

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

Abstract	1
Acknowledgements	4
1. Introduction.....	5
2. Related works.....	6
3. Methods.....	7
3.1. Data forwarding	7
3.1.1. Forward Unit	7
3.1.2. Data forwarding flow.....	8
3.1.3. Eliminated data stalls	9
3.2. Branch prediction	9
3.2.1. Branch prediction pipeline.....	9
3.2.2. Branch Predictor.....	11
3.2.3. Branch Checker	12
3.2.4. Eliminated control stalls.....	12
4. Methodology	13
5. Evaluation.....	14
5.1. Data forwarding	14
5.1.1. Before data forwarding.....	14
5.1.2. After data forwarding.....	15
5.2. Branch prediction	16
5.2.1. Correct branch prediction.....	16
5.2.2. Incorrect branch prediction	17
5.2.3. JAL instruction	19
5.2.4. JALR instruction	20
5.3. Overall speedup.....	21
6. Conclusions	22
7. References	23
8. Changes in the source code	24
9. Appendix A: Vec_Mul.asm.....	40
10. Appendix B: Vec_Mul.hex.....	42
11. Appendix C: Jacobi-1d.asm.....	43

12.	Appendix D: Jacobi-1d.hex	45
-----	---------------------------------	----

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.

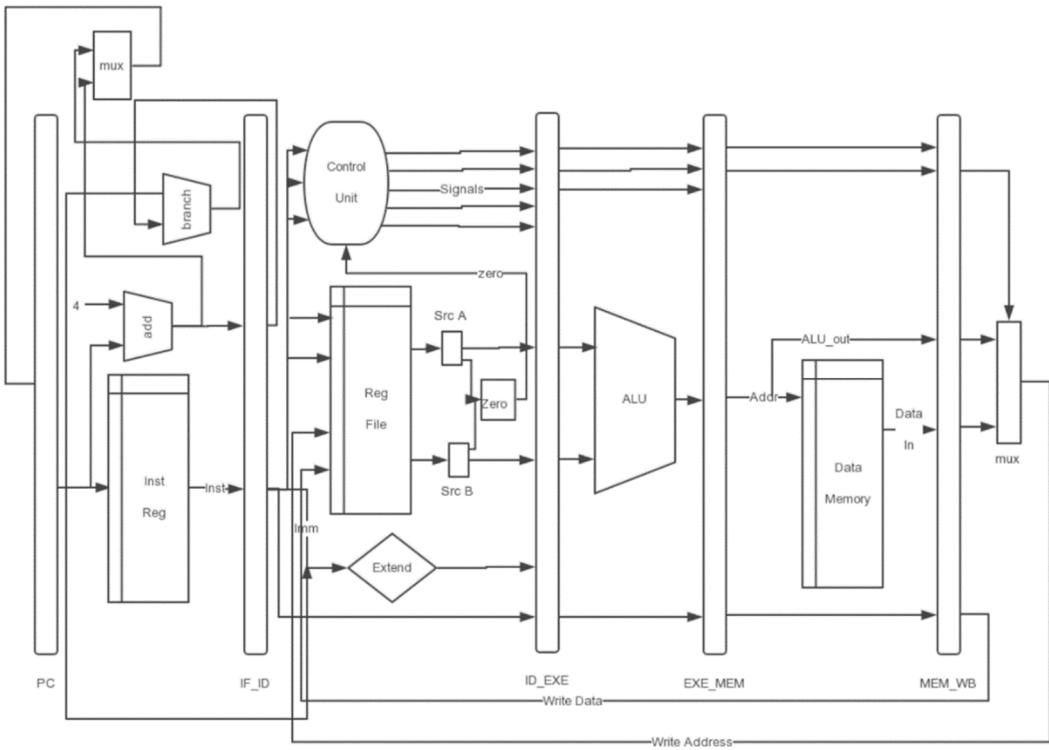


Figure 1: Simplified diagram of the microarchitecture of the base processor

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.

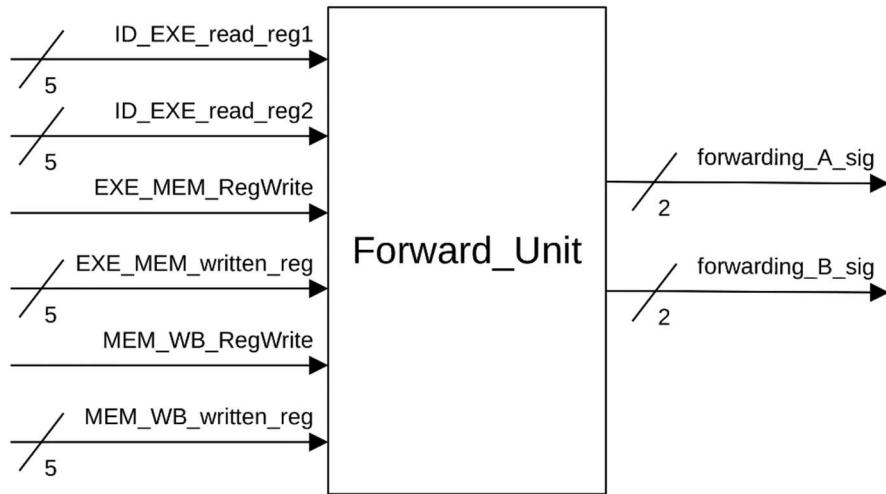


Figure 2: Forward Unit diagram

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

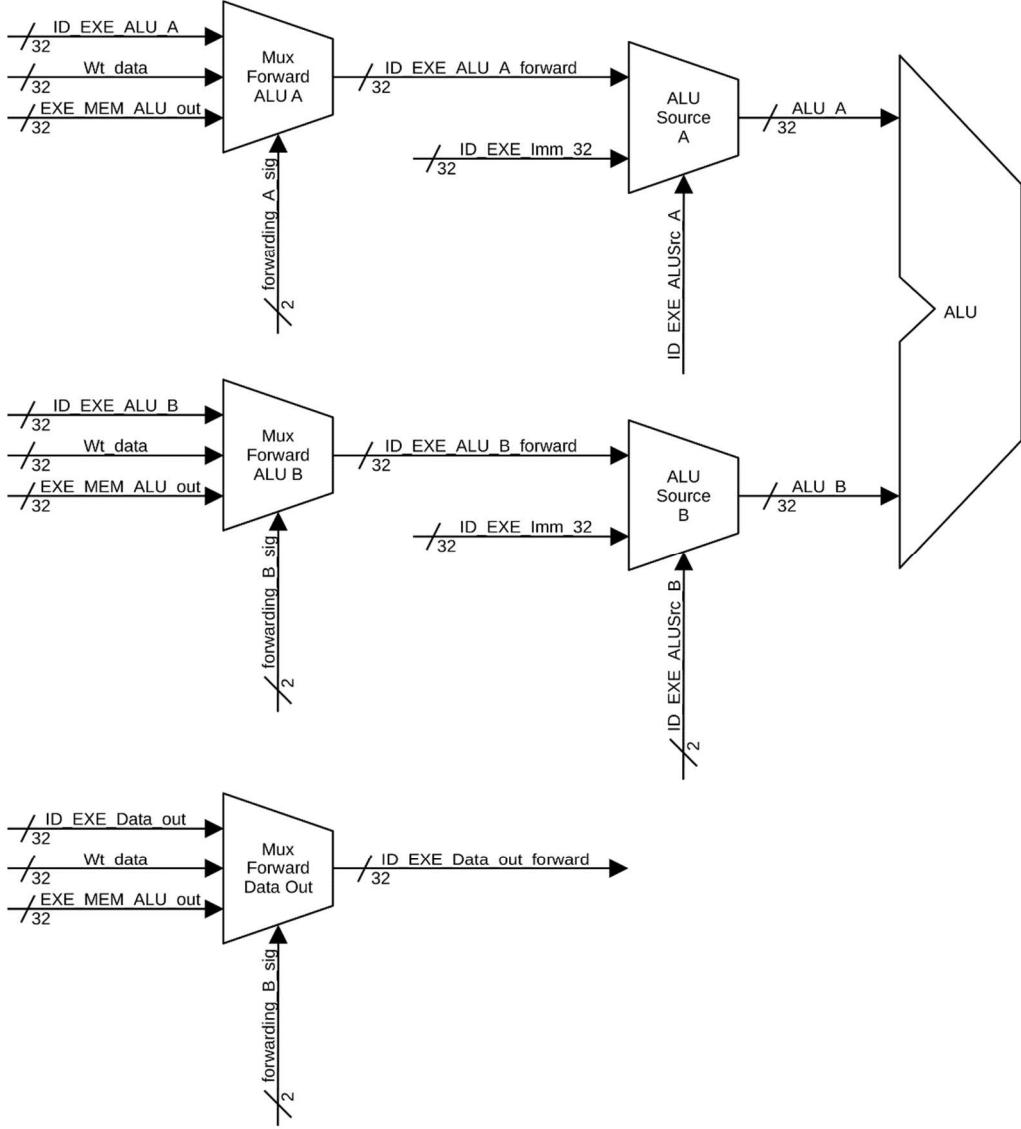


Figure 3: Forwarding multiplexers diagram

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.

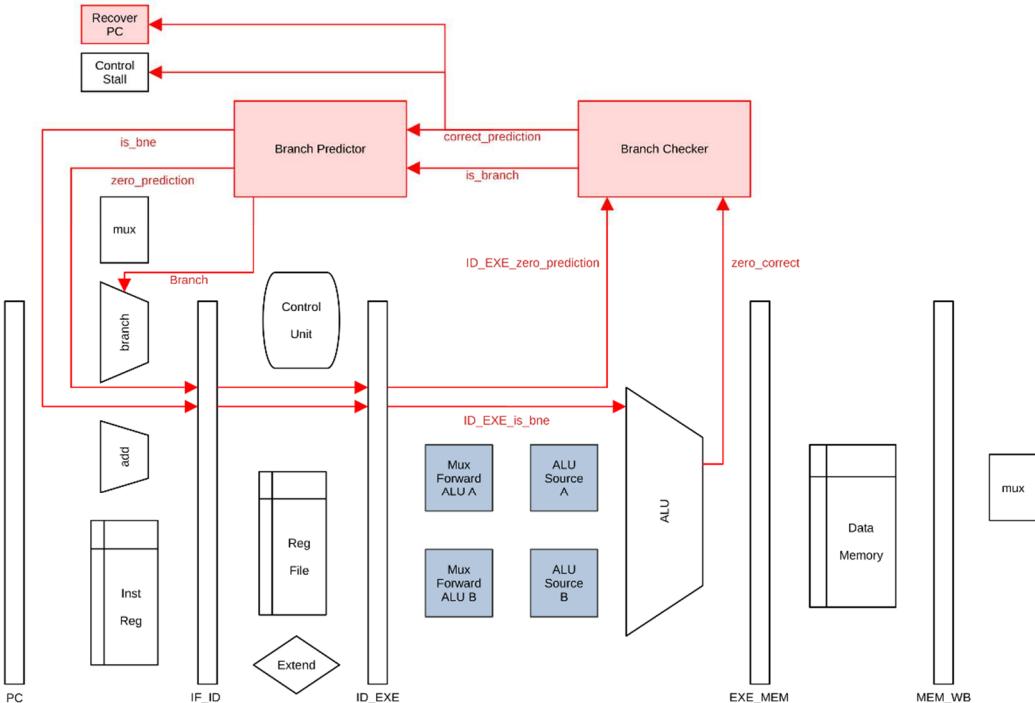


Figure 4: Simplified diagram for the branch prediction pipeline

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

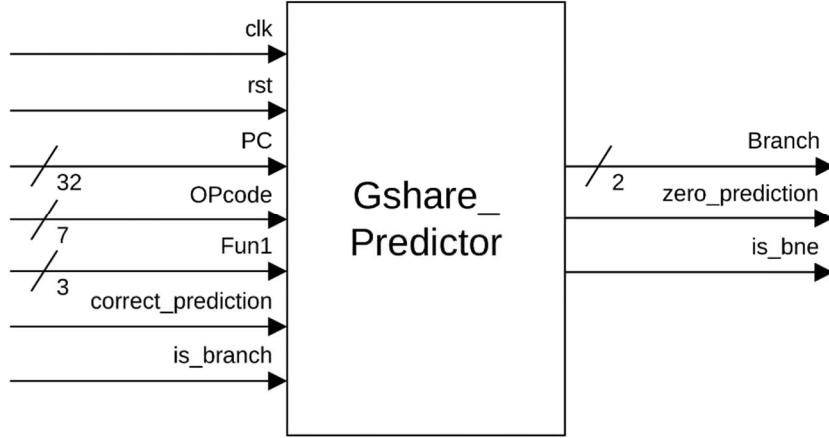


Figure 5: Branch Predictor diagram

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

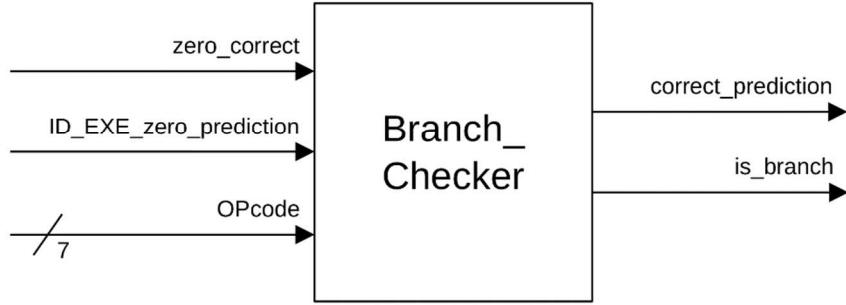


Figure 6: Branch Checker diagram

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

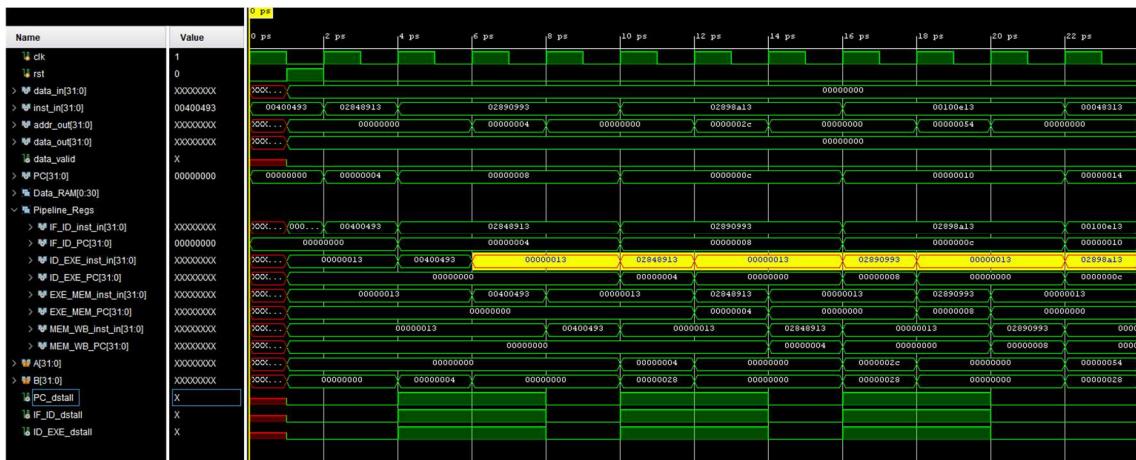


Figure 7: Simulation before implementing data forwarding

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

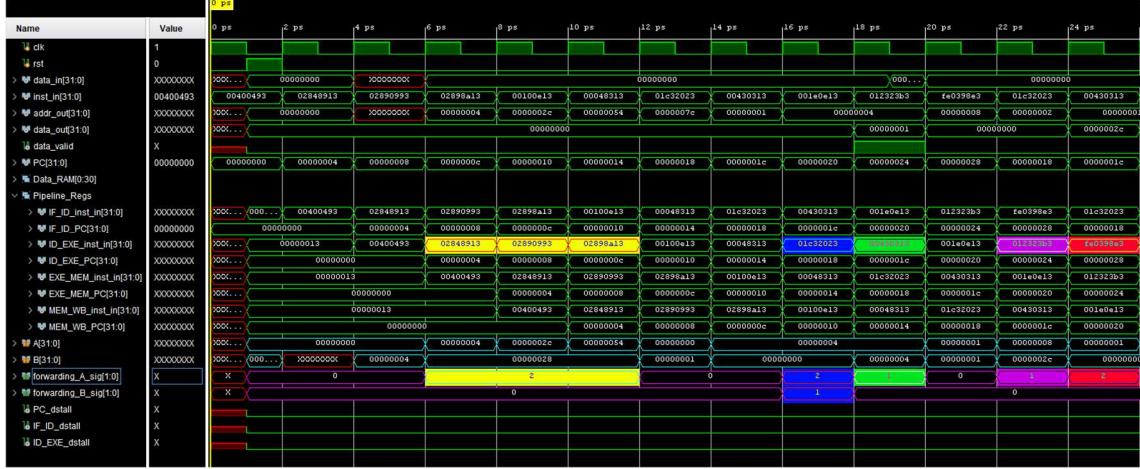


Figure 8: Simulation after implementing data forwarding

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_MEMORY 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_MEMORY 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_MEMORY 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

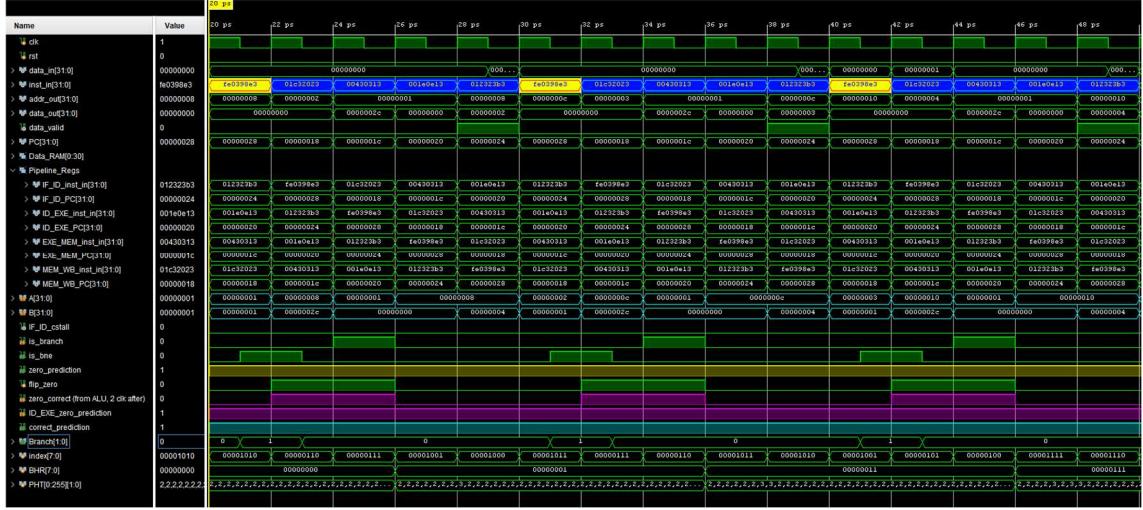


Figure 9: Simulation of a correct branch prediction

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

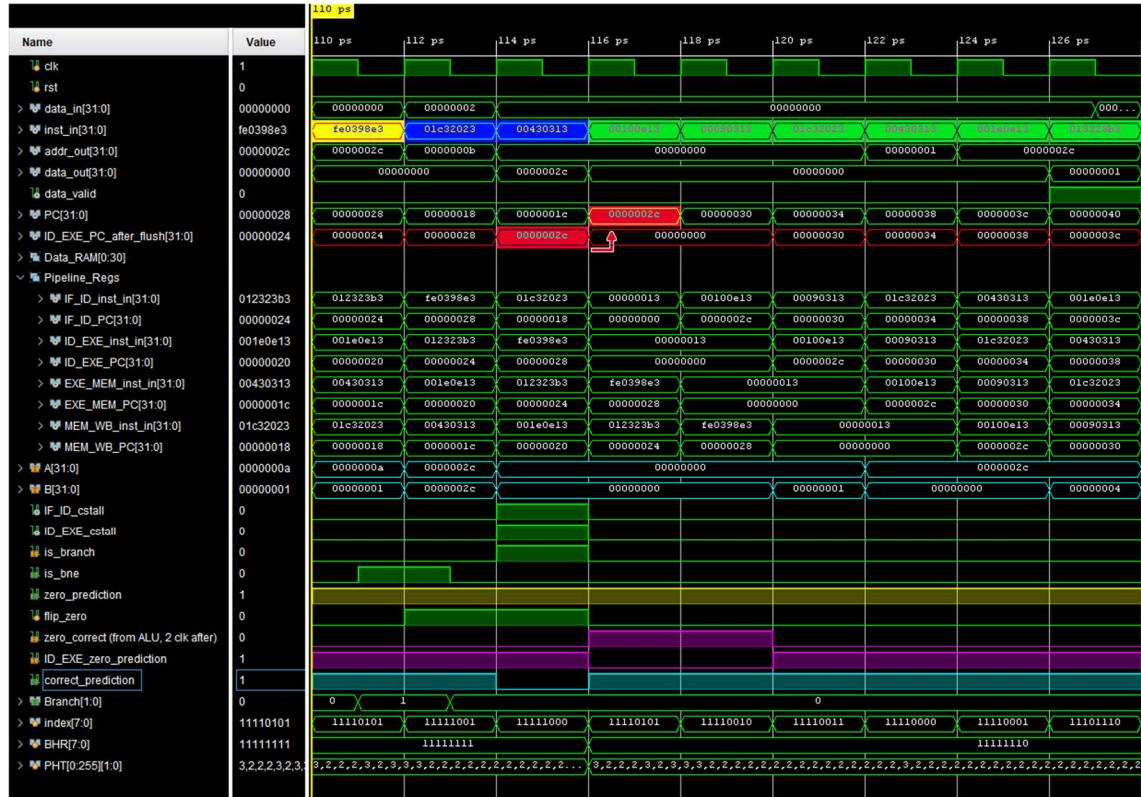


Figure 10: Simulation of a misprediction

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

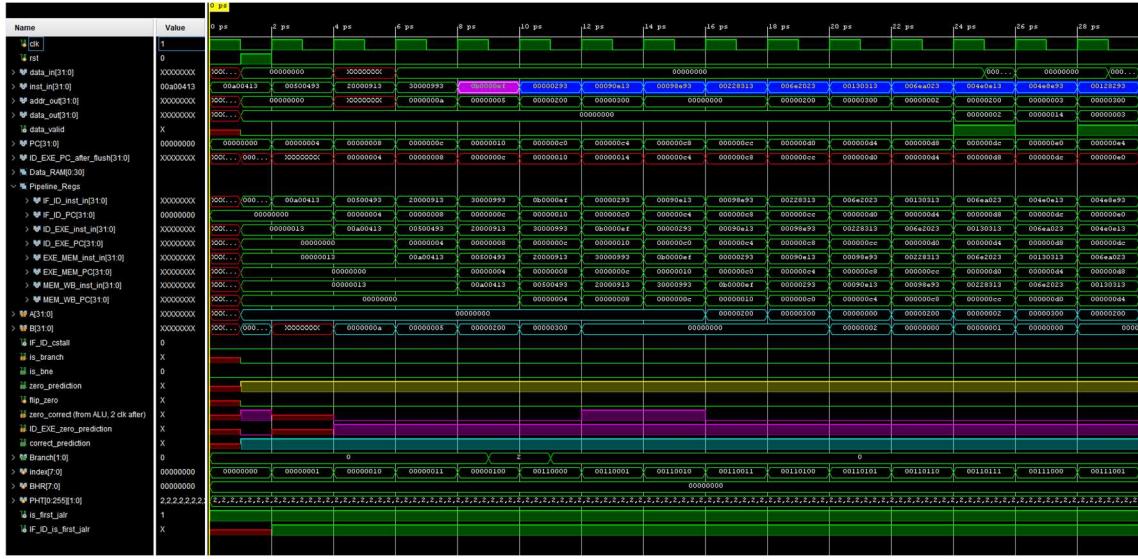


Figure 11: Simulation of a JAL instruction

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 // 0b00000ef

    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

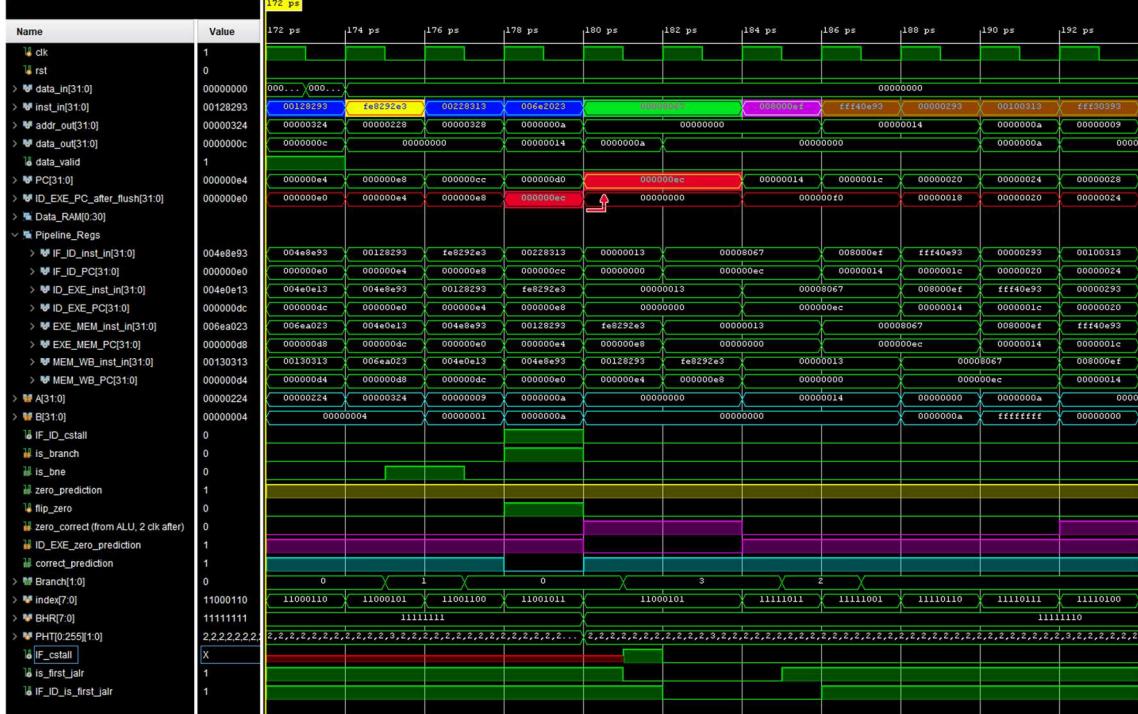


Figure 12: Simulation of a JALR instruction

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 // 0b00000ef

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]

```

... @@ -1,97 +1,105 @@
1 - `timescale 1ps / 1ps
2 - ///////////////////////////////////////////////////////////////////
3 - // Company:
4 - // Engineer:
5 - //
6 - // Create Date: 2021/03/10 19:39:55
7 - // Design Name:
8 - // Module Name: ALU
9 - // Project Name:
10 - // Target Devices:
11 - // Tool Versions:
12 - // Description:
13 - //
14 - // Dependencies:
15 - //
16 - // Revision:
17 - // Revision 0.01 - File Created
18 - // Additional Comments:
19 - //
20 - ///////////////////////////////////////////////////////////////////
21 -
22 -
23 - module ALU(
24 -     input [31:0] A,
25 -     input [31:0] B,
26 -     input [4:0] ALU_operation,
27 -     output reg signed [31:0] res,
28 -     output reg overflow,
29 -     output wire zero
30 - );
31 -
32 -     wire res_temp;
33 -     assign res_temp = res;
34 -     parameter one = 32'h00000001, zero_0 = 32'h00000000;
35 -     wire signed [31:0] A_temp, B_temp;
36 -     assign A_temp = A;
37 -     assign B_temp = B;
38 -         res = A & B;
39 -         overflow = 0;
40 -     end
41 -     5'b00001: begin // on
42 -         res = A | B;
43 -         overflow = 0;
44 -     end
45 -     5'b00010: begin // add
46 -         res = A_temp + B_temp;
47 -         if ((A[31] == 1 && B[31] == 0) || (A[31] == 0 && B[31]
48 - == 0 && res[31] == 1))
49 -             overflow = 1;
50 -         else overflow = 0;
51 -     end
52 -     5'b00011: begin // sub
53 -         res = A_temp - B_temp;
54 -         if ((A[31] == 1 && B[31] == 0 && res[31] == 0) || (A[31] == 0 && B[31]
55 - == 1 && res[31] == 1))
56 -             overflow = 1;
57 -         else overflow = 0;
58 -     end
59 -     5'b00100: begin // XOR
60 -         res = A ^ B;
61 -         overflow = 0;
62 -     end
63 -     5'b00101: begin // SLT
64 -         res = (A_temp < B_temp) ? one : zero_0;
65 -         overflow = 0;
66 -     end
67 -     5'b00110: begin // SLTU
68 -         res = (A < B) ? one : zero_0;
69 -         overflow = 0;
70 -     end
71 -     5'b00111: begin // SLL
72 -         res = (A << B);
73 -         overflow = 0;
74 -     end
75 -     5'b01000: begin // SRL
76 -         res = (A >> B);
77 -         overflow = 0;
78 -     end
79 -     5'b01001: begin // SRA
80 -         res = (A_temp >> B);
81 -         overflow = 0;
82 -     end
83 -     5'b01010: begin // BGE
84 -         res = (A_temp >= B_temp) ? one : zero_0;
85 -         overflow = 0;
86 -     end
87 -     5'b01011: begin // BGEU
88 -         res = (A > B) ? one : zero_0;
89 -         overflow = 0;
90 -     end
91 -     end
92 -         default: res = 32'hx;
93 -     endcase
94 - end
95 - assign zero = (res == 0) ? 1 : 0;
96 -
97 - endmodule
... + `timescale 1ps / 1ps
2 + ///////////////////////////////////////////////////////////////////
3 + // Company:
4 + // Engineer:
5 + //
6 + // Create Date: 2021/03/10 19:39:55
7 + // Design Name:
8 + // Module Name: ALU
9 + // Project Name:
10 + // Target Devices:
11 + // Tool Versions:
12 + // Description:
13 + //
14 + // Dependencies:
15 + //
16 + // Revision:
17 + // Revision 0.01 - File Created
18 + // Additional Comments:
19 + //
20 + ///////////////////////////////////////////////////////////////////
21 +
22 +
23 + module ALU(
24 +     input [31:0] A,
25 +     input [31:0] B,
26 +     input [4:0] ALU_operation,
27 +     input is_bne,
28 +         output reg signed [31:0] res,
29 +         output reg overflow,
30 +         output wire zero_correct // If zero_correct==1 and it is a branch instruction, then the
branch should have been taken
31 + );
32 +     reg flip_zero; // If flip_zero==1, we need to flip the zero signal before outputting
33 +     wire res_temp;
34 +     assign res_temp = res;
35 +     parameter one = 32'h00000001, zero_0 = 32'h00000000;
36 +     wire signed [31:0] A_temp, B_temp;
37 +     assign flip_zero = 0;
38 +         case (ALU_operation)
39 +             5'b00000: begin // and
40 +                 res = A & B;
41 +                 overflow = 0;
42 +             end
43 +             5'b00001: begin // or
44 +                 res = A | B;
45 +                 overflow = 0;
46 +             end
47 +             5'b00010: begin // add
48 +                 res = A_temp + B_temp;
49 +                 if ((A[31] == 1 && B[31] == 0 && res[31] == 0) || (A[31] == 0 && B[31]
50 + == 0 && res[31] == 1))
51 +                     overflow = 1;
52 +                 else overflow = 0;
53 +             end
54 +             5'b00011: begin // sub
55 +                 res = A_temp - B_temp;
56 +                 if ((A[31] == 1 && B[31] == 0 && res[31] == 0) || (A[31] == 0 && B[31]
57 + == 1 && res[31] == 1))
58 +                     overflow = 1;
59 +                 else overflow = 0;
60 +             end
61 +             5'b00100: begin // sub (and BNE and BNE)
62 +                 res = A_temp - B_temp;
63 +                 if ((A[31] == 1 && B[31] == 0 && res[31] == 0) || (A[31] == 0 && B[31]
64 + == 1 && res[31] == 1))
65 +                     overflow = 1;
66 +                 else overflow = 0;
67 +                 // If it is a BNE, flip the zero signal
68 +                 if (is_bne == 1'b1)
69 +                     flip_zero = 1;
70 +             end
71 +             5'b00101: begin // SLT (and BLT)
72 +                 res = (A_temp < B_temp) ? one : zero_0;
73 +                 flip_zero = 1; // For BLT
74 +                 overflow = 0;
75 +             end
76 +             5'b00110: begin // SLTU (and BLTU)
77 +                 res = (A < B) ? one : zero_0;
78 +                 flip_zero = 1; // For BLTU
79 +                 overflow = 0;
80 +             end
81 +             5'b00111: begin // SLL
82 +                 res = (A << B);
83 +                 overflow = 0;
84 +             end
85 +             5'b01000: begin // SRL
86 +                 res = (A >> B);
87 +                 overflow = 0;
88 +             end
89 +             5'b01001: begin // SRA
90 +                 res = (A_temp >> B);
91 +                 overflow = 0;
92 +             end
93 +             5'b01010: begin // BGE
94 +                 res = (A_temp >= B_temp) ? zero_0 : one;
95 +                 overflow = 0;
96 +             end
97 +             5'b01011: begin // BGEU
98 +                 res = (A > B) ? zero_0 : one;
99 +                 overflow = 0;
100 +             end
101 +         endcase
102 +         default: res = 32'hx;
103 +     end
104 +     assign zero_correct = (res == 0) ? -flip_zero : flip_zero;
105 + endmodule

```

```

49 RV32I/RV32I.srcts/sources_1/new/Branch_Checker.v ...
... @@ -0,0 +1,49 @@
1 + `timescale 1ns / 1ps
2 + //////////////////////////////////////////////////////////////////
3 + // Company:
4 + // Engineer:
5 + //
6 + // Create Date: 10.05.2023 22:09:27
7 + // Design Name:
8 + // Module Name: Branch_Checker
9 + // Project Name:
10 + // Target Devices:
11 + // Tool Versions:
12 + // Description:
13 + //
14 + // Dependencies:
15 + //
16 + // Revision:
17 + // Revision 0.01 - File Created
18 + // Additional Comments:
19 + //
20 + //////////////////////////////////////////////////////////////////
21 +
22 +
23 + module Branch_Checker(
24 +   // Input:
25 +   input zero_correct,
26 +   input ID_EXE_zero_prediction,
27 +   input [6:0] OPCODE,
28 +
29 +   // Output:
30 +   output reg correct_prediction,
31 +   output reg is_branch
32 + );
33 +
34 + always @(*) begin
35 +   // If ID_EXE_ALU_Control is 7'b1100011, then the instruction is a branch and we need to check
36 +   // the prediction
37 +   if (OPCODE == 7'b1100011) begin
38 +     is_branch = 1'b1;
39 +     if (zero_correct == ID_EXE_zero_prediction) begin
40 +       correct_prediction = 1'b1;
41 +     end else begin
42 +       correct_prediction = 1'b0;
43 +     end
44 +   end else begin
45 +     is_branch = 1'b0;
46 +     correct_prediction = 1'b1;
47 +   end
48 +
49 + endmodule

```

```

91 RV32I/RV32I.srcts/sources_1/new/Control_Stall.v ...
... @@ -1,33 +1,58 @@
1 - `timescale 1ps / 1ps
2 - //////////////////////////////////////////////////////////////////
3 - // Company:
4 - // Engineer:
5 - //
6 - // Create Date: 2021/03/12 08:45:29
7 - // Design Name:
8 - // Module Name: Control_Stall
9 - // Project Name:
10 - // Target Devices:
11 - // Tool Versions:
12 - // Description:
13 - //
14 - // Dependencies:
15 - //
16 - // Revision:
17 - // Revision 0.01 - File Created
18 - // Additional Comments:
19 - //
20 - //////////////////////////////////////////////////////////////////
21 -
22 -
23 - module Control_Stall(
24 -   input [1:0] Branch,
25 -   output reg IF_ID_cstall
26 - );
27 -   always @ (*) begin
28 -     IF_ID_cstall = 1'b0;
29 -     if (Branch[1:0] != 2'b00) begin
30 -       IF_ID_cstall = 1'b1;
31 -     end
32 -   end
33 - endmodule
1 + `timescale 1ps / 1ps
2 + //////////////////////////////////////////////////////////////////
3 + // Company:
4 + // Engineer:
5 + //
6 + // Create Date: 2021/03/12 08:45:29
7 + // Design Name:
8 + // Module Name: Control_Stall
9 + // Project Name:
10 + // Target Devices:
11 + // Tool Versions:
12 + // Description:
13 + //
14 + // Dependencies:
15 + //
16 + // Revision:
17 + // Revision 0.01 - File Created
18 + // Additional Comments:
19 + //
20 + //////////////////////////////////////////////////////////////////
21 -
22 -
23 + module Control_Stall(
24 +   input [1:0] Branch,
25 +   output reg IF_ID_is_first_jalr, // 1 if it was the first jalr out of the 2 in the bubble, 0
26 +   otherwise
27 +   input correct_prediction,
28 +   output reg IF_ID_cstall, ////////////////////////////////////////////////////////////////// ADD to main file
29 +   output reg IF_ID_cstall,
30 +   output reg ID_EXE_cstall,
31 +   output reg PC_cstall, // Unused
32 +   output reg is_first_jalr
33 + );
34 +   // Stalls for flushing the pipeline when the branch prediction is wrong
35 +   always @ (*) begin
36 +     IF_ID_cstall = 1'b0;
37 +     ID_EXE_cstall = 1'b0;
38 +     PC_cstall = 1'b0;
39 +     is_first_jalr = 1'b1;
40 +     // Stall IF_ID and ID_EXE if the branch prediction was wrong
41 +     if (correct_prediction == 1'b0) begin
42 +       IF_ID_cstall = 1'b1; // 1 to stall
43 +       ID_EXE_cstall = 1'b1; // 1 to stall
44 +     end
45 +     // Stall IF_ID if the branch was a JALR instruction
46 +     // We will only stall one, the second time that the JALR instruction is detected,
47 +     // Stall If the branch was a JALR instruction
48 +     else if (Branch == 2'b11) begin
49 +       if (IF_ID_is_first_jalr == 1'b1) begin
50 +         IF_cstall = 1'b1; // 1 to stall
51 +         is_first_jalr = 1'b0;
52 +       end else begin
53 +         IF_cstall = 1'b0; // 0 to not stall
54 +         is_first_jalr = 1'b0;
55 +       end
56 +     end
57 +   end
58 + endmodule

```

```

... @@ 1,209 +1,200 @@
1 - `timescale 1ps / 1ps
2 - ///////////////////////////////////////////////////////////////////
3 - // Company:
4 - // Engineer:
5 - //
6 - // Create Date: 2021/03/10 21:16:48
7 - // Design Name:
8 - // Module Name: Controller
9 - // Project Name:
10 - // Target Devices:
11 - // Tool Versions:
12 - // Description:
13 - //
14 - // Dependencies:
15 - //
16 - // Revision:
17 - // Revision 0.01 - File Created
18 - // Additional Comments:
19 - //
20 - ///////////////////////////////////////////////////////////////////
21 -
22 -
23 - module Controller(
24 -     input [6:0] OPcode,
25 -     input [2:0] Fun1,
26 -     input [6:0] Fun2,
27 -     input wire zero,
28 -     output reg ALUSrc_A,
29 -     output reg [1:0] ALUsrc_B,
30 -     output reg [1:0] DataReg,
31 -     output reg [1:0] Branch,
32 -     output reg Regwrite,
33 -     output reg mem_W,
34 -     output reg [4:0] ALU_Control,
35 -     output reg [1:0] B_M_W,
36 -     output reg sign
37 - );
38 -     always @(*) begin
39 -         ALUSrc_B = 0;
40 -         ALUSrc_A = 0;
41 -         DataReg = 2'b0;
42 -         DataReg = 2'b0;
43 -         Branch = 0;
44 -         Regwrite = 0;
45 -         mem_W = 0;
46 -         B_M_W = 2'b0; // default: immediate is a word
47 -         sign = 1'b1; // default: signed extension to "write_data"
48 -         case(OPcode)
49 -             // R
50 -             7'b0110001: begin
51 -                 Regwrite = 1;
52 -                 case(Fun1)
53 -                     3'b000: begin
54 -                         case(Fun2)
55 -                             7'b0000000: ALU_Control = 5'b00010; // ADD
56 -                             7'b0000000: ALU_Control = 5'b00011; // SUB
57 -                             default: ALU_Control = 5'b11111;
58 -                         endcase
59 -                     end
60 -                     3'b001: begin // SLL
61 -                         ALU_Control = 5'b00111;
62 -                     end
63 -                     3'b010: ALU_Control = 5'b00101; // SLT
64 -                     3'b011: ALU_Control = 5'b00110; // SLTU
65 -                     3'b100: ALU_Control = 5'b00100; // XOR
66 -                     3'b101: begin
67 -                         case(Fun2)
68 -                             7'b0000000: begin // SRL
69 -                                 ALU_Control = 5'b01000;
70 -                             end
71 -                             7'b0100000: begin // SRA
72 -                                 ALU_Control = 5'b01001;
73 -                             end
74 -                             default: ALU_Control = 5'b11111;
75 -                         endcase
76 -                     end
77 -                     3'b110: ALU_Control = 5'b00001; // OR
78 -                     3'b111: ALU_Control = 5'b00000; // AND
79 -                     default: ALU_Control = 5'b11111;
80 -                 endcase
81 -             // I
82 -             7'b0001001: begin
83 -                 Regwrite = 1;
84 -                 case (Fun1)
85 -                     3'b000: ALU_Control = 5'b00010; // ADDI
86 -                     ALUSrc_B = 2'b001;
87 -                 end
88 -                 3'b010: begin
89 -                     ALU_Control = 5'b000101; // SLTI
90 -                     ALUSrc_B = 2'b01;
91 -                 end
92 -                 3'b011: begin
93 -                     ALU_Control = 5'b00010; // SLTIU
94 -                     ALUSrc_B = 2'b01;
95 -                 end
96 -                 3'b100: begin
97 -                     ALU_Control = 5'b000100; // XORI
98 -                     ALUSrc_B = 2'b001;
99 -                 end
100 -                 3'b110: begin
101 -                     ALU_Control = 5'b00001; // ORI
102 -                     ALUSrc_B = 2'b001;
103 -                 end
104 -                 3'b111: begin
105 -                     ALU_Control = 5'b00000; // ANDI
106 -                     ALUSrc_B = 2'b00000;
107 -                     ALUSrc_B = 2'b001;
108 -                 end
109 -                 3'b000: begin
110 -                     ALU_Control = 5'b00011; // SLLI
111 -                     ALUSrc_B = 2'b001;
112 -                 end
113 -                 3'b101: begin
114 -                     ALUSrc_B = 2'b001;
115 -                     case (Fun2)
116 -                         7'b0000000: ALU_Control = 5'b01000; // SRLI
117 -                         7'b0100000: ALU_Control = 5'b01001; // SRAI
118 -                     endcase
119 -                 end
120 -             endcase

```

```

121 -         end
122 -         7'b00000011: begin      // 1
123 -             ALU_Control = 5'b000010;
124 -             ALUSrc_B = 2'b01;
125 -             DataToReg = 2'b01;
126 -             RegWrite = 1;
127 -             case (Fun1)
128 -                 3'b000: begin // LB
129 -                     B_M_W = 2'b01; // byte
130 -                 end
131 -                 3'b001: begin // LH
132 -                     B_M_W = 2'b10; // half word
133 -                 end
134 -                 3'b100: begin // LBU
135 -                     B_M_W = 2'b01; // byte
136 -                     sign = 1'b0;
137 -                 end
138 -                 3'b101: begin // LHU
139 -                     B_M_W = 2'b10; // half word
140 -                     sign = 1'b0;
141 -                 end
142 -                 // 3'b010; // LW
143 -             endcase
144 -         end
145 -         7'b01000011: begin // S
146 -             ALU_Control = 5'b000010;
147 -             ALUSrc_B = 2'b01;
148 -             mem_W = 1;
149 -             case (Fun1)
150 -                 3'b000: begin
151 -                     B_M_W = 2'b01; // byte
152 -                 end
153 -                 3'b001: begin
154 -                     B_M_W = 2'b10; // half word
155 -                 end
156 -                 // 3'b010; // SW
157 -             endcase
158 -         end
159 -         7'b11000011: begin      // Branch
160 -             case (Fun1)
161 -                 3'b000: begin // BEQ
162 -                     ALU_Control = 5'b000011;
163 -                     Branch = {1'b0, zero};
164 -                 end
165 -                 3'b001: begin // DNE
166 -                     ALU_Control = 5'b000011;
167 -                     Branch = {1'b0, ~zero};
168 -                 end
169 -                 3'b100: begin // BLT
170 -                     ALU_Control = 5'b000101;
171 -                     Branch = {1'b0, zero};
172 -                 end
173 -                 3'b101: begin // BGE
174 -                     ALU_Control = 5'b001010;
175 -                     Branch = {1'b0, zero};
176 -                 end
177 -                 3'b110: begin // BLTU
178 -                     ALU_Control = 5'b00110;
179 -                     Branch = {1'b0, zero};
180 -                 end
181 -                 3'b111: begin // BGEU
182 -                     ALU_Control = 5'b01011;
183 -                     Branch = {1'b0, zero};
184 -                 end
185 -             endcase
186 -         end
187 -         7'b11010111: begin      // jal
188 -             Branch = 2'b10;
189 -             DataToReg = 2'b11;
190 -             RegWrite = 1;
191 -         end
192 -         7'b11010111: begin // jalr
193 -             Branch = 2'b11;
194 -             DataToReg = 2'b11;
195 -             RegWrite = 1;
196 -         end
197 -         7'b01101011: begin // lui
198 -             DataToReg = 2'b10;
199 -             RegWrite = 1;
200 -         end
201 -         7'b00101011: begin // AUIPC
202 -             DataToReg = 2'b10;
203 -             RegWrite = 1;
204 -         end
205 -             default: ALU_Control = 5'b11111;
206 -         endcase
207 -     end
208 - endmodule
209 -

```

```

121 +         7'b00000011: begin      // 1
122 +             ALU_Control = 5'b000010;
123 +             ALUSrc_B = 2'b01;
124 +             DataToReg = 2'b01;
125 +             RegWrite = 1;
126 +             mem_W = 1;
127 +             case (Fun1)
128 +                 3'b000: begin // LB
129 +                     B_M_W = 2'b01; // byte
130 +                 end
131 +                 3'b001: begin // LH
132 +                     B_M_W = 2'b10; // half word
133 +                 end
134 +                 3'b100: begin // LBU
135 +                     B_M_W = 2'b01; // byte
136 +                     sign = 1'b0;
137 +                 end
138 +                 3'b101: begin // LHU
139 +                     B_M_W = 2'b10; // half word
140 +                     sign = 1'b0;
141 +                 end
142 +                 // 3'b010; // LW
143 +             endcase
144 +         end
145 +         7'b01000011: begin // S
146 +             ALU_Control = 5'b000010;
147 +             ALUSrc_B = 2'b01;
148 +             mem_W = 1;
149 +             case (Fun1)
150 +                 3'b000: begin
151 +                     B_M_W = 2'b01; // byte
152 +                 end
153 +                 3'b001: begin
154 +                     B_M_W = 2'b10; // half word
155 +                 end
156 +                 // 3'b010; // SW
157 +             endcase
158 +         end
159 +         7'b11000011: begin      // Branch
160 +             case (Fun1)
161 +                 3'b000: begin // BEQ
162 +                     ALU_Control = 5'b000011;
163 +                     Branch = {1'b0, zero};
164 +                 end
165 +                 3'b001: begin // DNE
166 +                     ALU_Control = 5'b000011;
167 +                     Branch = {1'b0, ~zero};
168 +                 end
169 +                 3'b100: begin // BLT
170 +                     ALU_Control = 5'b000101;
171 +                     Branch = {1'b0, zero};
172 +                 end
173 +                 3'b101: begin // BLTU
174 +                     ALU_Control = 5'b001010;
175 +                     Branch = {1'b0, zero};
176 +                 end
177 +                 3'b110: begin // BGEU
178 +                     ALU_Control = 5'b010111;
179 +                     Branch = {1'b0, zero};
180 +                 end
181 +                 3'b111: begin // jal
182 +                     DataToReg = 2'b11;
183 +                     RegWrite = 1;
184 +                 end
185 +             endcase
186 +         end
187 +         7'b11010111: begin      // lui
188 +             DataToReg = 2'b10;
189 +             RegWrite = 1;
190 +         end
191 +         7'b01101011: begin // lui
192 +             DataToReg = 2'b10;
193 +             RegWrite = 1;
194 +         end
195 +         7'b00101011: begin // AUIPC
196 +             DataToReg = 2'b10;
197 +             RegWrite = 1;
198 +         end
199 +     end
200 + endmodule

```

```

... @@ -1,53 +1,70 @@
1 - `timescale 1ps / 1ps
2 - ///////////////////////////////////////////////////////////////////
3 - // Company:
4 - // Engineer:
5 - //
6 - // Create Date: 2021/03/12 08:46:15
7 - // Design Name:
8 - // Module Name: Data_Stall
9 - // Project Name:
10 - // Target Devices:
11 - // Tool Versions:
12 - // Description:
13 - //
14 - // Dependencies:
15 - //
16 - // Revision:
17 - // Revision 0.01 - File Created
18 - // Additional Comments:
19 - //
20 - ///////////////////////////////////////////////////////////////////
21 -
22 -
23 - module Data_Stall(
24 -     input [4:0] IF_ID_written_reg,
25 -     input [4:0] IF_ID_read_reg1,
26 -     input [4:0] IF_ID_read_reg2,
27 -     input [4:0] ID_EXE_written_reg,
28 -     input [4:0] ID_EXE_read_reg1,
29 -     input [4:0] ID_EXE_read_reg2,
30 -     input [4:0] EXE_MEM_written_reg,
31 -     input [4:0] EXE_MEM_read_reg1,
32 -     input [4:0] EXE_MEM_read_reg2,
33 -
34 -     output reg PC_dstall,
35 -     output reg IF_ID_dstall,
36 -     output reg ID_EXE_dstall
37 - );
38 - always @ (*) begin
39 -     PC_dstall = 0;
40 -     IF_ID_dstall = 0;
41 -     ID_EXE_dstall = 0;
42 -     if (ID_EXE_written_reg != 0 && (ID_EXE_written_reg == IF_ID_read_reg1 || ID_EXE_written_reg == IF_ID_read_reg2)) begin
43 -         PC_dstall = 1;
44 -         IF_ID_dstall = 1;
45 -         ID_EXE_dstall = 1;
46 -     end
47 -     else if (EXE_MEM_written_reg != 0 && (EXE_MEM_written_reg == IF_ID_read_reg1 || EXE_MEM_written_reg == IF_ID_read_reg2)) begin
48 -         PC_dstall = 1;
49 -         IF_ID_dstall = 1;
50 -         ID_EXE_dstall = 1;
51 -     end
52 -     end
53 - endmodule

```



```

1 + `timescale 1ps / 1ps
2 + ///////////////////////////////////////////////////////////////////
3 + // Company:
4 + // Engineer:
5 + //
6 + // Create Date: 2021/03/12 08:46:15
7 + // Design Name:
8 + // Module Name: Data_Stall
9 + // Project Name:
10 + // Target Devices:
11 + // Tool Versions:
12 + // Description:
13 + //
14 + // Dependencies:
15 + //
16 + // Revision:
17 + // Revision 0.01 - File Created
18 + // Additional Comments:
19 + //
20 + ///////////////////////////////////////////////////////////////////
21 +
22 +
23 + module Data_Stall(
24 +     /*
25 +      * Load-Use Hazard Detection
26 +
27 +      * Check when using instruction is decoded in ID stage
28 +
29 +      * ALU operand register numbers in ID stage are given by
30 +      *     IF/ID.RegisterRs, IF/ID.RegisterRt
31 +
32 +      * Load-use hazard when
33 +      *     ID/EX.MemRead and
34 +      *     (ID/EX.RegisterRd = IF/ID.RegisterRs) or
35 +      *     (ID/EX.RegisterRd = IF/ID.RegisterRt)
36 +
37 +      * If detected, stall and insert bubble
38 +
39 +      * REFERENCE:
40 +      * "ELECT5140"
41 +      * Advanced Computer Architecture
42 +      * Prof. Wei Zhang, ECE, HKUST
43 +
44 +      * GENERAL INTRODUCTION*
45 +
46 +      * // Input:
47 +      *     input ID_EXE_mem_r,
48 +      *     input [4:0] ID_EXE_written_reg,
49 +      *     input [4:0] IF_ID_read_reg1,
50 +      *     input [4:0] IF_ID_read_reg2,
51 +
52 +      *     // Output:
53 +      *     output reg PC_dstall,
54 +      *     output reg IF_ID_dstall,
55 +      *     output reg ID_EXE_dstall
56 +
57 + );
58 + always @ (*) begin
59 +     PC_dstall = 0;
60 +     IF_ID_dstall = 0;
61 +     ID_EXE_dstall = 0;
62 +     if (ID_EXE_mem_r
63 +         && ((ID_EXE_written_reg == IF_ID_read_reg1)
64 +             || (ID_EXE_written_reg == IF_ID_read_reg2))) begin
65 +         PC_dstall = 1;
66 +         IF_ID_dstall = 1;
67 +         ID_EXE_dstall = 1;
68 +     end
69 + end
70 + endmodule

```

```

... @@ -0,0 +1,132 @@
1 - `timescale 1ps / 1ps
2 - ///////////////////////////////////////////////////////////////////
3 - // Company:
4 - // Engineer:
5 - //
6 - // Create Date: 2023/05/01 14:34:39
7 - // Design Name:
8 - // Module Name: Forward_Unit
9 - // Project Name:
10 - // Target Devices:
11 - // Tool Versions:
12 - // Description: Forwarding unit to perform data forwarding
13 - //
14 - // Dependencies:
15 - //
16 - // Revision:
17 - // Revision 0.01 - File Created
18 - // Additional Comments:
19 - //
20 - ///////////////////////////////////////////////////////////////////
21 -
22 - module Forward_Unit(
23 -     /*
24 -      * Check whether the Destination registers written by current instruction
25 -      * match the Source registers read by next instructions:
26 -      *     ID/EX.WriteRegister (Rd) = IF/ID.ReadRegister1 (Rs)
27 -      *     ID/EX.WriteRegister (Rd) = IF/ID.ReadRegister2 (Rt)
28 -      *     EX/MEM.WriteRegister = IF/ID.ReadRegister1
29 -      *     EX/MEM.WriteRegister = IF/ID.ReadRegister2
30 -      *     MEM/WB.WriteRegister = IF/ID.ReadRegister1
31 -      *     MEM/WB.WriteRegister = IF/ID.ReadRegister2
32 -
33 -      * EX hazard
34 -      *     // Rd = Rs
35 -      *     ^ if (EX/MEM.RegrWrite and (EX/MEM.RegisterRd != 0)
36 -      *           and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
37 -      *           forward EX/MEM.RegisterRd to ID/EX.RegisterRs
38 -
39 -      *     // Rd = Rt
40 -      *     ^ if (EX/MEM.RegrWrite and (EX/MEM.RegisterRd != 0)
41 -      *           and (EX/MEM.RegisterRd == ID/EX.RegisterRt))

```

```

42 +           Forward EX/MEM.RegisterRd to ID/EX.RegisterRt
43 +
44 +           MEM hazard
45 +           // Rd = Rs
46 +           // If (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
47 +           // and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd =
48 +           // ID/EX.RegisterRt))
49 +           // and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
50 +           Forward MEM/WB.RegisterRd to ID/EX.RegisterRs
51 +
52 +           // If (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
53 +           // and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd =
54 +           // ID/EX.RegisterRt))
55 +           // and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
56 +
57 +           Note: The "/* not EX hazard" line is added to prevent the MEM hazard from
58 +           Forwarding an outdated value from the MEM/WB pipeline register when there
59 +           is an EX hazard. This is because the EX stage is executed before the MEM,
60 +           and a newer instruction may have written to the register in the EX stage.
61 +
62 +           REFERENCE:
63 +           "ELECS140"
64 +           Advanced Computer Architecture
65 +           Prof. Wei Zhang, ECE, HKUST
66 +           GENERAL INTRODUCTION"
67 +
68 +           */
69 +
70 +           // Input:
71 +           input [4:0] ID_EXE_read_reg1, // ID/EXE.ReadRegister1
72 +           input [4:0] ID_EXE_read_reg2, // ID/EXE.ReadRegister2
73 +
74 +           input EXE_MEM_RegWrite, // EXE/MEM.RegWrite
75 +           input [4:0] EXE_MEM_written_Reg, // EXE/MEM.WriteRegister
76 +
77 +           input MEM_WB_RegWrite, // MEM/WB.RegWrite
78 +           input [4:0] MEM_WB_written_Reg, // MEM/WB.WriteRegister
79 +
80 +           // Output:
81 +           output reg [1:0] forwarding_A_sig, // forwarding signal for ALU input A (forwarding_A_sig =
82 +           EXE_forwarding_A | MEM_forwarding_A)
83 +           output reg [1:0] forwarding_B_sig // forwarding signal for ALU input B (forwarding_B_sig =
84 +           EXE_forwarding_B | MEM_forwarding_B)
85 +
86 +           always @(*) begin
87 +               // Initialize the forwarding signals
88 +               forwarding_A_sig = 2'b00;
89 +               forwarding_B_sig = 2'b00;
90 +
91 +               // EX hazard
92 +               // Rd = Rs
93 +               if ((EXE_MEM_RegWrite && EXE_MEM_written_Reg != 0)
94 +                   && (EXE_MEM_written_Reg == ID_EXE_read_Reg1)) begin
95 +                   // Note: 1'b1 means a binary value of 1 (the 1 on the right of the b)
96 +                   // with a width of 1 bit (the 1 on the left of the b)
97 +                   forwarding_A_sig[1] = 1'b1; // Signal 1 to forward
98 +               end
99 +               else begin
100 +                   forwarding_A_sig[1] = 1'b0; // Signal 0 to not forward
101 +               end
102 +
103 +               // Rd = Rt
104 +               if ((EXE_MEM_RegWrite && EXE_MEM_written_Reg != 0)
105 +                   && (EXE_MEM_written_Reg == ID_EXE_read_Reg2)) begin
106 +                   forwarding_B_sig[1] = 1'b1; // Signal 1 to forward
107 +               end
108 +               else begin
109 +                   forwarding_B_sig[1] = 1'b0; // Signal 0 to not forward
110 +
111 +               // MEM hazard (only if there is no EX hazard)
112 +               // Rd = Rs
113 +               if ((MEM_WB_RegWrite && MEM_WB_written_Reg != 0)
114 +                   && ((EXE_MEM_written_Reg != 0 && (EXE_MEM_written_Reg == ID_EXE_read_Reg1))) // not EX hazard
115 +                   && (MEM_WB_written_Reg == ID_EXE_read_Reg1)) begin
116 +                   forwarding_A_sig[0] = 1'b1; // Signal 1 to forward
117 +               end
118 +               else begin
119 +                   forwarding_A_sig[0] = 1'b0; // Signal 0 to not forward
120 +
121 +               // Rd = Rt
122 +               if ((MEM_WB_RegWrite && MEM_WB_written_Reg != 0)
123 +                   && ((EXE_MEM_written_Reg != 0 && (EXE_MEM_written_Reg == ID_EXE_read_Reg2))) // not EX hazard
124 +                   && (MEM_WB_written_Reg == ID_EXE_read_Reg2)) begin
125 +                   forwarding_B_sig[0] = 1'b1; // Signal 1 to forward
126 +               end
127 +               else begin
128 +                   forwarding_B_sig[0] = 1'b0; // Signal 0 to not forward
129 +               end
130 +
131 +           end
132 +       endmodule

```

```

v 156 ██████ RV32I/RV32I.srcc/sources_1/new/Gshare_Predictor.v [ ]
... @@ -0,0 +1,156 @@
1 + `timescale 1ns / 1ps
2 + //////////////////////////////////////////////////////////////////
3 + // Company:
4 + // Engineer:
5 + //
6 + // Create Date: 08.05.2023 19:33:16
7 + // Design Name:
8 + // Module Name: Gshare_Predictor
9 + // Project Name:
10 + // Target Devices:
11 + // Tool Versions:
12 + // Description:
13 + //
14 + // Dependencies:
15 + //
16 + // Revision:
17 + // Revision 0.01 - File Created
18 + // Additional Comments:
19 + //
20 + //////////////////////////////////////////////////////////////////
21 +
22 +
23 + module Gshare_Predictor(
24 + /*
25 + * Gshare 8/8 two-level branch predictor
26 + * 8 bits BMR
27 + * 256 entries PHT
28 + * 2 bits saturating up/down predictors per entry
29 +
30 + * PHT indexed by PC[9:2] XOR BMR[7:0]
31 + */
32 +
33 + // Input:
34 + input clk,
35 + input rst,
36 + input [31:0] PC,
37 + input [6:0] Opcode,
38 + input [2:0] Funct,
39 + input correct_prediction, // 1 if last prediction was correct, 0 if not
40 + input is_branch, // 1 if last instruction was a branch, 0 if not
41 +
42 + // Output:
43 + output reg [1:0] Branch,
44 + output reg zero_prediction, // If zero_prediction==1, then the branch is taken (except for BNE)
45 + output reg is_bne
46 + );
47 +
48 + // BMR
49 + reg [7:0] BMR = 8'b00000000;
50 + // PHT
51 + reg [1:0] PHT [0:255];
52 + integer i;
53 + initial begin // Initialize the 256 entries of PHT to 2'b10
54 + for (i = 0; i < 256; i = i + 1) begin
55 + PHT[i] = 2'b10;
56 + end
57 + end
58 +
59 + // Indexing
60 + reg [7:0] index = 8'b00000000;
61 + always @ (*) begin
62 + index = {PC[9:2]} ^ {BMR[7:0]}; // PC[9:2] XOR BMR[7:0]
63 +
64 + // TEST: index by GSELECT (concatenate PC[5:2] and BMR[3:0]);
65 + // index = {PC[5:2], BMR[3:0]};
66 + end
67 +
68 + // Update BMR (on rising edge of clk)
69 + always @ (posedge clk or posedge rst) begin
70 + if (rst == 1) begin
71 + BMR = 8'b00000000; // Reset back to 0
72 + end
73 + else begin
74 + // If last instruction was a branch, shift BMR left by 1 and add correct prediction to LSB
75 + if (is_branch == 1) begin
76 + BMR = {BMR[6:0], correct_prediction};
77 + end
78 + end
79 + end
80 +
81 + // Make new prediction (on falling edge of clk)
82 + always @ (nedge clk) begin
83 + // Initialize Branch to 00
84 + Branch = 0;
85 + // Put MSB of PHT[index] into zero_prediction
86 + zero_prediction = PHT[index][1];

```

```

87 +
88 +     is_bne = 0;
89 +
90 +     // Calculate Branch (Previously done in Control Unit)
91 +     case (Opcode)
92 +         // Branch instructions
93 +         7'b1100001: begin // Branch
94 +             case (Fun)
95 +                 3'b000: begin // BEQ
96 +                     Branch = {1'b0, zero_prediction};
97 +                 end
98 +                 3'b001: begin // BNE
99 +                     Branch = {1'b0, zero_prediction};
100 +                    // Indicate that this is a BNE instruction
101 +                    // So that the zero_correct in the ALU is flipped
102 +                    is_bne = 1;
103 +                 end
104 +                 3'b100: begin // BLT
105 +                     Branch = {1'b0, zero_prediction};
106 +                 end
107 +                 3'b101: begin // BGE
108 +                     Branch = {1'b0, zero_prediction};
109 +                 end
110 +                 3'b110: begin // BLTU
111 +                     Branch = {1'b0, zero_prediction};
112 +                 end
113 +                 3'b111: begin // BGEU
114 +                     Branch = {1'b0, zero_prediction};
115 +                 end
116 +             endcase
117 +         end
118 +
119 +         // Jump instructions
120 +         7'b1101111: begin // jal
121 +             Branch = 2'b10;
122 +         end
123 +         7'b11100111: begin // jalr
124 +             Branch = 2'b11;
125 +         end
126 +     endcase
127 + end
128 +
129 +     // Update PHT with the last prediction (on rising edge of clk)
130 +     always @ (posedge clk or posedge rst) begin
131 +         if (rst == 1) begin
132 +             // Reset back to weakly taken all entries
133 +             for (i = 0; i < 256; i = i + 1) begin
134 +                 PHT[i] = 2'b10;
135 +             end
136 +         end
137 +         else begin
138 +             // If last instruction was a branch, update PHT
139 +             if (is_branch == 1) begin
140 +                 // If last prediction was correct, increment PHT (up to 11)
141 +                 if (correct_prediction == 1) begin
142 +                     if (PHT[index] < 3) begin
143 +                         PHT[index] = PHT[index] + 1;
144 +                     end
145 +                 end
146 +                 // Else, decrement PHT (down to 00)
147 +                 else begin
148 +                     if (PHT[index] > 0) begin
149 +                         PHT[index] = PHT[index] - 1;
150 +                     end
151 +                 end
152 +             end
153 +         end
154 +     end
155 + endmodule
156 +

```

```

... 81  RV32I/RV32I.srcc/sources_1/new/REG32.v ...
1  ... @@ -1,40 +1,41 @@
2  - ///////////////////////////////////////////////////////////////////
3  - // Company:
4  - // Engineer:
5  - //
6  - // Create Date: 2021/03/10 19:39:55
7  - // Design Name:
8  - // Module Name: REG32
9  - // Project Name:
10 - // Target Devices:
11 - // Tool Versions:
12 - // Description:
13 - //
14 - // Dependencies:
15 - //
16 - // Revision:
17 - // Revision 0.01 - File Created
18 - // Additional Comments:
19 - //
20 - ///////////////////////////////////////////////////////////////////
21 -
22 -
23 - module REG32(
24 -     input clk,
25 -     input rst,
26 -     input CE,
27 -     input [31:0] D,
28 -     output reg [31:0] Q = 0,
29 -     input PC_dstall
30 - );
31 -
32 -     always @ (posedge clk or posedge rst) begin
33 -         if (rst == 1) Q <= 32'h00000000;
34 -         if (PC_dstall == 0) begin
35 -             if (rst == 1) Q <= 32'h00000000;
36 -             else if (CE) Q <= D;
37 -         end
38 -     end
39 - endmodule
40 -
41 + module REG32(
42 -     input clk,
43 -     input rst,
44 -     input CE,
45 -     input [31:0] D,
46 -     output reg [31:0] Q = 0,
47 -     input PC_dstall,
48 -     input PC_cstall
49 - );
50 -
51 -     always @ (posedge clk or posedge rst) begin
52 -         if (rst == 1) Q <= 32'h00000000;
53 -         if (PC_dstall == 0 && PC_cstall == 0) begin // Added PC_cstall
54 -             if (rst == 1) Q <= 32'h00000000;
55 -             else if (CE) Q <= D;
56 -         end
57 -     end
58 - endmodule
59 -
60 +

```

```

... @@ 1,90 -1,126 @@
1 . `timescale 1ps / 1ps
2 .
3 . // Company:
4 . // Engineer:
5 . //
6 . // Create Date: 2021/03/11 22:00:28
7 . // Design Name:
8 . // Module Name: REG_ID_EXE
9 . // Project Name:
10 . // Target Devices:
11 . // Tool Versions:
12 . // Description:
13 . //
14 . // Dependencies:
15 . //
16 . // Revision:
17 . // Revision 0.01 - File Created
18 . // Additional Comments:
19 . //
20 . ///////////////////////////////////////////////////////////////////////////////
21 .
22 .
23 . module REG_ID_EXE(
24 .     input clk,
25 .     input rst,
26 .     input CE,
27 .     input ID_EXE_dstall,
28 .
29 .     input [31:0] inst_in,
30 .     input [31:0] PC,
31 .     input [31:0] ALU_A,
32 .     input [31:0] ALU_B,
33 .     input [4:8] ALU_Control,
34 .     input [31:0] Data_out,
35 .     input mem_w,
36 .     input [1:8] DatatoReg,
37 .     input Regwrite,
38 .
39 .     input [4:8] written_Reg,
40 .     input [4:8] read_Regl,
41 .     input [4:8] read_Reg2,
42 .
43 .     output reg [31:0] ID_EXE_inst_in,
44 .     output reg [31:0] ID_EXE_PC = 0,
45 .     output reg [31:0] ID_EXE_ALU_A,
46 .     output reg [31:0] ID_EXE_ALU_B,
47 .     output reg [4:8] ID_EXE_ALU_Control,
48 .     output reg [31:0] ID_EXE_Data_out,
49 .     output reg ID_EXE_mem_w,
50 .     output reg [1:8] ID_EXE_DatatoReg,
51 .     output reg ID_EXE_Regwrite,
52 .
53 .     output reg [4:8] ID_EXE_written_Reg,
54 .     output reg [4:8] ID_EXE_read_Regl,
55 .     output reg [4:8] ID_EXE_read_Reg2
56 . );
57 .
58 . always @ (posedge clk or posedge rst) begin
59 .     if (rst == 1 || ID_EXE_dstall == 1) begin
60 .         ID_EXE_inst_in <= 32'h00000001;
61 .         ID_EXE_PC <= 32'h00000000;
62 .         ID_EXE_ALU_A <= 32'h00000000;
63 .         ID_EXE_ALU_B <= 32'h00000000;
64 .         ID_EXE_ALU_Control <= 5'h00000;
65 .         ID_EXE_Data_out <= 32'h00000000;
66 .         ID_EXE_mem_w <= 1'b0;
67 .         ID_EXE_DatatoReg <= 2'b00;
68 .         ID_EXE_Regwrite <= 1'b0;
69 .
70 .         ID_EXE_written_Reg <= 5'b00000;
71 .         ID_EXE_read_Regl <= 5'b00000;
72 .         ID_EXE_read_Reg2 <= 5'b00000;
73 .     end
74 .     else if (CE) begin
75 .         ID_EXE_inst_in <= inst_in;
76 .         ID_EXE_PC <= PC;
77 .         ID_EXE_ALU_A <= ALU_A;
78 .         ID_EXE_ALU_B <= ALU_B;
79 .         ID_EXE_ALU_Control <= ALU_Control;
80 .         ID_EXE_Data_out <= Data_out;
81 .         ID_EXE_mem_w <= mem_w;
82 .         ID_EXE_DatatoReg <= DatatoReg;
83 .         ID_EXE_Regwrite <= Regwrite;
84 .
85 .         ID_EXE_written_Reg <= written_Reg;
86 .         ID_EXE_read_Regl <= read_Regl;
87 .
88 .         ID_EXE_im32 <= 32'h00000000;
89 .         ID_EXE_ALU_src_A <= 1'b0;
90 .         ID_EXE_ALU_src_B <= 1'b0;
91 .         ID_EXE_ALU_src_C <= 1'b0;
92 .         ID_EXE_ALU_src_D <= 1'b0;
93 .         ID_EXE_ALU_src_E <= 1'b0;
94 .         ID_EXE_ALU_src_F <= 1'b0;
95 .         ID_EXE_ALU_src_G <= 1'b0;
96 .         ID_EXE_ALU_src_H <= 1'b0;
97 .         ID_EXE_ALU_src_I <= 1'b0;
98 .         ID_EXE_ALU_src_J <= 1'b0;
99 .         ID_EXE_ALU_src_K <= 1'b0;
100 .        ID_EXE_ALU_src_L <= 1'b0;
101 .        ID_EXE_ALU_src_M <= 1'b0;
102 .        ID_EXE_ALU_src_N <= 1'b0;
103 .        ID_EXE_ALU_src_O <= 1'b0;
104 .        ID_EXE_ALU_src_P <= 1'b0;
105 .        ID_EXE_ALU_src_Q <= 1'b0;
106 .        ID_EXE_ALU_src_R <= 1'b0;
107 .        ID_EXE_ALU_src_S <= 1'b0;
108 .        ID_EXE_ALU_src_T <= 1'b0;
109 .        ID_EXE_ALU_src_U <= 1'b0;
110 .        ID_EXE_ALU_src_V <= 1'b0;
111 .        ID_EXE_ALU_src_W <= 1'b0;
112 .        ID_EXE_ALU_src_X <= 1'b0;
113 .        ID_EXE_ALU_src_Y <= 1'b0;
114 .        ID_EXE_ALU_src_Z <= 1'b0;
115 .    end
116 . end
117 .
118 . endmodule

```

```

87 -     ID_EXE_read_reg2  <= read_reg2;
88 -   end
89 - end
90 - endmodule
91 +
92 +
93 +
94 +
95 +
96 +
97 +
98 +
99 +
100 +
101 +
102 +
103 +
104 +
105 +
106 +
107 +
108 +
109 +
110 +
111 +
112 +
113 +
114 +
115 +
116 +
117 +
118 +
119 +
120 +
121 +
122 +
123 +
124 +
125 +
126 + endmodule

```

```

... @@ -1,54 +1,75 @@
1 - `timescale 1ps / 1ps
2 - ///////////////////////////////////////////////////////////////////
3 - // Company:
4 - // Engineer:
5 - //
6 - // Create Date: 2021/03/10 21:05:36
7 - // Design Name:
8 - // Module Name: REG_IF_ID
9 - // Project Name:
10 - // Target Devices:
11 - // Tool Versions:
12 - // Description:
13 - //
14 - // Dependencies:
15 - //
16 - // Revision:
17 - // Revision 0.01 - File Created
18 - // Additional Comments:
19 - //
20 - ///////////////////////////////////////////////////////////////////
21 -
22 -
23 - module REG_IF_ID(
24 -   input clk,
25 -   input rst,
26 -   input CE,
27 -   input IF_ID_dstall,
28 -   input IF_ID_cstall,
29 -
30 -   input [31:0] inst_in,
31 -   input [31:0] PC,
32 -
33 -   output reg [31:0] If_ID_inst_in,
34 -   output reg [31:0] If_ID_PC = 0
35 - );
36 - always @ (posedge clk or posedge rst) begin
37 -   if (rst == 1) begin
38 -     If_ID_inst_in <= 32'h000000013;
39 -     If_ID_PC <= 32'h00000000;
40 -   end
41 -   // A bubble here is a nop, or rather, "addi x0, x0, 0"
42 -   if (IF_ID_dstall == 0) begin
43 -     if (rst == 1 || IF_ID_cstall == 1'b1) begin
44 -       If_ID_inst_in <= 32'h000000013;
45 -       If_ID_PC <= 32'h00000000;
46 -     end
47 -     else if (CE) begin
48 -       If_ID_inst_in <= inst_in;
49 -       If_ID_PC <= PC;
50 -     end
51 -   end
52 -   // else: if stall, then nothing changes here
53 - end
54 - endmodule
1 + `timescale 1ps / 1ps
2 + ///////////////////////////////////////////////////////////////////
3 + // Company:
4 + // Engineer:
5 + //
6 + // Create Date: 2021/03/10 21:05:36
7 + // Design Name: REG_IF_ID
8 + // Module Name: REG_IF_ID
9 + // Project Name:
10 + // Target Devices:
11 + // Tool Versions:
12 + // Description:
13 + //
14 + // Dependencies:
15 + //
16 + // Revision:
17 + // Revision 0.01 - File Created
18 + // Additional Comments:
19 + //
20 + ///////////////////////////////////////////////////////////////////
21 +
22 +
23 + module REG_IF_ID(
24 +   input clk,
25 +   input rst,
26 +   input CE,
27 +   input IF_ID_dstall,
28 +   input IF_ID_cstall,
29 +
30 +   input [31:0] inst_in,
31 +   input [31:0] PC,
32 +   input [31:0] Imm_32,
33 +   input [31:0] add_branch_out,
34 +   input zero_prediction,
35 +   input is_bne,
36 +   input is_first_jalr, // 1 if it is the first jalr out of the 2 in the bubble, 0 otherwise
37 +
38 +   output reg [31:0] If_ID_inst_in,
39 +   output reg [31:0] If_ID_PC = 0,
40 +   output reg [31:0] If_ID_Imm_32,
41 +   output reg [31:0] If_ID_addr_branch_out,
42 +   output reg If_ID_zero_prediction,
43 +   output reg If_ID_is_bne,
44 +   output reg If_ID_is_first_jalr
45 +
46 + );
47 - always @ (posedge clk or posedge rst) begin
48 -   if (rst == 1) begin
49 -     If_ID_inst_in <= 32'h000000013;
50 -     If_ID_PC <= 32'h00000000;
51 -   end
52 -   // A bubble here is a nop, or rather, "addi x0, x0, 0"
53 -   if (IF_ID_dstall == 0) begin
54 -     if (rst == 1 || IF_ID_cstall == 1'b1) begin
55 -       If_ID_inst_in <= 32'h000000013;
56 -       If_ID_PC <= 32'h00000000;
57 -       If_ID_Imm_32 <= 32'h00000000;
58 -       If_ID_addr_branch_out <= 32'h00000000;
59 -       If_ID_zero_prediction <= 1'b0;
60 -       If_ID_is_bne <= 1'b0;
61 -       If_ID_is_first_jalr <= 1'b1;
62 -     end
63 -     else if (CE) begin
64 -       If_ID_inst_in <= inst_in;
65 -       If_ID_PC <= PC;
66 -       If_ID_Imm_32 <= Imm_32;
67 -       If_ID_addr_branch_out <= add_branch_out;
68 -       If_ID_zero_prediction <= zero_prediction;
69 -       If_ID_is_bne <= is_bne;
70 -       If_ID_is_first_jalr <= is_first_jalr;
71 -     end
72 -   end
73 -   // else: if stall, then nothing changes here
74 - end
75 + endmodule

```

```

... @@ -1,60 +1,65 @@
1 - `timescale 1ps / 1ps
2 - ///////////////////////////////////////////////////////////////////
3 - // Company:
4 - // Engineer:
5 - //
6 - // Create Date: 2021/03/12 08:12:16
7 - // Design Name:
8 - // Module Name: REG_MEM_WB
9 - // Project Name:
10 - // Target Devices:
11 - // Tool Versions:
12 - // Description:
13 - //
14 - // Dependencies:
15 - //
16 - // Revision:
17 - // Revision 0.01 - File Created
18 - // Additional Comments:
19 - //
20 - ///////////////////////////////////////////////////////////////////
21 -
22 -
23 - module REG_MEM_WB(
24 -     input clk,
25 -     input rst,
26 -     input CE,
27 -     // Input
28 -     input [31:0] inst_in,
29 -     input [31:0] PC,
30 -     input [31:0] ALU_out,
31 -     input [1:0] DataToReg,
32 -     input Regwrite,
33 -     input [31:0] Data_in,
34 -     // Output
35 -     output reg [31:0] MEM_WB_inst_in,
36 -     output reg [31:0] MEM_WB_PC = 0,
37 -     output reg [31:0] MEM_WB_ALU_out,
38 -     output reg [1:0] MEM_WB_DataToReg,
39 -     output reg MEM_WB_Regwrite,
40 -     output reg [31:0] MEM_WB_Data_in
41 - );
42 -     always @ (posedge clk or posedge rst) begin
43 -         if (rst == 1) begin
44 -             MEM_WB_inst_in <= 32'h00000013;
45 -             MEM_WB_PC <= 32'h00000000;
46 -             MEM_WB_ALU_out <= 32'h00000000;
47 -             MEM_WB_DataToReg <= 2'b00;
48 -             MEM_WB_Regwrite <= 1'b0;
49 -             MEM_WB_Data_in <= 32'h00000000;
50 -         end
51 -         else if (CE) begin
52 -             MEM_WB_inst_in <= inst_in;
53 -             MEM_WB_PC <= PC;
54 -             MEM_WB_ALU_out <= ALU_out;
55 -             MEM_WB_DataToReg <= DataToReg;
56 -             MEM_WB_Regwrite <= Regwrite;
57 -             MEM_WB_Data_in <= Data_in;
58 -         end
59 -     end
60 - endmodule

```

```

... @@ -1,450 +1,611 @@
1 - `timescale 1ps / 1ps
2 - ///////////////////////////////////////////////////////////////////
3 - // Company:
4 - // Engineer:
5 - //
6 - // Create Date: 2021/03/10 19:39:55
7 - // Design Name:
8 - // Module Name: RV32IPCPU
9 - // Project Name:
10 - // Target Devices:
11 - // Tool Versions:
12 - // Description:
13 - //
14 - // Dependencies:
15 - //
16 - // Revision:
17 - // Revision 0.01 - File Created
18 - // Additional Comments:
19 - //
20 - ///////////////////////////////////////////////////////////////////
21 -
22 -
23 - module RV32IPCPU(
24 -     input clk,
25 -     input rst,
26 -     input [31:0] data_in, // MEM
27 -     input [31:0] inst_in, // IF, from PC_out
28 -
29 -     output [31:0] ALU_out, // From MEM, address out, for fetching data_in
30 -     output [31:0] data_out, // From MEM, to be written into data memory
31 -     output mem_w, // From MEM, write valid, for store instructions
32 -     output [31:0] PC_out // From IF
33 - );
34 -     wire V5;
35 -     wire N0;
36 -     wire [31:0] Imm_32;
37 -     wire [31:0] add_branch_out;
38 -     wire [31:0] add_jal_out;
39 -     wire [31:0] add_jalr_out;
40 -
41 -     wire [4:0] wt_addr;

```

```

42 -     wire [31:0] Wt_data;
43 -     wire [31:0] rdata_A;
44 -     wire [31:0] rdata_B;
45 -     wire [31:0] PC_wb;
46 -     wire [31:0] ALU_A;
47 -     wire [31:0] ALU_B;
48 -     assign VS = 1'b1;
49 -     assign NO = 1'b0;
50 -
51 -
52 -     wire zero; // ID
53 -     wire [1:0] Branch; // ID
54 -     wire ALUSrc_A; // EXE
55 -     wire [1:0] ALUSrc_B; // EXE
56 -     wire [4:0] ALU_Control; // EXE
57 -     wire RegWrite; // WB
58 -     wire [1:0] DataReg; // WB
59 -
60 -     wire [1:0] B_M_H; // WB // not used yet
61 -     wire sign; // WB // not used yet
62 - //     wire Regst; // WB
63 - //     wire Jali; // WB
64 -
65 - // IF_ID
66 -     wire [31:0] IF_ID_inst_in;
67 -     wire [31:0] IF_ID_PC;
68 -     wire [31:0] IF_ID_Data_out;
69 -     wire IF_ID_mem_w;
70 -     wire [4:0] IF_ID_written_reg;
71 -     wire [4:0] IF_ID_read_reg1;
72 -     wire [4:0] IF_ID_read_reg2;
73 -
74 - // ID_EXE
75 -     wire [31:0] ID_EXE_inst_in;
76 -     wire [31:0] ID_EXE_PC;
77 -     wire [31:0] ID_EXE_ALU_A;
78 -     wire [31:0] ID_EXE_ALU_B;
79 -     wire [4:0] ID_EXE_ALU_Control;
80 -     wire [31:0] ID_EXE_Data_out;
81 -     wire ID_EXE_mem_w;
82 -     wire [1:0] ID_EXE_DataReg;
83 -     wire ID_EXE_Regrite;
84 -     wire [4:0] ID_EXE_written_reg;
85 -     wire [4:0] ID_EXE_read_reg1;
86 -     wire [4:0] ID_EXE_read_reg2;
87 -
88 -     wire [31:0] ID_EXE_ALU_out;
89 -
90 - // EXE_MEM
91 -     wire [31:0] EXE_MEM_inst_in;
92 -     wire [31:0] EXE_MEM_PC;
93 -     wire [31:0] EXE_MEM_ALU_out;
94 -     wire [31:0] EXE_MEM_Data_out;
95 -     wire EXE_MEM_mem_w;
96 -     wire [1:0] EXE_MEM_DataReg;
97 -     wire EXE_MEM_Regrite;
98 -     wire [4:0] EXE_MEM_written_reg;
99 -     wire [4:0] EXE_MEM_read_reg1;
100 -    wire [4:0] EXE_MEM_read_reg2;
101 -
102 - // MEM_WB
103 -    wire [31:0] MEM_WB_inst_in;
104 -    wire [31:0] MEM_WB_PC;
105 -    wire [31:0] MEM_WB_ALU_out;
106 -    wire [31:0] MEM_WB_Data_in;
107 -    wire [1:0] MEM_WB_DataReg;
108 -    wire MEM_WB_Regrite;
109 -
110 - // Stall
111 -    wire PC_dstall;
112 -    wire IF_ID_cstall;
113 -    wire IF_ID_dstall;
114 -    wire ID_EXE_dstall;
115 -
116 -
117 -
118 -    Data_Stall_dstall_ (
119 -        .IF_ID_written_reg(IF_ID_written_reg),
120 -        .IF_ID_read_reg1(IF_ID_read_reg1),
121 -        .IF_ID_read_reg2(IF_ID_read_reg2),
122 -
123 -        .ID_EXE_written_reg(ID_EXE_written_reg),
124 -        .ID_EXE_read_reg1(ID_EXE_read_reg1),
125 -        .ID_EXE_read_reg2(ID_EXE_read_reg2),
126 -
127 -        .EXE_MEM_written_reg(EXE_MEM_written_reg),
128 -        .EXE_MEM_read_reg1(EXE_MEM_read_reg1),
129 -        .EXE_MEM_read_reg2(EXE_MEM_read_reg2),
130 -
131 -        .PC_dstall(PC_dstall),
132 -        .IF_ID_dstall(IF_ID_dstall),
133 -        .ID_EXE_dstall(ID_EXE_dstall)
134 -    );
135 -
136 -    Control_Stall_cstall_ (
137 -        .Branch(Branch[1:0]),
138 -        .IF_ID_cstall(IF_ID_cstall)
139 -    );
140 -
141 -    assign ALU_out = EXE_MEM_ALU_out;
142 -    assign Data_out = EXE_MEM_Data_out;
143 -    assign mem_w = EXE_MEM_mem_w;
144 -
145 - // IF:-
146 - // Control Signals:
147 - // 1. Branch - MUX5 : ID
148 - // References:
149 - // 1. inst_in - MUX5 : ID
150 - // 2. rdata_A - MUX5 : ID
151 - // 3. Imm_32 - ADD_branch : ID
152 - // Pass-on:
153 - // 1. inst_in (combinatorial)
154 - // 2. PC
155 - // Out:
156 - // 1. PC_out: for fetching inst_in

```

```

157 - REG32 _pc_ (
158 -   .CE(V5),
159 -   .clk(clk),
160 -   .D(PC_wb[31:0]),
161 -   .rst(rst),
162 -   .Q(PC_out[31:0]),
163 -   .PC_dstall(PC_dstall)
164 - );
165 - add_32 ADD_Branch {
166 -   .(IF_ID_PC[31:0]), // use the "PC" from ID stage
167 -   .(Imm_32[31:0]), // From ID stage
168 -   .(add_branch_out[31:0]) // actually this part belongs to IF_ID
169 - };
170 - add_32 ADD_JAL {
171 -   .(IF_ID_PC), // MIPS: PC+4, RISC-V: PC!!!
172 -   .((11(IF_ID_inst_in[31])), IF_ID_inst_in[31], IF_ID_inst_in[19:12], IF_ID_inst_in[20],
173 -     IF_ID_inst_in[30:21], 1'b0),
174 -   .(add_jal_out[31:0])
175 - };
176 - add_32 ADD_JALR {
177 -   .(data_A[31:0]),
178 -   .((20(IF_ID_inst_in[31])), IF_ID_inst_in[31:20]),
179 -   .(add_jalr_out[31:0])
179 - };
180 - Mux4to1b32 MUX {
181 -   .IO(PC_out[31:0] + 32'b0100), // From IF stage
182 -   .I1(add_branch_out[31:0]), // Containing "PC" from ID stage
183 -   .I2(add_jal_out[31:0]), // From ID stage
184 -   .I3(add_jalr_out[31:0]), // From ID stage
185 -   .S(branch[1:0]), // From ID
186 -   .O(PC_wb[31:0])
187 - };
188 -
189 -
190 - REG_IF_ID_if_id_ (
191 -   .clk(clk), .rst(rst), .CE(V5),
192 -   .IF_ID_dstall(IF_ID_dstall), .IF_ID_cstall(IF_ID_cstall),
193 -   // Input
194 -   .inst_in(inst_in),
195 -   .PC(PC_out),
196 -   // Output
197 -   .IF_ID_inst_in(IF_ID_inst_in),
198 -   .IF_ID_PC(IF_ID_PC)
199 - );
200 -
201 - // ID:-----
202 - // From IF:
203 - // 1. inst_in
204 - // 2. PC
205 - // Control Signals:
206 - // 1. Regrite : Regs : WB
207 - // 2. ALUSrc_A / ALUSrc_B (stops here)
208 - // References:
209 - // None
210 - // Pass-on:
211 - // 1. inst_in
212 - // Control_signals {
213 - // 2. ALU_Control
214 - // 3. DataReg
215 - // 4. mem_w
216 - // 5. RegWrite
217 - // )
218 - // 6. ALU_A
219 - // 7. ALU_B
220 - // 8. Data_out
221 - // 9. PC
222 - // Out:
223 - // None
224 -
225 - Get_rw_regs _rw_regs_ (
226 -   .inst_n(IF_ID_inst_in[31:0]),
227 -   .written_rg(IF_ID_written_rg),
228 -   .read_rg(IF_ID_read_rg),
229 -   .read_rg2(IF_ID_read_rg2)
230 - );
231 - Controller Ctrl_Unit (
232 -   // Input:
233 -   .OPCODE(IF_ID_inst_in[6:0]),
234 -   .Fun(IF_ID_inst_in[14:12]),
235 -   .Fun(IF_ID_inst_in[31:25]),
236 -   .zero(cero),
237 -   // Output:
238 -   .ALUSrc_A(ALUSrc_A),
239 -   .ALUSrc_B(ALUSrc_B[1:0]),
240 -   .AU_Control(AU_Control[4:0]),
241 -   .Branch(Branch[1:0]),
242 -   .DataReg(DataReg[1:0]),
243 -   .mem_w(IF_ID_mem_w),
244 -   .Regwrite(Regwrite),
245 -   .B_W(B_W),
246 -   .sign(sign) // not used yet
247 - );
248 -
249 - Regs U2 (.clk(clk),
250 -   .rst(rst),
251 -   .LSMEM_WB_Regwrite(), // From Write-Back stage
252 -   .R_addr_A(IF_ID_inst_in[19:15]), // ID
253 -   .R_addr_B(IF_ID_inst_in[24:20]), // ID
254 -   .Wt_addr(Wt_addr[4:0]), // From Write-Back stage
255 -   .Wt_data(Wt_data[31:0]), // From Write-Back stage
256 -   .data_A(data_A[31:0]),
257 -   .data_B(data_B[31:0])
258 - );
259 - SignExt_signed_ext_ (.inst_in(IF_ID_inst_in), .imm_32(Imm_32));
260 -
261 - Mux2to1b32 _alu_source_A_ (
262 -   .IO(IF_ID_Data_A[31:0]),
263 -   .I1(Imm_32[31:0]), // not used
264 -   .S(ALUSrc_A),
265 -   .O(ALU_A[31:0])
266 - );
267 -
268 - Mux4to1b32 _alu_source_B_ (
269 -   .IO(IF_ID_Data_B[31:0]),
270 -   .I1(Imm_32[31:0]),
271 -   .I2(),
272 -   .I3(),
273 -   .S(ALUSrc_B[1:0]),
274 -   .O(ALU_B[31:0])
275 - );
276 - assign IF_ID_Data_out = rdata_B;
277 - ID_Zero_Generator _id_zero_(.A(ALU_A), .B(ALU_B), .ALU_operation(ALU_Control), .zero(zero));
278 -
278 -
278 +
278 +   .PC_dstall(PC_dstall),
278 +   .IF_ID_dstall(IF_ID_dstall),
278 +   .ID_EXE_dstall(ID_EXE_dstall)
278 +
278 + );
278 +
278 + Control_Stall _cstall_ (
278 +   // Input:
278 +   .Branch(Branch[1:0]),
278 +   .IF_ID_is_first_jalr(IF_ID_is_first_jalr),
278 +   .correct_prediction(correct_prediction),
278 +
278 +   // Output:
278 +   .IF_cstall(IF_cstall),
278 +   .IF_ID_cstall(IF_ID_cstall),
278 +   .ID_EXE_cstall(ID_EXE_cstall),
278 +   .PC_cstall(PC_cstall),
278 +   .is_first_jalr(is_first_jalr)
278 +
278 + );
278 +
278 + assign ALU_out = EXE_MEM_ALU_out;
278 + assign data_out = EXE_MEM_Data_out;
278 + assign mem_w = EXE_MEM_mem_w;
278 +
278 + // IF:
278 + // Control Signals:
278 + // 1. Branch - MUXS : ID
278 + // References:
278 + // 1. inst_in - MUXS : ID
278 + // 2. rdata_A - MUXS : ID
278 + // 3. Imm_32 - ADD_Branch : ID
278 + // Pass-on:
278 + // 1. inst_in (combinatorial)
278 + // 2. PC
278 + // Out:
278 + // 1. PC_out: for fetching inst_in
278 +
278 + SignExt_signed_ext_ (.inst_in(inst_in), .imm_32(Imm_32)); // Moved from ID stage to IF stage
278 +
278 + REG32 _pc_ (
278 +   .CE(V5),
278 +   .clk(clk),
278 +   .D(PC_wb[31:0]),
278 +   .rst(rst),
278 +   .O(PC_out[31:0]),
278 +   .PC_dstall(PC_dstall),
278 +   .PC_cstall(PC_cstall)
278 +
278 +
278 + // Branch Predictor
278 + Gshare_Predictor _branch_predictor_ (
278 +   // Input:
278 +   .clk(clk),
278 +   .rst(rst),
278 +   .O(PC_out),
278 +   .PCcode(inst_in[6:0]),
278 +   .Fun(inst_in[14:12]),
278 +   .correct_prediction(correct_prediction),
278 +   .is_branch(is_branch),
278 +
278 +   // Output:
278 +   .Branch(Branch[1:0]),
278 +   .zero_prediction(zero_prediction),
278 +   .is_bne(is_bne)
278 +
278 +
278 + add_32 ADD_Branch {
278 +   .(PC_out[31:0]), // use the "PC" from IF stage
278 +   .b(Imm_32[31:0]), // From IF stage
278 +   .(add_branch_out[31:0])
278 +
278 + };
278 +
278 + add_32 ADD_JAL {
278 +   .(PC_out[31:0]), // MIPS: PC+4, RISC-V: PC!!!
278 +   .((11(IF_ID_inst_in[31])), IF_ID_inst_in[31], IF_ID_inst_in[19:12], IF_ID_inst_in[20],
278 +     IF_ID_inst_in[30:21], 1'b0),
278 +   .(add_jal_out[31:0])
278 +
278 + };
278 +
278 + add_32 ADD_JALR {
278 +   .(rdata_A[31:0]), // We cannot get the most updated value of rdata_A from IF stage, so we need
278 +   to stall for 1 cycle to get it at ID stage
278 +   .((20(IF_ID_inst_in[31])), IF_ID_inst_in[31:20]), // From ID stage
278 +   .(add_jalr_out_prov[31:0]) // Provisional value
278 + };
278 +
278 +
278 + // If IF_cstall == 1, we fetch the same instruction again
278 + Mux2to1b32 MUX_JALR_SELECT (
278 +   .IO(add_jalr_out_prov[31:0]),
278 +   .I1(IF_cstall),
278 +   .I2(IF_cstall),
278 +   .O(add_jalr_out[31:0])
278 +
278 +
278 + Mux4to1b32 MUXS (
278 +   .IO(PC_out[31:0] + 32'b0100), // From IF stage (PC+4) 00
278 +   .I1(add_branch_out[31:0]), // From IF stage 01
278 +   .I2(add_jal_out[31:0]), // From IF stage 10
278 +   .I3(add_jalr_out[31:0]), // From ID stage 11
278 +   .S(Branch[1:0]), // From IF stage
278 +
278 +   .O(unrecovered_PC[31:0])
278 +
278 +
278 + Mux2to1b32 Recover_PC (
278 +   .IO(ID_EXE_PC_after_flush[31:0]),
278 +   .I1(unrecovered_PC[31:0]),
278 +   .S((correct_prediction),
278 +   .O(PC_wb[31:0])
278 +
278 +
278 + REG_IF_ID_if_id_ (
278 +   .clk(clk), .rst(rst), .CE(V5),
278 +   .IF_ID_dstall(IF_ID_dstall), .IF_ID_cstall(IF_ID_cstall),
278 +
278 +   // Input
278 +   .inst_in(inst_in),
278 +   .PC(PC_out),
278 +   .Imm_32(Imm_32),
278 +   .add_branch_out(add_branch_out),
278 +   .zero_prediction(zero_prediction),
278 +   .is_bne(is_bne),
278 +   .is_first_jalr(is_first_jalr),
278 +
278 +   // Output
278 +   .IF_ID_inst_in(IF_ID_inst_in),
278 +   .IF_ID_PC(IF_ID_PC),
278 +   .IF_ID_imm_32(IF_ID_Imm_32),
278 +   .IF_ID_add_branch_out(IF_ID_add_branch_out),
278 +   .IF_ID_zero_prediction(IF_ID_zero_prediction),
278 +
278 +   .IF_ID_is_bne(IF_ID_is_bne),
278

```

```

279 -     REQ_ID_EXE _id_exe_ (
280 -         .clk(clk), .rst(rst), .CE(VS), .ID_EXE_dstall(ID_EXE_dstall),
281 -         // Input
282 -         .inst_in(IF_ID_inst_in),
283 -         .PC(IF_ID_PC),
284 -         // To EXE stage, ALU Operands A & B
285 -         .ALU_A(ALU_A),
286 -         .ALU_B(ALU_B),
287 -         // To EXE stage, ALU operation control signal
288 -         .ALU_Control(ALU_Control),
289 -         // To MEM stage, for sw instruction, data from rs2 register written into memory
290 -         .Data_out(IF_ID_Data_out),
291 -         // To MEM stage, for sw instruction, memor write enable signal
292 -         .mem_w(IF_ID_mem_w),
293 -         // To WB stage, for choosing different data written back to register file
294 -         .DatatoReg(DatatoReg),
295 -         // To WB stage, register file write valid
296 -         .Regwrite(Regwrite),
297 -         // For Data Hazard
298 -         .written_reg(IF_ID_written_reg), .read_reg1(IF_ID_read_reg1), .read_reg2(IF_ID_read_reg2),
299 -         // Output
300 -         .ID_EXE_inst_in(ID_EXE_inst_in),
301 -         .ID_EXE_PC(ID_EXE_PC),
302 -         .ID_EXE_ALU_A(ID_EXE_ALU_A),
303 -         .ID_EXE_ALU_B(ID_EXE_ALU_B),
304 -         .ID_EXE_ALU_Control(ID_EXE_ALU_Control),
305 -         .ID_EXE_Data_out(ID_EXE_Data_out),
306 -         .ID_EXE_mem_w(ID_EXE_mem_w),
307 -         .ID_EXE_DatatoReg(ID_EXE_DatatoReg),
308 -         .ID_EXE_Regwrite(ID_EXE_Regwrite),
309 -         // For Data Hazard
310 -         .ID_EXE_written_reg(ID_EXE_written_reg), .ID_EXE_read_reg1(ID_EXE_read_reg1),
311 -         .ID_EXE_read_reg2(ID_EXE_read_reg2)
312 -     );
313 -
314 -     // EXE:
315 -     // From ID:
316 -     // 1. inst_in
317 -     // Control_signals {
318 -         // 2. ALU_Control (stops here)
319 -         // 3. mem_w
320 -         // 4. DatatoReg
321 -         // 5. Regwrite
322 -         //
323 -         // 6. ALU_A (stops here)
324 -         // 7. ALU_B (stops here)
325 -         // 8. Data_out
326 -         // 9. PC
327 -         // Control Signals:
328 -         // 1. ALU_Control
329 -         // References:
330 -         // None
331 -         // Pass-on:
332 -         // 1. inst_in
333 -         // Control_signals {
334 -             // 2. DatatoReg (WB)
335 -             // 3. mem_w (MEM)
336 -             // 4. Regwrite (NB)
337 -             //
338 -             // 5. Data_out (used at MEM together with mem_w)
339 -             // 6. ALU_out (Addr_out outside) (used at both MEM and WB)
340 -             // 7. PC
341 -             // Out:
342 -             // None
343 -
344 -             ALU _alualu_ (
345 -                 .A(IF_EXE_ALU_A[31:0]),
346 -                 .B(IF_EXE_ALU_B[31:0]),
347 -                 .ALU_operation(IF_EXE_ALU_Control[4:0]),
348 -                 .res(IF_EXE_ALU_out[31:0]),
349 -                 .overflow(),
350 -                 .zero()
351 -             );
352 -
353 -             REQ_EXE_MEM _exe_mem_ (
354 -                 .clk(clk), .rst(rst), .CE(VS),
355 -                 // Input
356 -                 .inst_in(IF_EXE_inst_in),
357 -                 .PC(IF_EXE_PC),
358 -                 // To MEM stage
359 -                 .ALU_out(IF_EXE_ALU_out),
360 -                 .Data_out(IF_EXE_Data_out),
361 -                 .mem_w(IF_EXE_mem_w),
362 -                 // To WB stage
363 -                 .DatatoReg(IF_EXE_DatatoReg),
364 -                 .Regwrite(IF_EXE_Regwrite),
365 -
366 -                 .written_reg(IF_EXE_written_reg), .read_reg1(IF_EXE_read_reg1), .read_reg2(IF_EXE_read_reg2),
367 -                 // Output
368 -                 .EXE_MEM_inst_in(EXE_MEM_inst_in),
369 -                 .EXE_MEM_PC(EXE_MEM_PC),
370 -                 .EXE_MEM_ALU_out(EXE_MEM_ALU_out),
371 -                 .EXE_MEM_Data_out(EXE_MEM_Data_out),
372 -                 .EXE_MEM_mem_w(EXE_MEM_mem_w),
373 -                 .EXE_MEM_DatatoReg(EXE_MEM_DatatoReg),
374 -                 .EXE_MEM_Regwrite(EXE_MEM_Regwrite),
375 -
376 -                 .EXE_MEM_written_reg(EXE_MEM_written_reg), .EXE_MEM_read_reg1(EXE_MEM_read_reg1),
377 -                 .EXE_MEM_read_reg2(EXE_MEM_read_reg2)
378 -             );
379 -
380 -             // MEM:
381 -             // From EXE:
382 -             // 1. inst_in
383 -             // Control_signals {
384 -                 // 2. DatatoReg (WB)
385 -                 // 3. mem_w (stops here)
386 -                 // 4. Regwrite (WB)
387 -                 //
388 -                 // 5. Data_out (stops here)
389 -                 // 6. ALU_out (Addr_out outside) (used at both MEM and WB)
390 -                 // 7. PC
391 -                 // Control Signals:
392 -                 // 1. mem_w
393 -                 // Pass-on:
394 -                 // 1. inst_in
395 -                 // Control.signals {
396 -                     // 2. DatatoReg (WB)
397 -                     // 3. Regwrite (WB)
398 -                     .IF_ID_is_first_jalr(IF_ID_is_first_jalr)
399 -                 );
400 -
401 -                 // ID:
402 -                 // From IF:
403 -                 // 1. inst_in
404 -                 // 2. PC
405 -                 // Control Signals:
406 -                 // 1. Regwrite - Regs : WB
407 -                 // 2. ALUSrc_A / ALUSrc_B (stops here)
408 -                 // References:
409 -                 // None
410 -                 // Pass-on:
411 -                 // 1. inst_in
412 -                 // Control_signals {
413 -                     // 2. ALU_Control
414 -                     // 3. DatatoReg
415 -                     // 4. mem_w
416 -                     // 5. Regwrite
417 -                     //
418 -                     // 6. ALU_A
419 -                     // 7. ALU_B
420 -                     // 8. Data_out
421 -                     // 9. PC
422 -                     // Out:
423 -                     // None
424 -                     .Get_rw_regs _rw_regs_ (
425 -                         // Input:
426 -                         .inst_in(IF_ID_inst_in[31:0]),
427 -                         // Output:
428 -                         .written_reg(IF_ID_written_reg), // Note that IF_ID_written_reg is just the destination
429 -                         // register number obtained from decoding the instruction and is not in REG_IF_ID
430 -                         .read_reg1(IF_ID_read_reg1), // Note that IF_ID_read_reg1 is just the operand1 register
431 -                         // number obtained from decoding the instruction and is not in REG_IF_ID
432 -                         .read_reg2(IF_ID_read_reg2) // Note that IF_ID_read_reg2 is just the operand2 register
433 -                         // number obtained from decoding the instruction and is not in REG_IF_ID
434 -                     );
435 -
436 -                     Controller Ctrl_Unit (
437 -                         // Input:
438 -                         .Opcode(IF_ID_inst_in[6:0]),
439 -                         .Fun(IF_ID_inst_in[14:21]),
440 -                         .Fun2(IF_ID_inst_in[31:25]),
441 -
442 -                         // Output:
443 -                         .ALUSrc_A(ALUSrc_A),
444 -                         .ALUSrc_B(ALUSrc_B[1:0]),
445 -                         .AU_Control(ALU_Control[4:0]),
446 -                         .DatatoReg(DatatoReg[1:0]),
447 -                         .mem_w(IF_ID_mem_w),
448 -                         .mem_r(IF_ID_mem_r), // 1 if the instruction reads from memory
449 -                         .Regwrite(Regwrite),
450 -                         .B_H((B_H)),
451 -                         .sign(sign) // not used yet
452 -                     );
453 -
454 -                     // Get the data from the register file
455 -                     Regs U2 (.clk(clk),
456 -                             .rst(rst),
457 -                             .LS(MEM_W_Regwrite), // From Write-Back stage
458 -                             .R_addr_A(IF_ID_inst_in[18:15]), // ID
459 -                             .R_addr_B(IF_ID_inst_in[24:20]), // ID
460 -                             .WT_addr(WT_addr[4:0]), // From Write-Back stage
461 -                             .WT_data(WT_data[31:0]), // From Write-Back stage
462 -                             .rdata_A(rdata_A[31:0]),
463 -                             .rdata_B(rdata_B[31:0])
464 -                     );
465 -
466 -                     assign IF_ID_Data_out = rdata_B; // for sw instruction, data from rs2 register written into
467 -                     // memory, but we need to take into account data forwarding (_mux_forward_data_out_)
468 -
469 -                     REQ_ID_EXE _id_exe_ (
470 -                         .clk(clk), .rst(rst), .CE(VS), .ID_EXE_dstall(ID_EXE_dstall), .ID_EXE_cstall(ID_EXE_cstall),
471 -                         // Input
472 -                         .inst_in(IF_ID_inst_in),
473 -                         .PC(IF_ID_PC),
474 -                         .ID_EXE_ALU_A(IF_EXE_ALU_A),
475 -                         .ID_EXE_ALU_B(IF_EXE_ALU_B),
476 -                         .ID_EXE_Imm_32(IF_ID_Imm_32),
477 -                         // To EXE stage, ALU Operands A & B
478 -                         .ALU_A(data_A[31:0]),
479 -                         .ALU_B(data_B[31:0]),
480 -                         // To MEM stage for sw instruction, data from rs2 register written into memory
481 -                         .mem_r(IF_ID_mem_r),
482 -                         // To WB stage, choosing different data written back to register file
483 -                         .DatatoReg(DatatoReg),
484 -                         // To WB stage, register file write valid
485 -                         .Regwrite(Regwrite),
486 -                         // For Data Hazard
487 -                         .written_reg(IF_ID_written_reg), .read_reg1(IF_ID_read_reg1), .read_reg2(IF_ID_read_reg2),
488 -                         .add_branch_out(IF_ID_add_branch_out),
489 -                         .zero_prediction(IF_ID_zero_prediction),
490 -                         .is_bne(IF_ID_is_bne),
491 -
492 -                         // Output
493 -                         .ID_EXE_inst_in(IF_EXE_inst_in),
494 -                         .ID_EXE_PC(IF_EXE_PC),
495 -                         .ID_EXE_Imm_32(IF_EXE_Imm_32),
496 -                         .ID_EXE_ALU_A(IF_EXE_ALU_A), // Wire from REG_ID_EXF to the ALU Forwarding Multiplexer (A)
497 -                         .ID_EXE_ALU_B(IF_EXE_ALU_B), // Wire from REG_ID_EXF to the ALU Forwarding Multiplexer (B)
498 -                         .ID_EXE_ALUSrc_A(IF_EXE_ALUSrc_A),
499 -                         .ID_EXE_ALUSrc_B(IF_EXE_ALUSrc_B),
500 -                         .ID_EXE_ALU_Control(IF_EXE_ALU_Control),
501 -                         .ID_EXE_Data_out(IF_EXE_Data_out),
502 -                         .ID_EXE_mem_w(IF_EXE_mem_w),
503 -                         .ID_EXE_mem_r(IF_EXE_mem_r),
504 -                         .ID_EXE_DatatoReg(IF_EXE_DatatoReg),
505 -                         .ID_EXE_Regwrite(IF_EXE_Regwrite),
506 -                         // For Data Hazard
507 -                         .written_reg(IF_EXE_written_reg), .ID_EXE_read_reg1(IF_EXE_read_reg1),
508 -                         .ID_EXE_read_reg2(IF_EXE_read_reg2),
509 -                         .ID_EXE_zero_prediction(IF_EXE_zero_prediction),
510 -                         .ID_EXE_PC_after_flush(IF_EXE_PC_after_flush),
511 -                         .ID_EXE_is_bne(IF_EXE_is_bne),
512 -                     );

```

```

398 -      // )
399 -      // 4. ALU_out (Addr_out outside) (used at both MEM and WB)
400 -      // 5. PC
401 -      // 6. Data_in
402 -      // Out:
403 -      // Data_out & mem_w, ALU_out(as Addr_out)
404 -
405 -      REG_MEM_WB _mem_wb_ (
406 -        .clk(clk), .rst(rst), .CE(VS),
407 -        // Input
408 -        .inst_in(EXE_MEM_Inst_in),
409 -        .PC(EXE_MEM_PC),
410 -        .ALU_out(EXE_MEM_ALU_out),
411 -        .DataReg(EXE_MEM_DataReg),
412 -        .RegWrite(EXE_MEM_RegWrite),
413 -        ///// Comes from data memory
414 -        .Data_in(data_in),
415 -
416 -        // Output
417 -        .MEM_WB_Inst_in(MEM_WB_Inst_in),
418 -        .MEM_WB_PC(MEM_WB_PC),
419 -        .MEM_WB_ALU_out(MEM_WB_ALU_out),
420 -        .MEM_WB_DataReg(MEM_WB_DataReg),
421 -        .MEM_WB_RegWrite(MEM_WB_RegWrite),
422 -        .MEM_WB_Data_in(MEM_WB_Data_in),
423 -      );
424 -
425 -      // WB:-----
426 -      // From EXE:
427 -      // 1. inst_in
428 -      // Control signals (
429 -      // 2. DataReg (WB)
430 -      // 3. RegWrite (WB)
431 -      // )
432 -      // 4. ALU_out (Addr_out outside) (used at both MEM and WB)
433 -      // local:
434 -      wire [31:0] Lo_data;
435 -
436 -      assign Wt_addr[4:0] = MEM_WB_Inst_in[11:7]; // rd, except for branch and store instructions
437 -      LUT_or_ALUPC _Loa_ (
438 -        .inst_in(MEM_WB_Inst_in[31:0]),
439 -        .PC(MEM_WB_PC),
440 -        .data(Loa_data[31:0])
441 -      );
442 -      Mux4to1b32_MUX3 (
443 -        .IO(MEM_WB_ALU_out[31:0]), // Others
444 -        .I1(MEM_WB_Data_in[31:0]), // Load
445 -        .I2(Loa_data[31:0]), // LUI and AUIPC
446 -        .I3(MEM_WB_PC[31:0] + 32'b0100), // jal and jalr: PC + 4
447 -        .I4(MEM_WB_DataReg[1:0]),
448 -        .O(we_data[31:0]);
449 -
450 -    endmodule

```

```

398 +      // EXE:-----
399 +      // From ID:
400 +      // 1. inst_in
401 +      // Control_signals {
402 +      // 2. ALU_Control (stops here)
403 +      // 3. mem_w
404 +      // 4. DataReg
405 +      // 5. RegWrite
406 +      // )
407 +      // 6. ALU_A (stops here)
408 +      // 7. ALU_B (stops here)
409 +      // 8. Data_out
410 +      // 9. PC
411 +      // Control Signals:
412 +      // 1. ALU_Control
413 +      // References:
414 +      // None
415 +      // Pass-on:
416 +      // 1. Inst_in
417 +      // Control_signals {
418 +      // 2. DataReg (WB)
419 +      // 3. mem_w (MEM)
420 +      // 4. RegWrite (WB)
421 +      // )
422 +      // 5. Data_out (used at MEM together with mem_w)
423 +      // 6. ALU_out (Addr_out outside) (used at both MEM and WB)
424 +      // 7. PC
425 +      // Out:
426 +      // None
427 +
428 +      // ALU Forwarding Multiplexer (A)
429 +      Mux4to1b32_mux_forward_ALU_A_ (
430 -        .IO(ID_EXE_ALU_A[31:0]), // Wire from REG_ID_EXE
431 -        .I1(we_data[31:0]), // Output from MEM_WB Register
432 -        .I2(EXE_MEM_ALU_out[31:0]), // Output from EXE_MEM Register
433 -        .O(forwarding_A_sig[1:0]), // Forwarding signal
434 -        .O(ID_EXE_ALU_A_forward[31:0]) // Wire to ALU
435 +
436 +
437 +      // ALU Forwarding Multiplexer (B)
438 +      Mux4to1b32_mux_forward_ALU_B_ (
439 -        .IO(ID_EXE_ALU_B[31:0]), // Wire from REG_ID_EXE
440 -        .I1(we_data[31:0]), // Output from MEM_WB Register
441 -        .I2(EXE_MEM_ALU_out[31:0]), // Output from EXE_MEM Register
442 -        .O(forwarding_B_sig[1:0]), // Forwarding signal
443 -        .O(ID_EXE_ALU_B_forward[31:0]) // Wire to ALU
444 +
445 +
446 +      // Data Out Forwarding Multiplexer
447 +      Mux4to1b32_mux_forward_data_out_ (
448 -        .IO(ID_EXE_Data_out[31:0]), // Wire from REG_ID_EXE
449 -        .I1(we_data[31:0]), // Output from MEM_WB Register
450 -        .I2(EXE_MEM_ALU_out[31:0]), // Output from EXE_MEM Register
451 -        .O(forwarding_B_sig[1:0]), // Forwarding signal
452 -        .O(ID_EXE_Data_out_forward[31:0]) // Wire to EXE_MEM Register
453 +
454 +
455 +      Mux2to1b32_ALU_source_A_ ( // Moved from ID stage to EXE stage
456 -        .IO(ID_EXE_ALU_A_forward[31:0]),
457 -        .I1(ID_EXE_Imm_32[31:0]), // not used
458 -        .O(ID_EXE_ALUSrc_A),
459 -        .O(ALU_A[31:0])
460 +
461 +
462 +      Mux4to1b32_ALU_source_B_ ( // Moved from ID stage to EXE stage
463 -        .IO(ID_EXE_ALU_B_forward[31:0]),
464 -        .I1(ID_EXE_Imm_32[31:0]),
465 -        .I2(),
466 -        .I3(),
467 -        .O(ID_EXE_ALUSrc_B[1:0]),
468 -        .O(ALU_B[31:0])
469 +
470 +
471 +      ALU_ALUAlu_ (
472 -        // Input:
473 -        .A(ALU_A[31:0]),
474 -        .B(ALU_B[31:0]),
475 -        .ALU_operation(ID_EXE_ALU_Control[4:0]),
476 -        .is_pne(ID_EXE_is_pne),
477 -        // Output:
478 -        .res(ID_EXE_ALU_out[31:0]),
479 -        .overflow(),
480 -        .zero_correct(zero_correct) // Now zero is calculated by ALU (connected to Branch Checker to
determine if predicted correctly)
481 -      );
482 +
483 +      // Branch Checker
484 +      Branch_Checker_branch_checker_ (
485 +        // Input:
486 +        .zero_correct(zero_correct),
487 +        .ID_EXE_zero_prediction(ID_EXE_zero_prediction),
488 +        .OPcode(ID_EXE_Inst_in[6:0]),
489 +
490 +        // Output:
491 +        .correct_prediction(correct_prediction),
492 +        .is_branch(is_branch)
493 +
494 +
495 +      REG_EXE_MEM_exe_mem_ (
496 -        .clk(clk), .rst(rst), .CE(VS),
497 -        // Input
498 -        .Inst_in(ID_EXE_Inst_in),
499 -        .PC(ID_EXE_PC),
500 -        ///// To MEM stage
501 -        .ALU_out(ID_EXE_ALU_out),
502 -        .Data_out(ID_EXE_Data_out_forward),
503 -        .mem_w(ID_EXE_mem_w),
504 -        ///// To WB stage
505 -        .DataReg(ID_EXE_DataReg),
506 -        .RegWrite(ID_EXE_RegWrite),
507 -
508 -        .Written_reg(ID_EXE_written_reg), .read_reg(ID_EXE_read_reg1), .read_reg2(ID_EXE_read_reg2),
509 -
510 +        // Output
511 -        .EXE_MEM_Inst_in(EXE_MEM_Inst_in),
512 -        .EXE_MEM_PC(EXE_MEM_PC),
513 -        .EXE_MEM_ALU_out(EXE_MEM_ALU_out),
514 -        .EXE_MEM_Data_out(EXE_MEM_Data_out),
515 -        .EXE_MEM_mem_w(EXE_MEM_mem_w),
516 -        .EXE_MEM_DataReg(EXE_MEM_DataReg),
517 -        .EXE_MEM_RegWrite(EXE_MEM_RegWrite),
518 -
519 -        .EXE_MEM_written_reg(EXE_MEM_written_reg), .EXE_MEM_read_reg1(EXE_MEM_read_reg1),
520 -        .EXE_MEM_read_reg2(EXE_MEM_read_reg2)
520 +

```

```

521 + // MEM:-----
522 + // From EXE:
523 + // 1. inst_in
524 + // Control_signals {
525 + // 2. DatatoReg (WB)
526 + // 3. mem_w (stop here)
527 + // 4. RegWrite (WB)
528 + // }
529 + // }
530 + // 5. Data_out (stops here)
531 + // 6. ALU_out (Addr_out outside) (used at both MEM and WB)
532 + // 7. PC
533 + // Control Signals:
534 + // 1. mem_w
535 + // Pass-on:
536 + // 1. inst_in
537 + // Control_signals {
538 + // 2. DatatoReg (WB)
539 + // 3. RegWrite (WB)
540 + // }
541 + // 4. ALU_out (Addr_out outside) (used at both MEM and WB)
542 + // 5. PC
543 + // 6. data_in
544 + // Out:
545 + // Data_out & mem_w, ALU_out(as Addr_out)
546 +
547 + REQ_MEM_WB _mem_wb_(
548 + .clk(clk), .rst(rst), .CE(V5),
549 + // Input
550 + .inst_in(EXE_MEM_inst_in),
551 + .PC(EXE_MEM_PC),
552 + .ALU_out(EXE_MEM_ALU_out),
553 + .DatatoReg(EXE_MEM_DatatoReg),
554 + .RegWrite(EXE_MEM_RegWrite),
555 + .written_reg(EXE_MEM_written_reg),
556 + ///// Comes from data memory
557 + .Data_in(data_in),
558 +
559 + // Output
560 + .MEM_WB_inst_in(MEM_WB_inst_in),
561 + .MEM_WB_PC(MEM_WB_PC),
562 + .MEM_WB_ALU_out(MEM_WB_ALU_out),
563 + .MEM_WB_DatatoReg(MEM_WB_DatatoReg),
564 + .MEM_WB_RegWrite(MEM_WB_RegWrite),
565 + .MEM_WB_Data_in(MEM_WB_Data_in),
566 + .MEM_WB_written_reg(MEM_WB_written_reg)
567 + );
568 +
569 + // WB:-----
570 + // From EXE:
571 + // 1. inst_in
572 + // Control_signals {
573 + // 2. DatatoReg (WB)
574 + // 3. RegWrite (WB)
575 + // }
576 + // 4. ALU_out (Addr_out outside) (used at both MEM and WB)
577 + // Local:
578 + wire [31:0] loA_data;
579 +
580 + assign Wt_addr[4:0] = MEM_WB_inst_in[11:7]; // rd, except for branch and store instructions
581 + assign Wt_on_AUIPC_loa_(
582 + .inst_in(MEM_WB_inst_in[31:0]),
583 + .PC(MEM_WB_PC),
584 + .data((loA_data[31:0])
585 + ));
586 + Mux4to132_MUX3 (
587 + .I0(MEM_WB_ALU_out[31:0]), // Others
588 + .I1(MEM_WB_Data_in[31:0]), // Load
589 + .I2(loA_data[31:0]), // LUI and AUIPC
590 + .I3(MEM_WB_PC[31:0] + 32'b0100), // jal and jalr: PC + 4
591 + .O(MEM_WB_DatatoReg[1:0]),
592 + .O(Wt_data[31:0]));
593 +
594 + // Forward Unit
595 + Forward_Unit _forward_unit_(
596 + // Input:
597 + .ID_EXE_read_reg1(ID_EXE_read_reg1),
598 + .ID_EXE_read_reg2(ID_EXE_read_reg2),
599 +
600 + .EXE_MEM_RegWrite(EXE_MEM_RegWrite),
601 + .EXE_MEM_written_reg(EXE_MEM_written_reg),
602 +
603 + .MEM_WB_RegWrite(MEM_WB_RegWrite),
604 + .MEM_WB_written_reg(MEM_WB_written_reg),
605 +
606 + // Output:
607 + .forwarding_A_sig(forwarding_A_sig),
608 + .forwarding_B_sig(forwarding_B_sig),
609 + );
610 +
611 + 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
fd310e3
00100313
fff30393
00239393
013383b3
0003ae03
000e0f33
00438393
0003ae03
01cf0f33
00438393
0003ae03
01cf0f33
00030393
00239393
012383b3
01e3a023
00130313
fd310e3
00128293
f69296e3
00008067
00000293
00090e13
00098e93
00228313
006e2023
00130313
006ea023
004e0e13
004e8e93
00128293
fe8292e3
00008067
02092283
0092d293
ee328293
00502023
```