SME309 Final Project Report: ARMv3 Pipeline Processor with Extended Functions and Branch Prediction

Haihan Wu 12011419, Yuhang Wang 12111328

Yimao Liu 12110618, Feiqiao Bu 12111803

Jan 2024

Project Overview

In this project, we implemented an ARMv3 pipeline processor supporting 16 DP instructions, a 4KB cache, a multiplier, an FPU, and a branch predictor. A single-cycle RISC-V processor is implemented and compared with the ARMv3 processor. The structure of this report is given in table 1, and table 2 provides our division of labor and contribution percentage. "Running Vivado on Mac M1" tutorial is provided in Appendix C.

Section	**Problem**	**Content**
Part I	Problem 1 and 3	Baseline pipeline CPU and ALU
Part II	Problem 4	Cache
Part III	Problem 2 and 5	FPU and non-stalling multi-cycle instructions
Part IV	Problem 6	single-cycle RISC-V CPU
Part V	Bonus	Branch prediction

Table 1: Structure of this report.

Name	**Percentage**	**Task**
Haihan Wu	27%	System integration, cache, RISC-V, and branch prediction
Yuhang Wang	25%	ALU, hazard analysis, and assembly code design
Yimao Liu	24%	Non-stalling multi-cycle function, single-cycle FPU, multi-cycle FPU, and FPU-MUL ARM integration.
Feiqiao Bu	24%	

Table 2: Labor division and contribution percentage.

Contents

List of Figures

Part I: Baseline Pipeline Processor

1.1 ALU with 16 Data-processing Functions

Based on lab2, we extend the functionality of ALU. We need four bits ALU-control signal to indicate the operation. We need to add a c-ALU signal to give a correct carrier bit to ALU in 16 different functions. In order to ensure that our adder is compatible with subtraction, we set c-ALU in subtraction operation to simulate add-one operation of complement. As for operations with carry, we extract C flag from control unit to act as the carry input of ALU.

Table 3-2 Data-processing instructions

Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	Rd := Rn AND shifter_operand
0001	EOR	Logical Exclusive OR	Rd := Rn EOR shifter_operand
0010	SUB	Subtract	Rd := Rn - shifter_operand
0011	RSB	Reverse Subtract	Rd := shifter_operand - Rn
0100	ADD	Add	Rd := Rn + shifter_operand
0101	ADC	Add with Carry	Rd := Rn + shifter_operand + Carry Flag
0110	SBC	Subtract with Carry	Rd := Rn - shifter_operand - NOT(Carry Flag)
0111	RSC	Reverse Subtract with Carry	Rd := shifter_operand - Rn - NOT(Carry Flag)
1000	TST	Test	Update flags after Rn AND shifter_operand
1001	TEQ	Test Equivalence	Update flags after Rn EOR shifter_operand
1010	CMP	Compare	Update flags after Rn - shifter_operand
1011	CMN	Compare Negated	Update flags after Rn + shifter_operand
1100	ORR	Logical (inclusive) OR	Rd := Rn OR shifter_operand
1101	MOV	Move	Rd := shifter_operand (no first operand)
1110	BIC	Bit Clear	Rd := Rn AND NOT(shifter_operand)
1111	MVN	Move Not	Rd := NOT shifter_operand (no first operand)

Figure 1: 16 Data-Processing Instructions.

The simulation result of ALU is shown in Figure 2.

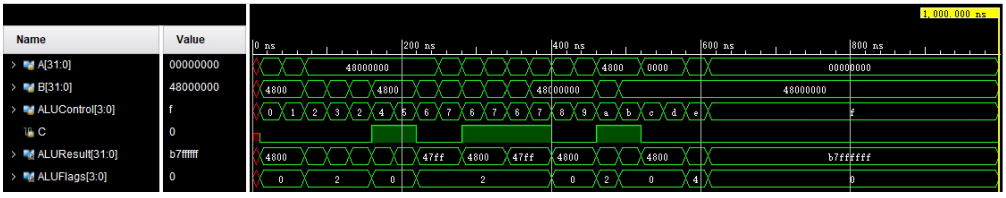

Figure 2: Simulation result of ALU.

1.2 Baseline Design Verification

1.2.1 Hazard Assembly Design

We neglect cache miss and long-cycle instructions like multiplication, division and float-point operation and mainly focus on data-operating, memory and branch instructions. Data

hazard and control hazard will occur in this five-stage pipeline as we separate the instruction memory and data memory. Four types of data hazard including one special case and one type of control hazard are included in our assembly file.

DP Data Hazard: as shown in Figure 3, R1 is used three times in following three instructions. The first two can be resolved by data forwarding and the last one can be handled by using different clock edges.

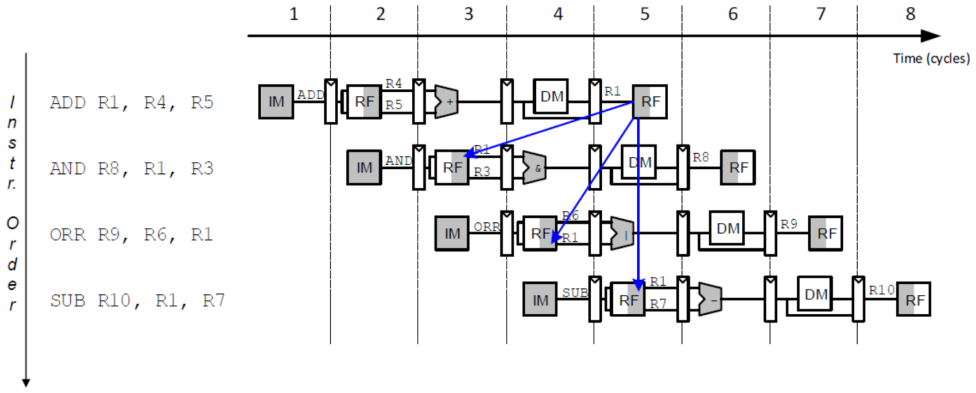

Figure 3: DP Data Hazard.

Instead of using different clock edges to update the register file (RF), whose latency is large and would decrease the maximum clock frequency, a different strategy is used for the data-forwarding. If the RF write is enabled and RA1/RA2 is equal to the WA3, then WD3 will be directly loaded as the RD1/RD2 output. So the data hazard in the 5^{th} clock cycle of Figure 3 is resolved.

Load and Use: as shown in Figure 4, R1 is used after LDR instruction. Since LDR instruction must use Mem stage, this leads to impossible data forwarding. This problem is solved by stall and flush.

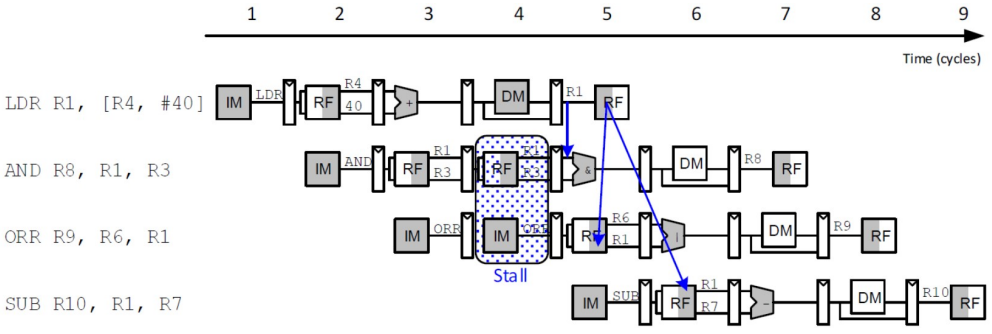

Figure 4: Load and Use Hazard.

Memory-memory Copy: as shown in Figure 5, R1 is used after LDR instruction. But as R1 is used in mem stage of the next instruction. This case can be solved by data forwarding, which is a contrast to Load and Use.

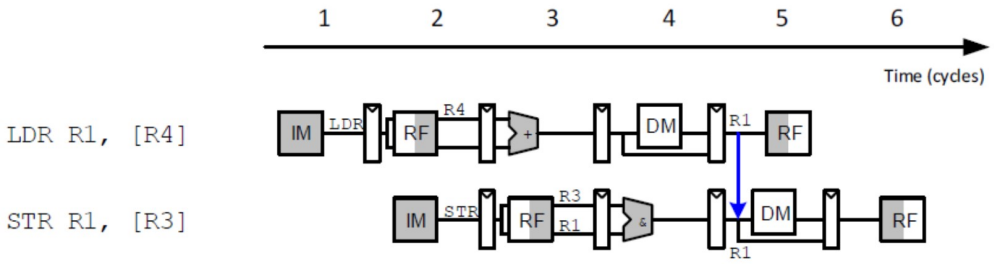

Figure 5: Mem-mem Copy Hazard.

Special case: as shown in Figure 6, DP instruction prior to STR instruction is an interesting case as it can only be solved by forwarding data from mem stage to exc stage under the hazard unit informed in theory lectures, which can be included to verify the robustness of our system.

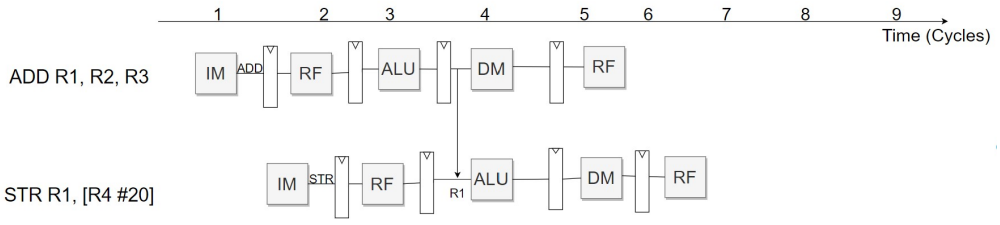

Figure 6: Special Hazard.

Control Hazard: as shown in Figure 7, control hazard happens as we can not predict whether the branch should be taken or not and what BTA is. Control hazard can be handled by early BTA and branch prediction.

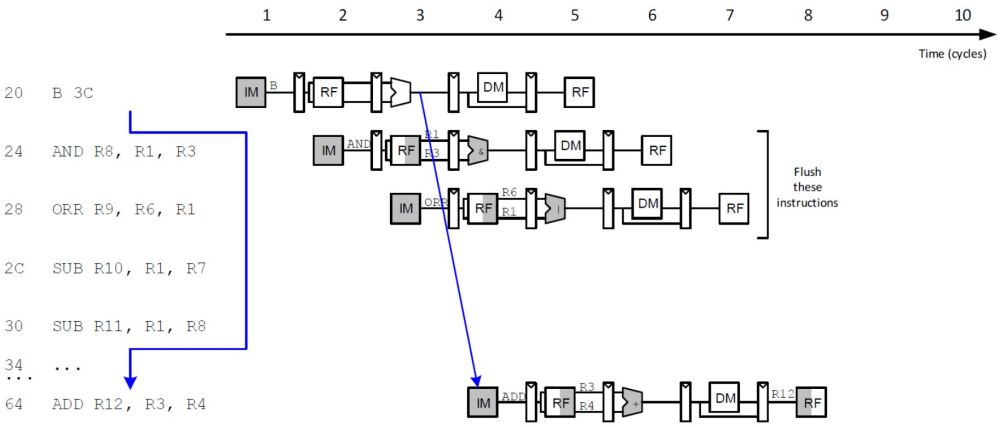

Figure 7: Control Hazard.

1.2.2 Simulation Result

The Problem1.s assembly program in Appendix B.1 is translated to the machine language and loaded in the Wrapper module. It includes 16 ALU functions and all hazards mentioned above.

The baseline processor passes the simulation because the R10 value becomes 0x41 at 435 ns (Figure 8). However, the signals are still active after the program execution, and the PC values are repeating in a manner of 156 → 160 → 164 → 156 → ... in Figure 9. This effect will significantly increase the CPU power dissipation. In Part 5.2, we found that the dynamic branch prediction could eliminate this issue.

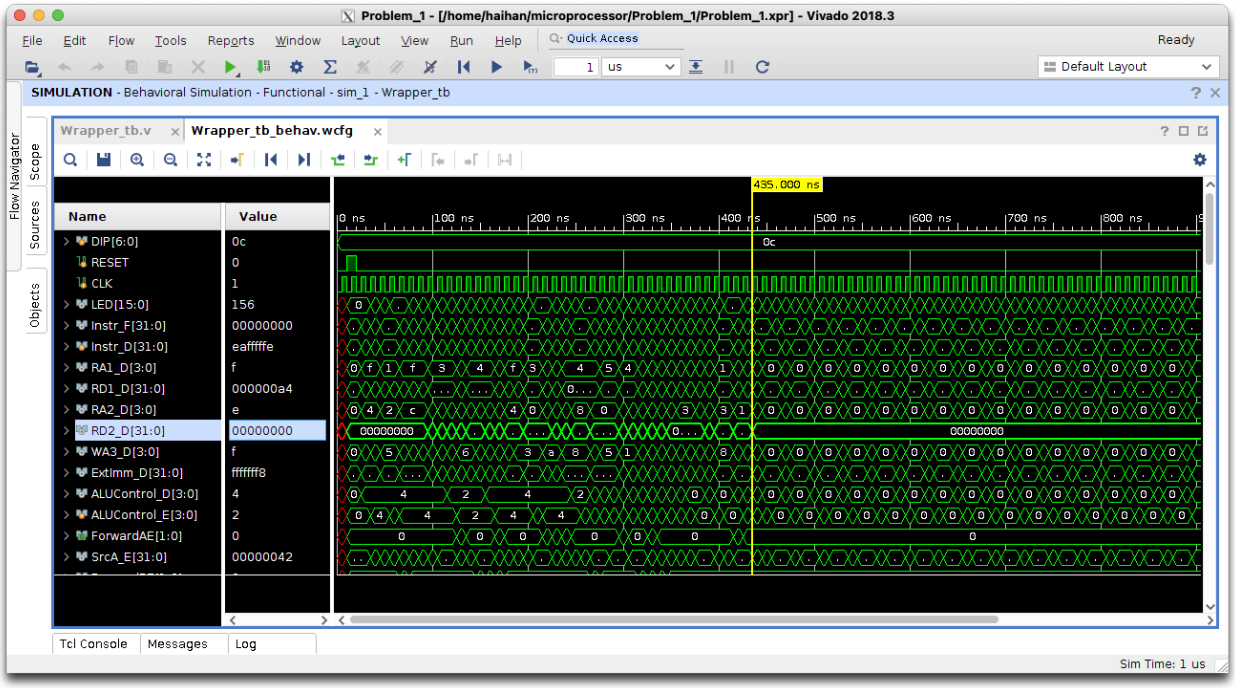

Figure 8: The baseline pipeline processor simulation waveform. The yellow line indicates the finishing point, however, the signals are still flipping after the execution.

Figure 9: Even though the program finishes at 435 ns, PC values (the same as LED output) are repeating endlessly between 156, 160, and 164.

Part II: Cache Implementation

2.1 Four-way Associative Cache Architecture

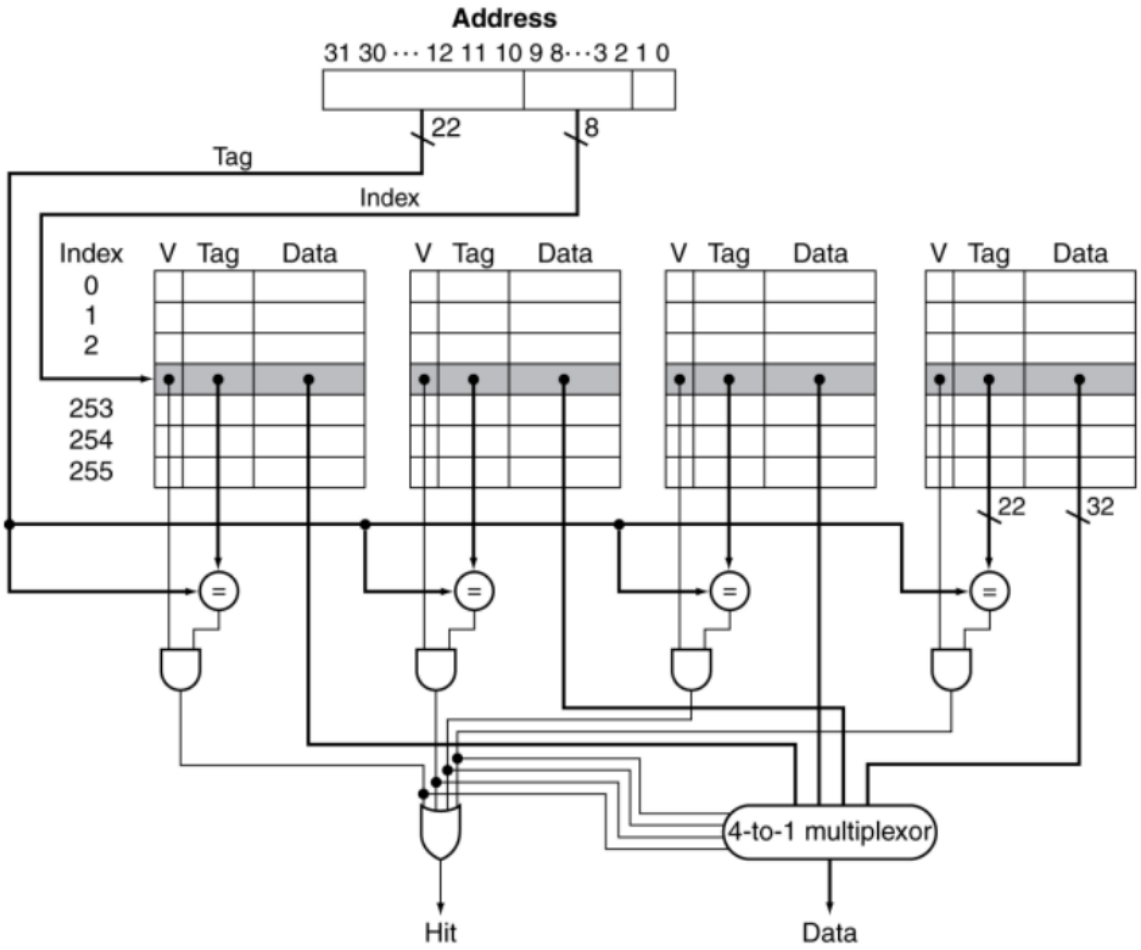

Figure 10: The block diagram of 4-way and 256-line associative cache.

Figure 10 shows the hardware architecture of a 4-way and 256-line associative cache supporting write-back and write-allocate. If the cache is hit, then the CPU directly reads from the cache or writes data in one clock cycle. If a read miss occurs, the cache will fetch the memory block from the main memory and send it to the processor. This requires 2 clock cycles.

As to the write miss, the corresponding block will be loaded to the cache according to the write-allocate mechanism and the CPU data will be stored in this cache block. There are four cache blocks in one line, so a replacement policy is added to determine which block to occupy. The empty cache blocks will be occupied first, then the clean blocks. If all the blocks are dirty, write back the first block to the main memory then and load the new blocks.

The cache is connected to the CPU and main memory according to Figure 11. In the CPU-cache interface, the R/W' is activated and the pipeline is stalled once **cpu_valid** is asserted.

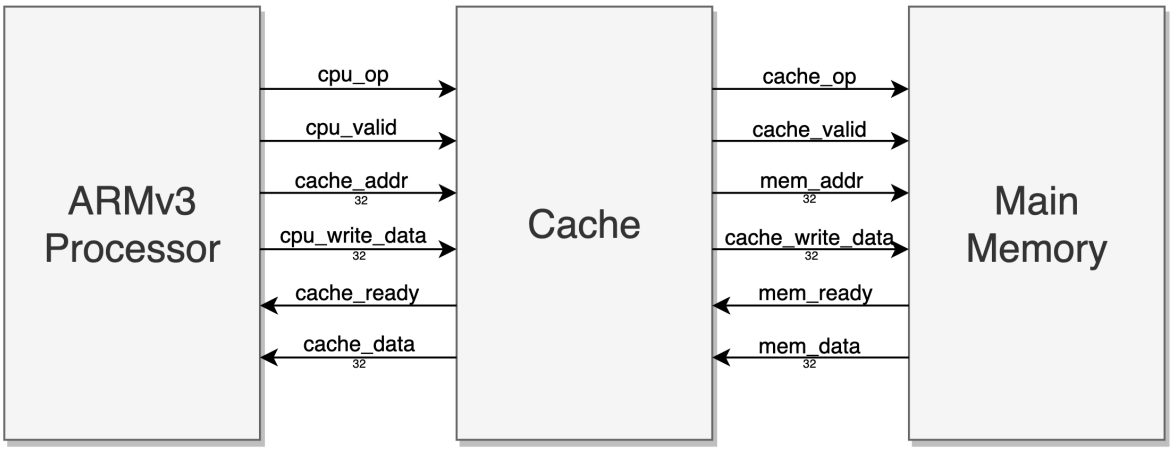

Figure 11: The cache connection to the CPU and main memory.

If **cache_ready**=1, the ARM core will load the cache data or confirm the cpu_write_data is loaded to the cache, so the CPU will wake up and execute the next instruction. The data communication protocol is the same at the interface between the cache and main memory.

All the functions above are verified by tb_SA_Cache.v. It passed the simulation and the waveform is give in Figure 12.

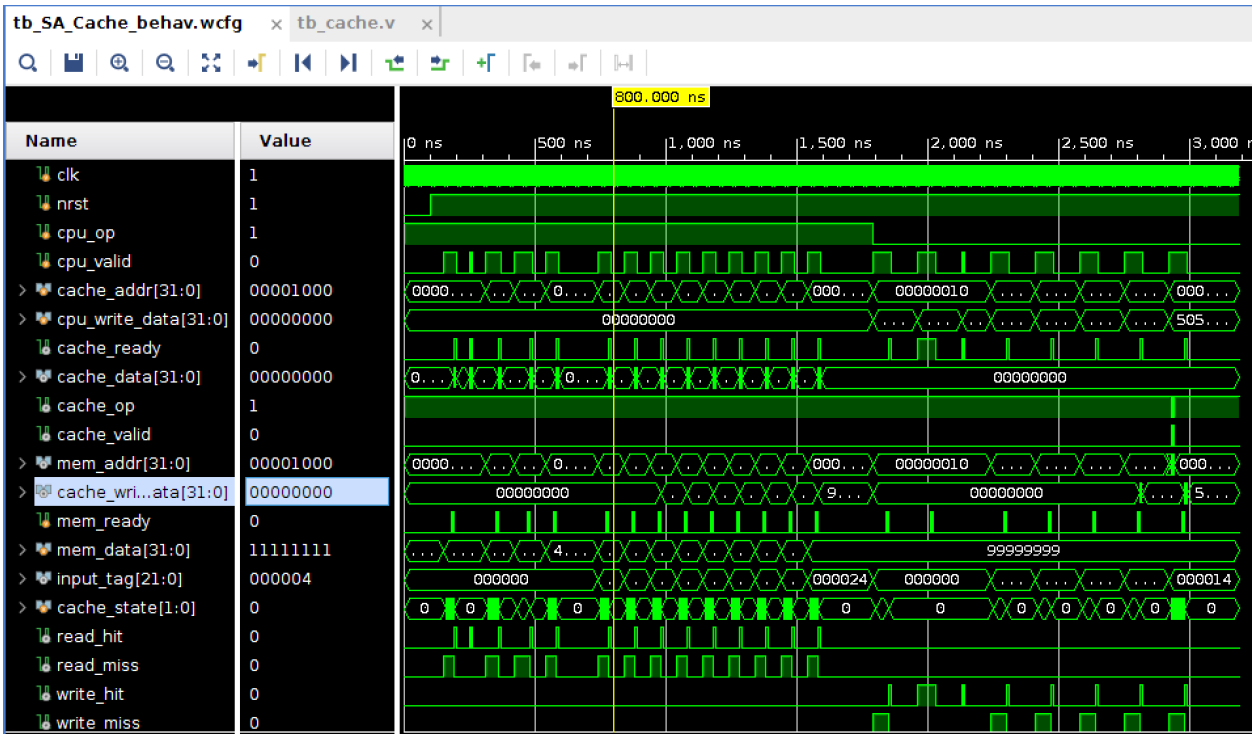

Figure 12: The simulation of cache.

2.2 CPU Verification with Cache

2.2.1 Cache Miss Hazard

We insert the cache into our system. So we need to consider the stalling due to cache miss. Such stalls will cause cache miss hazards. We have summarized four types of cache miss hazards.

The miss-solving mechanism of cache is important for us to dessolve cache hazards. A state machine of cache is shown in Figure 13. The state machine will be IDLE in case of read hit or write hit. It will be switched to "Write back" and changed into "Write allocate" after finishing data writing to memory if the block is dirty. It will return to IDLE after the required block is fetched to the cache.

The cache can't be read and written at the same time. This will cause more severe data hazards and even structure hazards. The waveform of three cache control signals, cpu_op, cpu_valid, and cache_ready, will also be exhibited to illustrate the whole process.

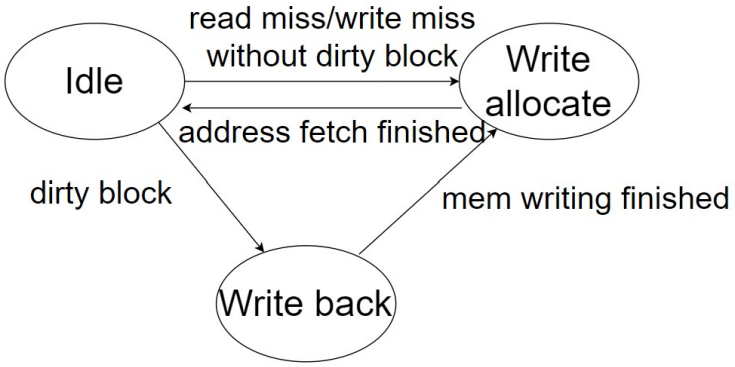

Figure 13: The state machine in cache control.

Read miss hazard: data cache read miss will result more cycles to finish LDR instructions. As the required data may be used in exe stage or mem stage. We can further divide them into two categories.

Read miss hazard with exe stage use: as shown in Figure 14, the required data may be used in exe stage like load and use and STR instruction memory address calculation. And the LDR stall will vary from one cycle to two cycles, depending on whether the required block is dirty or not.

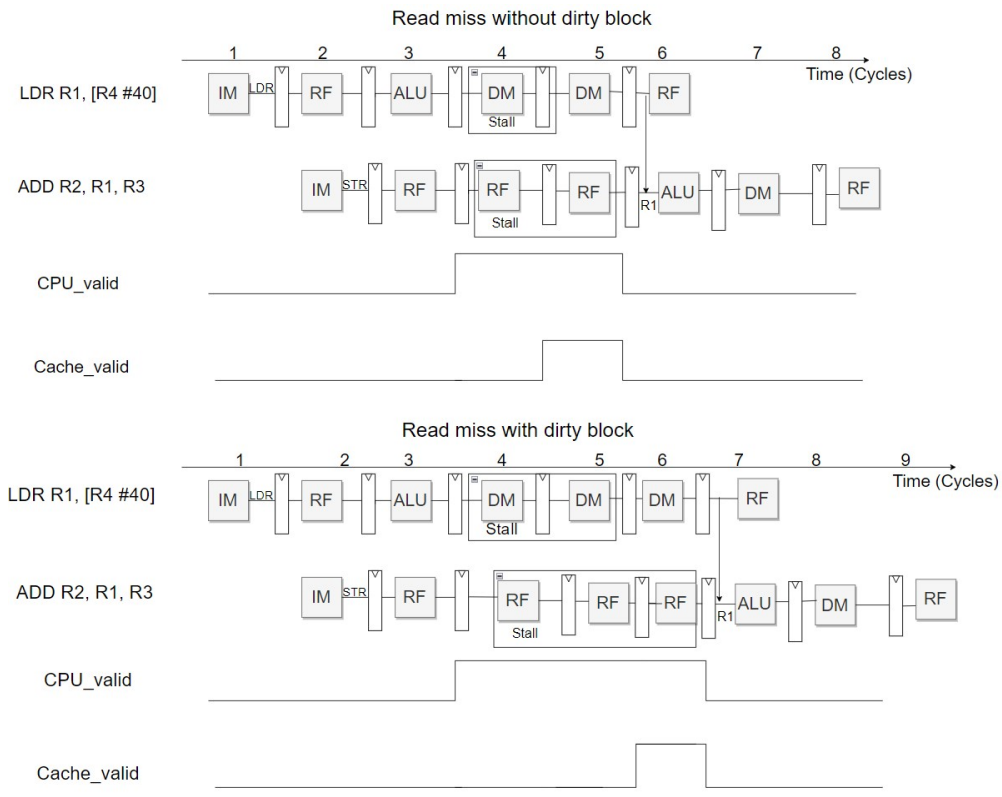

Figure 14: read miss hazard with E stage.

Read miss hazard with mem stage use: as shown in Figure 15, the required data may be used in mem stage like mem-mem copy which means the required data will be written to data cache.

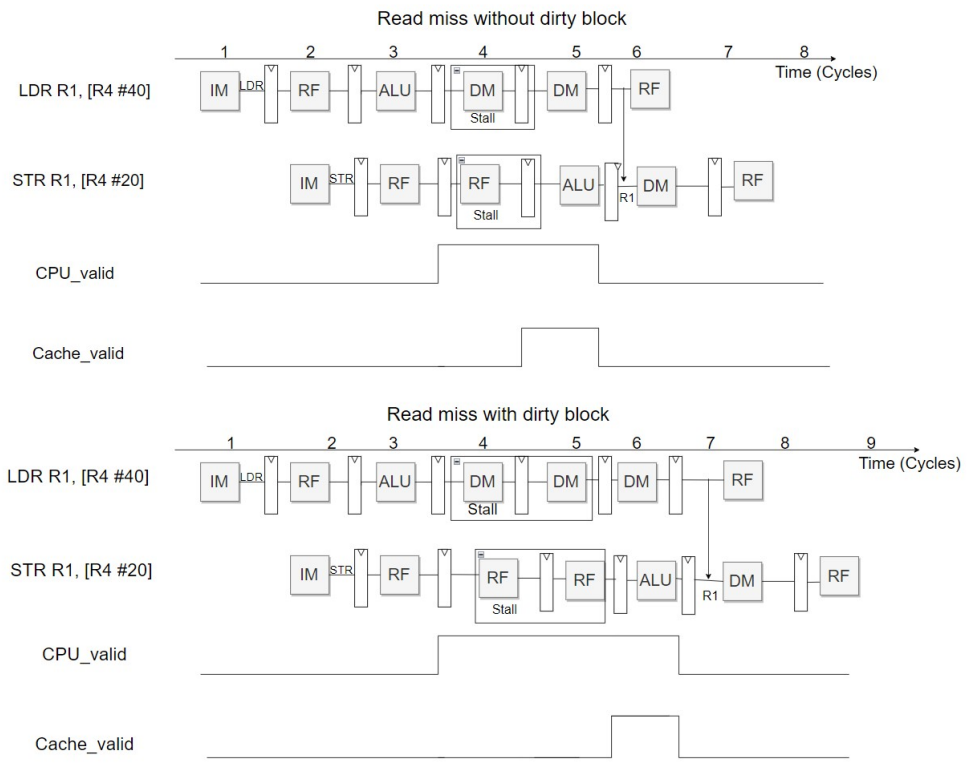

Figure 15: read miss hazard with M stage.

Write miss hazard: data cache write miss will result more cycles to finish STR instructions. The write miss may cause two cycles or three cycles use of data cache due to its write-back mechanism. Strictly, this is not data hazard. This can be thought of as a structure hazard with multiple instructions using data cache at the same time.

Write miss hazard with DP instructions: as shown in Figure 16, the DP instructions after STR will not cause write miss hazard as they never use data cache. So if the instructions after STR are all DP instructions. STR and DP operations can be implemented in parallel. Our cache has specialized unit to detect this case and realize them at the same time.

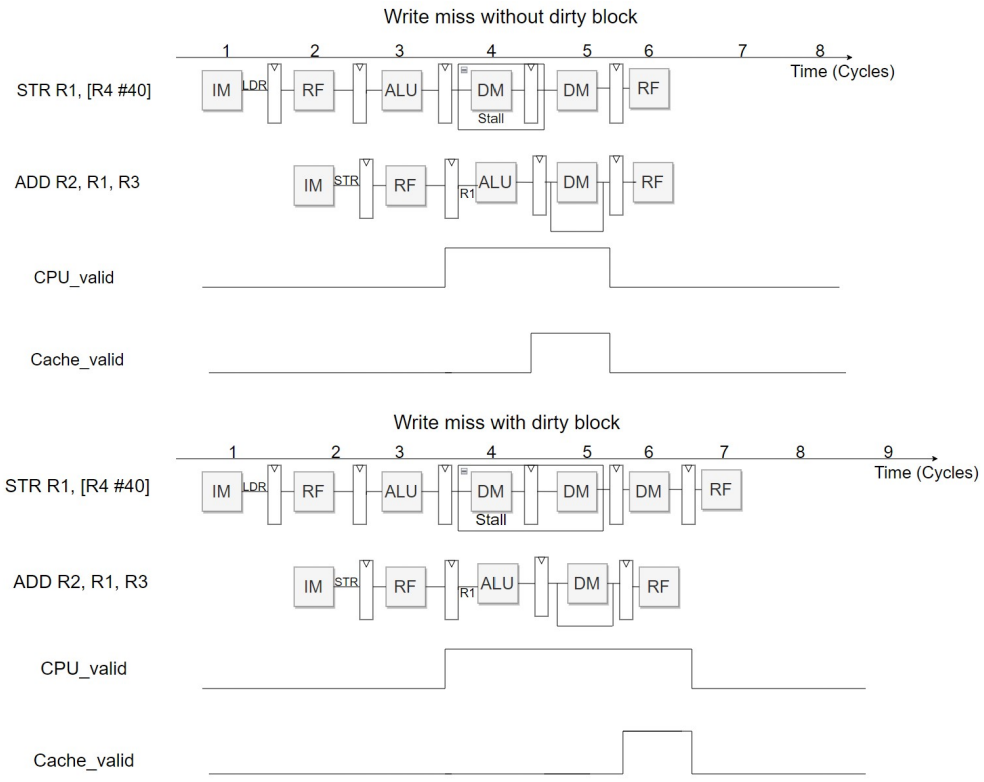

Figure 16: write miss hazard with DP instructions.

Write miss hazard with LDR instructions: as shown in Figure 17, the LDR instructions after STR will read data in data cache. As mentioned above, this is a structure hazard as data cache is used by LDR and STR at the same time. In our cache, we solve this problem through use of stall. Besides, the stall resulted from STR maybe three cycles. The case in Figure 18 will also cause write miss hazard.

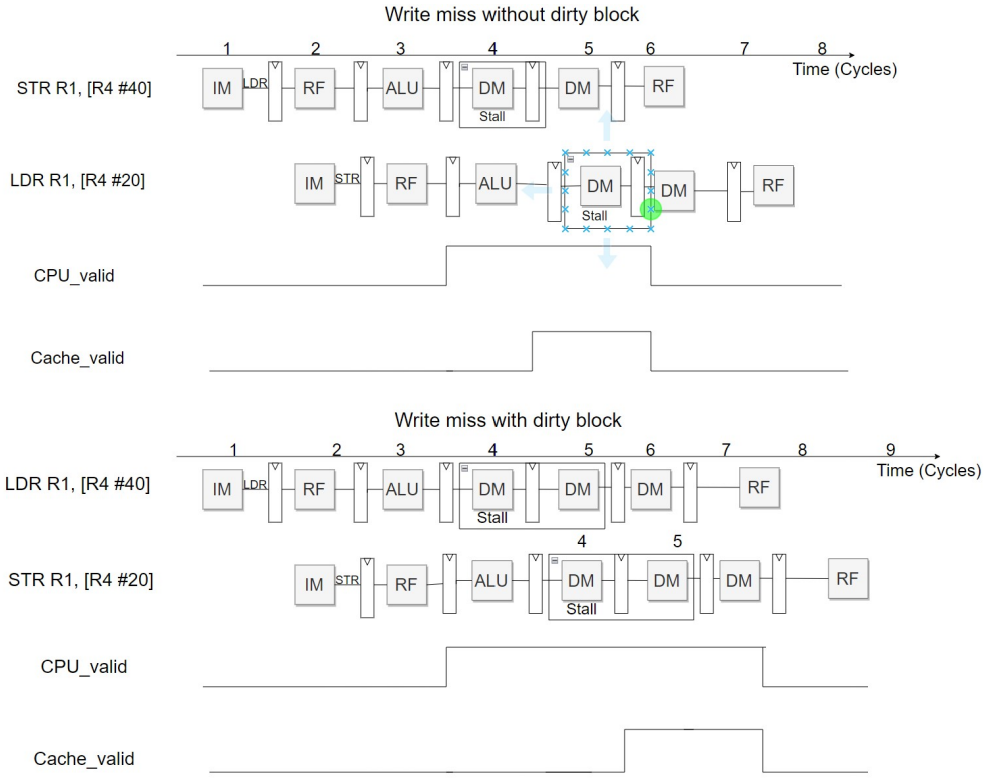

Figure 17: write miss hazard with LDR instructions.

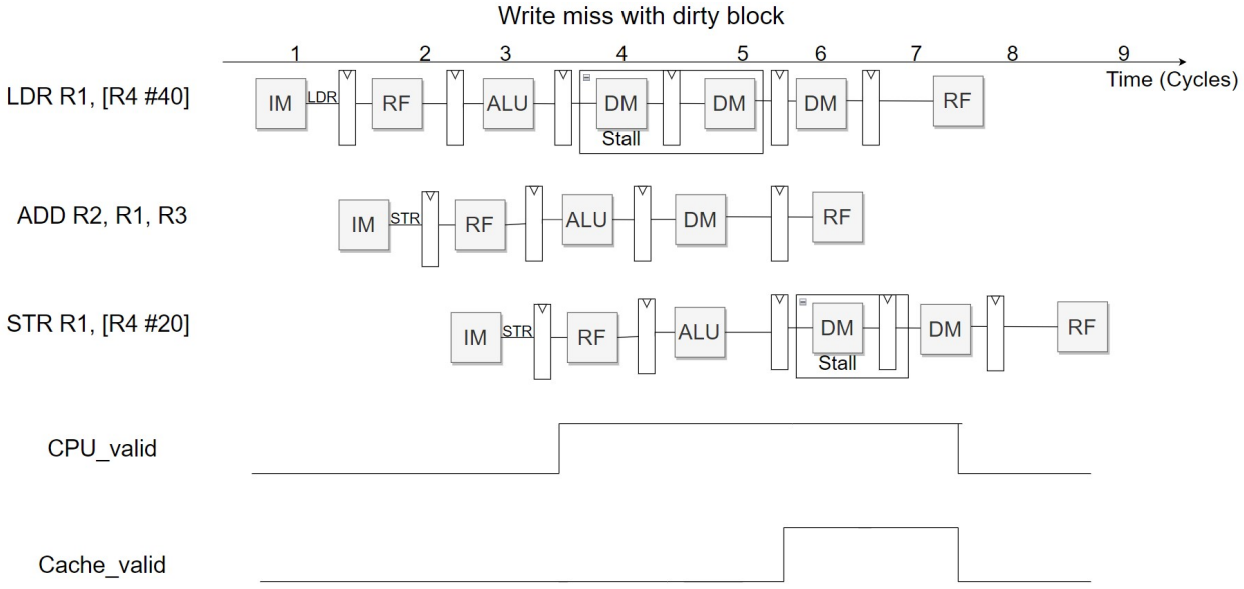

Figure 18: write miss hazard special case.

It seems that data forwarding is an effective way to solve data hazards. However, as shown in Figure 19, data forwarding will be invalid due to the addition of cache. In mem-mem copy

without cache, we solve data hazard by forwarding data from WB stage to Mem stage under the hazard unit as shown in Figure 5. But if a write miss is caused by STR instruction, it will force the cache to switch from state IDLE to state WriteBack or state WriteAllocate. So, the CPU can not write the forwarded data into the cache directly.

To solve this hazard, we can utilize the state register between Exe stage and Mem stage. Data stored in the register will decrease the stalling cycles. As shown in Figure 20, we forward data from WB stage to Exe stage, then it will stay in register and wait for cache controlling signal. We stall STR instruction in Decode stage to ensure the feasibility of this special kind of data forwarding.

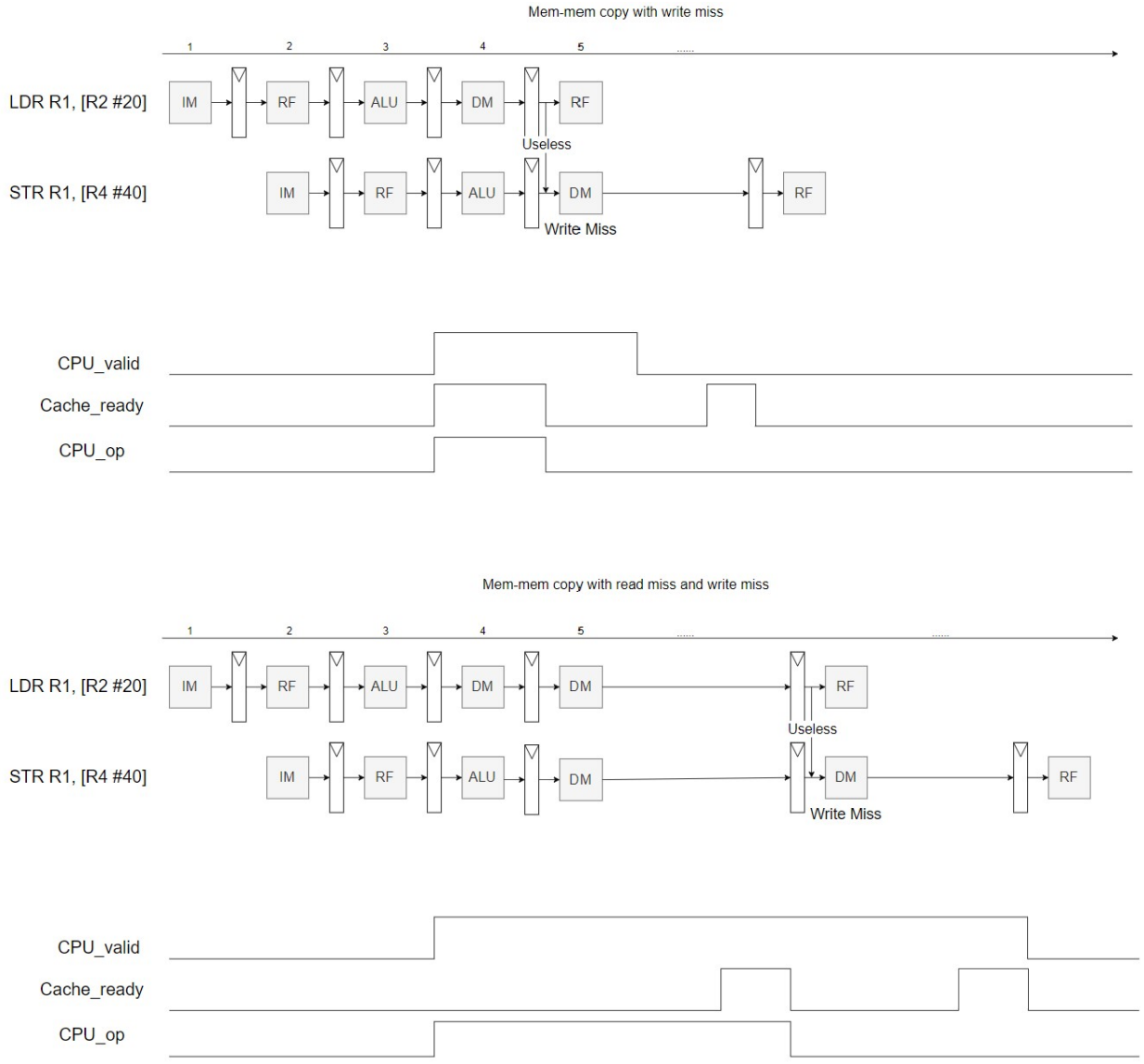

Figure 19: mem-mem copy with invalid data forwarding.

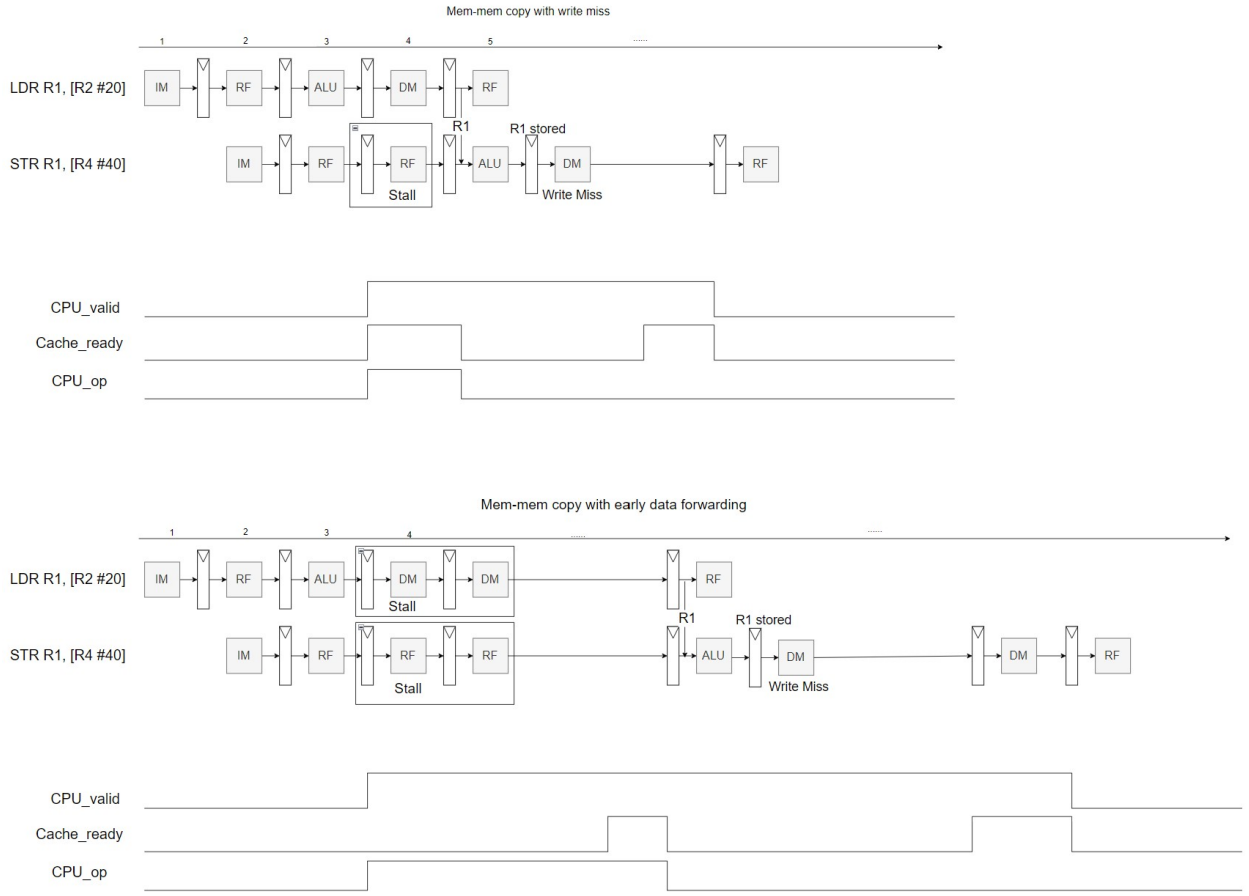

Figure 20: mem-mem copy with early data forwarding.

2.2.2 Assemble Design and Simulation Result

The baseline ARM core is connected to the 4-way associative cache in the Wrapper module. In order to check all the hazards analyzed above, the assembly program in Appendix B.5 covers all the corners.

Finally, the ARM-cache system passed the simulation. The result is provided in Figure 21.

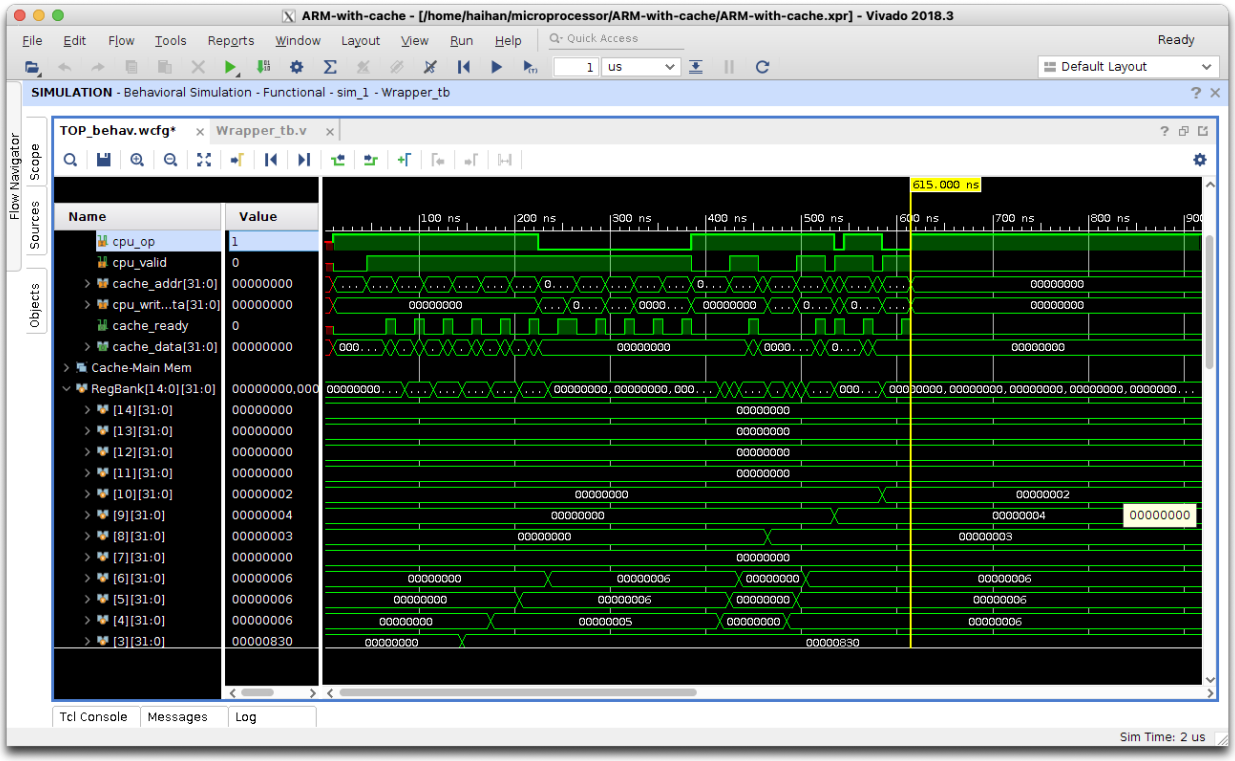

Figure 21: The simulation results of the ARM core with cache.

Part III: Multipliers and Floating-Point Unit

3.1 Floating-Point Unit Design Strategy

Number	Sign	Exponent	Mantissa
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

Figure 22: Special cases of FPU.

3.1.1 Single-cycle FPU Design

The floating-point module is divided into floating-point addition and subtraction and floating-point multiplication. In this module, multiplication uses combinational logic circuits, therefore it is a single cycle floating-point module. According to Figure 23, output NaN in the following situations.

返回NaN的运算有如下三种：

- 至少有一个参数是NaN的运算
- 不定式
  - 下列除法运算：0/0、∞/∞、∞/−∞、−∞/∞、−∞/−∞
  - 下列乘法运算：0×∞、0×−∞
  - 下列加法运算：∞ + (−∞)、(−∞) + ∞
  - 下列减法运算：∞ - ∞、(−∞) - (−∞)

Figure 23: The situation where the output is NaN.

Due to the fact that this module does not include division, the situation of division operation is ignored. This module marks special inputs that may cause NaN output, directly determines special values, and can directly output NaN. In this module, all bits of the exponent and mantissa are set to 1 as NaN. Firstly, split the input into sign bits, exponents, and mantissa, as shown in Figure 24. And expand the mantissa number to 24 bits, with the highest bit being 1.

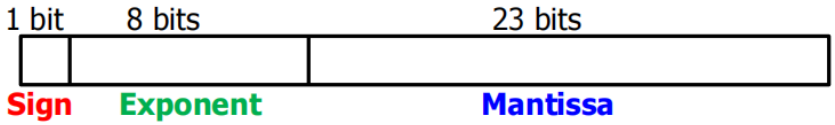

Figure 24: Format of Floating Point Numbers.

In the calculation of floating-point addition and subtraction, the difference between two exponents is first obtained, and then the mantissas of floating-point numbers with smaller exponents are shifted to the right by calling the shifteradd module. The amount of right shift is the difference between the two exponents. After aligning the mantissas, addition and subtraction are performed by judging the sign bit, where subtraction is achieved by taking the inverse addition.The new sign bit is the sign bit of a floating-point number with a larger mantissa. After the above process, we will obtain new sign bits, exponents, and mantissas, which are csign, cExponent, and cMantissa, respectively. Then, based on the first two digits and sign bits of cMantissa, determine the changes in cMantissa and cExponent, and obtain the final result. The judgment is divided into two blocks, whether the symbols of the two inputs are the same.Same situation: If the highest bit of cMantissa is 1, it is a carry, cExponent needs to be incremented by 1, and cMantissa needs to be shifted one bit to the right by calling shift2. Otherwise, cExponent and cMantissa will be the final results. Different situations: If cMantissa [23:0] is 0, the final result is 0. Otherwise, obtain the position of the first 1 of cMantissa as one (the position is 0 when the highest bit is 1), subtract one from cExponent, and call shift2 to obtain the new Mantissa, taking [29:7] to obtain the final result.

In floating-point multiplication, the sign bit of the final result is the difference between the sign bits of two inputs. Then obtain mulExponent and mulMantissa. MulExponent is the sum of the exponents of two inputs minus 127. MulMantissa is obtained by calling the combination logic MUL with two extended input Mantissa. Then it is to obtain the true result. If the mulExponent exceeds the exponential range, output NaN directly. Then determine the highest position of mulMantissa and mulExponent. If the highest bit of mulMantissa is 1 and mulExponent is not equal to 255, then mulExponent is increased by 1, and mulMantissa [46:24] is the exponent and Mantissa of the final result, respectively. If the highest bit of mulMantissa is 1 and mulExponent is equal to 255, it indicates result overflow, and the final result is NaN. If the highest bit of mulMantissa is 0, then the Exponent of the final result is mulExponent, and Mantissa is mulMantissa [45:23]. In this section, we have omitted the low Mantissa portion.

In the final results section, select the correct result based on FPUcontrol and the special values set.

The simulation test contains more than a dozen floating-point addition or multiplication instructions, which also involves the “Load and use” hazard (requiring a pause) and “Data forwarding”, which is enough to prove that our design not only successfully realized the floating-point operation function, but also showed good compatibility with the system of Problem 1 .

The simulation has achieved the expected waveform, as shown in Figure 25. The value of the register displayed on the waveform diagram meets the requirements mentioned in Appendix B.4.

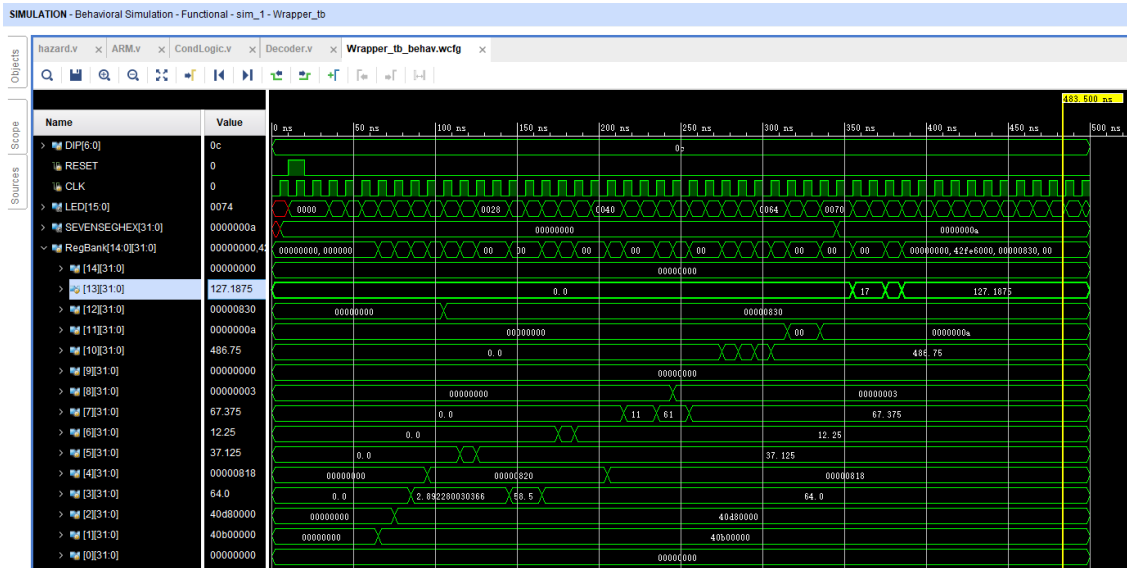

Figure 25: Single-cycle FPU simulation waveform.

We verified the correctness of the code by recording the Data_VAR_Mem values in the Wrapper module during the simulation and comparing them with the results after on-board test. Figure 26 presents values of Data_VAR_Mem and Figures 27 and 28 show the results on the FPGA, with the two data corresponding to floating point numbers of 37.125 and 12.25, respectively.

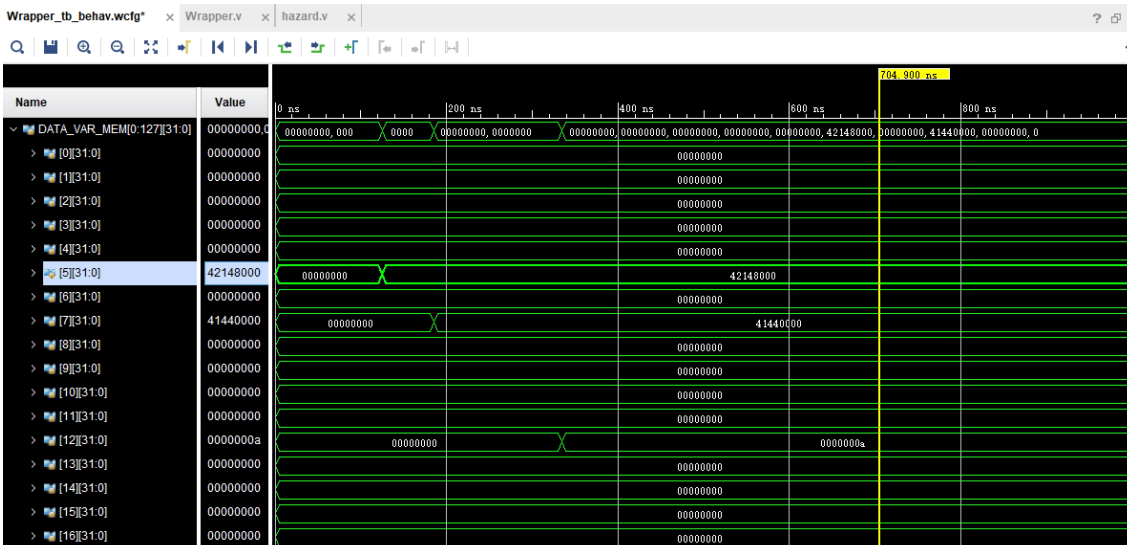

Figure 26: Single-cycle FPU Data_VAR_Mem.

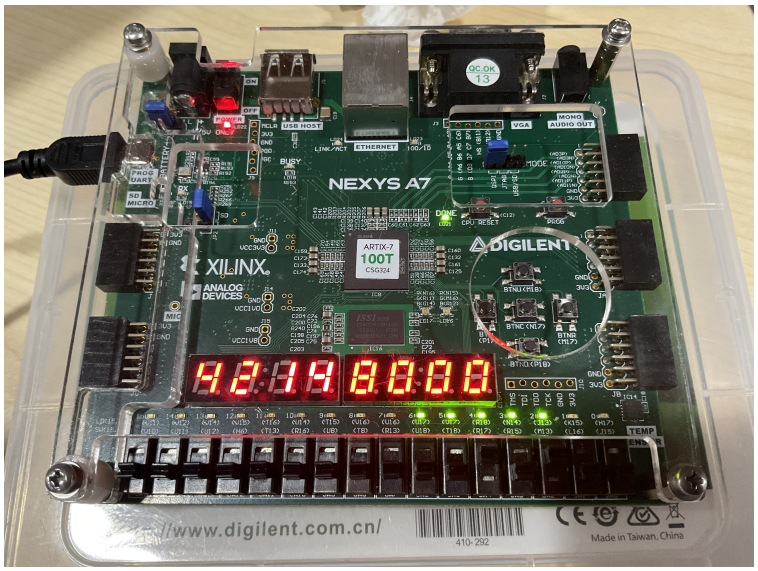

Figure 27: Data_VAR_Mem[5] on board.

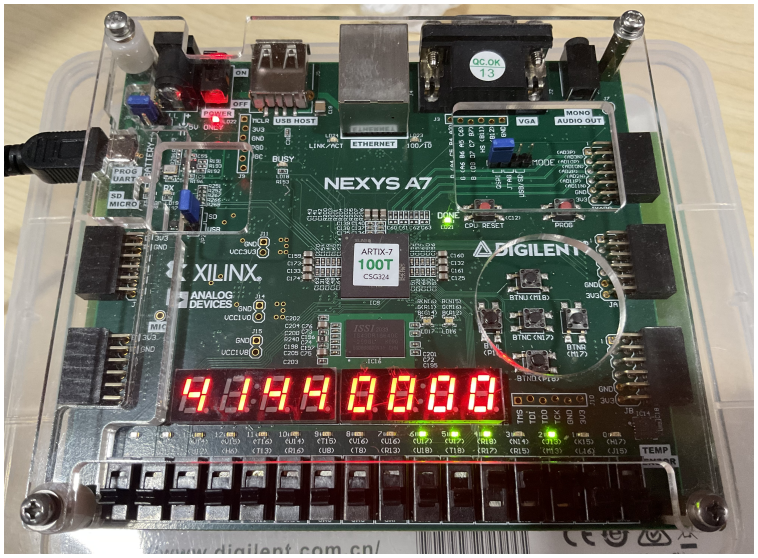

Figure 28: Data_VAR_Mem[7] on board.

3.1.2 Multi-cycle FPU Design

For the multi-cycle floating number operation unit design (FPU), we consider the large difference between addition and multiplication operation process and speed and use the combined logic of single-cycle operation to obtain addition result, while multiplication uses the shift multiplier based on the Lab 3.

In addition, we also learned to add input "start" signal and output "busy" and "ready" signals from the design of multi-cycle multiplications and division unit in Lab 3 to the FPU

module. The "start" signal is used to judge the beginning of the floating point operation, the "ready" signal is used to determine when the result will write back to the register and "busy" signal can help us judge the existence of hazard in multi-cycle operation.

Different from ordinary ALU or Mcycle modules, we need to extract the corresponding symbol, exponential and decimal bits before floating point operation. And then add leading 1 to obtain the sign-magnitude data. In the addition operation, considering the possibility of different symbolic bits, we turn the data into a 2's complement code for operation, and the data of different index values need to be shifted to align the decimal points.

Even worse, the same-sign addition may appear to carry while different sign addition may lead to the right shift of the decimal point, and the multiplication is also divided into carry and no carry (for the resulting floating point exponential bit is equal to exp_ A + exp_ B-127). Since we need to find the leading 1 to turn the result back to the floating point number, we used the for loop, and the position of the first 1 is very important for the value of the exponential bit.

In addition, as previously mentioned, there are some special cases in floating point operations, such as infinite addition and subtraction, zero multiplied by infinity, and multiplication overflow. At this time, we need to assign separate values to the operation results of these special cases, because as long as the judgment of special appears, we can immediately get these special results so these cases are as single-cycle operations as addition.

```
1 FPUReady = (FPUReady_mul | FPUReady_add | FPUReady_special);
```

When the code is implemented, our "busy" signal of FPU is equal to the "FPUbusy_mul" signal because addition and special case single cycle output, similar to ALU. For "ready" signal, it is is set to 1 when any of the three cases occurs and during rest of the time it is set to 0. It is important that when the multi-cycle floating-point number is multiplied (excluding hazard first), pipeline will continue to run other non-conflicting instructions and input signals of the FPU will change.

To solve this problem, we set several registers inside the FPU to store the value of the input port (updated when "start" is on) to prevent multi-cycle calculation errors. When the calculation ends ("busy" drops and "ready" rises), transfers the stored write port signal value back to the data path of the pipeline. As teacher mentioned in lab, the floating point operation does not need to go through the Memory stage and the operation results can be directly written back to Register File after the Execute stage. Therefore, we add a write port to the register and the corresponding control signal (connected by "ready" of FPU) and control result of FPU to write back separately.

Following Figure 29 and 30 are simulation results of multi-cycle FPU, which proves our design can satisfy the requirement.

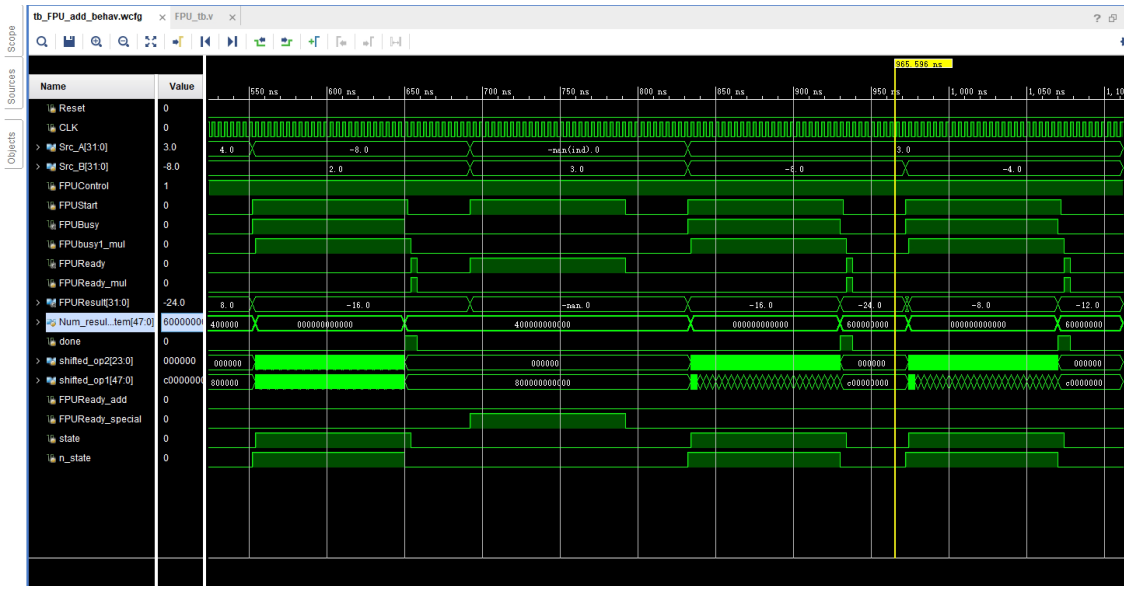

Figure 29: Simulation of multi-cycle FPU ADD.

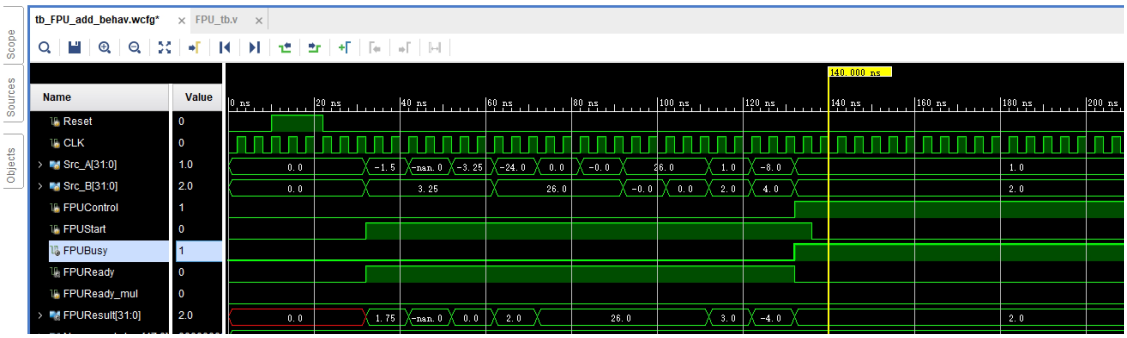

Figure 30: Simulation of multi-cycle FPU MUL.

3.1.3 Comparison of Single-Cyle and Multi-Cycle FPU

Latency	Total number of instance cells
Single-cycle FPU	1717
Multi-cycle FPU	1256

Table 3: Synthesis resources.

Finally, after comparing single-cycle and multi-cycle FPU with consuming resources and considering the hazard that needs to be solved in the overall project pipeline framework (will be mentioned in the later part), we adopted the single-cycle FPU scheme. Table 3 shows the hardware resources consumed after the synthesis of the two design schemes.

3.2 Non-stalling for Multi-cycle Instructions

Multipliers and Floating-Point Unit will not change flags.

3.2.1 Hazard for Non-stalling and Multi-cycle Instructions

Multi-cycle instructions like multiplication, division and float-point operations need many cycles to obtain the final result. Undoubtedly, this will cause data hazard as shown in Figure 31. Three control signal will detect the states of Mcycle and FPU units and stall CPU if data hazard happen.

FPU-busy and FPU-ready is used when the stall starts and when the stall should be over. Just like DP data hazard, data forwarding is used to improve the performance as soon as the operation is finished.

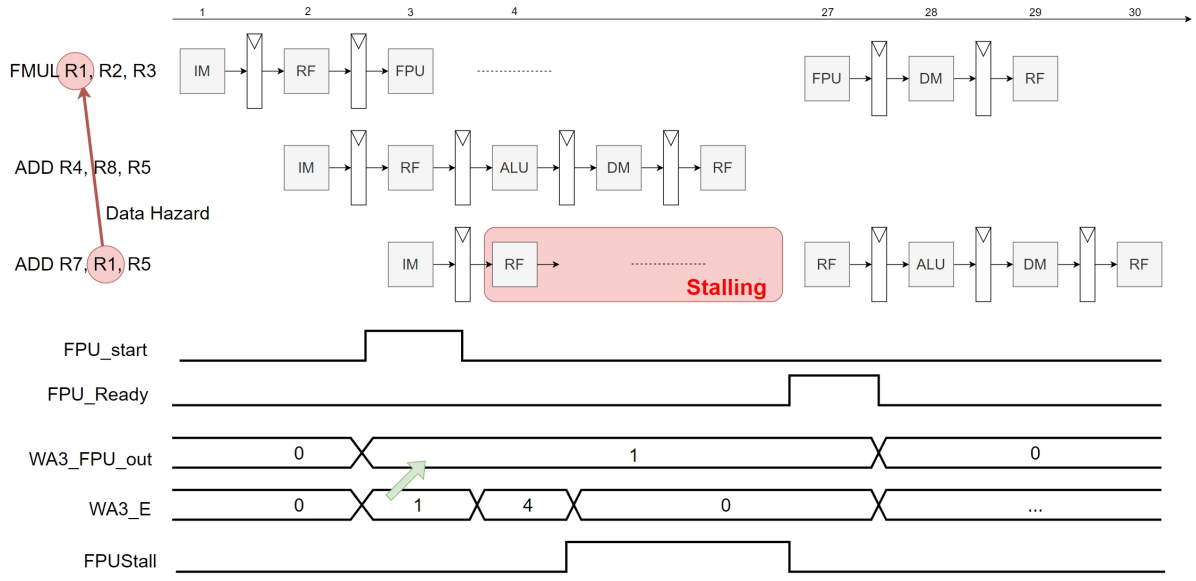

Figure 31: Multi-cycle DP instruction data hazard 1.

```
1 Match_1D_E_FPU = (RA1D == WA3E_FPU);
2 Match_2D_E_FPU = (RA2D == WA3E_FPU);
3 FPUstall = (Match_1D_E_FPU || Match_2D_E_FPU) & FPUBusyE;
```

As shown in Figure 31, we need to monitor whether the reading port of Decoder stage equals to the port written by FPU in the multi-cycle floating point multiplication operation ("busy" is equal to 1). If conflicts arise, we need to stall Decoder and Fetch stages and flush out Execute stage until the FPU operation is complete ("busy" changes back to 0).

There are two types of hazards, RAW hazard in Figure 32 and WAW hazard in 33. RAW could be dissolved by the stalling, which is explained in 3.1.2. As to WAW hazards, once the new instruction terminates the multi-cycle multiplier or FPU, controlled by **halt** signal.

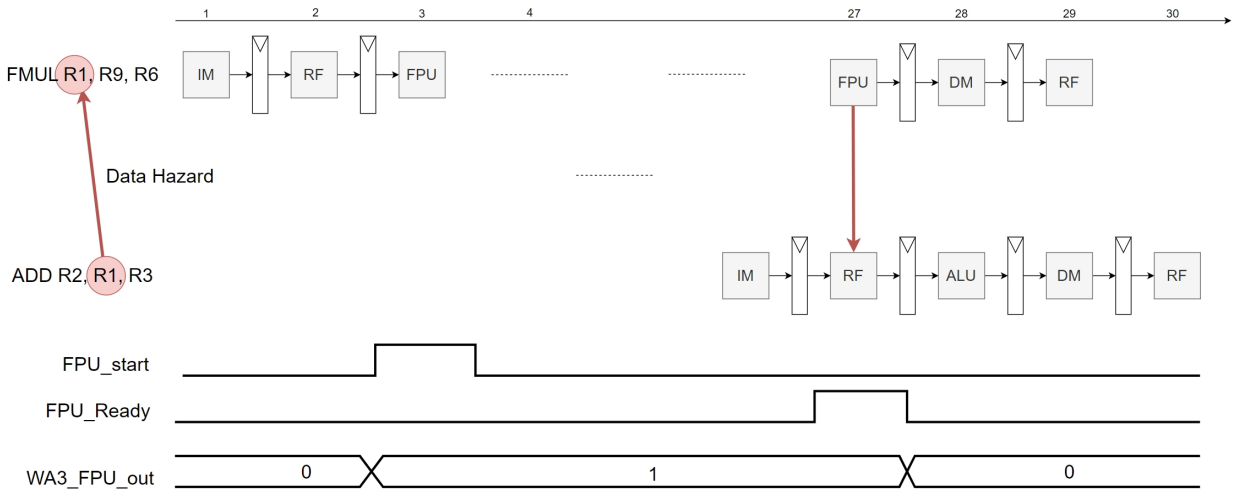

Figure 32: Multi-cycle DP instruction data hazard 2.

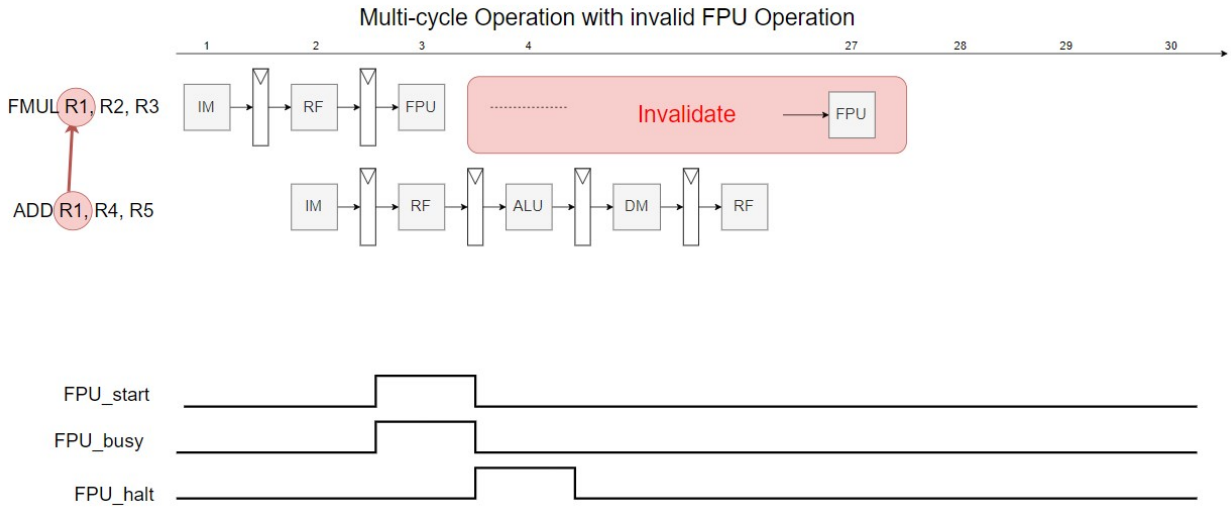

Figure 33: Multi-cycle DP instruction data hazard 2.

3.2.2 Design ideas for Non-stalling and Multi-cycle Instructions

We can first roughly divide it into four steps.

1. The first step is to split the multiplier between D and E and insert it into a five stage pipeline.
2. The second step is to backup the status signal of the multiplier, which includes two inputs belonging to the multiplier, written to registers, RegWrite, MenWrite, etc and FlushM.

3. The third step is to check whether the three registers A1, A2, and A3 of level D are the same as the E-level backup signal when the multiplier is running, and generate corresponding stall and flush signals.

4. The fourth step is to insert the signal matched with the operation result into the M-level and stage the D and F levels when the multiplier operation is completed.

We have added two signals in the Decoder, StartMCycle and MCycleOpMCycles represent the start of multi cycle operations and the resolution of multiplication and division. Tested StartMCycle only displays 1 for one cycle at level E. Then add a timer to start counting from the beginning of the operation and guess exactly 31 at the end of the operation. The next step is to backup the multi cycle operation exclusive signals such as RegWriteE and MenWriteE, WA3EMCycle, etc. of mul, and insert these signals into the M-level when the timer is 31.

At the beginning, only multiplication was considered here, without division. These exclusive signals are initially saved at the rising edge of StartE. Then comes the addition of FlushM, as M-level registers cannot be updated at the beginning of multi cycle operations. But without processing, M-level registers will still obtain incorrect values. Therefore, it is necessary to follow StartMCycleE refresh. Add three types of wizards. Due to differences in instruction formats, it is necessary to add a judgment condition to obtain the target register.

The next step is to debug using the testbench generated by the assembly program shown in Appendix B.3. According to the waveform observation, the signal required for multiple cycles was not correctly saved. The correct save needs to occur at the falling edge of the clock signal, and when StartMCycleE is 1, the saved data is correct. After correctly saving the output, it was found that the timer was ineffective and failed to follow the changes in the Busy signal, resulting in an issue with the output result.

The solution is to calculate that multiplication requires 330 ns, division requires 340 nanoseconds, and timer+1 requires 10 ns. Therefore, when the Busy signal is 1, the timer starts counting, and when Busy is 0, the timer clock is 0. To determine the output time of the result, a timer and MCycleOpMCycleE need to work together. When multiplying, the timer needs to be equal to 33, and when dividing, the timer needs to be equal to 34. In the above two situations, MCycleResultstall is set to 1, causing D and E levels to stall, and the output result and other backup signals are given to the corresponding M-level registers. It must be emphasized that when F and D levels stall, FlushE must be used, otherwise incorrect instruction waveforms may occur.

3.2.3 Simulation result for Non-stalling and Multi-cycle Instructions

After solving the bugs one by one, the simulation has achieved the desired waveform, as shown in Fig34. The value of the register displayed on the waveform diagram meets the requirements mentioned in Appendix B.3.

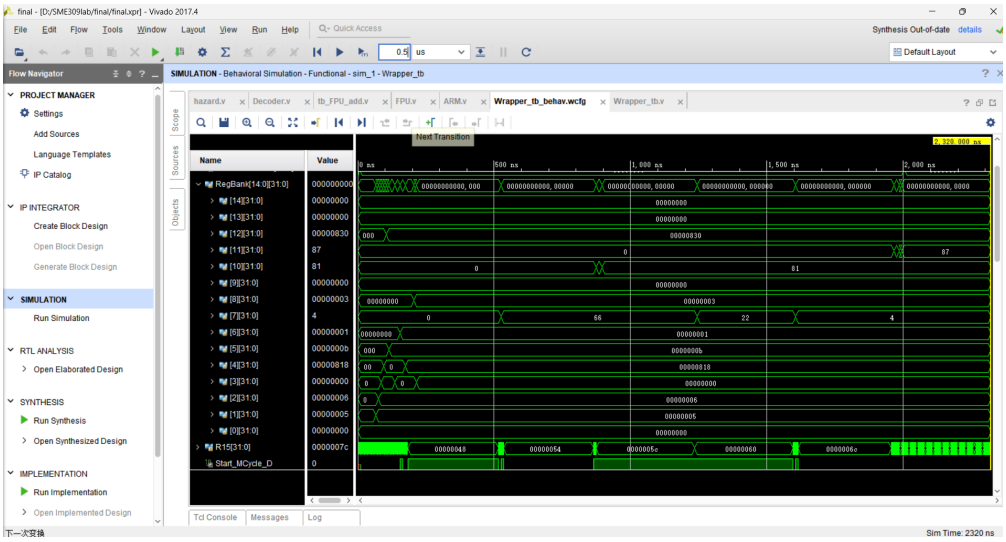

Figure 34: MCycle simulation waveform.

Figure 35: Stall when in conflict, and the pipeline works normally when there is no conflict.

Part IV: RISC-V Single-Cycle Processors

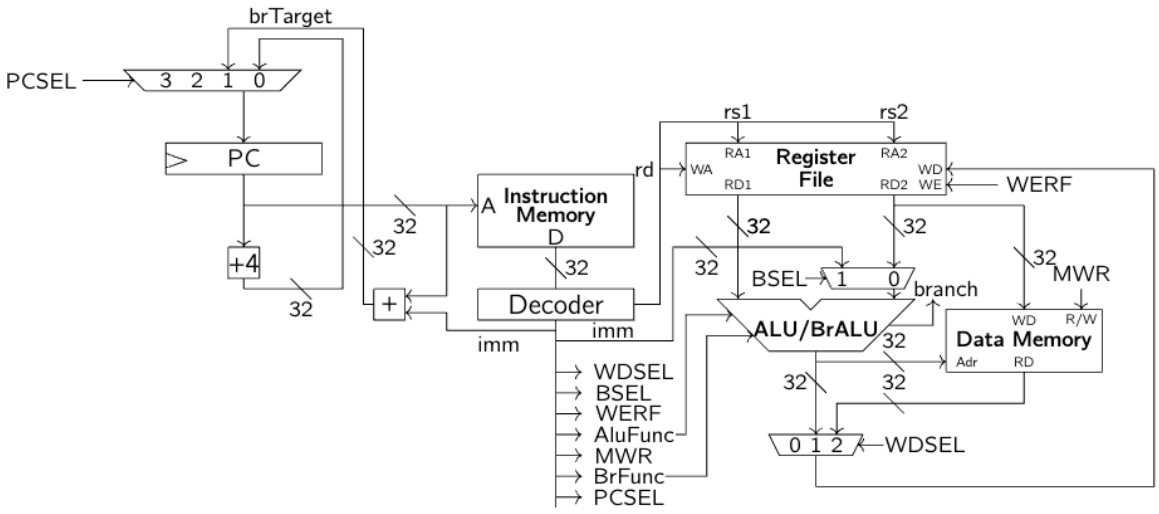

Figure 36: RISC-V RV32I Single-Cycle Processor Block Diagram.

The RISC-V RV32I single-cycle processor is implemented. Figure 36 shows its architecture and the instruction set architecture is provided in Appendix A.

One of the significant differences is that there are no NZCV flags in RISC-V. ARM uses NZCV flags to activate the conditional DP operations and branch jump. It usually takes two instructions for ARM. Instead, the RISC-V processor has a binary comparator in ALU so that it performs conditional jump with one instruction.

The Jal and Jalr instructions could save the return address of the previous function. After executing the subfunctions, the RISC-V CPU could load the address to the PC and go back to the main function location.

Due to the limitation of time, the source code is provided without simulation.

Part V: [Bonus] Branch Predictor

5.1 Branch Predictor Architecture

A branch predictor is implemented to eliminate the number of stalling cycles for branch instructions. It includes the branch history table (BHT) and branch target buffer (BTB). BHT stores the prediction bits, and BTB contains the branch target address (BTA). If the branch is predicted to be **taken**, then the BTA will be the next PC value; otherwise, nothing happens.

However, there are chances to predict the branch direction wrong: if predicted "taken", this BTA value will be compared with the correct ALUResult at the EXECUTE stage. The CPU will flush all the data after the DECODE stage and load the correct PC value to the program counter once the mis-prediction is detected. On the other hand, if the assumed "not taken" is wrong, the ALUResult_E, the taken branch address, will be uploaded to the BTB. The prediction bits update after every branch instruction.

The hardware architecture is provided in Figure 37. There are 16 lines, and each line has 2 prediction bits and a 32-bit branch target buffer. In our design, the number of executing instructions is small so that the address collision issue could be ignored and one-way BHT/BTB is sufficient. The **PC[5:2]** determines the address of BHT/BTB. We updated the PB in BHT and BTA in BTB by two control signals, Branch_MP and BTA_MP once misprediction of branch or BTA occurs, connecting to WE_PrPCSrc and WE_PrALUResult respectively.

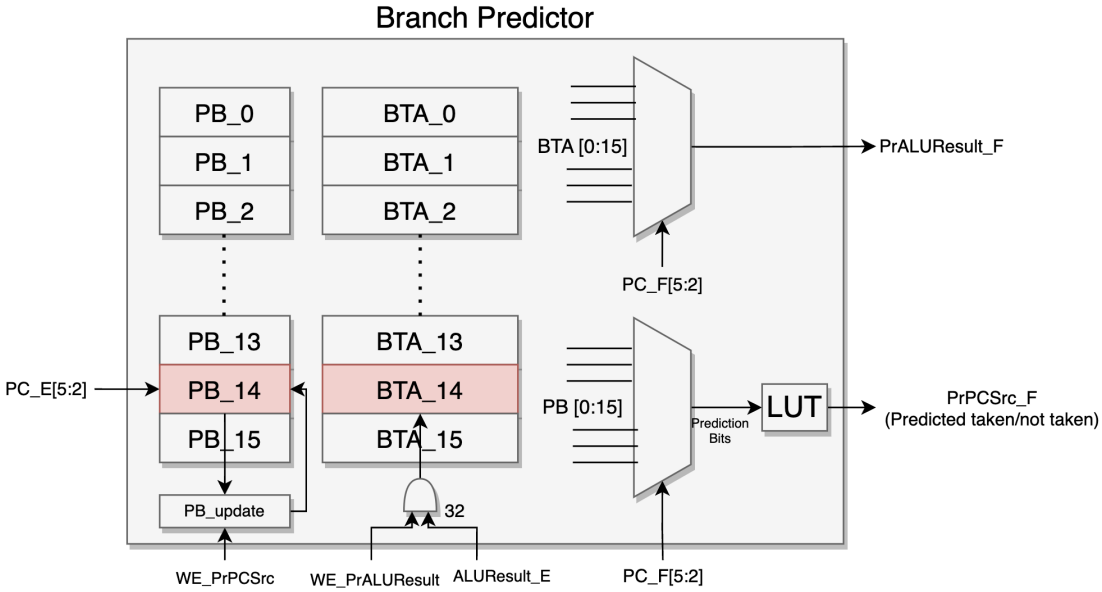

Figure 37: The 16-line and 2-prediction-bit branch predictor architecture. LUT provides predicted the branch taken or not taken from the prediction bits. The WE_PrPCSrc signal is asserted and the 2-bit BP at the line address is updated if the branch taken/not taken is mispredicted, while WE_PrALUResult updates BTA.

Figure 38 shows the exception handling mechanism of the program counter. Before branch check in the EXECUTE stage, if PrPCSrc_F is 1, then the BTA is taken and loaded to PC; otherwise load PC_plus_4F. If the misprediction of the branch or BTA happens, then the PC loads PC_E+4 if the prediction is taken and the actual outcome is not taken; otherwise, the ALUResult_E will be the next PC value.

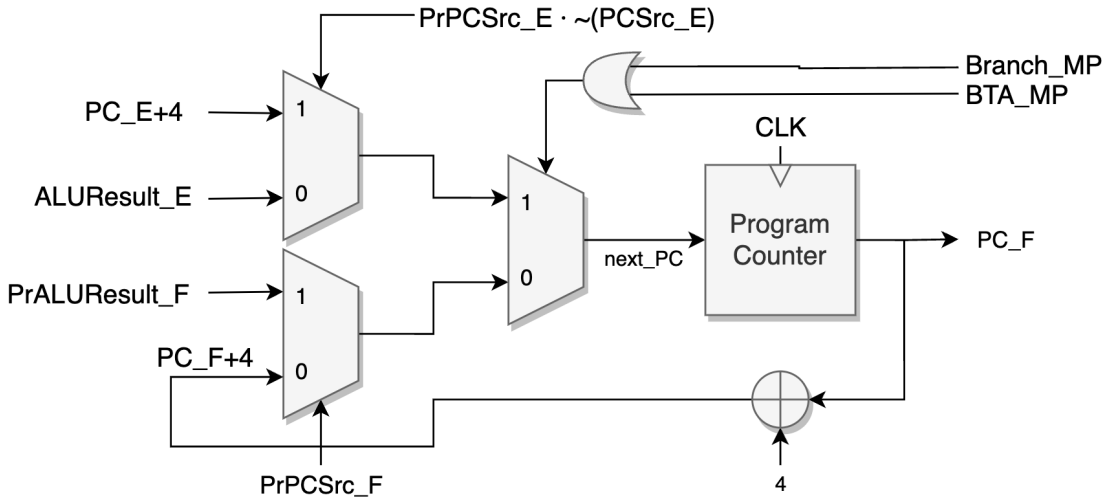

Figure 38: The program counter exception handling mechanism once mispredicted.

The connection of the branch predictor in the processor is illustrated in Figure 39 (from SME309 Lecture 7 Slides). But some connections are still missing. As a reminder, the **PC_plus_8D** (original R15 input of the register file, the same as PC_plus_4F) is not equal to the **PrALUResult_F_plus_4** after the jump by branch prediction. So the RF R15 input should be replaced by the expression below: (add a MUX in the hardware)

```
1 assign R15 = (PrPCSrc_F) ? PrALUResult_F_plus_4 : PC_plus_8D;
```

The FlushE connection in Figure 39 is also not completed because the load-and-use hazard (the following DP instruction after LDR should stall for one clock cycle) is not included in the schematic. So the new FlushE asserting function is

```
1 assign FlushE = LDRStall || ((BTA_MP | Branch_MP));
```

5.2 Verification and Benchmarking of Branch Predictor

The branch predicted pipeline processor is passed the assembly program in Appendix B.1. In order to demonstrate the performance, the Appendix B.2, which has repeated branch operations, is used to benchmark the baseline CPU and branch-predicted CPU.

The execution time result is listed in Table 4. The program finishes at 1205 ns (Figure 40) with dynamic branch prediction. The good news is that the branch preictor stabilizes all

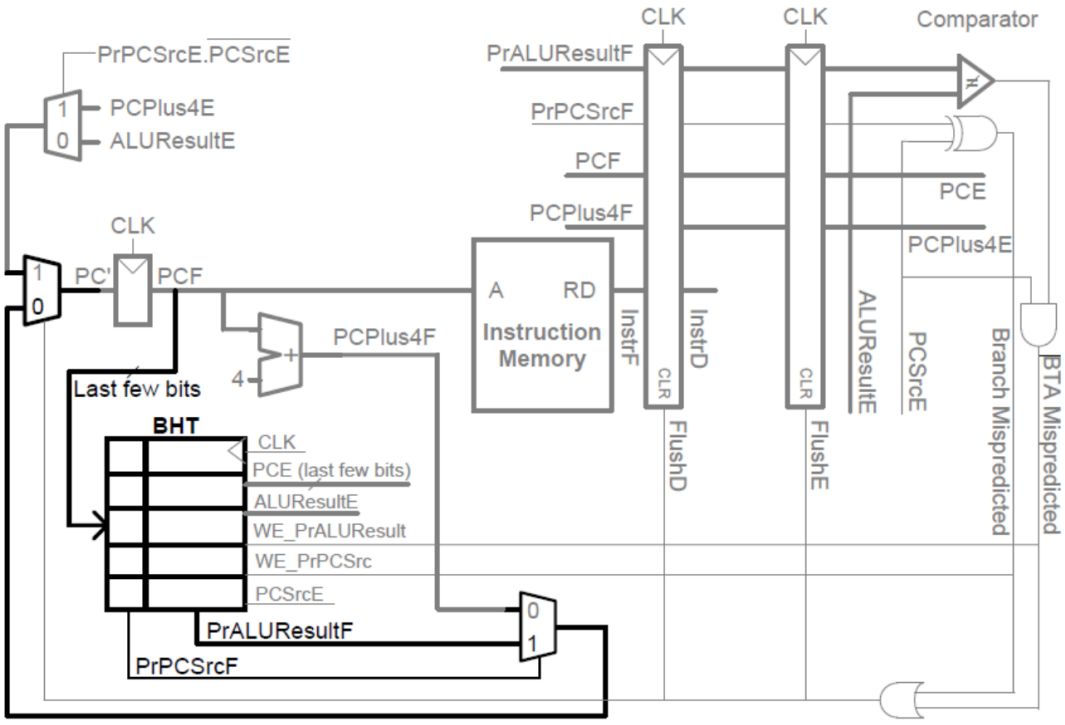

Figure 39: Connection of BHT/BTB in the ARM pipeline processor from Lecture 7 slides. If the misprediction is detected, two instructions after this branch instruction are flushed, and the correct PC value will be loaded to the program counter. Some logic is still missing for system integration.

the signals after the program execution. However, the baseline processor uses 1145 ns. The reason is that the BP also predicted a jump for a DP instruction, which introduces extra wasted clock cycles, so it does not necessarily provide a lower latency.

In order to reduce the number of false jumps, we add a column of tags to the design (/Branch_Prediction/Branch_Predictor_with_tag.v). Only branch-relevant PC values will be loaded in BTB. The execution time is 1105 ns (Table 4), lower than both above. These advantages will become more obvious if the number of instructions in the testbench increases.

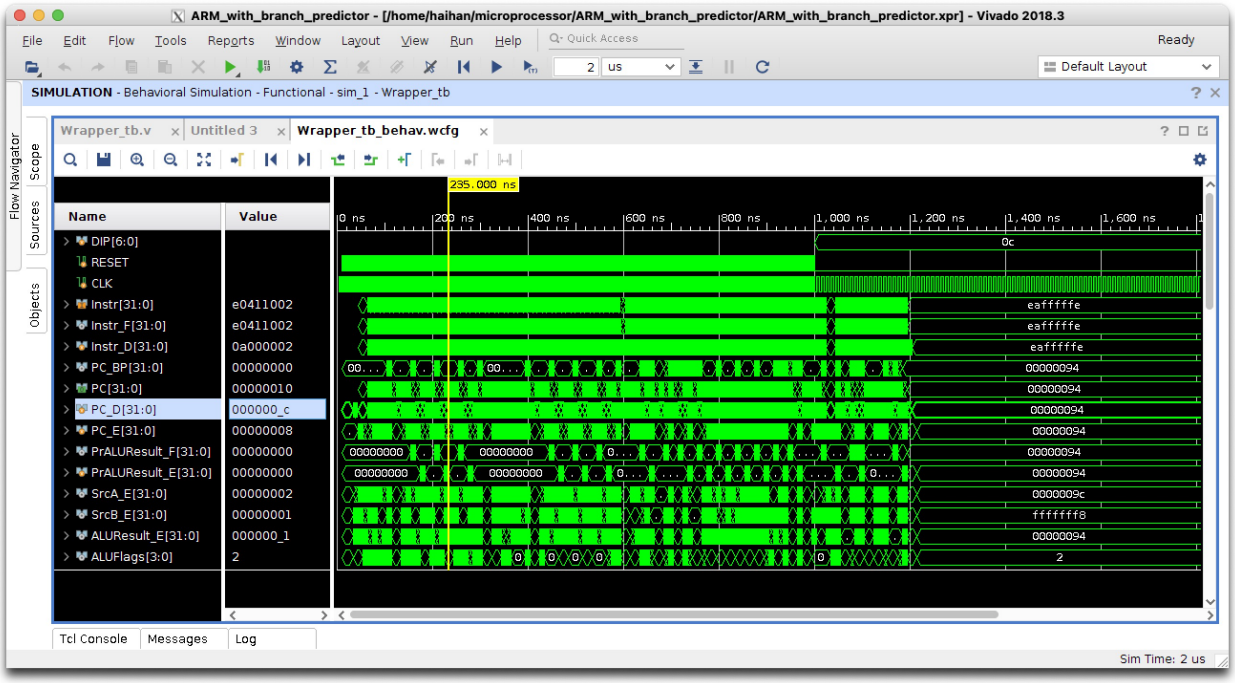

Figure 40: The simulation waveform of a pipeline processor with no-tag branch predictor.

CPU	Tag	Execution Time (ns)	Improved by (ns)
Baseline CPU	-	1145	-
Branch predicted CPU	no tags	1205	-60
Branch predicted CPU	has tags	1105	**40**

Table 4: Performance table of processors.

Appendix

Appendix A: RISC-V Instruction Set Architecture

RV32I Base Instruction Set (MIT 6.191 (6.004) subset)

imm[31:12]				rd	0110111	LUI
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 41: RISC-V RV32I Instruction Set Architecture from the MIT 6.191 course.

Appendix B: Assembly Program

Appendix B.1 Baseline Pipeline Assembly Program

B.1 lists the Problem 1 Assembly Program. The final register value of R10 is 0x00000041.

```
    LDR R1, constant1; R1=5
    LDR R2, constant2; R2=6
    ADD R5, R1, R2;
    LDR R3, addr1; 810
    LDR R4, addr2; 820
    LDR R12,addr3; 830

    STR R5, [R3,#4];
    ADD R3, R3, #8;
    LDR R5,[R3,#-4]; R5 = 11;

    SUB R6, R2, R1;  R6 = 1;
    STR R6, [R4,#-4];
    SUB R4, R4, #8; R4 = 0x00000818;
    LDR R6,[R4,#4];       R6 = 1;

    MUL R7,R5,R2;R7=66
    LDR R8,constant3; R8=3
    LDR R3,number0;R3=0
    ADDS R3,R3,#0; SET Z FLAG = 1
    ADD R10,R1,#10; R10=15;
    ADDS R10,R10,R7;

    LDR R8, [R4,#4]; R8 = 1;
    STR R8, [R4,#-4];
    ADD R4, R4, #0;
    SUB R5, R5, #0;
    ORR R5, R5, #0;
    LDR R1, [R4,#-4];

    ADD R1, R4, R5; R1 = 0x00000823;
    AND R8, R1, R3; R8 = 0x00000000;
    ORR R9, R6, R1;
    SUB R10, R1, R7;
    ORR R11, R12, R13;

    B BTA;
    AND R8, R1, R3;
    ORR R9, R6, R1;
    SUB R10, R1, R7;
```

```
41      SUB R11, R1, R8;
42
43  BTA
44      LDR R1, [R4,#-4]; R1 = 1;
45      AND R8, R1, R3;
46      ORR R9, R6, R1; R9 = 1;
47      SUB R10, R7, R1;
48
49  halt
50      B     halt
```

Appendix B.2 Branch-Prediction Assembly Program

B.2 lists the Branch-Prediction Assembly Program. The program finishes when R5 becomes 0.

```
1   LDR R1, constant1; R1=5
2   LDR R2, constant3; R2=1
3
4   Loop1
5   CMP R1, R2;
6   BEQ Out1;
7   SUB R1, R1, R2;
8   ADD R3, R3, #1; R3 act as a timer, R3 = 4;
9   B Loop1;
10
11  Out1
12  LDR R1, constant1; R1=5
13  ORR R2, R2, R2;
14  BIC R3, R3, R3; Set R3 to 0
15
16  Loop2
17  CMP R1, R2;
18  BEQ Out2;
19  ADDS R3, R3, #1;
20  SBC R1, R1, #0;
21  B Loop2;
22
23  Out2
24  LDR R5, constant2; R5=6
25  LDR R6, constant3; R6=1
26  AND R3, R3, #0; Set R3 to 0
27
28  Loop3
29  CMP R5, R6; Set C flag to one.
30  ADC R6, R6, #0;
31  ADD R3, R3, R6;
32  BNE Loop3;
33
34  SUB R3, R3, #20; R3 = 0
35  RSB R1, R3, R1; R1 = 1
36  EOR R6, R6, #0; R6 = 6
37  MOV R7, R6; R7 = R6 = 6
38  LDR R7, constant3; R7 = 1
39
40  Loop4
```

```
41      ADDS R5, R7, R5;
42      CMP R6, R5;
43      BNE Loop4; Two loops are backward case1
44
45  Loop5
46      RSC R5, R7, R5; R5 = 5
47      CMP R5, R7;
48
49      BEQ Loop5;
50      BNE Loop6;
51
52      ADD R5, R5, #5;
53      SUB R5, R6, #3;
54
55  Loop6
56      SUB R5, R5, #5;
57
58  halt
59      B    halt
```

Appendix B.3 Non-stalling MUL Assembly Program

B.3 lists the Non-stalling Multiplication Assembly Program. The program finishes when R11 becomes 0X0000 0057.

```
1         LDR R1, constant1; R1=5
2         LDR R2, constant2; R2=6
3         LDR R3, addr1; 810
4         LDR R4, addr2; 820
5         LDR R12,addr3; 830
6         ADD R5, R1, R2; R5 = a1 + a2;
7
8         STR R5, [R3,#4];
9         ADD R3, R3, #8;
10        LDR R5,[R3,#-4]; R5 = 11;
11
12        SUB R6, R2, R1;  R6 = 1;
13        STR R6, [R4,#-4];
14        SUB R4, R4, #8;
15        LDR R6,[R4,#4];  R6 = 1;
16
17        MUL R7,R5,R2;R7=66
18        LDR R8,constant3; R8=3
19        LDR R3,number0;R3=0
20        MULEQ R7,R1,R8; not execute,R7=66
21        ADDS R3,R3,#0; SET Z FLAG = 1
22        MULEQ R10,R1,R8; R10=15;
23        ADDS R10,R10,R7; R10 =66+15=81,flags are 0
24
25        DIV R7,R7,R8; R7=66/3=22
26        DIV R7,R7,R1; R7=22/5=4
27        DIVEQ R7,R2,R8; not execute, R7 = 4
28        ADDS R3,R3,#0; SET Z FLAG = 1
29        DIVEQ R11,R2,R8;R11=6/3=2;
30        ADD R11,R11,R7; R11=2+4=6
31        ADD R11,R11,R10;R11=81+6=87=0X0000 0057
32
33        STR R11,[R12];
34
35  halt
36        B    halt
```

Appendix B.4 FPU Assembly Program

B.4 lists the Floating-point Operation Unit Assembly Program. The program finishes when R13 becomes 0X42FE 6000.

```
        LDR R1, float1; R1=5.5
        LDR R2, float2; R2=6.75
        LDR R3, addr1; 810
        LDR R4, addr2; 820
        LDR R12,addr3; 830
        FADDS R5, R1, R2; R5=12.25
        FMULS R5, R1, R2; R5=37.125

        STR R5, [R3,#4]; R5=37.125
        LDR R3, float3;  R3=58.5
        FADDS R3, R3, R1; R3=64

        LDR R6, float4; R6=1.25
        FADDS R6, R2, R1; R6 = 12.25;
        STR R6, [R4,#-4]; R6 = 12.25;
        SUB R4, R4, #8; R4 = 812

        LDR R7, float5; R7=11.25
        FMULS R7,R7,R1;R7=61.875
        LDR R8,constant3; R8 = 3
        FMULS R7,R1,R6; R7=67.375
        ADD R4,R4,#0; R4 = 812
        LDR R10, float6; R10 = 0.5
        FMULS R10,R1,R3; R10=352;
        FADDS R10,R10,R7; R10 =419.375

        FADDS R10, R7, R10; R10=486.75
    LDR R11,constant2; R11=6
        ADD R11,R11,#4; R11=10
        STR R11,[R12];
        LDR R13, float7; R13=17.625
        FADDS R13, R13, R1; R13=23.125
        FMULS R13, R13, R1; R13=127.1875

halt
        B    halt
```

Appendix B.5 Cache Assembly Program

The final register values of R8, R9, and R10 are 3, 4, and 2, respectively.

```
    LDR R1, addr1;
    LDR R2, addr2;
    LDR R3, addr3;
    LDR R4, constant1;
    LDR R5, constant2;
    LDR R6, constant8;

    STR R4, [R3,#4];
    STR R5, [R3,#0];
    STR R6, [R1,#4];
    STR R4, [R1,#-4];
    STR R5, [R2,#4];
    STR R6,  [R2,#-4];

    LDR R7, constant1;
    ADD R4, R7, R7; R4=10
    ADD R5, R7, R7; R5=10
    ADD R6, R7, R7; R6=10

    LDR R8, constant3;
    ADD R4, R8, R8; R4=6
    ADD R5, R8, R8; R5=6
    ADD R6, R8, R8; R6=6

    LDR R9, constant4;
    STR R9, [R3,#4];
    LDR R10, constant5;
    STR R10, [R3,#8];

halt
    B    halt
```

Appendix B.6 System Assembly Program

B.5 lists the whole system Assembly Program. And it acts as the on-board test of the whole system. The ideal on-board implementation should be like this: When SW5 and SW4 pulls up, the SevenSegs can show 00000057. When SW5 pulls up, the SevenSegs can show 00000818.

```
1         LDR R1, constant1; R1=5
2         LDR R2, constant2; R2=6
3         LDR R3, addr1; 810
4         LDR R4, addr2; 820
5         LDR R12,addr3; 830
6         ADD R5, R1, R2; R5 = a1 + a2;
7
8         STR R5, [R3,#4];
9         ADD R3, R3, #8;
10        LDR R5,[R3,#-4]; R5 = 11;
11
12        SUB R6, R2, R1;  R6 = 1;
13        STR R6, [R4,#-4];
14        SUB R4, R4, #8;
15        LDR R6,[R4,#4];  R6 = 1;
16
17        MUL R7,R5,R2;R7=66
18        LDR R8,constant3; R8=3
19        LDR R3,number0;R3=0
20        MULEQ R7,R1,R8; not execute,R7=66
21        ADDS R3,R3,#0; SET Z FLAG = 1
22        MULEQ R10,R1,R8; R10=15;
23        ADDS R10,R10,R7; R10 =66+15=81,flags are 0
24
25        DIV R7,R7,R8; R7=66/3=22
26        DIV R7,R7,R1; R7=22/5=4
27        DIVEQ R7,R2,R8; not execute, R7 = 4
28        ADDS R3,R3,#0; SET Z FLAG = 1
29        DIVEQ R11,R2,R8;R11=6/3=2;
30        ADD R11,R11,R7; R11=2+4=6
31        ADD R11,R11,R10;R11=81+6=87=0X0000 0057
32
33        LDR R1, constant1; R1=5
34        LDR R13, constant4; R13=1
35
36
37  Loop1
38        CMP R1, R13;
```

```
39          BEQ Out1;
40          SUB R1, R1, R13;
41          B Loop1;
42
43  Out1
44          LDR R1, constant1; R1=5
45          ORR R13, R13, R13;
46          BIC R3, R3, R3; R3=0
47
48  Loop2
49          CMP R1, R13;
50          BEQ Out2;
51          ADDS R3, R3, #1;  R3 = 4
52          SBC R1, R1, #0;
53          B Loop2; Two loops are forward case2
54
55  Out2
56
57          LDR R5, constant2; R5=6
58          LDR R6, constant4; R6=1
59          AND R3, R3, #0; Set R3 to 0
60
61  Loop3
62          CMP R5, R6; Set C flag to one.
63          ADC R6, R6, #0;
64          ADD R3, R3, R6; R3 act as a special timer, and R3 = 20.
65          BNE Loop3;
66
67
68          SUB R3, R3, #20; R3 = 0
69          RSB R1, R3, R1; R1 = 1
70          EOR R6, R6, #0; R6 = 6
71          MOV R7, R6; R7 = R6 = 6
72          LDR R7, constant4; R7 = 1
73
74          LDR R1, constant1; R1=5
75          LDR R2, constant2; R2=6
76          ADD R5, R1, R2;  R5 = a1 + a2 = 11;
77          LDR R3, addr1; 810
78          LDR R4, addr2; 820
79          LDR R12,addr3; 830
80
81          STR R5, [R3,#4];
82          ADD R3, R3, #8;
83          LDR R5,[R3,#-4]; R5 = 11;
```

```
84
85          SUB R6, R2, R1;  R6 = 1;
86          STR R6, [R4,#-4];
87          SUB R4, R4, #8; R4 = 0x00000818;
88          LDR R6,[R4,#4];  R6 = 1;
89
90          ADD R7,R5,#55;R7=66
91          LDR R8,constant3; R8=3
92          LDR R3,number0;R3=0
93
94          LDR R8, [R4,#4]; R8 = 1;
95          STR R8, [R4,#-4];
96          ADD R4, R4, #0; R4 = 0x00000818;
97          SUB R5, R5, #0; R5 = 11;
98          ORR R5, R5, #0; R5 = 11;
99          LDR R1, [R4,#-4]; R1 = 1;
100         ADD R1, R4, R5; R1 = 0x00000823;
101         AND R8, R1, R3; R8 = 0x00000000;
102
103         LDR R1, float1; R1=5.5
104         LDR R2, float2; R2=6.75
105         LDR R3, addr1; 810
106         LDR R4, addr2; 820
107         LDR R12,addr3; 830
108         FADDS R5, R1, R2; R5=12.25
109         FMULS R5, R1, R2; R5=37.125
110
111         LDR R3, float3;  R3=58.5
112         FADDS R3, R3, R1; R3=64
113
114         LDR R6, float4; R6=1.25
115         FADDS R6, R2, R1; R6 = 12.25;
116         SUB R4, R4, #8; R4 = 0x00000818;
117
118         LDR R7, float5; R7=11.25
119         FMULS R7,R7,R1;R7=61.875
120         LDR R8,constant3; R8 = 3
121         FMULS R7,R1,R6; R7=67.375
122         ADD R4,R4,#0; R4 = 0x00000818;
123         LDR R10, float6; R10 = 0.5
124         FMULS R10,R1,R3; R10=352;
125         FADDS R10,R10,R7; R10 =419.375
126         FADDS R10, R7, R10; R10=486.75
127
128         LDR R14,addr2; R14=0x00000820;
```

```
129         STR R4, [R14]; R4=0x00000818;
130         STR R11,[R12]; R11=0X0000 0057
131
132
133 halt
134         B    halt
```

Appendix C: Running Vivado on Apple M1 CPU

The appendix is dedicated to running Xilinx Vivado on Macbooks with M1/M2 processors. The **docker container** could provide a separate Ubuntu operating system on macOS without taking up too much memory usage (docker is not exactly a virtual machine).
After downloading the Xilinx Vivado offline package in the docker, the X11 should be set up correctly to enable display from Docker to your Macbook. Make sure to click the box "allow connection from network clients" in your XQuartz setting manual (Figure 42). Detailed installation tutorial is provided on https://github.com/BillMrvica/SME309_FinalProj_ARM_v3/blob/haihan_final/Vivado_on_mac/README.md.

Figure 42: Enable X11-Xquartz display on macOS.

If you follow the tutorial carefully and set up everything correctly, you should be able to launch the software by typing the command **"vivado"** in your docker container's terminal. The Vivado GUI window pops up and now you can start your projects (Figure 43)!

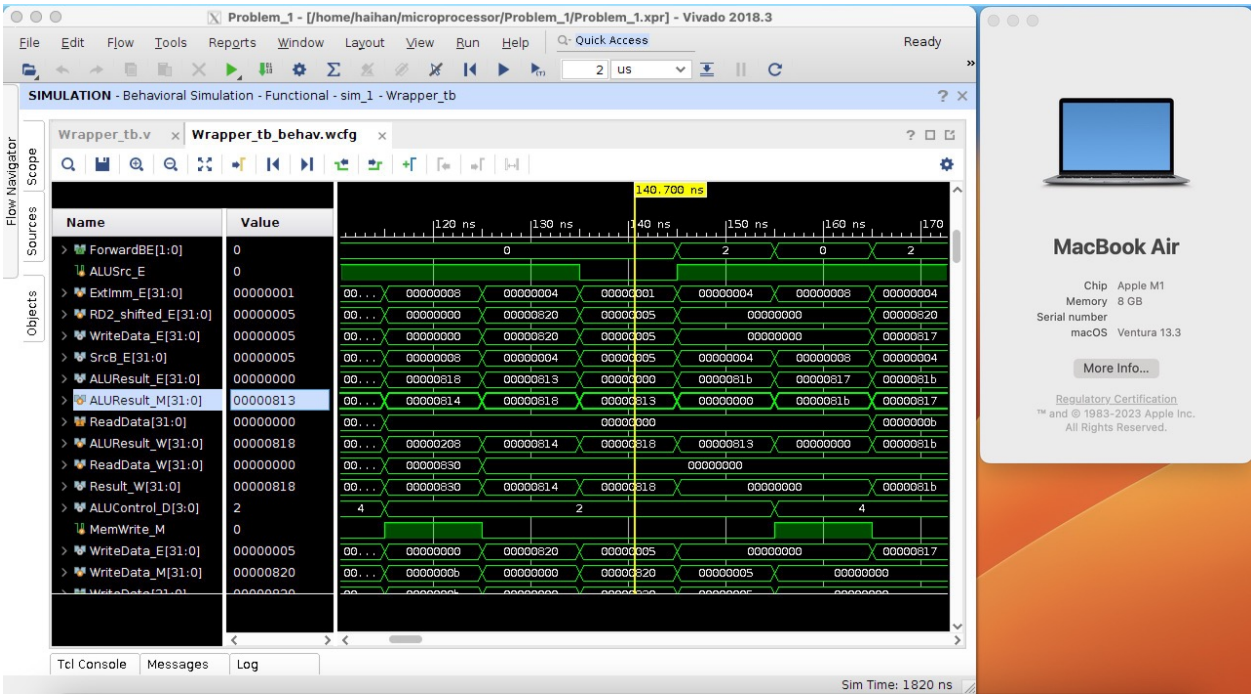

Figure 43: Vivado 2018.3 is running on my M1-CPU Macbook.