EE312: Report for Lab Assignment 5

Introduction

We implement the control path (the signals that coordinate the working of different components, e.g. deciding whether the mechanism for a jump should be activated) and data path (the wires and components through which data flows and is processed) for the pipelined CPU.
Its major components include instruction memory (where code is stored), data memory (where data is stored), ALU (for logic/math operations), register file (where data must be brought in before it can be operated on and stored afterwards), the pipeline register(which will store the control signals and data are to be forwarded to the next stage in each pipeline stage) and clock to generate the signals to move onto next instruction as well as to initialize all components. For the control path, we add a separate logic to generate the necessary control signals (e.g. register file write-enable etc) correctly depending on each instruction.
We also introduce a mechanism to detect data hazards that might occur in the pipeline and implement a **FORWARDING UNIT** and use the always_not_taken policy for branch prediction.

Design

In the RISCV_TOP module, we implement all the pipeline registers, such as a register to keep track of the PC of the instruction in each stage, the operands of the instruction in the EX stage, and so on. These registers not only allow us to increase the IPC via pipelining but also are key to our forwarding unit and branch prediction.

We designed a "pipe_control" module that generates the control signals for each stage of the pipeline. It is in this module that we implemented our forwarding unit and branch prediction, whose design is explained below. We implemented stalling in three cases: i. when a load is immediately followed by an operation on the loaded value, we stall for one cycle to ensure that the required value can be loaded, ii. when a jump is encountered, we stall for two cycles to ensure that the correct PC value will be taken, and iii. when it is found that our never-taken prediction is wrong, and the branch actually needs to be taken. Flushing is accomplished by

Forwarding Unit: The Forwarding Unit is a module within the pipe_control module. We implemented a forwarding unit using pipeline registers; we check the relationship between the different instructions in the pipeline. If we have an instruction whose operand register is an output register of an instruction further along in the pipeline (which is easily checked since each instruction includes its operand and output registers), we use the pipeline registers to bring that data to the instruction directly so that we do not have to stall the pipeline until the data is written back to the register file and then read again. This way, we ensure that accurate values of the operand are available to the instructions without stall. The exception which still requires stalling is a combination of loading data from the memory and then operating upon it; in this case, although the data can still be taken to the instruction in the EX stage, stall is still needed, although due to the forwarding it is reduced compared to a non-pipelined system. The hazard detection is performed in another module within the pipe_control module, called the Hazard Detection module.

Branch Prediction: We used the "not-taken" branch predictor, where whenever we encounter a branch, we assume that the branch will not be taken. In case it turns out that the branch condition evaluated to False, we just continue the execution and there is no stalling. In case it is found that we mispredicted, the pipeline is flushed (by this time we have 2 wrong instructions in the pipeline, since we find out the result of the condition evaluation in EX stage, and the instructions in the IF and ID stages are wrong; therefore, we turn them to NOP and this effectively stalls our pipeline for 2 cycles, and from the 3rd cycle onwards we start executing the right statement). Conversion to NOP is a form of pipeline flushing, i.e. we discard the current values in the pipeline registers of theses stages; since these values were not written to memory or register file, this has the same effect as not having run the wrong instructions at all.

<u>Implementation</u>

We introduce some new registers after each stage (pipeline registers) to store the control signals and data to be forwarded to the next stage.

Pipeline registers

Instruction fetch and Instruction Decode stage

1. Instruction
2. Halt signal
3. Incremented PC address (in case it is needed needed later for an instruction)

Instruction Decode and Execution stage

1. Instruction
2. Halt signal
3. Sign extended 32-bit immediate field
4. The register numbers to read the two registers
5. Incremented PC address

Execution and Data Memory Access stage

1. Instruction
2. Halt signal
3. Sign extended 32-bit immediate field
4. The register numbers to read the two registers
5. Results of the execution stage

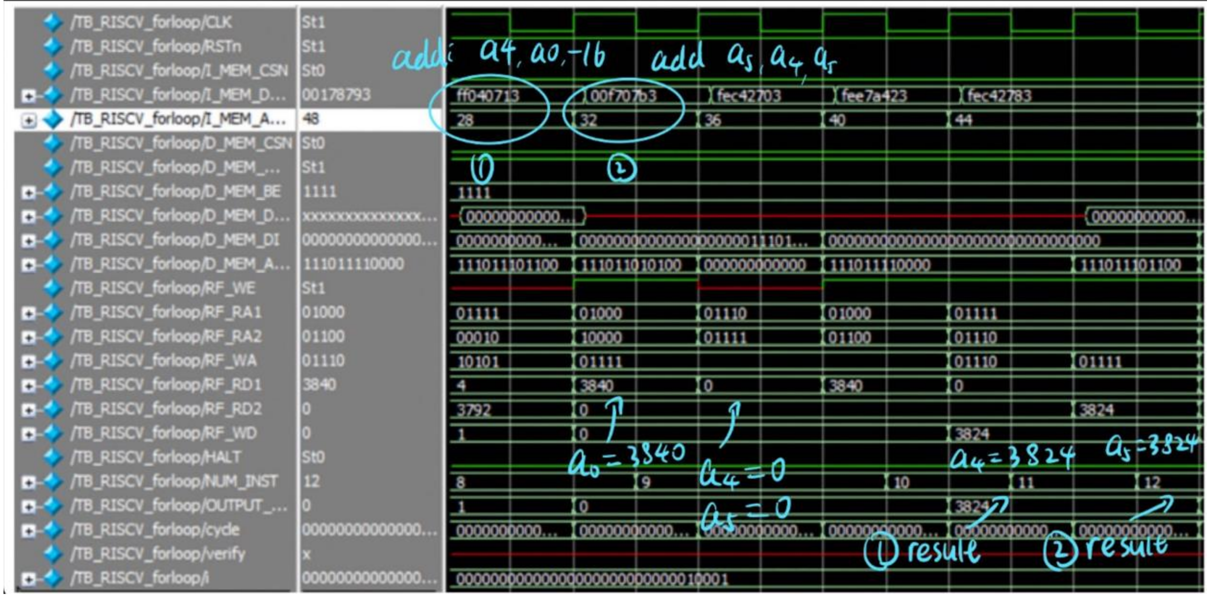
Data Memory Access and Write Back stage

1. Halt signal
2. Register name to which the data is to be written into

Forwarding Unit

There are 3 scenarios forwarding

1. From the memory to the execution stage
2. From the write back to the execution stage
3. From the write back stage to the instruction decode stage


/TB_RISCV_forloop/CLK St1
/TB_RISCV_forloop/RSTn St1
/TB_RISCV_forloop/I_MEM_CSN St0
/TB_RISCV_forloop/I_MEM_D... 00178793
/TB_RISCV_forloop/I_MEM_A... 48
/TB_RISCV_forloop/D_MEM_CSN St0
/TB_RISCV_forloop/D_MEM_... St1
/TB_RISCV_forloop/D_MEM_BE 1111
/TB_RISCV_forloop/D_MEM_D... xxxxxxxxxxxxxxx...
/TB_RISCV_forloop/D_MEM_DI 0000000000000...
/TB_RISCV_forloop/D_MEM_A... 111011110000
/TB_RISCV_forloop/RF_WE St1
/TB_RISCV_forloop/RF_RA1 01000
/TB_RISCV_forloop/RF_RA2 01100
/TB_RISCV_forloop/RF_WA 01110
/TB_RISCV_forloop/RF_RD1 3840
/TB_RISCV_forloop/RF_RD2 0
/TB_RISCV_forloop/RF_WD 0
/TB_RISCV_forloop/HALT St0
/TB_RISCV_forloop/NUM_INST 12
/TB_RISCV_forloop/OUTPUT_... 0
/TB_RISCV_forloop/cycle 0000000000000...
/TB_RISCV_forloop/verify x
/TB_RISCV_forloop/i 0000000000000...
ff040713 00f707b3 fec42703 fee7a423 fec42783
28 32 36 40 44
addi a4, a0, -16
add a5, a4, a5
①
②
a0 = 3840
a4 = 0
a5 = 0
a4 = 3824
a5 = 3824
① result
② result

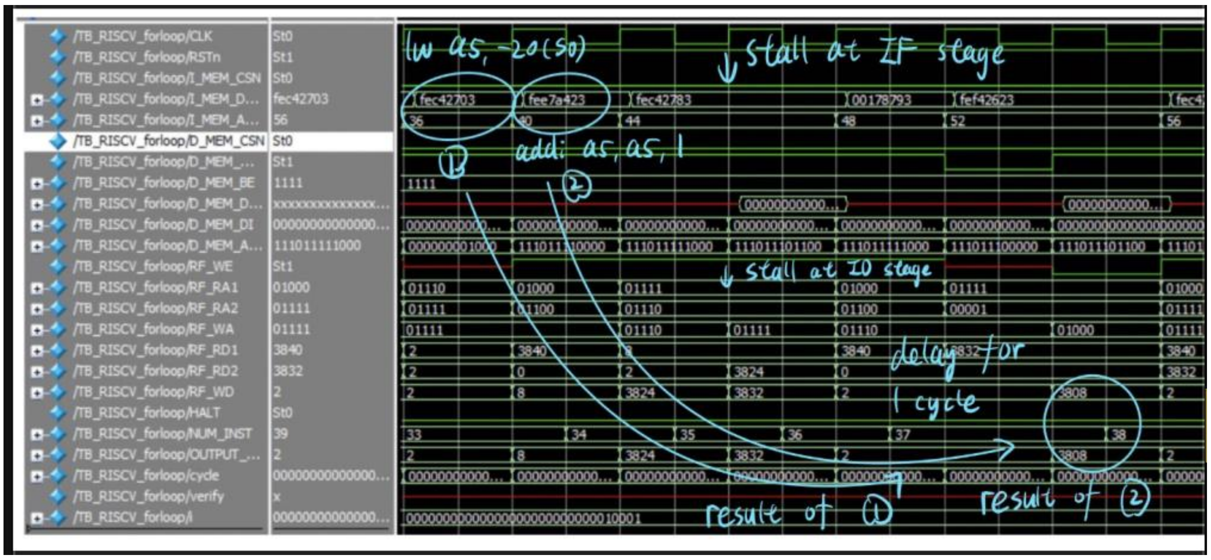
Data Forwarding from the Memory Access stage to Execution stage

Stalling mechanism and hazard detection

For all jump instructions, we stall the pipeline by 2 cycles

If we detect a data hazard for a load instruction, we stall the pipeline by 1 cycle.

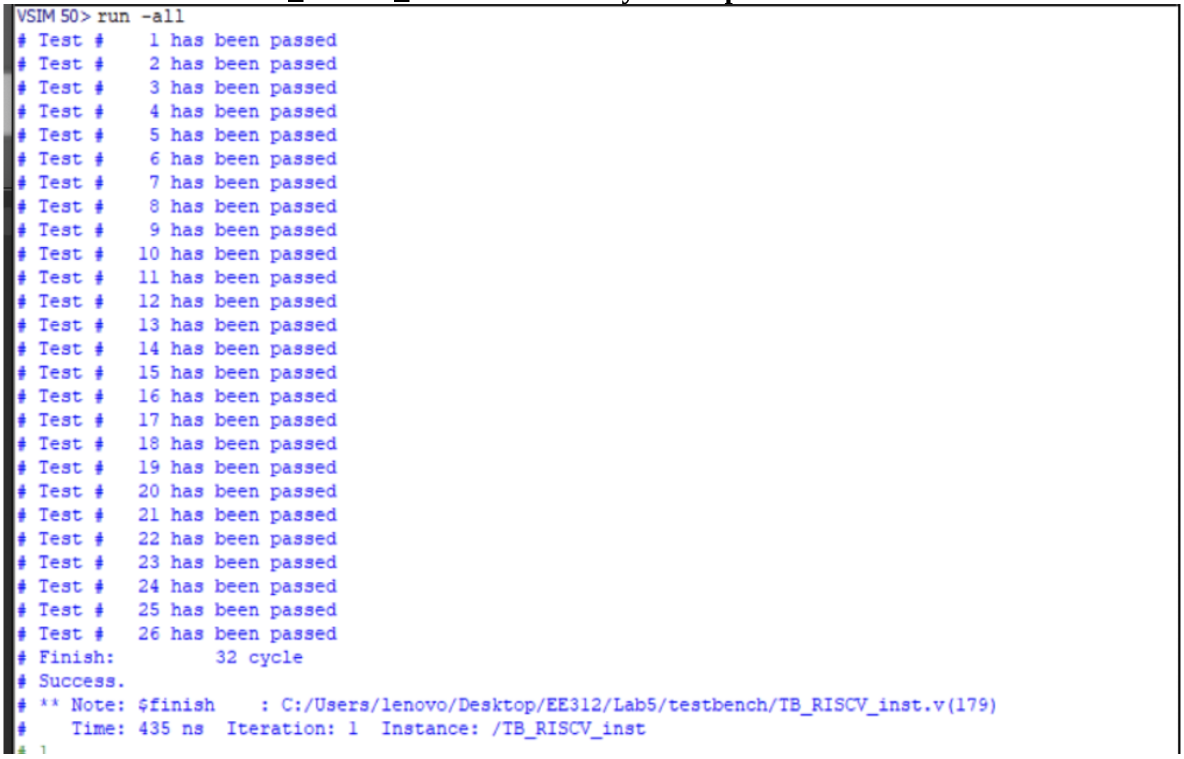
We detect a data hazard by checking if a same register is being used by the load instruction (currently in its execution stage) and an instruction which is in its instruction decode stage.



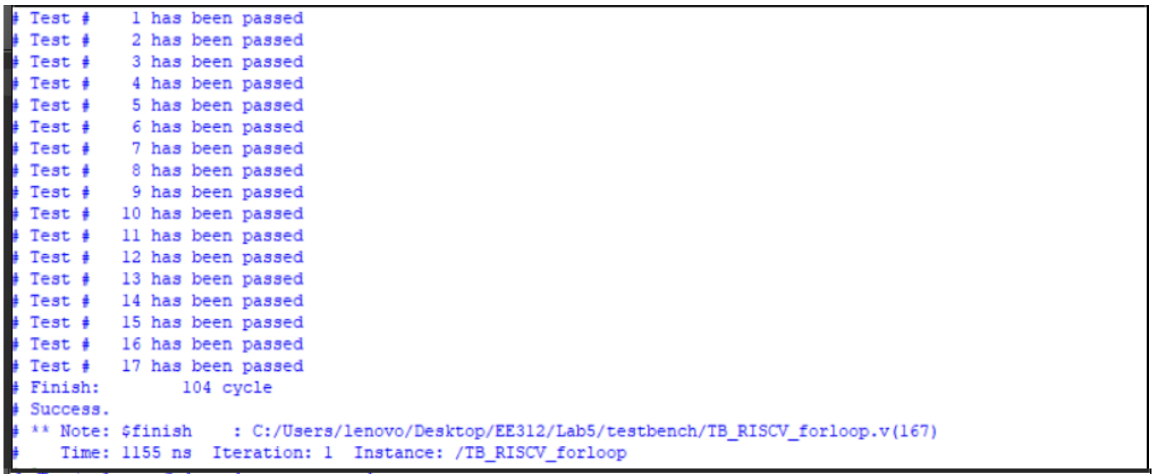
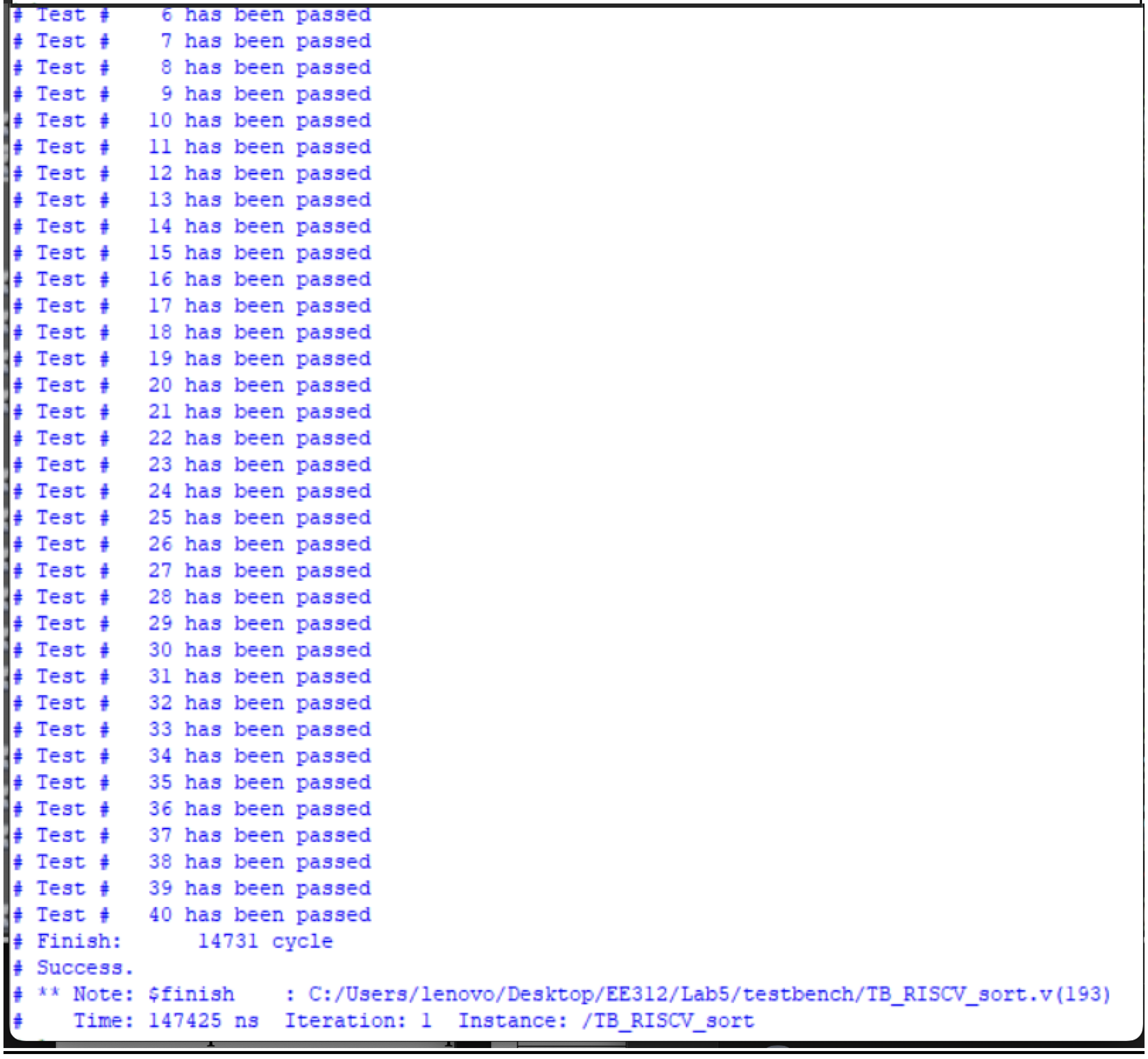
When we encounter data hazard in a load instruction

Evaluation

We ran the code using all three testbench files, forloop, inst and sort by placing one of them in the template folder and using the Simulate button on ModelSim. **We passed all 26 cases for TB_RISCV_inst with 32 cycles , all 17 cases for TB_RISCV_forloop with 104 cycles elapsed, and all 40 cases for TB_RISCV_sort with 14731 cycles elapsed.**



```
VSIM 50> run -all
# Test #    1 has been passed
# Test #    2 has been passed
# Test #    3 has been passed
# Test #    4 has been passed
# Test #    5 has been passed
# Test #    6 has been passed
# Test #    7 has been passed
# Test #    8 has been passed
# Test #    9 has been passed
# Test #   10 has been passed
# Test #   11 has been passed
# Test #   12 has been passed
# Test #   13 has been passed
# Test #   14 has been passed
# Test #   15 has been passed
# Test #   16 has been passed
# Test #   17 has been passed
# Test #   18 has been passed
# Test #   19 has been passed
# Test #   20 has been passed
# Test #   21 has been passed
# Test #   22 has been passed
# Test #   23 has been passed
# Test #   24 has been passed
# Test #   25 has been passed
# Test #   26 has been passed
# Finish:          32 cycle
# Success.
# ** Note: $finish    : C:/Users/lenovo/Desktop/EE312/Lab5/testbench/TB_RISCV_inst.v(179)
#    Time: 435 ns  Iteration: 1  Instance: /TB_RISCV_inst
```


```
# Test #    1 has been passed
# Test #    2 has been passed
# Test #    3 has been passed
# Test #    4 has been passed
# Test #    5 has been passed
# Test #    6 has been passed
# Test #    7 has been passed
# Test #    8 has been passed
# Test #    9 has been passed
# Test #   10 has been passed
# Test #   11 has been passed
# Test #   12 has been passed
# Test #   13 has been passed
# Test #   14 has been passed
# Test #   15 has been passed
# Test #   16 has been passed
# Test #   17 has been passed
# Finish:        104 cycle
# Success.
# ** Note: $finish    : C:/Users/lenovo/Desktop/EE312/Lab5/testbench/TB_RISCV_forloop.v(167)
#    Time: 1155 ns  Iteration: 1  Instance: /TB_RISCV_forloop
```

```
# Test #    6 has been passed
# Test #    7 has been passed
# Test #    8 has been passed
# Test #    9 has been passed
# Test #   10 has been passed
# Test #   11 has been passed
# Test #   12 has been passed
# Test #   13 has been passed
# Test #   14 has been passed
# Test #   15 has been passed
# Test #   16 has been passed
# Test #   17 has been passed
# Test #   18 has been passed
# Test #   19 has been passed
# Test #   20 has been passed
# Test #   21 has been passed
# Test #   22 has been passed
# Test #   23 has been passed
# Test #   24 has been passed
# Test #   25 has been passed
# Test #   26 has been passed
# Test #   27 has been passed
# Test #   28 has been passed
# Test #   29 has been passed
# Test #   30 has been passed
# Test #   31 has been passed
# Test #   32 has been passed
# Test #   33 has been passed
# Test #   34 has been passed
# Test #   35 has been passed
# Test #   36 has been passed
# Test #   37 has been passed
# Test #   38 has been passed
# Test #   39 has been passed
# Test #   40 has been passed
# Finish:      14731 cycle
# Success.
# ** Note: $finish    : C:/Users/lenovo/Desktop/EE312/Lab5/testbench/TB_RISCV_sort.v(193)
#    Time: 147425 ns  Iteration: 1  Instance: /TB_RISCV_sort
```

Discussion
We didn't face issues in implementation of the pipelined CPU but were unable to implement a branch target buffer.

Conclusion
This assignment marked our advancement to a processing paradigm where instructions are broken down into stages so that we can run several instructions at once and (ideally) not have any module be idle.