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

30/04/2021

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.

1. Introduction

This project was provided with a single-issue 5-stagebasic RISC-V processor [1], and a sketch of this processor structure is shown in Figure 1 [2]. 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.

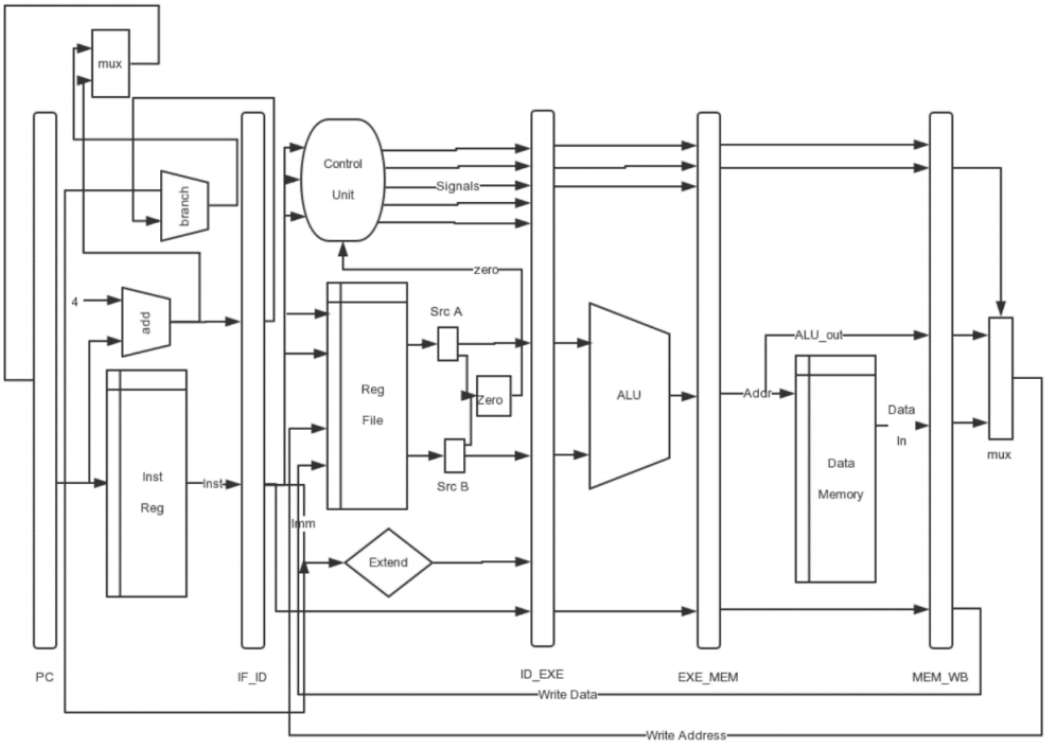

Figure 1: Sketch of provided RISC-V processor

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. Finally, a 2 way in-order superscalar 5-stage RISC-V processor would be introduced to decrease the CPI to below 1. And a dependency check module is needed to prevent data hazards.

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 cycle for each benchmark program would be recorded. Also, this project would discuss the improvement on the performance for the modified processor by comparing the execution cycle of the original single-issue 5-stage RISC-V processor and the modified 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 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 flow

Those data are forwarded from MEM stage and WB stage. By using the mux, those stages are sent the data by DF_control signal. In those stages there are two different types of data, one of the data from MEM (marked as data_in) and the other one is from ALU calculation (ALU_out). The DF_control will identify the hazard is from the Load instruction or from other types of instruction (such as add, addi, or etc.). For the ALU_B data, the design will be the same as the ALU_A data forwarding, the figure is only the stage at now.

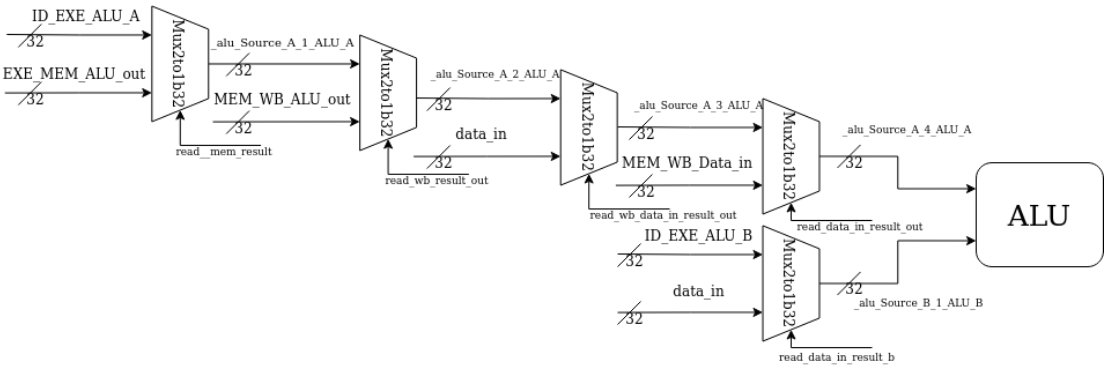

Figure 2: ALU data flow in EXE stage

2.1.2 DF_control design

The Data flow control design for the control signal to the mux. This control signal requires the read, write register address from the ID, EXE and MEM stage. In addition, using the signal DatatoReg can identify the instruction is Load or other signal. The mem_w signal is needed, because the SW instruction uses the rs2 as the destination to store, to check if the signal has any hazard the extra check for the rs2 is necessary for data forwarding.

To generate the signal for read_mem_result(ALU result from MEM stage) the comparator will compare the address from EXE and MEM stage, is the MEM rd register

and the EXE stage rs1 are the same the signal read_mem result will be high and switch the mux to take the data from ALU_reusult from MEM stage. It is similar to the read_wb_result_out but the detection is from the ID stage and the MEM stage.

For the data hazard between ID and MEM stage, the control signal(read_wb_result_out) will be stored to the shift register and held until the ID stage instruction is shifted to the EXE stage. So, a clk signal is required for the shift register.

For the signal read_data_in_result_out and read_wb_data_in_result_out are similar ideas to the ALU result forward, but those control is used to forward the Load data from MEM_wb module. Which the hazard comes from the LD instruction and other instruction (sample instruction is in the Appendix A).

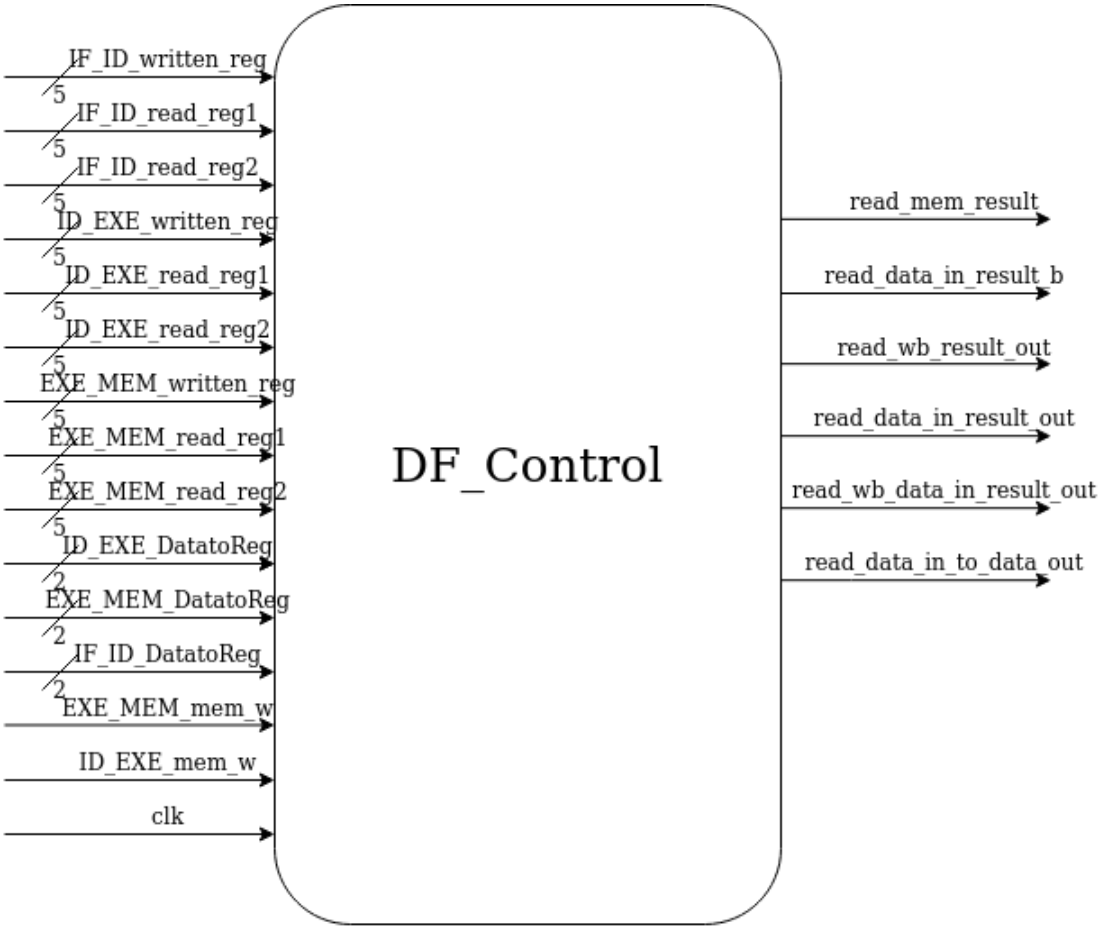

Figure 3: DF_Control overview

2.1.3 Futher design

The further design will combine the signal to be a 4 to 1 mux to have a better viewing and simpler design. Also, the data forward requires for the Branch prediction check.

Also consider the hazard for the branch prediction and jump instruction, there will also have the data forward module for those modules.

2.2 Branch predictor

From the original design, each stall would happen for each branch instruction. It is because the result for branch to be taken or not only can be obtain when the branch instruction is in ID stage. Then, branch predictor is trying to eliminate this stall by predict the branch result in the IF stage. And flush only happen when the prediction is wrong, and flush would cause one extra clock cycle to execute the program.

2.2.1 BHR_and_PHT

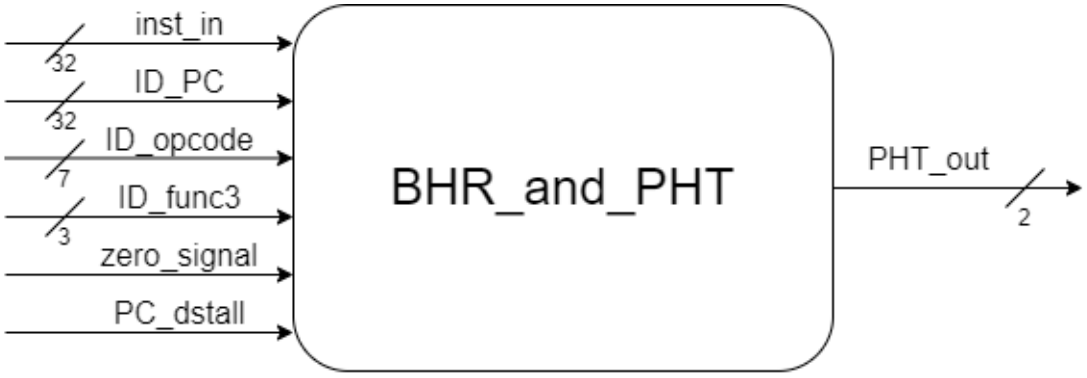

Figure 4: BHR_and_PHT overview

The figure above shows the input and output port of “BHR_and_PHT” module, and this module is used to store and update the Branch History Register (BHR) and Pattern History Table (PHT). “inst_in”, “ID_PC”, “ID_opcode”, “ID_func3”, “zero_signal”, “PC_dstall” are the input signal of this module. “PHT_out” is the output of this module.

“inst_in” is a 32-bit instruction from Instruction Fetch (IF) stage. “ID_PC”, “ID_opcode”, “ID_func3” and “zero_signal” means the PC in ID stage, opcode in ID stage, func3 in ID stage and zero signal generate by the zero generator in ID stage respectively. PC_dstall is the signal that is used to stall the module and not continue to execute when data hazard occurs and before data forwarding is introduced. 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 left for 1 bit. 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 16 entries and each entry stores content of 2-bit Branch Predictor and the 2-bit Branch Predictor uses the Saturating Up/Down scheme. Also, 4-bit BHR is referred to the address of the PHT 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, in order to update the content of PHT and BHR, the content of PHT would be read at the rising edge before the BHT has been updated. After that, the content of PHT is updated at the falling edge and stored at the location where it is read from. Also, BHR also would be updated after the falling edge. Ans this method can effectively read and update the BHR and PHT within 1 clock cycle.

2.2.2 IF_MUX_sel

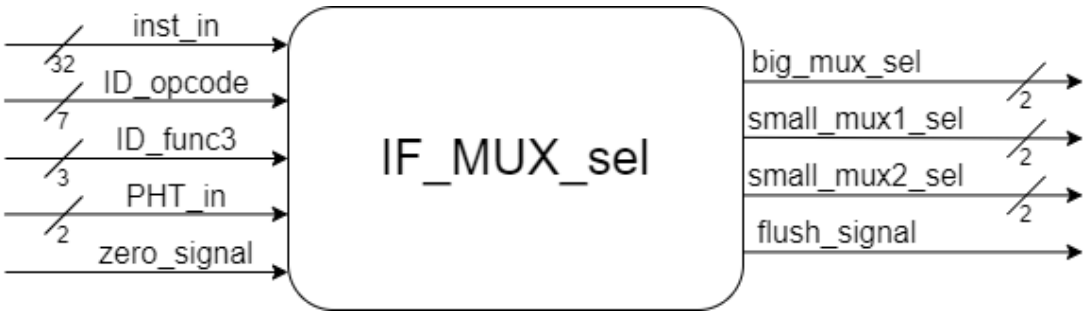

Figure 5: IF_MUX_sel overview

The figure above shows the input and output port of the "IF_MUX_sel" module, and this module is used to select the correct value for the program counter (PC) 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 and 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, which are the values that need to add them together to become the updated PC value. So, the control signal of the 2 MUX depends on the predicted result and changes the control signal of the MUX and flush

signal become logic “1” if the prediction is wrong. Furthermore, the instruction in IF and ID stage will determine the final update value of PC by controlling the 4-to-1 MUX. And the figure below shows the connection of the MUX and the “IF_MUX_sel” module.

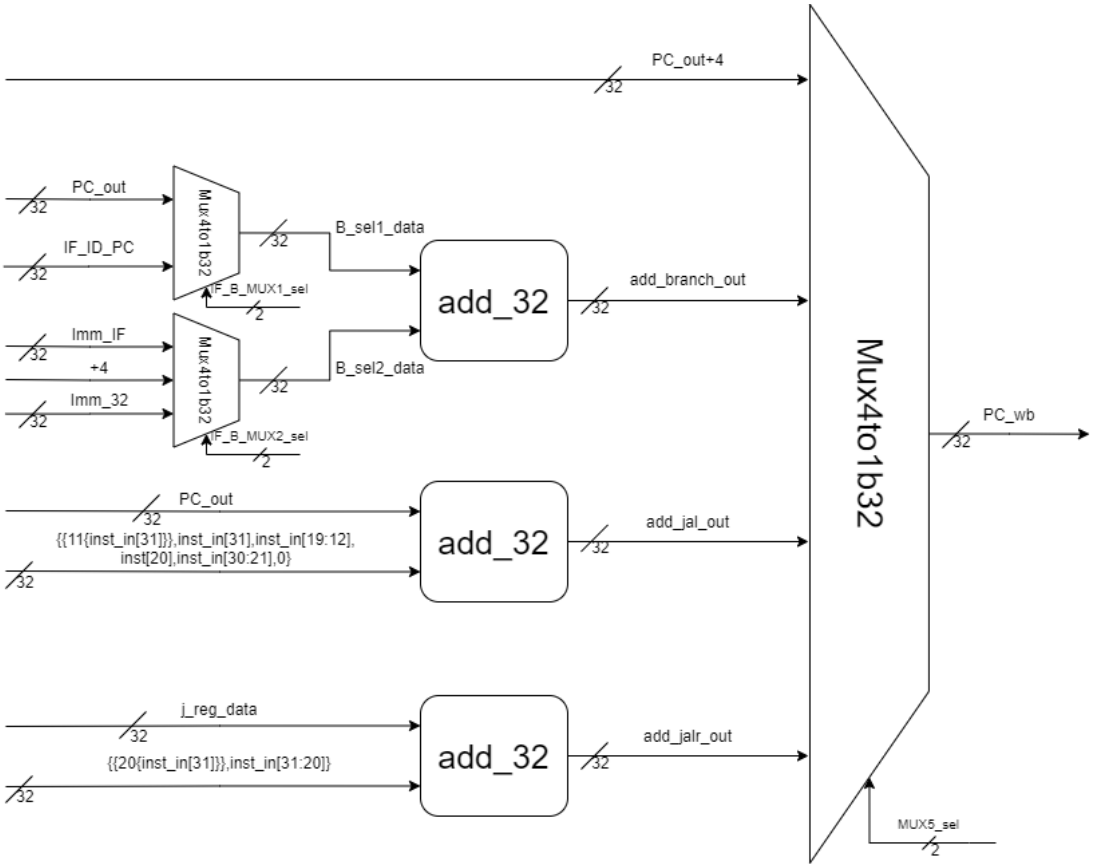

Figure 6: Connection of 4-to-1 MUX in IF stage

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 would add to the immediate value of the instruction in IF stage. When a branch is predicted as not taken, then current PC just added by 4 and executes in sequence. Also, 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 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.

In addition, according to Figure 6, when “IF_B_MUX1_sel” is 0 means select “PC_out” and 1 means select “IF_ID_PC”. Also, when “IF_B_MUX2_sel” is 0 means select “Imm_IF”, 1 means select positive integer 4, 2 means 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 discussed above. For example, both “IF_B_MUX1_sel” and “IF_B_MUX2_sel” are 0 when a branch is predicted as taken.

3. Intermediate/Preliminary Results

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.

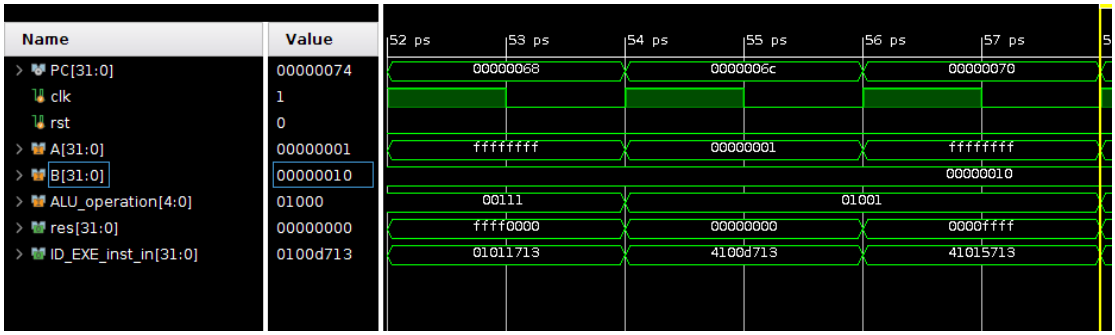

Figure 7: Simulation with the Appendix A instruction

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.

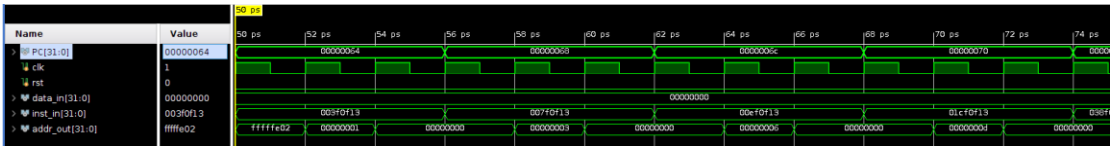

Figure 8: Simulation from the original design (using Appendix B program)

The Figure 9 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.

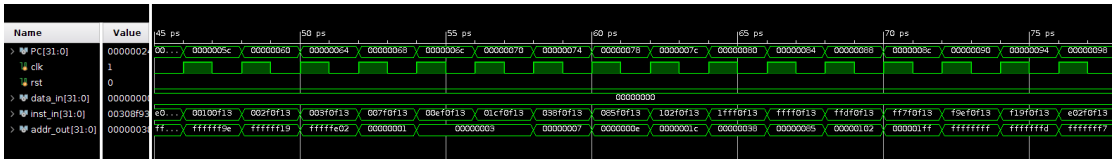

Figure 9: Simulation after adding the data forwarding design (using Appendix B program)

3.3 BHR_and_PHT

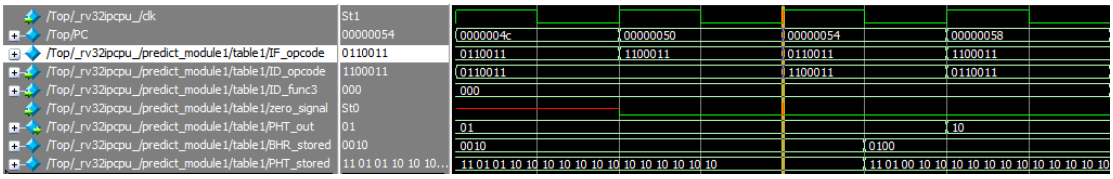

Figure 10: Simulation result when prediction is correct (using Appendix C program)

The waveform above shows an example of the program execution when prediction is correct. And the assembly code for this program is provided in Appendix C. According to Appendix C, a branch equal (BEQ) instruction is read when PC is 0x50. From the figure above, it also shows that the branch is predicted as not taken because the PHT is 01 when PC is 0x50. After 1 clock cycle later, 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 10 is logic “0”, so the branch should not be taken for instruction at PC equal to 0x50. So, the prediction for instruction when PC is 0x50 is correct. Furthermore, after the actual branch result is known, BHR and content of PHT has been updated from “0010” and “01” to “0100” and “00” respectively. As a result, the “BHR_and_PHT” module operate as expected, and the program is executed in sequence as expected which can be known by observing the PC value.

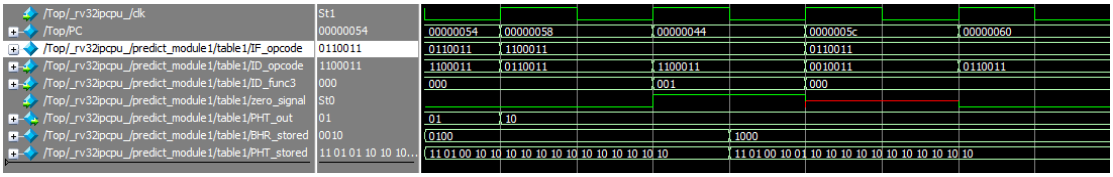

Figure 11: Simulation result when prediction is wrong (using Appendix C program)

The waveform above shows an example of the program execution when prediction is wrong. According to Appendix C, a branch not equal (BNE) instruction is read when PC is 0x58. From the figure above, it also shows that the branch is predicted as taken because the PHT is “10” when PC is 0x58. After 1 clock cycle later, “zero_signal” is logic “1”, so the branch should not be taken for instruction at PC equal to 0x58. Then,

the prediction for instruction when PC is 0x58 is not correct. It means the program should be executed in sequence after PC is 0x58. Also, after the actual branch result is known, BHR and content of PHT has been updated from “0100” and “10” to “1000” and “01” respectively. Consequently, the “BHR_and_PHT” module operates as expected, and the value of PC recovers back to 0x58+4, which is 0x5C, as expected.

3.4 IF_MUX_sel

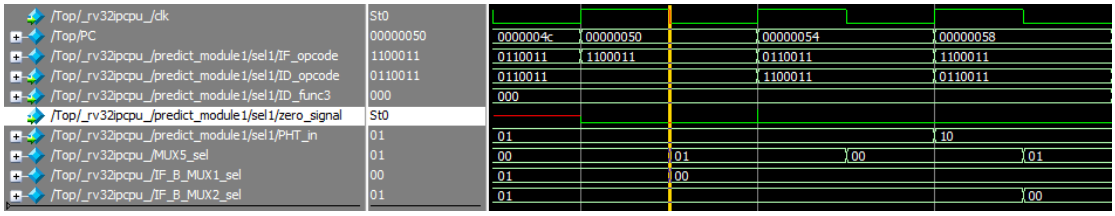

Figure 12: Simulation result for MUX select signal when prediction is correct

(using Appendix C program)

The waveform above is the same scenario as the Figure 12 shows. In other words, the branch is predicted as not taken and the prediction is correct. Also, according to Figure 12, the select signal “IF_B_MUX1_sel” would be 0 and the select signal “IF_B_MUX2_sel” would be 1 when the branch predicted was not taken and executed in sequence. And the waveform shows that the “IF_MUX_sel” operates as expected.

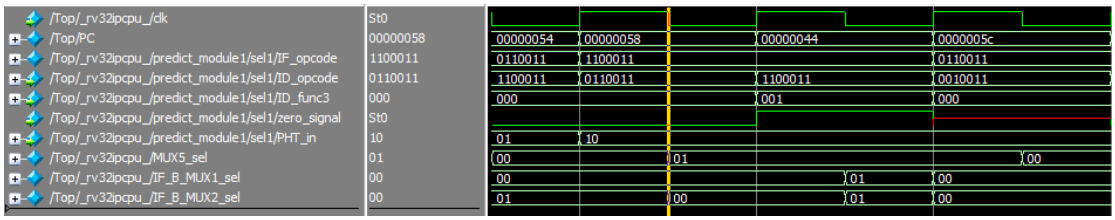

Figure 13: Simulation result for MUX select signal when prediction is wrong

(using Appendix C program)

The waveform above is the same scenario as the Figure 13 shows. In other words, the branch is predicted as taken and the prediction is not correct. Then, according to Figure 13, the select signal for both “IF_B_MUX1_sel” and “IF_B_MUX2_sel” would be 0 and jump to a calculated value of PC after PC is 0x58. After that, PC jumps to 0x44 but the actual branch result indicates that the prediction is wrong. Next, PC recovery is needed and recovered back to 0x58+4. And now, the select signal for both “IF_B_MUX1_sel” and “IF_B_MUX2_sel” would be 1. According to the waveform above, it shows that the “IF_MUX_sel” operates as expected.

4. 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].

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
```