

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_time*. We use this statistic while we branch predict - if the votes for*branch_taken_time* is more than the *branch_not_taken_time* , 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.

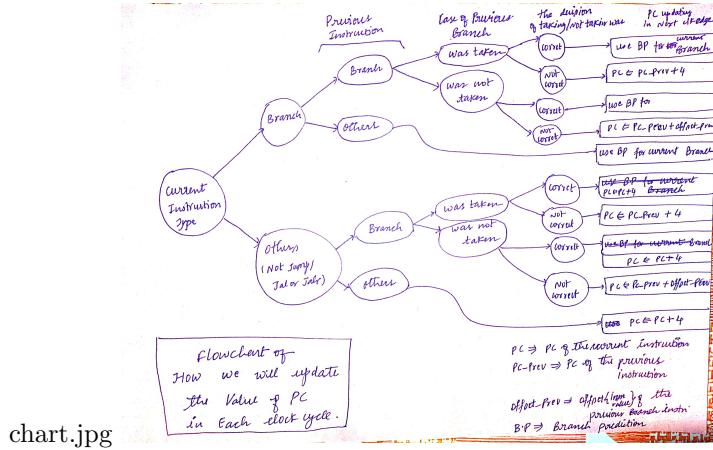


Figure 1: FLOW CHART OF DIFFERENT CASES

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_time* greater than *branch_not_taken_time*). 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 less than 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 greater than 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.

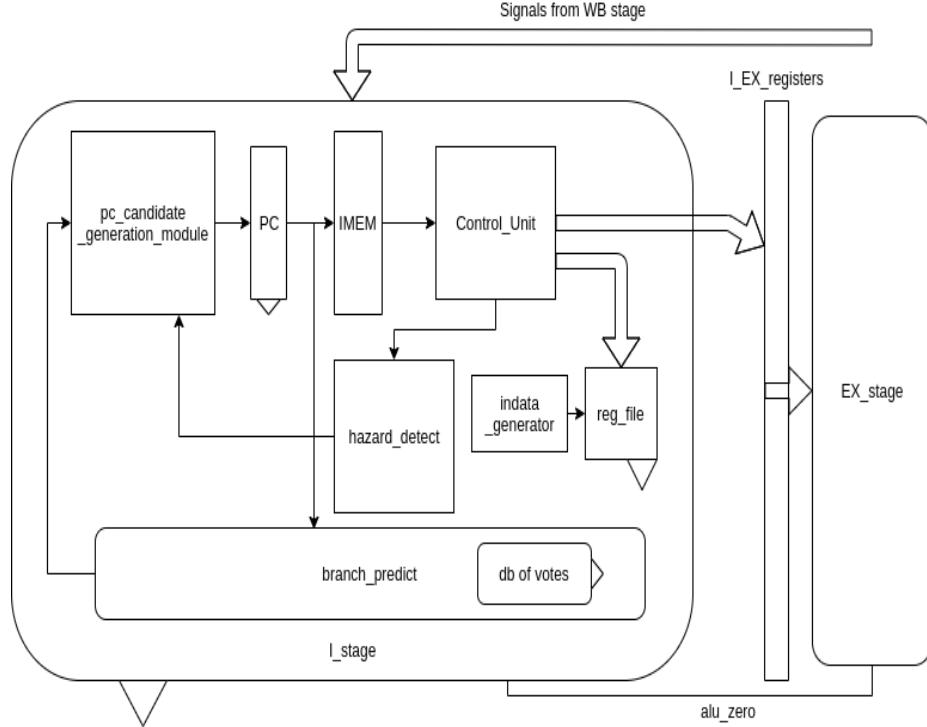


Figure 2: BRANCH PREDICTION BLOCK DIAGRAM.

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 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

```

1  00528433//ADD R8, R5, R5
2  00528433//ADD R8, R5, R5
3  00528433//ADD R8, R5, R5
4  FE301AE3//BNE R3, R0 PC<= PC-12(here PC = 12)
5  00528433//ADD R8, R5, R5
6  00528433//ADD R8, R5, R5
7  00528433//ADD R8, R5, R5
8  00528433//ADD R8, R5, R5
9  00528433//ADD R8, R5, R5
10 00528433//ADD R8, R5, R5
11 00528433//ADD R8, R5, R5
12 FE301AE3//BNE R3, R0 PC<= PC-12(here PC = 44)
13 00528433//ADD R8, R5, R5
14 00528433//ADD R8, R5, R5
15 00528433//ADD R8, R5, R5
16 FE301AE3//BNE R3, R0 PC<= PC-12(here PC = 60)
17 00528433//ADD R8, R5, R5
18 00528433//ADD R8, R5, R5
19 FE301AE3//BNE R3, R0 PC<= PC-12(here PC = 72)
20 00528433//ADD R8, R5, R5
21 00528433//ADD R8, R5, R5
22 00528433//ADD R8, R5, R5
23 00528433//ADD R8, R5, R5
24 FE301AE3//BNE R3, R0 PC<= PC-12(here PC = 92)
25 00528433//ADD R8, R5, R5
26 FE300AE3//BEQ R3, R0 PC<= PC-12(here PC = 100)
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
32 00000000

```

Figure 3: IMEM FOR BRANCH PREDICTION SIMULATION

value that needs to be read/written into should be from/into cache where there is less delay involved in reading/writing 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.

If for store/load word instruction, *daddr* lies within a particular range (which is implicitly 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.

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

3.2 Simulation Results

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

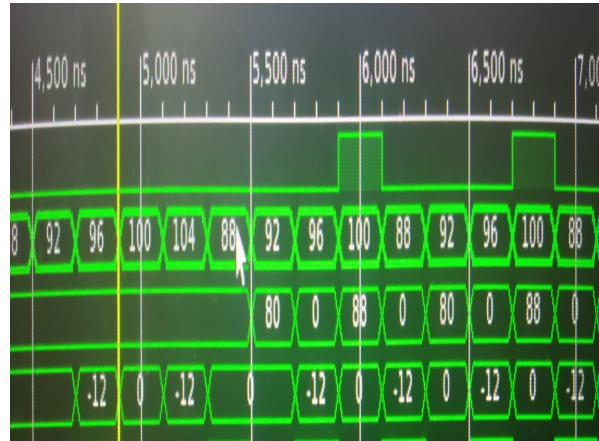


Figure 6: BRANCH PREDICTION SIMULATION RESULT



Figure 7: BRANCH PREDICTION TABLE1

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.

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.

Branch not taken times

| | |
|-----------|---|
| 0x0 | 1 |
| 0x1 | 1 |
| 0x2 | 1 |
| 0x3 | 1 |
| 0x4 | 1 |
| 0x5 | 1 |
| 0x6 | 1 |
| 0x7 | 1 |
| 0x8 | 1 |
| 0x9 | 1 |
| 0x0-0x9 | 0 |
| 0x10-0x1F | 0 |
| 0x20-0x3F | 0 |
| 0x40-0x5F | 0 |
| 0x60-0x7F | 0 |
| 0x80-0x9F | 0 |
| 0xA0-0xB0 | 0 |
| 0xC0-0xD0 | 0 |
| 0xE0-0xF0 | 0 |
| 0xF1-0xF2 | 0 |
| 0xF3-0xF4 | 0 |
| 0xF5-0xF6 | 0 |
| 0xF7-0xF8 | 0 |
| 0xF9-0xFA | 0 |
| 0xFB-0xFC | 0 |
| 0xFD-0xFE | 0 |
| 0xFF | 0 |

Figure 8: BRANCH PREDICTION TABLE2

| | 0 |
|-----|-----|
| 0x9 | x |
| 0x8 | x |
| 0x7 | x |
| 0x6 | x |
| 0x5 | 100 |
| 0x4 | 92 |
| 0x3 | 72 |
| 0x2 | 60 |
| 0x1 | 44 |
| 0x0 | 12 |

PCs of Branch

Figure 9: BRANCH PREDICTION TABLE3

4 Split up of works:

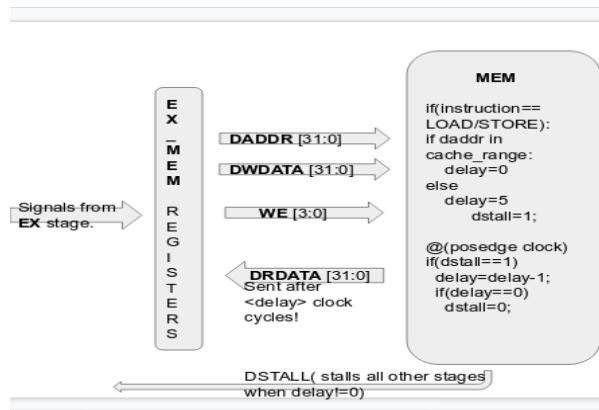
Both of us ideated together and wrote the code by splitting up different sub-modules for the 2 parts of assignment6. The instructions for the test cases were also made together. We have also shown the demo of assignment 6 to Professor Nitin when both team mates were present. Even report was done parallely by both of us (in overleaf).

Pruthvi: Branch prediction implementation of database registers, testing and debugging, Branch Prediction hazards, DMEM delay - stalling all other operations.

| | Branch Taken |
|-----|--------------|
| 0x9 | 0 |
| 0x8 | 0 |
| 0x7 | 0 |
| 0x6 | 0 |
| 0x5 | 6 |
| 0x4 | 0 |
| 0x3 | 0 |
| 0x2 | 0 |
| 0x1 | 0 |
| 0x0 | 0 |

Branch_taken_times

Figure 10: BRANCH PREDICTION TABLE4



Sundar: Branch prediction table ideation, DMEM delay for LOAD or STORE in MEM stage, updating voting list by keeping track of correctness of branches.

```

memp_mainmem x memz_commem x
1 0000fb3 //ADD R31, R0, R0
2 0000f93 //ANDI R1, R1, IMM=12'h000
3 0020103 //LH R2, 2(R0){STALL FOR 5 CLOCK CYCLES}
4 00044333 //XOR R6, R0, R8
5 0000f93
6 00044333
7 0000f93
8 0000fb3 //LW R7, 12(R0){NO STALL AS DADDR FALLS OUTSIDE DMEM RANGE-[0,8]}
9 01900193 //3ADDI R3, R0, 12'd25
10 01800213 //4ADDI R4, R0, 12'd24
11 01900193
12 01800213
13 01900193
14 01800213
15 01900193
16 01800213
17 00601283 //LH R5, 6(R0){STALL FOR 5 CLOCK CYCLES}
18 0000f93
19 0000f93
20 0000f93
21 0000f93
22 0000f93
23 0000f93
24 0000f93
25 0000fb3 //SB R0, 8(R0){STALL FOR 5 CLOCK CYCLES}
26 0000fb3
27 0000fb3
28 0000fb3
29 0000fb3
30 0000fb3
31 0000fb3
32 0000fb3

```

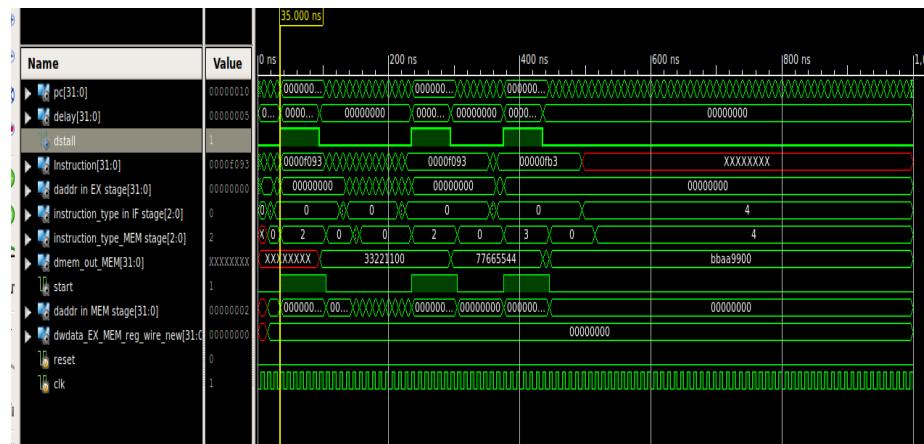


Figure 13: DMEM DELAY SIMULATION