Objective

The laboratory assignment for this week has two parts, each with slightly different objectives. For the first part of lab, we will attempt to improve the CPI of our processor from last lab by adding Dynamic Branch Prediction. The main objective of this part of lab is to improve our understanding of how simple Dynamic branch prediction works, and how it can improve performance. For the second part of lab, we will add support for exception handling within our MIPS processor. The objective of this part of lab is to provide an understanding of the procedures a processor must go through in order to gracefully recover from an internal fault.

Background

Last week, we found that we could improve the performance of a pipelined processor by simply assuming that a branch is going to evaluate as Not Taken and flushing the pipeline in the case that a branch is actually evaluated as Taken. This simple method of speculative execution reduces the number of pipeline stalls to zero in the case of a Not Taken branch. However, if a branch is evaluated as Taken, the penalty for that branch is still two stalls. While this simple technique works well for “Top-of-loop” branching structures (i.e. the branch is at the top of the loop and evaluates to Taken only when exiting the loop), it does not work well for “Bottom-of-loop” branching structures. To reduce the number of stalls for a Taken branch, we can
employ a simple dynamic branch prediction scheme which makes a prediction in the Fetch stage, causing the pipeline to refetch from the branch target during the Decode stage in the case of a *Taken* prediction. If we assume an ideal branch predictor (i.e. one that always predicts correctly), we can reduce the stalls for a *Taken* branch from two to only one. Obviously, our branch predictor is not ideal; however, we will use a 2-bit history predictor, which has been shown to provide 83.4% to 97.5% accuracy across a wide range of workloads [1].

So far, our processor follows a “Garbage In, Garbage Out” scheme, in which it has no way of knowing if it is doing something wrong nor does it have a mechanism of informing the user, in an effort to correct the problem. Therefore, we would like to implement some level of exception handling in which instructions can generate exceptions (i.e. detect a problem), and the processor can respond to those exceptions in a deterministic manner (i.e. attempt to correct the problem). Exceptions can be both internal and external. An example of an internal exception would be an arithmetic overflow, while an example of an external exception would be an interrupt. In this lab, we will focus on only two types of internal exceptions (Arithmetic overflow and Instruction Fetch Error) and will not deal with external exceptions. Although, once the hardware can handle...
one exception, handling additional exceptions is not much more challenging.

**Procedure**

1. **Implement Dynamic Branch Prediction.** Figure 1 shows the highlighted modifications that must be made to our datapath to support dynamic branch prediction. The following bulleted list explains them in depth.

   - The most obvious change to our current processor design is the inclusion of a Branch Predictor block. This block assumes the instruction being fetched is a branch and simply provides a best guess as to how it will be resolved based on previous history. The output is a single bit value such that ‘1’ represents *Taken*. For a prediction, the branch predictor requires the lower order bits of \( PC+4 \) of the current instruction being fetched (excluding of course the two least significant bits). Updates to the predictor are made when the branches are resolved (i.e. in the Execute stage). To perform an update, the branch predictor needs the \( PC+4 \) of the instruction in the Execute stage (as it might be branch), the branch signal from the Execute stage, and finally ALUZero for determining the output of the branch.

   - Clearly, the Hazard Detection Unit must change. The registered output of the branch predictor is fed into the Hazard Detection Unit, along with the *Branch* signal from the Control Unit. If *Branch* is active and the predictor output is ‘1’, the Hazard Unit must flush the instruction in the Fetch stage, re-fetching instructions starting at the correct target address. To support this operation, the multiplexer feeding into the *PC* must be expanded such that the branch target address in the Decode stage along with the re-fetch address in the Execute stage are inputs for speculative and non-speculated re-fetching respectively. Notice that the calculation of the branch target has been moved into the Decode stage as well.

   - Additional logic within the execute stage must be added in order to detect when a miss prediction was made in the Decode stage. In the event of a miss prediction, the Hazard Detection Unit must orchestrate a ‘flush’ in which the instruction in Decode stage is turned into a *nop* and the pipeline begins re-fetching at the correct target address. To detect a miss prediction, we must know what the prediction was and what the final outcome of the branch is. You will notice a **Flush** block which handles this detection within the latest revision of our processor block diagram (Figure 1).

   - Depending on what prediction was made, the correct re-fetch address will be either the branch target address calculated in the Decode stage or \( PC+4 \) of the branch in the Execute stage. Thus, you will notice an additional multiplexer in the Execute stage which handles this condition. Similarly, the ID/EX pipeline registers must hold both target addresses. We cannot simply move the re-fetch address multiplexer into the Decode stage and only register the re-fetch address because we need \( PC+4 \) for the *jal*. 
• To further reduce the complexity of the Hazard Detection Unit and aide in flushing the pipeline, Figure 1 shows a \textit{ID\_EX\_Flush} signal, which acts like the bubble signal for the EX/MEM pipeline registers (i.e. when active it resets the control registers). With this additional control signal, the Hazard Detection Unit can be reduced to a purely combinational logic block, meaning it should hold no state.

• To allow for “Bottom-of-loop” branching structures, our processor must support the \texttt{bne} instruction. To do so, we must expand our \texttt{Branch} signal to 2-bits wide such that the lsb is the existing \texttt{Branch} signal and the msb determines whether a \textit{Taken} is when AluZero is HIGH or LOW.

![Simple Branch Prediction with a BHT](image.png)

Figure 2: Simple Branch Prediction with a BHT

(a) Modify the Control Unit to support \texttt{bne} as discussed above. The updated module interface should be as follows:

```verilog
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
/* Branch[1] active if bne (i.e. Outcome is Taken if AluZero is LOW) */
output [1:0] Branch;
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
```
Laboratory Exercise #8

output MemRead; // active if reading from memory
output MemWrite; // active if writing into memory
output SignExtend; // active if sign extension is needed
output JumpSel; // select line for jump target address MUX
output [3:0] ALUOp; // control ALU operation

(b) For dynamic branch prediction, we will utilize a 2-bit counter as shown in the textbook. Please provide the state diagram for this scheme in your write-up.

(c) Describe the branch prediction module in Verilog. Figure 2 provides a block diagram of the branch predictor hardware that you will design. The Branch History Table (BHT) is nothing more than a memory array used to hold the 2-bit state for the various branches. The UpdateAddr and PredictAddr serve as address ports into the BHT, while the Branch signal is the write enable for the update operation. Please note that clock is not shown in the diagram but is implied. The branch predictor's module interface should be as follows:

module BranchPredictor (PredictAddr, Branch_EX, UpdateAddr, Outcome, Prediction, Clk);
input [4:0] PredictAddr; // PC+4 from Fetch stage for prediction
input [4:0] UpdateAddr; // PC+4 from EX stage (i.e. PC+4 for branch)
input Branch_EX; // branch in Ex stage
input Outcome; // active for Taken, otherwise Not Taken
input Clk;
output Prediction;

(d) Synthesize the branch predictor module and ensure no errors or warnings appear. Examine the synthesis report and explain, in your lab write-up, what FPGA hardware the synthesis engine used to construct your branch predictor. You do not have to provide the synthesis report, however.

(e) 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:

module HazardUnit(IF_write, PC_write, bubble, IF_ID,Flush, addrSel, Jump, JR, BranchPredict, Flush, Branch_ID, memReadEX, ID_Rs, ID_Rt, EX_Rw, MEM_Rw, EX_RegWrite, MEM_RegWrite, UseShamt, UseImmed);

output reg IF_write, PC_write, bubble; // controls IF and ID stages
output reg IF_ID,Flush; // flushes contents of control portion of the IF/ID regs
output reg [1:0] addrSel; // selects address to write into PC reg
input Jump, JR, BranchPredict, Branch_ID, memReadEX;
input Flush; // from Flush detection logic within the Execute stage
input UseShamt, UseImmed;
input [4:0] ID_Rs, ID_Rt, EX_Rw, MEM_Rw;
input EX_RegWrite, MEM_RegWrite;

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

Do not forget to add support for bne within the datapath.
Laboratory Exercise #8

(g) Simulate your design with the test bench used in the last lab and ensure your design passes all of the tests.

(h) 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.

Figure 3: MIPS Block Diagram With Exception Handling

2. **Implement Exception handling in the current MIPS processor.** The datapath must be modified to detect and handle exceptions appropriately. For this lab, we will handle arithmetic exceptions from the ALU and ‘unrecognized instruction’ exceptions from the control logic. When an exception occurs, the following steps must be performed in order to remain MIPS compatible:

   - The PC address of the offending instruction must be saved in the Exception Program Counter (EPC).
   - The reason for the exception must be stored in the Cause register. More detail on this will be provided shortly.
   - The pipeline must be flushed (including the instruction that caused the exception) and re-fetching must begin at a specific location specified by the MIPS architecture manual (usually 0x80000080).
It is here where the Operating System (OS) has exception handling code, which will make use of the Cause register to determine how to handle the exception and the EPC to identify the offending instruction and possibly return to it.

In addition to the above procedures, MIPS compatible machines must modify the contents of a Status register in order to mask interrupts. Since interrupts are out of the score of this lab, we will not concern ourselves with the Status register. Figure 3 shows the highlighted modifications that must be made to our datapath to support Exceptions. The following bulleted list explains them in depth.

- The most notable modifications to our current data path include the addition of the EPC and Cause registers, along with the inclusion of the Exception Detection logic. The EPC and Cause registers have already been discussed above, and the Exception Detection logic is contains an OR gate with three inputs. Two of those inputs come from the Control Unit and ALU Control module and indicate an invalid OPCode or an unsupported R-type instruction respectively. The third input comes from the ALU and indicates an arithmetic overflow has occurred. The Exception Detection logic also generates the appropriate exception code that must be store in the Cause register. Please note that we are supporting two types of exceptions, Arithmetic Overflow and Bus Error on Instruction Fetch. The latter of those two should be signaled when either the Control Unit or ALU Control module signals an invalid instruction.

- The address fed into the EPC must be the PC of the instruction that caused the exception; however, we only have PC+4 so we must include a subtract by 4 logic block.

- When an exception occurs, the offending instruction must also be flushed. To support this, an EX_MEM_Flush signal must be added to the control portion of the EX/MEM pipeline registers. This signal is tied to the output of the Exception Detection block, which is active when an exception has occurred. Likewise, to flush the instructions behind the offending instruction in the pipeline, the exception signal has also been fed into the flushing logic created in the first part of lab. Please note that the flushing operation is identical to that of a branch miss prediction for the Fetch and Decode stages.

- To allow for re-fetching at the exception handler address, we must expand the multiplexer added to the processor in the first section of lab. The control for this multiplexer is shared by the prediction and exception signals.

- The ALU, ALU Control module, and Control Unit must be modified to support the detection of exceptions. Additionally, the ID/EX control registers must be expanded to support another signal from the Control Unit which indicates an unrecognized instruction was decoded. The Hazard Detection Unit should require no change as it performs the same function as outlined in the first part of lab.

(a) Modify the Control Unit such that when an instruction with an unknown OpCode is decoded, an error signal is generated. The updated module interface should be as follows:
module PipelinedControl(Jump, Branch, RegDst, UseImm, UseShamt, RegSrc, Jal, RegWrite, MemRead, MemWrite, SignExtend, ALUOp, JumpSel, OpInstrError, OpCode, FuncCode);

input [5:0] FuncCode; // [5:0] from instruction
output Jump; // active if instruction is of jump type
/*! Branch[1] active if bne (i.e. Outcome is Taken if AluZero is LOW) */
Branch[0] active if instruction is a bne or beq (same operation as before)*/
output [1:0] Branch;
output [1:0] RegDst; // select line for RW MUX
output UseImm; // 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
output MemWrite; // active if writing into memory
output SignExtend; // active if sign extension is needed
output JumpSel; // select line for jump target address MUX
output [3:0] ALUOp; // control ALU operation
output OpInstrError; // active if opcode is unrecognized

(b) Modify the ALU Control module such that it provides an output signal that is active when an unrecognized R-type instruction is decoded. Ensure that this signal is not active when an instruction is not an R-type instruction.

The updated module interface should be as follows:

module ALUControl(ALUctrl, RtypeInstrError, ALUop, FuncCode);

input [3:0] ALUop; // from control unit
input [5:0] FuncCode; // from instruction field
output reg [3:0] ALUctrl; // command control for ALU
output reg RtypeInstrError; // active when invalid R-type

(c) Modify the ALU to include an Overflow signal that is active when overflow occurs during an ADD or SUB operation. Ensure this signal is never active for operation other than the ADD and SUB. Overflow occurs during an ADD when both input operands have the same sign, but the sign of the result differs. SUB is similar except the sign of the second operand is inverted.

(d) Modify the remainder of the datapath. For the Cause register, exceptions will store a cause code in bits 6 down to 2. Consult the MIPS reference card provided on the laboratory website to determine what value must be stored for each type of exception. In MIPS, the EPC and Cause registers are located within the I/O coprocessor, and special privileged instructions exist for reading the contents of these registers. Since we do not have a Verilog model for the I/O coprocessor, define the aforementioned registers as output ports so the test bench can read them.

The updated module interface should be as follows:

module PipelinedProc(
Laboratory Exercise #8

input CLK,  
input Reset_L,  
input [31:0] startPC,  
input [31:0] exceptAddr,  
output [31:0] dMemOut,  
output reg [31:0] Cause,  
output reg [31:0] EPC
);

(e) 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.

Note: This simulation should run considerably longer than the previous simulations.

(f) 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) Recalculate the CPI for Program 1 and compare it to that of last lab (i.e. compare dynamic to static branch prediction). Did it change significantly? Why or why not? Please provide some insight into your results.

(b) Explain what Program 6 does. Compute the CPI for Program 6 using the dynamic branch predictor and re-simulate the execution of Program 6 with the static branch prediction policy from last lab. You can easily do so by wiring the BranchPredict signal to ‘0’! Compare your results and provide some reasons for why they differ from those in Question 1.

(c) Make note of the run time (in clock cycles) that your processor with Dynamic Branch Prediction requires to execute Program 7. As in the previous problem, execute Program 7 with the Static Branch Prediction scheme of “assume Not Taken” and compare the two run times. Examine the code for Program 7, specifically taking note of the branches. Then provide some insight into the performance differences you observe.
(d) If we wanted to add support for Interrupts, what changes would we have to make to our current design? Please provide some detail. What modifications would we have to make to support system calls?

(e) Compare the final clock rate of your design with that of last lab. Which components make up the critical path for this lab? Please provide some insight into what changed. What improvements could be made to the existing datapath to increase the overall efficiency of the processor core?

References