Hazard Elimination on a RISC-V32i 5-stage Pipelined CPU

Darío Caballero Polo

dcp@connect.ust.hk

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

ELEC5140 Advanced Computer Architecture

Department of Electronic and Computer Engineering

Guided by Linfeng Du - linfeng.du@connect.ust.hk

Hong Kong SAR, China - 25/05/2023

Abstract

This project focuses on the optimization of a RISC-V32i 5-stage Pipelined CPU by eliminating data and control hazards. The optimization is achieved by adding data forwarding and branch prediction to the pipeline. The results show a significant improvement in the performance of the CPU, reducing the number of stalls and increasing the instruction throughput. This project demonstrates the effectiveness of these techniques in improving the performance of pipelined CPUs.

Table of Contents

Acknowledgements

I would like to thank Linfeng Du for all the help he has given me not only in the development of this project, but also throughout the entire course. I had a lot of doubts about how to face this project and he was able to successfully guide me through the entire process.

1. Introduction

Computers have been a central piece in human development since their conception. This raised the necessity of creating faster and more efficient architectures. One such architecture is the pipelined CPU, which implements instruction-level parallelism by dividing the instruction execution process into multiple stages and allowing multiple instructions to be executed simultaneously, trying to always keep every part of the processor busy and thus reducing the CPI.

However, pipelined CPUs can suffer from data and control hazards that can negatively impact their performance. Data hazards, in particular the true dependencies—which are the ones that can affect our single-issue CPU performance—occur when an instruction depends on the result of a previous one that has not yet completed execution. Control hazards occur when the outcome of a branch instruction is not known, causing the pipeline to stall while waiting for the outcome to be determined.

This report presents an in-depth analysis and optimization of a RISC-V32i 5-stage Pipelined CPU. The main goal of this project is to eliminate data and control hazards by implementing data forwarding and branch prediction. The results show a significant improvement in the performance of the CPU, demonstrating the effectiveness of these techniques in improving the performance of pipelined CPUs. It should be noted that this work is only focused on improving the CPI, and does not look into the frequency of the system.

2. Related works

The base CPU model was provided by Linfeng Du via GitHub. It supported 31 basic instructions from the RISC-V32i base instruction set and was implemented in Verilog. A RISC-V assembler was also provided to translate the assembly instructions into hexadecimal. [1]

This processor has a pipelined architecture but suffers from the aforementioned problems: true data dependencies and control hazards. Both are handled by stalling the pipeline, which negatively impacts the throughput of the system.

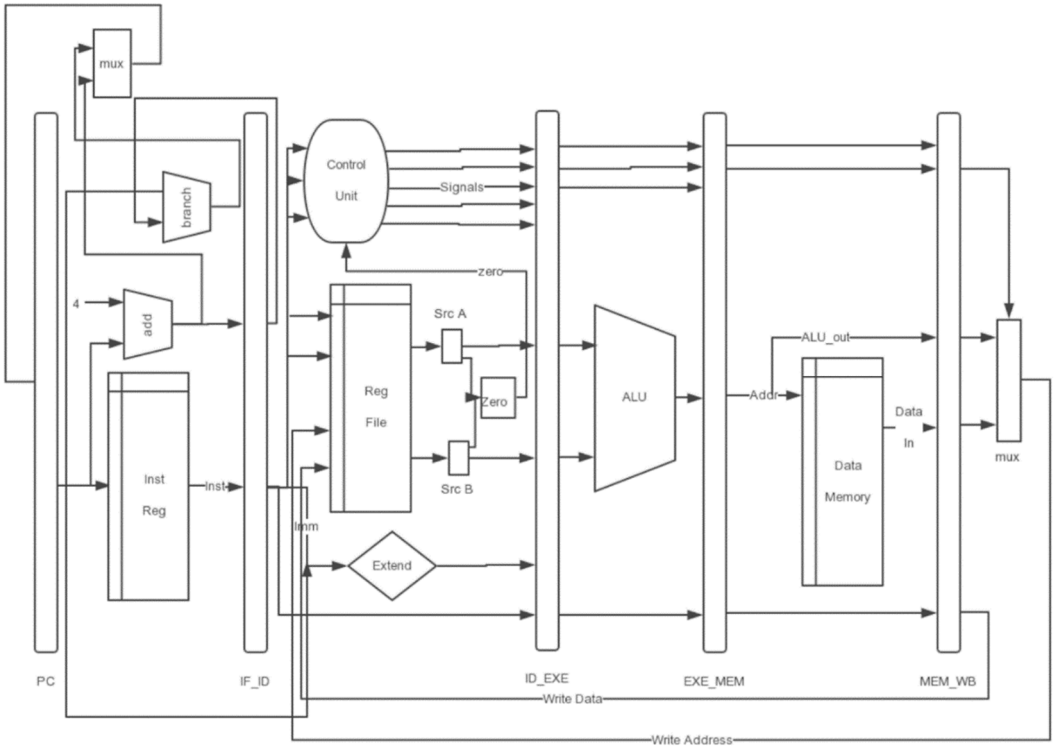


Figure 1: Simplified diagram of the microarchitecture of the base processor

The base processor from which we are working has a dedicated ID_ZERO_GENERATOR, which is used to determine the branch result, at the Instruction Decode stage. This can also be performed inside the ALU in the Execution stage. However, by moving it to the previous stage, the number of stall cycles required when a control hazard is encountered is reduced by one. This little optimization is possible because the data hazard stalls ensure that the operands in the Instruction Decode stage are always up to date. Nonetheless, after implementing data forwarding, the Instruction Decode stage may not have access to the most updated values for the operands, so a major reorganization of this part of the CPU would be needed.

All the information about the optimization techniques used in this project was taken from the lecture materials from ELEC5140 Advanced Computer Architecture by Prof. Wei Zhang. [2] [3]

3. Methods

3.1. Data forwarding

3.1.1. Forward Unit

All the logic related to data forwarding was included inside the Forward Unit.

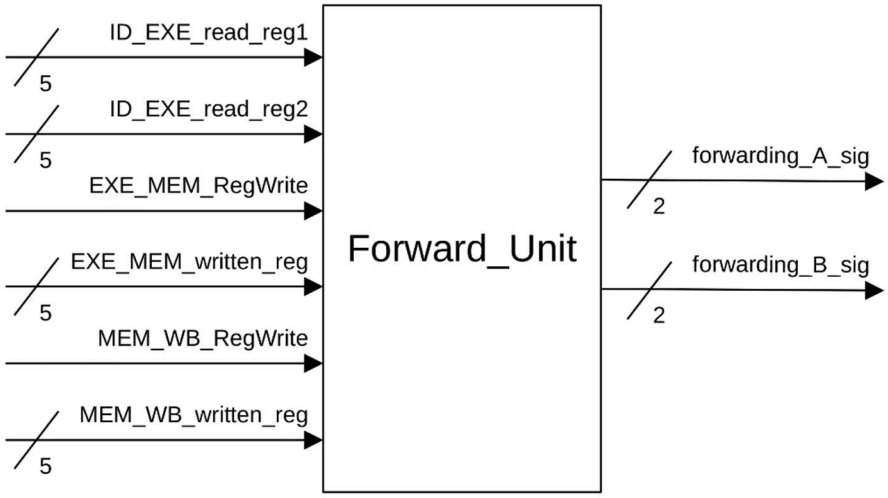


Figure 2: Forward Unit diagram

It receives as inputs the source registers (Rs and Rt, the ones storing the operands) of the instruction currently at the ID_EXE pipeline register: ID_EXE_read_reg1 and ID_EXE_read_reg2, as well as the written register of the instruction at the EXE_MEM pipeline register: EXE_MEM_written_reg. EXE_MEM_RegWrite is just a signal to indicate that the instruction at the EXE_MEM pipeline register wrote to a register. The remaining inputs are the equivalent ones for the instruction at the MEM_WB pipeline register.

The instruction inside the ID_EXE pipeline register is the one from which the ALU is going to take the operands. The one in the EXE_MEM went through the ALU one cycle ago, and the one in the MEM_WB two cycles ago. This means that the EXE_MEM and MEM_WB have the most updated values in that specific order.

By comparing the source registers and the written ones, we can detect the data hazards. If the instruction in the EXE_MEM register wrote into the first operand (operand A) of the instruction in the ID_EXE register, then we need to signal that the operand A should be forwarded from ID_EXE into the ALU. This is the same for operand B and the instruction in the EXE_MEM register. However, as stated before, the EXE_MEM pipeline register has more recent values, so if a dependency is detected in both EXE_MEM and MEM_WB simultaneously, only the latter should forward the data.

The two outputs of the module are the signals for forwarding the operators A and B. They have a width of two bits: the MSB indicates if forwarding is required from the EXE_MEM register, and the LSB from the MEM_WB register.

3.1.2. Data forwarding flow

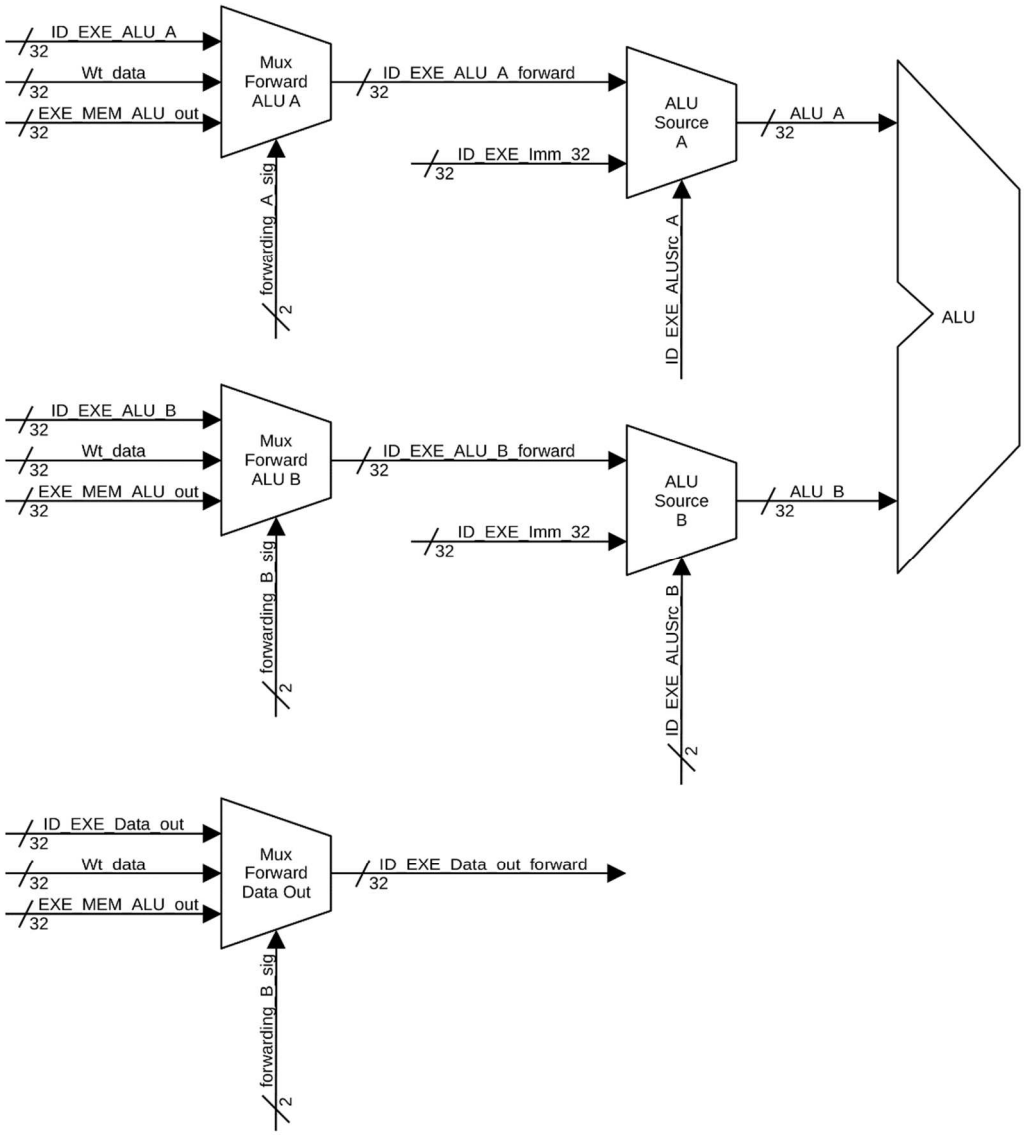


Figure 3: Forwarding multiplexers diagram

As shown in the figure, a major reorganization of the structure was performed to make the data forwarding work in conjunction with the branch prediction. The Zero Generator module was removed and its functions assigned to the ALU. This change was performed to locate the branch resolution in a point located after the data has already been forwarded, since the most recent values are needed for that. The ALU Source multiplexers were moved from the ID stage into the EXE, since they need to select between the read register operands (either forwarded or not) provided by the Forwarding Multiplexers, and

the immediate value. Finally, we also need to forward the Data Out for the writing instructions, since that does not go through the ALU, but is just passed from the ID_EXE to the EXE_MEM pipeline register directly.

3.1.3. Eliminated data stalls

By implementing data forwarding we made it possible to eliminate all data stalls except for load-use data hazards. Those occur when a load instruction writes into a register which is read by the next instruction. If that happens, the data is not ready until the MEM stage, so the second instruction must wait one cycle for the data to be read at the MEM stage and then forwarded to the ALU input. The load-use data hazards cannot be eliminated with data forwarding, and would require a superscalar architecture with reordering capabilities to do so.

3.2. Branch prediction

3.2.1. Branch prediction pipeline

For the branch prediction, two new modules were introduced: the Branch Predictor and the Branch Checker.

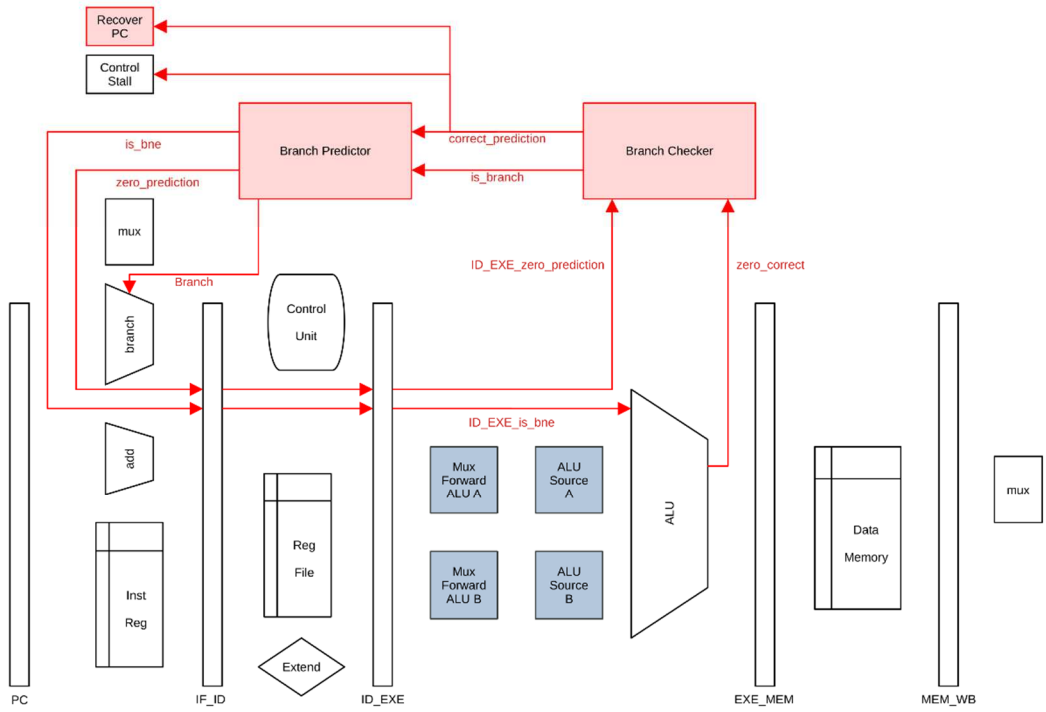


Figure 4: Simplified diagram for the branch prediction pipeline

In Figure 4 we can appreciate a very simplified diagram of the new pipeline. It was built off the Figure 1 diagram, but major differences are present. One of those is that all wires not directly related to the new branch prediction pipeline have been removed for visual clarity. Another one is that the data forwarding-related multiplexers—marked in blue—are located in the Execution stage, as mentioned in the last section. Finally, in red we find the modules containing the logic for the branch prediction and their wires.

In the ID stage, the Branch Predictor gives a Branch value without knowing the real one. This prediction is passed directly into the branching multiplexer, which decides the next PC to write back. Additionally, the prediction is passed through the pipeline registers as zero_prediction. It takes the value of 1 when the branch was taken and 0 when it was not taken. An additional is_bne signal is passed alongside it, which tells the ALU to invert its correct_zero value. Note that these values traverse from the ID to the EXE stage through the pipeline registers to ensure that they are synchronized with their corresponding instruction, and reach the ALU at the same time as the branching instruction itself. In the ALU, the branching result is ascertained, since all the operands already have their most updated value. The result is output as zero_correct. As before, it is 1 if the branch should have been taken, and 0 otherwise. Both zero_correct and ID_EXE_zero_prediction reach the Branch Checker module at the same time. This module just checks whether they are equal or not. If they are, the output correct_prediction is 1. Otherwise, it is 0. It should be noted that the Branch Checker also receives the Operation Code as an input, so if the instruction is not a branching one, correct_prediction also takes the value of 1. That means that whenever correct_prediction is 0, we can be sure that a misprediction occurred. To fix it, the IF_ID and ID_EXE pipeline registers are flushed when correct_prediction is 0. That is handled by the Control Stall module. In addition to that, we recover the correct PC. To do so, when we did the prediction in the IF stage, we passed through the pipeline registers the PC value that should have been taken provided the prediction was incorrect. That is, if we predicted Taken, we would have passed PC + 4. If we predicted Not Taken, we would have passed the branching address. Finally, the information about the correctness of the last prediction is passed back to the Branch Predictor. There, the BHR and the PHT entries are updated according to it.

3.2.2. Branch Predictor

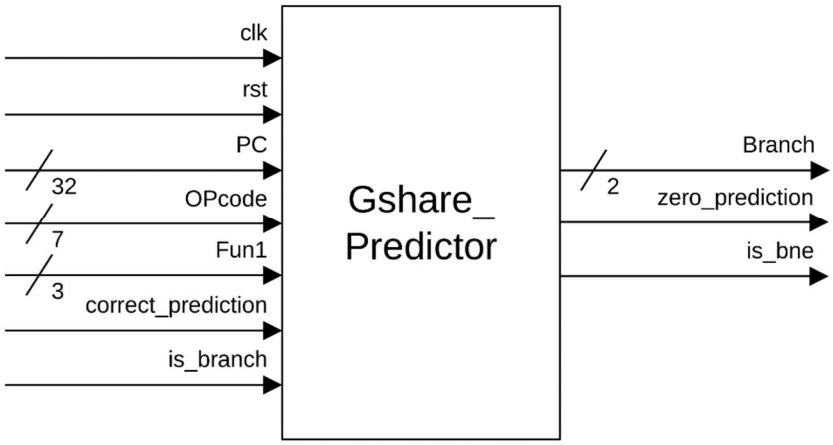


Figure 5: Branch Predictor diagram

The functioning of the Branch Predictor is quite interesting. It needs to be synchronized by the clock and at the same time update the BHR and the PHT entries based on the inputs before making a new prediction. This is achieved by updating the Branch History Register and the Pattern History Table on the rising edge of the clock, and making a new prediction on the falling edge.

The BHR has 8 bits, and it is used in combination with the PC to index the 256 entries of the PHT. We implemented a Gshare 8/8 indexing, meaning that the PHT index is the result of performing an XOR operation between the BHR and the PC. It should be noted that instead of taking the 8 least significant bits of the PC, we take PC[9:2]. That is made to extract more useful information from the PC, since the 2 least significant bits are always 0 due to it being increased in steps of 4. Each entry of the PHT contains a 2-bit saturating up/down counter.

The Branch output was previously calculated by the control unit in the ID stage based on the ID_ZERO_GENERATOR output. In the new pipeline, the Branch output is calculated by the Branch Predictor in the IF stage based on the zero-value given by the indexed predictor in the Pattern History Table. That zero_prediction is also output, along with the is_bne signal, for the reasons discussed in the last section.

3.2.3. Branch Checker

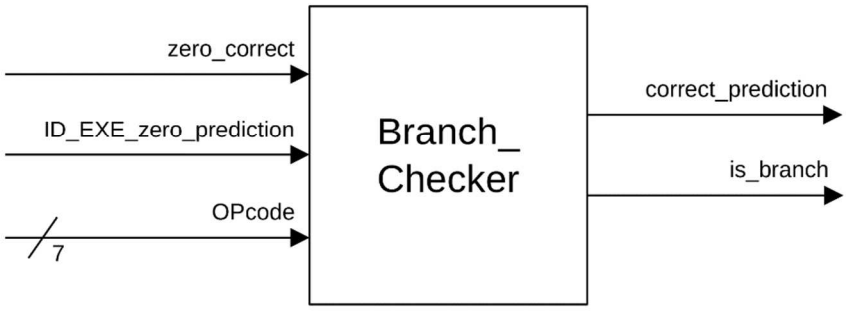


Figure 6: Branch Checker diagram

The Branch Checker is simple in its functioning and design. It receives zero_correct directly from the ALU in the EXE stage, and the ID_EXE_zero_prediction, which is the prediction made at the IF stage for the same instruction. Those are compared and if they are equal, correct_prediction is 1. Otherwise, it is 0. The Opcode is needed as input for the module to signal whether or not the checked instruction was a branch. If it was not a branch, it outputs a correct prediction regardless of the equality of the zero inputs.

3.2.4. Eliminated control stalls

Most control stalls were eliminated. All the branching instructions now generate no bubbles—except for the recovery when a misprediction is performed—, and the JAL instruction was also reworked to be resolved at the IF stage. However, the JALR instruction needs to read from a register, which is done at the ID stage in the case that no forwarding is required. Because of that, we stall the IF for one clock. To do so, we fetch the JALR instruction again. We keep track of whether a specific JALR instruction is the first of the two that need to be fetched to determine if we need to fetch it again. It should be noted that this only works if the instruction does not require data forwarding. If the register value needs to be forwarded from the EXE_MEM pipeline register or the WB stage, then we would need to add one or two more stall cycles, respectively. This was not implemented at the time of writing this report, but it does not affect the evaluations, since the benchmark codes do not have RAW dependencies on the JALR instructions.

4. Methodology

This project was developed using Verilog, with the testing being conducted on Vivado ML Edition - 2022.2. The project schedule was divided into two main cycles. In the first one, data forwarding was designed, implemented, and tested. In the second one, the same was performed for branch prediction. Each of those cycles started with a low-level diagram of the main modules and wires to implement. With that reference, the Verilog code was written. With that first implementation, the simulation on the provided testing code was run. Based on the simulation results, changes were made to the Verilog code. It was an iterative process, in which the simulation errors were fixed step-by-step. For the branch prediction implementation, a substantial redesign of the data forwarding module was required for everything to work together. As a result, more testing on the data forwarding was performed.

5. Evaluation

The evaluation was performed on the two provided benchmarks: Vec_Mul and Jacobi-1d. The assembly and hexadecimal code for both can be found in the appendices.

5.1. Data forwarding

The following simulations correspond to the Vec_Mul benchmark. The initial lines were used as a demonstration, and a color code was applied to facilitate the identification of the instructions within the simulation.

Vec_Mul.asm:

```
// Hint 1: A label looks like this:
Start:
// Hint 2: you can use '//' or '#' to make comments

// In Memory
//// Vector A: length 10
//// then Vector B: length 10
//// then Vector C: length 10
//// This program plays C = A * B

addi s1, x0, 4  # A matrix base address
addi s2, s1, 40 # B matrix base address
addi s3, s2, 40 # C matrix base address
addi s4, s3, 40 # end of C matrix

// 1. Prepare data in A
addi t3, x0, 1 # value to be initialized to vector A (1 ~ 10)
addi t1, s1, 0 # for (i = 0; ...
InitA:

sw t3, 0(t1)   # A[i] = i;

addi t1, t1, 4 # ... i += 1)
addi t3, t3, 1 # increment value
slt t2, t1, s2 # ... i < 10; ...
bne t2, x0, InitA

...
```

5.1.1. Before data forwarding

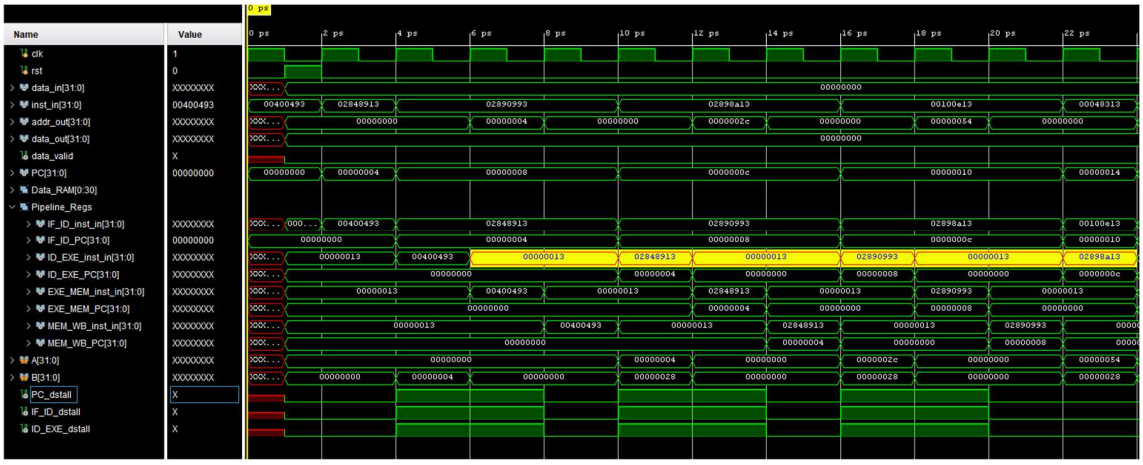


Figure 7: Simulation before implementing data forwarding

The simulation shows that before the data forwarding was implemented, the 3 instructions highlighted in yellow take 9 cycles to execute. Each instruction takes 3 cycles, two of which are stalls. The stalls can be identified by the operation code 00000013 (NOP operation).

5.1.2. After data forwarding

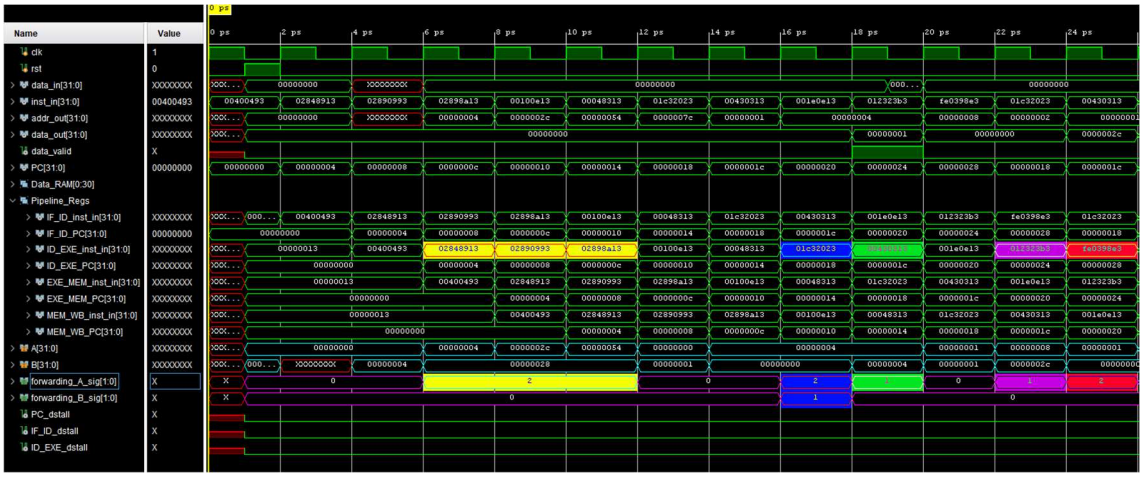


Figure 8: Simulation after implementing data forwarding

After implementing data forwarding, the simulation shows that the 3 instructions in yellow are executed in 3 cycles, each taking 1 to complete. For these instructions we get a times 3 speedup. Looking at the assembly code, all 3 instructions need to have operator A forwarded from the previous instruction. That is, the forwarding is performed from the EXE_MEM pipeline register into the ALU input. This is signaled by the value 2 in the forwarding_A_sig wire.

The instruction marked in blue forwards both operands: operand A is taken from the EXE_MEM register and operand B from the WB stage.

The instructions in green and pink forward operator A from the WB stage, and the one in red, from the EXE_MEM_ pipeline register.

5.2. Branch prediction

For the branch prediction evaluation, we will use both the Vec_Mul and Jacobi-1d benchmarks.

5.2.1. Correct branch prediction

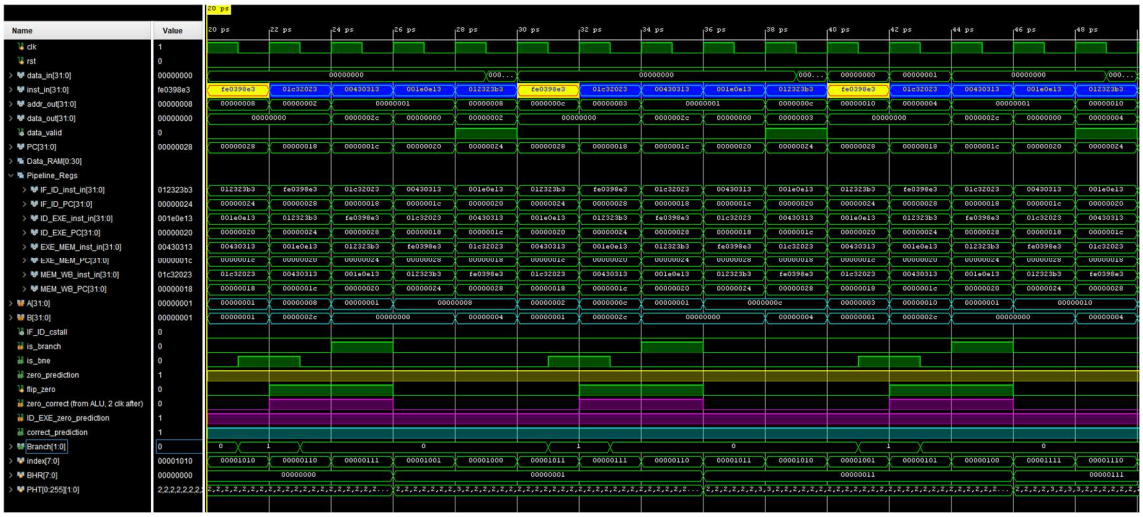


Figure 9: Simulation of a correct branch prediction

The figure above runs on the following code of Vec_Mul.asm:

```
...

// 1. Prepare data in A
addi t3, x0, 1 # value to be initialized to vector A (1 ~ 10)
addi t1, s1, 0 # for (i = 0; ...
InitA:

sw t3, 0(t1)   # A[i] = i;

addi t1, t1, 4 # ... i += 1)
addi t3, t3, 1 # increment value
slt t2, t1, s2 # ... i < 10; ...
bne t2, x0, InitA

// 2. Prepare data in B
addi t3, x0, 1 # value to be initialized to vector B (also 1 ~ 10)
addi t1, s2, 0 # for (i = 0; ...
InitB:

sw t3, 0(t1)   # B[i] = i;

addi t1, t1, 4 # ... i += 1)
addi t3, t3, 1 # increment value
slt t2, t1, s3 # ... i < 10; ...

...
```

As seen, the branch instruction—marked in yellow—is followed by the instructions inside the loop—highlighted in blue—. If we take a look at the signals in the bottom part of the wave table, we get a good insight into the inner working of the system. The first signals that change when the branch instruction is fetched are Branch and is_bne, which take the value of 1 at the immediate falling edge. Also, even though it is not noticeable

in this case, zero_prediction also updates in the same falling edge. However, since all the predictors are initialized to “10” and the prediction is made based on the MSB of that number, zero_prediction is always 1 at the beginning, which makes it appear as if the signal does not change its value. Those three signals that are generated in the falling edge of the same clock at which the branch instruction is fetched make it possible to fetch the next instruction with no stall. 2 clock cycles after the branch instruction is fetched, it reaches the ALU. There, the true result of the branch is calculated and output as zero_correct. In the same clock cycle, it is compared with ID_EXE_zero_prediction (the pink signals are compared) and correct_prediction and is_branch are determined. Those two signals are fed back to the Branch Predictor, and the PHT is updated at the next cycle.

5.2.2. Incorrect branch prediction

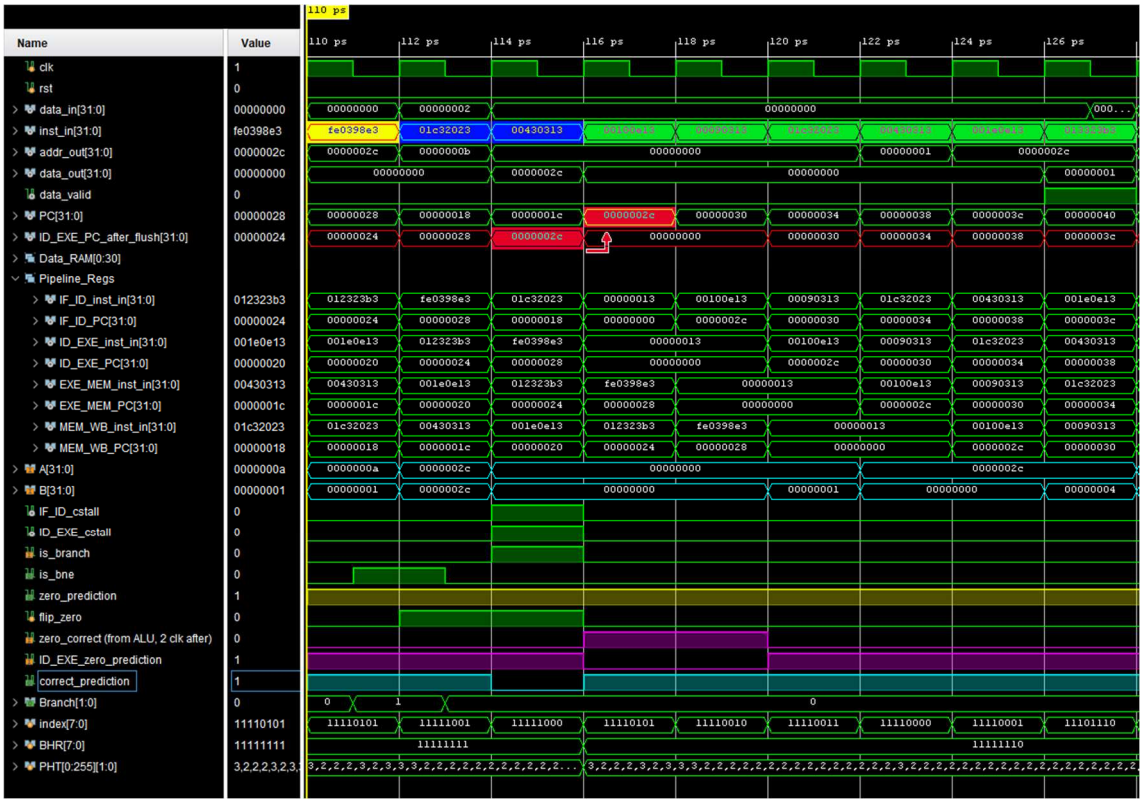


Figure 10: Simulation of a misprediction

For this next simulation, we will also highlight the next lines of code:

```
...

// 1. Prepare data in A
addi t3, x0, 1 # value to be initialized to vector A (1 ~ 10)
addi t1, s1, 0 # for (i = 0; ...
InitA:

sw t3, 0(t1)   # A[i] = i;

addi t1, t1, 4 # ... i += 1)
addi t3, t3, 1 # increment value
slt t2, t1, s2 # ... i < 10; ...
bne t2, x0, InitA

// 2. Prepare data in B
addi t3, x0, 1 # value to be initialized to vector B (also 1 ~ 10)
addi t1, s2, 0 # for (i = 0; ...
InitB:

sw t3, 0(t1)   # B[i] = i;

addi t1, t1, 4 # ... i += 1)
addi t3, t3, 1 # increment value
slt t2, t1, s3 # ... i < 10; ...
...
```

After 10 iterations of the loop, we should not branch. However, the predictor incorrectly decides that we should. As seen in the figure above, in the falling edge of the clock cycle in which the branch instruction is fetched, Branch is set to 1, indicating that the branch should be taken. With that, the first instruction of the loop is fetched, and then the second one. Those are the instructions highlighted in blue. However, while the second instruction is being fetched, the ALU is also checking whether the prediction should have been taken or not. We can appreciate that zero_correct and ID_EXE_zero_prediction are different. The latter is 0 while the former is 1. This comparison is made by the Branch Checker, and it outputs a correct_prediction of 0, while also signaling that the checked instruction was indeed a branch with is_branch being 1. When those two signals take those specific values, the Control Stall module makes IF_ID_cstall and ID_EXE_cstall take the value of 1, flushing the contents of the IF_ID and ID_EXE pipeline registers. This, in combination with the PC recovery—highlighted in red in the simulation—, allows the CPU to continue with the correct execution path. The PC is recovered thanks to ID_EXE_PC_after_flush, which is a value in the ID_EXE register that always holds the alternative branch address in case of a misprediction. After the correct PC is loaded, we can see that the execution flows through the instructions marked in green.

5.2.3. JAL instruction

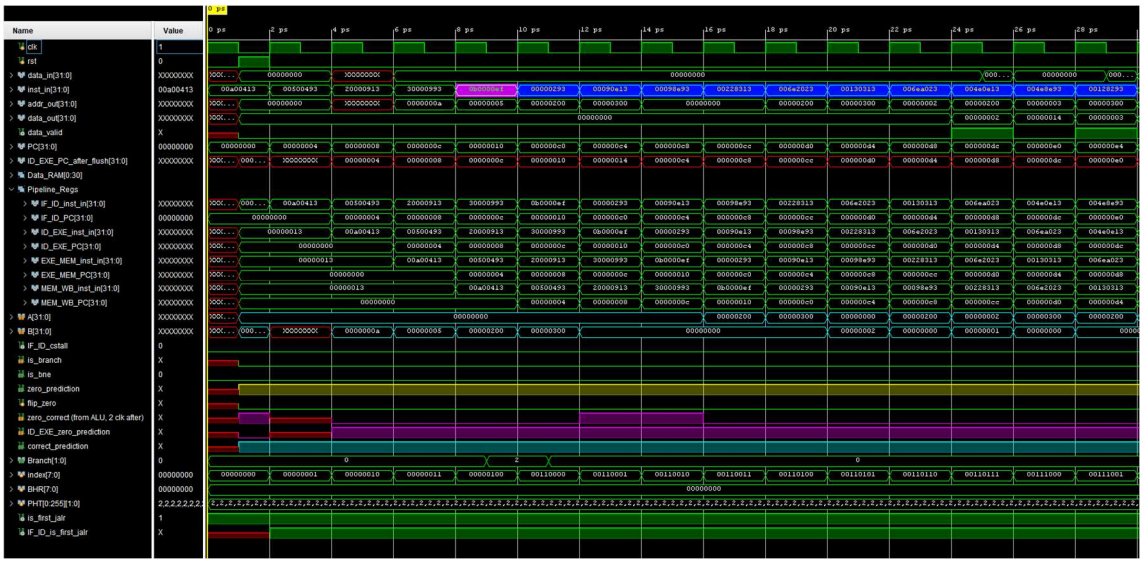


Figure 11: Simulation of a JAL instruction

For testing the JAL instruction, we will use the Jacobi-1d code:

```
main:
addi s0, x0, 10 // int n = N (N==10);
addi s1, x0, 5 // int tsteps = 5;
addi s2, x0, 0x200 // starting addr of array A
addi s3, x0, 0x300 // starting addr of array B

jal ra, init_array  // 0b0000ef

jal ra, kernel      // 008000ef

jal ra, continue    // 0d8000ef

// ----------------------------
kernel:
...
k_loop_t:
...
k_loop_i1:
...
k_loop_i2:
// ----------------------------

// ----------------------------
init_array:
addi t0, x0, 0 // t0: i         // 00000293
addi t3, s2, 0 // addr of A
addi t4, s3, 0 // addr of B
init_loop_i:
addi t1, t0, 2 // data for A[i]
sw t1, 0(t3) // A[i] = i + 2
addi t1, t1, 1 // data for B[i]
sw t1, 0(t4) // B[i] = i + 3
addi t3, t3, 4
addi t4, t4, 4
addi t0, t0, 1
bne t0, s0, init_loop_i         // fe8292e3
jalr x0, ra, 0                  // 00008067
// ----------------------------
continue:
...
```

The implementation of the JAL instruction was mostly unaltered. However, the one-cycle stall was removed, and the jump is directly performed. This is observed in the simulation. After the JAL instruction—highlighted in pink— is fetched, we continue with the code in blue, which is the one indicated in the jump instruction. The fact that the return address was correctly stored can be seen in the next simulation.

5.2.4. JALR instruction

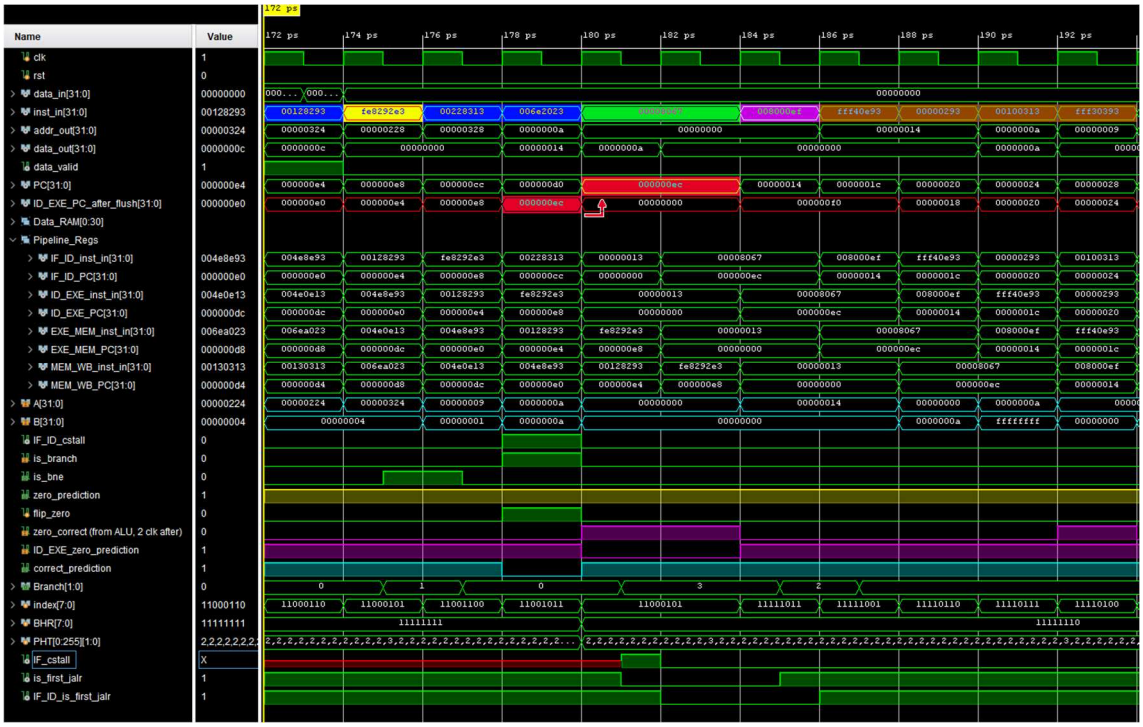


Figure 12: Simulation of a JALR instruction

For testing the JALR instruction, we will also use the Jacobi-1d code, but with some different highlighting:

```
main:
addi s0, x0, 10 // int n = N (N==10);
addi s1, x0, 5 // int tsteps = 5;
addi s2, x0, 0x200 // starting addr of array A
addi s3, x0, 0x300 // starting addr of array B

jal ra, init_array  // 0b0000ef

jal ra, kernel      // 008000ef

jal ra, continue    // 0d8000ef

// ----------------------------
kernel:
addi t4, s0, -1     // fff40e93
addi t0, x0, 0

k_loop_t:
addi t1, x0, 1

k_loop_i1:
addi t2, t1, -1
...
k_loop_i2:
```

```
// ---------------------------

// ---------------------------
init_array:
addi t0, x0, 0 // t0: i         // 00000293
addi t3, s2, 0 // addr of A
addi t4, s3, 0 // addr of B
init_loop_i:
addi t1, t0, 2 // data for A[i]
sw t1, 0(t3) // A[i] = i + 2
addi t1, t1, 1 // data for B[i]
sw t1, 0(t4) // B[i] = i + 3
addi t3, t3, 4
addi t4, t4, 4
addi t0, t0, 1
bne t0, s0, init_loop_i         // fe8292e3
jalr x0, ra, 0                  // 00008067
// ---------------------------
continue:
...
```

This simulation starts after the JAL one, once the init_loop_i—marked in blue— has been run all the intended number of times. When the branch instruction—in yellow— is encountered again, we incur in a misprediction and two of the loop instructions are fetched again. After that, the flow is corrected and the JALR instruction—in green—is fetched. When this happens, we still do not have the value of the register that needs to be read. Because of that, we fetch the Instruction Fetch stage one clock cycle (IF_cstall takes the value of 1). To prevent the pipeline from stalling indefinitely, the signals is_first_jalr and IF_ID_is_first_jalr were introduced. In combination, they ensure that only two JALR instructions in a row are fetched, and then the flow continues. When the register is read, the jump is executed. Here we can see that the return address stored by the JAL instruction was correct, since we return to the instruction highlighted in pink, which happens to be another JAL instruction, which takes the flow to the brown section.

5.3. Overall speedup

A major speedup was observed after implementing data forwarding and branch prediction. The use of a Gselect 4/4 predictor as a substitute for the Gshare 4/4 was tested, obtaining the same results, probably due to the large number of PHT entries and the loops in the benchmarks not being very complex, with little nesting. This may create a sparse PHT that does not respond to changes in how the indexing is done.

Table 1: Speedup on the different benchmarks

	Non-optimized	Optimized	Speedup
Vec_Mul	850 clk	462 clk	1.840
Jacobi-1d	3585 clk	1748 clk	2.051

6. Conclusions

In conclusion, the implementation of data forwarding and branch prediction on a single-issue the provided RISC-V32i 5-stage Pipelined CPU has resulted in a significant improvement in performance. The addition of these features has led to the reduction by half of the execution cycles in the Jacobi-1d benchmark, and a very close result for the Vec_mul one. This demonstrates the effectiveness of these techniques in improving the efficiency of pipelined processors in regard to the CPI. Further research could explore the impact of superscalar architectures on improving performance beyond the single clock per instruction. Additionally, the JALR stall insertion should be fixed for the current implementation, allowing data forwarding as discussed previously.

7. References

[1] L. Du, "RISC-V 32i CPU and Assembler," 6 May 2022. [Online]. Available: https://github.com/RipperJ/RISC-V_CPU. [Accessed 1 May 2023].

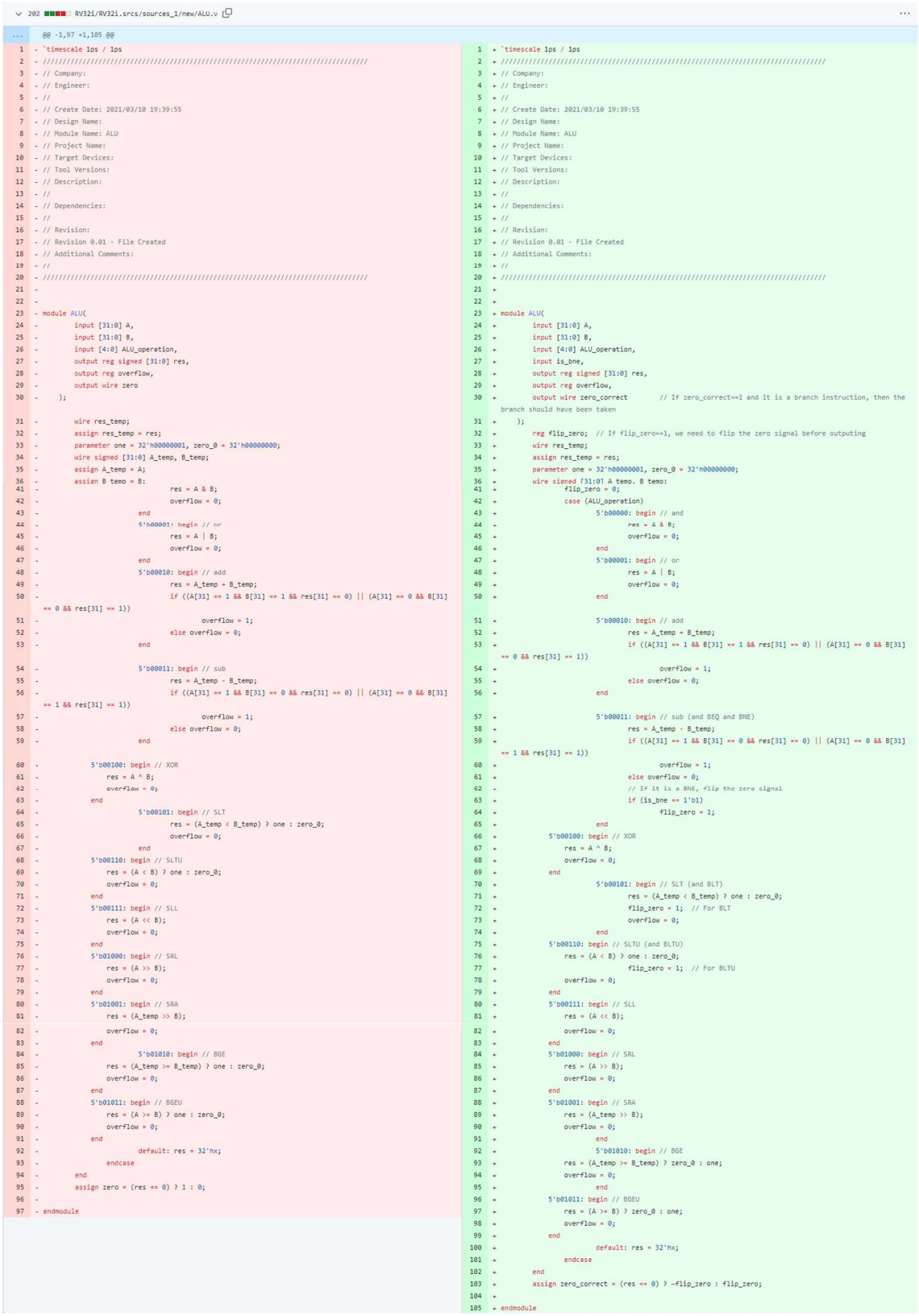
[2] W. Zhang, "GENERAL INTRODUCTION," ECE, HKUST, ELEC5140 Advanced Computer Architecture, 2023.

[3] W. Zhang, "LECTURE 4: SUPERSCALAR ARCHITECTURE INTRODUCTION AND INSTRUCTION FLOW," ECE, HKUST, ELEC5140 Advanced Computer Architecture, 2023.

[4] D. Caballero Polo, "ELEC 5140 final project: Hazard elimination on a RISC-V32i 5-stage Pipelined CPU," 23 May 2023. [Online]. Available: https://github.com/Dario-CP/ELEC-5140-project.

8. Changes in the source code

The code can be found in my GitHub repository. [4]



```
202 RV32I/RV32I.srcs/sources_1/new/ALU.v
@@ -1,97 +1,105 @@
```

```
- `timescale 1ps / 1ps
- //////////////////////////////////////////////////////////////////////////////////
- // Company:
- // Engineer:
- //
- // Create Date: 2021/03/10 19:39:55
- // Design Name:
- // Module Name: ALU
- // Project Name:
- // Target Devices:
- // Tool Versions:
- // Description:
- //
- // Dependencies:
- //
- // Revision:
- // Revision 0.01 - File Created
- // Additional Comments:
- //
- //////////////////////////////////////////////////////////////////////////////////
-
-
- module ALU(
-         input [31:0] A,
-         input [31:0] B,
-         input [4:0] ALU_operation,
-         output reg signed [31:0] res,
-         output reg overflow,
-         output wire zero
-     );

-         wire res_temp;
-         assign res_temp = res;
-         parameter one = 32'h00000001, zero_0 = 32'h00000000;
-         wire signed [31:0] A_temp, B_temp;
-         assign A_temp = A;
-         assign B_temp = B;
-                                 res = A & B;
-                                 overflow = 0;
-                         end
-                         5'b00001: begin // or
-                                 res = A | B;
-                                 overflow = 0;
-                         end
-                         5'b00010: begin // add
-                                 res = A_temp + B_temp;
-                                 if ((A[31] == 1 && B[31] == 1 && res[31] == 0) || (A[31] == 0 && B[31] == 0 && res[31] == 1))
-                                         overflow = 1;
-                                 else overflow = 0;
-                         end

-                         5'b00011: begin // sub
-                                 res = A_temp - B_temp;
-                                 if ((A[31] == 1 && B[31] == 0 && res[31] == 0) || (A[31] == 0 && B[31] == 1 && res[31] == 1))
-                                         overflow = 1;
-                                 else overflow = 0;
-                         end

-             5'b00100: begin // XOR
-                 res = A ^ B;
-                 overflow = 0;
-             end
-                         5'b00101: begin // SLT
-                                 res = (A_temp < B_temp) ? one : zero_0;
-                                 overflow = 0;
-                         end
-             5'b00110: begin // SLTU
-                 res = (A < B) ? one : zero_0;
-                 overflow = 0;
-             end
-             5'b00111: begin // SLL
-                 res = (A << B);
-                 overflow = 0;
-             end
-             5'b01000: begin // SRL
-                 res = (A >> B);
-                 overflow = 0;
-             end
-             5'b01001: begin // SRA
-                 res = (A_temp >> B);
-                 overflow = 0;
-             end
-                         5'b01010: begin // BGE
-                 res = (A_temp >= B_temp) ? one : zero_0;
-                 overflow = 0;
-             end
-             5'b01011: begin // BGEU
-                 res = (A >= B) ? one : zero_0;
-                 overflow = 0;
-             end
-                         default: res = 32'hx;
-                 endcase
-         end
-         assign zero = (res == 0) ? 1 : 0;

- endmodule
```

```
+ `timescale 1ps / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 2021/03/10 19:39:55
+ // Design Name:
+ // Module Name: ALU
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module ALU(
+         input [31:0] A,
+         input [31:0] B,
+         input [4:0] ALU_operation,
+         input is_bne,
+         output reg signed [31:0] res,
+         output reg overflow,
+         output wire zero_correct        // If zero_correct==1 and it is a branch instruction, then the branch should have been taken
+     );
+         reg flip_zero;  // If flip_zero==1, we need to flip the zero signal before outputing
+         wire res_temp;
+         assign res_temp = res;
+         parameter one = 32'h00000001, zero_0 = 32'h00000000;
+         wire signed [31:0] A_temp, B_temp;
+                 flip_zero = 0;
+                 case (ALU_operation)
+                         5'b00000: begin // and
+                                 res = A & B;
+                                 overflow = 0;
+                         end
+                         5'b00001: begin // or
+                                 res = A | B;
+                                 overflow = 0;
+                         end

+                         5'b00010: begin // add
+                                 res = A_temp + B_temp;
+                                 if ((A[31] == 1 && B[31] == 1 && res[31] == 0) || (A[31] == 0 && B[31] == 0 && res[31] == 1))
+                                         overflow = 1;
+                                 else overflow = 0;
+                         end

+                         5'b00011: begin // sub (and BEQ and BNE)
+                                 res = A_temp - B_temp;
+                                 if ((A[31] == 1 && B[31] == 0 && res[31] == 0) || (A[31] == 0 && B[31] == 1 && res[31] == 1))
+                                         overflow = 1;
+                                 else overflow = 0;
+                                 // If it is a BNE, flip the zero signal
+                                 if (is_bne == 1'b1)
+                                         flip_zero = 1;
+                         end
+             5'b00100: begin // XOR
+                 res = A ^ B;
+                 overflow = 0;
+             end
+                         5'b00101: begin // SLT (and BLT)
+                                 res = (A_temp < B_temp) ? one : zero_0;
+                                 flip_zero = 1;  // For BLT
+                                 overflow = 0;
+                         end
+             5'b00110: begin // SLTU (and BLTU)
+                 res = (A < B) ? one : zero_0;
+                                 flip_zero = 1;  // For BLTU
+                 overflow = 0;
+             end
+             5'b00111: begin // SLL
+                 res = (A << B);
+                 overflow = 0;
+             end
+             5'b01000: begin // SRL
+                 res = (A >> B);
+                 overflow = 0;
+             end
+             5'b01001: begin // SRA
+                 res = (A_temp >> B);
+                 overflow = 0;
+                         end
+                         5'b01010: begin // BGE
+                 res = (A_temp >= B_temp) ? zero_0 : one;
+                 overflow = 0;
+                         end
+             5'b01011: begin // BGEU
+                 res = (A >= B) ? zero_0 : one;
+                 overflow = 0;
+             end
+                         default: res = 32'hx;
+                 endcase
+         end
+         assign zero_correct = (res == 0) ? ~flip_zero : flip_zero;
+
+ endmodule
```

49 RV32i/RV32i.srcs/sources_1/new/Branch_Checker.v

@@ -0,0 +1,49 @@

```
+ `timescale 1ns / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 10.05.2023 22:09:27
+ // Design Name:
+ // Module Name: Branch_Checker
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module Branch_Checker(
+     // Input:
+     input zero_correct,
+     input ID_EXE_zero_prediction,
+     input [6:0] OPcode,
+
+     // Output:
+     output reg correct_prediction,
+     output reg is_branch
+     );
+
+     always @(*) begin
+         // If ID_EXE_ALU_Control is 7'b1100011, then the instruction is a branch and we need to check the prediction
+         if (OPcode == 7'b1100011) begin
+             is_branch = 1'b1;
+             if (zero_correct == ID_EXE_zero_prediction) begin
+                 correct_prediction = 1'b1;
+             end else begin
+                 correct_prediction = 1'b0;
+             end
+         end else begin
+             is_branch = 1'b0;
+             correct_prediction = 1'b1;
+         end
+     end
+
+ endmodule
```

91 RV32i/RV32i.srcs/sources_1/new/Control_Stall.v

@@ -1,33 +1,58 @@

```
- `timescale 1ps / 1ps
- //////////////////////////////////////////////////////////////////////////////////
- // Company:
- // Engineer:
- //
- // Create Date: 2021/03/12 08:45:29
- // Design Name:
- // Module Name: Control_Stall
- // Project Name:
- // Target Devices:
- // Tool Versions:
- // Description:
- //
- // Dependencies:
- //
- // Revision:
- // Revision 0.01 - File Created
- // Additional Comments:
- //
- //////////////////////////////////////////////////////////////////////////////////
-
-
- module Control_Stall(
-         input [1:0] Branch,
-         output reg IF_ID_cstall
-
-     );
-     always @ (*) begin
-         IF_ID_cstall = 1'b0;
-         if (Branch[1:0] != 2'b00) begin
-             IF_ID_cstall = 1'b1;
-         end
-     end
- endmodule
+ `timescale 1ps / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 2021/03/12 08:45:29
+ // Design Name:
+ // Module Name: Control_Stall
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module Control_Stall(
+         input [1:0] Branch,
+         input IF_ID_is_first_jalr,  // 1 if it was the first jalr out of the 2 in the bubble, 0 otherwise
+         input correct_prediction,
+         output reg IF_cstall,   ////////////////////////// ADD to main file
+         output reg IF_ID_cstall,
+         output reg ID_EXE_cstall,
+         output reg PC_cstall,    // Unused
+         output reg is_first_jalr
+     );
+     // Stalls for flushing the pipeline when the branch prediction is wrong
+     always @ (*) begin
+         IF_ID_cstall = 1'b0;
+         ID_EXE_cstall = 1'b0;
+         PC_cstall = 1'b0;
+         is_first_jalr = 1'b1;
+         // Stall IF_ID and ID_EXE if the branch prediction was wrong
+         if (correct_prediction == 1'b0) begin
+             IF_ID_cstall = 1'b1;    // 1 to stall
+             ID_EXE_cstall = 1'b1;   // 1 to stall
+         end
+
+     // Stall IF_ID if the branch was a JALR instruction
+     // We will only stall one, the second time that the JALR instruction is detected,
+         // Stall IF if the branch was a JALR instruction
+         else if (Branch == 2'b11) begin
+             if (IF_ID_is_first_jalr == 1'b1) begin
+                 IF_cstall = 1'b1;    // 1 to stall
+                 is_first_jalr = 1'b0;
+             end else begin
+                 IF_cstall = 1'b0;    // 0 to not stall
+                 is_first_jalr = 1'b0;
+             end
+         end
+     end
+ endmodule
```

409 RV32i/RV32i.srcs/sources_1/new/Controler.v

@@ -1,209 +1,200 @@

```
- `timescale 1ps / 1ps
- //////////////////////////////////////////////////////////////////////////////////
- // Company:
- // Engineer:
- //
- // Create Date: 2021/03/10 21:16:48
- // Design Name:
- // Module Name: Controler
- // Project Name:
- // Target Devices:
- // Tool Versions:
- // Description:
- //
- // Dependencies:
- //
- // Revision:
- // Revision 0.01 - File Created
- // Additional Comments:
- //
- //////////////////////////////////////////////////////////////////////////////////
-
-
- module Controler(
-                 input [6:0] OPcode,
-                 input [2:0] Fun1,
-                 input [6:0] Fun2,
-                 input wire zero,
-                 output reg ALUSrc_A,
-                 output reg [1:0] ALUSrc_B,
-                 output reg [1:0] DatatoReg,
-                 output reg [1:0] Branch,
-                 output reg RegWrite,
-                 output reg mem_w,
-                 output reg [4:0] ALU_Control,
-                 output reg [1:0] B_H_W,
-                 output reg sign
-                 );
-         always @(*) begin
-                 ALUSrc_B = 0;
-                 ALUSrc_A = 0;
-                 DatatoReg = 2'b0;
-                 DatatoReg = 2'b0;
-                 Branch = 0;
-                 RegWrite = 0;
-                 mem_w = 0;
-                 B_H_W = 2'b0; // default: immediate is a word
-                 sign = 1'b1; // default: signed extension to "write_data"
-                 case(OPcode)
-                     // R
-                         7'b0110011: begin
-                                 RegWrite = 1;
-                                 case(Fun1)
-                                     3'b000: begin
-                                         case (Fun2)
-                                             7'b0000000: ALU_Control = 5'b00010; // ADD
-                                             7'b0100000: ALU_Control = 5'b00011; // SUB
-                                             default: ALU_Control = 5'b11111;
-                                         endcase
-                                     end
-                                     3'b001: begin // SLL
-                                         ALU_Control = 5'b00111;
-                                     end
-                                     3'b010: ALU_Control = 5'b00101; // SLT
-                                     3'b011: ALU_Control = 5'b00110; // SLTU
-                                     3'b100: ALU_Control = 5'b00100; // XOR
-                                     3'b101: begin
-                                         case (Fun2)
-                                             7'b0000000: begin // SRL
-                                 ALU_Control = 5'b01000;
-                             end
-                             7'b0100000: begin // SRA
-                                 ALU_Control = 5'b01001;
-                             end
-                             default: ALU_Control = 5'b11111;
-                                         endcase
-                                     end
-                                     3'b110: ALU_Control = 5'b00001; // OR
-                                     3'b111: ALU_Control = 5'b00000; // AND
-                                         default: ALU_Control = 5'b11111;
-                                 endcase
-                         end
-                         // I
-                         7'b0010011: begin
-                             RegWrite = 1;
-                             case (Fun1)
-                                     ALU_Control = 5'b00010; // ADDI
-                         ALUSrc_B = 2'b01;
-                                 end
-                                 3'b010: begin
-                                     ALU_Control = 5'b00101; // SLTI
-                                     ALUSrc_B = 2'b01;
-                                 end
-                                 3'b011: begin
-                                     ALU_Control = 5'b00110; // SLTIU
-                         ALUSrc_B = 2'b01;
-                                 end
-                                 3'b100: begin
-                         ALU_Control = 5'b00100; // XORI
-                         ALUSrc_B = 2'b01;
-                                 end
-                                 3'b110: begin
-                         ALU_Control = 5'b00001; // ORI
-                         ALUSrc_B = 2'b01;
-                                 end
-                                 3'b111: begin
-                         ALU_Control = 5'b00000; // ANDI
-                         ALUSrc_B = 2'b01;
-                                 end
-                                 3'b001: begin
-                                     ALU_Control = 5'b00111; // SLLI
-                         ALUSrc_B = 2'b01;
-                                 end
-                                 3'b101: begin
-                                     ALUSrc_B = 2'b01;
-                                     case (Fun2)
-                                         7'b0000000: ALU_Control = 5'b01000; // SRLI
-                                         7'b0100000: ALU_Control = 5'b01001; // SRAI
-                                     endcase
-                                 end
-                             endcase
-
+ `timescale 1ps / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 2021/03/10 21:16:48
+ // Design Name:
+ // Module Name: Controler
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module Controler(
+                 input [6:0] OPcode,
+                 input [2:0] Fun1,
+                 input [6:0] Fun2,
+                 output reg ALUSrc_A,
+                 output reg [1:0] ALUSrc_B,
+                 output reg [1:0] DatatoReg,
+                 output reg RegWrite,
+                 output reg mem_w,
+                 output reg mem_r,
+                 output reg [4:0] ALU_Control,
+                 output reg [1:0] B_H_W,
+                 output reg sign
+                 );
+         always @(*) begin
+                 ALUSrc_B = 0;
+                 ALUSrc_A = 0;
+                 DatatoReg = 2'b0;
+                 RegWrite = 0;
+                 RegWrite = 0;
+                 mem_w = 0;
+                 mem_r = 0;
+                 B_H_W = 2'b0; // default: immediate is a word
+                 sign = 1'b1; // default: signed extension to "write_data"
+                 case(OPcode)
+                     // R
+                         7'b0110011: begin
+                                 RegWrite = 1;
+                                 case(Fun1)
+                                     3'b000: begin
+                                         case (Fun2)
+                                             7'b0000000: ALU_Control = 5'b00010; // ADD
+                                             7'b0100000: ALU_Control = 5'b00011; // SUB
+                                             default: ALU_Control = 5'b11111;
+                                         endcase
+                                     end
+                                     3'b001: begin // SLL
+                                         ALU_Control = 5'b00111;
+                                     end
+                                     3'b010: ALU_Control = 5'b00101; // SLT
+                                     3'b011: ALU_Control = 5'b00110; // SLTU
+                                     3'b100: ALU_Control = 5'b00100; // XOR
+                                     3'b101: begin
+                                         case (Fun2)
+                                             7'b0000000: begin // SRL
+                                 ALU_Control = 5'b01000;
+                             end
+                             7'b0100000: begin // SRA
+                                 ALU_Control = 5'b01001;
+                             end
+                             default: ALU_Control = 5'b11111;
+                                         endcase
+                                     end
+                                     3'b110: ALU_Control = 5'b00001; // OR
+                                     3'b111: ALU_Control = 5'b00000; // AND
+                                         default: ALU_Control = 5'b11111;
+                                 endcase
+                         end
+                         // I
+                         7'b0010011: begin
+                             RegWrite = 1;
+                             case (Fun1)
+                                 3'b000: begin
+                         ALUSrc_B = 2'b01;
+                                 end
+                                 3'b010: begin
+                                     ALU_Control = 5'b00101; // SLTI
+                                     ALUSrc_B = 2'b01;
+                                 end
+                                 3'b011: begin
+                                     ALU_Control = 5'b00110; // SLTIU
+                         ALUSrc_B = 2'b01;
+                                 end
+                                 3'b100: begin
+                         ALU_Control = 5'b00100; // XORI
+                         ALUSrc_B = 2'b01;
+                                 end
+                                 3'b110: begin
+                         ALU_Control = 5'b00001; // ORI
+                         ALUSrc_B = 2'b01;
+                                 end
+                                 3'b111: begin
+                         ALU_Control = 5'b00000; // ANDI
+                         ALUSrc_B = 2'b01;
+                                 end
+                                 3'b001: begin
+                                     ALU_Control = 5'b00111; // SLLI
+                         ALUSrc_B = 2'b01;
+                                 end
+                                 3'b101: begin
+                                     ALUSrc_B = 2'b01;
+                                     case (Fun2)
+                                         7'b0000000: ALU_Control = 5'b01000; // SRLI
+                                         7'b0100000: ALU_Control = 5'b01001; // SRAI
+                                     endcase
+                                 end
+                             endcase
+
```

```
-                       end
-                       7'b0000011: begin       // l
-                               ALU_Control = 5'b00010;
-                               ALUSrc_B = 2'b01;
-                               DatatoReg = 2'b01;
-                               RegWrite = 1;
-                               case (Fun1)
-                                   3'b000: begin // LB
-                                       B_H_W = 2'b01; // byte
-                                   end
-                                   3'b001: begin // LH
-                                       B_H_W = 2'b10; // half word
-                                   end
-                                   3'b100: begin // LBU
-                                       B_H_W = 2'b01; // byte
-                                       sign = 1'b0;
-                                   end
-                                   3'b101: begin // LHU
-                                       B_H_W = 2'b10; // half word
-                                       sign = 1'b0;
-                                   end
-                                   // 3'b010:; // LW
-                               endcase
-                       end
-                       7'b0100011: begin   // S
-                           ALU_Control = 5'b00010;
-                           ALUSrc_B = 2'b01;
-                           mem_w = 1;
-                           case (Fun1)
-                               3'b000: begin
-                                   B_H_W = 2'b01; // byte
-                               end
-                               3'b001: begin
-                                   B_H_W = 2'b10; // half word
-                               end
-                               // 3'b010: ; // SW
-                           endcase
-                       end
-                       7'b1100011: begin       // Branch
-                           case (Fun1)
-                               3'b000: begin // BEQ
-                                   ALU_Control = 5'b00011;
-                                   Branch = {1'b0, zero};
-                               end
-                               3'b001: begin // BNE
-                                   ALU_Control = 5'b00011;
-                                   Branch = {1'b0, ~zero};
-                               end
-                               3'b100: begin // BLT
-                                   ALU_Control = 5'b00101;
-                                   Branch = {1'b0, zero};
-                               end
-                               3'b101: begin // BGE
-                                   ALU_Control = 5'b01010;
-                                   Branch = {1'b0, zero};
-                               end
-                               3'b110: begin // BLTU
-                                   ALU_Control = 5'b00110;
-                                   Branch = {1'b0, zero};
-                               end
-                               3'b111: begin // BGEU
-                                   ALU_Control = 5'b01011;
-                                   Branch = {1'b0, zero};
-                               end
-                           endcase
-                       end
-                       7'b1101111: begin       // jal
-                               Branch = 2'b10;
-                               DatatoReg = 2'b11;
-                               RegWrite = 1;
-                       end
-           7'b1100111: begin   // jalr
-               Branch = 2'b11;
-               DatatoReg = 2'b11;
-               RegWrite = 1;
-           end
-           7'b0110111: begin    // lui
-               DatatoReg = 2'b10;
-               RegWrite = 1;
-           end
-           7'b0010111: begin   // AUIPC
-               DatatoReg = 2'b10;
-               RegWrite = 1;
-           end
-                       default: ALU_Control = 5'b11111;
-               endcase
-       end
- endmodule
-
```

```
+                       7'b0000011: begin       // l
+                               ALU_Control = 5'b00010;
+                               ALUSrc_B = 2'b01;
+                               DatatoReg = 2'b01;
+                               RegWrite = 1;
+                               mem_r = 1;
+                               case (Fun1)
+                                   3'b000: begin // LB
+                                       B_H_W = 2'b01; // byte
+                                   end
+                                   3'b001: begin // LH
+                                       B_H_W = 2'b10; // half word
+                                   end
+                                   3'b100: begin // LBU
+                                       B_H_W = 2'b01; // byte
+                                       sign = 1'b0;
+                                   end
+                                   3'b101: begin // LHU
+                                       B_H_W = 2'b10; // half word
+                                       sign = 1'b0;
+                                   end
+                                   // 3'b010:; // LW
+                               endcase
+                       end
+                       7'b0100011: begin   // S
+                           ALU_Control = 5'b00010;
+                           ALUSrc_B = 2'b01;
+                           mem_w = 1;
+                           case (Fun1)
+                               3'b000: begin
+                                   B_H_W = 2'b01; // byte
+                               end
+                               3'b001: begin
+                                   B_H_W = 2'b10; // half word
+                               end
+                               // 3'b010: ; // SW
+                           endcase
+                       end
+                       7'b1100011: begin       // Branch
+                           case (Fun1)
+                               3'b000: begin // BEQ
+                                   ALU_Control = 5'b00011;
+                               end
+                               3'b001: begin // BNE
+                                   ALU_Control = 5'b00011;
+                               end
+                               3'b100: begin // BLT
+                                   ALU_Control = 5'b00101;
+                               end
+                               3'b101: begin // BGE
+                                   ALU_Control = 5'b01010;
+                               end
+                               3'b110: begin // BLTU
+                                   ALU_Control = 5'b00110;
+                               end
+                               3'b111: begin // BGEU
+                                   ALU_Control = 5'b01011;
+                               end
+                           endcase
+                       end
+                       7'b1101111: begin       // jal
+                               DatatoReg = 2'b11;
+                               RegWrite = 1;
+                       end
+           7'b1100111: begin   // jalr
+               DatatoReg = 2'b11;
+               RegWrite = 1;
+           end
+           7'b0110111: begin    // lui
+               DatatoReg = 2'b10;
+               RegWrite = 1;
+           end
+           7'b0010111: begin   // AUIPC
+               DatatoReg = 2'b10;
+               RegWrite = 1;
+           end
+                       default: ALU_Control = 5'b11111;
+               endcase
+       end
+ endmodule
```

123 RV32i/RV32i.srcs/sources_1/new/Data_Stall.v

```
@@ -1,53 +1,70 @@
- `timescale 1ps / 1ps
- //////////////////////////////////////////////////////////////////////////////////
- // Company:
- // Engineer:
- //
- // Create Date: 2021/03/12 08:46:15
- // Design Name:
- // Module Name: Data_Stall
- // Project Name:
- // Target Devices:
- // Tool Versions:
- // Description:
- //
- // Dependencies:
- //
- // Revision:
- // Revision 0.01 - File Created
- // Additional Comments:
- //
- //////////////////////////////////////////////////////////////////////////////////
-
-
- module Data_Stall(
-         input [4:0] IF_ID_written_reg,
-         input [4:0] IF_ID_read_reg1,
-         input [4:0] IF_ID_read_reg2,
-         input [4:0] ID_EXE_written_reg,
-         input [4:0] ID_EXE_read_reg1,
-         input [4:0] ID_EXE_read_reg2,
-         input [4:0] EXE_MEM_written_reg,
-         input [4:0] EXE_MEM_read_reg1,
-         input [4:0] EXE_MEM_read_reg2,
-
-         output reg PC_dstall,
-         output reg IF_ID_dstall,
-         output reg ID_EXE_dstall
-     );
-     always @ (*) begin
-         PC_dstall = 0;
-         IF_ID_dstall = 0;
-         ID_EXE_dstall = 0;
-         if (ID_EXE_written_reg != 0 && (ID_EXE_written_reg == IF_ID_read_reg1 || ID_EXE_written_reg == IF_ID_read_reg2)) begin
-                 PC_dstall = 1;
-                 IF_ID_dstall = 1;
-                 ID_EXE_dstall = 1;
-         end
-         else if (EXE_MEM_written_reg != 0 && (EXE_MEM_written_reg == IF_ID_read_reg1 || EXE_MEM_written_reg == IF_ID_read_reg2)) begin
-                 PC_dstall = 1;
-                 IF_ID_dstall = 1;
-                 ID_EXE_dstall = 1;
-         end
-     end
- endmodule
+ `timescale 1ps / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 2021/03/12 08:46:15
+ // Design Name:
+ // Module Name: Data_Stall
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module Data_Stall(
+         /*
+         Load-Use Hazard Detection
+
+         Check when using instruction is decoded in ID stage
+
+         ALU operand register numbers in ID stage are given by
+                 * IF/ID.RegisterRs, IF/ID.RegisterRt
+
+         Load-use hazard when
+                 * ID/EX.MemRead and
+                 ((ID/EX.RegisterRd = IF/ID.RegisterRs) or
+                 (ID/EX.RegisterRd = IF/ID.RegisterRt))
+
+         If detected, stall and insert bubble
+
+         REFERENCE:
+         "ELEC5140
+         Advanced Computer Architecture
+         Prof. Wei Zhang, ECE, HKUST
+
+         GENERAL INTRODUCTION"
+         */
+
+         // Input:
+         input ID_EXE_mem_r,
+
+         input [4:0] ID_EXE_written_reg,
+
+         input [4:0] IF_ID_read_reg1,
+         input [4:0] IF_ID_read_reg2,
+
+         // Output:
+         output reg PC_dstall,
+         output reg IF_ID_dstall,
+         output reg ID_EXE_dstall
+     );
+     always @ (*) begin
+         PC_dstall = 0;
+         IF_ID_dstall = 0;
+         ID_EXE_dstall = 0;
+         if (ID_EXE_mem_r
+         && ((ID_EXE_written_reg == IF_ID_read_reg1)
+         || (ID_EXE_written_reg == IF_ID_read_reg2))) begin
+             PC_dstall = 1;
+             IF_ID_dstall = 1;
+             ID_EXE_dstall = 1;
+         end
+     end
+ endmodule
```

132 RV32i/RV32i.srcs/sources_1/new/Forward_Unit.v

```
@@ -0,0 +1,132 @@
+ `timescale 1ps / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 2023/05/01 14:34:39
+ // Design Name:
+ // Module Name: Forward_Unit
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description: Forwarding unit to perform data forwarding
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+ module Forward_Unit(
+     /*
+     Check whether the Destination registers written by current instruction
+     match the Source registers read by next instructions:
+         ID/EX.WriteRegister (Rd) = IF/ID.ReadRegister1 (Rs)
+         ID/EX.WriteRegister (Rd) = IF/ID.ReadRegister2 (Rt)
+         EX/MEM.WriteRegister = IF/ID.ReadRegister1
+         EX/MEM.WriteRegister = IF/ID.ReadRegister2
+         MEM/WB.WriteRegister = IF/ID.ReadRegister1
+         MEM/WB.WriteRegister = IF/ID.ReadRegister2
+
+     EX hazard
+       // Rd = Rs
+       * if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
+         and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
+             Forward EX/MEM.RegisterRd to ID/EX.RegisterRs
+
+       // Rd = Rt
+       * if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
+         and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
```

```
+            Forward EX/MEM.RegisterRd to ID/EX.RegisterRt
+
+    MEM hazard
+      // Rd = Rs
+      * if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
+        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))   // not EX hazard
+        and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
+            Forward MEM/WB.RegisterRd to ID/EX.RegisterRs
+
+      // Rd = Rt
+      * if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
+        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))   // not EX hazard
+        and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
+            Forward MEM/WB.RegisterRd to ID/EX.RegisterRt
+
+    Note: The "// not EX hazard" line is added to prevent the MEM hazard from
+    forwarding an outdated value from the MEM/WB pipeline register when there
+    is an EX hazard. This is because the EX stage is executed before the MEM,
+    and a newer instruction may have written to the register in the EX stage.
+
+    REFERENCE:
+    "ELEC5140
+    Advanced Computer Architecture
+    Prof. Wei Zhang, ECE, HKUST
+    GENERAL INTRODUCTION"
+
+    */
+        // Input:
+        input [4:0] ID_EXE_read_reg1,       // ID/EXE.ReadRegister1
+        input [4:0] ID_EXE_read_reg2,       // ID/EXE.ReadRegister2
+
+        input EXE_MEM_RegWrite,             // EXE/MEM.RegWrite
+        input [4:0] EXE_MEM_written_reg,    // EXE/MEM.WriteRegister
+
+        input MEM_WB_RegWrite,              // MEM/WB.RegWrite
+        input [4:0] MEM_WB_written_reg,     // MEM/WB.WriteRegister
+
+        // Output:
+        output reg [1:0] forwarding_A_sig,  // forwarding signal for ALU input A (forwarding_A_sig = EXE_forwarding_A | MEM_forwarding_A)
+        output reg [1:0] forwarding_B_sig  // forwarding signal for ALU input B (forwarding_B_sig = EXE_forwarding_B | MEM_forwarding_B)
+    );
+
+    always @(*) begin
+        // Initialize the forwarding signals
+        forwarding_A_sig = 2'b00;
+        forwarding_B_sig = 2'b00;
+
+        // EX hazard
+        // Rd = Rs
+        if ((EXE_MEM_RegWrite && EXE_MEM_written_reg != 0)
+        && (EXE_MEM_written_reg == ID_EXE_read_reg1)) begin
+            // Note: 1'b1 means a binary value of 1 (the 1 on the right of the b)
+            // with a width of 1 bit (the 1 on the left of the b)
+            forwarding_A_sig[1] = 1'b1;   // Signal 1 to forward
+        end
+        else begin
+            forwarding_A_sig[1] = 1'b0;   // Signal 0 to not forward
+        end
+
+        // Rd = Rt
+        if ((EXE_MEM_RegWrite && EXE_MEM_written_reg != 0)
+        && (EXE_MEM_written_reg == ID_EXE_read_reg2)) begin
+            forwarding_B_sig[1] = 1'b1;   // Signal 1 to forward
+        end
+        else begin
+            forwarding_B_sig[1] = 1'b0;   // Signal 0 to not forward
+        end
+
+        // MEM hazard (only if there is no EX hazard)
+        // Rd = Rs
+        if ((MEM_WB_RegWrite && MEM_WB_written_reg != 0)
+        && (!(EXE_MEM_written_reg != 0 && (EXE_MEM_written_reg == ID_EXE_read_reg1)))   // not EX hazard
+        && (MEM_WB_written_reg == ID_EXE_read_reg1)) begin
+            forwarding_A_sig[0] = 1'b1;   // Signal 1 to forward
+        end
+        else begin
+            forwarding_A_sig[0] = 1'b0;   // Signal 0 to not forward
+        end
+
+        // Rd = Rt
+        if ((MEM_WB_RegWrite && MEM_WB_written_reg != 0)
+        && (!(EXE_MEM_written_reg != 0 && (EXE_MEM_written_reg == ID_EXE_read_reg2)))   // not EX hazard
+        && (MEM_WB_written_reg == ID_EXE_read_reg2)) begin
+            forwarding_B_sig[0] = 1'b1;   // Signal 1 to forward
+        end
+        else begin
+            forwarding_B_sig[0] = 1'b0;   // Signal 0 to not forward
+        end
+
+    end
+endmodule
```

156 RV32i/RV32i.srcs/sources_1/new/Gshare_Predictor.v

@@ -0,0 +1,156 @@

```
+ `timescale 1ns / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 08.05.2023 19:33:16
+ // Design Name:
+ // Module Name: Gshare_Predictor
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module Gshare_Predictor(
+     /*
+     Gshare 8/8 two-level branch predictor
+     8 bits BHR
+     256 entries PHT
+     2 bits saturating up/down predictors per entry
+
+     PHT indexed by PC[9:2] XOR BHR[7:0]
+     */
+
+     // Input:
+     input clk,
+     input rst,
+     input [31:0] PC,
+     input [6:0] OPcode,
+     input [2:0] Fun1,
+     input correct_prediction,   // 1 if last prediction was correct, 0 if not
+     input is_branch,            // 1 if last instruction was a branch, 0 if not
+
+     // Output:
+     output reg [1:0] Branch,
+     output reg zero_prediction, // If zero_prediction==1, then the branch is taken (except for BNE)
+     output reg is_bne
+     );
+
+     // BHR
+     reg [7:0] BHR = 8'b00000000;
+     // PHT
+     reg [1:0] PHT [0:255];
+     integer i;
+     initial begin   // Initialize the 256 entries of PHT to 2'b10
+         for (i = 0; i < 256; i = i + 1) begin
+             PHT[i] = 2'b10;
+         end
+     end
+
+     // Indexing
+     reg [7:0] index = 8'b00000000;
+     always @ (*) begin
+         index = {PC[9:2]} ^ {BHR[7:0]}; // PC[9:2] XOR BHR[7:0]
+
+         // TEST: index by GSELECT (concatenate PC[5:2] and BHR[3:0])
+         // index = {PC[5:2], BHR[3:0]};
+     end
+
+     // Update BHR (on rising edge of clk)
+     always @ (posedge clk or posedge rst) begin
+         if (rst == 1) begin
+             BHR = 8'b00000000;     // Reset back to 0
+         end
+         else begin
+             // If last instruction was a branch, shift BHR left by 1 and add correct prediction to LSB
+             if (is_branch == 1) begin
+                 BHR = {BHR[6:0], correct_prediction};
+             end
+         end
+     end
+
+     // Make new prediction (on falling edge of clk)
+     always @ (negedge clk) begin
+         // Initialize Branch to 00
+         Branch = 0;
+         // Put MSB of PHT[index] into zero_prediction
+         zero_prediction = PHT[index][1];
```

```
+
+        is_bne = 0;
+
+        // Calculate Branch (Previously done in Control Unit)
+        case (OPcode)
+            // Branch instructions
+            7'b1100011: begin   // Branch
+                case (Fun1)
+                    3'b000: begin // BEQ
+                        Branch = {1'b0, zero_prediction};
+                    end
+                    3'b001: begin // BNE
+                        Branch = {1'b0, zero_prediction};
+                        // Indicate that this is a BNE instruction
+                        // So that the zero_correct in the ALU is flipped
+                        is_bne = 1;
+                    end
+                    3'b100: begin // BLT
+                        Branch = {1'b0, zero_prediction};
+                    end
+                    3'b101: begin // BGE
+                        Branch = {1'b0, zero_prediction};
+                    end
+                    3'b110: begin // BLTU
+                        Branch = {1'b0, zero_prediction};
+                    end
+                    3'b111: begin // BGEU
+                        Branch = {1'b0, zero_prediction};
+                    end
+                endcase
+            end
+
+            // Jump instructions
+            7'b1101111: begin   // jal
+                                Branch = 2'b10;
+                        end
+            7'b1100111: begin   // jalr
+                Branch = 2'b11;
+            end
+        endcase
+    end
+
+    // Update PHT with the last prediction (on rising edge of clk)
+    always @ (posedge clk or posedge rst) begin
+        if (rst == 1) begin
+            // Reset back to weakly taken all entries
-           for (i = 0; i < 256; i = i + 1) begin
+                PHT[i] = 2'b10;
+            end
+        end
+        else begin
+            // If last instruction was a branch, update PHT
+            if (is_branch == 1) begin
+                // If last prediction was correct, increment PHT (up to 11)
+                if (correct_prediction == 1) begin
+                    if (PHT[index] < 3) begin
+                        PHT[index] = PHT[index] + 1;
+                    end
+                end
+                // Else, decrement PHT (down to 00)
+                else begin
+                    if (PHT[index] > 0) begin
+                        PHT[index] = PHT[index] - 1;
+                    end
+                end
+            end
+        end
+    end
+
+endmodule
```

81 RV32i/RV32i.srcs/sources_1/new/REG32.v

```
...  @@ -1,40 +1,41 @@
-  `timescale 1ps / 1ps
-  //////////////////////////////////////////////////////////////////////////////////
-  // Company:
-  // Engineer:
-  //
-  // Create Date: 2021/03/10 19:39:55
-  // Design Name:
-  // Module Name: REG32
-  // Project Name:
-  // Target Devices:
-  // Tool Versions:
-  // Description:
-  //
-  // Dependencies:
-  //
-  // Revision:
-  // Revision 0.01 - File Created
-  // Additional Comments:
-  //
-  //////////////////////////////////////////////////////////////////////////////////
-
-
-  module REG32(
-      input clk,
-      input rst,
-      input CE,
-      input [31:0] D,
-      output reg [31:0] Q = 0,
-      input PC_dstall
-      );
-
-      always @ (posedge clk or posedge rst) begin
-              if (rst == 1) Q <= 32'h00000000;
-              if (PC_dstall == 0) begin
-                      if (rst == 1) Q <= 32'h00000000;
-                      else if (CE) Q <= D;
-                  end
-      end
-
-  endmodule
+  `timescale 1ps / 1ps
+  //////////////////////////////////////////////////////////////////////////////////
+  // Company:
+  // Engineer:
+  //
+  // Create Date: 2021/03/10 19:39:55
+  // Design Name:
+  // Module Name: REG32
+  // Project Name:
+  // Target Devices:
+  // Tool Versions:
+  // Description:
+  //
+  // Dependencies:
+  //
+  // Revision:
+  // Revision 0.01 - File Created
+  // Additional Comments:
+  //
+  //////////////////////////////////////////////////////////////////////////////////
+
+
+  module REG32(
+      input clk,
+      input rst,
+      input CE,
+      input [31:0] D,
+      output reg [31:0] Q = 0,
+      input PC_dstall,
+      input PC_cstall
+      );
+
+      always @ (posedge clk or posedge rst) begin
+              if (rst == 1) Q <= 32'h00000000;
+              if (PC_dstall == 0 && PC_cstall == 0) begin // Added PC_cstall
+                      if (rst == 1) Q <= 32'h00000000;
+                      else if (CE) Q <= D;
+                  end
+      end
+
+  endmodule
```

216 RV32I/RV32I.srcs/sources_1/new/REG_ID_EXE.v

@@ -1,90 +1,126 @@

```
- `timescale 1ps / 1ps
- //////////////////////////////////////////////////////////////////////////////////
- // Company:
- // Engineer:
- //
- // Create Date: 2021/03/11 22:00:28
- // Design Name:
- // Module Name: REG_ID_EXE
- // Project Name:
- // Target Devices:
- // Tool Versions:
- // Description:
- //
- // Dependencies:
- //
- // Revision:
- // Revision 0.01 - File Created
- // Additional Comments:
- //
- //////////////////////////////////////////////////////////////////////////////////
-
-
- module REG_ID_EXE(
-         input clk,
-         input rst,
-         input CE,
-         input ID_EXE_dstall,
-
-         input [31:0] inst_in,
-         input [31:0] PC,
-         input [31:0] ALU_A,
-         input [31:0] ALU_B,
-         input [4:0] ALU_Control,
-         input [31:0] Data_out,
-         input mem_w,
-         input [1:0] DatatoReg,
-         input RegWrite,
-
-         input [4:0] written_reg,
-         input [4:0] read_reg1,
-         input [4:0] read_reg2,
-
-         output reg [31:0] ID_EXE_inst_in,
-         output reg [31:0] ID_EXE_PC = 0,
-         output reg [31:0] ID_EXE_ALU_A,
-         output reg [31:0] ID_EXE_ALU_B,
-         output reg [4:0] ID_EXE_ALU_Control,
-         output reg [31:0] ID_EXE_Data_out,
-         output reg ID_EXE_mem_w,
-         output reg [1:0] ID_EXE_DatatoReg,
-         output reg ID_EXE_RegWrite,
-
-         output reg [4:0] ID_EXE_written_reg,
-         output reg [4:0] ID_EXE_read_reg1,
-         output reg [4:0] ID_EXE_read_reg2
-     );
-
-     always @ (posedge clk or posedge rst) begin
-         if (rst == 1 || ID_EXE_dstall == 1) begin
-             ID_EXE_inst_in      <= 32'h00000013;
-             ID_EXE_PC           <= 32'h00000000;
-             ID_EXE_ALU_A        <= 32'h00000000;
-             ID_EXE_ALU_B        <= 32'h00000000;
-             ID_EXE_ALU_Control  <= 5'b00000;
-             ID_EXE_Data_out     <= 32'h00000000;
-             ID_EXE_mem_w        <= 1'b0;
-             ID_EXE_DatatoReg    <= 2'b00;
-             ID_EXE_RegWrite     <= 1'b0;
-
-             ID_EXE_written_reg  <= 5'b00000;
-             ID_EXE_read_reg1    <= 5'b00000;
-             ID_EXE_read_reg2    <= 5'b00000;
-         end
-         else if (CE) begin
-             ID_EXE_inst_in      <= inst_in;
-             ID_EXE_PC           <= PC;
-             ID_EXE_ALU_A        <= ALU_A;
-             ID_EXE_ALU_B        <= ALU_B;
-             ID_EXE_ALU_Control  <= ALU_Control;
-             ID_EXE_Data_out     <= Data_out;
-             ID_EXE_mem_w        <= mem_w;
-             ID_EXE_DatatoReg    <= DatatoReg;
-             ID_EXE_RegWrite     <= RegWrite;
-
-             ID_EXE_written_reg  <= written_reg;
-             ID_EXE_read_reg1    <= read_reg1;
+ `timescale 1ps / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 2021/03/11 22:00:28
+ // Design Name:
+ // Module Name: REG_ID_EXE
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module REG_ID_EXE(
+         input clk,
+         input rst,
+         input CE,
+         input ID_EXE_dstall,
+         input ID_EXE_cstall,
+
+         input [31:0] inst_in,
+         input [31:0] PC,
+         input [31:0] Imm_32,
+         input [31:0] ALU_A,
+         input [31:0] ALU_B,
+         input ALUSrc_A,
+         input [1:0] ALUSrc_B,
+         input [4:0] ALU_Control,
+         input [31:0] Data_out,
+         input [31:0] add_branch_out,
+         input mem_w,
+         input mem_r,
+         input [1:0] DatatoReg,
+         input RegWrite,
+         input zero_prediction,
+         input is_bne,
+
+         input [4:0] written_reg,
+         input [4:0] read_reg1,
+         input [4:0] read_reg2,
+
+         output reg [31:0] ID_EXE_inst_in,
+         output reg [31:0] ID_EXE_PC = 0,
+         output reg [31:0] ID_EXE_Imm_32,
+         output reg [31:0] ID_EXE_ALU_A,
+         output reg [31:0] ID_EXE_ALU_B,
+         output reg ID_EXE_ALUSrc_A,
+         output reg [1:0] ID_EXE_ALUSrc_B,
+         output reg [4:0] ID_EXE_ALU_Control,
+         output reg [31:0] ID_EXE_Data_out,
+         output reg ID_EXE_mem_w,
+         output reg ID_EXE_mem_r,
+         output reg [1:0] ID_EXE_DatatoReg,
+         output reg ID_EXE_RegWrite,
+         output reg ID_EXE_zero_prediction,
+         output reg [31:0] ID_EXE_PC_after_flush,
+         output reg ID_EXE_is_bne,
+
+         output reg [4:0] ID_EXE_written_reg,
+         output reg [4:0] ID_EXE_read_reg1,
+         output reg [4:0] ID_EXE_read_reg2
+     );
+
+     always @ (posedge clk or posedge rst) begin
+         if (rst == 1 || ID_EXE_dstall == 1 || ID_EXE_cstall == 1) begin
+             ID_EXE_inst_in      <= 32'h00000013;    // nop
+             ID_EXE_PC           <= 32'h00000000;
+             ID_EXE_Imm_32       <= 32'h00000000;
+             ID_EXE_ALU_A        <= 32'h00000000;
+             ID_EXE_ALU_B        <= 32'h00000000;
+             ID_EXE_ALUSrc_A     <= 1'b0;
+             ID_EXE_ALUSrc_B     <= 2'b00;
+             ID_EXE_ALU_Control  <= 5'b00000;
+             ID_EXE_Data_out     <= 32'h00000000;
+             ID_EXE_mem_w        <= 1'b0;
+             ID_EXE_mem_r        <= 1'b0;
+             ID_EXE_DatatoReg    <= 2'b00;
```

```
-            ID_EXE_read_reg2    <= read_reg2;
-        end
-    end
- endmodule
```

```
+            ID_EXE_RegWrite     <= 1'b0;
+            ID_EXE_zero_prediction <= 1'b0;
+            ID_EXE_PC_after_flush <= 1'b0;
+            ID_EXE_is_bne       <= 1'b0;
+
+            ID_EXE_written_reg  <= 5'b00000;
+            ID_EXE_read_reg1    <= 5'b00000;
+            ID_EXE_read_reg2    <= 5'b00000;
+        end
+        else if (CE) begin
+            ID_EXE_inst_in      <= inst_in;
+            ID_EXE_PC           <= PC;
+            ID_EXE_Imm_32       <= Imm_32;
+            ID_EXE_ALU_A        <= ALU_A;
+            ID_EXE_ALU_B        <= ALU_B;
+            ID_EXE_ALUSrc_A     <= ALUSrc_A;
+            ID_EXE_ALUSrc_B     <= ALUSrc_B;
+            ID_EXE_ALU_Control  <= ALU_Control;
+            ID_EXE_Data_out     <= Data_out;
+            ID_EXE_mem_w        <= mem_w;
+            ID_EXE_mem_r        <= mem_r;
+            ID_EXE_DatatoReg    <= DatatoReg;
+            ID_EXE_RegWrite     <= RegWrite;
+            ID_EXE_zero_prediction <= zero_prediction;
+            // ID_EXE_PC_after_flush <= PC + 4 if we predicted T (zero_prediction == 1)
+            // ID_EXE_PC_after_flush <= branching address if we predicted NT (zero_prediction == 0)
+            if (zero_prediction == 1) begin
+                ID_EXE_PC_after_flush <= PC[31:0] + 32'b0100;
+            end
+            else begin
+                ID_EXE_PC_after_flush <= add_branch_out;
+            end
+            ID_EXE_is_bne       <= is_bne;
+
+            ID_EXE_written_reg  <= written_reg;
+            ID_EXE_read_reg1    <= read_reg1;
+            ID_EXE_read_reg2    <= read_reg2;
+        end
+    end
+ endmodule
```

129 RV32I/RV32i.srcs/sources_1/new/REG_IF_ID.v

```
@@ -1,54 +1,75 @@
- `timescale 1ps / 1ps
- //////////////////////////////////////////////////////////////////////////////////
- // Company:
- // Engineer:
- //
- // Create Date: 2021/03/10 21:05:36
- // Design Name:
- // Module Name: REG_IF_ID
- // Project Name:
- // Target Devices:
- // Tool Versions:
- // Description:
- //
- // Dependencies:
- //
- // Revision:
- // Revision 0.01 - File Created
- // Additional Comments:
- //
- //////////////////////////////////////////////////////////////////////////////////
-
-
- module REG_IF_ID(
-         input clk,
-         input rst,
-         input CE,
-         input IF_ID_dstall,
-         input IF_ID_cstall,
-
-         input [31:0] inst_in,
-         input [31:0] PC,
-
-         output reg [31:0] IF_ID_inst_in,
-         output reg [31:0] IF_ID_PC = 0
-     );
-     always @ (posedge clk or posedge rst) begin
-         if (rst == 1) begin
-             IF_ID_inst_in <= 32'h00000013;
-             IF_ID_PC <= 32'h00000000;
-         end
-         // A bubble here is a nop, or rather, "addi x0, x0, 0"
-         if (IF_ID_dstall == 0) begin
-             if (rst == 1 || IF_ID_cstall == 1'b1) begin
-                 IF_ID_inst_in <= 32'h00000013;
-                 IF_ID_PC <= 32'h00000000;
-             end
-             else if (CE) begin
-                 IF_ID_inst_in <= inst_in;
-                 IF_ID_PC <= PC;
-             end
-         end
-         // else: if stall, then nothing changes here
-     end
- endmodule
```

```
+ `timescale 1ps / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 2021/03/10 21:05:36
+ // Design Name:
+ // Module Name: REG_IF_ID
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module REG_IF_ID(
+         input clk,
+         input rst,
+         input CE,
+         input IF_ID_dstall,
+         input IF_ID_cstall,
+
+         input [31:0] inst_in,
+         input [31:0] PC,
+         input [31:0] Imm_32,
+         input [31:0] add_branch_out,
+         input zero_prediction,
+         input is_bne,
+         input is_first_jalr,    // 1 if it is the first jalr out of the 2 in the bubble, 0 otherwise
+
+         output reg [31:0] IF_ID_inst_in,
+         output reg [31:0] IF_ID_PC = 0,
+         output reg [31:0] IF_ID_Imm_32,
+         output reg [31:0] IF_ID_add_branch_out,
+         output reg IF_ID_zero_prediction,
+         output reg IF_ID_is_bne,
+         output reg IF_ID_is_first_jalr
+
+     );
+     always @ (posedge clk or posedge rst) begin
+         if (rst == 1) begin
+             IF_ID_inst_in <= 32'h00000013;
+             IF_ID_PC <= 32'h00000000;
+         end
+         // A bubble here is a nop, or rather, "addi x0, x0, 0"
+         if (IF_ID_dstall == 0) begin
+             if (rst == 1 || IF_ID_cstall == 1'b1) begin
+                 IF_ID_inst_in <= 32'h00000013;
+                 IF_ID_PC <= 32'h00000000;
+                 IF_ID_Imm_32 <= 32'h00000000;
+                 IF_ID_add_branch_out <= 32'h00000000;
+                 IF_ID_zero_prediction <= 1'b0;
+                 IF_ID_is_bne <= 1'b0;
+                 IF_ID_is_first_jalr <= 1'b1;
+             end
+             else if (CE) begin
+                 IF_ID_inst_in <= inst_in;
+                 IF_ID_PC <= PC;
+                 IF_ID_Imm_32 <= Imm_32;
+                 IF_ID_add_branch_out <= add_branch_out;
+                 IF_ID_zero_prediction <= zero_prediction;
+                 IF_ID_is_bne <= is_bne;
+                 IF_ID_is_first_jalr <= is_first_jalr;
+             end
+         end
+         // else: if stall, then nothing changes here
+     end
+ endmodule
```

125 RV32i/RV32i.srcs/sources_1/new/REG_MEM_WB.v

```
@@ -1,60 +1,65 @@
- `timescale 1ps / 1ps
- //////////////////////////////////////////////////////////////////////////////////
- // Company:
- // Engineer:
- //
- // Create Date: 2021/03/12 08:12:16
- // Design Name:
- // Module Name: REG_MEM_WB
- // Project Name:
- // Target Devices:
- // Tool Versions:
- // Description:
- //
- // Dependencies:
- //
- // Revision:
- // Revision 0.01 - File Created
- // Additional Comments:
- //
- //////////////////////////////////////////////////////////////////////////////////
-
-
- module REG_MEM_WB(
-         input clk,
-         input rst,
-         input CE,
-         // Input
-         input [31:0] inst_in,
-         input [31:0] PC,
-         input [31:0] ALU_out,
-         input [1:0] DatatoReg,
-         input RegWrite,
-         input [31:0] Data_in,
-         // Output
-         output reg [31:0] MEM_WB_inst_in,
-         output reg [31:0] MEM_WB_PC = 0,
-         output reg [31:0] MEM_WB_ALU_out,
-         output reg [1:0] MEM_WB_DatatoReg,
-         output reg MEM_WB_RegWrite,
-         output reg [31:0] MEM_WB_Data_in
-     );
-     always @ (posedge clk or posedge rst) begin
-         if (rst == 1) begin
-             MEM_WB_inst_in      <= 32'h00000013;
-             MEM_WB_PC           <= 32'h00000000;
-             MEM_WB_ALU_out      <= 32'h00000000;
-             MEM_WB_DatatoReg    <= 2'b00;
-             MEM_WB_RegWrite     <= 1'b0;
-             MEM_WB_Data_in      <= 32'h00000000;
-         end
-         else if (CE) begin
-             MEM_WB_inst_in      <= inst_in;
-             MEM_WB_PC           <= PC;
-             MEM_WB_ALU_out      <= ALU_out;
-             MEM_WB_DatatoReg    <= DatatoReg;
-             MEM_WB_RegWrite     <= RegWrite;
-             MEM_WB_Data_in      <= Data_in;
-         end
-     end
- endmodule
+ `timescale 1ps / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 2021/03/12 08:12:16
+ // Design Name:
+ // Module Name: REG_MEM_WB
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module REG_MEM_WB(
+         input clk,
+         input rst,
+         input CE,
+         // Input
+         input [31:0] inst_in,
+         input [31:0] PC,
+         input [31:0] ALU_out,   // Data from ALU
+         input [1:0] DatatoReg,
+         input RegWrite,
+         input [31:0] Data_in,   // Data from memory (load)
+         input [4:0] written_reg,
+
+         // Output
+         output reg [31:0] MEM_WB_inst_in,
+         output reg [31:0] MEM_WB_PC = 0,
+         output reg [31:0] MEM_WB_ALU_out,
+         output reg [1:0] MEM_WB_DatatoReg,
+         output reg MEM_WB_RegWrite,
+         output reg [31:0] MEM_WB_Data_in,
+         output reg [4:0] MEM_WB_written_reg
+     );
+     always @ (posedge clk or posedge rst) begin
+         if (rst == 1) begin
+             MEM_WB_inst_in      <= 32'h00000013;
+             MEM_WB_PC           <= 32'h00000000;
+             MEM_WB_ALU_out      <= 32'h00000000;
+             MEM_WB_DatatoReg    <= 2'b00;
+             MEM_WB_RegWrite     <= 1'b0;
+             MEM_WB_Data_in      <= 32'h00000000;
+             MEM_WB_written_reg  <= 5'b00000;
+         end
+         else if (CE) begin
+             MEM_WB_inst_in      <= inst_in;
+             MEM_WB_PC           <= PC;
+             MEM_WB_ALU_out      <= ALU_out;
+             MEM_WB_DatatoReg    <= DatatoReg;
+             MEM_WB_RegWrite     <= RegWrite;
+             MEM_WB_Data_in      <= Data_in;
+             MEM_WB_written_reg  <= written_reg;
+         end
+     end
+ endmodule
```

1,061 RV32i/RV32i.srcs/sources_1/new/RV32iPCPU.v

```
@@ -1,450 +1,611 @@
- `timescale 1ps / 1ps
- //////////////////////////////////////////////////////////////////////////////////
- // Company:
- // Engineer:
- //
- // Create Date: 2021/03/10 19:39:55
- // Design Name:
- // Module Name: RV32iPCPU
- // Project Name:
- // Target Devices:
- // Tool Versions:
- // Description:
- //
- // Dependencies:
- //
- // Revision:
- // Revision 0.01 - File Created
- // Additional Comments:
- //
- //////////////////////////////////////////////////////////////////////////////////
-
-
- module RV32iPCPU(
-     input clk,
-     input rst,
-     input [31:0] data_in,   // MEM
-     input [31:0] inst_in,   // IF, from PC_out
-
-     output [31:0] ALU_out,  // From MEM, address out, for fetching data_in
-     output [31:0] data_out, // From MEM, to be written into data memory
-     output mem_w,           // From MEM, write valid, for store instructions
-     output [31:0] PC_out    // From IF
-     );
-     wire V5;
-     wire N0;
-     wire [31:0] Imm_32;
-     wire [31:0] add_branch_out;
-     wire [31:0] add_jal_out;
-     wire [31:0] add_jalr_out;
-
-     wire [4:0] Wt_addr;
+ `timescale 1ps / 1ps
+ //////////////////////////////////////////////////////////////////////////////////
+ // Company:
+ // Engineer:
+ //
+ // Create Date: 2021/03/10 19:39:55
+ // Design Name:
+ // Module Name: RV32iPCPU
+ // Project Name:
+ // Target Devices:
+ // Tool Versions:
+ // Description:
+ //
+ // Dependencies:
+ //
+ // Revision:
+ // Revision 0.01 - File Created
+ // Additional Comments:
+ //
+ //////////////////////////////////////////////////////////////////////////////////
+
+
+ module RV32iPCPU(
+     input clk,
+     input rst,
+     input [31:0] data_in,   // MEM
+     input [31:0] inst_in,   // IF, from PC_out
+
+     output [31:0] ALU_out,  // From MEM, address out, for fetching data_in
+     output [31:0] data_out, // From MEM, to be written into data memory
+     output mem_w,           // From MEM, write valid, for store instructions
+     output [31:0] PC_out    // From IF
+     );
+     wire V5;
+     wire N0;
+     wire [31:0] Imm_32;
+     wire [31:0] add_branch_out;
+     wire [31:0] add_jal_out;
+     wire [31:0] add_jalr_out;
+     wire [31:0] add_jalr_out_prov;
+
```

```
-    wire [31:0] Wt_data;
-    wire [31:0] rdata_A;
-    wire [31:0] rdata_B;
-    wire [31:0] PC_wb;
-    wire [31:0] ALU_A;
-    wire [31:0] ALU_B;
-    assign V5 = 1'b1;
-    assign N0 = 1'b0;
-
-    wire zero;              // ID
-    wire [1:0] Branch;      // ID
-    wire ALUSrc_A;          // EXE
-    wire [1:0] ALUSrc_B;    // EXE
-    wire [4:0] ALU_Control; // EXE
-    wire RegWrite;          // WB
-    wire [1:0] DatatoReg;   // WB
-
-    wire [1:0] B_H_W;       // WB // not used yet
-    wire sign;              // WB // not used yet
- //    wire RegDst; // WB
- //    wire Jal; // WB
-
-    // IF_ID
-    wire [31:0] IF_ID_inst_in;
-    wire [31:0] IF_ID_PC;
-    wire [31:0] IF_ID_Data_out;
-    wire IF_ID_mem_w;
-    wire [4:0] IF_ID_written_reg;
-    wire [4:0] IF_ID_read_reg1;
-    wire [4:0] IF_ID_read_reg2;
-
-    // ID_EXE
-    wire [31:0] ID_EXE_inst_in;
-    wire [31:0] ID_EXE_PC;
-    wire [31:0] ID_EXE_ALU_A;
-    wire [31:0] ID_EXE_ALU_B;
-    wire [4:0] ID_EXE_ALU_Control;
-    wire [31:0] ID_EXE_Data_out;
-    wire ID_EXE_mem_w;
-    wire [1:0] ID_EXE_DatatoReg;
-    wire ID_EXE_RegWrite;
-    wire [4:0] ID_EXE_written_reg;
-    wire [4:0] ID_EXE_read_reg1;
-    wire [4:0] ID_EXE_read_reg2;
-
-    wire [31:0] ID_EXE_ALU_out;
-
-    // EXE_MEM
-    wire [31:0] EXE_MEM_inst_in;
-    wire [31:0] EXE_MEM_PC;
-    wire [31:0] EXE_MEM_ALU_out;
-    wire [31:0] EXE_MEM_Data_out;
-    wire EXE_MEM_mem_w;
-    wire [1:0] EXE_MEM_DatatoReg;
-    wire EXE_MEM_RegWrite;
-    wire [4:0] EXE_MEM_written_reg;
-    wire [4:0] EXE_MEM_read_reg1;
-    wire [4:0] EXE_MEM_read_reg2;

-    // MEM_WB
-    wire [31:0] MEM_WB_inst_in;
-    wire [31:0] MEM_WB_PC;
-    wire [31:0] MEM_WB_ALU_out;
-    wire [31:0] MEM_WB_Data_in;
-    wire [1:0] MEM_WB_DatatoReg;
-    wire MEM_WB_RegWrite;

-   // Stall
-   wire PC_dstall;
-   wire IF_ID_cstall;
-   wire IF_ID_dstall;
-   wire ID_EXE_dstall;


-    Data_Stall _dstall_ (
-        .IF_ID_written_reg(IF_ID_written_reg),
-        .IF_ID_read_reg1(IF_ID_read_reg1),
-        .IF_ID_read_reg2(IF_ID_read_reg2),

-        .ID_EXE_written_reg(ID_EXE_written_reg),
-        .ID_EXE_read_reg1(ID_EXE_read_reg1),
-        .ID_EXE_read_reg2(ID_EXE_read_reg2),

-        .EXE_MEM_written_reg(EXE_MEM_written_reg),
-        .EXE_MEM_read_reg1(EXE_MEM_read_reg1),
-        .EXE_MEM_read_reg2(EXE_MEM_read_reg2),

-        .PC_dstall(PC_dstall),
-        .IF_ID_dstall(IF_ID_dstall),
-        .ID_EXE_dstall(ID_EXE_dstall)
-        );

-    Control_Stall _cstall_ (
-        .Branch(Branch[1:0]),
-        .IF_ID_cstall(IF_ID_cstall)
-        );

-    assign ALU_out = EXE_MEM_ALU_out;
-    assign data_out = EXE_MEM_Data_out;
-    assign mem_w = EXE_MEM_mem_w;

-    // IF:-------------------------------------------------------------------------------------
-    // Control Signals:
-    //   1. Branch - MUX5 : ID
-    // References:
-    //   1. inst_in - MUX5 : ID
-    //   2. rdata_A - MUX5 : ID
-    //   3. Imm_32 - ADD_Branch : ID
-    // Pass-on:
-    //   1. inst_in (combinatorial)
-    //   2. PC
-    // Out:
-    //   1. PC_out: for fetching inst_in
```

```
+    wire [4:0] Wt_addr;
+    wire [31:0] Wt_data;
+    wire [31:0] rdata_A;
+    wire [31:0] rdata_B;
+    wire [31:0] PC_wb;
+    wire [31:0] ALU_A;
+    wire [31:0] ALU_B;
+    assign V5 = 1'b1;
+    assign N0 = 1'b0;
+
+
+    wire zero_prediction;   // ID
+    wire zero_correct;      // EXE
+    wire correct_prediction;
+    wire [31:0] unrecovered_PC;
+    wire is_branch;
+    wire is_bne;
+    wire is_first_jalr;
+    wire [1:0] Branch;      // IF
+    wire [4:0] ALU_Control; // EXE
+    wire ALUSrc_A;          // ID
+    wire [1:0] ALUSrc_B;    // ID
+    wire RegWrite;          // WB
+    wire [1:0] DatatoReg;   // WB
+
+    wire [1:0] B_H_W;       // WB // not used yet
+    wire sign;              // WB // not used yet
+ //    wire RegDst; // WB
+ //    wire Jal; // WB
+
+    // IF_ID
+    wire [31:0] IF_ID_inst_in;
+    wire [31:0] IF_ID_PC;
+    wire [31:0] IF_ID_Imm_32;
+    wire [31:0] IF_ID_Data_out;
+    wire [31:0] IF_ID_add_branch_out;
+    wire IF_ID_zero_prediction;
+    wire IF_ID_is_bne;
+    wire IF_ID_is_first_jalr;
+    wire IF_ID_mem_w;
+    wire IF_ID_mem_r;
+    wire [4:0] IF_ID_written_reg;
+    wire [4:0] IF_ID_read_reg1;
+    wire [4:0] IF_ID_read_reg2;
+
+    // ID_EXE
+    wire [31:0] ID_EXE_inst_in;
+    wire [31:0] ID_EXE_PC;
+    wire [31:0] ID_EXE_Imm_32;
+    wire [31:0] ID_EXE_ALU_A;
+    wire [31:0] ID_EXE_ALU_B;
+    wire ID_EXE_ALUSrc_A;
+    wire [1:0] ID_EXE_ALUSrc_B;
+    wire [31:0] ID_EXE_ALU_A_forward;
+    wire [31:0] ID_EXE_ALU_B_forward;
+    wire [31:0] ID_EXE_Data_out_forward;
+    wire [4:0] ID_EXE_ALU_Control;
+    wire [31:0] ID_EXE_Data_out;
+    wire ID_EXE_mem_w;
+    wire [1:0] ID_EXE_DatatoReg;
+    wire ID_EXE_RegWrite;
+    wire ID_EXE_zero_prediction;
+    wire ID_EXE_is_bne;
+    wire [31:0] ID_EXE_PC_after_flush;
+    wire [4:0] ID_EXE_written_reg;
+    wire [4:0] ID_EXE_read_reg1;
+    wire [4:0] ID_EXE_read_reg2;
+
+    wire [31:0] ID_EXE_ALU_out;
+
+    // EXE_MEM
+    wire [31:0] EXE_MEM_inst_in;
+    wire [31:0] EXE_MEM_PC;
+    wire [31:0] EXE_MEM_ALU_out;
+    wire [31:0] EXE_MEM_Data_out;
+    wire EXE_MEM_mem_w;
+    wire [1:0] EXE_MEM_DatatoReg;
+    wire EXE_MEM_RegWrite;
+    wire [4:0] EXE_MEM_written_reg;
+    wire [4:0] EXE_MEM_read_reg1;
+    wire [4:0] EXE_MEM_read_reg2;
+
+    // MEM_WB
+    wire [31:0] MEM_WB_inst_in;
+    wire [31:0] MEM_WB_PC;
+    wire [31:0] MEM_WB_ALU_out;
+    wire [31:0] MEM_WB_Data_in;
+    wire [1:0] MEM_WB_DatatoReg;
+    wire MEM_WB_RegWrite;
+    wire [4:0] MEM_WB_written_reg;
+
+    // Stall
+    wire PC_dstall;
+    wire PC_cstall;
+    wire IF_cstall;
+    wire IF_ID_cstall;
+    wire IF_ID_dstall;
+    wire ID_EXE_dstall;
+    wire ID_EXE_cstall;
+
+    // Forwarding
+    wire [1:0] forwarding_A_sig;
+    wire [1:0] forwarding_B_sig;
+
+
+    Data_Stall _dstall_ (
+        // Input:
+        .ID_EXE_mem_r(ID_EXE_mem_r),

+        .IF_ID_read_reg1(IF_ID_read_reg1),
+        .IF_ID_read_reg2(IF_ID_read_reg2),

+        .ID_EXE_written_reg(ID_EXE_written_reg),

+        // Output:
```

```
-    REG32 _pc_ (
-        .CE(V5),
-        .clk(clk),
-        .D(PC_wb[31:0]),
-        .rst(rst),
-        .Q(PC_out[31:0]),
-        .PC_dstall(PC_dstall)
-        );
-    add_32  ADD_Branch (
-        .a(IF_ID_PC[31:0]),         // use the "PC" from ID stage
-        .b(Imm_32[31:0]),           // From ID stage
-        .c(add_branch_out[31:0])    // actually this part belongs to IF_ID
-        );
-    add_32 ADD_JAL (
-        .a(IF_ID_PC),               // MIPS: PC+4, RISC-V: PC!!!
-        .b({{11{IF_ID_inst_in[31]}}, IF_ID_inst_in[31], IF_ID_inst_in[19:12], IF_ID_inst_in[20], IF_ID_inst_in[30:21], 1'b0}),
-        .c(add_jal_out[31:0])
-        );
-    add_32 ADD_JALR (
-        .a(rdata_A[31:0]),
-        .b({{20{IF_ID_inst_in[31]}}, IF_ID_inst_in[31:20]}),
-        .c(add_jalr_out[31:0])
-        );
-    Mux4to1b32  MUX5 (
-        .I0(PC_out[31:0] + 32'b0100),   // From IF stage
-        .I1(add_branch_out[31:0]),      // Containing "PC" from ID stage
-        .I2(add_jal_out[31:0]),         // From ID stage
-        .I3(add_jalr_out[31:0]),        // From ID stage
-        .s(Branch[1:0]),                // From ID
-        .o(PC_wb[31:0])
-        );
-
-    REG_IF_ID _if_id_ (
-        .clk(clk), .rst(rst), .CE(V5),
-        .IF_ID_dstall(IF_ID_dstall), .IF_ID_cstall(IF_ID_cstall),
-        // Input
-        .inst_in(inst_in),
-        .PC(PC_out),
-        // Output
-        .IF_ID_inst_in(IF_ID_inst_in),
-        .IF_ID_PC(IF_ID_PC)
-        );
-
-    // ID:-------------------------------------------------------------------------------
-    // From IF:
-    //   1. inst_in
-    //   2. PC
-    // Control Signals:
-    //   1. RegWrite - Regs : WB
-    //   2. ALUSrc_A / ALUSrc_B (stops here)
-    // References:
-    //   None
-    // Pass-on:
-    //   1. inst_in
-       //   Control_signals {
-       //   2. ALU_Control
-       //   3. DatatoReg
-       //   4. mem_w
-       //   5. RegWrite
-       //   }
-    //   6. ALU_A
-    //   7. ALU_B
-    //   8. Data_out
-    //   9. PC
-    // Out:
-    //   None
-
-    Get_rw_regs _rw_regs_ (
-        .inst_in(IF_ID_inst_in[31:0]),
-        .written_reg(IF_ID_written_reg),
-        .read_reg1(IF_ID_read_reg1),
-        .read_reg2(IF_ID_read_reg2)
-        );
-    Controler  Ctrl_Unit (
-        // Input:

-        .OPcode(IF_ID_inst_in[6:0]),
-        .Fun1(IF_ID_inst_in[14:12]),
-        .Fun2(IF_ID_inst_in[31:25]),
-        .zero(zero),
-        // Output:
-        .ALUSrc_A(ALUSrc_A),
-        .ALUSrc_B(ALUSrc_B[1:0]),
-        .ALU_Control(ALU_Control[4:0]),
-        .Branch(Branch[1:0]),
-        .DatatoReg(DatatoReg[1:0]),
-        .mem_w(IF_ID_mem_w),
-        .RegWrite(RegWrite),
-        .B_H_W(B_H_W),                  // not used yet
-        .sign(sign)                     // not used yet
-        );

-    Regs U2 (.clk(clk),
-             .rst(rst),
-             .L_S(MEM_WB_RegWrite),             // From Write-Back stage
-             .R_addr_A(IF_ID_inst_in[19:15]),   // ID
-             .R_addr_B(IF_ID_inst_in[24:20]),   // ID
-             .Wt_addr(Wt_addr[4:0]),            // From Write-Back stage
-             .Wt_data(Wt_data[31:0]),           // From Write-Back stage
-             .rdata_A(rdata_A[31:0]),
-             .rdata_B(rdata_B[31:0])
-             );
-    SignExt _signed_ext_ (.inst_in(IF_ID_inst_in), .imm_32(Imm_32));

-    Mux2to1b32  _alu_source_A_ (
-        .I0(rdata_A[31:0]),
-        .I1(Imm_32[31:0]),   // not used
-        .s(ALUSrc_A),
-        .o(ALU_A[31:0])
-        );

-    Mux4to1b32  _alu_source_B_ (
-        .I0(rdata_B[31:0]),
-        .I1(Imm_32[31:0]),
-        .I2(),
-        .I3(),
-        .s(ALUSrc_B[1:0]),
-        .o(ALU_B[31:0]
-        ));
-    assign IF_ID_Data_out = rdata_B;
-    ID_Zero_Generator _id_zero_ (.A(ALU_A), .B(ALU_B), .ALU_operation(ALU_Control), .zero(zero));
-
```

```
+        .PC_dstall(PC_dstall),
+        .IF_ID_dstall(IF_ID_dstall),
+        .ID_EXE_dstall(ID_EXE_dstall)
+        );
+
+    Control_Stall _cstall_ (
+        // Input:
+        .Branch(Branch[1:0]),
+        .IF_ID_is_first_jalr(IF_ID_is_first_jalr),
+        .correct_prediction(correct_prediction),
+        // Output:
+        .IF_cstall(IF_cstall),
+        .IF_ID_cstall(IF_ID_cstall),
+        .ID_EXE_cstall(ID_EXE_cstall),
+        .PC_cstall(PC_cstall),
+        .is_first_jalr(is_first_jalr)

+        );
+
+    assign ALU_out = EXE_MEM_ALU_out;
+    assign data_out = EXE_MEM_Data_out;
+    assign mem_w = EXE_MEM_mem_w;
+
+    // IF:-------------------------------------------------------------------------------------------
+    // Control Signals:
+    //   1. Branch - MUX5 : ID
+    // References:
+    //   1. inst_in - MUX5 : ID
+    //   2. rdata_A - MUX5 : ID
+    //   3. Imm_32 - ADD_Branch : ID
+    // Pass-on:
+    //   1. inst_in (combinatorial)
+    //   2. PC
+    // Out:
+    //   1. PC_out: for fetching inst_in
+
+    SignExt _signed_ext_ (.inst_in(inst_in), .imm_32(Imm_32));   // Moved from ID stage to IF stage
+
+    REG32 _pc_ (
+        .CE(V5),
+        .clk(clk),
+        .D(PC_wb[31:0]),
+        .rst(rst),
+        .Q(PC_out[31:0]),
+        .PC_dstall(PC_dstall),
+        .PC_cstall(PC_cstall)
+        );
+
+    // Branch Predictor
-    Gshare_Predictor _branch_predictor_ (
+        // Input:
+        .clk(clk),
+        .rst(rst),
+        .PC(PC_out),
+        .OPcode(inst_in[6:0]),
+        .Fun1(inst_in[14:12]),
+        .correct_prediction(correct_prediction),
+        .is_branch(is_branch),
+
+        // Output:
+        .Branch(Branch[1:0]),
+        .zero_prediction(zero_prediction),
+        .is_bne(is_bne)
+        );
+
+    add_32  ADD_Branch (
+        .a(PC_out[31:0]),         // use the "PC" from IF stage
+        .b(Imm_32[31:0]),           // From IF stage
+        .c(add_branch_out[31:0])
+        );
+    add_32 ADD_JAL (
+        .a(PC_out[31:0]),               // MIPS: PC+4, RISC-V: PC!!!
+        .b({{11{inst_in[31]}}, inst_in[31], inst_in[19:12], inst_in[20], inst_in[30:21], 1'b0}),
+        .c(add_jal_out[31:0])
+        );
+    add_32 ADD_JALR (
+        .a(rdata_A[31:0]),  // We cannot get the most updated value of rdata_A from IF stage, so we need to stall for 1 cycle to get it at ID stage
+        .b({{20{IF_ID_inst_in[31]}}, IF_ID_inst_in[31:20]}),    // From ID stage
+        .c(add_jalr_out_prov[31:0])     // Provisional value
+        );
+
+    // If IF_cstall == 1, we fetch the same instruction again
+    Mux2to1b32 MUX_JALR_SELECT (
+        .I0(add_jalr_out_prov[31:0]),
+        .I1(PC_out[31:0]),
+        .s(IF_cstall),
+        .o(add_jalr_out[31:0])
+        );
+
+    Mux4to1b32 MUX5 (
+        .I0(PC_out[31:0] + 32'b0100),   // From IF stage (PC+4)             00
+        .I1(add_branch_out[31:0]),      // From IF stage                    01
+        .I2(add_jal_out[31:0]),         // From IF stage                    10
+        .I3(add_jalr_out[31:0]),        // From ID stage                    11
+        .s(Branch[1:0]),                // From IF stage
+        .o(unrecovered_PC[31:0])
+        );
+
+    Mux2to1b32 Recover_PC (
+        .I0(ID_EXE_PC_after_flush[31:0]),
+        .I1(unrecovered_PC[31:0]),
+        .s(correct_prediction),
+        .o(PC_wb[31:0])
+        );

+    REG_IF_ID _if_id_ (
+        .clk(clk), .rst(rst), .CE(V5),
+        .IF_ID_dstall(IF_ID_dstall), .IF_ID_cstall(IF_ID_cstall),
+        // Input
+        .inst_in(inst_in),
+        .PC(PC_out),
+        .Imm_32(Imm_32),
+        .add_branch_out(add_branch_out),
+        .zero_prediction(zero_prediction),
+        .is_bne(is_bne),
+        .is_first_jalr(is_first_jalr),
+        // Output
+        .IF_ID_inst_in(IF_ID_inst_in),
+        .IF_ID_PC(IF_ID_PC),
+        .IF_ID_Imm_32(IF_ID_Imm_32),
+        .IF_ID_add_branch_out(IF_ID_add_branch_out),
+        .IF_ID_zero_prediction(IF_ID_zero_prediction),
+        .IF_ID_is_bne(IF_ID_is_bne),
```

```
-    REG_ID_EXE _id_exe_ (
-        .clk(clk), .rst(rst), .CE(V5), .ID_EXE_dstall(ID_EXE_dstall),
-        // Input
-        .inst_in(IF_ID_inst_in),
-        .PC(IF_ID_PC),
-        //// To EXE stage, ALU Operands A & B
-        .ALU_A(ALU_A),
-        .ALU_B(ALU_B),
-        //// To EXE stage, ALU operation control signal
-        .ALU_Control(ALU_Control),
-        //// To MEM stage, for sw instruction, data from rs2 register written into memory
-        .Data_out(IF_ID_Data_out),
-        //// To MEM stage, for sw instruction, memor write enable signal
-        .mem_w(IF_ID_mem_w),
-        //// To WB stage, for choosing different data written back to register file
-        .DatatoReg(DatatoReg),
-        //// To WB stage, register file write valid
-        .RegWrite(RegWrite),
-        //// For Data Hazard
-        .written_reg(IF_ID_written_reg), .read_reg1(IF_ID_read_reg1), .read_reg2(IF_ID_read_reg2),
-
-        // Output
-        .ID_EXE_inst_in(ID_EXE_inst_in),
-        .ID_EXE_PC(ID_EXE_PC),
-        .ID_EXE_ALU_A(ID_EXE_ALU_A),
-        .ID_EXE_ALU_B(ID_EXE_ALU_B),
-        .ID_EXE_ALU_Control(ID_EXE_ALU_Control),
-        .ID_EXE_Data_out(ID_EXE_Data_out),
-        .ID_EXE_mem_w(ID_EXE_mem_w),
-        .ID_EXE_DatatoReg(ID_EXE_DatatoReg),
-        .ID_EXE_RegWrite(ID_EXE_RegWrite),
-        //// For Data Hazard
-        .ID_EXE_written_reg(ID_EXE_written_reg), .ID_EXE_read_reg1(ID_EXE_read_reg1),
.ID_EXE_read_reg2(ID_EXE_read_reg2)
-        );
-
-    // EXE:----------------------------------------------------------------------------------------
-    // From ID:
-    //   1. inst_in
-        //   Control_signals {
-        //   2. ALU_Control (stops here)
-        //   3. mem_w
-        //   4. DatatoReg
-        //   5. RegWrite
-        //   }
-    //   6. ALU_A (stops here)
-    //   7. ALU_B (stops here)
-    //   8. Data_out
-    //   9. PC
-    // Control Signals:
-    //   1. ALU_Control
-    // References:
-    //   None
-    // Pass-on:
-    //   1. inst_in
-        //   Control_signals {
-        //   2. DatatoReg (WB)
-        //   3. mem_w (MEM)
-        //   4. RegWrite (WB)
-        //   }
-    //   5. Data_out (used at MEM together with mem_w)
-    //   6. ALU_out (Addr_out outside) (used at both MEM and WB)
-    //   7. PC
-    // Out:
-    //   None
-
-    ALU _alualu_ (
-        .A(ID_EXE_ALU_A[31:0]),

-        .B(ID_EXE_ALU_B[31:0]),
-        .ALU_operation(ID_EXE_ALU_Control[4:0]),
-        .res(ID_EXE_ALU_out[31:0]),
-        .overflow(),
-        .zero()
-        );
-
-    REG_EXE_MEM _exe_mem_ (
-        .clk(clk), .rst(rst), .CE(V5),
-        // Input
-        .inst_in(ID_EXE_inst_in),
-        .PC(ID_EXE_PC),
-        //// To MEM stage
-        .ALU_out(ID_EXE_ALU_out),
-        .Data_out(ID_EXE_Data_out),
-        .mem_w(ID_EXE_mem_w),
-        //// To WB stage
-        .DatatoReg(ID_EXE_DatatoReg),
-        .RegWrite(ID_EXE_RegWrite),

-        .written_reg(ID_EXE_written_reg), .read_reg1(ID_EXE_read_reg1), .read_reg2(ID_EXE_read_reg2),

-        // Output
-        .EXE_MEM_inst_in(EXE_MEM_inst_in),
-        .EXE_MEM_PC(EXE_MEM_PC),
-        .EXE_MEM_ALU_out(EXE_MEM_ALU_out),
-        .EXE_MEM_Data_out(EXE_MEM_Data_out),
-        .EXE_MEM_mem_w(EXE_MEM_mem_w),
-        .EXE_MEM_DatatoReg(EXE_MEM_DatatoReg),
-        .EXE_MEM_RegWrite(EXE_MEM_RegWrite),

-        .EXE_MEM_written_reg(EXE_MEM_written_reg), .EXE_MEM_read_reg1(EXE_MEM_read_reg1),
.EXE_MEM_read_reg2(EXE_MEM_read_reg2)
-        );

-    // MEM:----------------------------------------------------------------------------------------
-    // From EXE:
-    //   1. inst_in
-        //   Control_signals {
-        //   2. DatatoReg (WB)
-        //   3. mem_w (stops here)
-        //   4. RegWrite (WB)
-        //   }
-    //   5. Data_out (stops here)
-    //   6. ALU_out (Addr_out outside) (used at both MEM and WB)
-    //   7. PC
-    // Control Signals:
-    //   1. mem_w

-    // Pass-on:
-    //   1. inst_in
-        //   Control_signals {
-        //   2. DatatoReg (WB)
-        //   3. RegWrite (WB)

+        .IF_ID_is_first_jalr(IF_ID_is_first_jalr)
+        );
+
+   // ID:-----------------------------------------------------------------------------------------
+   // From IF:
+   //   1. inst_in
+   //   2. PC
+   // Control Signals:
+   //   1. RegWrite - Regs : WB
+   //   2. ALUSrc_A / ALUSrc_B (stops here)
+   // References:
+   //   None
+   // Pass-on:
+   //   1. inst_in
+       //   Control_signals {
+       //   2. ALU_Control
+       //   3. DatatoReg
+       //   4. mem_w
+       //   5. RegWrite
+       //   }
+   //   6. ALU_A
+   //   7. ALU_B
+   //   8. Data_out
+   //   9. PC
+   // Out:
+   //   None
+
+    Get_rw_regs _rw_regs_ (
+        // Input:
+        .inst_in(IF_ID_inst_in[31:0]),
+        // Output:
+        .written_reg(IF_ID_written_reg),    // Note that IF_ID_written_reg is just the destination
register number obtained from decoding the instruction and is not in REG_IF_ID
+        .read_reg1(IF_ID_read_reg1),        // Note that IF_ID_read_reg1 is just the operand1 register
number obtained from decoding the instruction and is not in REG_IF_ID
+        .read_reg2(IF_ID_read_reg2)         // Note that IF_ID_read_reg2 is just the operand2 register
number obtained from decoding the instruction and is not in REG_IF_ID
+        );
+
+    Controler Ctrl_Unit (
+        // Input:
+        .OPcode(IF_ID_inst_in[6:0]),
+        .Fun1(IF_ID_inst_in[14:12]),
+        .Fun2(IF_ID_inst_in[31:25]),
+
+        // Output:
+        .ALUSrc_A(ALUSrc_A),
+        .ALUSrc_B(ALUSrc_B[1:0]),
+        .ALU_Control(ALU_Control[4:0]),
+        .DatatoReg(DatatoReg[1:0]),
+        .mem_w(IF_ID_mem_w),
+        .mem_r(IF_ID_mem_r),            // 1 if the instruction reads from memory
+        .RegWrite(RegWrite),
+        .B_H_W(B_H_W),                  // not used yet
+        .sign(sign)                     // not used yet
+        );
+
+    // Get the data from the register file
+    Regs U2 (.clk(clk),
+             .rst(rst),
+             .L_S(MEM_WB_RegWrite),             // From Write-Back stage
+             .R_addr_A(IF_ID_inst_in[19:15]),   // ID
+             .R_addr_B(IF_ID_inst_in[24:20]),   // ID
+             .Wt_addr(Wt_addr[4:0]),            // From Write-Back stage
+             .Wt_data(Wt_data[31:0]),           // From Write-Back stage
+             .rdata_A(rdata_A[31:0]),
+             .rdata_B(rdata_B[31:0])
+             );
+
+    assign IF_ID_Data_out = rdata_B;    // for sw instruction, data from rs2 register written into
memory, but we need to take into account data forwarding (_mux_forward_data_out_)

+    REG_ID_EXE _id_exe_ (
+        .clk(clk), .rst(rst), .CE(V5), .ID_EXE_dstall(ID_EXE_dstall), .ID_EXE_cstall(ID_EXE_cstall),
+        // Input
+        .inst_in(IF_ID_inst_in),
+        .PC(IF_ID_PC),
+        .Imm_32(IF_ID_Imm_32),
+        //// To EXE stage, ALU Operands A & B
+        .ALU_A(rdata_A[31:0]),
+        .ALU_B(rdata_B[31:0]),
+        //// To EXE stage, ALU operands source selection
+        .ALUSrc_A(ALUSrc_A),
+        .ALUSrc_B(ALUSrc_B),
+        //// To EXE stage, ALU operation control signal
+        .ALU_Control(ALU_Control),
+        //// To MEM stage, for sw instruction, data from rs2 register written into memory
+        .Data_out(IF_ID_Data_out),
+        //// To MEM stage, for sw instruction, memor write enable signal
+        .mem_w(IF_ID_mem_w),
+        //// To ID stage, to detect load-use data hazard
+        .mem_r(IF_ID_mem_r),
+        //// To WB stage, for choosing different data written back to register file
+        .DatatoReg(DatatoReg),
+        //// To WB stage, register file write valid
+        .RegWrite(RegWrite),
+        //// For Data Hazard
+        .written_reg(IF_ID_written_reg), .read_reg1(IF_ID_read_reg1), .read_reg2(IF_ID_read_reg2),
+        .add_branch_out(IF_ID_add_branch_out),
+        .zero_prediction(IF_ID_zero_prediction),
+        .is_bne(IF_ID_is_bne),
+
+        // Output

+        .ID_EXE_inst_in(ID_EXE_inst_in),
+        .ID_EXE_PC(ID_EXE_PC),
+        .ID_EXE_Imm_32(ID_EXE_Imm_32),
+        .ID_EXE_ALU_A(ID_EXE_ALU_A),    // Wire from REG_ID_EXE to the ALU Forwarding Multiplexer (A)
+        .ID_EXE_ALU_B(ID_EXE_ALU_B),    // Wire from REG_ID_EXE to the ALU Forwarding Multiplexer (B)
+        .ID_EXE_ALUSrc_A(ID_EXE_ALUSrc_A),
+        .ID_EXE_ALUSrc_B(ID_EXE_ALUSrc_B),
+        .ID_EXE_ALU_Control(ID_EXE_ALU_Control),
+        .ID_EXE_Data_out(ID_EXE_Data_out),
+        .ID_EXE_mem_w(ID_EXE_mem_w),
+        .ID_EXE_mem_r(ID_EXE_mem_r),
+        .ID_EXE_DatatoReg(ID_EXE_DatatoReg),
+        .ID_EXE_RegWrite(ID_EXE_RegWrite),
+        //// For Data Hazard
+        .ID_EXE_written_reg(ID_EXE_written_reg), .ID_EXE_read_reg1(ID_EXE_read_reg1),
.ID_EXE_read_reg2(ID_EXE_read_reg2),
+        .ID_EXE_zero_prediction(ID_EXE_zero_prediction),
+        .ID_EXE_PC_after_flush(ID_EXE_PC_after_flush),
+        .ID_EXE_is_bne(ID_EXE_is_bne)
+        );
+
```

```
-         //   }
-     //   4. ALU_out (Addr_out outside) (used at both MEM and WB)
-     //   5. PC
-     //   6. Data_in
-     // Out:
-     //   Data_out & mem_w, ALU_out(as Addr_out)
-
-     REG_MEM_WB _mem_wb_ (
-         .clk(clk), .rst(rst), .CE(V5),
-         // Input
-         .inst_in(EXE_MEM_inst_in),
-         .PC(EXE_MEM_PC),
-         .ALU_out(EXE_MEM_ALU_out),
-         .DatatoReg(EXE_MEM_DatatoReg),
-         .RegWrite(EXE_MEM_RegWrite),
-         //// Comes from data memory
-         .Data_in(data_in),
-
-         // Output
-         .MEM_WB_inst_in(MEM_WB_inst_in),
-         .MEM_WB_PC(MEM_WB_PC),
-         .MEM_WB_ALU_out(MEM_WB_ALU_out),
-         .MEM_WB_DatatoReg(MEM_WB_DatatoReg),
-         .MEM_WB_RegWrite(MEM_WB_RegWrite),
-         .MEM_WB_Data_in(MEM_WB_Data_in)
-         );
-
-     // WB:----------------------------------------------------------------------------------------
-     // From EXE:
-     //   1. inst_in
-        //   Control_signals {
-        //   2. DatatoReg (WB)
-        //   3. RegWrite (WB)
-        //   }
-     //   4. ALU_out (Addr_out outside) (used at both MEM and WB)
-     // Local:
-     wire [31:0] LoA_data;
-
-     assign Wt_addr[4:0] = MEM_WB_inst_in[11:7]; // rd, except for branch and store instructions
-     LUI_or_AUIPC _loa_ (
-         .inst_in(MEM_WB_inst_in[31:0]),
-         .PC(MEM_WB_PC),
-         .data(LoA_data[31:0])
-         );
-     Mux4to1b32  MUX3 (
-         .I0(MEM_WB_ALU_out[31:0]),          // Others
-         .I1(MEM_WB_Data_in[31:0]),          // Load
-         .I2(LoA_data[31:0]),                // LUI and AUIPC
-         .I3(MEM_WB_PC[31:0] + 32'b0100),    // jal and jalr: PC + 4
-         .s(MEM_WB_DatatoReg[1:0]),
-         .o(Wt_data[31:0]));
-
- endmodule
```

```
+     // EXE:---------------------------------------------------------------------------------------
+     // From ID:
+     //   1. inst_in
+         //   Control_signals {
+         //   2. ALU_Control (stops here)
+         //   3. mem_w
+         //   4. DatatoReg
+         //   5. RegWrite
+         //   }
+     //   6. ALU_A (stops here)
+     //   7. ALU_B (stops here)
+     //   8. Data_out
+     //   9. PC
+     // Control Signals:
+     //   1. ALU_Control
+     // References:
+     //   None
+     // Pass-on:
+     //   1. inst_in
+         //   Control_signals {
+         //   2. DatatoReg (WB)
+         //   3. mem_w (MEM)
+         //   4. RegWrite (WB)
+         //   }
+     //   5. Data_out (used at MEM together with mem_w)
+     //   6. ALU_out (Addr_out outside) (used at both MEM and WB)
+     //   7. PC
+     // Out:
+     //   None
+
+     // ALU Forwarding Multiplexer (A)
+     Mux4to1b32 _mux_forward_alu_a_ (
+         .I0(ID_EXE_ALU_A[31:0]),    // Wire from REG_ID_EXE
+         .I1(Wt_data[31:0]),         // Output from MEM_WB Register
+         .I2(EXE_MEM_ALU_out[31:0]), // Output from EXE_MEM Register
+         .s(forwarding_A_sig[1:0]),  // Forwarding signal
+         .o(ID_EXE_ALU_A_forward[31:0])   // Wire to ALU
+         );
+
+     // ALU Forwarding Multiplexer (B)
+     Mux4to1b32 _mux_forward_alu_b_ (
+         .I0(ID_EXE_ALU_B[31:0]),    // Wire from REG_ID_EXE
+         .I1(Wt_data[31:0]),         // Output from MEM_WB Register
+         .I2(EXE_MEM_ALU_out[31:0]), // Output from EXE_MEM Register
+         .s(forwarding_B_sig[1:0]),  // Forwarding signal
+         .o(ID_EXE_ALU_B_forward[31:0])   // Wire to ALU
+         );
+
+     // Data Out Forwarding Multiplexer
+     Mux4to1b32 _mux_forward_data_out_ (
+         .I0(ID_EXE_Data_out[31:0]),     // Wire from REG_ID_EXE
+         .I1(Wt_data[31:0]),             // Output from MEM_WB Register
+         .I2(EXE_MEM_ALU_out[31:0]),    // Output from EXE_MEM Register
+         .s(forwarding_B_sig[1:0]),   // Forwarding signal
+         .o(ID_EXE_Data_out_forward[31:0])   // Wire to EXE_MEM Register
+         );
+
+     Mux2to1b32 _alu_source_A_ (    // Moved from ID stage to EXE stage
+         .I0(ID_EXE_ALU_A_forward[31:0]),
+         .I1(ID_EXE_Imm_32[31:0]),   // not used
+         .s(ID_EXE_ALUSrc_A),
+         .o(ALU_A[31:0])
+         );
+
+     Mux4to1b32 _alu_source_B_ (    // Moved from ID stage to EXE stage
+         .I0(ID_EXE_ALU_B_forward[31:0]),
+         .I1(ID_EXE_Imm_32[31:0]),
+         .I2(),
+         .I3(),
+         .s(ID_EXE_ALUSrc_B[1:0]),
+         .o(ALU_B[31:0]
+         ));
+
+     ALU _alualu_ (
+         // Input:
+         .A(ALU_A[31:0]),
+         .B(ALU_B[31:0]),
+         .ALU_operation(ID_EXE_ALU_Control[4:0]),
+         .is_bne(ID_EXE_is_bne),
+         // Output:
+         .res(ID_EXE_ALU_out[31:0]),
+         .overflow(),
+         .zero_correct(zero_correct) // Now zero is calculated by ALU (connected to Branch Checker to determine if predicted correctly)
+         );
+
+     // Branch Checker
+     Branch_Checker _branch_checker_ (
+         // Input:
+         .zero_correct(zero_correct),
+         .ID_EXE_zero_prediction(ID_EXE_zero_prediction),
+         .OPcode(ID_EXE_inst_in[6:0]),
+
+         // Output:
+         .correct_prediction(correct_prediction),
+         .is_branch(is_branch)
+         );
+
+     REG_EXE_MEM _exe_mem_ (
+         .clk(clk), .rst(rst), .CE(V5),
+         // Input
+         .inst_in(ID_EXE_inst_in),
+         .PC(ID_EXE_PC),
+         //// To MEM stage
+         .ALU_out(ID_EXE_ALU_out),
+         .Data_out(ID_EXE_Data_out_forward),
+         .mem_w(ID_EXE_mem_w),
+         //// To WB stage
+         .DatatoReg(ID_EXE_DatatoReg),
+         .RegWrite(ID_EXE_RegWrite),
+
+         .written_reg(ID_EXE_written_reg), .read_reg1(ID_EXE_read_reg1), .read_reg2(ID_EXE_read_reg2),
+
+         // Output
+         .EXE_MEM_inst_in(EXE_MEM_inst_in),
+         .EXE_MEM_PC(EXE_MEM_PC),
+         .EXE_MEM_ALU_out(EXE_MEM_ALU_out),
+         .EXE_MEM_Data_out(EXE_MEM_Data_out),
+         .EXE_MEM_mem_w(EXE_MEM_mem_w),
+         .EXE_MEM_DatatoReg(EXE_MEM_DatatoReg),
+         .EXE_MEM_RegWrite(EXE_MEM_RegWrite),
+
+         .EXE_MEM_written_reg(EXE_MEM_written_reg), .EXE_MEM_read_reg1(EXE_MEM_read_reg1), .EXE_MEM_read_reg2(EXE_MEM_read_reg2)
+         );
```

```
+
+    // MEM:-------------------------------------------------------------------------------------------------
+    // From EXE:
+    //   1. inst_in
+        //   Control_signals {
+        //   2. DatatoReg (WB)
+        //   3. mem_w (stops here)
+        //   4. RegWrite (WB)
+        //   }
+    //   5. Data_out (stops here)
+    //   6. ALU_out (Addr_out outside) (used at both MEM and WB)
+    //   7. PC
+    // Control Signals:
+    //   1. mem_w
+    // Pass-on:
+    //   1. inst_in
+        //   Control_signals {
+        //   2. DatatoReg (WB)
+        //   3. RegWrite (WB)
+        //   }
+    //   4. ALU_out (Addr_out outside) (used at both MEM and WB)
+    //   5. PC
+    //   6. Data_in
+    // Out:
+    //   Data_out & mem_w, ALU_out(as Addr_out)
+
+    REG_MEM_WB _mem_wb_ (
+        .clk(clk), .rst(rst), .CE(V5),
+        // Input
+        .inst_in(EXE_MEM_inst_in),
+        .PC(EXE_MEM_PC),
+        .ALU_out(EXE_MEM_ALU_out),
+        .DatatoReg(EXE_MEM_DatatoReg),
+        .RegWrite(EXE_MEM_RegWrite),
+        .written_reg(EXE_MEM_written_reg),
+        //// Comes from data memory
+        .Data_in(data_in),
+
+        // Output
+        .MEM_WB_inst_in(MEM_WB_inst_in),
+        .MEM_WB_PC(MEM_WB_PC),
+        .MEM_WB_ALU_out(MEM_WB_ALU_out),
+        .MEM_WB_DatatoReg(MEM_WB_DatatoReg),
+        .MEM_WB_RegWrite(MEM_WB_RegWrite),
+        .MEM_WB_Data_in(MEM_WB_Data_in),
+        .MEM_WB_written_reg(MEM_WB_written_reg)
+        );
+
+    // WB:-------------------------------------------------------------------------------------------------
+    // From EXE:
+    //   1. inst_in
+       //   Control_signals {
+        //   2. DatatoReg (WB)
+        //   3. RegWrite (WB)
+        //   }
+    //   4. ALU_out (Addr_out outside) (used at both MEM and WB)
+    // Local:
+    wire [31:0] LoA_data;
+
+    assign Wt_addr[4:0] = MEM_WB_inst_in[11:7]; // rd, except for branch and store instructions
+    LUI_or_AUIPC _loa_ (
+        .inst_in(MEM_WB_inst_in[31:0]),
+        .PC(MEM_WB_PC),
+        .data(LoA_data[31:0])
+        );
+    Mux4to1b32  MUX3 (
+        .I0(MEM_WB_ALU_out[31:0]),          // Others
+        .I1(MEM_WB_Data_in[31:0]),          // Load
+        .I2(LoA_data[31:0]),                // LUI and AUIPC
+        .I3(MEM_WB_PC[31:0] + 32'b0100),    // jal and jalr: PC + 4
+        .s(MEM_WB_DatatoReg[1:0]),
+        .o(Wt_data[31:0]));
+
+    // Forward Unit
+    Forward_Unit _forward_unit_ (
+        // Input:
+        .ID_EXE_read_reg1(ID_EXE_read_reg1),
+        .ID_EXE_read_reg2(ID_EXE_read_reg2),
+
+        .EXE_MEM_RegWrite(EXE_MEM_RegWrite),
+        .EXE_MEM_written_reg(EXE_MEM_written_reg),
+
+        .MEM_WB_RegWrite(MEM_WB_RegWrite),
+        .MEM_WB_written_reg(MEM_WB_written_reg),
+
+        // Output:
+        .forwarding_A_sig(forwarding_A_sig),
+        .forwarding_B_sig(forwarding_B_sig)
+        );
+
+ endmodule
```

ID_Zero_Generator.v is no longer used.

It should be noted that some comments in the original code, mostly those that specify the pipeline structure and order, were not modified and do not represent the final structure of the CPU. Thus, they should not be taken as a reference.

9. Appendix A: Vec_Mul.asm

```
// Hint 1: A label looks like this:
Start:
// Hint 2: you can use '//' or '#' to make comments

// In Memory
//// Vector A: length 10
//// then Vector B: length 10
//// then Vector C: length 10
//// This program plays C = A * B

addi s1, x0, 4  # A matrix base address
addi s2, s1, 40 # B matrix base address
addi s3, s2, 40 # C matrix base address
addi s4, s3, 40 # end of C matrix

// 1. Prepare data in A
addi t3, x0, 1 # value to be initialized to vector A (1 ~ 10)
addi t1, s1, 0 # for (i = 0; ...
InitA:

sw t3, 0(t1)   # A[i] = i;

addi t1, t1, 4 # ... i += 1)
addi t3, t3, 1 # increment value
slt t2, t1, s2 # ... i < 10; ...
bne t2, x0, InitA

// 2. Prepare data in B
addi t3, x0, 1 # value to be initialized to vector B (also 1 ~ 10)
addi t1, s2, 0 # for (i = 0; ...
InitB:

sw t3, 0(t1)   # B[i] = i;

addi t1, t1, 4 # ... i += 1)
addi t3, t3, 1 # increment value
slt t2, t1, s3 # ... i < 10; ...
bne t2, x0, InitB

// 3. Do `C = A * B`, element by element
addi t1, s1, 0  # -----
addi t2, s2, 0
addi t3, s3, 0
VecMul:

lw a0, 0(t1)    # ---
lw a1, 0(t2)
addi s5, x0, 0
addi t5, x0, 0
Mul:

add s5, s5, a0

addi t5, t5, 1
slt t6, t5, a1
bne t6, x0, Mul # ---

sw s5, 0(t3)

addi t1, t1, 4
addi t2, t2, 4
addi t3, t3, 4
slt t4, t3, s4
bne t4, x0, VecMul # -----
```

```
// 4. set memory[0] to 1, which means finished
addi t1, x0, 1
sw t1, 0(x0)
```

10. Appendix B: Vec_Mul.hex

```
00400493
02848913
02890993
02898a13
00100e13
00048313
01c32023
00430313
001e0e13
012323b3
fe0398e3
00100e13
00090313
01c32023
00430313
001e0e13
013323b3
fe0398e3
00048313
00090393
00098e13
00032503
0003a583
00000a93
00000f13
00aa8ab3
001f0f13
00bf2fb3
fe0f9ae3
015e2023
00430313
00438393
004e0e13
014e2eb3
fc0e96e3
00100313
00602023
```

11. Appendix C: Jacobi-1d.asm

```
main:
addi s0, x0, 10 // int n = N (N==10);
addi s1, x0, 5 // int tsteps = 5;
addi s2, x0, 0x200 // starting addr of array A
addi s3, x0, 0x300 // starting addr of array B

jal ra, init_array  // 0b0000ef

jal ra, kernel      // 008000ef

jal ra, continue    // 0d8000ef

// ----------------------------
kernel:
addi t4, s0, -1     // fff40e93
addi t0, x0, 0
k_loop_t:

addi t1, x0, 1
k_loop_i1:
addi t2, t1, -1
slli t2, t2, 2
add t2, t2, s2
lw t3, 0(t2)
add t5, t3, x0
addi t2, t2, 4
lw t3, 0(t2)
add t5, t5, t3
addi t2, t2, 4
lw t3, 0(t2)
add t5, t5, t3

addi t2, t1, 0
slli t2, t2, 2
add t2, t2, s3
sw t5, 0(t2)
addi t1, t1, 1
bne t1, t4, k_loop_i1

addi t1, x0, 1
k_loop_i2:
addi t2, t1, -1
slli t2, t2, 2
add t2, t2, s3
lw t3, 0(t2)
add t5, t3, x0
addi t2, t2, 4
lw t3, 0(t2)
add t5, t5, t3
addi t2, t2, 4
lw t3, 0(t2)
add t5, t5, t3

addi t2, t1, 0
slli t2, t2, 2
add t2, t2, s2
sw t5, 0(t2)
addi t1, t1, 1
bne t1, t4, k_loop_i2

addi t0, t0, 1
bne t0, s1, k_loop_t

jalr x0, ra, 0
```

```
// ----------------------------

// ----------------------------
init_array:
addi t0, x0, 0 // t0: i         // 00000293
addi t3, s2, 0 // addr of A
addi t4, s3, 0 // addr of B
init_loop_i:
addi t1, t0, 2 // data for A[i]
sw t1, 0(t3) // A[i] = i + 2
addi t1, t1, 1 // data for B[i]
sw t1, 0(t4) // B[i] = i + 3
addi t3, t3, 4
addi t4, t4, 4
addi t0, t0, 1
bne t0, s0, init_loop_i         // fe8292e3
jalr x0, ra, 0                  // 00008067
// ----------------------------

continue:
// now A[8] == 146744
lw t0, 32(s2) // t0 = A[8]
srli t0, t0, 9 // t0 = 146744 >> 9 = 286 = 0x11e
addi t0, t0, -285
sw t0, 0(x0)
```

12. Appendix D: Jacobi-1d.hex

```
00a00413
00500493
20000913
30000993
0b0000ef
008000ef
0d8000ef
fff40e93
00000293
00100313
fff30393
00239393
012383b3
0003ae03
000e0f33
00438393
0003ae03
01cf0f33
00438393
0003ae03
01cf0f33
00030393
00239393
013383b3
01e3a023
00130313
fdd310e3
00100313
fff30393
00239393
013383b3
0003ae03
000e0f33
00438393
0003ae03
01cf0f33
00438393
0003ae03
01cf0f33
00030393
00239393
012383b3
01e3a023
00130313
fdd310e3
00128293
f69296e3
00008067
00000293
00090e13
00098e93
00228313
006e2023
00130313
006ea023
004e0e13
004e8e93
00128293
fe8292e3
00008067
02092283
0092d293
ee328293
00502023
```