Department of Electronics and Computer Engineering

The Hong Kong University of Science and Technology

ELEC5140 Advanced Computer Architecture Project

Kei Hong Chan & Yiu Fai Lam

25/05/2021

Micro-Architecture Optimization on single-issue 5-stage RISC-V processor

Abstract

RISC-V is an instruction set architecture (ISA), which comes from the University of California, Berkeley. Those instructions can be used to build a processor. Since the RISC-V is a free and open ISA, people are not required to apply any patents to design a processor chip. With the aid of the RISC-V, everyone can customize and fully utilize their own chip, without wasting any space or energy for extra function or extra storage that are designed by other companies. Nowadays, many products are required to use microprocessor to do some basic and efficient functions in an embedded system, in general computing and in advanced high-performance computing, companies and researchers are therefore actively involved in the RISC-V community.

The aim of this project is to modify the single-issue 5-stage RISC-V processor to improve the performance of the RISC-V processor by reducing the execution time of programs. The implementation of the processor would be in Verilog code. The challenge of this project is to solve the problem of hazard. Such as data hazard and control hazard.

Content

1. Introduction and related work

A simple 32-bit single-issue 5-stage RISC-V processor is implemented by RipperJ and posted on github [1]. The processor provided a list of simple instructions. Such as “ADD”, “SUB”, “LW”, “BNE”, “JALR”, “SLTI”. In addition, a zero generator is implemented which is used for branch instruction, and it is placed in ID stage. And a sketch of this processor structure is shown in Figure 1.

However, this simple RISC-V processor uses stall to solve hazard problems, which will degrade the performance significantly. Then, our task is to improve the processor and reduce the Clock Per Instruction (CPI) to as low as possible. We designed the following modules to be added in the processor to achieve low CPI, and detail discussion of each module is in section 2.

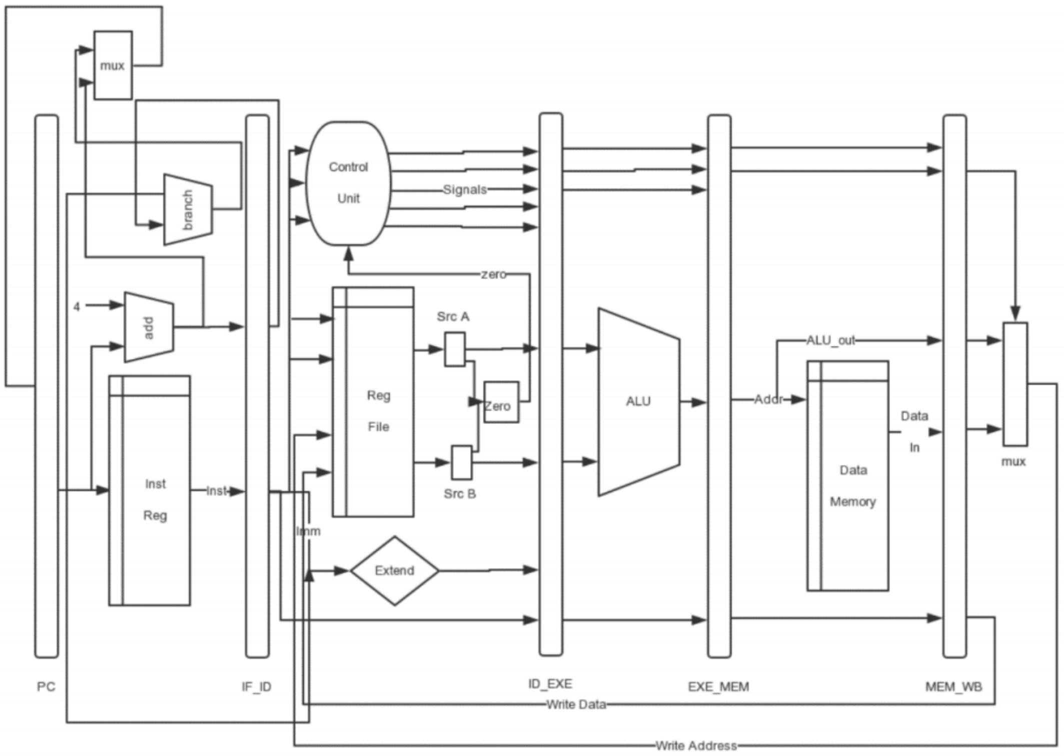


Figure 1: Sketch of provided RISC-V processor, from [2]

The implementation of additional modules begins with data forwarding, which used to solve the data hazard of read-after-write (RAW) dependency. Then, branch predictor is used to predict whether the branch instruction is taken or not, which used to reduce control hazard. Also, the zero generator in instruction decode (ID) stage aiding the branch predictor to update the content as soon as possible. In addition, the data forwarding module also needs to forward the data to the ID stage to allow the zero generator to compute with the latest data. And a dependency check module is needed in instruction fetch (IF) stage to prevent data hazards for load and branch instruction.

Since the aim of this project is to reduce the execution time of programs. Therefore, a set of benchmark programs would be used, and the execution time for each benchmark program would be recorded. Also, this project would discuss the improvement on performance of the optimized processor by comparing the execution cycle of the original single-issue 5-stage RISC-V processor and the optimized processor.

2. Methods/Methodology

2.1 Data forwarding

From the original processor, it needs to stall while hazards happen. On the point of above, using the data forwarding is necessary to emulate the stall to achieve higher CPI for the processor. Therefore, when the data required from the execution (EXE) stage and the data is still not being stored into the register, the processor can forward the data to the ALU for further calculation rather than waiting for the data to be stored in the register.

2.1.1 ALU data forwarding

Data Hazard
When doing the pipelining design, the instruction is executed at the same time in different stage. Therefore, the data dependency will happen while the instruction is not completed. To solve this situation the data can forward from memory/write-back (MEM/WB) stage to the EXE/ID stage first to use the data as soon as possible.

In this project, because the processor is single-issue, the data hazard is only need to consider on the RAW situation. For example, Table 1 shows the instruction sequence of RAW situation, R2 has data hazard between the first instruction and the second instruction. Furthermore, the R2 also has dependency in between first and third instruction. On the other words, the table 1 has two data hazard which come from MEM stage to EXE stage and WB stage to EXE stage.

RAW in ALU data	RAW in MEM data	RAW in SW ALU data	RAW in SW MEM data
ADD R2,R1,R3	LW R2,0(R1)	ADD R2,R1,R3	LW R2,0(R1)
ADD R4,R2,R3	ADD R3, R2,R4	SW R2,0(R1)	SW R2,0(R1)
ADD R5,R2,R1	ADD R5, R2,R4	SW R2,0(R1)	SW R2,0(R1)

Table 1: RAW Data Hazard demonstration

Data forward
Figure 2 shows the EXE stage circuit, the ALU data and data out data connected with a 4-to-1 mux and a 2-to-1 mux. The 4-to-1 mux is used to choose the data for forwarding, the first half signal “EXE_MEM_ALU_out” and “data_in” is the data from MEM stage and the second half signal “MEM_WB_ALU_out” and “MEM_WB_Data_in” is the

WB stage signal. After this, the 2-to-1 mux is used to choose the signal in between original data from register and forward data from the 4-to-1 mux. The control signal will be discussed in section 2.1.2.

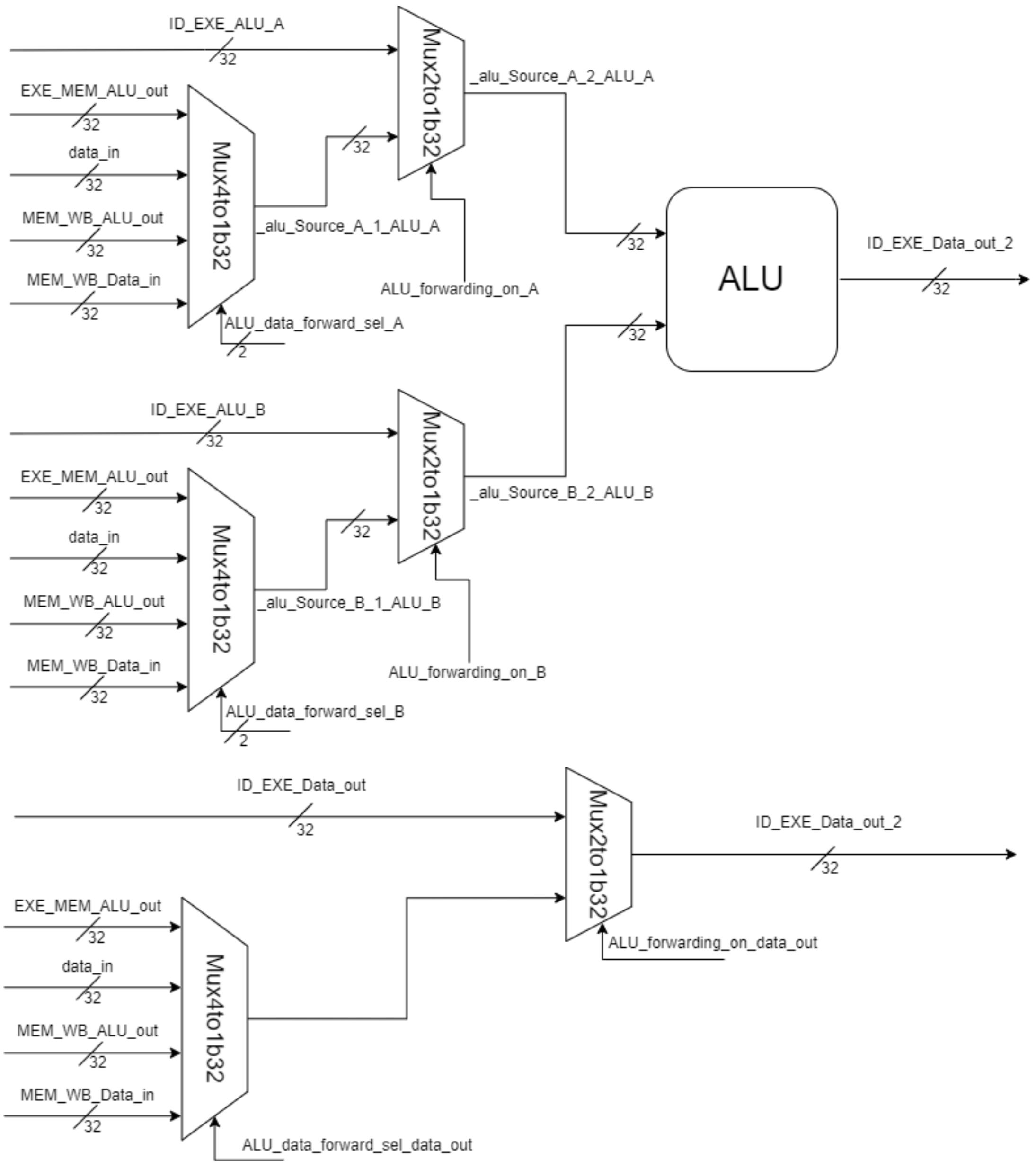


Figure 2: ALU data flow in EXE stage

2.1.2 DF_control design

The “DF_control” module is designed for control which data to be token. To check which data need to be forward, the “DF_control” module is required the read and write address from ID, EXE and MEM stage. In addition, “DatatoReg” and “mem_w” signal also needs from the above stage to determent which type of instruction is using. For more detail design it can review in the Appendix.I, it has shown all the condition for the DF_Control design and notice which pin was used to check the hazard.

ALU_data_forward_sel_X and ALU_forwarding_on_X

The control signal "ALU_data_forward_sel" is the main selection signal for data forwarding, it comes from 4 signals that generated in "DF_control" module. Those 4 signals are the signal checking for ALU and memory and the stage checking for EXE and MEM. By compare the "EXE_MEM_written_data" signal to each "read_reg" signals, the result signal can identify which data signal has data dependency and select the data for forwarding.

SW instruction

Due to the SW instruction needs a "data_out" signal for memory module to store the data, the EXE stage was designed a data forwarding module to do so. Therefore, the "DF_control" module required "mem_w" signal to check is the instruction has data dependency on "data_out" signal or not.

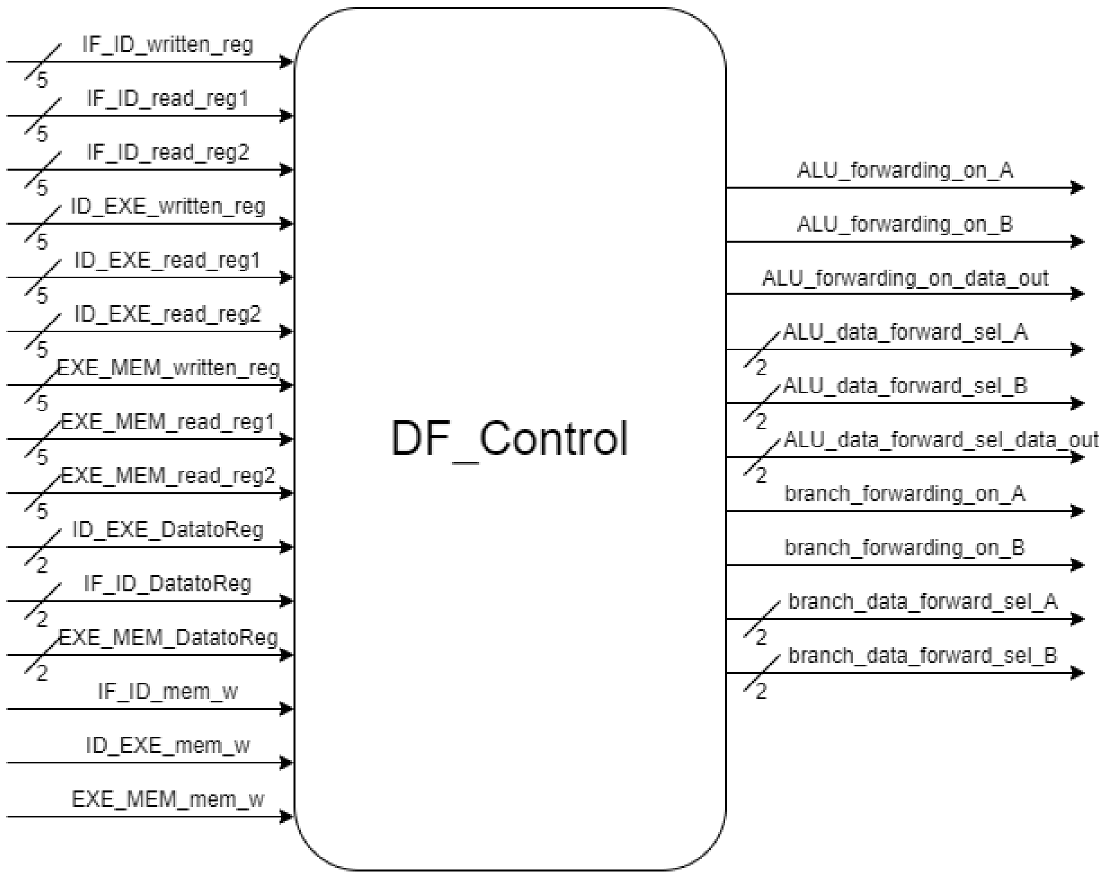


Figure 3: DF_Control overview

2.1.3 ID data forwarding

The ID data forwarding was used for the “ID_Zero_Generator” module, which used for the branch predictor actual result check. Figure 4 shows the structure of the module, it is similar to the data forward in EXE stage. The difference is the result of ALU is need to forward back to "ID_Zero_Generator" if dependency occurs. Also, the data in WB stage is not required to forward to "ID_Zero_Generator" because the data is stored in the data register within the same clock cycle at the falling edge, and the correct values is output from the register and read by the "ID_Zero_Generator".

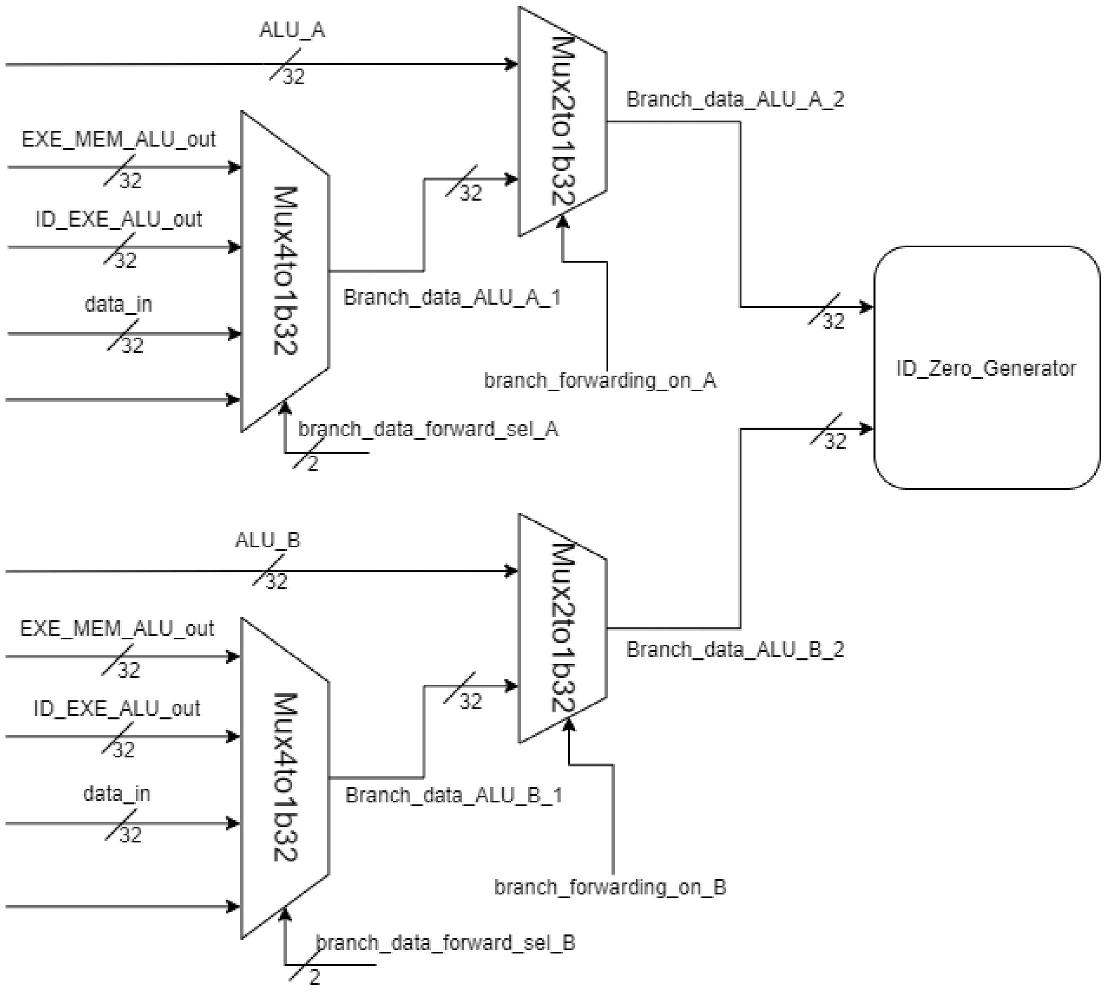


Figure 4: Data flow for ID_Zero_Generator

2.2 Branch instruction and Branch predictor

From the original design, an extra stall would happen when a branch instruction is taken. It means the performance would significantly degrade when branch instruction tends to be taken. Then, branch predictor is trying to eliminate this stall by predict the branch result in the IF stage. Also, a flush operation is needed for this modification, and flush would cause one extra clock cycle to execute the program. In addition, flush only happen when the prediction is wrong.

2.2.1 BHR_and_PHT

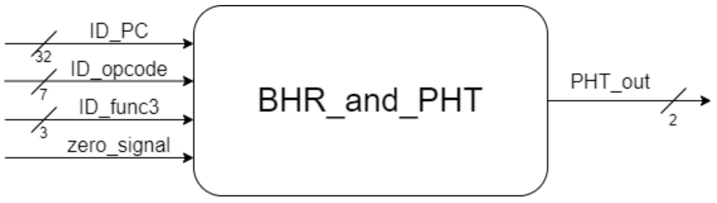


Figure 5: BHR_and_PHT overview

The figure above shows the input and output ports of "BHR_and_PHT" module, and this module is used to store and update the Branch History Register (BHR) and Pattern History Table (PHT). "ID_PC", "ID_opcode", "ID_func3", "zero_signal" are the input signals of this module. "PHT_out" is the output of this module.

"ID_PC", "ID_opcode", "ID_func3" and "zero_signal" means the program counter (PC) in ID stage, opcode in ID stage, func3 in ID stage and zero signal generated by the zero generator in ID stage respectively. "PHT_out" is connected to the "IF_mux_sel" module and passes the information of PHT.

The BHR is a 4-bit global BHR, it would be updated by shift 1 bit left. Which means the least significant bit is used to store the actual result of the current computed branch and other bits record the actual result of the previous computed branch. Moreover, for BHR, "1" means actual branch taken and "0" means actual branch not taken.

The PHT has 128 entries and each entry stores content of 2-bit Branch Predictor, and the 2-bit Branch Predictor uses the Saturating Up/Down scheme. Also, the combination of ID_PC[4:2] and 4-bit global BHR is referred to the address of the PHT. By using this technique, it can reduce the probability for aliasing. And the content of PHT used to predict whether the branch instruction in IF stage is taken or not. And the content of PHT would pass to the "IF_MUX_sel" module.

In addition, to update the content of PHT and BHR, the content of PHT is updated at the rising edge and stored at the location where it is read from before the rising edge. Also, BHR would be updated simultaneously with the content of PHT. After that, the content of PHT would be stored at the falling edge and pass to "IF_MUX_sel" module before the BHR has been updated. This method can effectively read and update the BHR and PHT within 1 clock cycle.

2.2.2 IF_MUX_sel

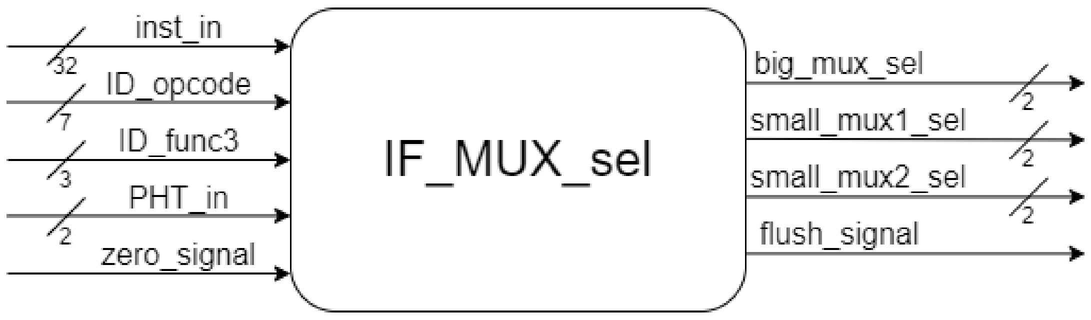


Figure 6: IF_MUX_sel overview

The figure above shows the input and output port of the "IF_MUX_sel" module, and this module is used to select the correct PC value for the PC module to update. "inst_in", "ID_opcode", "ID_func3", "PHT_in", "zero_signal" are the input signals of this module. "big_mux_sel", "small_mux1_sel", "small_mux2_sel", "flush_signal" are the output of this module.

The signal of "inst_in", "ID_opcode", "ID_func3" and "zero_signal" is the same as the "BHR_and_PHT" module. And "PHT_out" is the content of PHT that passes from the "BHR_and_PHT" module. "big_mux_sel", "small_mux1_sel", "small_mux2_sel" are the select signals for the MUX in IF stage. "flush_signal" is the signal used to flush the PC value of the PC module, send a NOP instruction to ID stage, and flush the instruction which is stored in the "REG_IF_ID" module.

This module is connected between the "BHR_and_PHT" module and the select signal of the multiplexer (MUX) in IF stage. The content of PHT from the "BHR_and_PHT" module is used to control the output of the 2 MUX (highlighted orange in Figure 8), which are the values that need to add them together to become the PC value for next instruction to fetch. The control signal of the 2 MUX depends on the predicted result and actual branch result. Also, the flush signal become logic "1", if the prediction is wrong. Furthermore, the instruction in IF and ID stage will determine which calculated PC value should be selected by controlling the select signal of 4-to-1 MUX. And the figures below show the connection of the MUX before and after adding the "IF_MUX_sel" module. The lines and font in colour indicate the change of signals and components.

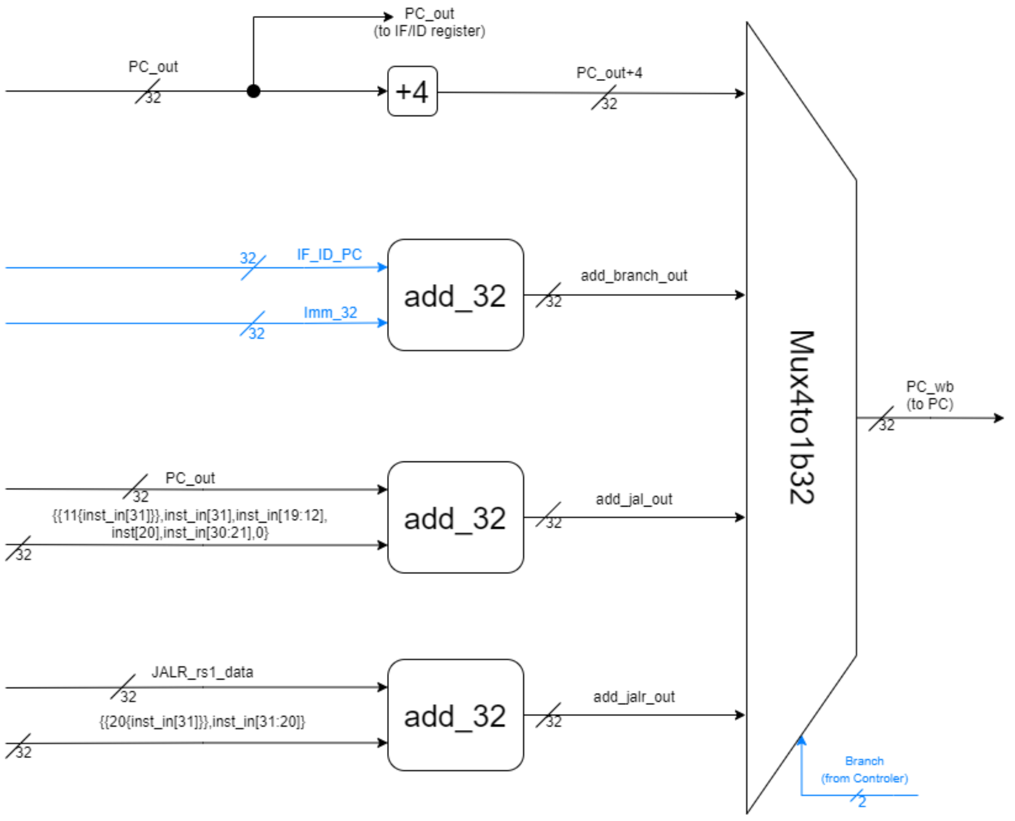


Figure 7: Connection of MUX in IF stage (before adding "IF_MUX_sel" module)

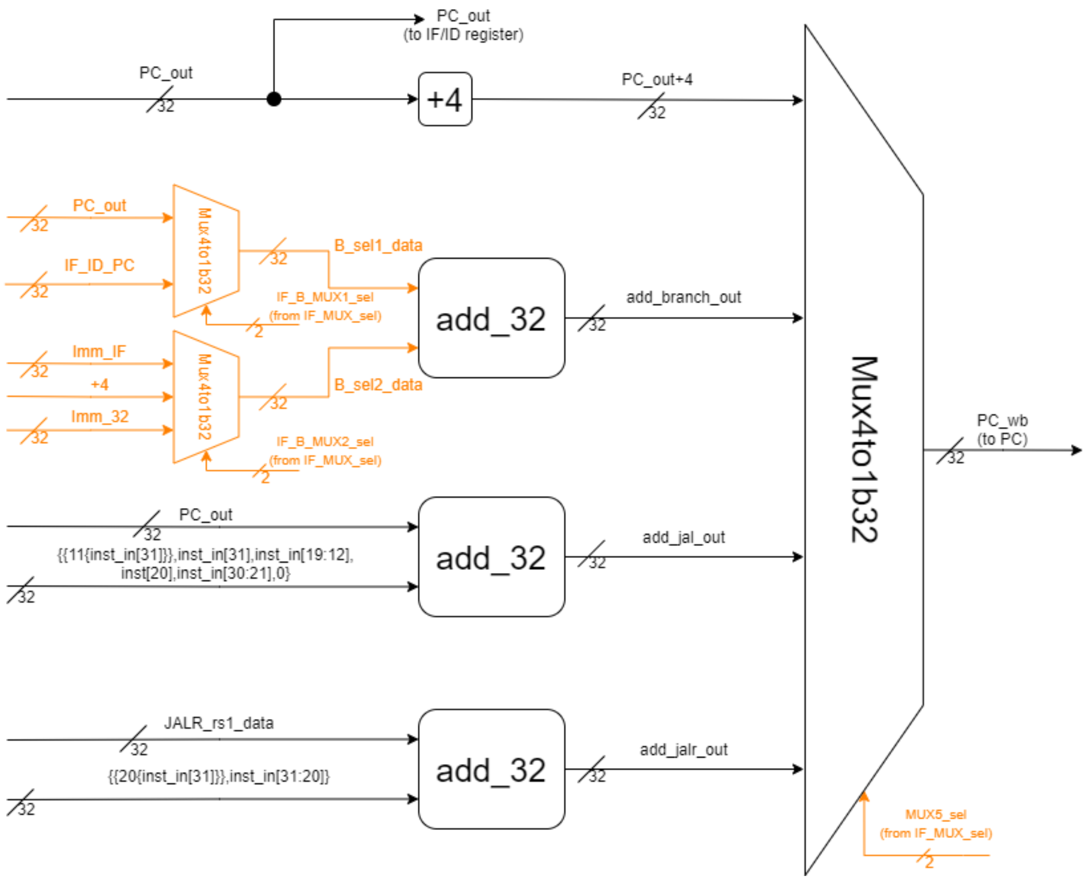


Figure 8: Connection of MUX in IF stage (after adding "IF_MUX_sel" module)

Where “IF_B_MUX1_sel” is connected to “small_mux1_sel”, “IF_B_MUX2_sel” is connected to “small_mux2_sel” and “MUX5_sel” is connected to “big_mux_sel”.

In terms of the scenario of branch, when a branch is predicted as taken, the current PC value would add to an immediate value, which is extracted from the instruction in IF stage. When the branch is predicted as not taken, then the current PC is added by 4 and executes in sequence. If the prediction is correct, then PC value is no need to recover and continue execute without any stall. Furthermore, if a branch is predicted as taken and actually should not be taken, which means the program should be executed in sequence, then the value of PC in ID stage would be added by 4, and this calculated value is used to recover the value of PC. If a branch is predicted as not taken and actually should be taken, which means the program should be jump to the “LABEL” of the program, then the value of PC in ID stage would be added by the value of “Imm_32” to jump to the “LABEL” of the program. Since flush is needed for prediction wrong, an extra clock cycle would be introduced.

In addition, according to Figure 8, when “IF_B_MUX1_sel” is 0, it will select “PC_out” and 1 will select “IF_ID_PC”. Also, when “IF_B_MUX2_sel” is 0, it will select “Imm_IF”, 1 will select positive integer 4, 2 will select “Imm_32”. Moreover, “Imm_IF” is an immediate value extracted from “inst_in” and extended to 32 bits. In addition, 0,1,2,3 for “MUX5_sel” means select “PC_out+4”, “add_branch_out”, “add_jal_out”, “add_jalr_out” respectively. Then, the select signal for MUX would follow the scenario of branch as discussed above. For example, both “IF_B_MUX1_sel” and “IF_B_MUX2_sel” are 0 when a branch is predicted as taken.

2.2.3 branch_load_check

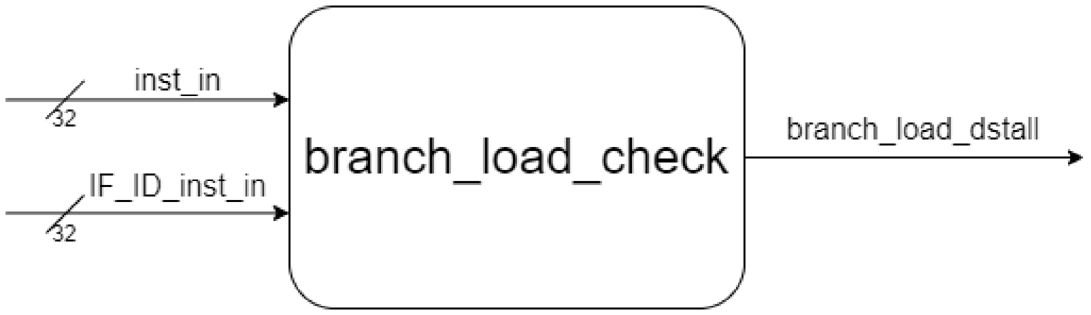


Figure 9: branch_load_check overview

The figure above shows the input and output port of the “branch_load_check” module, and this module is used to check the dependency between branch and load instruction. According to our design, if a load instruction is followed with a branch instruction and have dependency, the data is not able to forward to the zero generator to calculate the actual branch result. For instant, if the branch instruction already in the ID stage, the load instruction is still in the execution (EXE) stage and calculating the memory address if no extra stall. And the RAM data only can be read in memory (MEM) stage. Then, it causes a problem. So, an extra 1 clock cycle stall is needed to allow the load instruction is in MEM stage when the branch instruction is in ID stage, also read the data from memory and forward back to the zero generator in ID stage.

From Figure 9, “inst_in” is the instruction in IF stage and “IF_ID_inst_in” is the instruction in ID stage. “branch_load_dstall” is the output signal which connected to IF/ID register and PC module. Input signals are used to check the opcode in both ID and IF stage, and the register address will be extract from the input signal to check dependency. If load instruction is in ID stage and branch instruction is in IF stage, also the write address (rd) for load instruction is match to the read address (rs1 or rs2) for branch instruction. Then, “branch_load_dstall” will output logic 1. After that, the PC value will not update, and the IF/ID register will send a NOP instruction to ID stage.

2.3 Eliminate jump stall

Similar to branch instruction, the PC value for jump instruction can only be obtained in ID stage. It is because the component to extract immediate value from instruction is in ID stage, and the register that used to store the executed temporary data also located in ID stage. So, an extra stall is introduced to allow the jump instruction to go to ID stage and execute the correct PC value for original processor.

The method to eliminate the jump instruction stall is to add a MUX before the instruction memory, and this MUX will select the correct PC value for different instructions and sent it to instruction memory. Also, the control signal of this MUX (PC_jump_sel) is control by the “Controler” module. When JAL instruction is in ID stage, “PC_jump_sel” is changed to 1. When JALR instruction is in ID stage, “PC_jump_sel” is changed to 2. Moreover, the PC value that send to the instruction memory also will send to the register which is placed between IF and ID. Furthermore, the input signals “add_jal_out” and “add_jalr_out” in 4-to-1 MUX is modified to add an extra value of 4 to both signals. It is because the calculated PC value for jump instruction is directly sent to the IF/ID register, then the PC value of next instruction would be jumped PC value add with 4 when there has a jump instruction. The figures below show the connection of the MUX before and after eliminate the jump stall. The lines and font in colour indicate the change in signals and components

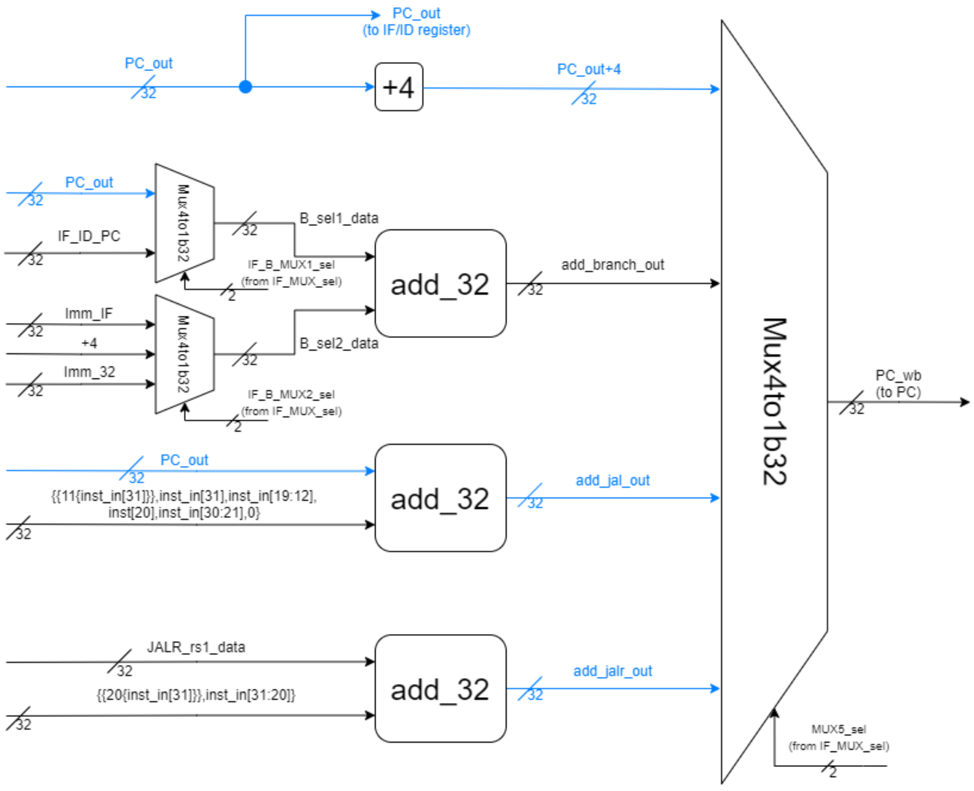


Figure 10: Connection of MUX in IF stage (before eliminate jump stall)

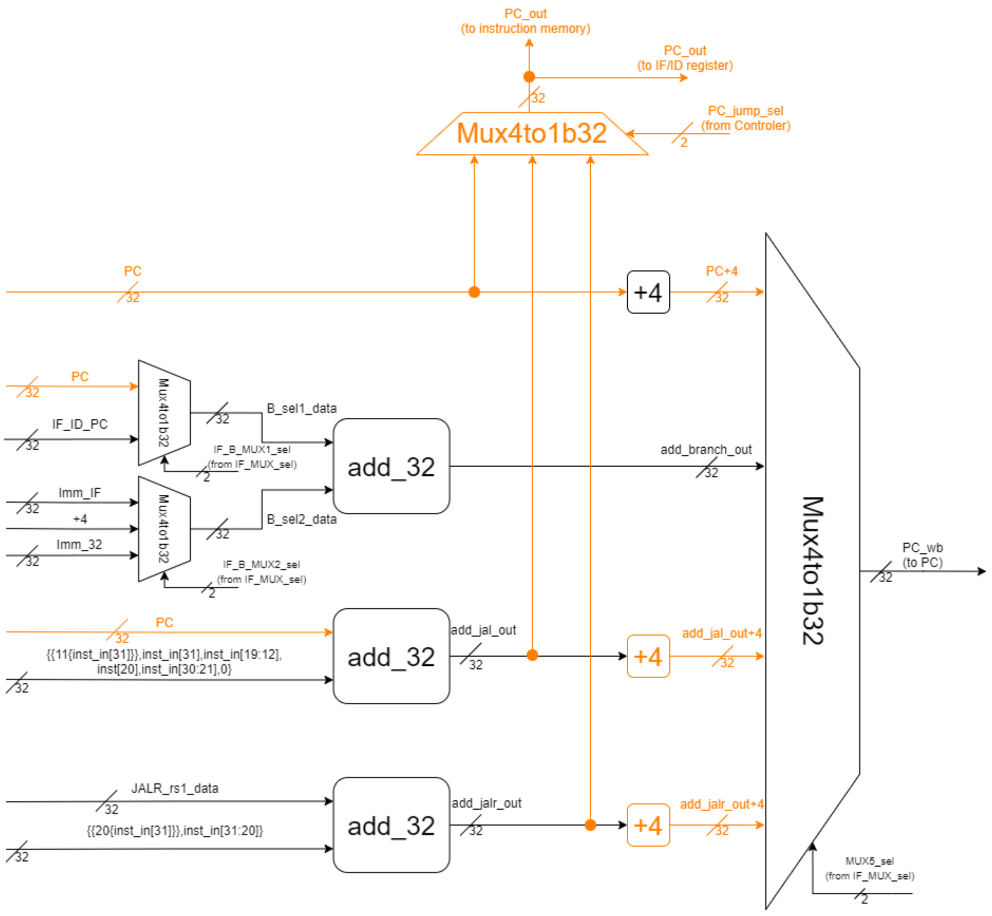


Figure 11: Connection of MUX in IF stage (after eliminate jump stall)

2.4 Load_extend and Store_extend

Figure 12 and Figure 13 show the modules that used to support other types of load and store instruction, and Table 2 shows the Load and Store instruction added for the processor. From Figure 12, "RAM_data_in", "EXE_MEM_LOAD_type", "EXE_MEM_LOAD_sign" are the inputs of "LOAD_extend" module. "data_in" is the output of "LOAD_extend" module. From Figure 13, "ID_EXE_Data_out_2", "ID_EXE_STORE_type" are the inputs of "STORE_extend" module. "ID_EXE_Data_extended" is the output of "STORE_extend" module. In addition, "STORE_extend" module is placed in EXE stage and extend the data after data forwarding for store instruction, "LOAD_extend" is placed in MEM stage and extend the data after received the data from RAM.

The meaning of "LOAD_type" and "STORE_type" is what type of instruction is for the load and store instruction. For example, LH means load half word from the memory, and "EXE_MEM_LOAD_type" would be 2 and "EXE_MEM_LOAD_sign" would be 1. Then, only least significant 16 bit is extracted from the "RAM_data_in" and extend the most significant bits by copying the 16th bit for 16 times. Also, LBU means load unsigned byte from the memory, and "EXE_MEM_LOAD_type" would be 1 and "EXE_MEM_LOAD_sign" would be 0. Then, only least significant 8 bit is extracted from the "RAM_data_in" and extend the most significant 24 bits by inserting 0's. And similar implementation is applied to the "STORE_extend" module but without a sign signal.

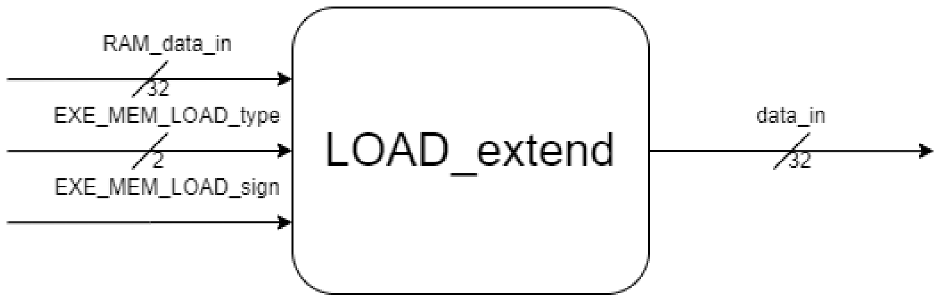


Figure 12: LOAD_extend overview

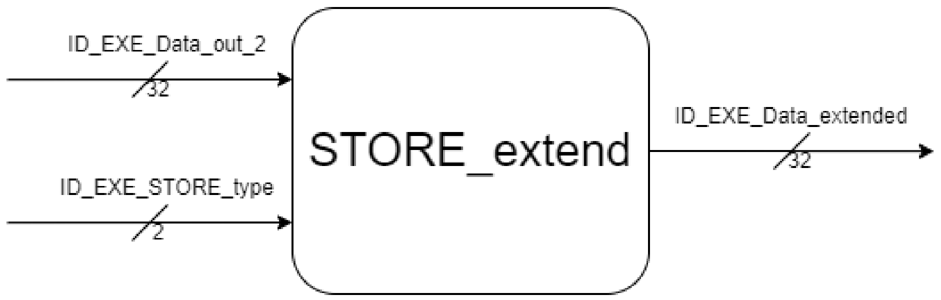


Figure 13: STORE_extend overview

Format	Name	Assembly	Semantics	Description
I-type, I-immediate	LH	lh rd, imm(rs1)	R[rd] = M_4B[R[rs1] + sext(imm)]	Load 16 bits (2 bytes) and sign-extends to 32-bit
I-type, I-immediate	LB	lb rd, imm(rs1)	R[rd] = M_4B[R[rs1] + sext(imm)]	Load 8 bits (1 bytes) and sign-extends to 32-bit
I-type, I-immediate	LHU	lhu rd, imm(rs1)	R[rd] = M_4B[R[rs1] + sext(imm)]	Load 16 bits (2 bytes) and zero-extends to 32-bit
I-type, I-immediate	LBU	lbu rd, imm(rs1)	R[rd] = M_4B[R[rs1] + sext(imm)]	Load 8 bits (1 bytes) and sign-extends to 32-bit
S-type	SH	sh rs2, imm(rs1)	M_4B[R[rs1] + sext(imm)] = R[rs2]	Store 16 bits (2 bytes) and sign-extends to 32-bit
S-type	SB	sb rs2, imm(rs1)	M_4B[R[rs1] + sext(imm)] = R[rs2]	Store 8 bits (1 bytes) and sign-extends to 32-bit

Table 2: Load and Store instruction added for optimized processor

3. Testing

To ensure the optimized processor operate as expect, several programs are used to test and ensure the functionality of the optimized processor. And the simulation results are used to show the operation of the optimized processor and each module are operate as expect.

3.1 General test

The original processor already included several instructions. To check the instruction is running properly the processor was tested with the instruction test from Appendix A and check the instruction is running as the file state from [3]. As a result, the instruction SRAI was found which is not following the instruction from [3]. The following figure is the simulation following Appendix A instruction. The instruction 41015713 is the instruction SRAI x14, x2, 16 and the x2 register is the value of -1 (0xFFFFFFFF). For SRAI instruction the value of the register should be taken as the value from MSB and copy the result to shift the data. However, in this situation the result shows the value was not copied and replaced with zero (result 0x0000FFFF). This will be fixed in the future.

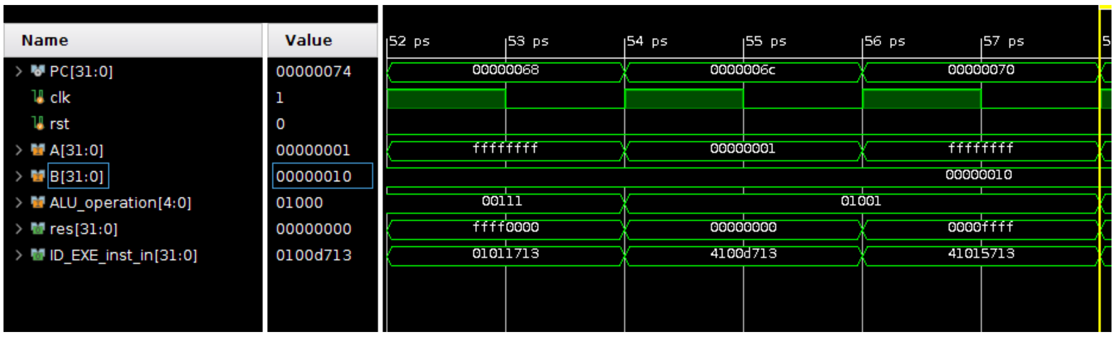


Figure 14: Simulation with the Appendix A instruction

3.2 Data forwarding

The figure is showing the original design of the processor will store while doing the instruction flow shown in Appendix B, as the figure shows when the program counter = 0x64 the program will stall for 3 clock cycles to prevent the data hazard and get the correct result. On the other word, the stall will decrease the CPI of the processor, to improve this situation the data forwarding is needed.

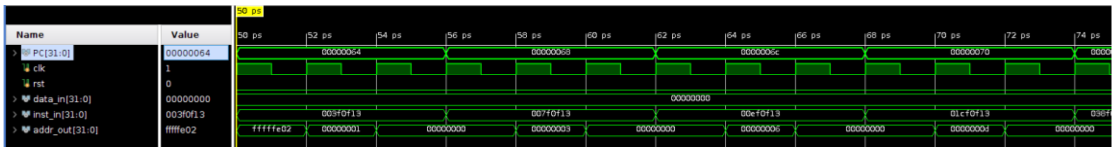


Figure 15: Simulation from the original design (using Appendix B program)

The Figure 16 shows after the data forwarding and removed the data hazard module, the processor can get a correct answer and do the instruction in 1 cycle clock. Therefore, it shows the data forward module is working properly.

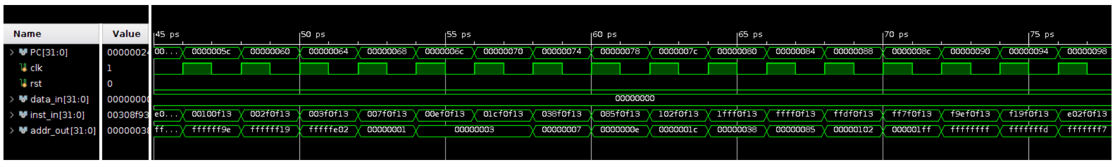


Figure 16: Simulation after adding the data forwarding design (using Appendix B program)

3.3 BHR_and_PHT

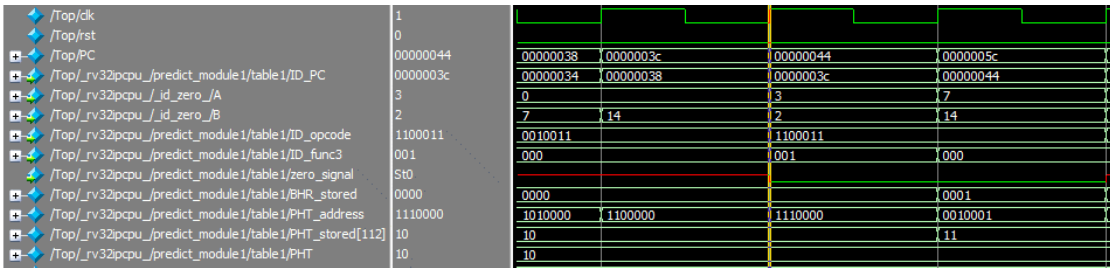

/Top/clk
/Top/rst
/Top/PC
/Top/_rv32ipcpu_/predict_module1/table1/ID_PC
/Top/_rv32ipcpu_/_id_zero_/A
/Top/_rv32ipcpu_/_id_zero_/B
/Top/_rv32ipcpu_/predict_module1/table1/ID_opcode
/Top/_rv32ipcpu_/predict_module1/table1/ID_func3
/Top/_rv32ipcpu_/predict_module1/table1/zero_signal
/Top/_rv32ipcpu_/predict_module1/table1/BHR_stored
/Top/_rv32ipcpu_/predict_module1/table1/PHT_address
/Top/_rv32ipcpu_/predict_module1/table1/PHT_stored[112]
/Top/_rv32ipcpu_/predict_module1/table1/PHT

Figure 17: Simulation result when prediction is correct (using Appendix C program)

The waveform above shows an example for prediction is correct. And the assembly code for this program is provided in Appendix C. According to Appendix C, a branch not equal (BNE) instruction is read when PC is 0x3C. From the figure above, it also shows that the branch is predicted as taken because the PHT is “10” when PC is 0x3C, and PC jumps to 0x44 in next clock cycle. Then, the actual branch result can be observed by “zero_signal”, when “zero_signal” is a logic 0, it means the two values which are read from the register in ID stage are not equal and vice versa. And “zero_signal” from the waveform of Figure 17 is logic 0, so the branch should be taken for instruction at PC 0x3C. So, the prediction for instruction when PC is 0x3C is correct. Furthermore, after the actual branch result is known, BHR and content of PHT has been updated from “0000” and “10” to “0001” and “11” at PHT address 112 respectively. As a result, the “BHR_and_PHT” module operate as expected, and the program is executed without recover the PC value.

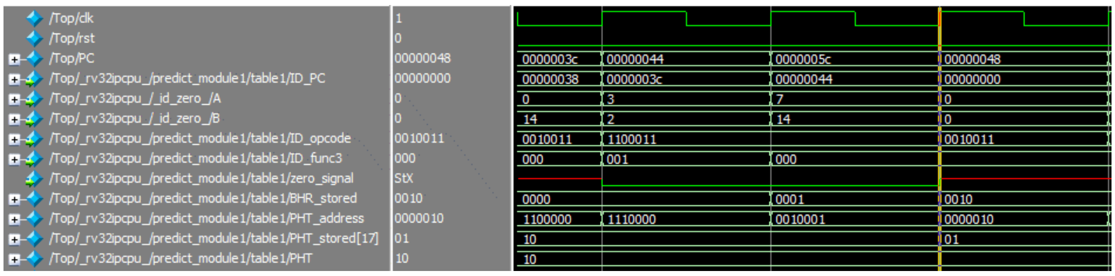

/Top/clk
/Top/rst
/Top/PC
/Top/_rv32ipcpu_/predict_module1/table1/ID_PC
/Top/_rv32ipcpu_/_id_zero_/A
/Top/_rv32ipcpu_/_id_zero_/B
/Top/_rv32ipcpu_/predict_module1/table1/ID_opcode
/Top/_rv32ipcpu_/predict_module1/table1/ID_func3
/Top/_rv32ipcpu_/predict_module1/table1/zero_signal
/Top/_rv32ipcpu_/predict_module1/table1/BHR_stored
/Top/_rv32ipcpu_/predict_module1/table1/PHT_address
/Top/_rv32ipcpu_/predict_module1/table1/PHT_stored[17]
/Top/_rv32ipcpu_/predict_module1/table1/PHT

Figure 18: Simulation result when prediction is wrong (using Appendix C program)

The waveform above shows an example for prediction is wrong. According to Appendix C, a branch equal (BEQ) instruction is read when PC is 0x44. From the figure above, it also shows that the branch is predicted as taken because the PHT is “10” when PC is 0x44, and PC jumps to 0x5C in next clock cycle. After that, “zero_signal” is logic 0, so the branch should not be taken for instruction at PC 0x44. Then, the prediction for instruction when PC is 0x44 is not correct. It means the program should be executed in sequence after PC is 0x44. Then, the value of PC recovers back to 0x44+4 after PC is 0x5C. Also, after the actual branch result is known, BHR and content of PHT has been updated from “0001” and “10” to “0010” and “01” at PHT address 17 respectively. Consequently, the “BHR_and_PHT” module operates as expected.

3.4 IF_MUX_sel

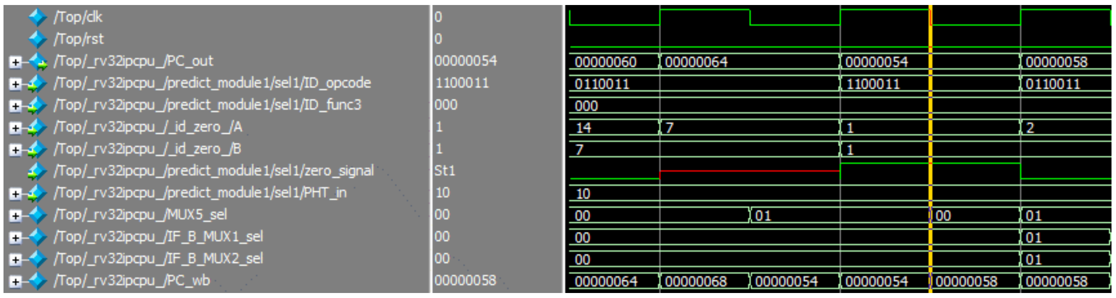


Figure 19: Simulation result for MUX select signal when prediction is correct

(using Appendix C program)

The waveform above is the similar scenario as the Figure 17 shows. In other words, the branch is predicted as taken and the prediction is correct. When the branch is predict as taken, the PC value for next clock cycle is “PC_out” + “Imm_IF”, it means “IF_B_MUX1_sel” and “IF_B_MUX2_sel” are both 0 as discussed in section 2.2.2. Also, from section 2.2.2, the select signal “MUX5_sel” should be 1 because the calculated PC value for PC module to update is from “add_branch_out” for branch instruction. Then, according to section 2.2.2, the select signal “MUX5_sel” is changed to 0 for correct prediction and not recover the PC value in the next clock cycle. From Figure 19, it shows the “MUX5_sel” is 1, “IF_B_MUX1_sel” and “IF_B_MUX2_sel” are both 0 when predicting the branch result. And it shows the “MUX5_sel” is 0 and the “PC_wb” value is PC_out+4 when actual branch result is known. So, the waveform shows that the “IF_MUX_sel” operates as expected.

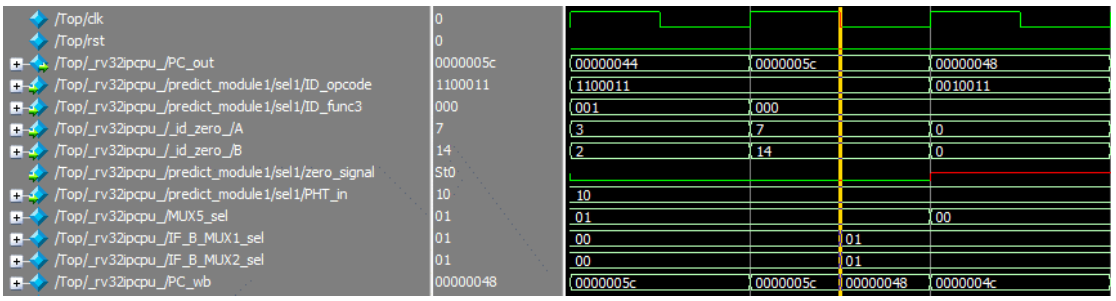

/Top/clk 0
/Top/rst 0
/Top/_rv32ipcpu_/PC_out 0000005c
/Top/_rv32ipcpu_/predict_module1/sel1/ID_opcode 1100011
/Top/_rv32ipcpu_/predict_module1/sel1/ID_func3 000
/Top/_rv32ipcpu_/_id_zero_/A 7
/Top/_rv32ipcpu_/_id_zero_/B 14
/Top/_rv32ipcpu_/predict_module1/sel1/zero_signal St0
/Top/_rv32ipcpu_/predict_module1/sel1/PHT_in 10
/Top/_rv32ipcpu_/MUX5_sel 01
/Top/_rv32ipcpu_/IF_B_MUX1_sel 01
/Top/_rv32ipcpu_/IF_B_MUX2_sel 01
/Top/_rv32ipcpu_/PC_wb 00000048
00000044
0000005c
00000048
1100011
0010011
001
000
3
7
0
2
14
0
10
01
00
00
01
00
01
0000005c
0000005c
00000048
0000004c

Figure 20: Simulation result for MUX select signal when prediction is wrong

(using Appendix C program)

The waveform above is the same scenario as the Figure 18 shows. In other words, the branch is predicted as taken and the prediction is not correct. Then, according to section 2.2.2, the select signal for both "IF_B_MUX1_sel" and "IF_B_MUX2_sel" would be 0 because the PC value for next clock cycle is "PC_out" + "Imm_IF", and "MUX5_sel" would be 1 because the calculated PC value for PC module to update is from "add_branch_out" for branch instruction. From Figure 20, PC is jump to a calculated value after PC is 0x44. After PC jumps to 0x5C, the actual branch result indicates that the prediction is wrong. Next, PC recovery is needed and recovered back to 0x44+4. And now, the select signal for both "IF_B_MUX1_sel" and "IF_B_MUX2_sel" would be 1 and "MUX5_sel" would be 1 when follow the discussion in section 2.2.2. Then, according to the waveform above, it shows that the "IF_MUX_sel" operates as expected since PC successfully jumps to 0x5C for prediction and recovered back to 0x48 after actual branch result is known.

3.5 branch_load_check

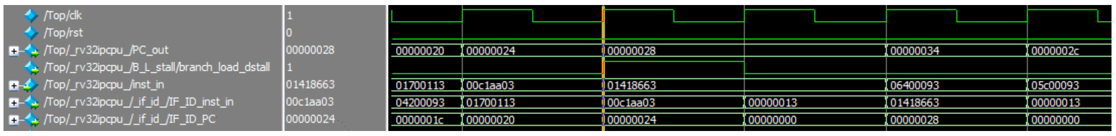

/Top/clk 1
/Top/rst 0
/Top/_rv32ipcpu_/PC_out 00000028
/Top/_rv32ipcpu_/B_L_stall/branch_load_dstall 1
/Top/_rv32ipcpu_/inst_in 01418663
/Top/_rv32ipcpu_/_if_id_/IF_ID_inst_in 00c1aa03
/Top/_rv32ipcpu_/_if_id_/IF_ID_PC 00000024
00000020
00000024
00000028
00000034
0000002c
01700113
00c1aa03
01418663
06400093
05c00093
04200093
01700113
00c1aa03
00000013
01418663
00000013
0000001c
00000020
00000024
00000000
00000028
00000000

Figure 21: Simulation result for "branch_load_check" module test (using Appendix D program)

The waveform above shows an example to prove the "branch_load_check" module operate as expect. According to Figure 21 and Appendix D, it shows that when load instruction is followed with a branch instruction and have dependency, "branch_load_dstall" signal is set to logic 1. After 1 clock cycle, IF/ID register detected the "branch_load_dstall" signal is logic 1, it passes a NOP instruction to the ID stage and PC value in ID is set to 0. After that, the processor executes the branch instruction after the "branch_load_dstall" signal is logic 0, and the branch predicter predict the branch is taken. Then, the PC value jumps to 0x34 but it recovers in the next clock cycle because prediction is wrong. So, the "branch_load_check" module operates as expected.

3.6 jump instruction test

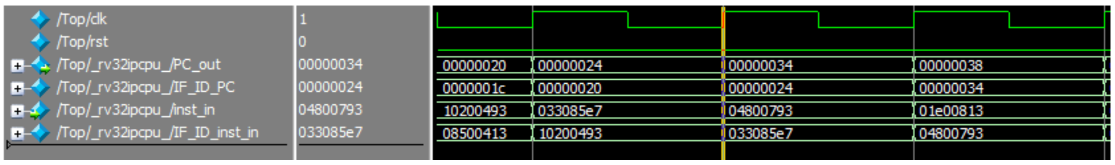

/Top/clk
/Top/rst
/Top/_rv32ipcpu_/PC_out
/Top/_rv32ipcpu_/IF_ID_PC
/Top/_rv32ipcpu_/inst_in
/Top/_rv32ipcpu_/IF_ID_inst_in
1
0
00000034
00000024
04800793
033085e7
00000020 00000024 00000034 00000038
0000001c 00000020 00000024 00000034
10200493 033085e7 04800793 01e00813
08500413 10200493 033085e7 04800793

Figure 22: Simulation result for JALR (using Appendix E program)

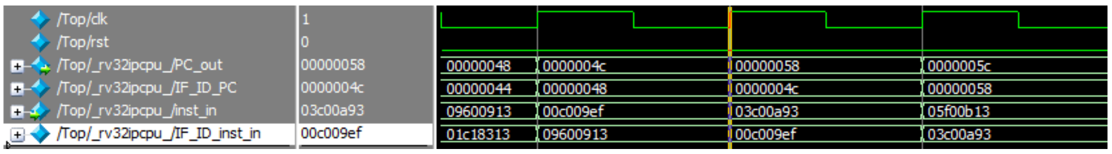

/Top/clk
/Top/rst
/Top/_rv32ipcpu_/PC_out
/Top/_rv32ipcpu_/IF_ID_PC
/Top/_rv32ipcpu_/inst_in
/Top/_rv32ipcpu_/IF_ID_inst_in
1
0
00000058
0000004c
03c00a93
00c009ef
00000048 0000004c 00000058 0000005c
00000044 00000048 0000004c 00000058
09600913 00c009ef 03c00a93 05f00b13
01c18313 09600913 00c009ef 03c00a93

Figure 23: Simulation result for JAL (using Appendix E program)

The waveforms above show the execution result of a program, and this program included both jump and branch instruction. According to Appendix E, it shows that the branch instruction should not be execute even the branch is taken. And Figure 22 clearly shows that when JALR instruction is executed, the next instruction, which is a branch instruction, is not fetched from the instruction memory and directly jump to the PC value of 0x34. As well as this, similar situation is applied to JAL instruction and Figure 23 proved that it jumps to the correct PC value, which is 0x58. To concluded, the design in section 2.3 executes the testing program without any error.

3.7 Load_extend and Store_extend

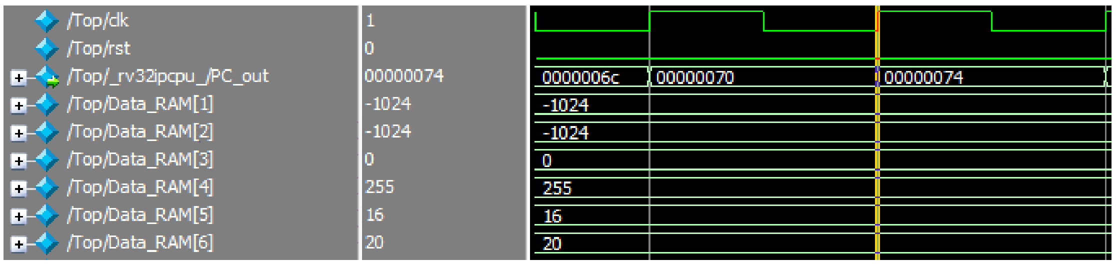


Figure 24: Data stored in RAM after program execution (Appendix F)

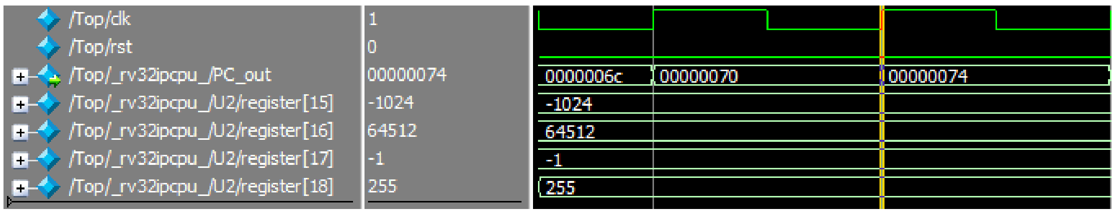

/Top/clk
/Top/rst
/Top/_rv32ipcpu_/PC_out
/Top/_rv32ipcpu_/U2/register[15]
/Top/_rv32ipcpu_/U2/register[16]
/Top/_rv32ipcpu_/U2/register[17]
/Top/_rv32ipcpu_/U2/register[18]
1
0
00000074
-1024
64512
-1
255
0000006c 00000070 00000074
-1024
64512
-1
255

Figure 25: Data stored in register after program execution (Appendix F)

The waveforms above show the data stored in RAM and register after the program finish execution, and the program code is shown in Appendix F. According to Appendix F, the program expects the RAM will store -1024, -1024, 0, 255, 16, 20 into the RAM with address 4, 8, 12, 16, 20, 24 respectively. From Figure 24, it clearly shows that the correct value is stored in the specific location of the RAM. Moreover, the program expects the register will store -1024, 64512, -1, 255 into register x15, x16, x17, x18 respectively. And Figure 25 proved that the correct value is stored in the specific location of the register. Which means the “LOAD_extend” and “STORE_extend” module operates as expected.

4. Evaluation

Test program type	original processor execution time	optimized processor execution time	Speeded up (optimized vs original)
data dependency + branch + jump (Appendix G)	1110 clock cycle	609 clock cycle	1.823 times faster than original processor
branch + jump (Appendix H)	913 clock cycle	814 clock cycle	1.122 times faster than original processor

Table 3: Execution time comparison between original and optimized processor

The table above shows the execution time for programs in Appendix G and H. Both programs are run a simple for loop with 100 iterations. The difference is the program shows in Appendix G have data dependency while Appendix H did not.

From Table 3, it shows that the optimized processor improved the performance significantly (1.823 times). The reason is the number of stalls for data hazard and control hazard is reduced by using the method mentioned in session 2. As well as the Appendix H program, the optimized processor is 1.122 times faster than original processor when the program does not have any data dependency. The branch instruction for Appendix H program normally is not taken. Therefore, the optimized processor is only speeded up for 1.122 times. In theory, if the branch for the Appendix H program is always taken, the optimized can speed up to around 1.2 times compare to the original processor, because the original processor will stall one clock cycle on each loop, which hundred stall in total.

5. Conclusion

In this project, the optimized processor enhanced the performance successfully by using various technique and components. And the execution time is reduced significantly when the number of stalls data hazard is reduced.

6. Statement of Work

Kei Hong Chan (50%):

- implement, design and test data forwarding part
- original processor testing
- eliminate jump stall
- report edit
- presentation slides

Yiu Fai Lam (50%):

- implement, design and test branch prediction part
- eliminate jump stall
- add and test other types of load and store instruction
- report edit
- presentation slides

7. Reference

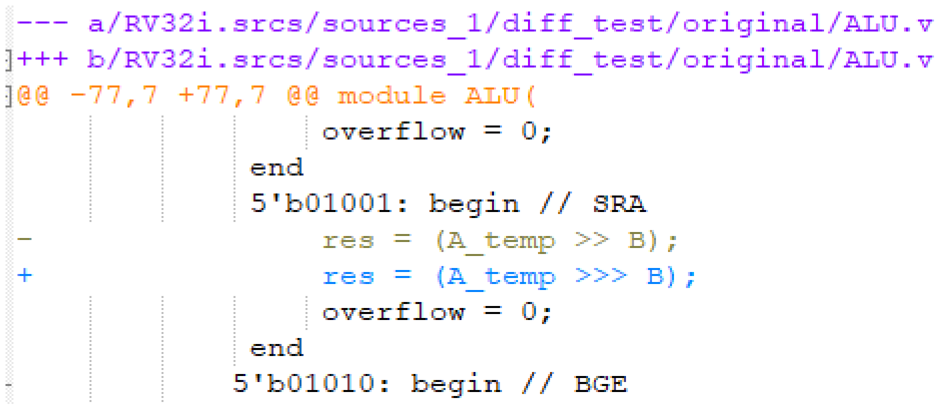
[1] "RipperJ/RISC-V_CPU", GitHub. [Online]. Available:
https://github.com/RipperJ/RISC-V_CPU
[Accessed: 02- Apr- 2021].

[2] ELEC 5140 Advanced Computer Architecture Project, The Hong Kong University of Science and Technology, 2021. [Online]. Available:
https://canvas.ust.hk/courses/36420/files/folder/unfiled?preview=4439071
[Accessed: 02- Apr- 2021].

[3] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, RISC-V Foundation, 2019, p. 148. [Online]. Available:
https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf
[Accessed: 02- Apr- 2021].

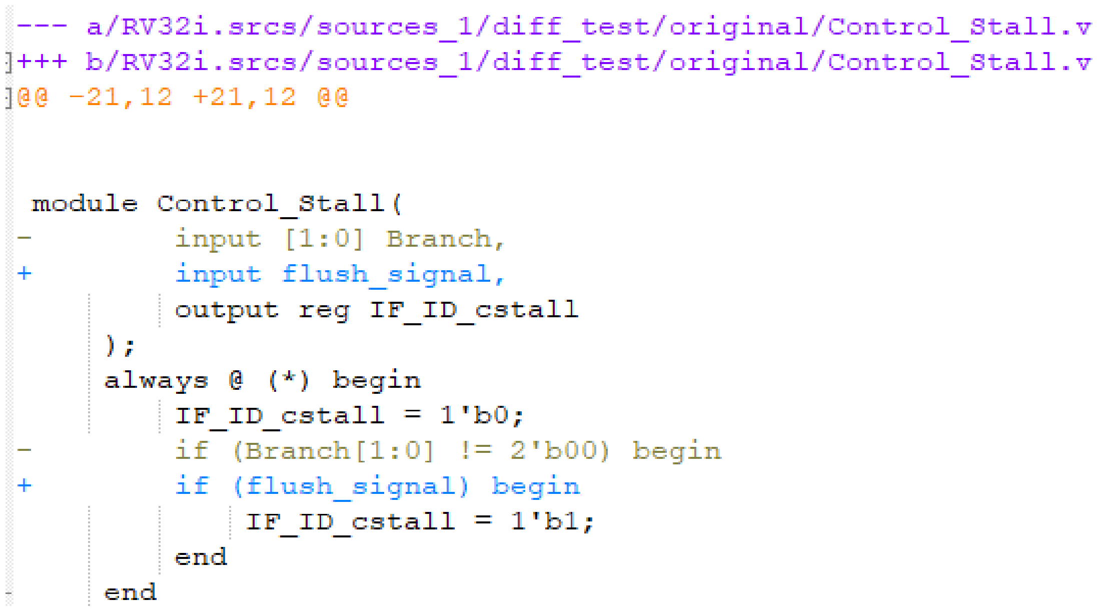
8. Changes in the source code

ALU.v:



```
--- a/RV32i.srcs/sources_1/diff_test/original/ALU.v
+++ b/RV32i.srcs/sources_1/diff_test/original/ALU.v
@@ -77,7 +77,7 @@ module ALU(
                    overflow = 0;
                end
                5'b01001: begin // SRA
-                   res = (A_temp >> B);
+                   res = (A_temp >>> B);
                    overflow = 0;
                end
               5'b01010: begin // BGE
```

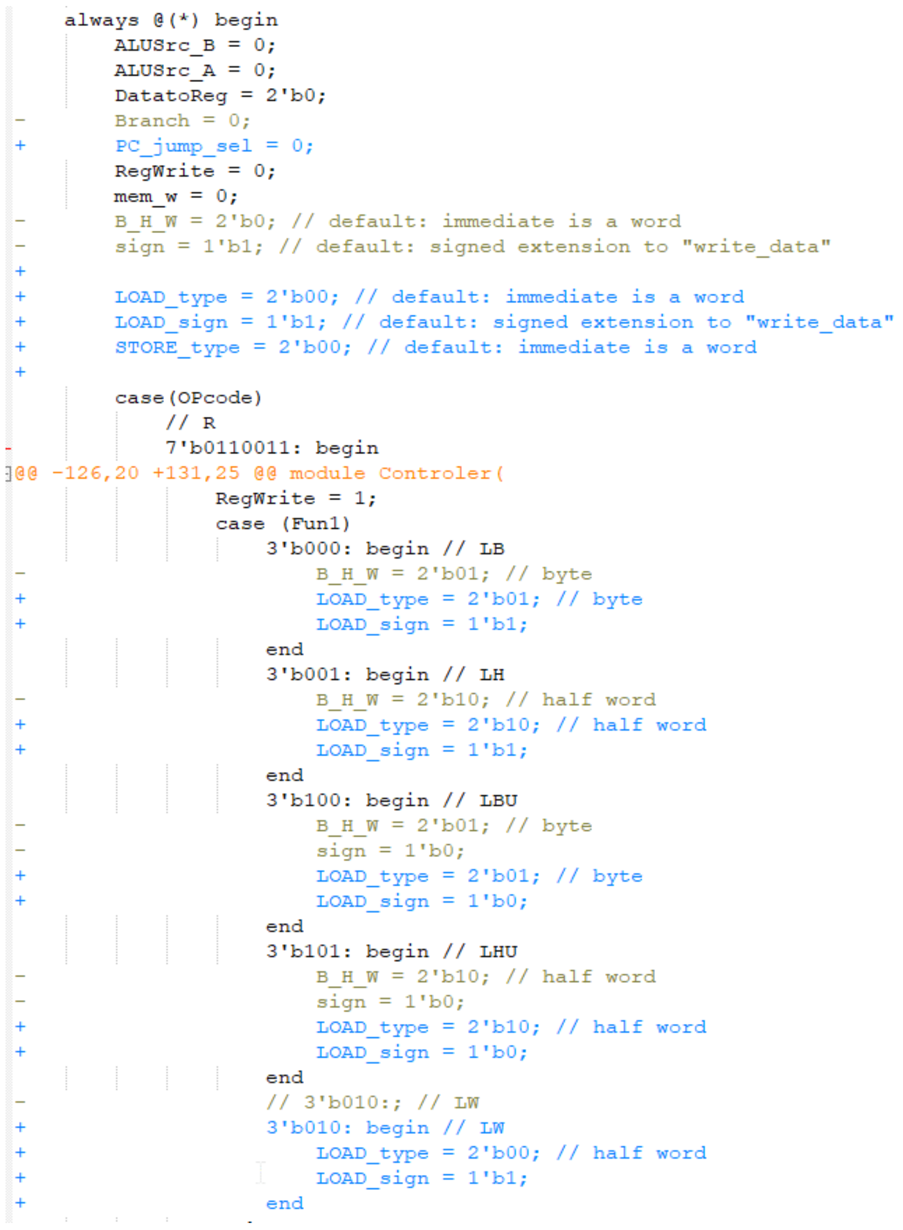
Control_Stall.v



```
--- a/RV32i.srcs/sources_1/diff_test/original/Control_Stall.v
+++ b/RV32i.srcs/sources_1/diff_test/original/Control_Stall.v
@@ -21,12 +21,12 @@


 module Control_Stall(
-        input [1:0] Branch,
+        input flush_signal,
        output reg IF_ID_cstall
    );
    always @ (*) begin
        IF_ID_cstall = 1'b0;
-        if (Branch[1:0] != 2'b00) begin
+        if (flush_signal) begin
            IF_ID_cstall = 1'b1;
        end
    end
```

Controler.v:

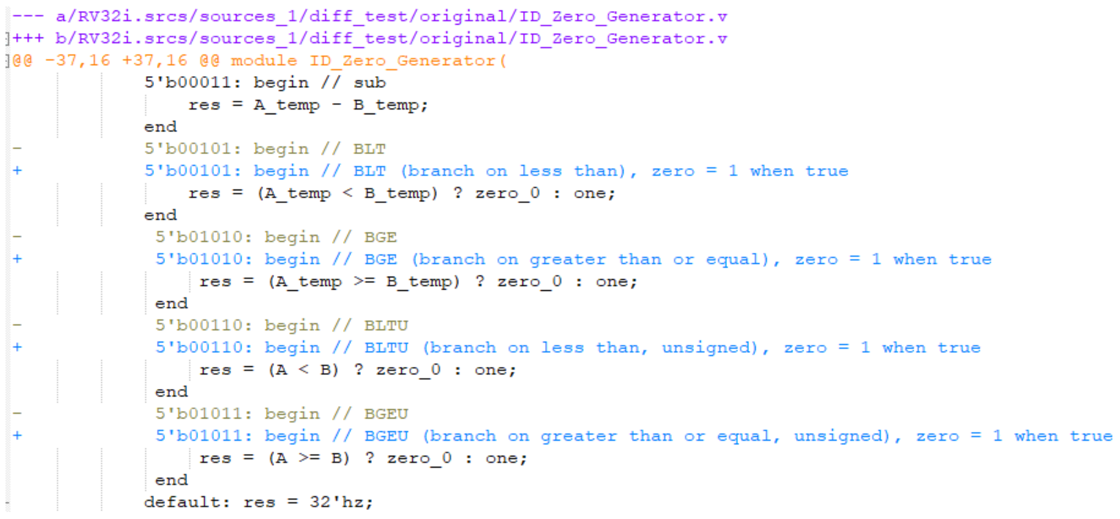
```
--- a/RV32i.srcs/sources_1/diff_test/original/Controler.v
+++ b/RV32i.srcs/sources_1/diff_test/original/Controler.v
@@ -28,22 +28,27 @@ module Controler(
         output reg ALUSrc_A,
         output reg [1:0] ALUSrc_B,
         output reg [1:0] DatatoReg,
-        output reg [1:0] Branch,
+        output reg [1:0] PC_jump_sel,
         output reg RegWrite,
         output reg mem_w,
         output reg [4:0] ALU_Control,
-        output reg [1:0] B_H_W,
-        output reg sign
+
+        output reg [1:0] LOAD_type,
+        output reg LOAD_sign,
+        output reg [1:0] STORE_type
         );
```



```
     always @(*) begin
         ALUSrc_B = 0;
         ALUSrc_A = 0;
         DatatoReg = 2'b0;
-        Branch = 0;
+        PC_jump_sel = 0;
         RegWrite = 0;
         mem_w = 0;
-        B_H_W = 2'b0; // default: immediate is a word
-        sign = 1'b1; // default: signed extension to "write_data"
+
+        LOAD_type = 2'b00; // default: immediate is a word
+        LOAD_sign = 1'b1; // default: signed extension to "write_data"
+        STORE_type = 2'b00; // default: immediate is a word
+
         case(OPcode)
             // R
-            7'b0110011: begin
]@@ -126,20 +131,25 @@ module Controler(
                 RegWrite = 1;
                 case (Fun1)
                     3'b000: begin // LB
-                        B_H_W = 2'b01; // byte
+                        LOAD_type = 2'b01; // byte
+                        LOAD_sign = 1'b1;
                     end
                     3'b001: begin // LH
-                        B_H_W = 2'b10; // half word
+                        LOAD_type = 2'b10; // half word
+                        LOAD_sign = 1'b1;
                     end
                     3'b100: begin // LBU
-                        B_H_W = 2'b01; // byte
-                        sign = 1'b0;
+                        LOAD_type = 2'b01; // byte
+                        LOAD_sign = 1'b0;
                     end
                     3'b101: begin // LHU
-                        B_H_W = 2'b10; // half word
-                        sign = 1'b0;
+                        LOAD_type = 2'b10; // half word
+                        LOAD_sign = 1'b0;
                     end
-                    // 3'b010:; // LW
+                    3'b010: begin // LW
+                        LOAD_type = 2'b00; // half word
+                        LOAD_sign = 1'b1;
+                    end
```

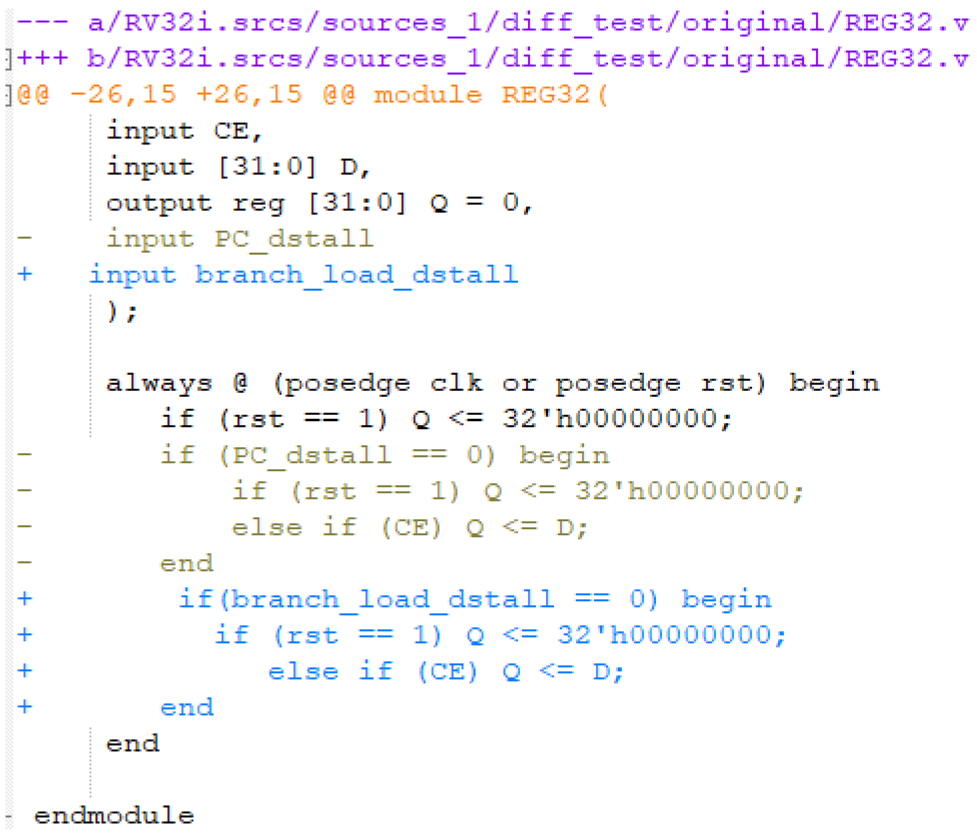
```
                    endcase
                end
                7'b0100011: begin   // S
@@ -148,49 +158,45 @@ module Controler(
                    mem_w = 1;
                    case (Fun1)
                        3'b000: begin
-                           B_H_W = 2'b01; // byte
+                           STORE_type = 2'b01; // byte
                        end
                        3'b001: begin
-                           B_H_W = 2'b10; // half word
+                           STORE_type = 2'b10; // half word
                        end
-                       // 3'b010: ; // SW
+                       3'b010: begin // SW
+                           STORE_type = 2'b00;
+                       end
                    endcase
                end
                7'b1100011: begin   // Branch
                    case (Fun1)
                        3'b000: begin // BEQ
                            ALU_Control = 5'b00011;
-                           Branch = {1'b0, zero};
                        end
                        3'b001: begin // BNE
                            ALU_Control = 5'b00011;
-                           Branch = {1'b0, ~zero};
                        end
                        3'b100: begin // BLT
                            ALU_Control = 5'b00101;
-                           Branch = {1'b0, zero};
                        end
                        3'b101: begin // BGE
                            ALU_Control = 5'b01010;
-                           Branch = {1'b0, zero};
                        end
                        3'b110: begin // BLTU
                            ALU_Control = 5'b00110;
-                           Branch = {1'b0, zero};
                        end
                        3'b111: begin // BGEU
                            ALU_Control = 5'b01011;
-                           Branch = {1'b0, zero};
                        end
                    endcase
                end
                7'b1101111: begin   // jal
-                   Branch = 2'b10;
+                   PC_jump_sel = 2'b01;
                    DatatoReg = 2'b11;
                    RegWrite = 1;
                end
                 7'b1100111: begin   // jalr
-                    Branch = 2'b11;
+                    PC_jump_sel = 2'b10;
                     DatatoReg = 2'b11;
                     RegWrite = 1;
                 end
```

ID_Zero_Generator.v:



```
--- a/RV32i.srcs/sources_1/diff_test/original/ID_Zero_Generator.v
+++ b/RV32i.srcs/sources_1/diff_test/original/ID_Zero_Generator.v
@@ -37,16 +37,16 @@ module ID_Zero_Generator(
            5'b00011: begin // sub
                res = A_temp - B_temp;
            end
-           5'b00101: begin // BLT
+           5'b00101: begin // BLT (branch on less than), zero = 1 when true
                res = (A_temp < B_temp) ? zero_0 : one;
            end
-            5'b01010: begin // BGE
+            5'b01010: begin // BGE (branch on greater than or equal), zero = 1 when true
                res = (A_temp >= B_temp) ? zero_0 : one;
            end
-            5'b00110: begin // BLTU
+            5'b00110: begin // BLTU (branch on less than, unsigned), zero = 1 when true
                res = (A < B) ? zero_0 : one;
            end
-            5'b01011: begin // BGEU
+            5'b01011: begin // BGEU (branch on greater than or equal, unsigned), zero = 1 when true
                res = (A >= B) ? zero_0 : one;
            end
            default: res = 32'hz;
```

REG32.v:

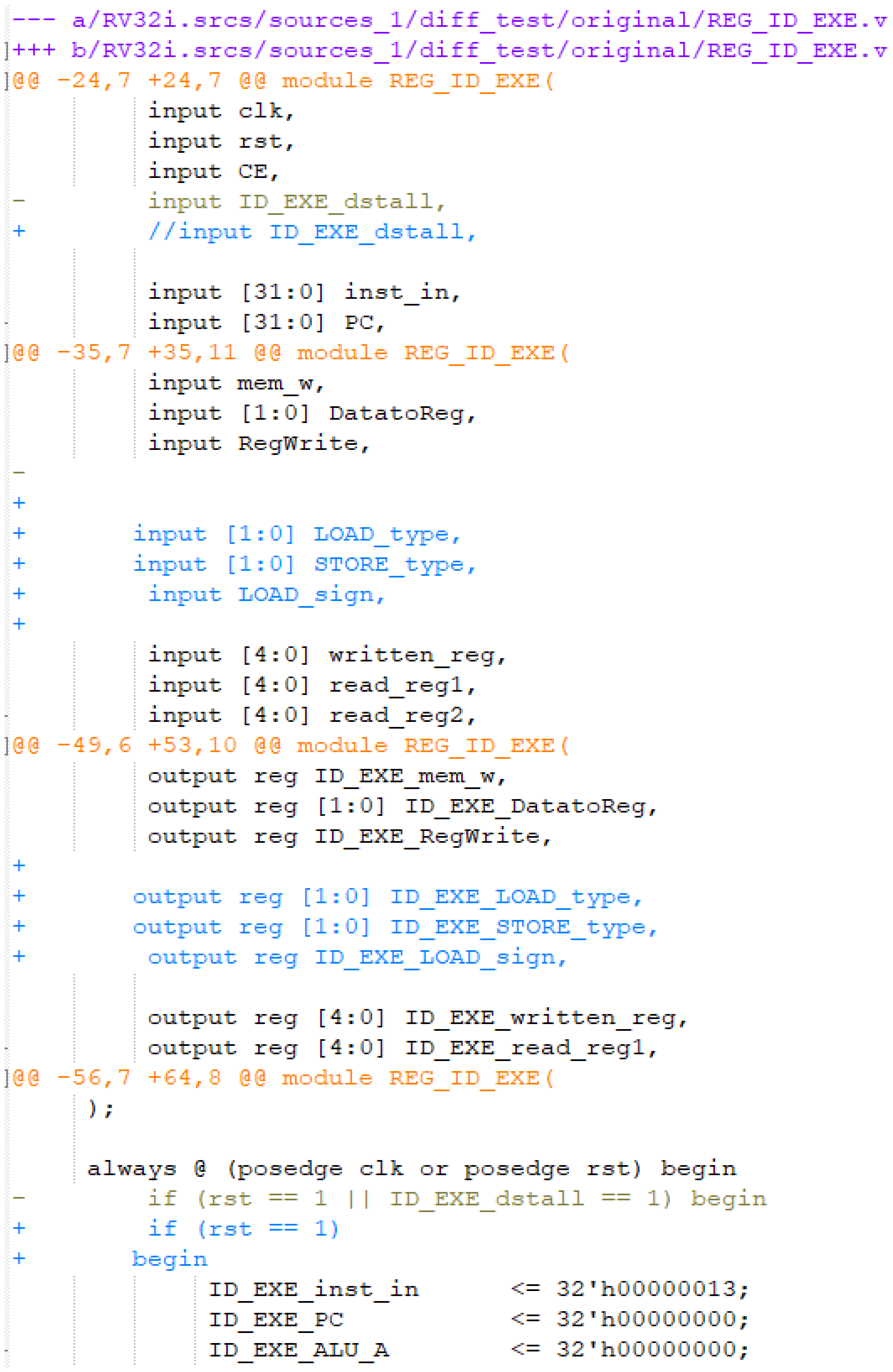


```
--- a/RV32i.srcs/sources_1/diff_test/original/REG32.v
+++ b/RV32i.srcs/sources_1/diff_test/original/REG32.v
@@ -26,15 +26,15 @@ module REG32(
     input CE,
     input [31:0] D,
     output reg [31:0] Q = 0,
-    input PC_dstall
+   input branch_load_dstall
     );

     always @ (posedge clk or posedge rst) begin
        if (rst == 1) Q <= 32'h00000000;
-       if (PC_dstall == 0) begin
-           if (rst == 1) Q <= 32'h00000000;
-           else if (CE) Q <= D;
-       end
+        if(branch_load_dstall == 0) begin
+          if (rst == 1) Q <= 32'h00000000;
+             else if (CE) Q <= D;
+       end
     end

 endmodule
```

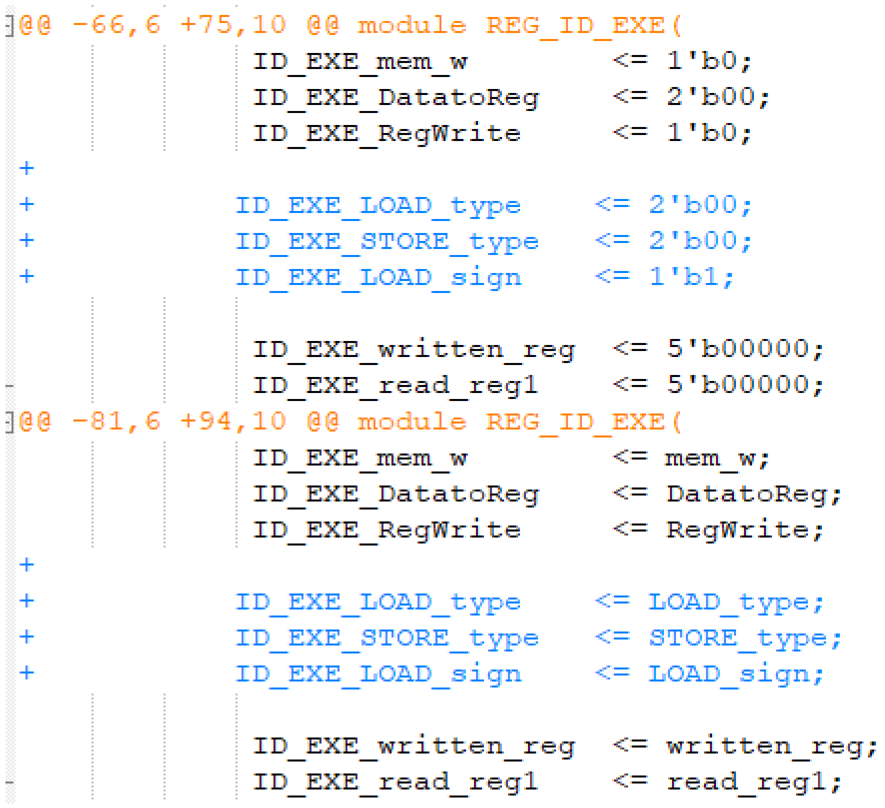
REG_EXE_MEM.v:

```
--- a/RV32i.srcs/sources_1/diff_test/original/REG_EXE_MEM.v
+++ b/RV32i.srcs/sources_1/diff_test/original/REG_EXE_MEM.v
@@ -32,6 +32,10 @@ module REG_EXE_MEM(
         input mem_w,
         input [1:0] DatatoReg,
         input RegWrite,
+
+        input [1:0] ID_EXE_LOAD_type,
+        input ID_EXE_LOAD_sign,
+
         input [4:0] written_reg,
         input [4:0] read_reg1,
         input [4:0] read_reg2,
@@ -44,6 +48,10 @@ module REG_EXE_MEM(
         output reg EXE_MEM_mem_w,
         output reg [1:0] EXE_MEM_DatatoReg,
         output reg EXE_MEM_RegWrite,
+
+        output reg [1:0] EXE_MEM_LOAD_type,
+        output reg EXE_MEM_LOAD_sign,
+
         output reg [4:0] EXE_MEM_written_reg,
         output reg [4:0] EXE_MEM_read_reg1,
         output reg [4:0] EXE_MEM_read_reg2
@@ -57,6 +65,10 @@ module REG_EXE_MEM(
             EXE_MEM_mem_w       <= 1'b0;
             EXE_MEM_DatatoReg   <= 2'b00;
             EXE_MEM_RegWrite    <= 1'b0;
+
+            EXE_MEM_LOAD_type   <= 2'b00;
+            EXE_MEM_LOAD_sign   <= 1'b1;
+
             EXE_MEM_written_reg <= 5'b00000;
             EXE_MEM_read_reg1   <= 5'b00000;
             EXE_MEM_read_reg2   <= 5'b00000;
@@ -69,6 +81,10 @@ module REG_EXE_MEM(
             EXE_MEM_mem_w       <= mem_w;
             EXE_MEM_DatatoReg   <= DatatoReg;
             EXE_MEM_RegWrite    <= RegWrite;
+
+            EXE_MEM_LOAD_type   <= ID_EXE_LOAD_type;
+            EXE_MEM_LOAD_sign   <= ID_EXE_LOAD_sign;
+
             EXE_MEM_written_reg <= written_reg;
             EXE_MEM_read_reg1   <= read_reg1;
             EXE_MEM_read_reg2   <= read_reg2;
```

REG_ID_EXE.v:

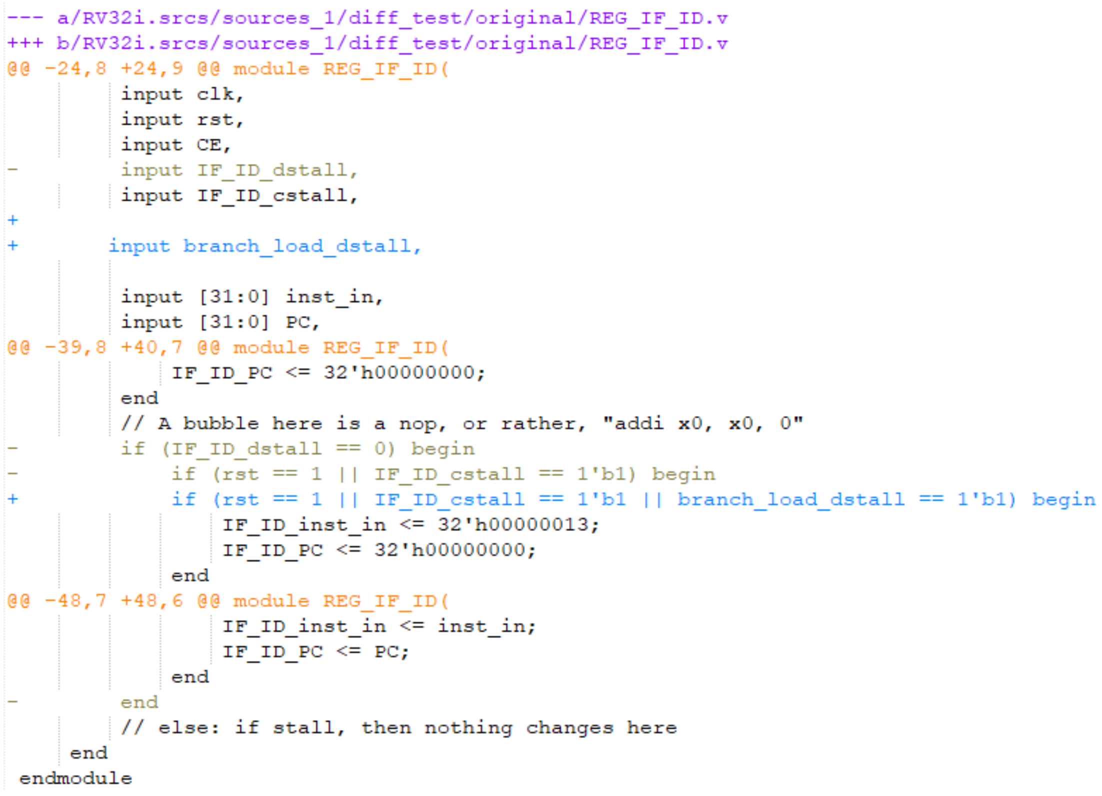


```
--- a/RV32i.srcs/sources_1/diff_test/original/REG_ID_EXE.v
+++ b/RV32i.srcs/sources_1/diff_test/original/REG_ID_EXE.v
@@ -24,7 +24,7 @@ module REG_ID_EXE(
        input clk,
        input rst,
        input CE,
-       input ID_EXE_dstall,
+       //input ID_EXE_dstall,

        input [31:0] inst_in,
        input [31:0] PC,
@@ -35,7 +35,11 @@ module REG_ID_EXE(
        input mem_w,
        input [1:0] DatatoReg,
        input RegWrite,
-
+
+      input [1:0] LOAD_type,
+      input [1:0] STORE_type,
+       input LOAD_sign,
+
        input [4:0] written_reg,
        input [4:0] read_reg1,
        input [4:0] read_reg2,
@@ -49,6 +53,10 @@ module REG_ID_EXE(
        output reg ID_EXE_mem_w,
        output reg [1:0] ID_EXE_DatatoReg,
        output reg ID_EXE_RegWrite,
+
+      output reg [1:0] ID_EXE_LOAD_type,
+      output reg [1:0] ID_EXE_STORE_type,
+       output reg ID_EXE_LOAD_sign,

        output reg [4:0] ID_EXE_written_reg,
        output reg [4:0] ID_EXE_read_reg1,
@@ -56,7 +64,8 @@ module REG_ID_EXE(
    );

    always @ (posedge clk or posedge rst) begin
-       if (rst == 1 || ID_EXE_dstall == 1) begin
+       if (rst == 1)
+      begin
            ID_EXE_inst_in       <= 32'h00000013;
            ID_EXE_PC            <= 32'h00000000;
            ID_EXE_ALU_A         <= 32'h00000000;
```



```
@@ -66,6 +75,10 @@ module REG_ID_EXE(
              ID_EXE_mem_w          <= 1'b0;
              ID_EXE_DatatoReg      <= 2'b00;
              ID_EXE_RegWrite       <= 1'b0;
+
+            ID_EXE_LOAD_type      <= 2'b00;
+            ID_EXE_STORE_type     <= 2'b00;
+            ID_EXE_LOAD_sign      <= 1'b1;

              ID_EXE_written_reg    <= 5'b00000;
              ID_EXE_read_reg1      <= 5'b00000;
@@ -81,6 +94,10 @@ module REG_ID_EXE(
              ID_EXE_mem_w          <= mem_w;
              ID_EXE_DatatoReg      <= DatatoReg;
              ID_EXE_RegWrite       <= RegWrite;
+
+            ID_EXE_LOAD_type      <= LOAD_type;
+            ID_EXE_STORE_type     <= STORE_type;
+            ID_EXE_LOAD_sign      <= LOAD_sign;

              ID_EXE_written_reg    <= written_reg;
              ID_EXE_read_reg1      <= read_reg1;
```

REG_IF_ID.v:

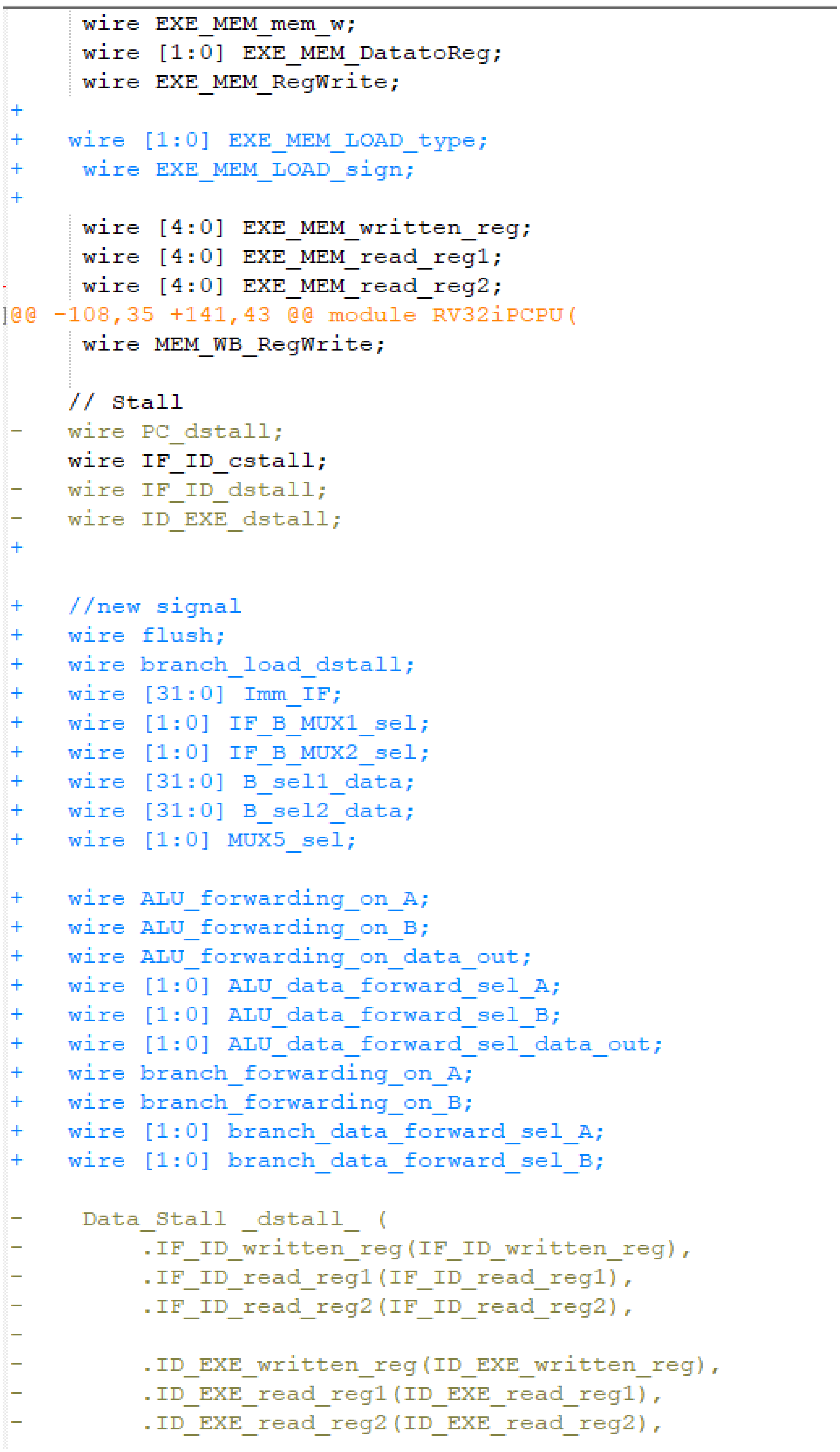


```
--- a/RV32i.srcs/sources_1/diff_test/original/REG_IF_ID.v
+++ b/RV32i.srcs/sources_1/diff_test/original/REG_IF_ID.v
@@ -24,8 +24,9 @@ module REG_IF_ID(
        input clk,
        input rst,
        input CE,
-       input IF_ID_dstall,
        input IF_ID_cstall,
+
+      input branch_load_dstall,

        input [31:0] inst_in,
        input [31:0] PC,
@@ -39,8 +40,7 @@ module REG_IF_ID(
            IF_ID_PC <= 32'h00000000;
        end
        // A bubble here is a nop, or rather, "addi x0, x0, 0"
-       if (IF_ID_dstall == 0) begin
-           if (rst == 1 || IF_ID_cstall == 1'b1) begin
+           if (rst == 1 || IF_ID_cstall == 1'b1 || branch_load_dstall == 1'b1) begin
                IF_ID_inst_in <= 32'h00000013;
                IF_ID_PC <= 32'h00000000;
            end
@@ -48,7 +48,6 @@ module REG_IF_ID(
                IF_ID_inst_in <= inst_in;
                IF_ID_PC <= PC;
            end
-       end
        // else: if stall, then nothing changes here
    end
 endmodule
```

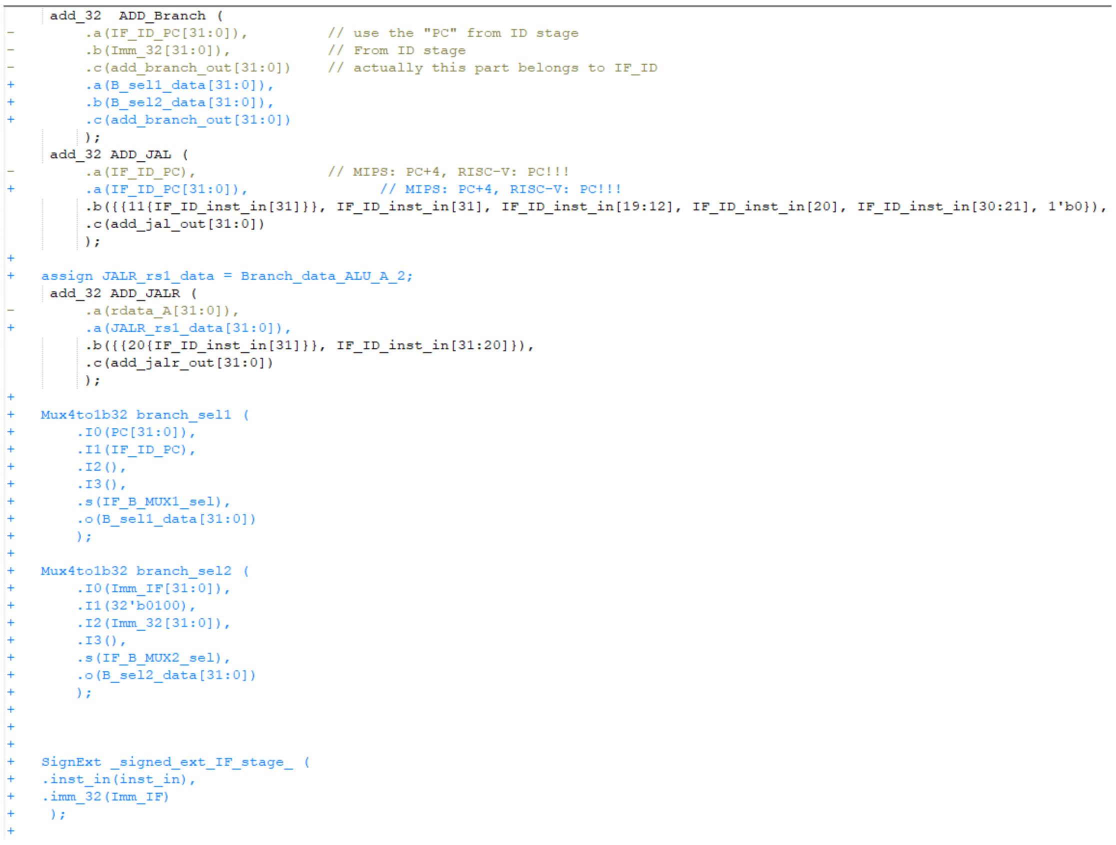
RV32iPCPU.v:

```diff
--- a/RV32i.srcs/sources_1/diff_test/original/RV32iPCPU.v
]+++ b/RV32i.srcs/sources_1/diff_test/original/RV32iPCPU.v
]@@ -23,16 +23,17 @@
 module RV32iPCPU(
     input clk,
     input rst,
-    input [31:0] data_in,    // MEM
+    input [31:0] RAM_data_in,    // MEM
     input [31:0] inst_in,    // IF, from PC_out

-    output [31:0] ALU_out,   // From MEM, address out, for fetching data_in
+    output [31:0] ALU_out,   // From MEM, address out, for fetching RAM_data_in
     output [31:0] data_out,  // From MEM, to be written into data memory
     output mem_w,            // From MEM, write valid, for store instructions
     output [31:0] PC_out     // From IF
     );
     wire V5;
     wire N0;
+   wire [31:0] PC;
     wire [31:0] Imm_32;
     wire [31:0] add_branch_out;
     wire [31:0] add_jal_out;
]@@ -40,6 +41,7 @@ module RV32iPCPU(

     wire [4:0] Wt_addr;
     wire [31:0] Wt_data;
+//     wire [31:0] Wt_data_1;//for data forward from ALU out
     wire [31:0] rdata_A;
     wire [31:0] rdata_B;
     wire [31:0] PC_wb;
]@@ -50,15 +52,18 @@ module RV32iPCPU(


     wire zero;               // ID
-    wire [1:0] Branch;       // ID
+    wire [1:0] PC_jump_sel; // ID
     wire ALUSrc_A;           // EXE
     wire [1:0] ALUSrc_B;     // EXE
     wire [4:0] ALU_Control;  // EXE
     wire RegWrite;           // WB
     wire [1:0] DatatoReg;    // WB

-    wire [1:0] B_H_W;        // WB // not used yet
-    wire sign;               // WB // not used yet
+   wire [1:0] LOAD_type;
+   wire LOAD_sign;
+   wire [1:0] STORE_type;
+
+   wire [31:0] data_in;
```

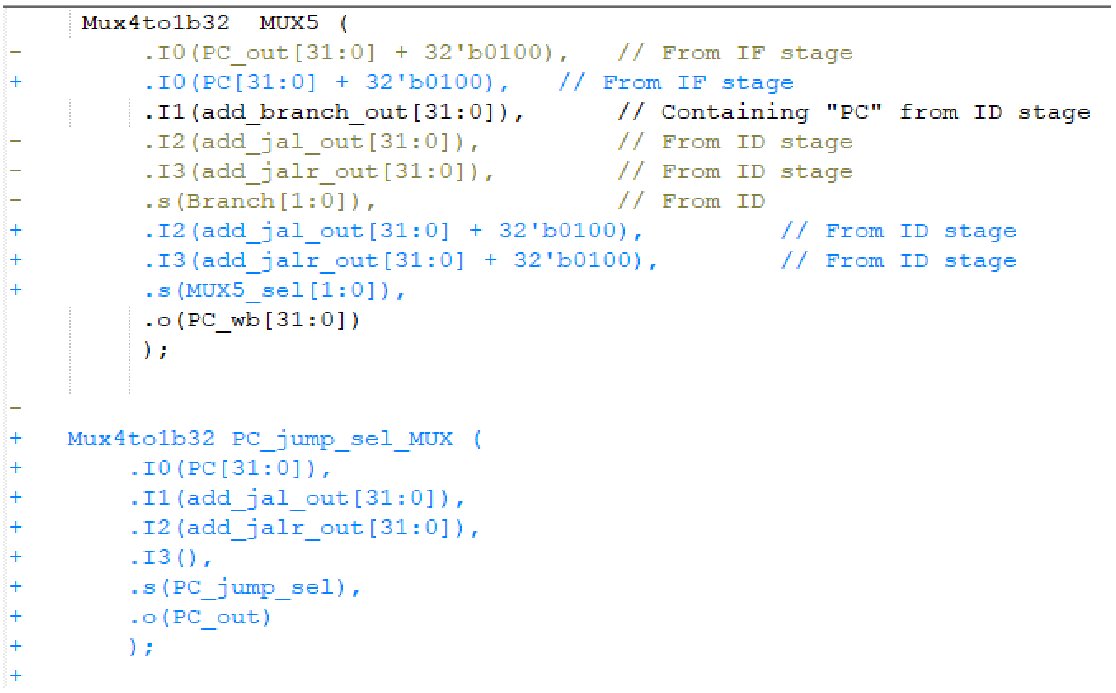
```
 //     wire RegDst; // WB
 //     wire Jal; // WB

@@ -71,6 +76,12 @@ module RV32iPCPU(
      wire [4:0] IF_ID_read_reg1;
      wire [4:0] IF_ID_read_reg2;

+     //ID
+     wire [31:0] Branch_data_ALU_A_1;
+     wire [31:0] Branch_data_ALU_B_1;
+     wire [31:0] Branch_data_ALU_A_2;
+     wire [31:0] Branch_data_ALU_B_2;
+
      // ID_EXE
      wire [31:0] ID_EXE_inst_in;
      wire [31:0] ID_EXE_PC;
@@ -81,20 +92,42 @@ module RV32iPCPU(
      wire ID_EXE_mem_w;
      wire [1:0] ID_EXE_DatatoReg;
      wire ID_EXE_RegWrite;
+
+    wire [1:0] ID_EXE_LOAD_type;
+    wire [1:0] ID_EXE_STORE_type;
+     wire ID_EXE_LOAD_sign;
+
      wire [4:0] ID_EXE_written_reg;
      wire [4:0] ID_EXE_read_reg1;
      wire [4:0] ID_EXE_read_reg2;

      wire [31:0] ID_EXE_ALU_out;
+
+    wire [31:0] ID_EXE_Data_extended;

+     // EXE
+     wire [31:0] _alu_Source_A_1_ALU_A;
+     wire [31:0] _alu_Source_A_2_ALU_A;
+     wire [31:0] _alu_Source_B_1_ALU_B;
+     wire [31:0] _alu_Source_B_2_ALU_B;
+     wire [31:0] ID_EXE_Data_out_1;
+     wire [31:0] ID_EXE_Data_out_2;
+    wire [31:0] _data_out_Source;
+    wire [31:0] _data_out_Source_1;
+
      // EXE_MEM
      wire [31:0] EXE_MEM_inst_in;
      wire [31:0] EXE_MEM_PC;
      wire [31:0] EXE_MEM_ALU_out;
      wire [31:0] EXE_MEM_Data_out;
+     wire [31:0]EXE_MEM_Data_out_1;
```



```
        wire EXE_MEM_mem_w;
        wire [1:0] EXE_MEM_DatatoReg;
        wire EXE_MEM_RegWrite;
+
+    wire [1:0] EXE_MEM_LOAD_type;
+     wire EXE_MEM_LOAD_sign;
+
        wire [4:0] EXE_MEM_written_reg;
        wire [4:0] EXE_MEM_read_reg1;
-       wire [4:0] EXE_MEM_read_reg2;
]@@ -108,35 +141,43 @@ module RV32iPCPU(
        wire MEM_WB_RegWrite;

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

+    //new signal
+    wire flush;
+    wire branch_load_dstall;
+    wire [31:0] Imm_IF;
+    wire [1:0] IF_B_MUX1_sel;
+    wire [1:0] IF_B_MUX2_sel;
+    wire [31:0] B_sel1_data;
+    wire [31:0] B_sel2_data;
+    wire [1:0] MUX5_sel;

+    wire ALU_forwarding_on_A;
+    wire ALU_forwarding_on_B;
+    wire ALU_forwarding_on_data_out;
+    wire [1:0] ALU_data_forward_sel_A;
+    wire [1:0] ALU_data_forward_sel_B;
+    wire [1:0] ALU_data_forward_sel_data_out;
+    wire branch_forwarding_on_A;
+    wire branch_forwarding_on_B;
+    wire [1:0] branch_data_forward_sel_A;
+    wire [1:0] branch_data_forward_sel_B;

-     Data_Stall _dstall_ (
-         .IF_ID_written_reg(IF_ID_written_reg),
-         .IF_ID_read_reg1(IF_ID_read_reg1),
-         .IF_ID_read_reg2(IF_ID_read_reg2),
-
-         .ID_EXE_written_reg(ID_EXE_written_reg),
-         .ID_EXE_read_reg1(ID_EXE_read_reg1),
-         .ID_EXE_read_reg2(ID_EXE_read_reg2),
-
-           .EXE_MEM_written_reg(EXE_MEM_written_reg),
-           .EXE_MEM_read_reg1(EXE_MEM_read_reg1),
-           .EXE_MEM_read_reg2(EXE_MEM_read_reg2),
-
-           .PC_dstall(PC_dstall),
-           .IF_ID_dstall(IF_ID_dstall),
-           .ID_EXE_dstall(ID_EXE_dstall)
-           );
-
+    wire [31:0] JALR_rs1_data;
+
     Control_Stall _cstall_ (
-           .Branch(Branch[1:0]),
+           .flush_signal(flush),
           .IF_ID_cstall(IF_ID_cstall)
           );
+
+    branch_load_check B_L_stall  (
+        .inst_in(inst_in),
+        .IF_ID_inst_in(IF_ID_inst_in),
+
+           .branch_load_dstall(branch_load_dstall)
+     );

     assign ALU_out = EXE_MEM_ALU_out;
     assign data_out = EXE_MEM_Data_out;
@@ -159,37 +200,85 @@ module RV32iPCPU(
          .clk(clk),
         .D(PC_wb[31:0]),
         .rst(rst),
-           .Q(PC_out[31:0]),
-           .PC_dstall(PC_dstall)
+           .Q(PC[31:0]),
+         .branch_load_dstall(branch_load_dstall)
         );
+    branch_predict predict_module1(
+        .clk(clk), .rst(rst), .CE(V5),
+        .IF_instr(inst_in),
+        .ID_instr(IF_ID_inst_in),
+        .ID_PC(IF_ID_PC),
+        .zero(zero),
+        .PC_mux_sel(MUX5_sel[1:0]),
+        .branch_PC_mux1_sel(IF_B_MUX1_sel),
+        .branch_PC_mux2_sel(IF_B_MUX2_sel),
+        .flush_signal(flush)
+    );
```

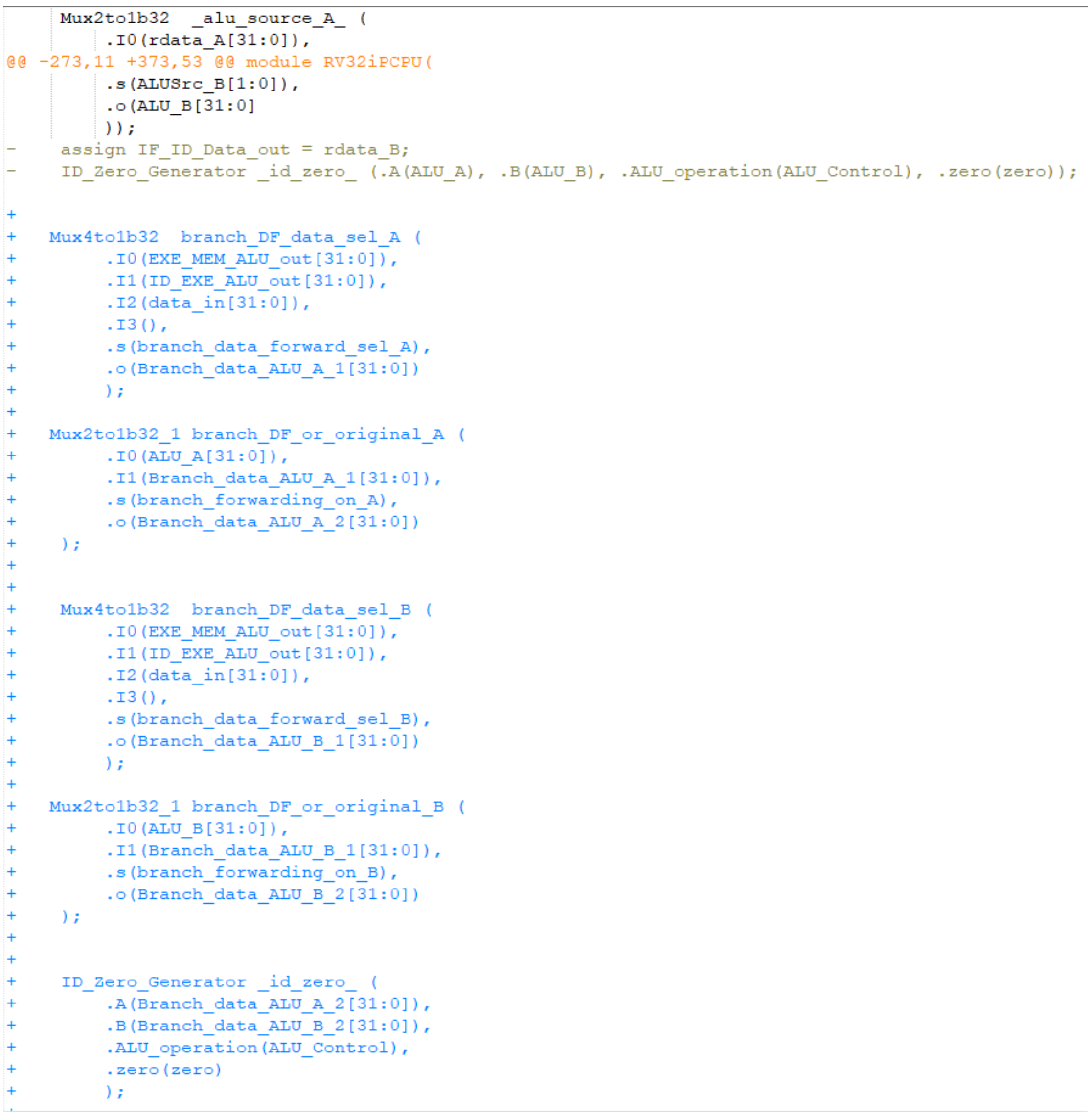


```
     add_32  ADD_Branch (
-        .a(IF_ID_PC[31:0]),          // use the "PC" from ID stage
-        .b(Imm_32[31:0]),            // From ID stage
-        .c(add_branch_out[31:0])     // actually this part belongs to IF_ID
+        .a(B_sel1_data[31:0]),
+        .b(B_sel2_data[31:0]),
+        .c(add_branch_out[31:0])
         );
     add_32 ADD_JAL (
-        .a(IF_ID_PC),                // MIPS: PC+4, RISC-V: PC!!!
+        .a(IF_ID_PC[31:0]),                // MIPS: PC+4, RISC-V: PC!!!
         .b({{11{IF_ID_inst_in[31]}}, IF_ID_inst_in[31], IF_ID_inst_in[19:12], IF_ID_inst_in[20], IF_ID_inst_in[30:21], 1'b0}),
         .c(add_jal_out[31:0])
         );
+
+   assign JALR_rs1_data = Branch_data_ALU_A_2;
     add_32 ADD_JALR (
-        .a(rdata_A[31:0]),
+        .a(JALR_rs1_data[31:0]),
         .b({{20{IF_ID_inst_in[31]}}, IF_ID_inst_in[31:20]}),
         .c(add_jalr_out[31:0])
         );
+
+   Mux4to1b32 branch_sel1 (
+       .I0(PC[31:0]),
+       .I1(IF_ID_PC),
+       .I2(),
+       .I3(),
+       .s(IF_B_MUX1_sel),
+       .o(B_sel1_data[31:0])
+       );
+
+   Mux4to1b32 branch_sel2 (
+       .I0(Imm_IF[31:0]),
+       .I1(32'b0100),
+       .I2(Imm_32[31:0]),
+       .I3(),
+       .s(IF_B_MUX2_sel),
+       .o(B_sel2_data[31:0])
+       );
+
+
+
+   SignExt _signed_ext_IF_stage_ (
+   .inst_in(inst_in),
+   .imm_32(Imm_IF)
+    );
+
```

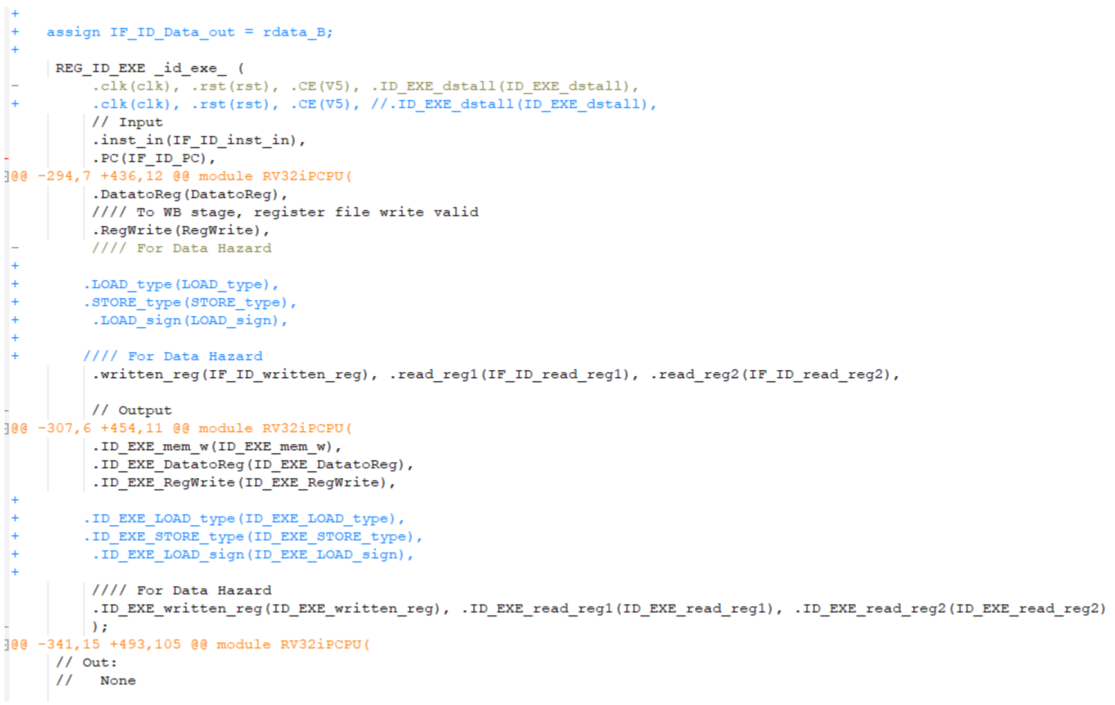


```
     Mux4to1b32  MUX5 (
-        .I0(PC_out[31:0] + 32'b0100),   // From IF stage
+        .I0(PC[31:0] + 32'b0100),   // From IF stage
         .I1(add_branch_out[31:0]),      // Containing "PC" from ID stage
-        .I2(add_jal_out[31:0]),         // From ID stage
-        .I3(add_jalr_out[31:0]),        // From ID stage
-        .s(Branch[1:0]),                // From ID
+        .I2(add_jal_out[31:0] + 32'b0100),          // From ID stage
+        .I3(add_jalr_out[31:0] + 32'b0100),         // From ID stage
+        .s(MUX5_sel[1:0]),
         .o(PC_wb[31:0])
         );

-
+   Mux4to1b32 PC_jump_sel_MUX (
+       .I0(PC[31:0]),
+       .I1(add_jal_out[31:0]),
+       .I2(add_jalr_out[31:0]),
+       .I3(),
+       .s(PC_jump_sel),
+       .o(PC_out)
+       );
+
```

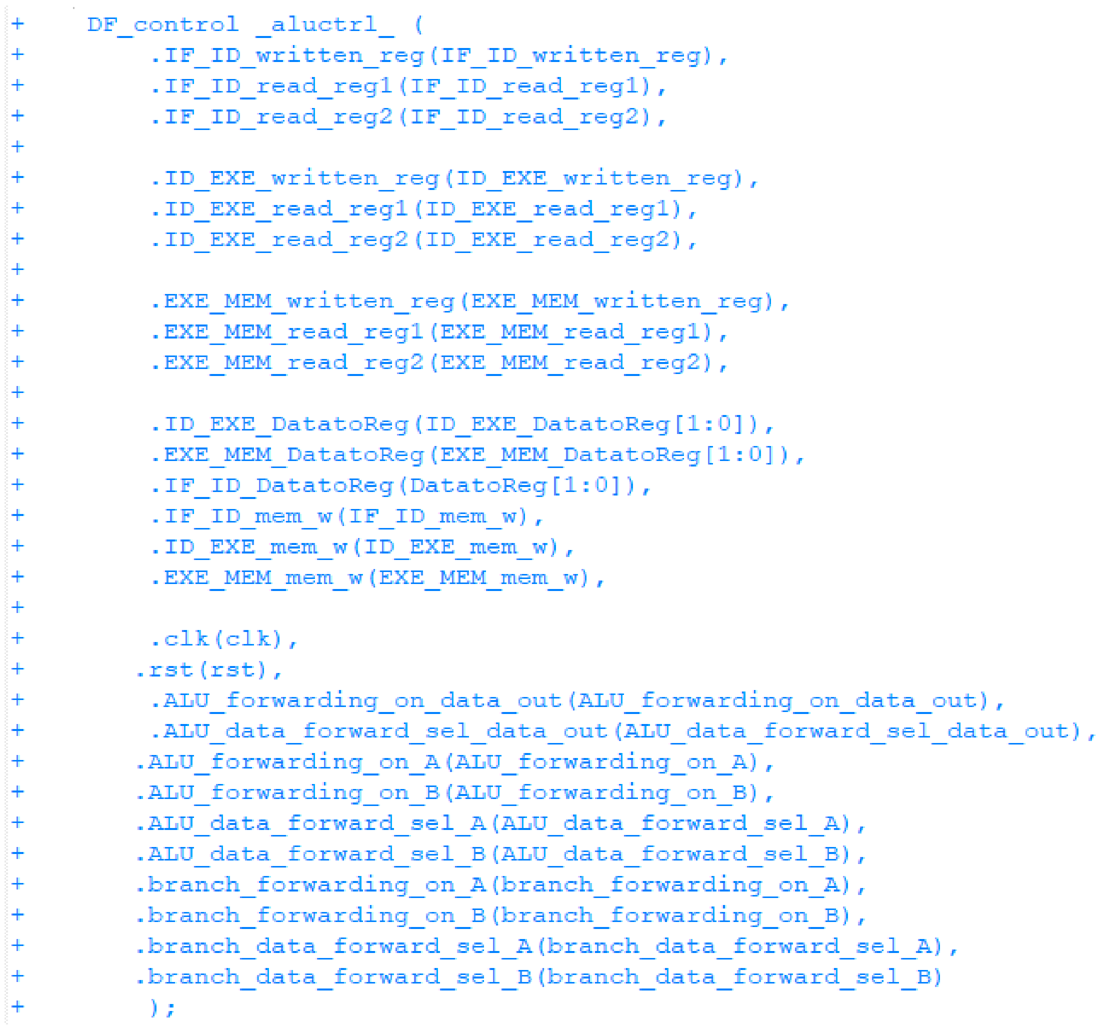
```diff
     REG_IF_ID _if_id_ (
         .clk(clk), .rst(rst), .CE(V5),
-         .IF_ID_dstall(IF_ID_dstall), .IF_ID_cstall(IF_ID_cstall),
+         .IF_ID_cstall(IF_ID_cstall),
+        .branch_load_dstall(branch_load_dstall),
         // Input
         .inst_in(inst_in),
         .PC(PC_out),
@@ -233,18 +322,26 @@ module RV32iPCPU(
         .OPcode(IF_ID_inst_in[6:0]),
         .Fun1(IF_ID_inst_in[14:12]),
         .Fun2(IF_ID_inst_in[31:25]),
-         .zero(zero),
+         .zero(zero),               //not used anymore
         // Output:
         .ALUSrc_A(ALUSrc_A),
         .ALUSrc_B(ALUSrc_B[1:0]),
         .ALU_Control(ALU_Control[4:0]),
-         .Branch(Branch[1:0]),
+         .PC_jump_sel(PC_jump_sel[1:0]),
         .DatatoReg(DatatoReg[1:0]),
         .mem_w(IF_ID_mem_w),
         .RegWrite(RegWrite),
-         .B_H_W(B_H_W),                  // not used yet
-         .sign(sign)                     // not used yet
+        .LOAD_type(LOAD_type),
+        .LOAD_sign(LOAD_sign),
+        .STORE_type(STORE_type)
         );
+
+//     Mux2to1b32_1 _alu_data_forward (
+//         .I0(Wt_data[31:0]),
+//         .I1(ID_EXE_ALU_out[31:0]),
+//         .s(replace_data_to_alu_data),
+////         .o(Wt_data_1[31:0])
+//        );

    Regs U2 (.clk(clk),
             .rst(rst),
@@ -256,7 +353,10 @@ module RV32iPCPU(
             .rdata_A(rdata_A[31:0]),
             .rdata_B(rdata_B[31:0])
             );
-    SignExt _signed_ext_ (.inst_in(IF_ID_inst_in), .imm_32(Imm_32));
+    SignExt _signed_ext_ (
+    .inst_in(IF_ID_inst_in),
+     .imm_32(Imm_32)
+     );
```



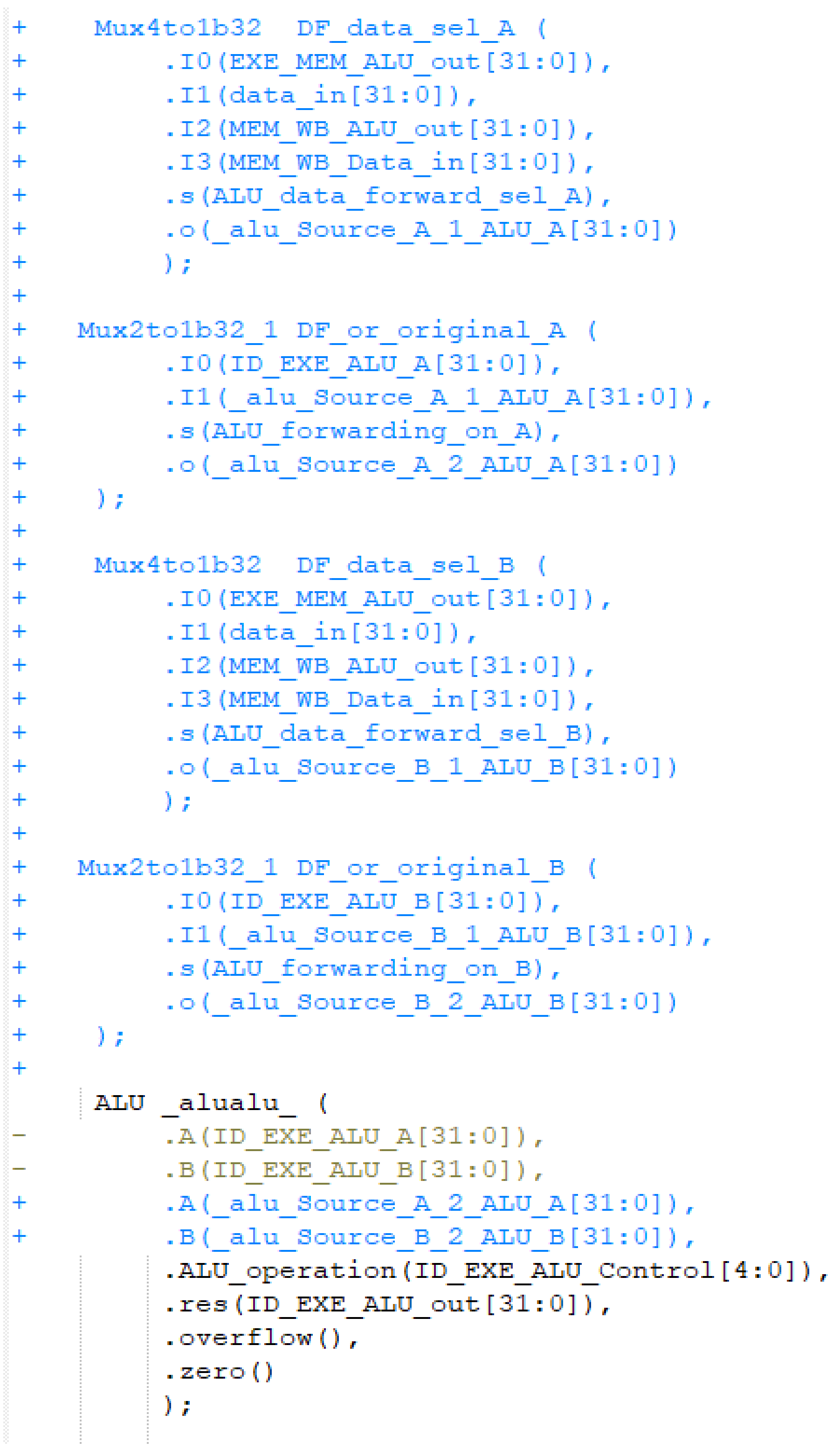
```
     Mux2to1b32   _alu_source_A_  (
         .I0(rdata_A[31:0]),
@@ -273,11 +373,53 @@ module RV32iPCPU(
         .s(ALUSrc_B[1:0]),
         .o(ALU_B[31:0]
         ));
-    assign IF_ID_Data_out = rdata_B;
-    ID_Zero_Generator _id_zero_ (.A(ALU_A), .B(ALU_B), .ALU_operation(ALU_Control), .zero(zero));

+
+   Mux4to1b32  branch_DF_data_sel_A (
+        .I0(EXE_MEM_ALU_out[31:0]),
+        .I1(ID_EXE_ALU_out[31:0]),
+        .I2(data_in[31:0]),
+        .I3(),
+        .s(branch_data_forward_sel_A),
+        .o(Branch_data_ALU_A_1[31:0])
+        );
+
+   Mux2to1b32_1 branch_DF_or_original_A (
+        .I0(ALU_A[31:0]),
+        .I1(Branch_data_ALU_A_1[31:0]),
+        .s(branch_forwarding_on_A),
+        .o(Branch_data_ALU_A_2[31:0])
+    );
+
+
+    Mux4to1b32  branch_DF_data_sel_B (
+        .I0(EXE_MEM_ALU_out[31:0]),
+        .I1(ID_EXE_ALU_out[31:0]),
+        .I2(data_in[31:0]),
+        .I3(),
+        .s(branch_data_forward_sel_B),
+        .o(Branch_data_ALU_B_1[31:0])
+        );
+
+   Mux2to1b32_1 branch_DF_or_original_B (
+        .I0(ALU_B[31:0]),
+        .I1(Branch_data_ALU_B_1[31:0]),
+        .s(branch_forwarding_on_B),
+        .o(Branch_data_ALU_B_2[31:0])
+    );
+
+
+    ID_Zero_Generator _id_zero_  (
+        .A(Branch_data_ALU_A_2[31:0]),
+        .B(Branch_data_ALU_B_2[31:0]),
+        .ALU_operation(ALU_Control),
+        .zero(zero)
+        );
```



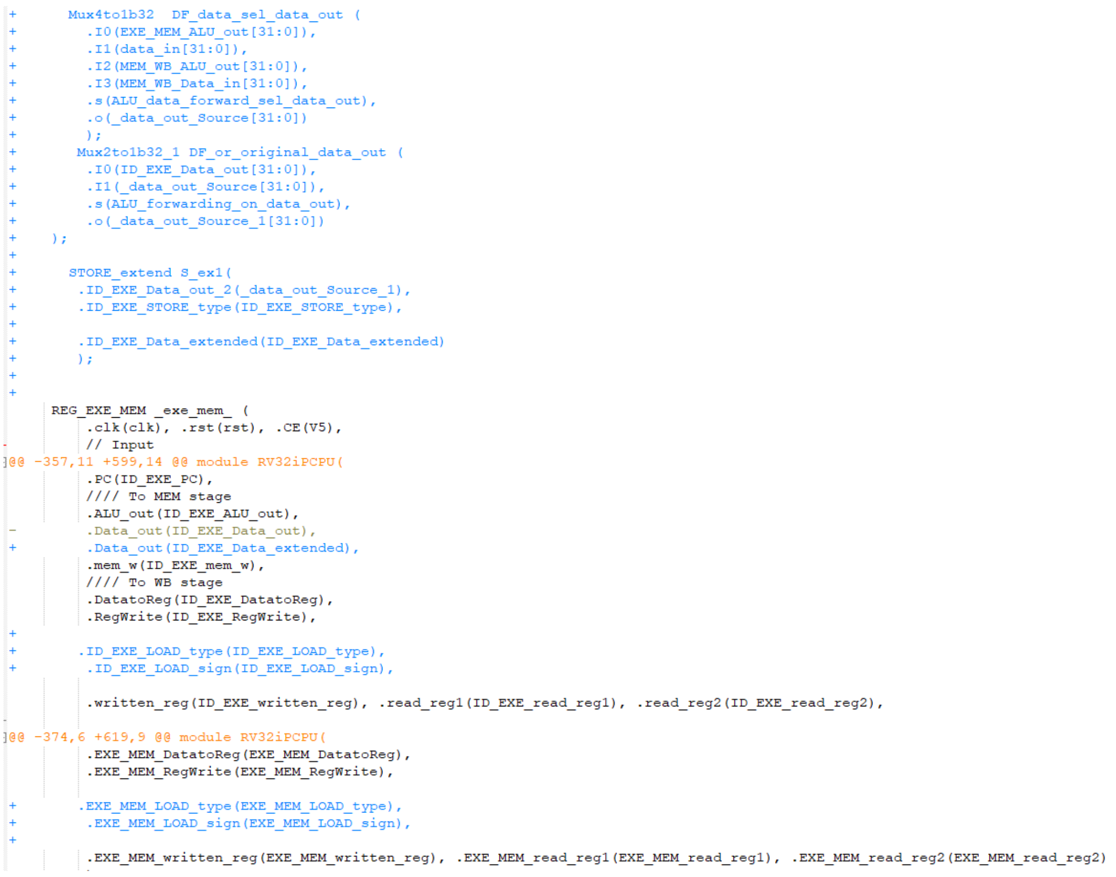
```
+
+   assign IF_ID_Data_out = rdata_B;
+
    REG_ID_EXE _id_exe_ (
-       .clk(clk), .rst(rst), .CE(V5), .ID_EXE_dstall(ID_EXE_dstall),
+       .clk(clk), .rst(rst), .CE(V5), //.ID_EXE_dstall(ID_EXE_dstall),
        // Input
        .inst_in(IF_ID_inst_in),
-       .PC(IF_ID_PC),
]@@ -294,7 +436,12 @@ module RV32iPCPU(
        .DatatoReg(DatatoReg),
        //// To WB stage, register file write valid
        .RegWrite(RegWrite),
-       //// For Data Hazard
+
+      .LOAD_type(LOAD_type),
+      .STORE_type(STORE_type),
+       .LOAD_sign(LOAD_sign),
+
+      //// For Data Hazard
        .written_reg(IF_ID_written_reg), .read_reg1(IF_ID_read_reg1), .read_reg2(IF_ID_read_reg2),

-       // Output
]@@ -307,6 +454,11 @@ module RV32iPCPU(
        .ID_EXE_mem_w(ID_EXE_mem_w),
        .ID_EXE_DatatoReg(ID_EXE_DatatoReg),
        .ID_EXE_RegWrite(ID_EXE_RegWrite),
+
+      .ID_EXE_LOAD_type(ID_EXE_LOAD_type),
+      .ID_EXE_STORE_type(ID_EXE_STORE_type),
+       .ID_EXE_LOAD_sign(ID_EXE_LOAD_sign),
+
        //// For Data Hazard
        .ID_EXE_written_reg(ID_EXE_written_reg), .ID_EXE_read_reg1(ID_EXE_read_reg1), .ID_EXE_read_reg2(ID_EXE_read_reg2)
-       );
]@@ -341,15 +493,105 @@ module RV32iPCPU(
    // Out:
    //   None
```



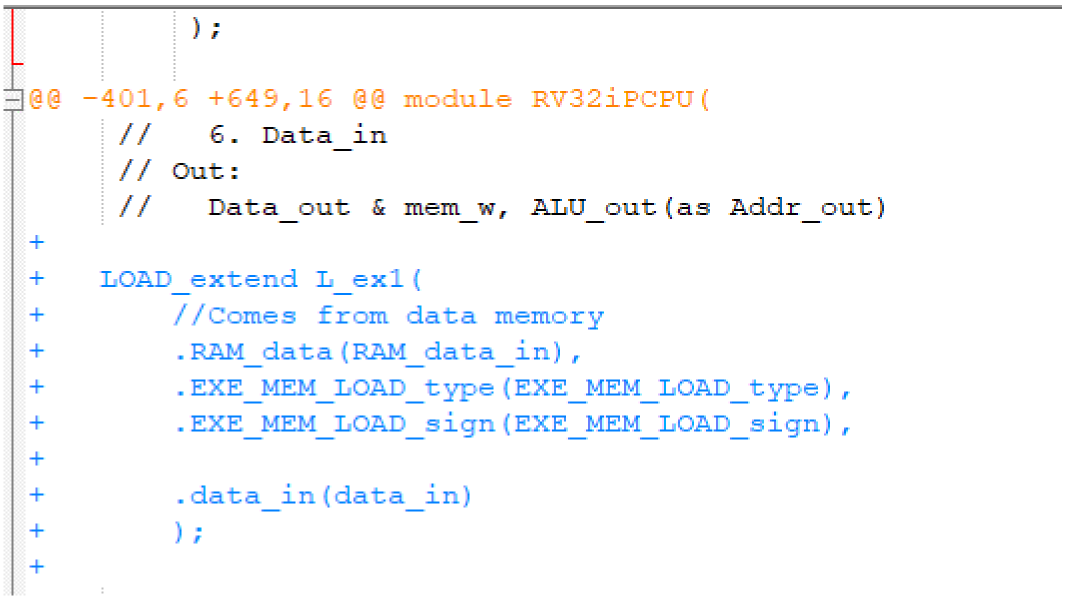
```
+    DF_control _aluctrl_ (
+        .IF_ID_written_reg(IF_ID_written_reg),
+        .IF_ID_read_reg1(IF_ID_read_reg1),
+        .IF_ID_read_reg2(IF_ID_read_reg2),
+
+        .ID_EXE_written_reg(ID_EXE_written_reg),
+        .ID_EXE_read_reg1(ID_EXE_read_reg1),
+        .ID_EXE_read_reg2(ID_EXE_read_reg2),
+
+        .EXE_MEM_written_reg(EXE_MEM_written_reg),
+        .EXE_MEM_read_reg1(EXE_MEM_read_reg1),
+        .EXE_MEM_read_reg2(EXE_MEM_read_reg2),
+
+        .ID_EXE_DatatoReg(ID_EXE_DatatoReg[1:0]),
+        .EXE_MEM_DatatoReg(EXE_MEM_DatatoReg[1:0]),
+        .IF_ID_DatatoReg(DatatoReg[1:0]),
+        .IF_ID_mem_w(IF_ID_mem_w),
+        .ID_EXE_mem_w(ID_EXE_mem_w),
+        .EXE_MEM_mem_w(EXE_MEM_mem_w),
+
+        .clk(clk),
+       .rst(rst),
+        .ALU_forwarding_on_data_out(ALU_forwarding_on_data_out),
+        .ALU_data_forward_sel_data_out(ALU_data_forward_sel_data_out),
+       .ALU_forwarding_on_A(ALU_forwarding_on_A),
+       .ALU_forwarding_on_B(ALU_forwarding_on_B),
+       .ALU_data_forward_sel_A(ALU_data_forward_sel_A),
+       .ALU_data_forward_sel_B(ALU_data_forward_sel_B),
+       .branch_forwarding_on_A(branch_forwarding_on_A),
+       .branch_forwarding_on_B(branch_forwarding_on_B),
+       .branch_data_forward_sel_A(branch_data_forward_sel_A),
+       .branch_data_forward_sel_B(branch_data_forward_sel_B)
+        );
```



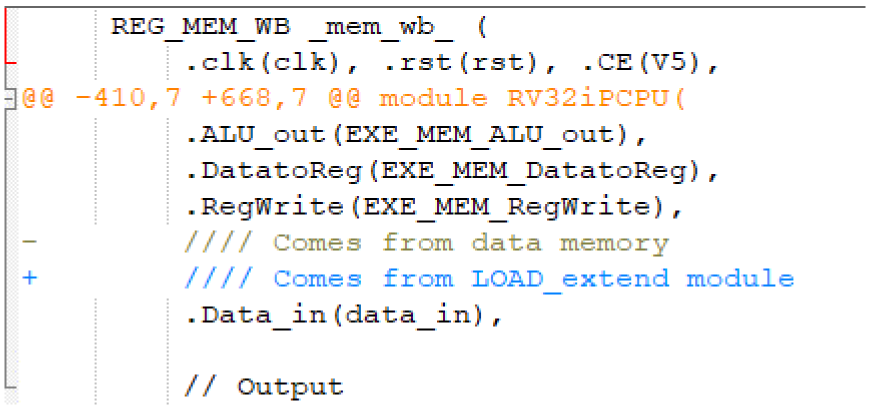
```
+    Mux4to1b32  DF_data_sel_A (
+        .I0(EXE_MEM_ALU_out[31:0]),
+        .I1(data_in[31:0]),
+        .I2(MEM_WB_ALU_out[31:0]),
+        .I3(MEM_WB_Data_in[31:0]),
+        .s(ALU_data_forward_sel_A),
+        .o(_alu_Source_A_1_ALU_A[31:0])
+        );
+
+   Mux2to1b32_1 DF_or_original_A (
+        .I0(ID_EXE_ALU_A[31:0]),
+        .I1(_alu_Source_A_1_ALU_A[31:0]),
+        .s(ALU_forwarding_on_A),
+        .o(_alu_Source_A_2_ALU_A[31:0])
+    );
+
+    Mux4to1b32  DF_data_sel_B (
+        .I0(EXE_MEM_ALU_out[31:0]),
+        .I1(data_in[31:0]),
+        .I2(MEM_WB_ALU_out[31:0]),
+        .I3(MEM_WB_Data_in[31:0]),
+        .s(ALU_data_forward_sel_B),
+        .o(_alu_Source_B_1_ALU_B[31:0])
+        );
+
+   Mux2to1b32_1 DF_or_original_B (
+        .I0(ID_EXE_ALU_B[31:0]),
+        .I1(_alu_Source_B_1_ALU_B[31:0]),
+        .s(ALU_forwarding_on_B),
+        .o(_alu_Source_B_2_ALU_B[31:0])
+    );
+
     ALU _alualu_ (
-        .A(ID_EXE_ALU_A[31:0]),
-        .B(ID_EXE_ALU_B[31:0]),
+        .A(_alu_Source_A_2_ALU_A[31:0]),
+        .B(_alu_Source_B_2_ALU_B[31:0]),
         .ALU_operation(ID_EXE_ALU_Control[4:0]),
         .res(ID_EXE_ALU_out[31:0]),
         .overflow(),
         .zero()
         );
```



```
+      Mux4to1b32  DF_data_sel_data_out (
+        .I0(EXE_MEM_ALU_out[31:0]),
+        .I1(data_in[31:0]),
+        .I2(MEM_WB_ALU_out[31:0]),
+        .I3(MEM_WB_Data_in[31:0]),
+        .s(ALU_data_forward_sel_data_out),
+        .o(_data_out_Source[31:0])
+        );
+       Mux2to1b32_1 DF_or_original_data_out (
+        .I0(ID_EXE_Data_out[31:0]),
+        .I1(_data_out_Source[31:0]),
+        .s(ALU_forwarding_on_data_out),
+        .o(_data_out_Source_1[31:0])
+    );
+
+      STORE_extend S_ex1(
+       .ID_EXE_Data_out_2(_data_out_Source_1),
+       .ID_EXE_STORE_type(ID_EXE_STORE_type),
+
+       .ID_EXE_Data_extended(ID_EXE_Data_extended)
+       );
+
+
     REG_EXE_MEM _exe_mem_ (
         .clk(clk), .rst(rst), .CE(V5),
-        // Input
@@ -357,11 +599,14 @@ module RV32iPCPU(
         .PC(ID_EXE_PC),
         //// To MEM stage
         .ALU_out(ID_EXE_ALU_out),
-        .Data_out(ID_EXE_Data_out),
+        .Data_out(ID_EXE_Data_extended),
         .mem_w(ID_EXE_mem_w),
         //// To WB stage
         .DatatoReg(ID_EXE_DatatoReg),
         .RegWrite(ID_EXE_RegWrite),
+
+       .ID_EXE_LOAD_type(ID_EXE_LOAD_type),
+        .ID_EXE_LOAD_sign(ID_EXE_LOAD_sign),

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

@@ -374,6 +619,9 @@ module RV32iPCPU(
         .EXE_MEM_DatatoReg(EXE_MEM_DatatoReg),
         .EXE_MEM_RegWrite(EXE_MEM_RegWrite),

+       .EXE_MEM_LOAD_type(EXE_MEM_LOAD_type),
+        .EXE_MEM_LOAD_sign(EXE_MEM_LOAD_sign),
+
        .EXE_MEM_written_reg(EXE_MEM_written_reg), .EXE_MEM_read_reg1(EXE_MEM_read_reg1), .EXE_MEM_read_reg2(EXE_MEM_read_reg2)
```

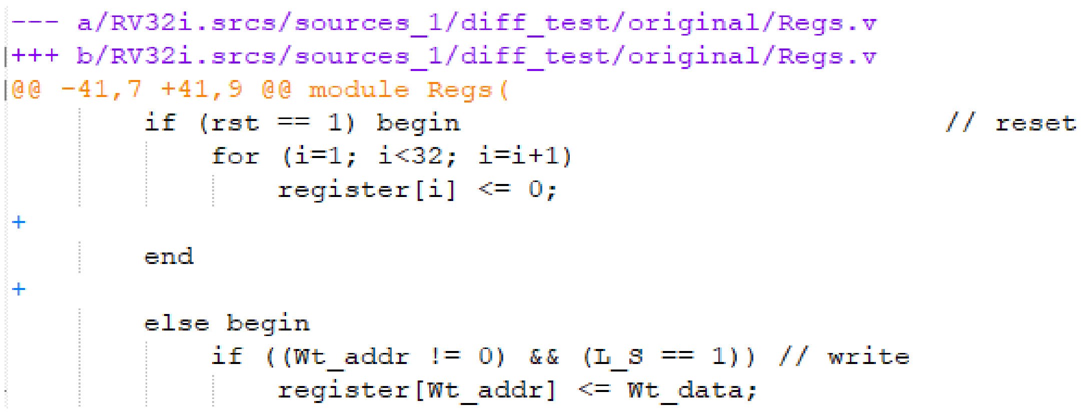


```
            );

@@ -401,6 +649,16 @@ module RV32iPCPU(
        //    6. Data_in
        // Out:
        //    Data_out & mem_w, ALU_out(as Addr_out)
+
+      LOAD_extend L_ex1(
+          //Comes from data memory
+          .RAM_data(RAM_data_in),
+          .EXE_MEM_LOAD_type(EXE_MEM_LOAD_type),
+          .EXE_MEM_LOAD_sign(EXE_MEM_LOAD_sign),
+
+          .data_in(data_in)
+          );
+
```



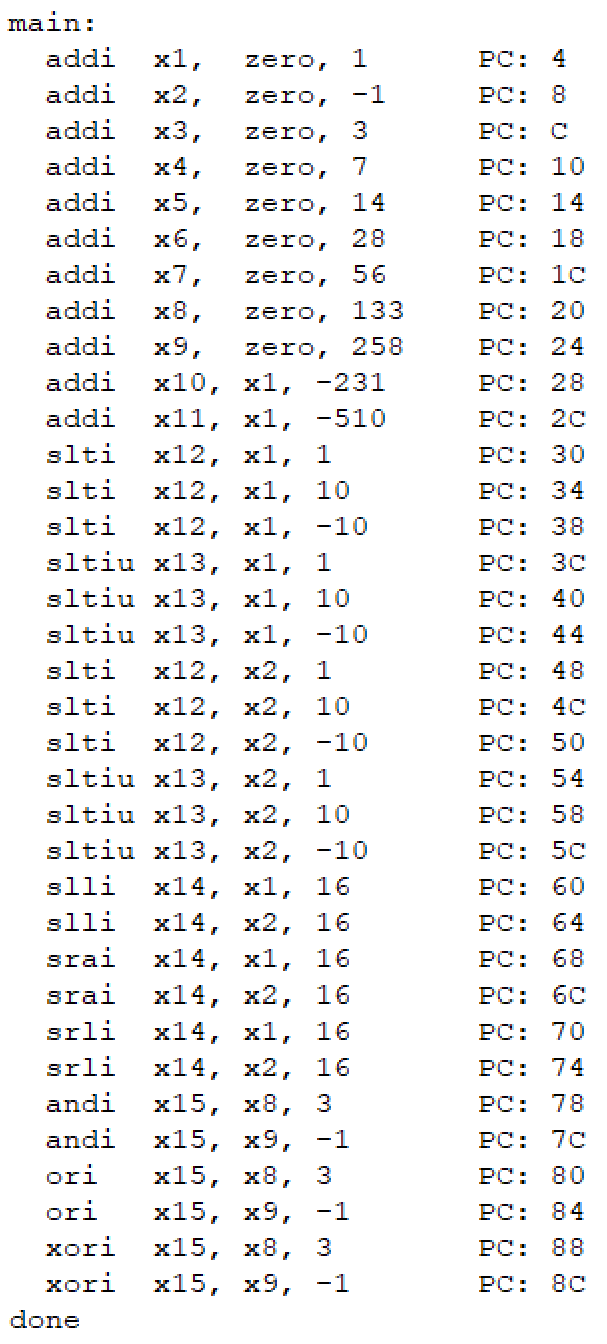
```
        REG_MEM_WB _mem_wb_ (
            .clk(clk), .rst(rst), .CE(V5),
@@ -410,7 +668,7 @@ module RV32iPCPU(
            .ALU_out(EXE_MEM_ALU_out),
            .DatatoReg(EXE_MEM_DatatoReg),
            .RegWrite(EXE_MEM_RegWrite),
-           //// Comes from data memory
+           //// Comes from LOAD_extend module
            .Data_in(data_in),

            // Output
```

Regs.v:



```
--- a/RV32i.srcs/sources_1/diff_test/original/Regs.v
+++ b/RV32i.srcs/sources_1/diff_test/original/Regs.v
@@ -41,7 +41,9 @@ module Regs(
        if (rst == 1) begin                                 // reset
            for (i=1; i<32; i=i+1)
                register[i] <= 0;
+
        end
+
        else begin
            if ((Wt_addr != 0) && (L_S == 1)) // write
                register[Wt_addr] <= Wt_data;
```

Appendix A: Instruction for processor arith check

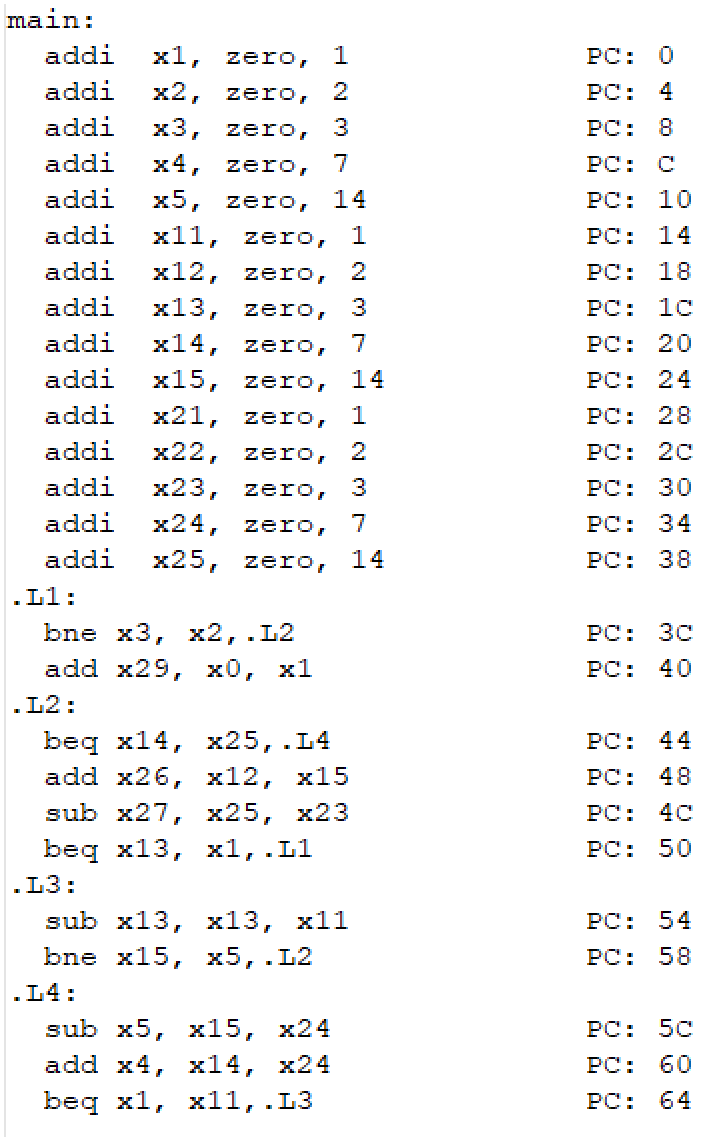


```
main:
  addi  x1,   zero, 1       PC: 4
  addi  x2,   zero, -1      PC: 8
  addi  x3,   zero, 3       PC: C
  addi  x4,   zero, 7       PC: 10
  addi  x5,   zero, 14      PC: 14
  addi  x6,   zero, 28      PC: 18
  addi  x7,   zero, 56      PC: 1C
  addi  x8,   zero, 133     PC: 20
  addi  x9,   zero, 258     PC: 24
  addi  x10, x1, -231       PC: 28
  addi  x11, x1, -510       PC: 2C
  slti  x12, x1, 1          PC: 30
  slti  x12, x1, 10         PC: 34
  slti  x12, x1, -10        PC: 38
  sltiu x13, x1, 1          PC: 3C
  sltiu x13, x1, 10         PC: 40
  sltiu x13, x1, -10        PC: 44
  slti  x12, x2, 1          PC: 48
  slti  x12, x2, 10         PC: 4C
  slti  x12, x2, -10        PC: 50
  sltiu x13, x2, 1          PC: 54
  sltiu x13, x2, 10         PC: 58
  sltiu x13, x2, -10        PC: 5C
  slli  x14, x1, 16         PC: 60
  slli  x14, x2, 16         PC: 64
  srai  x14, x1, 16         PC: 68
  srai  x14, x2, 16         PC: 6C
  srli  x14, x1, 16         PC: 70
  srli  x14, x2, 16         PC: 74
  andi  x15, x8, 3          PC: 78
  andi  x15, x9, -1         PC: 7C
  ori   x15, x8, 3          PC: 80
  ori   x15, x9, -1         PC: 84
  xori  x15, x8, 3          PC: 88
  xori  x15, x9, -1         PC: 8C
done
```

Appendix B: Instruction for data hazard study

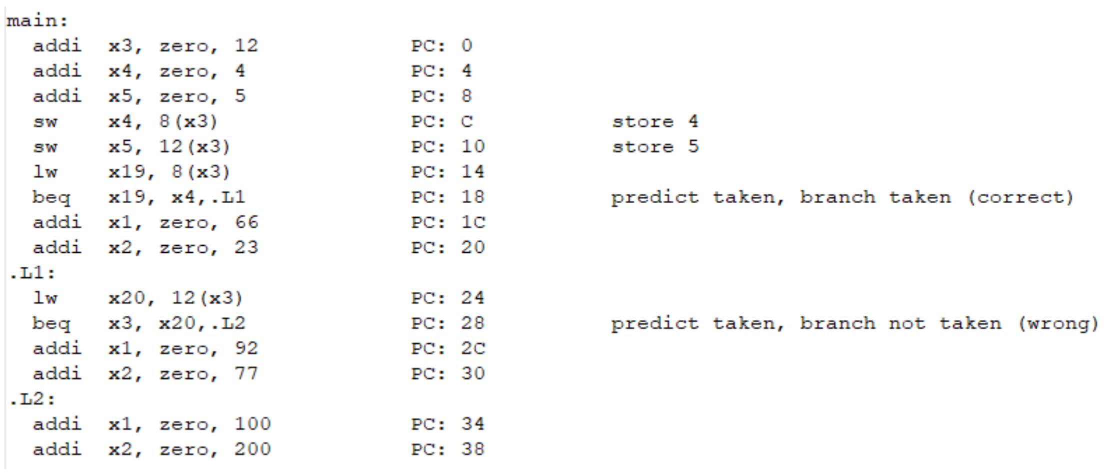
```
main:
  addi x0,  x1, 1           PC: 4
  addi x0,  x1, 2           PC: 8
  addi x0,  x1, 3           PC: C
  addi x0,  x1, 7           PC: 10
  addi x0,  x1, 14          PC: 14
  addi x0,  x1, 28          PC: 18
  addi x0,  x1, 56          PC: 1C
  addi x31, x1, 1           PC: 20
  addi x31, x1, 2           PC: 24
  addi x31, x1, 3           PC: 28
  addi x31, x1, 7           PC: 2C
  addi x31, x1, 14          PC: 30
  addi x31, x1, 28          PC: 34
  addi x31, x1, 56          PC: 38
  addi x31, x1, 133         PC: 3C
  addi x31, x1, 258         PC: 40
  addi x31, x1, 511         PC: 44
  addi x31, x1, -1          PC: 48
  addi x31, x1, -3          PC: 4C
  addi x31, x1, -9          PC: 50
  addi x31, x1, -98         PC: 54
  addi x31, x1, -231        PC: 58
  addi x31, x1, -510        PC: 5C
  addi x30, x0, 1           PC: 60
  addi x30, x30, 2          PC: 64
  addi x30, x30, 3          PC: 68
  addi x30, x30, 7          PC: 6C
  addi x30, x30, 14         PC: 70
  addi x30, x30, 28         PC: 74
  addi x30, x30, 56         PC: 78
  addi x30, x30, 133        PC: 7C
  addi x30, x30, 258        PC: 80
  addi x30, x30, 511        PC: 84
  addi x30, x30, -1         PC: 88
  addi x30, x30, -3         PC: 8C
  addi x30, x30, -9         PC: 90
  addi x30, x30, -98        PC: 94
  addi x30, x30, -231       PC: 98
  addi x30, x30, -510       PC: 9C
  done
```

Appendix C: Instruction for branch predictor study

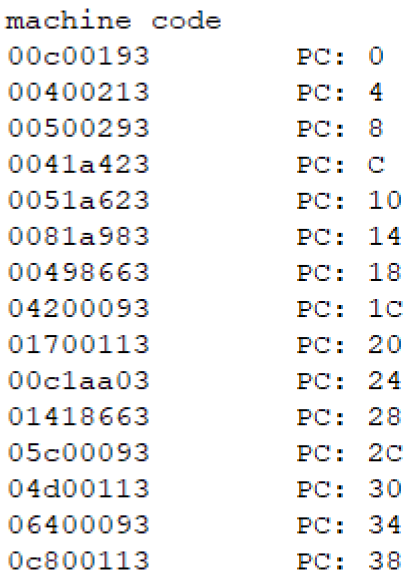


```
main:
  addi   x1, zero, 1               PC: 0
  addi   x2, zero, 2               PC: 4
  addi   x3, zero, 3               PC: 8
  addi   x4, zero, 7               PC: C
  addi   x5, zero, 14              PC: 10
  addi   x11, zero, 1              PC: 14
  addi   x12, zero, 2              PC: 18
  addi   x13, zero, 3              PC: 1C
  addi   x14, zero, 7              PC: 20
  addi   x15, zero, 14             PC: 24
  addi   x21, zero, 1              PC: 28
  addi   x22, zero, 2              PC: 2C
  addi   x23, zero, 3              PC: 30
  addi   x24, zero, 7              PC: 34
  addi   x25, zero, 14             PC: 38
.L1:
  bne x3, x2,.L2                   PC: 3C
  add x29, x0, x1                  PC: 40
.L2:
  beq x14, x25,.L4                 PC: 44
  add x26, x12, x15                PC: 48
  sub x27, x25, x23                PC: 4C
  beq x13, x1,.L1                  PC: 50
.L3:
  sub x13, x13, x11                PC: 54
  bne x15, x5,.L2                  PC: 58
.L4:
  sub x5, x15, x24                 PC: 5C
  add x4, x14, x24                 PC: 60
  beq x1, x11,.L3                  PC: 64
```

Appendix D: Instruction for load and branch instruction dependency study

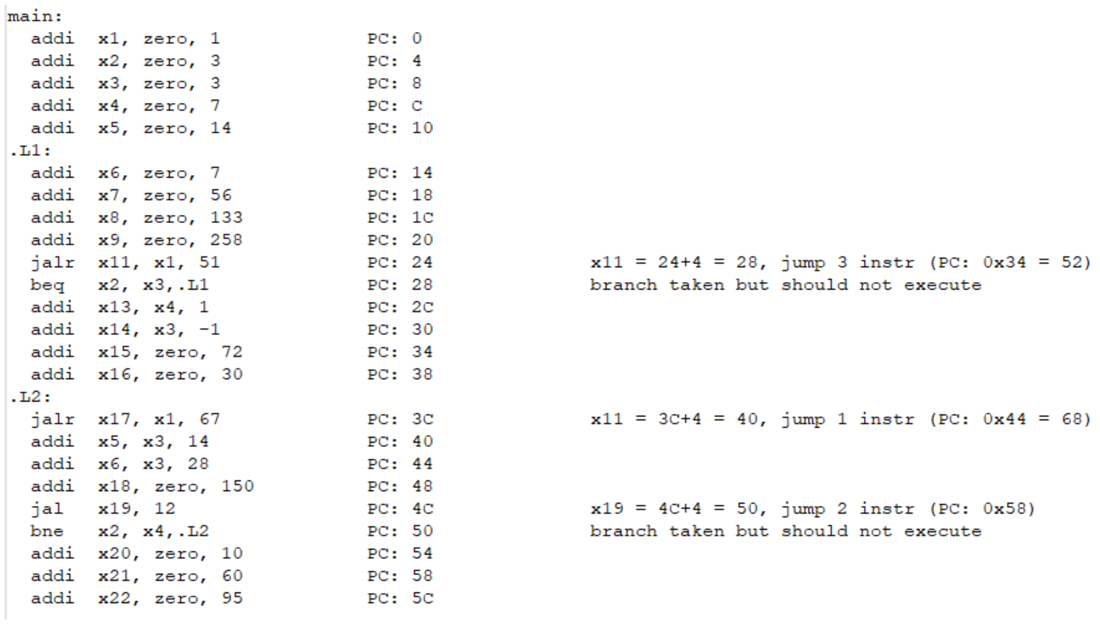


```
main:
  addi  x3, zero, 12          PC: 0
  addi  x4, zero, 4           PC: 4
  addi  x5, zero, 5           PC: 8
  sw    x4, 8(x3)             PC: C          store 4
  sw    x5, 12(x3)            PC: 10         store 5
  lw    x19, 8(x3)            PC: 14
  beq   x19, x4,.L1           PC: 18         predict taken, branch taken (correct)
  addi  x1, zero, 66          PC: 1C
  addi  x2, zero, 23          PC: 20
.L1:
  lw    x20, 12(x3)           PC: 24
  beq   x3, x20,.L2           PC: 28         predict taken, branch not taken (wrong)
  addi  x1, zero, 92          PC: 2C
  addi  x2, zero, 77          PC: 30
.L2:
  addi  x1, zero, 100         PC: 34
  addi  x2, zero, 200         PC: 38
```



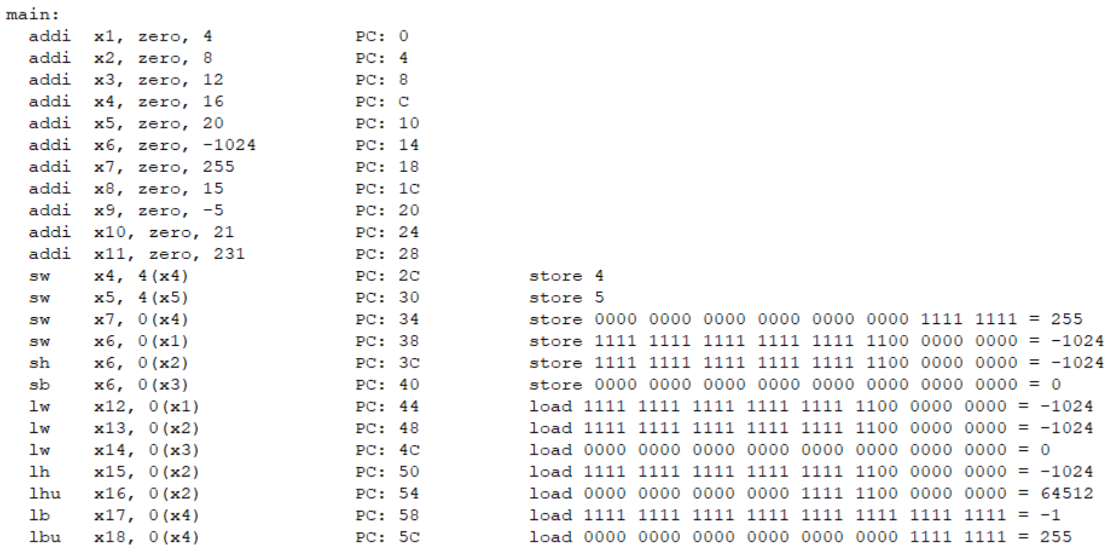
```
machine code
00c00193        PC: 0
00400213        PC: 4
00500293        PC: 8
0041a423        PC: C
0051a623        PC: 10
0081a983        PC: 14
00498663        PC: 18
04200093        PC: 1C
01700113        PC: 20
00c1aa03        PC: 24
01418663        PC: 28
05c00093        PC: 2C
04d00113        PC: 30
06400093        PC: 34
0c800113        PC: 38
```

Appendix E: Instruction for jump stall removal study



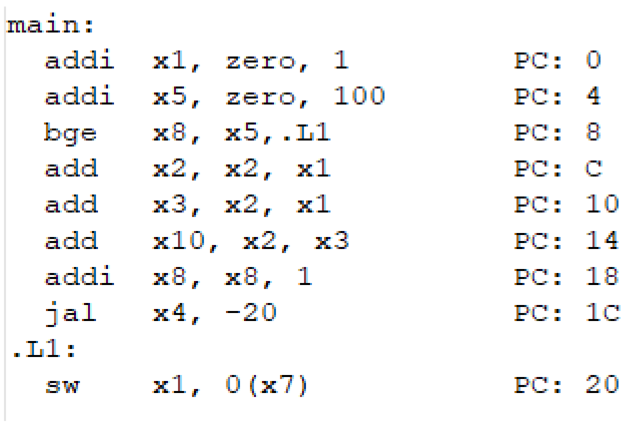
```
main:
  addi  x1, zero, 1          PC: 0
  addi  x2, zero, 3          PC: 4
  addi  x3, zero, 3          PC: 8
  addi  x4, zero, 7          PC: C
  addi  x5, zero, 14         PC: 10
.L1:
  addi  x6, zero, 7          PC: 14
  addi  x7, zero, 56         PC: 18
  addi  x8, zero, 133        PC: 1C
  addi  x9, zero, 258        PC: 20
  jalr  x11, x1, 51          PC: 24             x11 = 24+4 = 28, jump 3 instr (PC: 0x34 = 52)
  beq   x2, x3,.L1           PC: 28             branch taken but should not execute
  addi  x13, x4, 1           PC: 2C
  addi  x14, x3, -1          PC: 30
  addi  x15, zero, 72        PC: 34
  addi  x16, zero, 30        PC: 38
.L2:
  jalr  x17, x1, 67          PC: 3C             x11 = 3C+4 = 40, jump 1 instr (PC: 0x44 = 68)
  addi  x5, x3, 14           PC: 40
  addi  x6, x3, 28           PC: 44
  addi  x18, zero, 150       PC: 48
  jal   x19, 12              PC: 4C             x19 = 4C+4 = 50, jump 2 instr (PC: 0x58)
  bne   x2, x4,.L2           PC: 50             branch taken but should not execute
  addi  x20, zero, 10        PC: 54
  addi  x21, zero, 60        PC: 58
  addi  x22, zero, 95        PC: 5C
```

Appendix F: Instruction for load and store extension study



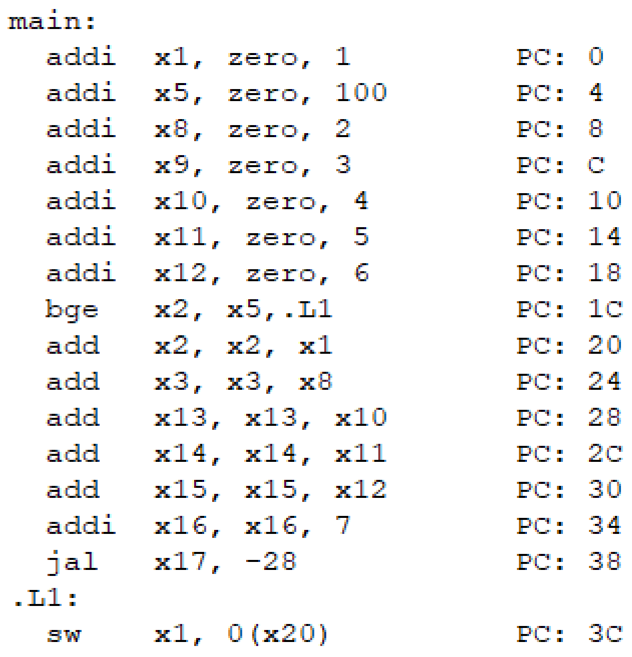
```
main:
  addi  x1, zero, 4           PC: 0
  addi  x2, zero, 8           PC: 4
  addi  x3, zero, 12          PC: 8
  addi  x4, zero, 16          PC: C
  addi  x5, zero, 20          PC: 10
  addi  x6, zero, -1024       PC: 14
  addi  x7, zero, 255         PC: 18
  addi  x8, zero, 15          PC: 1C
  addi  x9, zero, -5          PC: 20
  addi  x10, zero, 21         PC: 24
  addi  x11, zero, 231        PC: 28
  sw    x4, 4(x4)             PC: 2C        store 4
  sw    x5, 4(x5)             PC: 30        store 5
  sw    x7, 0(x4)             PC: 34        store 0000 0000 0000 0000 0000 0000 1111 1111 = 255
  sw    x6, 0(x1)             PC: 38        store 1111 1111 1111 1111 1111 1100 0000 0000 = -1024
  sh    x6, 0(x2)             PC: 3C        store 1111 1111 1111 1111 1111 1100 0000 0000 = -1024
  sb    x6, 0(x3)             PC: 40        store 0000 0000 0000 0000 0000 0000 0000 0000 = 0
  lw    x12, 0(x1)            PC: 44        load 1111 1111 1111 1111 1111 1100 0000 0000 = -1024
  lw    x13, 0(x2)            PC: 48        load 1111 1111 1111 1111 1111 1100 0000 0000 = -1024
  lw    x14, 0(x3)            PC: 4C        load 0000 0000 0000 0000 0000 0000 0000 0000 = 0
  lh    x15, 0(x2)            PC: 50        load 1111 1111 1111 1111 1111 1100 0000 0000 = -1024
  lhu   x16, 0(x2)            PC: 54        load 0000 0000 0000 0000 1111 1100 0000 0000 = 64512
  lb    x17, 0(x4)            PC: 58        load 1111 1111 1111 1111 1111 1111 1111 1111 = -1
  lbu   x18, 0(x4)            PC: 5C        load 0000 0000 0000 0000 0000 0000 1111 1111 = 255
```

Appendix G: Instruction for speed test (data dependency + branch + jump)



```
main:
  addi  x1, zero, 1         PC: 0
  addi  x5, zero, 100       PC: 4
  bge   x8, x5,.L1          PC: 8
  add   x2, x2, x1          PC: C
  add   x3, x2, x1          PC: 10
  add   x10, x2, x3         PC: 14
  addi  x8, x8, 1           PC: 18
  jal   x4, -20             PC: 1C
.L1:
  sw    x1, 0(x7)           PC: 20
```

Appendix H: Instruction for speed test (branch + jump)



```
main:
  addi  x1, zero, 1        PC: 0
  addi  x5, zero, 100      PC: 4
  addi  x8, zero, 2        PC: 8
  addi  x9, zero, 3        PC: C
  addi  x10, zero, 4       PC: 10
  addi  x11, zero, 5       PC: 14
  addi  x12, zero, 6       PC: 18
  bge   x2, x5,.L1         PC: 1C
  add   x2, x2, x1         PC: 20
  add   x3, x3, x8         PC: 24
  add   x13, x13, x10      PC: 28
  add   x14, x14, x11      PC: 2C
  add   x15, x15, x12      PC: 30
  addi  x16, x16, 7        PC: 34
  jal   x17, -28           PC: 38
.L1:
  sw    x1, 0(x20)         PC: 3C
```

Appendix I: Data Forward Control Condition design

Condition 0	Condition 1	Condition 2	Condition 3	Forward data	From Stage	To Stage	Data replce to
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == ID_EXE_read_reg_1	DatatoReg == 00	N/A	ALU_data	MEM	EXE	ALU_A
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_1	DatatoReg == 00	N/A		WB		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == ID_EXE_read_reg_1	DatatoReg == 01	N/A	Memory data	MEM		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_1	DatatoReg == 01	N/A		WB		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == ID_EXE_read_reg_2	DatatoReg == 00	ID_EXE_mem_w=0	ALU_data	MEM		ALU_B
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_2	DatatoReg == 00	ID_EXE_mem_w=0		WB		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == ID_EXE_read_reg_2	DatatoReg == 01	ID_EXE_mem_w=0	Memory data	MEM		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_2	DatatoReg == 01	ID_EXE_mem_w=0		WB		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == ID_EXE_read_reg_2	DatatoReg == 00	ID_EXE_mem_w=1	ALU_data	MEM		data_out
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_2	DatatoReg == 00	ID_EXE_mem_w=1		WB		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == ID_EXE_read_reg_2	DatatoReg == 01	ID_EXE_mem_w=1	Memory data	MEM		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_2	DatatoReg == 01	ID_EXE_mem_w=1		WB		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_1	DatatoReg == 00	N/A	ALU_data	MEM	ID	ALU_A
ID_EXE_written_reg != 0	ID_EXE_written_reg ==IF_ID_read_reg1	DatatoReg == 00	N/A		WB		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_2	DatatoReg == 00	N/A		MEM		ALU_B
ID_EXE_written_reg != 0	ID_EXE_written_reg ==IF_ID_read_reg2	DatatoReg == 00	N/A		WB		
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_1	DatatoReg == 01	N/A	Memory data	MEM		ALU_A
EXE_MEM_written_reg != 0	EXE_MEM_written_reg == IF_ID_read_reg_2	DatatoReg == 01	N/A				ALU_B