EE 309 Project Report
Hazard Control

Team:

- JAINESH MANISH MEHTA (210071001)
- ISHAAN MANHAR (210070033)
- HARSHRAJ CHAUDHARI (210040060)
- KUSHAGRA GEHLOT (21d070041)

Our pipeline was built on 2 assumptions i.e. No dependency of 2 instructions which are close by and No change in control flow. As we know programmers can't work with these two constraints and would not make efficient programs, hence we need to counter the 2 assumptions using some technique. For all the hazards we might or might not get penalties depending on the hazard. We have implemented 2 features in our pipeline which will help us counter the hazards i.e Stalling and Instruction Flushing.

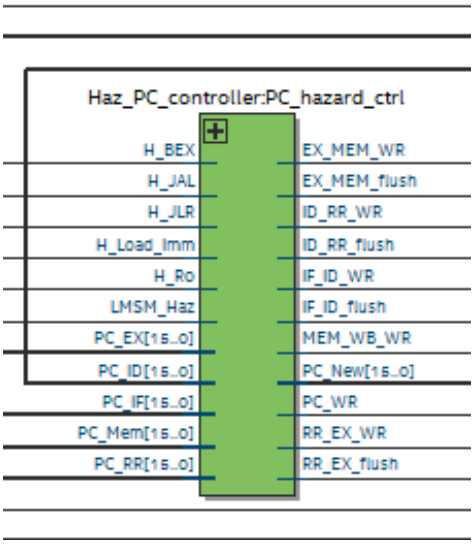
● Stalling

Stalling means we need to let some parts of the pipeline to stop and stay in their respective stage and the other part to move forward till they finish. We implement this in our PC-Hazard controller which will reset the write enable pins of the pipeline registers which are needed to be stalled and reset the PC write enable so that no new instructions are lost in fetching and the PC doesn't update itself.

LMSM Hazard: This is technically not a hazard if we use multiple data lines for data storing and loading but as we have limited resources we break the instruction into 8 Load or Store instructions using a sequential logic in Instruction Decode, and while this instruction is running we stall the pipeline of IF stage and ID keeps updating using the LMSM logic

● Instruction Flushing

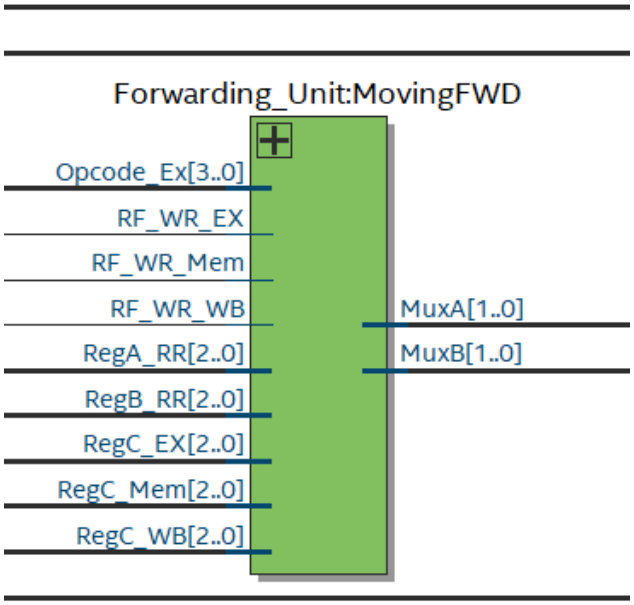
As we have used not take predictions for our Branching instructions we need to flush the instructions that our fetched after a taken branch so that they don't affect the behavior of the machine. We have implemented this using a Flush or a Cancel bit. This bit is given to all pipeline registers by the PC-Hazard Controller. If this bit in one then the instruction does not change the register file or memory or the flags. If you want to flush the instruction in RR stage we give a cancel bit to RREX pipeline register which is ORed with the cancel bit which was already passing from IDRR pipeline register.



1.Data Hazards and Forwarding Unit:

All the data hazards are fixed with zero penalty loss in cycles except 3 cases namely when R0 is the destination and source and when Load has immediate dependency (to be discussed later).

We have implemented a forwarding unit which checks the destination register of the previous instructions and see if the instruction which is in RR stage requires that instruction (RsID==Rd of EX or Mem or WB). Once it detects this hazard it forwards the signal using a MUX where the select lines of the MUX is given by our Forwarding Detection Unit , by implementing the following we won't be getting any cycle penalty and would not require stalling.



Now let's discuss about the exceptional cases:

1. **R0 destination Instruction**: In this we let the instruction come to MEM stage and once it reaches their our R0_Haz detector detects the hazard and gives signal to our branch and Hazard controller which then flushes all the instructions preceding and now the new instructions will be fetched from new R0 as the PC, hence giving us a 4 cycle penalty.
2. **R0 Source instruction**: As the programmer is unaware of the underlying computer architecture where we use R0 as the PC in a 6 stage pipeline and when we reach RR stage and fetch R0 the user intended to fetch the PC of that instruction but due to pipelining it will be fetching the PC of the instruction in IF stage. To counter this we added a Mux which gives PC of the R0 source instruction instead of the actual R0.
3. **Load Immediate:** We cannot solve this without any penalty loss of cycles as we dont have the data which needed to be loaded in the conflicting register hence our Load Immediate Hazard detects this subsequently informs our branch and hazard controller which then stalls the pipeline proceeding Load instruction by resetting all the pipeline registers' write enables.

2. Control Hazards:

We tried to implement a Branch Predictor but we were getting latches error and we couldn't fix it hence we implemented a not taken prediction (no prediction basically..XD).

We resolve our branching instructions in the following manner:

1. ID: JAL -Haz_JAL
2. RR: JLR, JRI -Haz_JLR
3. EX: BEQ, BLT, BEX -Haz_BEX
4. MEM: R0 destination instructions (pseudo branching) -R0_hazard

We have a similar detection and resolving method for all the branching instructions which could be understood by the following steps taken by our machine.

1. We deliver the correct PC to the PC-Hazard Controller all the time
2. The detector detects the hazard using the OPcodes (and the flag values for conditional branching or Destination register for R0 instructions).
3. Once the PC-Hazard Controller gets informed it flushes all the instructions after our hazarding instruction by giving them a cancel bit and updates R0 to the correct PC.

We will be getting the penalty according to the stage where the branch is resolved.

Errors in our design

1.Adding adder RR stage (Won’t create incorrectness):

If you see our datapath, we have added an adder in parallel to Register file thinking it won’t create an issue but later we realised that the stage would require Register value from RF and will then add it to the immediate hence giving us 2 delays which nullifies the purpose of pipelining. We identified this error in the later stages of our project and for solving this it would require adding multiple muxes in EX stage and changing code of Detection and hazard resolving units. Since this error would not give any incorrectness but instead will lower the maximum frequency of our cpu (given RR is the longest stage which is less likely due to our large RAM and ROM), we let this error slide by.

2.LMSM after a taken branch (Will give incorrectness):

According to our design we are implementing LMSM by a sequentially sending LW instructions to RR stage from ID and the IF stage is stalled. And we are flushing the instructions after a taken branch by giving a cancel or flushing bit to the stages. So when ID is in LMSM mode that time cancel bit would not remove LMSM instruction from ID stage.

There could be minor errors in the priority of the hazards as there are many combinations of the instructions which call the hazards.
Whichever we found while RTL testing we fixed it but there is a good probability of more corner cases like these which will cause incorrectness in our CPU.