the average CPI of all three programs.

   (d) Given the clock rate and average CPI, compute the MIPS (Million Instructions Per Second) rating for both versions of processor. Which one is faster and why?