Objective

The objective of lab this week is to extend the pipelined MIPS processor developed in the last lab to support subroutine calls by implement the jump-and-link and jump-register instructions, i.e. jal and jr. In addition, simple static branch prediction will be added to the microarchitecture. At the end of lab, you will have the opportunity to compare the CPI of the processor from last lab with that of the one in this lab in order to gain an understanding of how simple branch prediction can improve performance.

Background

Thus far, our processor is only able to execute a subset of the core MIPS instructions. That subset is capable of performing a wide variety of useful computations; however, in an effort to improve the usefulness of our processor, we would like it to support subroutine calls. Remember that subroutine calls improve code density, while reducing overall code complexity. As discussed in lecture, jal and jr in MIPS are used for jumping to a subroutine and returning from a subroutine, respectively. Both instructions are of a form of the jump instruction already supported by our processor so much of the necessary hardware already exists in the datapath and control logic. The jal instruction, simply saves the PC+4 address into the $ra register prior to jumping, while the jr instruction jumps to the address contained within the register specified by the rs field.

As discussed previously, branches reduce the overall efficiency of a processor by introducing stalls into the pipeline. Recall that branches in our pipeline are resolved in the Execute stage, causing a 2-stall penalty
if the branch is *Taken* and a 1-stall penalty if the branch is *Not Taken*. At this point, our method of handling branches is to stall the pipeline as soon as a branch is decoded; however, we already have mechanisms in place for flushing instructions in both the fetch and decode stages. Therefore, in this lab, we will modify our design such that when a branch is decoded, it will continue to speculatively fetch instructions in the *Not Taken* path. It turns out, the cost of this modification is very low in terms of hardware resources, while the reduction in stalls can be quite significant. In the new design, the penalty for a branch *Not Taken* is reduced to 0 stalls.

![Figure 1: MIPS Block Diagram with Jal and Jr Support](image)

**Procedure**

1. **Implement the jal and jr instructions.** Figure 1 shows highlighted modifications that must be made to our datapath to support *jal* and *jr*. The following bulleted list explains them in depth.
   - Among the aforementioned modifications is a multiplexer after the ALU to allow the PC+4 of a *jal* instruction to be written into the register file. Likewise, the *Rw* multiplexer in the Execute
stage has been extended to include the value ‘31’ which corresponds to the $ra register in MIPS. For the \texttt{jr} instruction, a multiplexer has been added to allow selection between the jump target address and the Ra bus, which corresponds to the rs field of the current instruction.

- One thing to note is that no forwarding paths exists for the \texttt{jr} instruction. This is not a problem, however, since the \texttt{jr} instruction would typically be used with a \texttt{jal} instruction. In this situation, the \texttt{jal} instruction would require one stall to start fetching at the subroutine address, and at least one instruction other than the \texttt{jr} instruction would be expected within the subroutine, which provides enough time for the return address to be latched into the register file. That being said, it is still necessary that the Hazard Unit stall in the atypical case. Consequently, the \texttt{Rw} and \texttt{RegWrite} signals being fed into the Forwarding Unit must also be sent to the Hazard Unit. This is reflected in Figure 1 with the addition of the concatenation operator at the bottom. Obviously, these signals will have to be added to the port list for the Hazard Unit. Likewise, additional logic to detect the hazard involving a \texttt{jr} instruction must be added. This will also require the use of a \texttt{Jr} signal from the Control Unit, which indicates when a \texttt{jr} instruction is being decoded.

- Modifications also necessary but not shown in the diagram involve changes to the Control Unit. The \texttt{jr} instruction is actually an R-type instruction, which adds a slight amount of complication to our very simple decoder. Logic within the control unit must be created to use the function code within the instruction to detect this special R-type instruction and essentially deactivate the register write enable signal and activate the jump signal. Likewise, a \texttt{Jr} signal to control the new multiplexer and signal the Hazard Unit must be added. The \texttt{jal} instruction, on the other hand, is an immediate type instruction and can be dealt with in a similar fashion as the jump instruction. However, a \texttt{Jal} signal must be added to the port list of the Control Unit for the additional datapath multiplexer, and the \texttt{RegDst} signal must be expanded to 2-bits wide to allow for the selection of ‘31’ as a register destination.

(a) Determine the opcode and function code for the new instructions discussed above. Modify the Control unit to support them and re-synthesize to ensure no latches are described in your design. The updated module interface should be as follows:

\begin{verbatim}
module PipelinedControl (Jump, Branch, RegDst, UseImmed, UseShamt, RegSrc, Jal, RegWrite, MemRead, MemWrite, SignExtend, ALUOp, JumpSel, OpCode, FuncCode);

    input [5:0] FuncCode; // [5:0] from instruction
    output Jump; // active if instruction is of jump type
    output Branch; // active if instruction is of branch type
    output [1:0] RegDst; // select line for RW MUX
    output UseImmed; // active if using immediate field
    output UseShamt; // active if using shamt field
    output RegSrc; // select line for BusW MUX
    output Jal; // indicates a Jump and link
    output RegWrite; // active if Writing into register file
    output MemRead; // active if reading from memory
\end{verbatim}
(b) For the modifications to the Hazard Unit, draw the new state diagram and provide a block diagram for the additional logic required.

(c) Modify the Hazard Unit as discussed above and re-synthesize to ensure no errors or warnings exist. The updated module interface should be as follows:

```verilog
module HazardUnit (IF_write, PC_write, bubble, addrSel, Jump, JR, Branch, ALUZero, memReadEX, ID_Rs, ID_Rt, EX_Rw, MEM_Rw, EX_RegWrite, MEM_RegWrite, UseShamt, UseImmed, Clk, Rst);
output reg IF_write, PC_write, bubble; // for stalling and flushing IF and ID stages
output reg [1:0] addrSel; // for selecting next PC address
input Jump, JR, Branch, ALUZero, memReadEX, Clk, Rst;
input UseShamt, UseImmed;
input [4:0] ID_Rs, ID_Rt, EX_Rw, MEM_Rw;
input EX_RegWrite, MEM_RegWrite;
```

(d) Make the appropriate modifications to the datapath and re-synthesize the entire design. Ensure no new warnings appear and that there are no errors.

(e) Download the updated instruction memory Verilog file from the laboratory website along with the latest test bench which includes tests for the recently added instructions. Simulate the provided test bench and ensure your design passes all of the tests.

(f) Copy the console output from the simulation and the final synthesis report into your lab write-up. Be sure to take note of the final clock rate as there will be questions about it at the end of lab.

2. **Implement Static Branch Prediction.** Figure 2 shows the modifications to the current MIPS processor that must be made in order to support Static Branch prediction. The following bulleted list explains these modifications in detail:

- The changes to the actual datapath in our design are very minor. Prior to Static Branch Prediction, the Branch signal was fed into the Hazard Unit during decode so that the pipeline could be stalled appropriately. However, the improved design does not stall the pipeline when branch is detected but rather flushes the pipeline when a branch miss prediction is detected. As a result, we must let the Branch signal propagate with the branch instruction to the Execute stage. It is there were the Branch signal is fed into the Hazard Unit.

- The majority of the changes you will make in this section of lab will be in the Hazard Unit. Interestingly enough, our improved policy of handling branches simplifies the Hazard Unit! Since we would like the processor to simply continue fetching down the Not Taken path, our Hazard
Figure 2: MIPS Block Diagram with Static Branch Prediction
Unit does not even need to know a branch is being processed until it makes to the Execute stage, where it is resolved. When the Branch is resolved and found to be Not Taken, no action is needed. However, when a Branch is found to be Taken (i.e. a miss prediction has been detected since the Not Taken path was assumed), the instructions in the Fetch and Decode stages are incorrect and must be flushed. Note that the flushing must take top priority over all other hazards detected, and the processor must begin fetching instructions from the branch target.

(a) Redraw the updated state diagram of the Hazard Unit.
(b) Make the above modifications to both the Hazard Unit and the datapath. The Hazard Unit’s module interface should not change.
(c) Re-simulate with the latest test bench and instruction memory and ensure all of the tests pass. Include the latest console output of the simulation in your lab write-up.
(d) Re-synthesize the entire design and ensure no errors or new warnings appear. Include the synthesis report 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) Compare the synthesis report from last lab with the one generated during Part 1 of this lab. Did the clock rate of your design change at all? Why or why not? Please explain the results! Can you think of any improvements that could be made to the current design?
   (b) Recompute the CPI of your processor executing Program 1. Did it change? Why or why not?
   (c) Recompute the average CPI of your processor executing all programs including the new program for testing jal and jr. Compare the new CPI with that of last lab. Explain the changes.
   (d) Compare the synthesis report from the last part of lab to that of the first part of lab. Was there a cost in terms of resource utilization for the improved design? If so, what was it? Explain your results.
   (e) Examine the loop in Program 1. What characteristic does it have that makes static branch prediction worth while? Describe what would happen to the CPI of our processor executing Program 1 if the number of loop iterations were to increase.