EE2003-ASSIGNMENT 6 — REPORT

Sundar Raman P(EE17B069), Pruthvi Raj R G(EE17B114)

BRANCH PREDICTIONS and DMEM DELAY.

1 Introduction

As an extension of assignment 4 which is a multi cycle pipelined CPU, we attempted to make external 'BRANCH PREDICTOR' hardware to try to prevent stalls/roll backs because of wrong branches and 'DMEM DELAY' to create different explicit delay for different memory blocks (Cache vs DMEM) to simulate real life scenario. Both of them were simulated for critical instruction sets and hardware implemented for the same and worked as expected.

2 Branch Prediction

2.1 Explanation of idea

Programs often contain a lot of 'for' loops or while blocks. In class we described a pipeline and discussed how branch is a hazard and how we can handle it. We had chosen to always 'not' to take the branch(move on to PC+4) while the 'verdict' of the branch is getting prepared in later stages of the pipeline. However by doing so we generally 'waste' a clock cycle if the branch is supposed to be actually taken. Similar thing happens if we always decide to always take the branch.

Hence we think that the decision on whether to take or not take the branch(branch prediction) prior to the 'verdict' must be based on some statistics of previous encounters with those branch statements. Hence we came up with a simple statistic - we save the 'verdict' of every branch statement by allowing it to vote for *branch_not_taken_times* or *branch_taken_times*. We use this statistic while we branch predict - if the votes for*branch_taken_time* is more than the *branch_not_taken_times* , instead of loading PC+4, we load the branched PC and vice versa.

This we help us treat every branch statement individually based on their statistics. We read that, in general the program contain 'few' 'for' loops in which most of the time is used up for the execution. We also observed that in most of the for loops, the voting will be heavily biased towards one side(either

'branch_not_taken_times' or *'branch_taken_time')*. Hence our statistic of simple voting works well in saving time in each execution of branch statement.

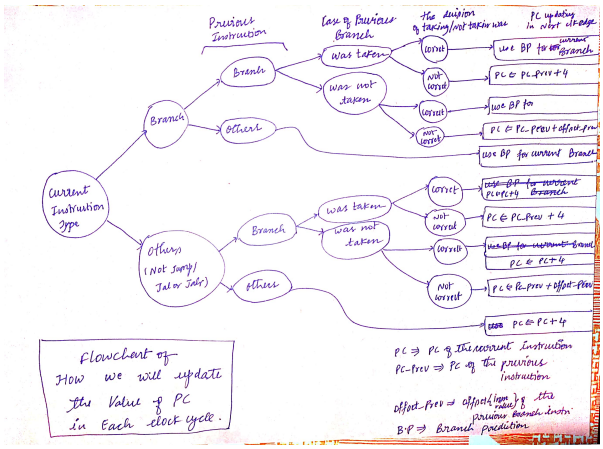


chart.jpg

Figure 1: FLOW CHART OF DIFFERENT CASES

2.2 Simulation Results

Explanation of the test case:
1. The test case includes 6 branch instructions. Only the last one should always be taken(it loops back to PC -12). Rest of the branches should never be taken.
2. The first time a branch instruction is encountered, it will be saved into the database with its pc, the branch pc(if branch was taken). Later if it encounters the same branch(PC value) it will use the statistic inside the database to predict whether the branch should be taken or not (take if *branch_taken_times* ¿ *branch_not_taken_times*). It will vote in the database again after the EX once the *alu_zero* signal is ready.
3. In this test case, in the case of branch with PC = 100(which should always be taken), when it was first encountered(t ¡ 5,000 ns), PC was updated to PC + 4(104)as there was no statistic to predict the branch and was rolled back to PC = 88 after the EX stage.But the next times(t¿5,000 ns) the same branch was encountered, it Branch predicted directly to PC = 88 using the statistic.
4. The static of voting can be seen in the database registers.

2.3 Hardware Implementation Results

`WHY WE CHOSE THE 4 STAGE PIPELINE:` We wanted to explore something different, something which is not described in the textbook. This required us to

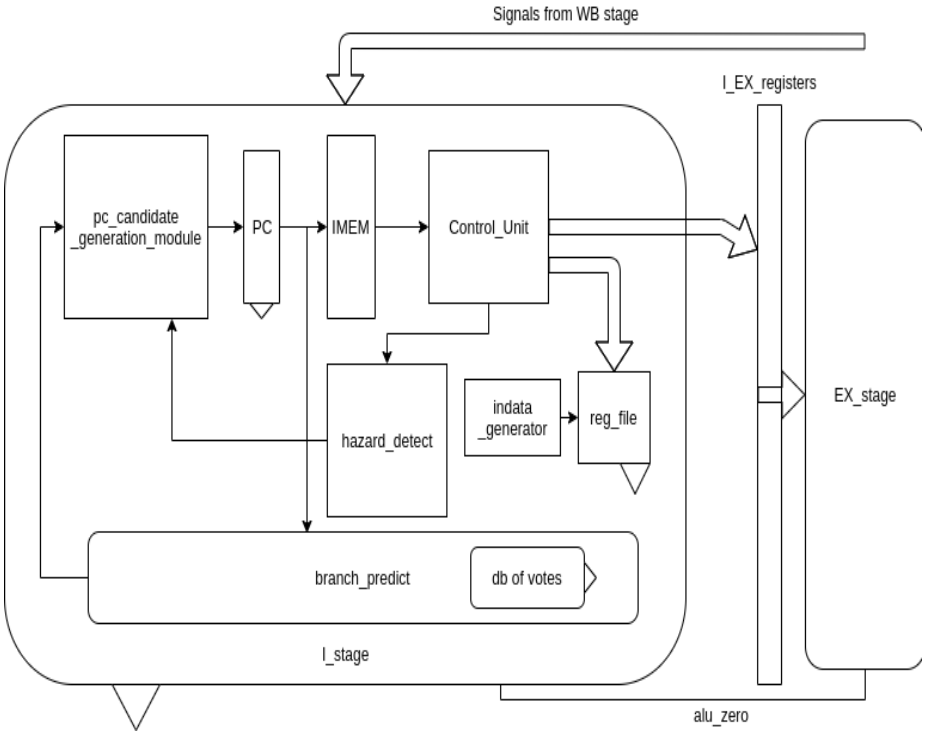


Figure 2: BRANCH PREDICTION BLOCK DIAGRAM.

ideate again on all types of errors possible in a 4 stage pipeline. We were also able to compare the results with our friends who designed a 5 stage pipeline. The difference in the maximum speed possible for running was found to be small. While the 5 stage pipeline could run at 200 MHz speed, our 4 stage pipeline could run at a speed of 190MHz. We clubbed the IF and ID stages and called it as "I" stage. *Branch prediction* module along with 4 stage pipelining was tested in Spartan 3E board ($XC3S500E\_FG320$) given in IE lab and working as expected.

3 DMEM delay

3.1 Explanation of idea

To simulate the real life scenario where depending on the *daddr* (address which goes to the data memory (Harvard arch.) to fetch/write values), the required value that needs to be read/written into should be from/into cache where there is less delay involved in reading/writting or DMEM where there is more delay involved (takes more clock cycles to fetch/write data), explicit DMEM delay inside MEM stage based on *daddr* is created.

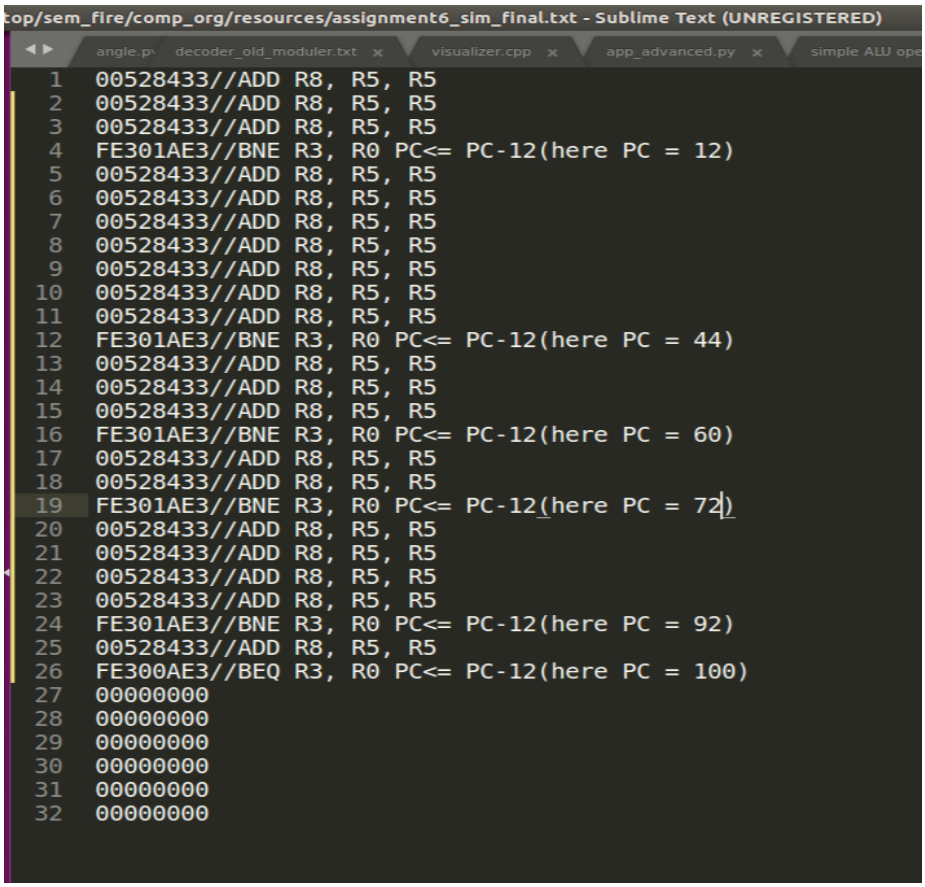


Figure 3: IMEM FOR BRANCH PREDICTION SIMULATION

If for store/load word instruction, daddr lies within a particular range (which is implicitely defined in MEM stage), then 'dstall' is set to 1 and 'delay' is set to a particular value (which is 5 in our case) where all operations (pc update, passing of values between internal registers to stages etc.) stalls for `delay` clock cycles (stalling occurs as long as dstall==1) after which normal pipelining operations are resumed and drdata and dwdata are set to the correct values. The above mentioned is similar to fetching data/writing into external primary/secondary memory which might take lot of clock cycles. If *daddr* doesn't fall within a particular range then delay and dstall are zero and conditions are normal as usual which is similar to fetching from cache.

3.2 Simulation Results

IMEM file contents and delay values corresponding to different LOAD/STORE instructions can be infered from the figure. Simulation results can also be seen

QUICK REFERENCE FOR CONTROL SIGNALS IN SIMULATION:

CONTROL SIGNAL	DESCRIPTION
pc	PC of the current instruction in "I" stage
current_instruction_type	Type of instruction in current PC 0 - ALU 1 - Branch 2 - Load 3 - Store/other
pc_branch	PC to updated if(in next clk edge) if branch is taken
pc_case	If current PC matches with a saved ones, then its register number.
take_branch	The branch prediction control signal If 1 ⇒ take the branch If 0 ⇒ do not take the branch
total_branch_count	Number of different PCs(which are branches) already saved in the database.

Figure 4: QUICK REFERENCE FOR CONTROL SIGNALS

QUICK REFERENCE FOR DATABASE REGISTERS

REGISTER NAME	DESCRIPTION
pc_probable_branch	The PC value of the branch instructions
pc_branch_location	The PC value to branch to if the corresponding branch is taken
pc_taken_times	Number of times this branch was taken
pc_not_taken_times	Number of times this branch was not taken

Figure 5: QUICK REFERENCE FOR DATA BASE REGISTERS

to adhere to it. Delay doesn't occur even if it is LOAD or STORE instruction in MEM stage but *daddr* falls in cache range.

Figure 6: BRANCH PREDICTION SIMULATION RESULT

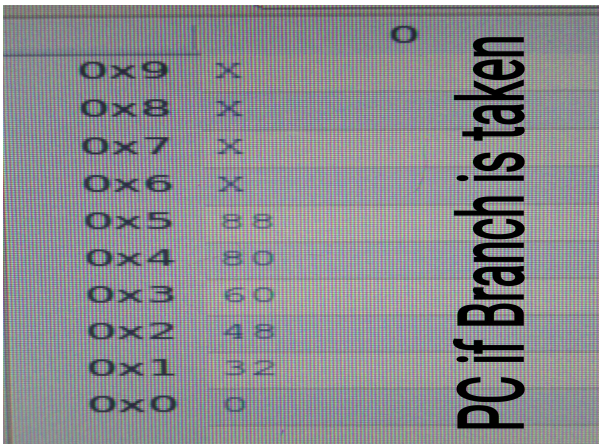


Figure 7: BRANCH PREDICTION TABLE1

3.3 Hardware Implementation Results

From the synthesis report inference, maximum clock frequency is 190 MHz. We implemented it on Zed board ($XC7Z020\_CLG484 - ZynqbasedSoC$) borrowed from CFI and verified the simulation using manual clock and VIO. Hardware implementable file can be found in the ZIP file.

4 Split up of works:

Both of us ideated together and wrote the code by splitting up different sub-modules for the 2 parts of assignment6. The instructions for the test

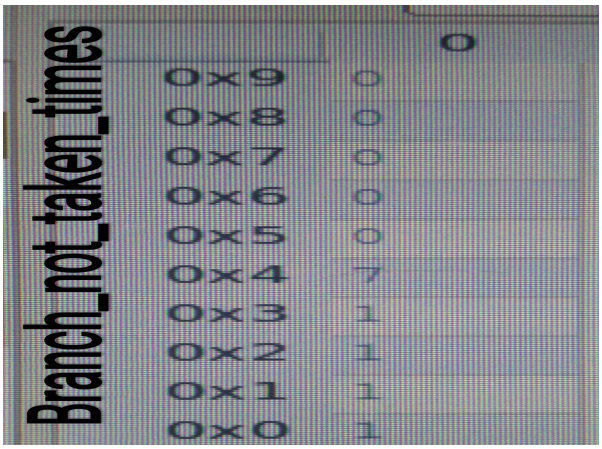


Figure 8: BRANCH PREDICTION TABLE2

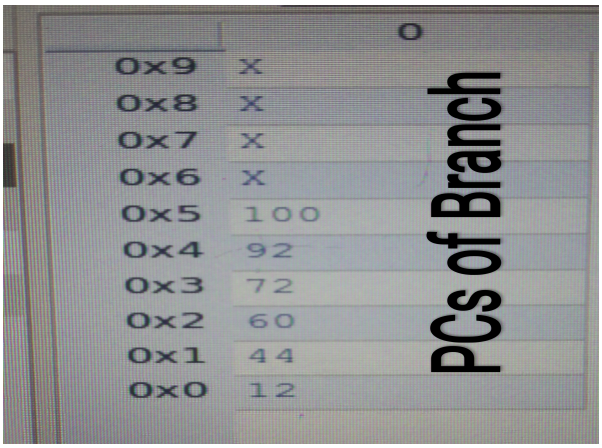


Figure 9: BRANCH PREDICTION TABLE3

cases were also made together. We have also shown the demo of assignment 6 to Professor Nitin when both team mates were present. Even report was done parallely by both of us (in overleaf).
Sundar: Branch prediction table ideation, DMEM delay for LOAD or STORE in MEM stage, updating voting list by keeping track of correctness of branches.
Pruthvi: Branch prediction implementation of database registers,testing and debugging,Branch Prediction hazards,DMEM delay - stalling all other operations

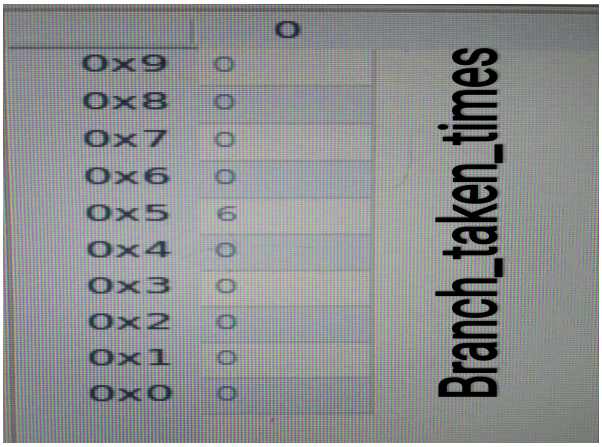
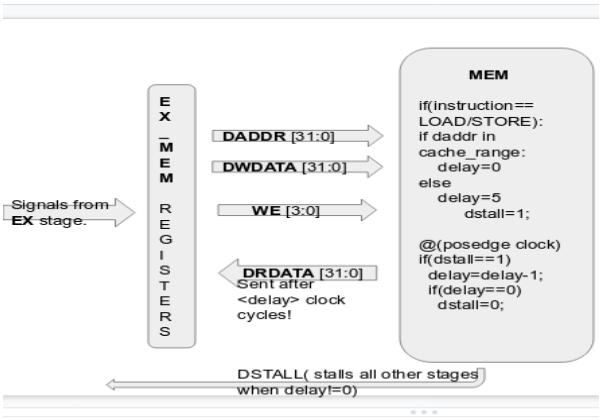


Figure 10: BRANCH PREDICTION TABLE4



```
00000fb3 //ADD R31, R0, R0
0000f093 //ANDI R1, R1, IMM=12'h000
00201103 //LH R2, 2(R0){STALL FOR 5 CLOCK CYCLES}
00044333 //XOR R6, R0, R8
0000f093
00044333
0000f093
00c02383 //LW R7, 12(R0){NO STALL AS DADDR FALLS OUTSIDE DMEM RANGE-[0,8]}
01900193 //3ADDI R3, R0, 12'd25
01800213 //4ADDI R4, R0, 12'd24
01900193
01800213
01900193
01800213
01900193
01800213
00601283 //LH R5, 6(R0){STALL FOR 5 CLOCK CYCLES}
0000f093
0000f093
0000f093
0000f093
0000f093
0000f093
0000f093
00000423 //SB R0, 8(R0){STALL FOR 5 CLOCK CYCLES}
00000fb3
00000fb3
00000fb3
00000fb3
00000fb3
00000fb3
00000fb3
```

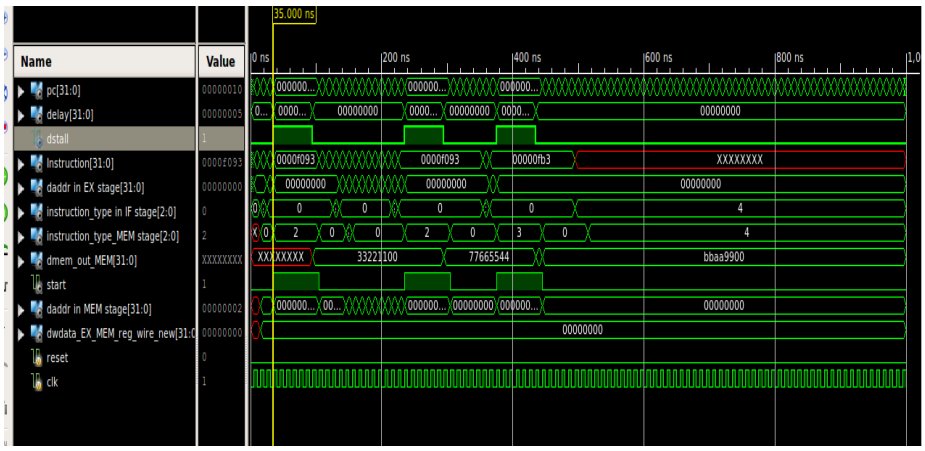


Figure 13: DMEM DELAY SIMULATION