EE446 Laboratory Project Report

Pipelined Processor with Hazard Unit and Branch Predictor

Murat Bayındır – 2231371

Ramazan Cem Çıtak – 2231686

Submission Date: Wednesday, June 14, 2023

***Abstract*— This paper describes the theoretical and experimental results of the project of Pipelined Processor with Hazard Unit and Branch Predictor.**

***Keywords*— Pipelined Processor, Hazard Unit, Branch Predictor, FPGA, DE1-SoC, Verilog HDL.**

I. Introduction

The aim of this project is to design a pipelined processor with hazard and branch predictor units. Firstly, according to the requirements of the project, we implemented changes in the datapath of a pipelined processor. Then, after designing the hazard and branch predictor units, we test the processor with a number of Assembly test codes.

II. Requirements for the Project and Background Information

A. Requirements of the Project

Firstly, a datapath that is capable of executing the Instruction Set Architecture (ISA) in the project manual. Then, having a working pipelined processor with a modified datapath, hazard unit and branch predictor hardware should be designed in Verilog HDL and Schematic Editor of Quartus. Finally, a testbench should be written to test the new datapath, hazard unit, and branch predictor.

B. Pipelined Processor

A pipelined processor divides a single-cycle processor operation into 5 stages: Fetch, Decode, Execute, Memory, and Writeback. Thus, in the ideal case, the throughput of the pipelined processor is 5 times faster than the single-cycle processor [1].

C. Hazard Unit

The hazard unit handles the situations where a result of an operation in an instruction is needed by the next instructions before the preceding instructions are completed.

Using NOP (No Operation) instruction is the software solution for hazards. On the other hand, a more useful and efficient solution to this problem is hazard units that are similar to the one designed in this project.

D. Branch Predictor

A branch predictor makes guesses about whether the branch is taken or not. For example, loops in the programs use branches many times, which gives a clue about branching. This is static branch prediction.

Also, there is dynamic branch prediction. In this project, we tried to implement a dynamic one that consists of Branch Target Buffer (BTB), Global History Register (GHR), and Pattern History Table (PHT). BTB is a table that has the destination address of a branch and information whether that branch is taken or not.

III. Implementation of the Project and Test Results

The project has four main parts: Datapath, Hazard Unit, Branch Predictor, and Testbench. For the simulation results, Vivado software was used.

A. Datapath Changes

a. Description of Data-Processing Instructions

ADD, SUB, AND, ORR, MOV, and CMP are data-processing instructions that we implemented in the project. These instructions make arithmetical and logical operations using register and immediate values.

An additional MOV operation exists in our ISA as its register level operations are the following:

Rd← (imm8 rr rot<< 1)

Its usage is MOV Rd,rot-imm8 where Rd are a destination register.

b. Description of Memory Instructions

STR and LDR are memory instructions used in the project. Using STR, we write a register value to an address in the memory. LDR loads a register by a value in a memory address. Addresses are kept in the registers as well as immediate values.

Micro-operations for STR:
Mem[Rn + imm12] ← Rd

Micro-operations for LDR:
Rd ← Mem[Rn + imm12]

c. Description of Branch Instructions

B, BEQ, BL, and BX are the branch instructions in the ISA that we implemented in the project. For branch instructions, Program Counter (PC) is added with 8, and then an immediate value is added to PC to reach the target address. B means branch, and BEQ is conditioned branch on equality.
BL puts the (PC+4) to Link Register R14, and then makes a branch operation.

Micro-operations for BL:
PC ← (PC + 8) + (imm24<< 2), R14 ← PC + 4

BX exchanges the PC with a register value in our ISA.

B. Hazard Unit

We tried to implement hazard unit by applying forwarding in Verilog HDL and created its schematic version in Quartus. Then, in the datapath schematic, we combined it with the whole datapath. The hazard unit connections can be seen in Figure 1 to connect it with the datapath.

Figure 1. The Hazard Unit to be connected to datapath [1]

C. Branch Predictor

We tried to implement BTB, GHR and PHT in Verilog HDL as well as adding it to our datapath.

The branch predictor maintains a BTB, a tag table, and a saturation counter array, each indexed by INDEX_SIZE. The BTB stores the actual target address for each branch instruction, the tag table stores a part of the PC as a tag, and the counter array implements a two-bit saturating counter as the branch history table (BHT).

The module operates as follows:

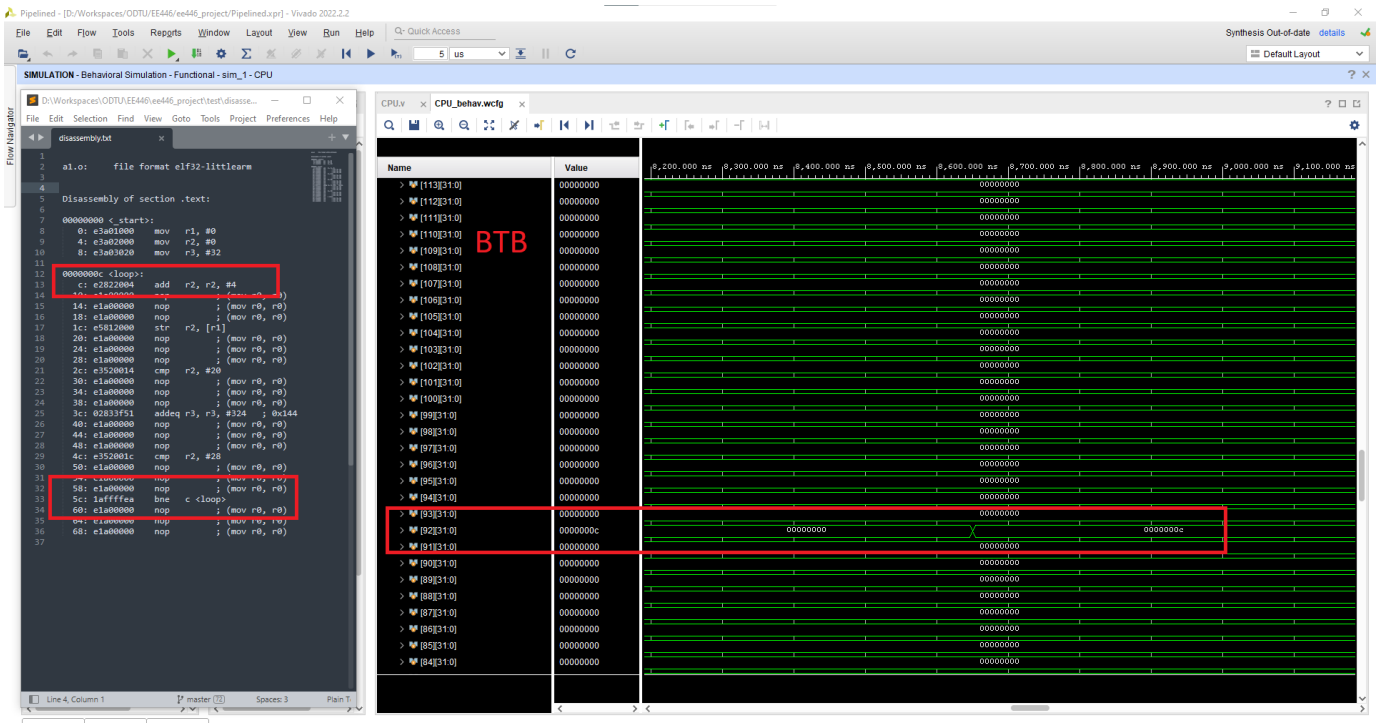
At every positive edge of the clock, if reset is high, it initializes the BTB, tag table, and counter array to their default values and sets the prediction history to 00. If reset is not high, it shifts the prediction history register.

The PredictTaken wire is assigned based on whether the tag stored in the tag table matches the high-order bits of the PC and whether the corresponding counter value is greater than or equal to 2. This forms the basis of the branch prediction.

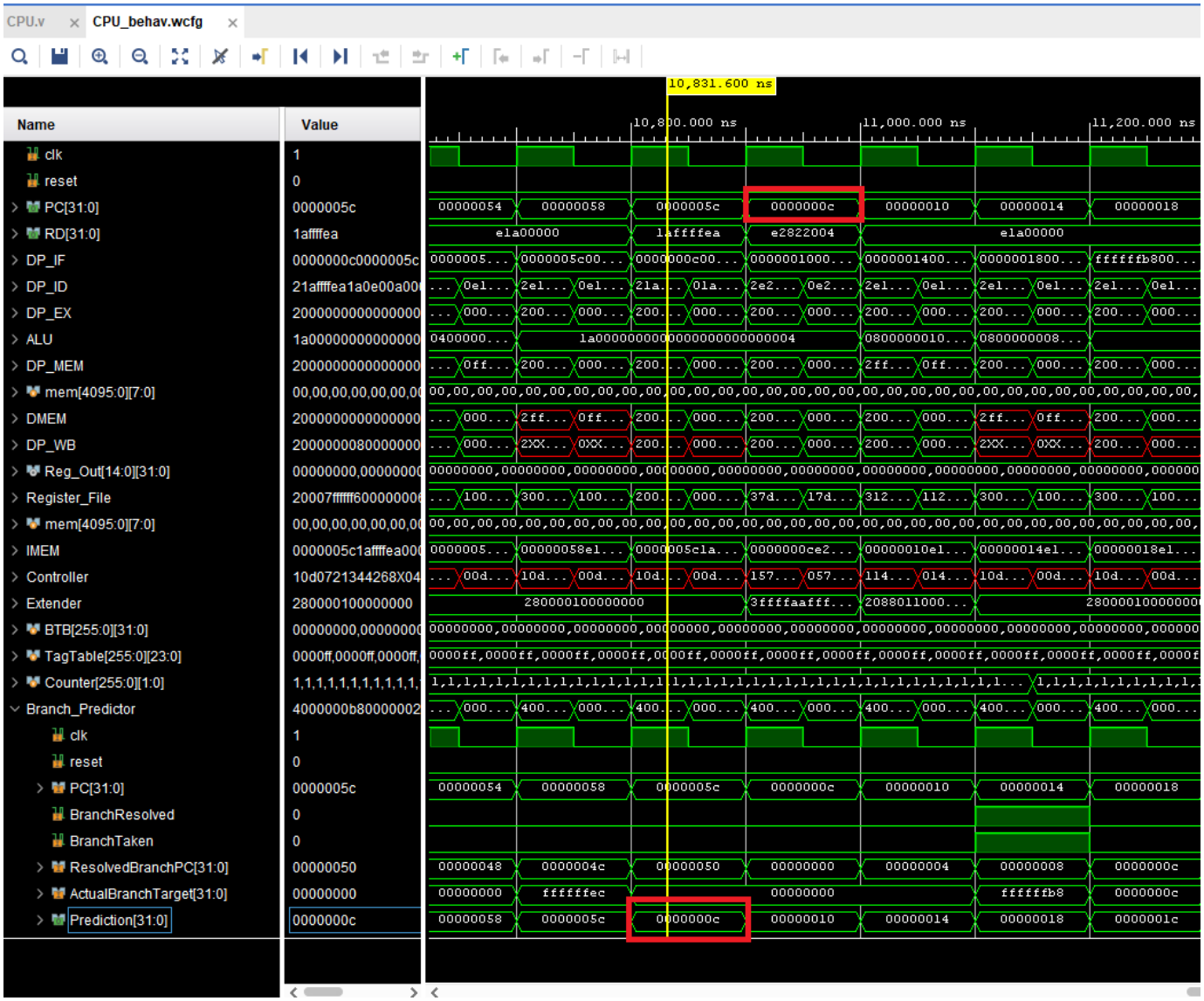
The Prediction wire is assigned the value from the BTB if the branch is predicted to be taken, or the next sequential address (PC+4) otherwise.

At every negative edge of the clock, if a branch instruction has been resolved (BranchResolved is high), the module updates the BTB, tag table, and counter based on whether the branch was taken or not taken.

Here when the branch instruction comes BTB is updated when the branch target address is calculated.



Next time when the branch occurs since we have the BTA in BTB , branch prediction starts to work and PC is correctly updated.

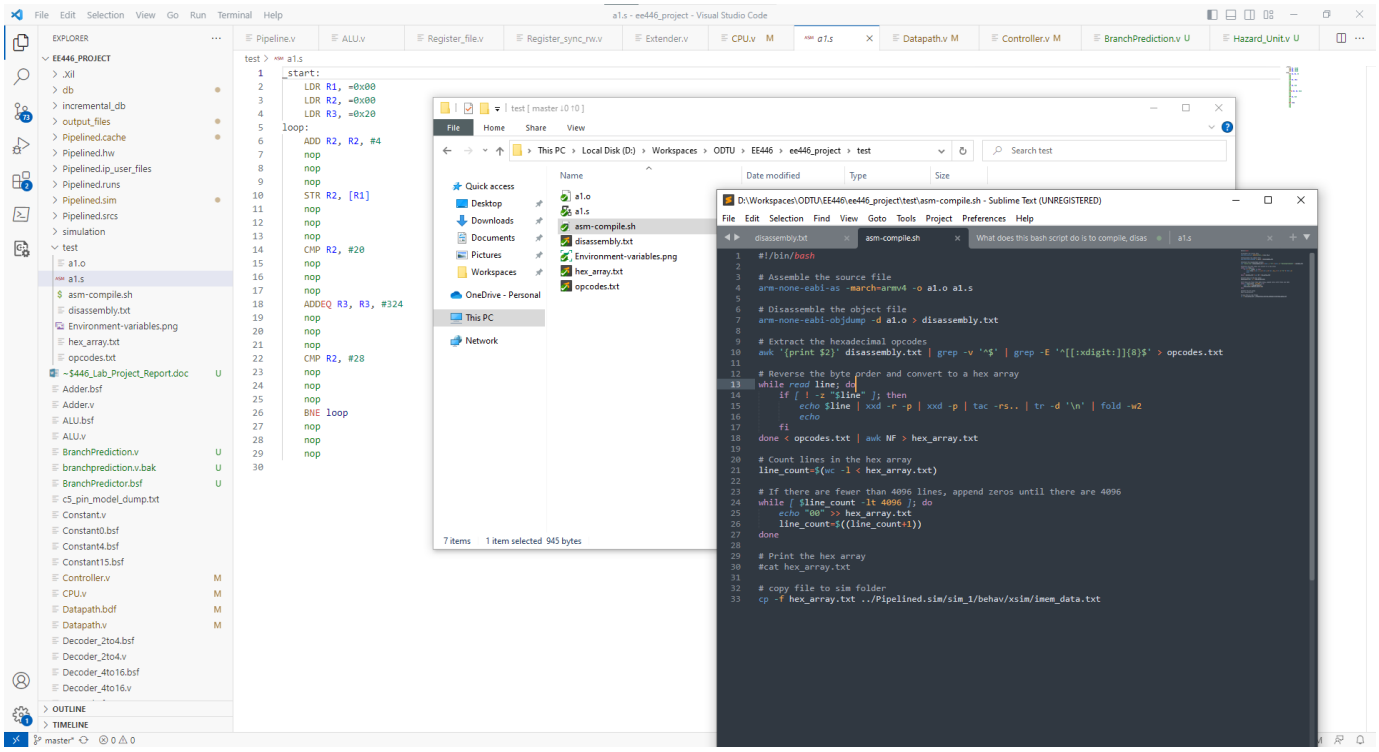


D. *Testbench*

a. *Converting Assembly Code to Machine Language*

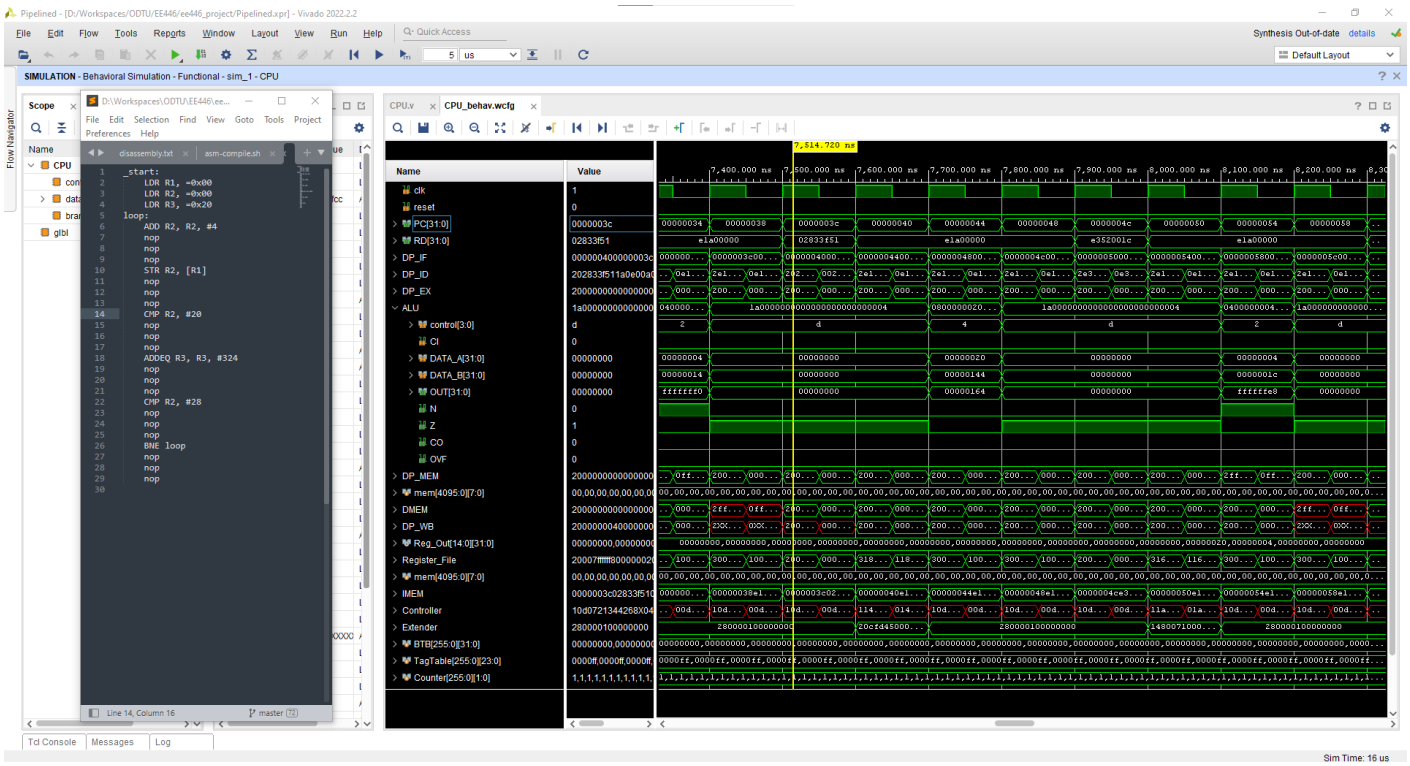
We used GNU Arm Embedded Toolchain to generate the machine code that is used in the Verilog HDL code in the Quartus project to execute Assembly codes and routines.

We created a bash script to ease our testing. What does this bash script do is to compile, disassemble, reorganize, and zero-pad the test ARM assembly code, and copy the memory data to the simulator folder so that we will be able to use the asm program in our mcu simulation.



b. *Test Results*

We simulated our design and obtained an example screen as in Figure.



IV. Conclusion

In this project, we used our theoretical knowledge of the pipelined processor covered in lectures by implementing it in an FPGA board. Designing a certain instruction set architecture, hazard unit and branch predictor developed our understanding of computer architecture and coding and using schematic skills in Quartus software.

V. The Sharing of the Tasks In the Project

Table 1: The Sharing of the Tasks

Tasks	Murat Bayındır	Ramazan Cem Çıtak
Datapath Modifications	% 50	% 50
Hazard Unit	% 20	% 80
Branch Predictor	% 80	% 20
Testbench	% 80	% 20
TOTAL	% 57.5	% 42.5

References

[1] S. L. Harris and David Money Harris, *Digital Design and Computer Architecture (ARM Edition)*. Elsevier, 2016.