IITB PipeLine : MicroProcessors Project 2

Sachin Goyal : 150020069
Srivatsan Sridhar : 150070005
Tanya Choudhary : 150070033
Pranav Sankhe : 150070009

The basic philosophy of pipeline implementation of the microprocessor design is to keep of fetching the next instructions and not wait for the previous one to get completed. That is we want a CPI = 1.

In the 6 stage pipelined design we made: THe name of 6 stages are ->
IF ID RR EX Mem Wb

IF Stage :
Instruction is fetched in this stage, we have our instruction memory here, the fetched instruction opcode is forwarded to IF_ID Pipeline register. There is also a NOP instruction which will be fetched when we want to stall. We also forward the PC and PC+1 value in the IF_ID Pipeline register since it may be required later if the destination register turns out to be R7 or the instruction is JAL or JLR.

ID Stage:
Instruction is decoded in this stage, we generate all the control signals which will be required later in the ID stage. Example -> MemWrite, RegWrite, Source Registers and destination registers, carry enable or zero flag enable.

RR Stage:
In this stage we have our register file. We have modified our register file where the R7 has a special write enable because it has to store the PC value. We have 4 muxes here in the RR stage ALU1 mux , ALU2 mux, T1

mux and T2 mux which select which data to pass forward to the ALU stage based on forwarding requirements.

EX, Mem and WB stage are the normal implementations as taught in the class.

PC and R7
The register 7 has to store the PC value, but that value should be written in the R7 only when that instruction reaches the WB stage. So we forward the PC value through all the stages. Also we have modified our Register file which is now capable to write in R7 along with write in any other registers. If R7 is the destination register for any instruction, then the output of that instruction will be stored in R7 and we will branch to the appropriate location. Our code tries to use minimum number of stalls in the case when you will have to branch because of R7 being modified. We forward the R7 value to the instruction memory as early as possible. If the instruction is LHI, R7 value is forwarded from the ID stage itself and thus only 1 stall. If add,adc,adz,ndu,ndc,ndz,adi instructions are there 3 stalls will be used and in case of memory based instructions 4 stalls.

Data Forwarding
We have implemented forwarding when data hazard is there. In the ID stage we find the source and destination registers of an instruction. If the source register is found to be equal to the destination register of any instruction which is in the later stages(Ex, mem or WB) , we forward the appropriate values for which we generate the signals, ALU1 mux, ALU2 mux, T1 mux, T2 mux.

Control Hazards
In case of BEQ, we are assuming branch as not taken (can be changed to “branch taken” by complementing the PC_mux signal value). If our assumption is wrong, we will be having 2 stalls.

For JAL we have 1 stall, for JLR 2 stalls(since we get the register value only in the RR stage)

Conditional Instructions

For the ADC ADZ instructions, there is a new hazard. The carry flag gets modified only in the write back stage. So if an ADD instruction (which modifies the carry flag) is followed immediately by the ADC instruction we have a potential hazard. For this we use a dummy carry flag (invisible to the programmer) which gets modified in the EX stage itself and thus we use that to modify the control signals for ADC and ADZ instruction based on whether the need to be executed or not.

LM and SM

For the LM and SM instructions we use a combination of Priority encoder and a decoder. This is done basically to reduce the number of iterations for this instruction to the number of bits equal to 1 (number of registers to be modified). We generate a VALID2 bit, which basically tells us whether more then 1 bits are still high, and therefore you will be stalling the IF stage till more then 1 bit is high(VALID 2 is high). As we write in each register we go on setting the corresponding bit back to 0. In the last iteration, VALId 2 goes down to 0 and thus we go out of the loop.

What we didn't implement ->

Due to lack of time, we didn't implement the 1 hazard which can be due to the Z flag being modified by a LW instruction and then LW being followed immediately by a ADZ instruction.

XXXX