Project 3

- Pipelined MIPS Simulator -

Prepared by

Lee Chang Yoon

32183641@dankook.ac.kr

Left Days: 5
March 7, 2022

Table of Contents

1. Introduction

The pipeline is the idea of dividing the instruction processing cycle into distinct processing stages. Also, it means that microarchitecture processes a different instruction in each stage. Instructions consecutively in program order are processed in consecutive stages.

In this report, we will describe pipelined microarchitecture operation, its implementation, and the evaluation of the project.

The work presented in this report is the third part of a large project designed to implement and optimize the microarchitecture that uses the MISP ISA. We will apply the advanced concepts to the former single-cycle microarchitecture to enhance the performance of the microarchitecture.

The first step in this project is to specify the requirements for the pipelined microarchitecture. Second, we move on to concepts critically related to microarchitecture: multi-cycle microarchitecture, pipelined microarchitecture, performance analysis, data dependency handling, control dependency handling, and branch prediction. Third, we will state the data path, which includes the latches, forwarding unit, data hazard detection unit, branch prediction unit, and program definitions. Fourth, we will describe how we implemented the data paths and pipelined MIPS simulator according to the hardware components and program definitions. Then, there will be some results by executing the binary programs using an implemented simulator. In the end, we will evaluate the pipelined MIPS simulator with performance comparison with the single-cycle MISP simulator, and the flow of the data paths based on some assumptions.

2. Requirements

Index	Requirement
1	Compare performance of pipelined vs. single-cycle microarchitecture. a) Program should produce correct output. b) Compare the number of CPU clock cycles for single-cycle and pipelined simulator execution.
2	Before execution, the binary file is loaded into the memory. If PC is 0xFFFF:FFFF, it completes execution, and halts. a) Initial value of register RA is 0xFFFF:FFFF. b) Initial value of register SP is 0x100:0000. c) Initial value of other registers are 0x0000:0000.
3	Implement five-stage pipelined MIPS processor emulator. It should implement IF, ID, EX, MEM, and WB stages. a) For each stages, instruction execution has to be latched. We have to store the execution state for each stages. b) We can sperate instruction memory and data memory. c) We need to generate clock, which is increased only after all the work done in the all stages. d) Resolve data dependency among instructions, either stalling, score-boarding, and forwarding (semantic). e) Resolve control dependency among instructions. We can optionally implement branch prediction mechanisms.
4	At the end of each cycle, the simulator should print out the changed architecture state from the previous state. a) Microarchitecture state includes the following data: general register, PC, memory, internal data structures. b) Microarchitecture state includes the invisible latch values. c) We can only print out only changed state.
5	Program completion/terminal condition: a) At the end of the execution, we need to print out the calculated value. b) If PC moves to 0xFFFF:FFFF, the program prints out the result and terminates. c) The final return value is stored in V0 (or r2) register. d) The printout statistics may include total # of instructions, # of memory operation instructions, # of register operation instructions, # of branch instructions, # of not-taken branches, and # of jump instructions.

Figure 1 - Requirement Specification

Figure 1 shows the requirements for a pipelined MIPS simulator. The implementations for these requirements will be described in detail afterward.

3. Concepts

Before we get into the description of the pipelined MIPS simulator, we will briefly discuss the concepts that are mainly used in the implementation: single-cycle microarchitecture, multi-cycle microarchitecture, pipelined microarchitecture, performance analysis, data dependency handling, control dependency handling, and branch prediction.

3-1. Single-Cycle Microarchitecture

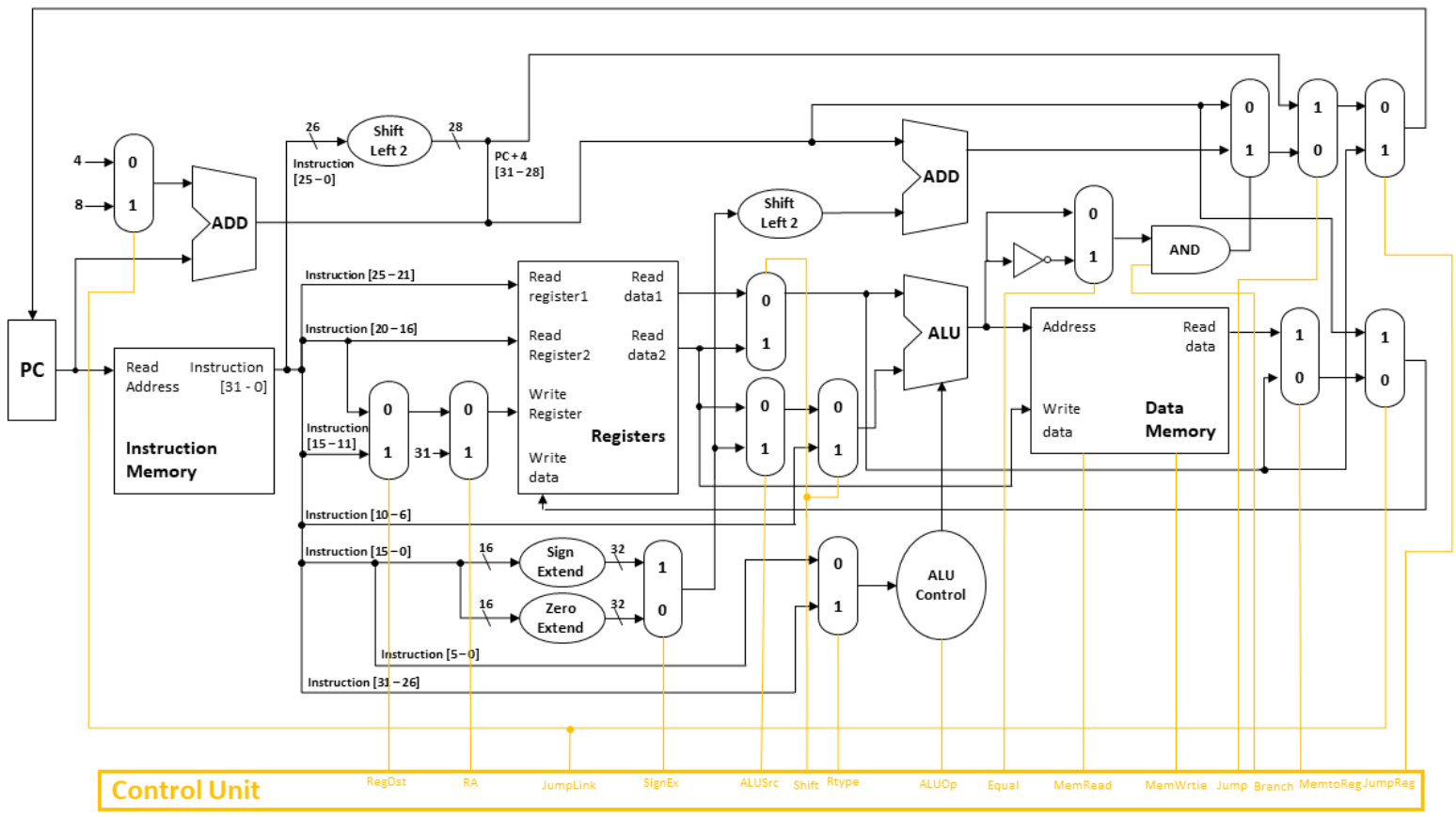


Figure 2 - Single-Cycle Microarchitecture (*Project2*, *CHANGYOON*).

In single-cycle machines, each instruction takes a single clock cycle, and all the states are updated at the end of an instruction's execution (Project2, CHANGYOON). Cycle execution time is determined by the slowest instruction execution time, even though many instructions do not need that long execution time to execute. Therefore, in MIPS ISA, because the instruction that takes the longest time is load word (LW) instruction, one cycle follows the flow of load word (LW) instruction execution (Project2, CHANGYOON). Single-cycle microarchitecture has three main concepts: combinational reading, synchronous writing, and synchronous memory. In Combinational reading, the output of the read data port is a combinational function of the register file contents and the corresponding read select port. In synchronous writing, the selected register is updated with the positive edge clock transition when the write enablement is asserted, and this cannot affect read output in between clock edges. In synchronous memory, contrast this with memory that tells when data is ready. For example, the ready bit indicates that the read or write is done. Figure 2 shows the total data paths of the single-cycle microarchitecture that can execute instructions in the form of the MIPS ISA. The following microarchitecture can execute most of the MIPS instructions except the instructions that deal with floating points.

- According to single-cycle MIPS simulator in Project 2, our former simulator can execute the instruction types below:

1. R-Type and I-Type arithmetic instructions: add, and, nor, or, slt, sub, lui, sll, srl
2. Load Word instructions: lw
3. Store Word instructions: sw
4. Branch Taken and Branch Not Taken instructions: beq, bne
5. Jump instructions: j, jal, jr, jalr

3-2. Multi-Cycle Microarchitecture

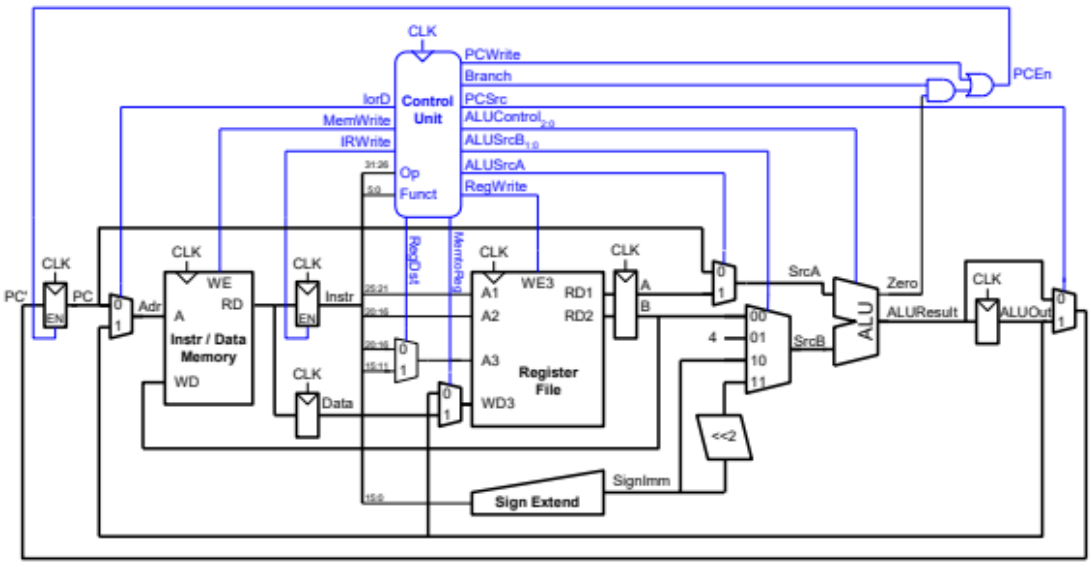


Figure 3 - Multi-Cycle Microarchitecture (*Multi-cycle MIPS processor - ETH Z*)

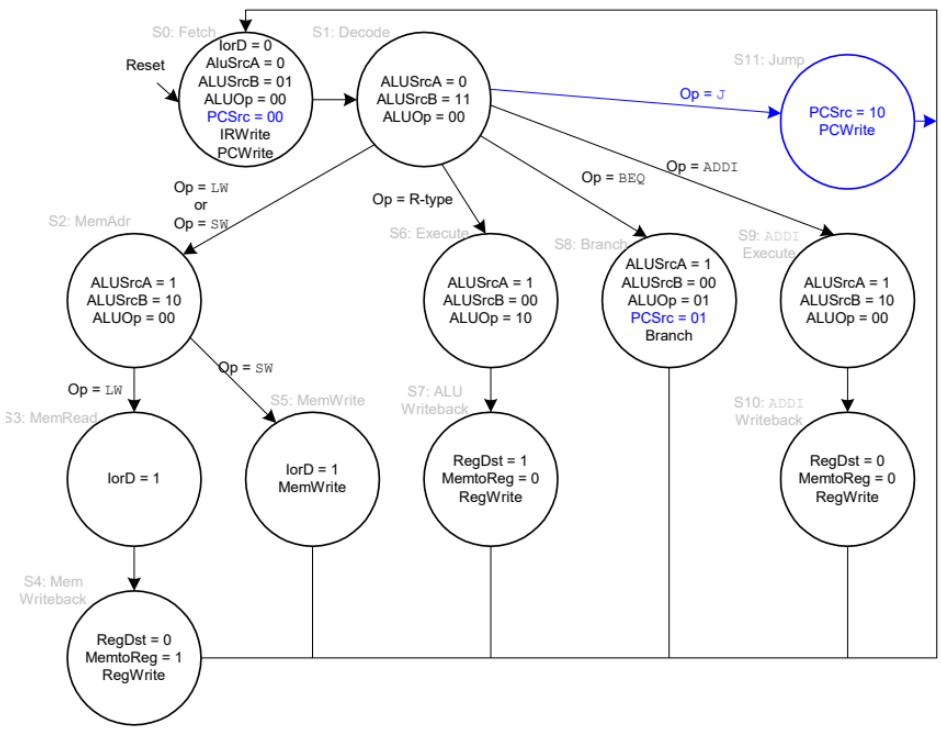


Figure 4 - Controller State (*Multi-cycle MIPS processor - ETH Z*)

Single-cycle microarchitecture has a cycle processing time of load word (LW) instruction, while other instructions have a shorter execution time. It results in architectural inefficiency. To increase the performance of the following microarchitecture, researchers tried to use fewer cycles to perform the instruction and increase the clock frequency. As a result, the concept of multi-cycle has emerged.

A multi-cycle data path breaks up instructions into separate steps. Each step takes a single clock cycle, and each functional unit can be used more than once in instruction if it is used in different clock cycles. This implementation can reduce the amount of hardware needed. In short, it reduces average instruction time.

Multi-cycle microarchitecture offers higher clock speed, a faster execution time for simple instructions, and reusability on expensive hardware, but payment of sequencing overhead can happen frequently (*Multi-cycle MIPS processor - ETH Z*). Figure 3 shows the data paths of multi-cycle microarchitecture. The control unit on the following microarchitecture operates as a state machine. Figure 4 shows the main controller operation due to the current state of multi-cycle microarchitecture. While instruction is executed on multi-cycle microarchitecture, the controller state changes. For every state of the processor, the main controller decides the operation due to the state graph presented in Figure 4.

Multi-cycle microarchitecture allows the operation to take a different number of cycles and reduces the average clock per instruction (CPI) time. Therefore, it can increase the instruction processing performance. However, more complex control of multi-cycle microarchitecture and overheads of the following architecture are the remaining problems to solve.

3-3. Pipelined Microarchitecture

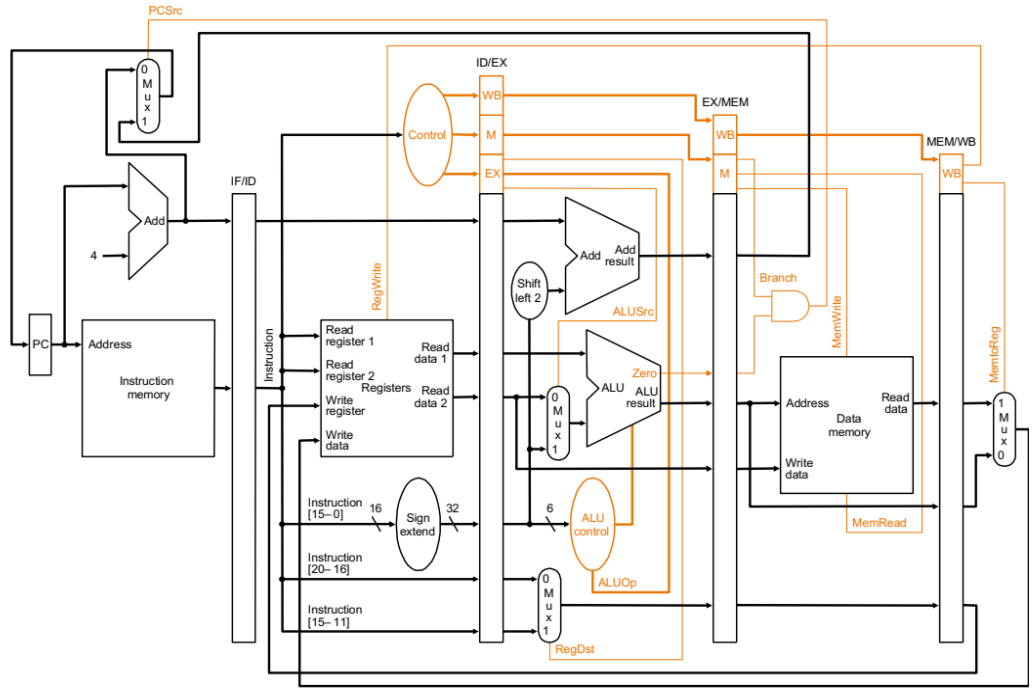


Figure 5 - Pipelined Microarchitecture (*18-447 computer architecture – ECE: course page*)

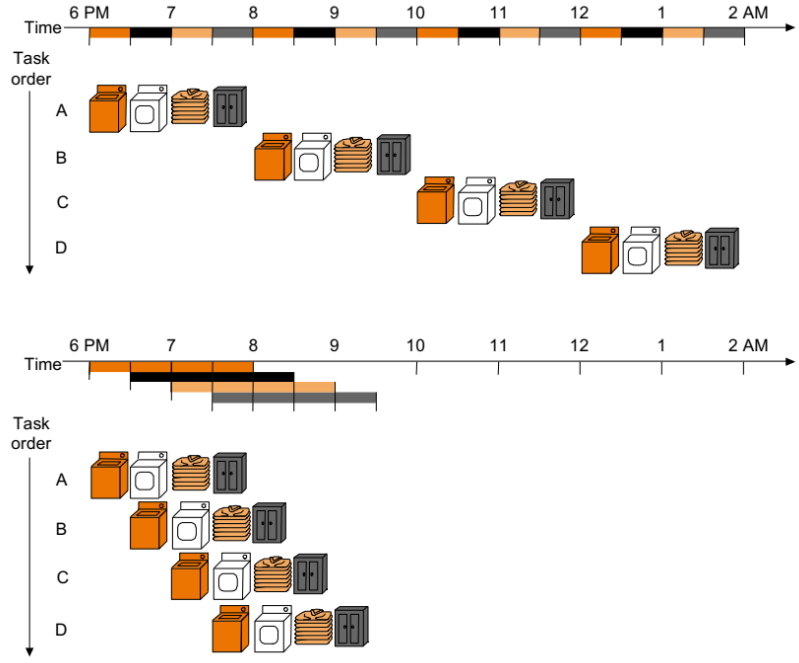


Figure 6 - Pipeline Execution (*18-447 computer architecture - ECE: course page*)

Multi-cycle microarchitecture breaks up instructions into separate steps. Although we separate the steps of the instruction execution in the different clock stages, it is still linearly executed. If we simply configure the multi-cycle microarchitecture, blank spaces occur between each step. Therefore, the concept of the pipeline has emerged.

Pipelined microarchitecture executes multiple instructions at the same time. This method is also called instruction-level parallelism. Figure 6 shows how the instructions are parallelly executed in the pipeline method compared to the linear execution. The basic idea of pipelining is to divide instruction execution into smaller steps. For each stage, one cycle is provided. This method allows us to minimize the cycle time, but to implement this method, we must remember the execution state for every stage (18-447 computer architecture - ECE: course page). In every clock cycle, a new instruction is fetched from the instruction memory. Also, in every clock cycle, each instruction execution stage gets new and different instructions. As a result, the pipeline stage must be dependent only upon the output of the previous stage, which means that there are no more global values in the microarchitecture. To implement a pipelined microarchitecture, we should clearly define the input and output of stages, make each stage execution to be only dependent upon input data, and produce the output data for the next stage. Figure 5 shows the implementation of pipelined microarchitecture that can execute instructions based on the MIPS ISA.

By implementing the following microarchitecture, we can enhance the performance of the processor. The details and implementation of the pipelined microarchitecture will be discussed in section 4.

3-4. Performance Analysis

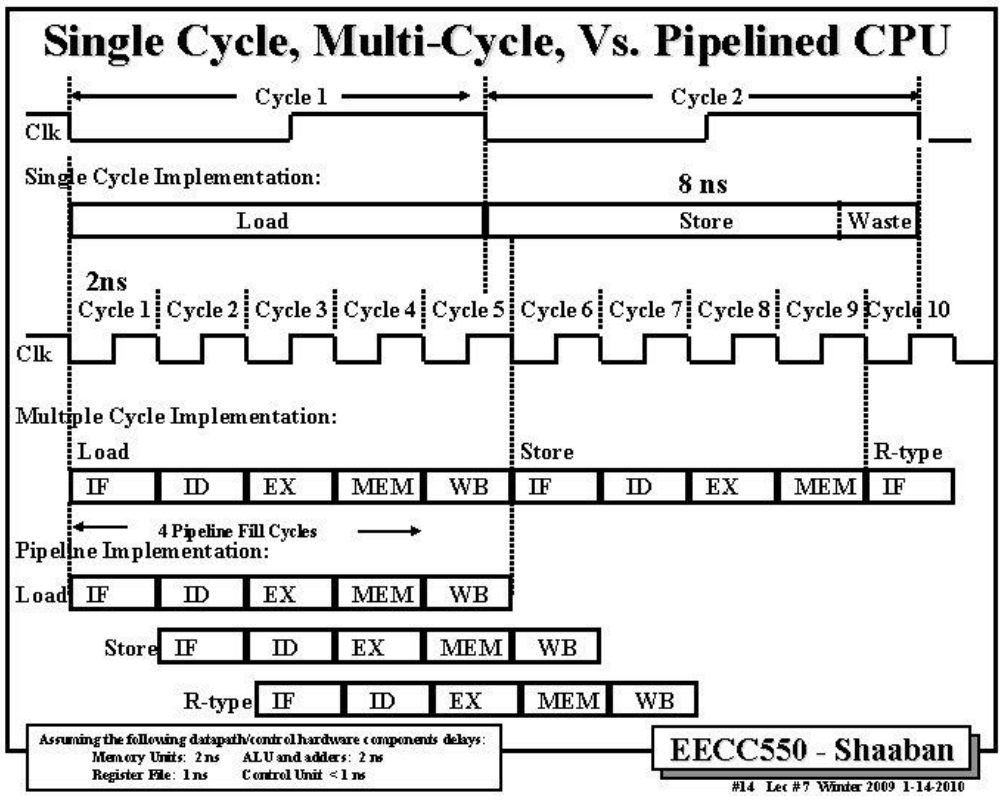


Figure 7 - Performance of the Microarchitectures (*EECC 550 Winter 2012 Home Page*)

Figure 7 shows the clock cycles for single-cycle CPU, multi-cycle CPU, and pipelined CPU. According to the following figure, we can check that when we apply the multi-cycle method, CPI is increased, but clock cycle time is decreased. Therefore, the total performance is increased. In the case of a pipeline, we can decrease the CPI by setting the instructions to be executed in every cycle. Then we can enhance the performance of the processor. As a result, we can check that the pipelined microarchitecture shows the most efficient clock time execution.

3-5. Data Dependency Handling

Ideally, in pipeline execution, the same instructions are executed, independent instructions are also executed, and there is a uniform execution time for each stage. In real pipeline execution, different instructions are coming to be executed, some instructions have a dependency, and it has a non-uniform execution time.

One of the dependencies that happens frequently in pipelined microarchitecture is data dependency. This happens when the two instructions in the pipeline use the same register values. The result should follow the original programming semantics, which also means that it should have the same result as a single-cycle operation. There are three types of data dependency: read-after-write, writ-after-read, and write-after-write.

- Read-after-Write (RAW)

  Flow dependence

  $r_3 \leftarrow r_1$ op r_2

  $r_5 \leftarrow r_3$ op r_4 (Yoo, *10-data_dependency*)

  In this case, r3 in second line is read before it is written in first line. It occurs the data dependency.

- Write-after-Read (WAR)

  Anti dependence

  $r_3 \leftarrow r_1$ op r_2

  $r_1 \leftarrow r_4$ op r_5 (Yoo, *10-data_dependency*)

  If r1 in second line is written before r1 read operation in first line, it occurs data dependency.

- Write-after-Write (WAW)

  Output-dependence

  $r_3 \leftarrow r_1$ op r_2
  $r_5 \leftarrow r_3$ op r_4
  $r_3 \leftarrow r_6$ op r_7 (Yoo, *10-data_dependency*)

  If instruction in third line overwrite the r3 value, then second line instruction will read the wrong r3 value. Then data dependency happens.

How can we handle these data dependencies? Anti and output dependencies are easier to handle. By writing values to the destination in one stage and program order, we can solve the following dependencies. The main data dependency to solve is flow dependences. There are five methods to handle the flow dependences, which are detected and wait, detect and forward/bypass, detect and eliminate, predict the needed value, verify after execution, and do something else (fine-grained multithreading). Detect and eliminate is solved at the software level, so we do not need to detect dependencies at the hardware level. Also, we do not need to detect data dependencies in the do something else method. Predicting the needed value and verifying after execution method is not widely used. Therefore, in this section, we will mainly discuss detect and wait, and detect and forward/bypass methods which are presented in score boarding and forwarding.

3-5-1. Stalling

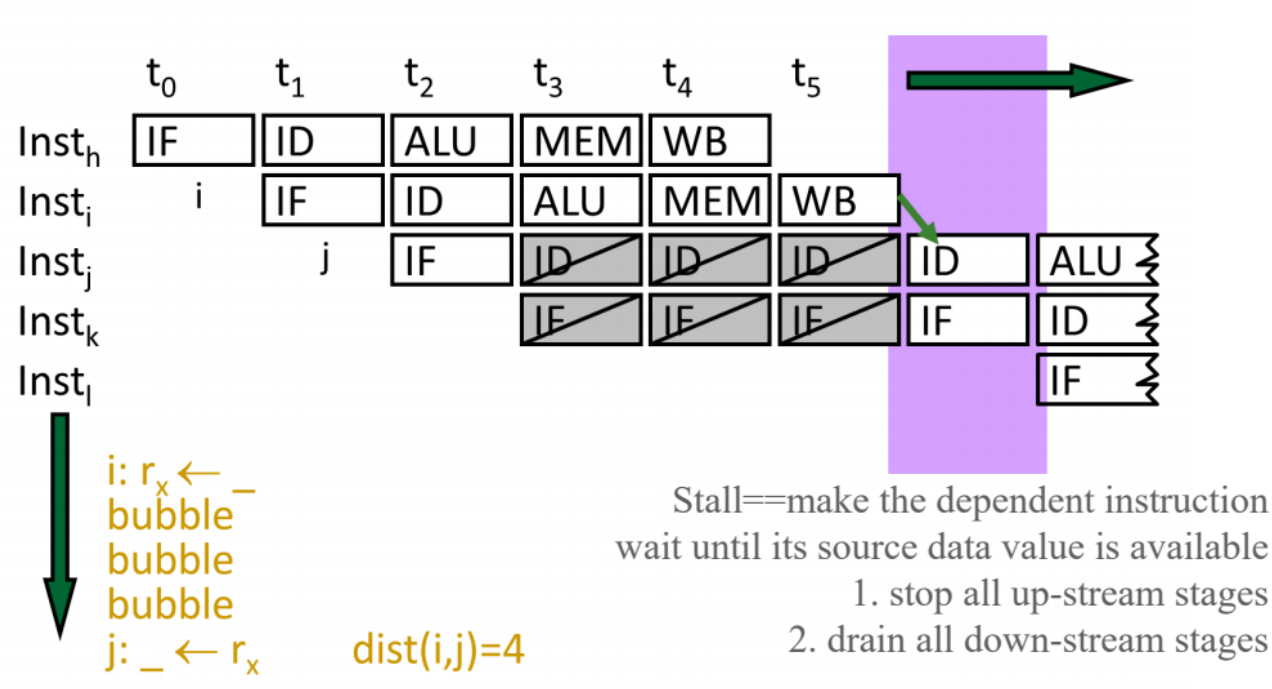


Figure 8 - Pipeline Stall (Yoo, *10-data_dependency*)

- $(rs(IR_{ID})==dest_{EX})$ && $use\_rs(IR_{ID})$ && $RegWrite_{EX}$ or
- $(rs(IR_{ID})==dest_{MEM})$ && $use\_rs(IR_{ID})$ && $RegWrite_{MEM}$ or
- $(rs(IR_{ID})==dest_{WB})$ && $use\_rs(IR_{ID})$ && $RegWrite_{WB}$ or
- $(rt(IR_{ID})==dest_{EX})$ && $use\_rt(IR_{ID})$ && $RegWrite_{EX}$ or
- $(rt(IR_{ID})==dest_{MEM})$ && $use\_rt(IR_{ID})$ && $RegWrite_{MEM}$ or
- $(rt(IR_{ID})==dest_{WB})$ && $use\_rt(IR_{ID})$ && $RegWrite_{WB}$

Figure 9 - Stall Condition (Yoo, *10-data_dependency*)

A stall is a cycle in the pipeline without new input. Data dependency can be resolved by adding stalls between the dependent instructions. Figure 8 shows the pipeline stall case. Data dependency happens between instruction I and instruction j. To avoid this problem, the architecture added a stall in instruction j and delayed the execution of instruction j. The resolving operation for data dependency happens in the condition presented in Figure 9. These stalls occur with additional unrequired waiting delay times. Therefore, they decrease the performance of the processor due to the delay in instruction execution. As a result, it is important to minimize the stalls in the pipeline to optimize the performance of the processor

3-5-2. Score Boarding

Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb	~~1~~→0	1
…	…	1	
31	XXX	1	

Figure 10 - Score Board (Yoo, *10-data_dependency*)

Score boarding is the one of the methods to resolve the data dependency by using valid bit and stall. In score boarding method, each register in the register file has a valid bit associated with each instruction. If an instruction writes some value to the register, it resets the valid bit. Each instruction in the decode stage checks if all source and destination registers are valid. If the valid bit is true, the processor does not have to stall. There is no dependence. However, if the valid bit is false, then it stalls the instruction. The advantage of this method is that it is simple. We can implement this method just by adding one valid bit per register. The disadvantage of this method is that it needs to stall for all types of dependencies, not only flow dependencies.

3-5-3. Forwarding

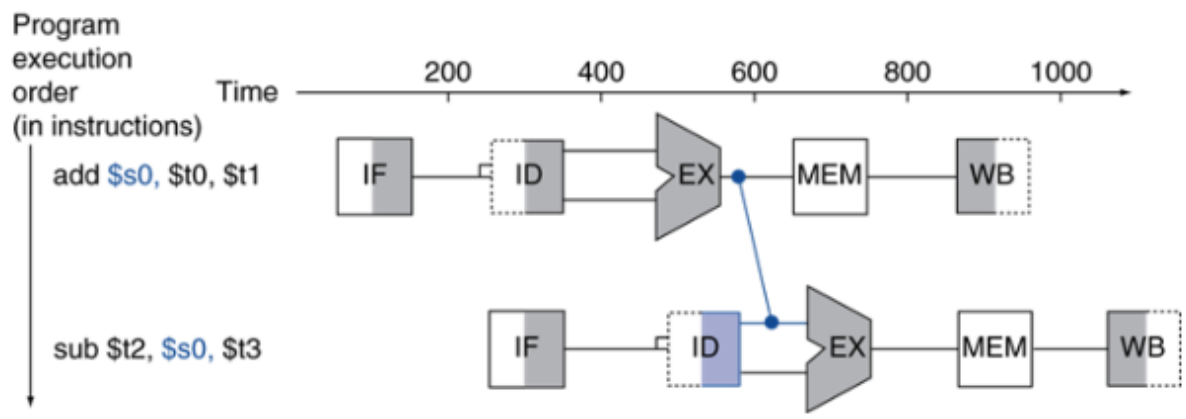


Figure 11 - Case 1 (Distance 1)

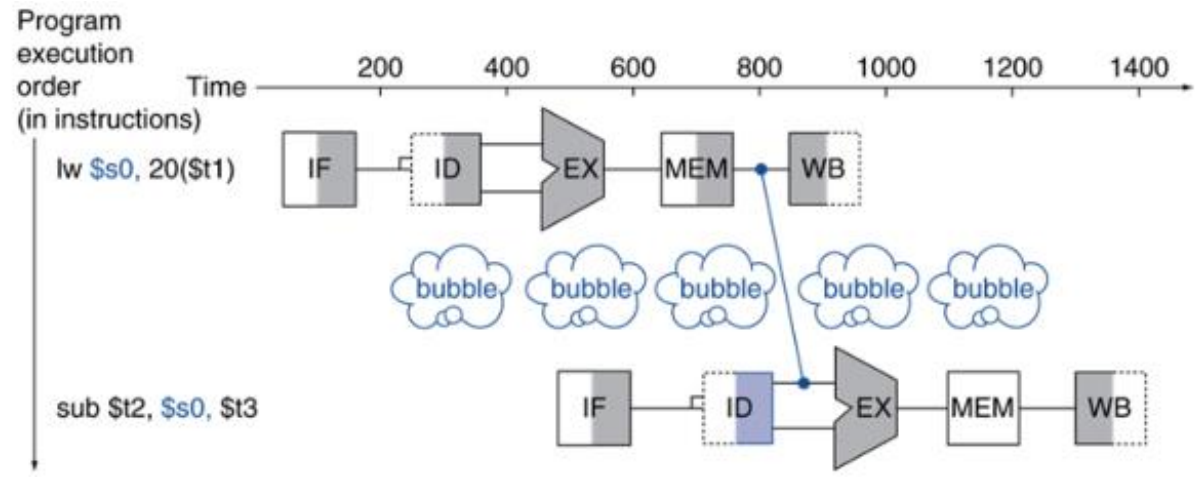


Figure 12 - Case 2 (Distance 2)

if (rs_{EX}!=0) && (rs_{EX}==$dest_{MEM}$) && $RegWrite_{MEM}$ then
  forward operand from MEM stage // dist=1
else if (rs_{EX}!=0) && (rs_{EX}==$dest_{WB}$) && $RegWrite_{WB}$ then
  forward operand from WB stage // dist=2
else
  use A_{EX} (operand from register file) // dist >= 3

Figure 13 - Forwarding Conditions (Yoo, *10-data_dependency*)

Stall decreases the performance of the processor. Therefore, if we can resolve the data dependency with lower number of stalls, the performance of the processor will increase. One of the ways to resolve data dependency without stall is using combinational dependence check logic. Combinational dependence check logic is a special logic that checks if any instruction in later stages is supposed to be written to any source register of the instruction that is being decoded (18-447 computer architecture - ECE: course page). In this method, we do not need to stall on anti and output dependences. However, following logic is more complex than a scoreboard, and logic becomes more complex as we make the pipeline deeper and wider.

One of the combinational dependence check logics is data forwarding/bypassing. The problem started in data dependency is that a dependent instruction must wait in the decode stage until the producer instruction writes its value in the register file. The goal of this method is to avoid stalling the pipeline unnecessarily. The data forwarding/bypassing method observes the data value needed by the dependent instruction, and the needed values are supplied directly from a later stage in the pipeline right after the values are available. By applying this method, dependent instruction can move into the pipeline until the point the value can be supplied, and this decreases the number of stalling.

In the data forward/bypassing method, we should consider 3 types of dependent cases. Figure 11 shows the first case of data dependency. As we can check from the figure, the second instruction is to use the value in the decode stage, which is calculated from the execution stage in the first instruction. In this case, the distance between the stages is 1. Figure 12 shows the second case of data dependency. The figure presents the situation of the second instruction using the value in the decode stage, which is the real value from the data memory in the first instruction. In this case, the distance between the stages is 2. The last case is the situation of the distance between the stages being bigger than equal to 3. In this case, we can ignore this case by ordering the decode stage after the writeback stage. Figure 13 shows the conditions that include the cases presented above.

By implementing the hardware component that detects and operates in the condition presented in Figure 13, we can apply the data forward/bypassing method and optimize the processor. The details and implementation will be discussed in section 4.

3-6. Control Dependency Handling

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional (BEQ, BNE)	Unknown	2	Execution (register dependent)
Unconditional (J)	Always taken	1	Decode (PC + offset)
Call (JAL)	Always taken	1	Decode (PC + offset)
Return (JR)	Always taken	Many	Execution (register dependent)
Indirect (JR)	Always taken	Many	Execution (register dependent)

Figure 14 - Jump Types (Yoo, *11-control_dependence*)

Another dependency that we must resolve in pipelined microarchitecture is handling control dependency. Control dependency deals with the problem of what the fetch PC should be in the next cycle. From this perspective, all the instructions are controlled depending on the previous instruction (Yoo, 11-control_dependence). If the fetched instruction is a non-control-flow instruction, the next fetch PC is the address of the next sequential instruction. If we know the size of the fetched instruction, we can easily determine the next-sequential instruction's address. However, if the instruction that is fetched is a control-flow instruction, we must build the logic to determine the next fetch PC. Figure 14 shows the control-flow instructions in MIPS ISA. In the case of jump (J) and jump and link (JAL) instructions, they are always taken in the decode stage. One additional instruction that is in the address right after the following instructions are fetched. In the case of jump register (JR) instruction, it is always taken in the execution stage. We believe that it updates the PC value in the decode stage, however, due to the forwarding due to data dependency, it should be updated in the execution stage. Two additional instructions in the address right after the following instructions are fetched. The instructions that have more complicated situations are branch instructions, such as branch equal (BEQ) and branch not equal (BNE). In this case, it shares the common situation with jump register (JR) instruction, but also, it must be determined whether the branch operation is taken or not.

Handling control dependency is critical to keeping the pipeline full of the correct sequence of dynamic instructions. There are several methods to resolve the control dependences: stall, branch prediction, branch delay slot, fine-grained multithreading, predicated execution, and multipath execution.

One of the dependencies that happens frequently in pipelined microarchitecture is data dependency. This happens when the two instructions in the pipeline use the same register values. The result should follow the original programming semantics, which also means that it should have the same result as a single-cycle operation. There are three types of data dependency: read-after-write, writ-after-read, and write-after-write. Stalling the pipeline until we know the next fetch address is the simplest way to resolve the control dependency, but it is not an efficient way. The branch delay slot is implemented by the software, such as the compiler. Fine-grained multithreading predicated execution and multipath execution will not be covered in this project. Therefore, only branch prediction will be covered in this project.

3-7. Branch Prediction

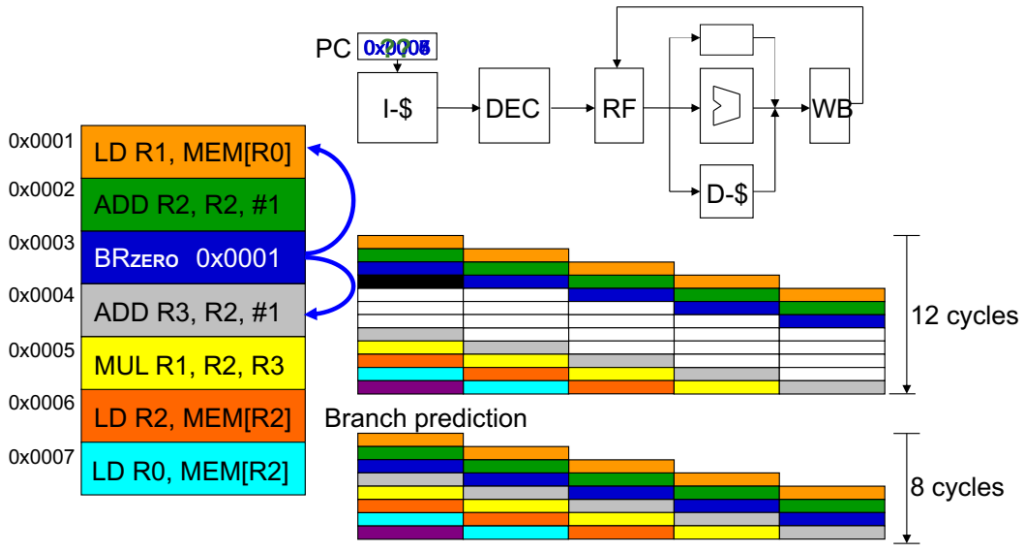


Figure 15 - Branch Prediction (Yoo, *11-control_dependence*)

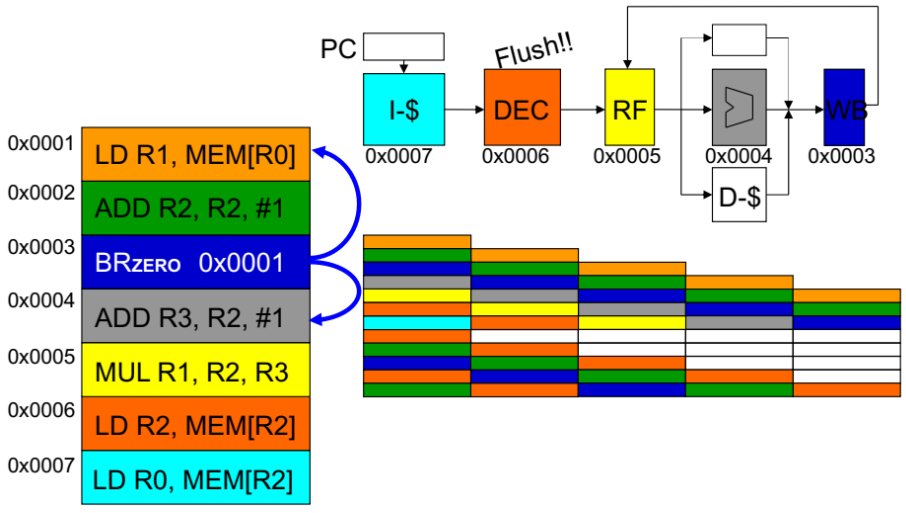


Figure 16 - Misprediction Penalty (Yoo, *11-control_dependence*)

Branch prediction is based on guessing the next fetch instruction. By guessing the next instruction when a branch is fetched, we can keep the pipeline full. This method requires guessing the direction and target of a branch. (Yoo, *11-control_dependence*). Figure 15 shows the difference in the execution between not applying the branch prediction and applying the branch prediction. Due to the fewer stalls in branch prediction applied execution, we can check that branch prediction can increase the performance of the processor. However, if the branch prediction misses, there is a penalty. Figure 16 shows the penalty for branch prediction missing. If the branch prediction fails, the processor flushes the instructions and values in the decode stage. This also means that cycle execution is delayed.

In short, if the prediction is correct, there is no penalty. However, if the prediction fails, 2 bubbles are inserted. Therefore, to decrease the penalty of the branch prediction to increase the processor's performance, it is important to reduce the ratio of a branch misprediction. There are two main methods for branch prediction: compile-time prediction and run-time prediction.

3-7-1. Compile Time Prediction (Static)

In compile time prediction, which is also known as static prediction, it executes the predict-not-taken or predict-taken operation. However, if the misprediction occurs, it flushes the instructions and retries the execution from the branched instruction. This can be so-called as the structure of the microarchitecture is determined due to the instruction type. the following method shows a low hit rate of branch prediction, and therefore, it is inefficient. In this project, we will cover two methods that are based on the compile-time prediction which are always-not-taken and always-taken.

3-7-2. Run Time Prediction (Dynamic)

Due to the inefficiency of compile-time prediction, runtime prediction has come out. The main idea of the dynamic branch prediction is to predict the next fetch address in the run time. Run time prediction requires three things to be predicted at the fetch stage: determination of whether the fetched instruction is a branch or not, conditional branch direction, and branch target address. To fulfill the requirements of the following method, we need to store the target address from the previous instance and access it with the PC. As a result, the branch target buffer (BTB) has come out. The usage of the branch target buffer will be discussed in section 4. In this project, we will cover four methods that are based on the run time prediction which are 1-bit last time prediction, 2-bit counter-based prediction, 2-level global prediction, and 2-level local prediction.

4. Pipelined Microarchitecture

In this section, we will mainly discuss the ideas and methods to implement our pipelined MIPS simulator. In pipelined microarchitecture, multiple instructions are executed in the same cycle. In every clock cycle, a new instruction is fetched from the instruction memory, and each instruction execution stage gets new and different instructions. Our implementation of pipelined MIPS simulator includes these concepts, and it is implemented by following pipelined data paths, latches, forward units, hazard detection units, and various branch predictors.

4-1. Implemented Datapath

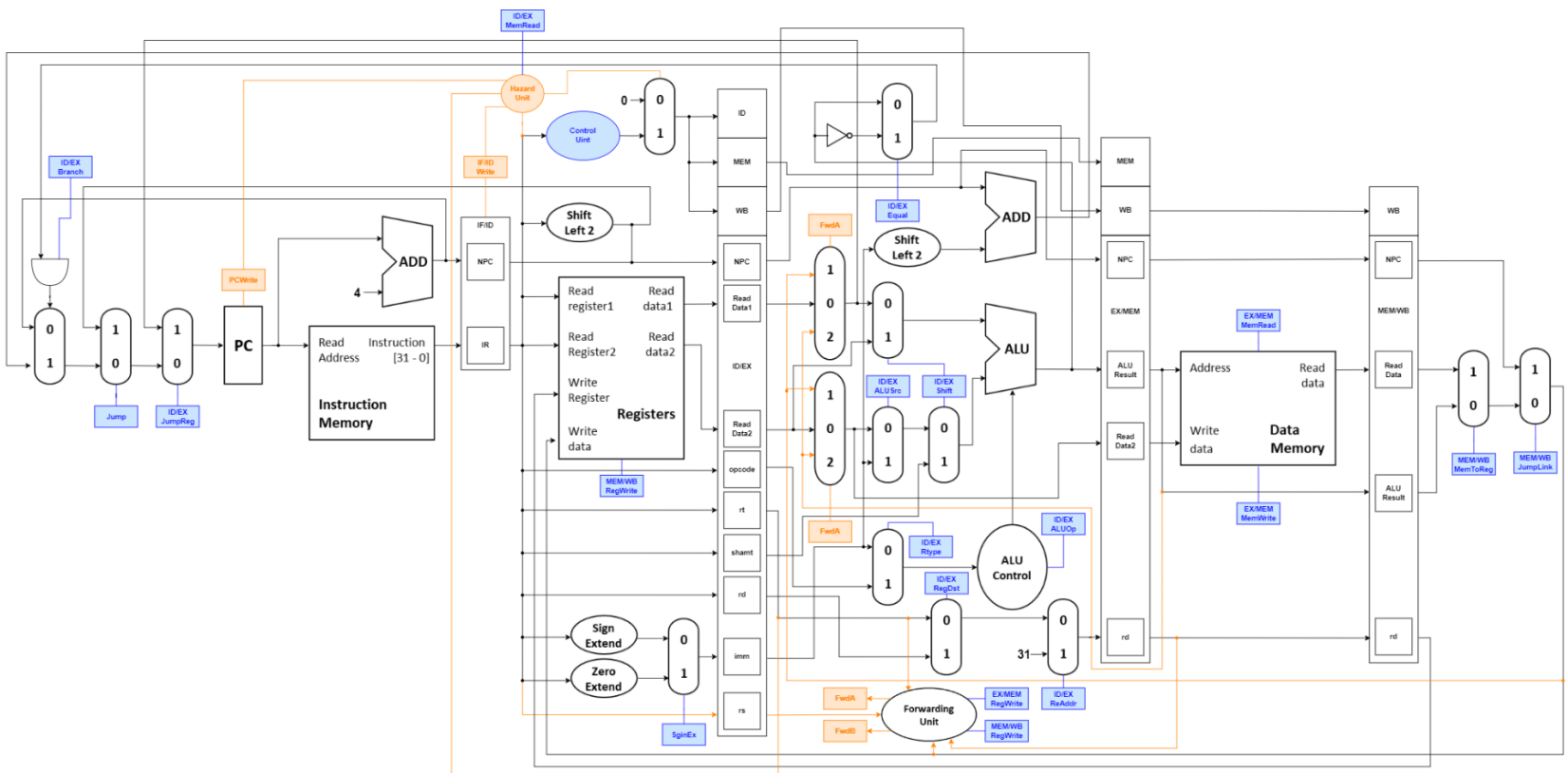


Figure 17 - Pipelined Microarchitecture

Figure 17 shows the total data paths of our own pipelined microarchitecture that supports MISP ISA. This structure contains the concepts which are presented above sections, such as pipeline, data dependency control, etc. However, we did not express the detailed structures of latches and branch predictors. They will be discussed in later sections.

4-2. Latch

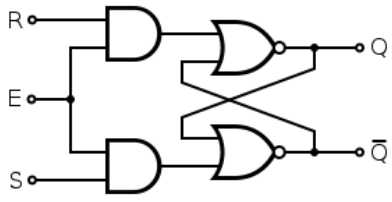


Figure 18 - Flip Flop

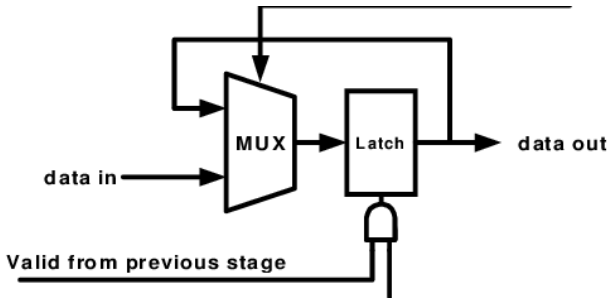


Figure 19 - Latch in Pipeline (*A pipeline latch. A valid bit from the previous stage is used to gate ...*)

In pipeline implementation, each stage composes of a functional unit followed by a latch, and all latches are synchronized (Chongstitvatana, *Pipeline*). The latch is built using various flip-flops, which are represented in Figure 18. The number of stages determines the increment of the processing speed up. When a task is divided into several stages which enable overlap operations total time for executing tasks can be shortened but the delay (setting time of latch) of each stage remains constant. This delay time is the limit of speedup. Therefore, the latch circuit must be very fast in the pipelined microarchitecture.

The pipeline is not always filled. When the flow of a pipeline is interrupted, all stages before interruption are stopped, which is called a stall. The rest of the pipeline can continue to function. To control this operation, the latch contains a valid bit. A valid bit from the previous stage is used to gate the clock signal. The hold signal from the succeeding stage has used the multiplexer to recirculate data from being stalled (a *pipeline latch. A valid bit from the previous stage is used to gate ...*). The structure of the latch is presented in Figure 19.

By applying latch to the architecture, we can implement the pipelined microarchitecture processor. One consideration is that the accessing time of the latch should be shorter than the clock cycle time. If it gets longer than the clock cycle time, this will occur inefficiency of the microarchitecture when the total cycle of the architecture is divided into more and more stages. Nowadays, the accessing time to the latch is almost the same as the clock cycle time of the processor.

4-3. Data Forward Unit

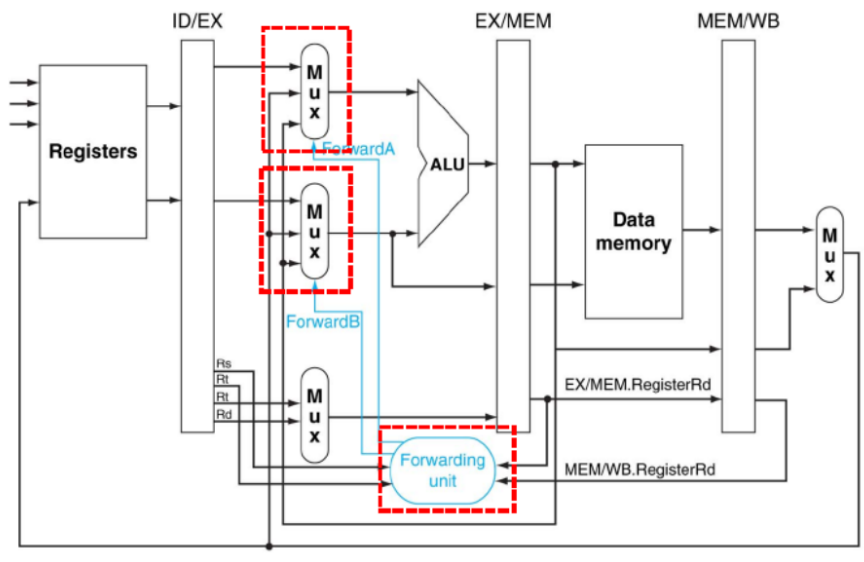


Figure 20 - Data Forwarding Unit (Hayes, *Computer Architecture and organization* 1978)

MUX Control Signal	Single Value	MUX Selection
ForwardA	00	First ALU operand = ID/EX.A
	10	First ALU operand = EX/MEM.AR
	01	First ALU operand = MEM/WB.A or MEM/WB.LR
ForwardB	00	Second ALU operand = ID/EX.A
	10	Second ALU operand = EX/MEM.AR
	01	Second ALU operand = MEM/WB.A or MEM/WB.LR

Figure 21 - Conditions for Forwarding Control Signals

As we mentioned in section 3-5-3, data forwarding can resolve the data dependency with decreased number of stalls compare to the stalling and score boarding method. Figure 20 shows the implementation of forward unit in the pipelined microarchitecture. To implement the data forwarding method to the architecture, we must observe and determine whether the data dependency happened or not. Detection of data dependency follows the conditions below:

- If register number is 0, it must not be forwarded.
- Former instruction must be the one that writes the value into the register file.
- Compare the data with register number:
  1. EX/MEM.RD == ID/EX.RS
  2. EX/MEM.RD == ID/EX.RT
  3. MEM/WB.RD == ID/EX.RS
  4. MEM/WB.RD == ID/EX.RT

These conditions are presented as a table in Figure 21. By implementing forward unit to operate in these conditions, we can apply the data forwarding method, and resolve the data dependency of pipelined microarchitecture.

4-4. Hazard Detection Unit

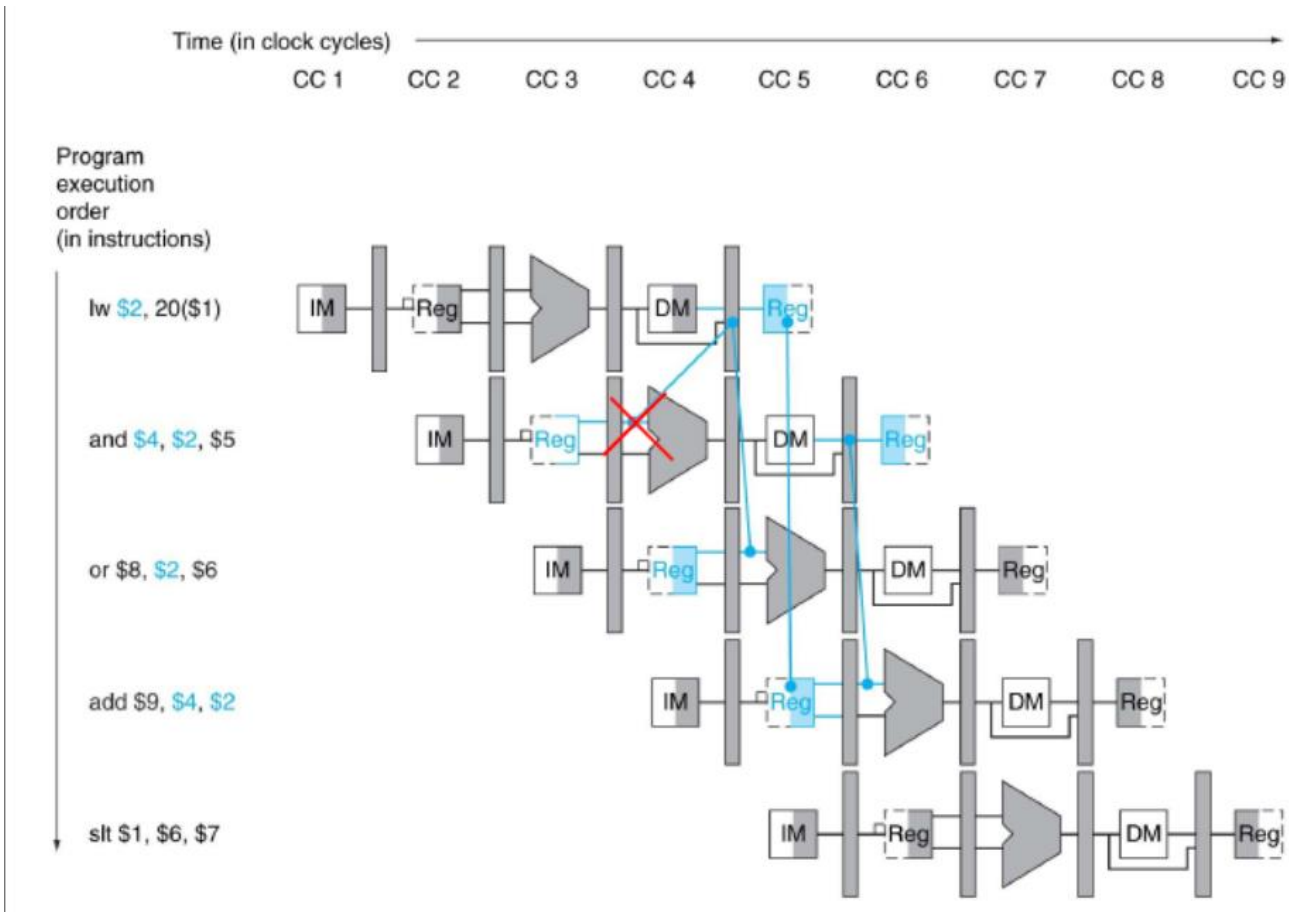


Figure 22 - Load Use Data Dependency (Hayes, *Computer Architecture, and organization* 1978)

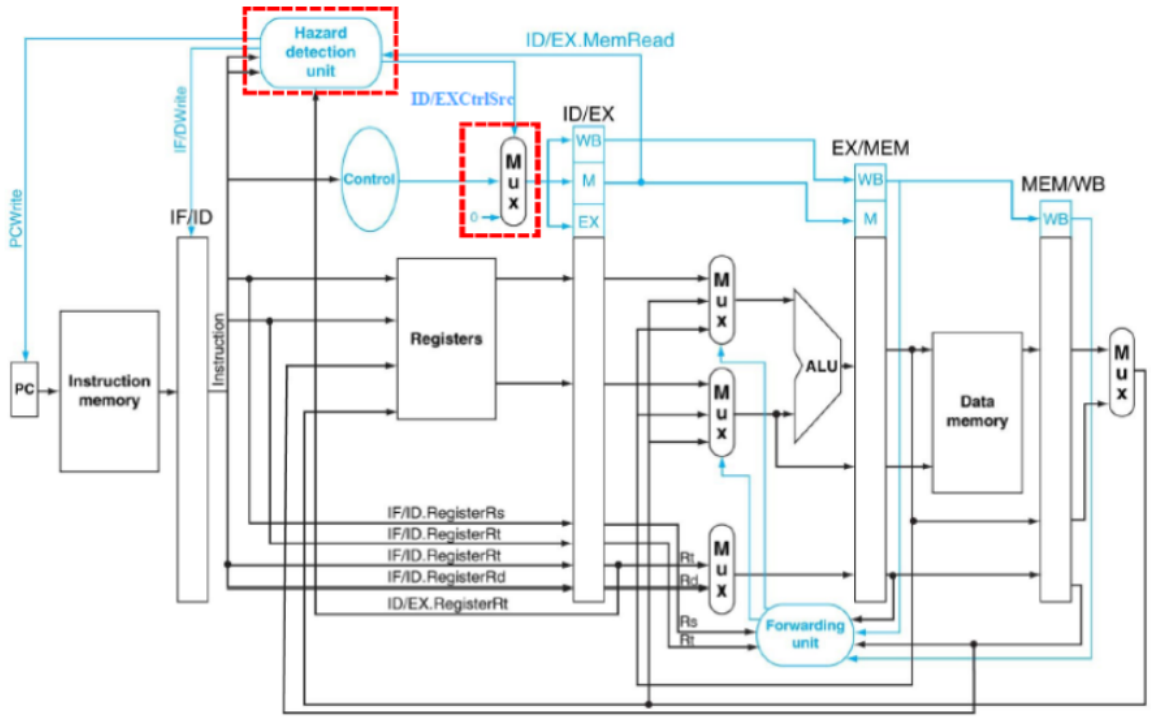


Figure 23 - Hazard Detection Unit (Hayes, *Computer Architecture, and organization* 1978)

Even though we resolve most of the data dependencies by applying the data forwarding, there is still data dependency that is not resolved. Unresolved data dependency is a load-use case. In Figure 22, the instruction that comes right after load instruction requires the result from load instruction. The solution to the following data dependency is to stall the pipeline, which is also called generating the pipeline bubble. Pipeline stalls in the following method:

1. Stall the data-dependent instructions for 1 cycle in the instruction decode (ID) stage.
2. Blocks the update of IF/ID registers and a PC.
3. Insert NOP instructions in the ID/EX register.

To implement the following conditions, the hazard detection unit presented in Figure 23 should generate the following control signals: PCWrite, IF/IDWrite, ID/EXCtrlSrc. PCWrite signal determines whether to write the PC value or not. IF/IDWrite signal determines to write IF/ID registers, and ID/EXCtrlSrc chooses the control input to the ID/EX registers. By applying a hazard detection unit to the microarchitecture, we can resolve the load-use data dependency.

4-5. Branch Prediction

Branch prediction can be one of the methods to resolve the control dependency in the architecture. As mentioned in section 3.7, if we apply branch prediction and predict the branch condition, we can avoid stalling. However, misprediction of branch conditions occurs with a penalty of a single clock cycle. These operations can be implemented when the branch execution is operated in the instruction decode (ID) stage. Branch execution follows the following operations:

1. Calculate the branch target address in the instruction decode (ID) stage.
    - Move adder for branch target address calculation, from execution (EX) stage to instruction decode (ID) stage.
    - PC + 4 value if already exists in IF/ID.NPC
    - Offset that must be added with the PC can be generated by IF/ID.IR[15:0].
    - branch target address is calculated for all the instructions and if it is not necessary, it does not use it.

2. Determine the branch decision in the instruction decode (ID) stage.
    - Compare the rs and rt values from the register file.
    - Add a comparator in the instruction decode (ID) stage.

In section 5, the various methods of branch prediction will be discussed. By implementing these branch prediction methods and applying them in the microarchitecture, we can resolve the control dependency of the architecture.

4-6. Program Definition

Before implementing the pipelined MISP simulator in the physical schema, we will state the additional program definitions from the project 2, the single-cycle MIPS simulator, that will be used in the real implementation.

- Global Variables

Variables	Data Type	Definition
PREDICTOR	Char	Predictor Index: 1. ANT, 2. ALT, 3. LTP1, 4. LTP2, 5. GSH, 6. LVL2
BHTENTRY	0x100	Branch History Table entry size
PHTENTRY	0x100	Pattern History Table entry size
LHTENTRY	0x100	Local History Table entry size
FwdA	Uint8_t	Forward A control signal
FwdB	Uint8_t	Forward B control signal
PCWrite	Bool	PCWrite control signal
IFIDWrite	Bool	IFIDWrite control signal
IDEXCtrlSrc	Bool	IDEXCtrlSrc control signal
IFID	IFID_LAT	IF/ID Latch
IDEX	IDEX_LAT	ID/EX Latch
EXMEM	EXMEM_LAT	EX/MEM Latch
MEMWB	MEMWB_LAT	MEM/WB Latch
GHR	Uint8_t	Global History Register
PHT	Uint8_t	Pattern History Table
LHT	Uint8_t	Local History Table
BHT	BHT_ELEMENT	Branch History Table which has following entries: 1. Branch instruction address 2. Branch target address 3. Prediction bits
branch_hit	Int	Counter of branch hit operation
branch_miss	Int	Counter of branch miss operation

- Modules and Functions

Modules	Functions	Definition
MUX	MUX2	If the given signal is true, it returns the second input data. Else, it returns first input data.
	MUX3	If the given signal is 0 -> return input1, 1 -> return input2, 2 -> return input3.
LAT	LAT_Init	Initialize the latches. Set all the values into 0.
	LAT_Update	Update the latches. Shift the values from right to left.
	IDEX_LAT_WIRE	Wire the data into the ID/EX latch.
	IDEX_EXMEM_CPY	Copy the control signals in ID/EX latch into EX/MEM latch.
	EXMEM_LAT_WIRE	Wire the data into the EX/MEM latch.
	EXMEM_MEMWB_CPY	Copy the control signals in EX/MEM latch into MEM/WB latch.
	MEMWB_LAT_WIRE	Wire the data into the MEM/WB latch.
FW	FW_Init	Initialize the forward unit variables into 0.
	FW_Operation	Operate the forward unit and generates the forward control signals, such as FwdA an FwdB.
HU	HU_Init	Initializae the hazard detection unit variables into 0.
	HU_Operation	Operate the hazard detection unit and generates the control signals, such as PCWrite, IFIDWrite, and IDEXCtrlSrc.
STT	ANT_Execution	Execute the branch prediction operation of Always Not Taken predictor.
	ALT_Init	Initialize the branch prediction variables used in Always Taken predictor.
	ALT_Prediction	Predict the branch prediction operation of Always Taken predictor.
	ALT_Execution	Execute the branch prediction operation of Always Taken predictor.
LTP1	LTP1_Init	Initialize the branch prediction variables used in 1-bit Last Time predictor.
	LTP1_Prediction	Predict the branch prediction operation of 1-bit Last Time predictor.
	LPT1_Execution	Execute the branch prediction operation of 1-bit Last Time predictor.
LTP2	LTP2_Init	Initialize the branch prediction variables used in 2-bit Last Time predictor.
	LTP2_Prediction	Predict the branch prediction operation of 2-bit Last Time predictor.
	LTP2_Execution	Execute the branch prediction operation of 2-bit Last Time predictor.
GSH	GSH_Init	Initialize the branch prediction variables used in Global Share predictor.
	GSH_Prediction	Predict the branch prediction operation of Global Share predictor.
	GSH_Execution	Execute the branch prediction operation of Global Share predictor.
LVL2	LVL2_Init	Initialize the branch prediction variables used in 2-level Local predictor.
	LVL2_Prediction	Predict the branch prediction operation of 2-level Local predictor.
	LVL2_Execution	Execute the branch prediction operation of 2-level Local predictor.
LOG	LOG_Init	Initialize the counter variables used in logs.
	LOG_Print	Print-out the counter values in the form of the logs. It contains following information: 1. Cycle count 2. Total instruction 3. Memory access operation 4. Register write operation 5. Branch taken operation 6. Branch prediction hit 7. Branch prediction miss 8. Jump operation
CPU	IF	Instruction Fetch stage.
	ID	Instruction Decode stage.
	EX	Execution stage.
	MEM	Memory Access stage.
	WB	Write Back stage
	Init	Initialization.
	terminate	Termination.

5. Implementation

5-1. Latch (LAT)

```
/* initialize latches of each stage*/
void LAT_Init() {
    IFID[0].valid = false;
    IDEX[0].valid = false;
    EXMEM[0].valid = false;
    MEMWB[0].valid = false;

    memset(&IFID[0], 0, sizeof(IFID_LAT));
    memset(&IFID[1], 0, sizeof(IFID_LAT));
    memset(&IDEX[0], 0, sizeof(IDEX_LAT));
    memset(&IDEX[1], 0, sizeof(IDEX_LAT));
    memset(&EXMEM[0], 0, sizeof(EXMEM_LAT));
    memset(&EXMEM[1], 0, sizeof(EXMEM_LAT));
    memset(&MEMWB[0], 0, sizeof(MEMWB_LAT));
    memset(&MEMWB[1], 0, sizeof(MEMWB_LAT));
}


/* update latches */
void LAT_Update() {
    memcpy(&IFID[1], &IFID[0], sizeof(IFID_LAT));
    memcpy(&IDEX[1], &IDEX[0], sizeof(IDEX_LAT));
    memcpy(&EXMEM[1], &EXMEM[0], sizeof(EXMEM_LAT));
    memcpy(&MEMWB[1], &MEMWB[0], sizeof(MEMWB_LAT));

    memset(&IFID[0], 0, sizeof(IFID_LAT));
    memset(&IDEX[0], 0, sizeof(IDEX_LAT));
    memset(&EXMEM[0], 0, sizeof(EXMEM_LAT));
    memset(&MEMWB[0], 0, sizeof(MEMWB_LAT));
}
```

Figure 24 - Latch Init and Update

```
/* wire the values into IDEX latch*/
void IDEX_LAT_WIRE() {
    IDEX[0].rs = inst->rs;
    IDEX[0].rt = inst->rt;
    IDEX[0].rd = inst->rd;
    IDEX[0].NPC = IFID[1].NPC;
    IDEX[0].funct = inst->funct;
    IDEX[0].shamt = inst->shamt;
    IDEX[0].opcode = inst->opcode;
    IDEX[0].BHTIdx = IFID[1].BHTIdx;
    IDEX[0].PHTIdx = IFID[1].PHTIdx;
    IDEX[0].LHTIdx = IFID[1].LHTIdx;
    IDEX[0].Control.PreTaken = IFID[1].PreTaken;
    IDEX[0].Control.BHTFound = IFID[1].BHTFound;
    IDEX[0].imm = MUX2(ZeroExtend(inst->immediate), SignExtend(inst->immediate), SignEx);
}


/* copy the signals and values from IDEX latch to EXMEM latch */
void IDEX_EXMEM_CPY() {
    EXMEM[0].Control.ALUOp = IDEX[1].Control.ALUOp;
    EXMEM[0].Control.ALUSrc = IDEX[1].Control.ALUSrc;
    EXMEM[0].Control.Branch = IDEX[1].Control.Branch;
    EXMEM[0].Control.Equal = IDEX[1].Control.Equal;
    EXMEM[0].Control.JumpLink = IDEX[1].Control.JumpLink;
    EXMEM[0].Control.JumpReg = IDEX[1].Control.JumpReg;
    EXMEM[0].Control.MemRead = IDEX[1].Control.MemRead;
    EXMEM[0].Control.MemToReg = IDEX[1].Control.MemToReg;
    EXMEM[0].Control.MemWrite = IDEX[1].Control.MemWrite;
    EXMEM[0].Control.RegWrite = IDEX[1].Control.RegWrite;
    EXMEM[0].Control.Rtype = IDEX[1].Control.Rtype;
    EXMEM[0].Control.Shift = IDEX[1].Control.Shift;
}
```

Figure 25 - Latch Wire and Copy 1

```
/* wire the values into EXMEM latch*/
void EXMEM_LAT_WIRE() {
    EXMEM[0].NPC = IDEX[1].NPC;
    EXMEM[0].bcond = EXMEM[0].ALUResult;
    EXMEM[0].BrTarget = BranchAddr(IDEX[1].NPC, IDEX[1].imm);
    EXMEM[0].rd = MUX2(MUX2(IDEX[1].rt, IDEX[1].rd, IDEX[1].Control.RegDst), ra, IDEX[1].Control.ReAddr);
}


/* copy the signals and values from EXMEM latch to MEMWB latch*/
void EXMEM_MEMWB_CPY() {
    MEMWB[0].Control.MemToReg = EXMEM[1].Control.MemToReg;
    MEMWB[0].Control.RegWrite = EXMEM[1].Control.RegWrite;
    MEMWB[0].Control.JumpLink = EXMEM[1].Control.JumpLink;
}


/* wire the values into MEMWB latch */
void MEMWB_LAT_WIRE() {
    MEMWB[0].rd = EXMEM[1].rd;
    MEMWB[0].NPC = EXMEM[1].NPC;
    MEMWB[0].ALUResult = EXMEM[1].ALUResult;
}
```

Figure 26 - Latch Wire and Copy 2

The figures above show the functions that are used in the latch operation. LAT_Init() function initialize all of the valid bits in each latch into false, and sets values in latches into 0. LAT_Update() function shifts all of the values in each latch, from left to right, and sets the left values to 0. The functions that end with “WIRE” wire the values that are generated by the instruction from the previous or current stage. Functions that end with "CPY” copy the control signals from the previous stage. By implementing the latch functions which are presented in the figures above, we can control the latches in each stage.

5-2. Forward Unit (FW)

```
/* initialize forward unit signal */
void FW_Init() {
    FwdA = 0b00;
    FwdB = 0b00;
}


/* forward unit operation for data dependency control */
void FW_Operation() {
    FwdA = 0b00;
    FwdB = 0b00;

    // EXMEM forwarding
    if (EXMEM[0].rd != 0 && EXMEM[0].Control.RegWrite) {
        if (EXMEM[0].rd == IDEX[0].rs) {
            FwdA = 0b10;
        }
        if (EXMEM[0].rd == IDEX[0].rt) {
            FwdB = 0b10;
        }
    }

    // MEMWB forwarding
    if (MEMWB[0].rd != 0 && MEMWB[0].Control.RegWrite) {
        if (EXMEM[0].rd != IDEX[0].rs && MEMWB[0].rd == IDEX[0].rs) {
            FwdA = 0b01;
        }
        if (EXMEM[0].rd != IDEX[0].rt && MEMWB[0].rd == IDEX[0].rt) {
            FwdB = 0b01;
        }
    }
}
```

Figure 27 – Forwarding Init and Operation

- For Register rs
  if (EX/MEM.RegWrite
    and (EX/MEM.RD ≠ 0)
    and (EX/MEM.RD == ID/EX.RS))
  ForwardA =10
- For Register rt
  if (EX/MEM.RegWrite
    and (EX/MEM.RD ≠ 0)
    and (EX/MEM.RD == ID/EX.RT))
  ForwardB =10
- For Register rs
  if (MEM/WB.RegWrite
    and (MEM/WB.RD ≠ 0)
    and ((EX/MEM.RD≠ID/EX.RS) or (EX/MEM.RegWrite==0))
    and (MEM/WB.RD == ID/EX.RS))
  ForwardA =01
- For Register rt
  if (MEM/WB.RegWrite
    and (MEM/WB.RD ≠ 0)
    and ((EX/MEM.RD≠ID/EX.RS) or (EX/MEM.RegWrite==0))
    and (EX/MEM.RD == ID/EX.RT))
  ForwardB =01

Figure 28 – Forwarding Conditions

According to the table that presents the conditions for forwarding control signal in section 4-3, we can get the pseudo-code presented in Figure 28. Figure 27 is the code implementation based on the following conditions. One of the considerations is that we must set the initial forwarding control signals to zero in the initial stage of the function operation, as presented in red rectangular. The reason why is that it should output its original RS or RT value as there are no data dependencies happen. By implementing the following code and executing them after a cycle, we can forward the values and resolve the data dependency on regular cases.

5-3. Hazard Detection Unit (HU)

```
/* initialize hazard detection unit signals */
void HU_Init() {
    PCWrite = true;
    IFIDWrite = true;
    IDEXCtrlSrc = false;
}


/* hazard detection unit operation*/
void HU_Operation() {
    if (IDEX[0].Control.MemRead && (IDEX[0].rd == inst->rs || IDEX[0].rd == inst->rt)) {
        PCWrite = false;
        IFIDWrite = false;
        IDEXCtrlSrc = true;
    }
    else {
        PCWrite = true;
        IFIDWrite = true;
        IDEXCtrlSrc = false;
    }
}
```

Figure 29 – Hazard Detection Unit Init and Operation

if (ID/EX.MemRead
  and ((ID/EX.RT == IF/ID.RS)
    or (ID/EX.RT == IF/ID.RT))
  PCWrite =0
  IF/IDWrite = 0
  ID/EXCtrlSrc = 1

Figure 30 – Hazard Detection Condition

To resolve the data dependency in the case of load-use dependency, we must implement a hazard detection unit to resolve it. Figure 30 is the pseudo-code for hazard conditions due to the situation presented in section 4-4. Figure 29 is the code implementation according to the conditions. One of the considerations in this function is that we should always set the control signals into the original form if the current processor state does not fit into the condition, which is presented in red rectangular. By implementing the following functions, we can resolve the load-use data dependency with a hazard detection unit.

5-4. Branch Predictor

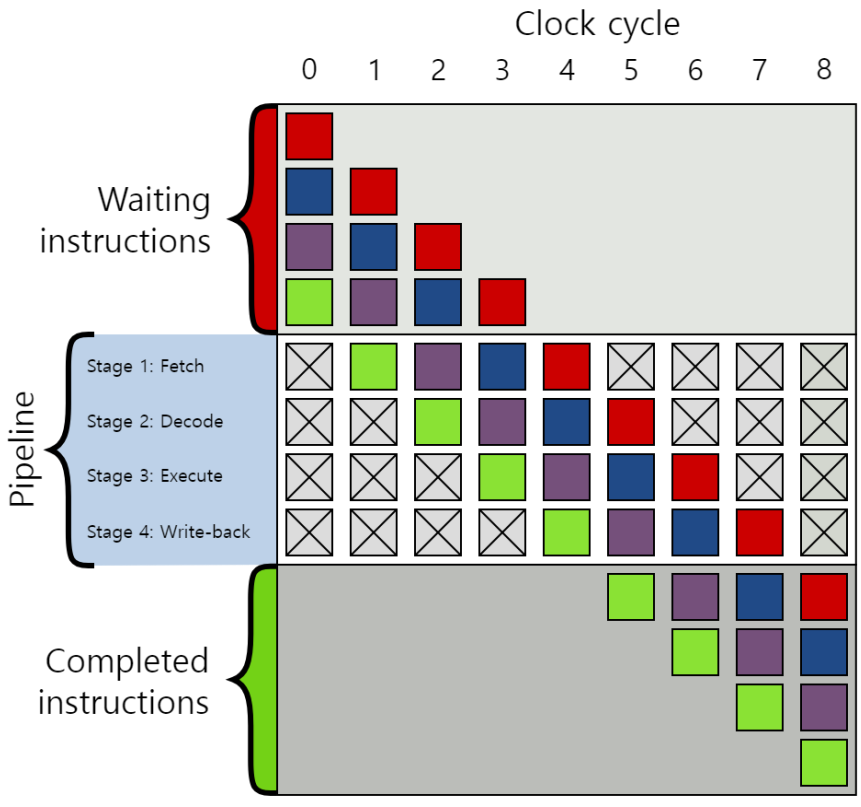


Figure 31 - Branch Prediction

In this section, we will mainly discuss the concepts of each branch predictors and their implementations. The three main points of the branch predictor are whether the instruction is branch instruction or not, whether the branch condition is taken or not, and the branch target address. We will introduce six branch predictors, which are always not taken (ANT), always taken (ALT), 1-bit last time predictor (LPT1), 2-bit counter-based predictor (LTP2), 2-level global predictor (GSH), and 2-level local predictor (LVL2). For each branch predictor, we will discuss how each predictor settles the following three main points.

5-4-1. Always Not Taken Branch Prediction (ANT)

```
/* always not taken branch predictior execution */
void ANT_Execution() {
    bool BrTaken = MUX2(EXMEM[0].bcond && EXMEM[0].Control.Branch, !EXMEM[0].bcond && EXMEM[0].Control.Branch, EXMEM[0].Control.Equal);
    if (IDEX[1].opcode == BEQ || IDEX[1].opcode == BNE) {
        branch_taken++;
        branch_hit = branch_taken - branch_miss;
    }

    if (BrTaken) {
        /* miss */
        branch_miss++;
        // printf("Branch Prediction Miss\n");
        if (PCWrite) PC = MUX2(PC, BranchAddr(IDEX[1].NPC, IDEX[1].imm), BrTaken);

        /* chang the valid bit of IFID and IDEX latches */
        IFID[0].valid = false;
        IFID[1].valid = false;
        IDEX[0].valid = false;
    }
    else {
        /* hit */
        // printf("Branch Prediction Hit\n");
    }
}
```

Figure 32 - Always Not Taken Branch Predictor Operation

Always not taken branch prediction is one of the methods based on the compile-time prediction (static). In this method, we assume that the branch prediction will be always not taken. Always not taking branch prediction is the simplest way to implement the branch predictor. It does not need the branch target buffer (BTB) and direction prediction. It shows low accuracy, about 30 ~ 40%. The compiler can layout the code such that the likely path is the not taken path (Yoo, 11-control_dependence). Figure 31 shows the implementation of the always not taken branch predictor execution function.

5-4-2. Always Taken Branch Prediction (ALT)

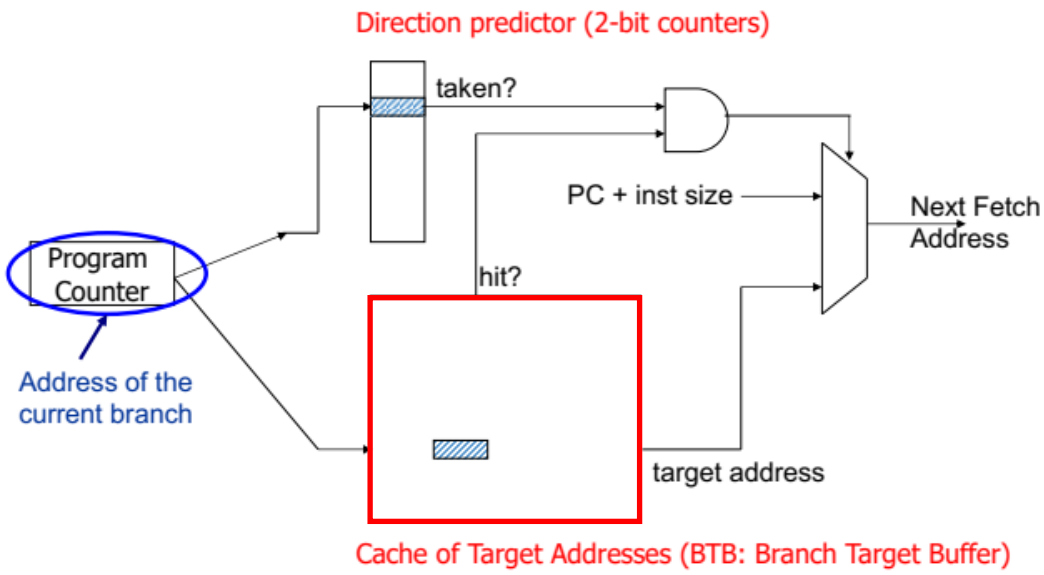


Figure 33 - Branch Target Buffer (BTB) (Yoo, *12-branch_prediction*)

```
/* initialize always taken branch predictor */
void ALT_Init() {
    memset(&BHT, -1, sizeof(BHT));
}


/* always taken predictor branch prediction */
void ALT_Prediction() {
    /* set index for branch history table by using the last two bits from the PC value */
    int bht_idx = BHTENTRY - 1;
    bht_idx <<= 2;
    bht_idx &= PC;
    bht_idx >>= 2;
    IFID[0].BHTIdx = bht_idx;

    if (PC == BHT[bht_idx].branch_inst_addr) {
        IFID[0].BHTFound = true;
        IFID[0].NPC = PCAddr(PC);
        if (PCWrite) PC = BHT[bht_idx].branch_target_addr;
    }
    else {
        IFID[0].BHTFound = false;
        if (PCWrite) PC = PCAddr(PC);
        IFID[0].NPC = PC;
    }
}
```

Figure 34 – Always Taken Branch Predictor Init and Prediction

```
/* always taken branch prediction execution */
void ALT_Execution() {
    if (IDEX[1].opcode == BEQ || IDEX[1].opcode == BNE) branch_taken++;
    bool BrTaken = MUX2(EXMEM[0].bcond && EXMEM[0].Control.Branch, !EXMEM[0].bcond && EXMEM[0].Control.Branch, EXMEM[0].Control.Equal);

    if (IDEX[1].Control.BHTFound) {
        if (BrTaken) {
            /* hit */
            branch_hit++;
            // printf("Branch Prediction Hit\n");
        }
        else {
            /* miss */
            branch_miss++;
            // printf("Branch Prediction Miss\n");
            if (PCWrite) PC = IDEX[1].NPC;

            /* chang the valid bit of IFID and IDEX latches */
            IFID[0].valid = false;
            IFID[1].valid = false;
            IDEX[0].valid = false;
        }
    }
    else {
        if (BrTaken) {
            /* miss */
            branch_hit++;
            // printf("Branch Prediction Miss\n");
            BHT[IDEX[1].BHTIdx].branch_inst_addr = IDEX[1].NPC - 4; // update instruction address
            if (PCWrite) PC = MUX2(PC, BranchAddr(IDEX[1].NPC, IDEX[1].imm), BrTaken);
            BHT[IDEX[1].BHTIdx].branch_target_addr = PC; // update target address

            /* chang the valid bit of IFID and IDEX latches */
            IFID[0].valid = false;
            IFID[1].valid = false;
            IDEX[0].valid = false;
        }
        else {
            /* hit */
        }
    }
}
```

Figure 35 – Always Taken Branch Predictor Operation

Always Taken branch prediction is also a method based on the compile-time prediction (static). In this method, it does not require direction prediction, but it requires a branch target buffer. Branch target buffer (BTB) can be used to reduce the performance penalty of branches by predicting the path of the branch and caching information about the branch (WWW2.EECS.BERKELEY.EDU). Up to four types of information can be cached in a BTB: a tag identifying the branch the information corresponds to, prediction information for predicting the path of the branch, the branch target address, and instruction bytes from the branch target address. A BTB with all four types of information works as follows. As each instruction is fetched from memory, the instruction address is used to index into the BTB. If a valid BTB entry is found for that address, the instruction is a branch. The branch path is predicted using the branch's prediction information. If the branch is predicted taken, the pipeline will be supplied with the cached instruction in BTB. If the branch is predicted not taken, the processor continues fetching sequentially after the branch. After the processor finishes executing the branch, it checks to see if the BTB correctly predicted the branch. If it has, all is well, and the processor continues sequentially. If the branch was predicted incorrectly, the processor must flush the pipeline, and the branch target address must be updated after the branch. Figure 32 shows the hardware implementation of the branch target buffer (BTB).

By using the branch target buffer (BTB) to the always taken branch predictor, we can implement the following prediction method. This method shows better accuracy than always not taken branch prediction, which is about 60 ~ 70%, and backward branches, which takes the target address lower than branch PC, are usually taken. Based on this concept, we can implement the always taken branch predictor presented in Figures 33 and 34. The backward branch is implemented as the red rectangular presented in Figure 34.

5-4-3. 1-bit Last Time Branch Prediction

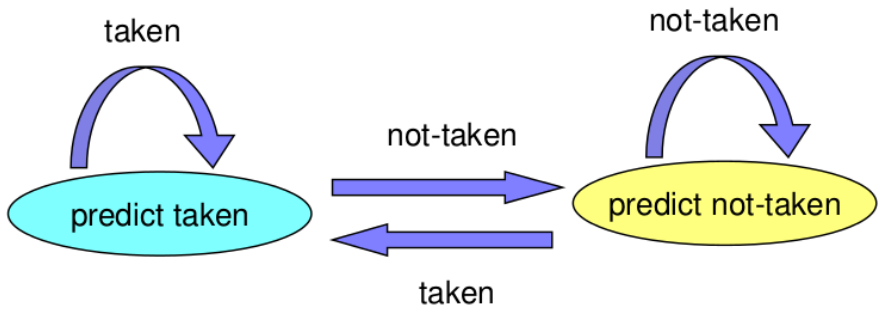


Figure 36 - 1-bit Last Time Prediction State Machine

```
loop: ...
        SLTI R2,R1,#10 ;is i<10?
        BNEZ R2,loop   ;branch if yes
```

Figure 37 - For Loop (*McGill University*)

Iteration #:	1	2	3	4	5	6	7	8	9	10
Predicted branch outcome:	N	T	T	T	T	T	T	T	T	T
Actual branch outcome:	T	T	T	T	T	T	T	T	T	N

Figure 38 - For Loop Prediction (*McGill University*)

```
/* check 1-bit for branch prediction
   0: not taken
   1: taken */
if (PC == BHT[bht_idx].branch_inst_addr) {
   IFID[0].BHTFound = true;
   switch (BHT[bht_idx].prediction_bit)
   {
   case 0:
      IFID[0].PreTaken = false;
      if (PCWrite) PC = PCAddr(PC);
      IFID[0].NPC = PC;
      break;

   case 1:
      IFID[0].PreTaken = true;
      IFID[0].NPC = PCAddr(PC);
      if (PCWrite) PC = BHT[bht_idx].branch_target_addr;
      break;
   }
}
else {
   IFID[0].BHTFound = false;
   IFID[0].PreTaken = false;
   if (PCWrite) PC = PCAddr(PC);
   IFID[0].NPC = PC;
}
```

Figure 39 - 2-bit Counter-Based Branch Predictor Prediction

```
if (BrTaken) {
   /* hit */
   branch_hit++;
   // printf("Branch Prediction Hit\n");
   BHT[IDEX[1].BHTIdx].prediction_bit = 1
}
else {
   /* miss */
   branch_miss++;
   // printf("Branch Prediction Miss\n");
   BHT[IDEX[1].BHTIdx].prediction_bit = 0;
   if (PCWrite) PC = IDEX[1].NPC;

   /* chang the valid bit of IFID and IDEX latches */
   IFID[0].valid = false;
   IFID[1].valid = false;
   IDEX[0].valid = false;
}
```

Figure 40 - 2-bit Counter-Based Branch Predictor Operation 1

```
if (BrTaken) {
   /* miss */
   branch_miss++;
   // printf("Branch Prediction Miss\n");
   BHT[IDEX[1].BHTIdx].prediction_bit = 1;
   if (PCWrite) PC = BHT[IDEX[1].BHTIdx].branch_target_addr;

   /* chang the valid bit of IFID and IDEX latches */
   IFID[0].valid = false;
   IFID[1].valid = false;
   IDEX[0].valid = false;
}
else {
   /* hit */
   branch_hit++;
   // printf("Branch Prediction Hit\n");
   BHT[IDEX[1].BHTIdx].prediction_bit = 0;
}
```

Figure 41 - 2-bit Counter-Based Branch Predictor Operation 2

```
else { // not found in BHT during IF stage
    if (BrTaken) { // BHT should be updated in this condition
        /* miss */
        branch_miss++;
        // printf("Branch Prediction Miss\n");
        BHT[IDEX[1].BHTIdx].branch_inst_addr = IDEX[1].NPC - 4; // update instruction address
        if (PCWrite) PC = MUX2(PC, BranchAddr(IDEX[1].NPC, IDEX[1].imm), BrTaken);
        BHT[IDEX[1].BHTIdx].branch_target_addr = PC; // update target address
        BHT[IDEX[1].BHTIdx].prediction_bit = 1; // update prediction bit into strongly taken

        /* chang the valid bit of IFID and IDEX latches */
        IFID[0].valid = false;
        IFID[1].valid = false;
        IDEX[0].valid = false;
    }
    else {
        /* hit */
    }
}
```

Figure 42 - 2-bit Counter-Based Branch Operation 3

We have checked the two of compile time branch prediction methods, which are always not taken, and always taken branch prediction. The common disadvantage of these techniques is that they cannot adapt to the dynamic changes in branch behavior. This can be migrated by a dynamic compiler, but not at a fine granularity.

To enhance the performance of the branch predictor, we can apply the run time prediction (dynamic). Dynamic branch prediction is a method that predicts branches based on dynamic information. In dynamic branch prediction, the processor predicts whether to jump to the branch target address or not, by having a prediction based on the history of the execution of branches. This history can be adapted to dynamic changes in branch behavior. Also, it does not require static profiling. However, the architecture of the processor becomes more complex due to the additional hardware requirements.

One of the techniques based on this run time prediction is a 1-bit last time branch predictor. The main idea of this technique is to use the history of a branch's past outcomes to predict its future outcomes. 1-bit last time branch predictor works in the following way: when we execute a branch, the processor first checks if it was taken when it was last executed. If yes, the processor predicts that it will be taken this time; if not, the processor predicts that it will not be taken this time. The pattern of this result is stated as a state machine presented in Figure 35. Check out the example in Figure 36. The loop presented in this figure is iterated 10 times and involves one branch. Then, the branch is taken in the first 9 iterations, and not taken in the last iteration. The result of the prediction by 1-bit last time branch predictor is shown in Figure 37. We can check that the ratio of the prediction is higher than the previous techniques.

1-bit last time branch predictor is implemented by a branch history table. This table records the information of every branch encountered in the program. Each table entry contains one prediction bit for the following branch, which is presented as zero if the prediction is not taken, and one if the prediction is taken. The prediction bit is used to predict the branch outcome. It is updated after the branch's actual outcome is known. Figures 38 to 41 show the implemented functions of the 1-bit last time branch predictor. Red rectangular on the figures show how we implemented the following method with the required operations that we have mentioned previously.

This technique is more efficient than the previous static branch predictions. There is a high ratio of the prediction for the for-loop iterations. However, it is still not the best way to predict the branch operations. One problem with this technique is that the last time predictor changes the prediction from taken to not taken too quickly, even though the branch may be mostly taken or mostly not taken. As a result, we need a better way to enhance the branch prediction by solving this problem.

5-4-4. 2-bit Counter-Based Branch Prediction

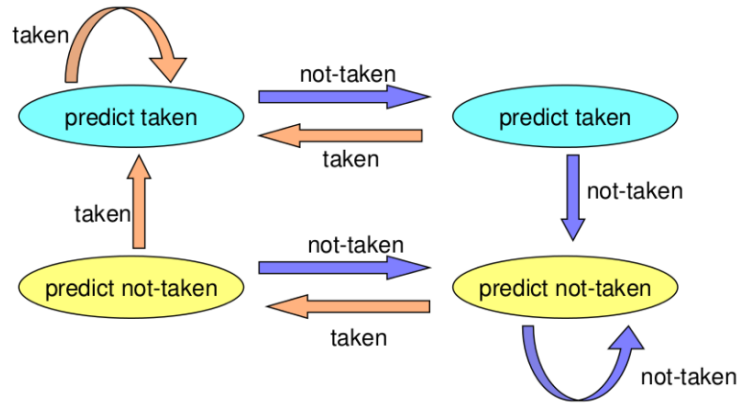


Figure 43 - 2-bit Counter-Based Prediction State Machine

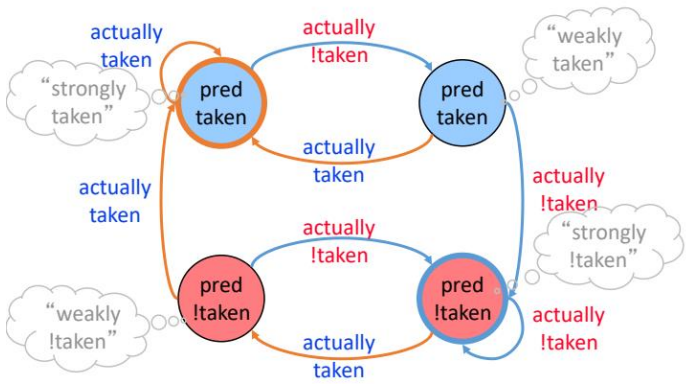


Figure 44 - Hysteresis 2-bit Counter-Based Prediction State Machine (Yoo, *12-branch_prediction*)

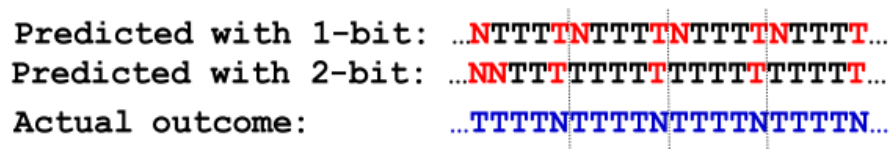


Figure 45 - Comparison between 1-bit and 2-bit Predictors (*McGill University*)

```
/* check 2-bit for branch prediction
   0b00: strongly not taken
   0b01: weakly not taken
   0b10: weakly taken
   0b11: strongly taken */
if (PC == BHT[bht_idx].branch_inst_addr) {
   IFID[0].BHTFound = true;
   switch (BHT[bht_idx].prediction_bit)
   {
   case 0b00:
   case 0b01:
      IFID[0].PreTaken = false;
      if (PCWrite) PC = PCAddr(PC);
      IFID[0].NPC = PC;
      break;

   case 0b10:
   case 0b11:
      IFID[0].PreTaken = true;
      IFID[0].NPC = PCAddr(PC);
      if (PCWrite) PC = BHT[bht_idx].branch_target_addr;
      break;
   }
}
else {
   IFID[0].BHTFound = false;
   IFID[0].PreTaken = false;
   if (PCWrite) PC = PCAddr(PC);
   IFID[0].NPC = PC;
}
```

```
if (BrTaken) {
   /* hit */
   branch_hit++;
   // printf("Branch Prediction Hit\n");
   if (BHT[IDEX[1].BHTIdx].prediction_bit == 0b10) // weakly taken -> strongly taken
      BHT[IDEX[1].BHTIdx].prediction_bit = 0b11;
}
else {
   /* miss */
   branch_miss++;
   // printf("Branch Prediction Miss\n");
   if (BHT[IDEX[1].BHTIdx].prediction_bit >= 0b10)
      BHT[IDEX[1].BHTIdx].prediction_bit -= 1;
   if (PCWrite) PC = IDEX[1].NPC;

   /* chang the valid bit of IFID and IDEX latches */
   IFID[0].valid = false;
   IFID[1].valid = false;
   IDEX[0].valid = false;
}
```

```
else { // predicted not to be taken
   if (BrTaken) {
      /* miss */
      branch_miss++;
      // printf("Branch Prediction Miss\n");
      if (BHT[IDEX[1].BHTIdx].prediction_bit <= 0b01) // strongly not taken -> weakly not taken or weakly not taken
         BHT[IDEX[1].BHTIdx].prediction_bit += 1;
      if (PCWrite) PC = BHT[IDEX[1].BHTIdx].branch_target_addr;

      /* chang the valid bit of IFID and IDEX latches */
      IFID[0].valid = false;
      IFID[1].valid = false;
      IDEX[0].valid = false;
   }
   else {
      /* hit */
      branch_hit++;
      // printf("Branch Prediction Hit\n");
      if (BHT[IDEX[1].BHTIdx].prediction_bit == 0b01) // weakly not taken -> strongly not taken
         BHT[IDEX[1].BHTIdx].prediction_bit = 0b00;
   }
}
```

Figure 46 - 2-bit Counter-Based Branch Predictor Prediction and Operation 1

```
else { // not found in BHT during IF stage
    if (BrTaken) { // BHT should be updated in this condition
        /* miss */
        branch_miss++;
        // printf("Branch Prediction Miss\n");
        BHT[IDEX[1].BHTIdx].branch_inst_addr = IDEX[1].NPC - 4; // update instruction address
        if (PCWrite) PC = MUX2(PC, BranchAddr(IDEX[1].NPC, IDEX[1].imm), BrTaken);
        BHT[IDEX[1].BHTIdx].branch_target_addr = PC; // update target address
        BHT[IDEX[1].BHTIdx].prediction_bit = 0b11; // update prediction bit into strongly taken

        /* chang the valid bit of IFID and IDEX latches */
        IFID[0].valid = false;
        IFID[1].valid = false;
        IDEX[0].valid = false;
    }
    else {
        /* hit */
    }
}
```

Figure 47 - 2-bit Counter-Based Branch Predictor Operation 2

The solution to the previous technique is to add hysteresis to the predictor so that the prediction does not change on a single different outcome. By using 2 bits to track the history of predictions for a branch instead of a single bit, and adding 2 states for taken or not taken, we can improve the 1-bit last time branch predictor.

An improved technique that includes the concepts which are mentioned above is a 2-bit counter-based branch predictor. In this method, each branch is associated with a 2-bit counter. The additional bit in the pattern provides hysteresis. A strong prediction does not change with a single different branch outcome. Figure 42 shows the state of the branching pattern, and Figure 43 shows another way of applying hysteresis. As a result, by applying the concepts of a 2-bit counter-based branch predictor, we can implement the functions shown in Figures 45 to 46. The features of the predictor are implemented as red rectangular on the figures.

Then, let's check the performance of the following technique with the previous method. From Figure 44, we can check that due to the slow convergence of the pattern bit, the hit ratio of the branch prediction increases. Therefore, we can check that the 2-bit counter-based branch predictor shows better performance than the 1-bit last time branch predictor. In common cases, this technique shows 85 ~ 90% of accuracy.

It seems like the following technique is a good method for branch prediction, but it is still needed to be enhanced. Think about the following cases: a branch's outcome can be correlated with other branches' outcomes. Also, a branch's outcome can be correlated with past outcomes of the same branch. If we consider the following cases, we can increase the performance of the branch prediction for the processor.

5-4-5. 2-Level Global Branch Prediction

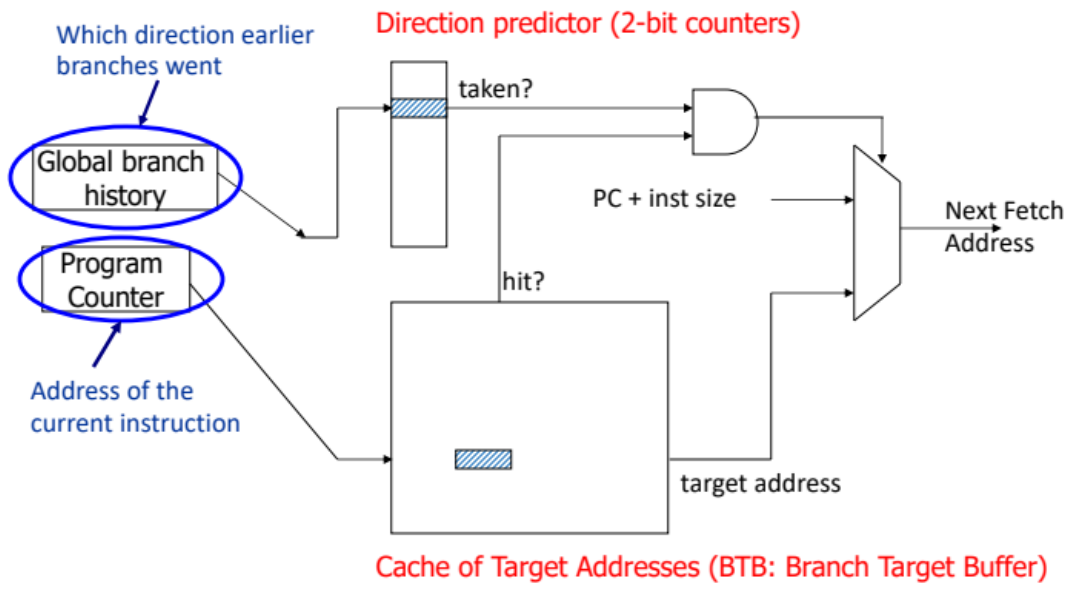


Figure 48 - 2-Level Global History Predictor (Yoo, *12-branch_prediction*)

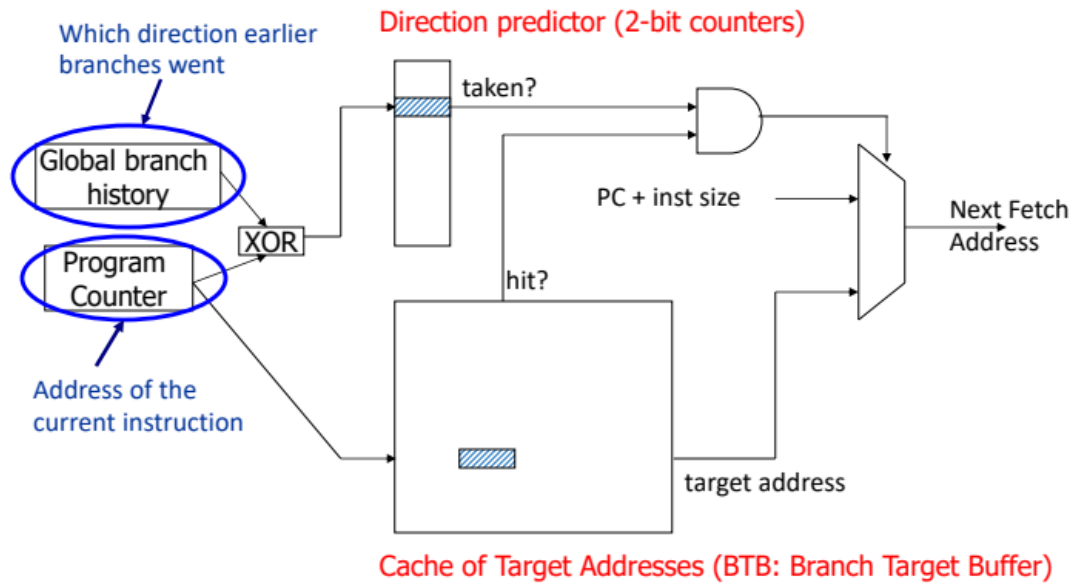


Figure 49 - 2-Level Global Share Predictor (Yoo, *12-branch_prediction*)

```
void GSH_Prediction() {
    int bht_idx = BHTENTRY - 1;
    bht_idx <<= 2;
    bht_idx &= PC;
    bht_idx >>= 2;
    IFID[0].BHTIdx = bht_idx;

    int pht_idx = PHTENTRY - 1;
    pht_idx <<= 2;
    pht_idx &= (GHR ^ PC);
    pht_idx >>= 2;
    IFID[0].PHTIdx = pht_idx;

    /* check 2-bit for branch prediction
       0b00: strongly not taken
       0b01: weakly not taken
       0b10: weakly taken
       0b11: strongly taken */
    if (PC == BHT[bht_idx].branch_inst_addr) {
        IFID[0].BHTFound = true;
        switch (PHT[pht_idx])
        {
        case 0b00:
        case 0b01:
            IFID[0].PreTaken = false;
            if (PCWrite) PC = PCAddr(PC);
            IFID[0].NPC = PC;
            break;

        case 0b10:
        case 0b11:
            IFID[0].PreTaken = true;
            IFID[0].NPC = PCAddr(PC);
            if (PCWrite) PC = BHT[bht_idx].branch_target_addr;
            break;
        }
    }
    else {
        IFID[0].BHTFound = false;
        IFID[0].PreTaken = false;
        if (PCWrite) PC = PCAddr(PC);
        IFID[0].NPC = PC;
    }
}
```

```
else { // not found in BHT during IF stage
    if (BrTaken) { // BHT should be updated in this condition
        /* miss */
        branch_miss++;
        // printf("Branch Prediction Miss\n");
        BHT[IDEX[1].BHTIdx].branch_inst_addr = IDEX[1].NPC - 4; // update instruction address
        if (PCWrite) PC = MUX2(PC, BranchAddr(IDEX[1].NPC, IDEX[1].imm), BrTaken);
        BHT[IDEX[1].BHTIdx].branch_target_addr = PC; // update target address
        PHT[IDEX[1].PHTIdx] = 0b11; // update prediction bit into strongly taken

        /* chang the valid bit of IFID and IDEX latches */
        IFID[0].valid = false;
        IFID[1].valid = false;
        IDEX[0].valid = false;
    }
    else {
        /* hit */
    }
}

/* global histroy register update */
if (IDEX[1].opcode == BEQ || IDEX[1].opcode == BNE) {
    branch_taken++;
    GHR <<= 1;
    GHR |= BrTaken;
}
```

Figure 50 - 2-Level Global Share Predictor Prediction and Operation

Let's check the first consideration to enhance the 2-bit counter-based branch predictor, which is that a branch's outcome can be correlated with other branches' outcomes. This means that the recently executed branch outcome in the executing path is correlated with the outcome of the next branch.

2-level global branch predictor contains this concept. The idea of this technique is to associate branch outcomes with global taken and not taken history of all branches. This method makes a prediction based on the outcome of the branch the last time the same global branch history was encountered. To implement the following method, we must add the following processes: Keep track of the global taken and not taken history of all branches in a register, global history register (GHR), and use GHR to index into a table that recorded the outcome that was seen for that GHR value in the recent past, which is called pattern history table that contains the 2-bit counters. Figure 47 shows the implementation of the following method.

One problem with this technique is that the index can be polluted due to accessing the same index of the pattern history table. To avoid this situation, we applied the global share technique. By adding more context information to the global predictor to consider which branch is being predicted, we can resolve the following situation. In global share prediction, we use the result of the XOR operation between GHR and PC as an index to the pattern history table. This occurs with more context information and better utilization of the pattern history table. However, the increment of access latency. Figure 48 shows the implementation of global share branch prediction.

Figure 49 shows the implementations of the 2-level global share branch predictor, which includes the concepts that are presented above. Each red rectangular presented in the figure shows the additional implementation for the following method. By using these functions, we can enhance our pipelined MIPS simulator with better branch prediction.

5-4-6. 2-Level Local Branch Prediction

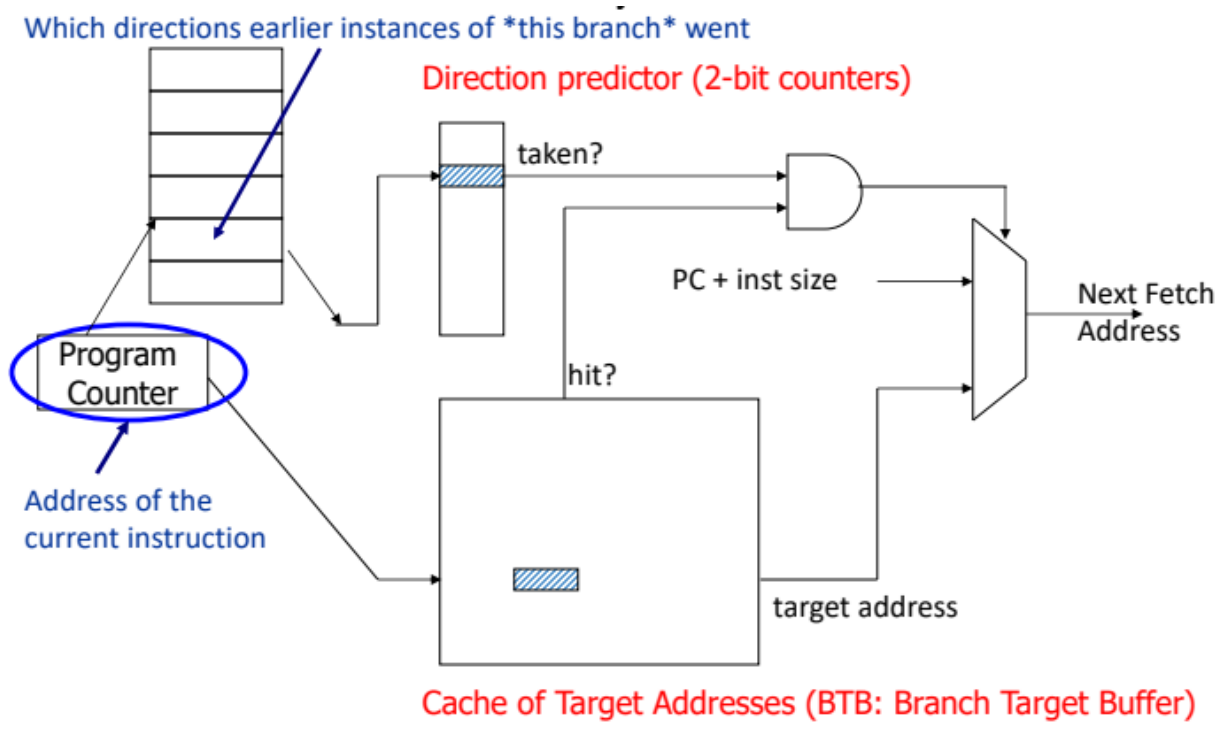


Figure 51 - 2-Level Local Branch Predictor (Yoo, *12-branch_prediction*)

```
int bht_idx = BHTENTRY - 1;
bht_idx <<= 2;
bht_idx &= PC;
bht_idx >>= 2;
IFID[0].BHTIdx = bht_idx;

int lht_idx = LHTENTRY - 1;
lht_idx <<= 2;
lht_idx &= PC;
lht_idx >>= 2;
IFID[0].LHTIdx = lht_idx;

int pht_idx = PHTENTRY - 1;
pht_idx <<= 2;
pht_idx &= LHT[lht_idx];
pht_idx >>= 2;
IFID[0].PHTIdx = pht_idx;
```

```
/* local histroy register update */
if (IDEX[1].opcode == BEQ || IDEX[1].opcode == BNE) {
    branch_taken++;
    LHT[IDEX[1].LHTIdx] <<= 1;
    LHT[IDEX[1].LHTIdx] |= BrTaken;
}
```

Figure 52 - 2-Level Local Branch Predictor Prediction and Operation

Another consideration to enhance the 2-bit counter-based predictor is a branch's outcome can be correlated with past outcomes of the same branch.

The 2-level local branch predictor contains the following concepts: The idea of this method is to have a per-branch history register, which associates the predicted outcome of a branch with the taken and not taken history of the same branch. This technique makes a prediction based on the outcome of the branch the last time the same local branch history was encountered. To implement this method, we must use two levels of history, which are the pre-branch history register and history at that history register value. The first level contains a set of local history registers, which select the history register based on the PC of the branch. The second level contains a table of saturating counters for each history entry. At this level, the direction that the branch took the last time shows the same history. Figure 50 shows the implementation of the following method.

Figure 51 shows the additional implementation of codes from the previous technique, which is a 2-level global branch predictor. One difference between these methods is that we added a table and indexed it with the local history registers. By using the functions in Figure 51, we can enhance the results of the pipelined MIPS simulator with the following branch predictor.

5-5. CPU

```
/* main */
void main(int argc, char* argv[]) {
    PROGRAM = argv[1];
    PREDICTOR = argv[2][0];

    /* initialization */
    init();

    /* cycle start */
    do {
        LINE;
        cycle++;

        /* execute the stages */
        MEM();
        WB(); // write back first so that write operation happens before read operation
        IF();
        ID();
        EX();

        /* fix the data hazards and the delay slot due to memory read operaiton*/
        HU_Operation();
        FW_Operation();

        /* print out the logs */
        LOG_Print();

        /* update the latches */
        LAT_Update();
        LINE;
    } while (PC < 0xffffffff);

    /* termination */
    terminate();
}
```

Figure 53 – Main Function

Figure 53 shows the source code of the main function of the pipelined MIPS simulator. First, it executes the 5 stages, which are instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write back (WB). Second, it resolves the data dependencies by using the forward unit and hazard detection unit. After forwarding the data, we will print out the logs and update the latches.

There are two considerations in this source code. First, the sequence of stage execution is slightly different from the ordinary single-cycle execution. Memory access stage and write back stage are initially executed. The reason why we implemented the cycle in this way is that to resolve the data dependencies, the register write operation must happen earlier than the register read operation. The second is data forwarding and data hazard detection right after the execution stage. The reason why we have done this is that the values of data-dependent registers should be determined before the values are written in the register file. By implementing the main function that includes the following considerations, we can build the proper pipelined MIPS simulator.

5-5-1. Instruction Fetch (IF)

```
/* IF (instruction fetch) stage*/
void IF() {
    if (PC != 0xffffffff) {
        total_instruction++;
        IFID[0].valid = true; // change the valid bit into 1
        IFID[0].IR = IM_ReadMemory(PC); // fetch the instruction
        switch (PREDICTOR)
        {
        case '1': { PC = PCAddr(PC); IFID[0].NPC = PC;  break; }
        case '2': ALT_Prediction();     break;
        case '3': LTP1_Prediction();    break;
        case '4': LTP2_Prediction();    break;
        case '5': GSH_Prediction();     break;
        case '6': LVL2_Prediction();    break;
        default: {
            printf("Error: wrong branch predictor selection.\n");
            exit(0);
        }
        }
    }
}
```

Figure 54 - Instruction Fetch Function

Figure 54 shows the source code of instruction fetch (IF) function. First, it changes the valid bit into true, so that the IF/ID latch can start operation. Then, it reads the instruction from the instruction memory, and make a prediction according to the following predictor chosen by PREDICTOR variable.

5-5-2. Instruction Decode (ID)

```
/* ID (instruction decode) stage */
void ID() {
    if (IFID[1].valid) {
        /* instruction decode */
        inst->opcode = (IFID[1].IR >> 26) & 0x3f;
        inst->rs = (IFID[1].IR >> 21) & 0x1f;
        inst->rt = (IFID[1].IR >> 16) & 0x1f;
        inst->rd = (IFID[1].IR >> 11) & 0x1f;
        inst->shamt = (IFID[1].IR >> 6) & 0x1f;
        inst->funct = (IFID[1].IR >> 0) & 0x3f;
        inst->immediate = (IFID[1].IR >> 0) & 0xffff;
        inst->address = (IFID[1].IR >> 0) & 0x3ffffff;

        /* execute stage */
        RF_Read(inst->rs, inst->rt);
        CU_Operation(inst->opcode, inst->funct);

        IDEX[0].valid = true; // change the valid bit into 1
        if (!IDEXCtrlSrc) IDEX_EXMEM_CPY();
        else memset(&IDEX[0].Control, 0, sizeof(IDEX_SIG)); // if data hazard is detected, flush the control signals
        IDEX_LAT_WIRE();

        if (PCWrite && Jump) jump_instruction++;
        if (PCWrite) PC = MUX2(PC, JumpAddr(IFID[1].NPC, inst->address), Jump);
    }
}
```

Figure 55 - Instruction Decode Function

Figure 55 shows the source code of instruction decode (ID) function. It operates when the valid bit from previous stage is true. First, it decodes the given instruction according to the MIPS ISA. Second it reads the values from the registers and generates control signals. Third, it changes the valid bit into true, so that the ID/EX latch can start operation. If data dependency is detected, it flushes the control signals into zero. Else, it copies the control signals and wires the values into the next latch. At the end, the jump operation is executed.

5-5-3. Execution (EX)

```
/* EX (execution) stage */
void EX() {
    if (IDEX[1].valid) {
        /* execute stage */
        uint32_t ALUControl = ALU_Control(IDEX[1].Control.ALUOp, MUX2(IDEX[1].opcode, IDEX[1].funct, IDEX[1].Control.Rtype));
        EXMEM[0].ReadData1 = MUX3(IDEX[1].ReadData1, MUX2(MEMWB[1].ALUResult, MEMWB[1].ReadData, MEMWB[1].Control.MemToReg), EXMEM[1].ALUResult, FwdA);
        EXMEM[0].ReadData2 = MUX3(IDEX[1].ReadData2, MUX2(MEMWB[1].ALUResult, MEMWB[1].ReadData, MEMWB[1].Control.MemToReg), EXMEM[1].ALUResult, FwdB);
        EXMEM[0].ALUResult = ALU_Operation(ALUControl, MUX2(EXMEM[0].ReadData1, EXMEM[0].ReadData2, IDEX[1].Control.Shift),
                                                       MUX2(MUX2(EXMEM[0].ReadData2, IDEX[1].imm, IDEX[1].Control.ALUSrc), IDEX[1].shamt, IDEX[1].Control.Shift));

        EXMEM[0].valid = true; // change the valid bit into 1
        EXMEM_MEMWB_CPY();
        EXMEM_LAT_WIRE();

        switch (PREDICTOR)
        {
        case '1': ANT_Execution();    break;
        case '2':  ALT_Execution();    break;
        case '3': LTP1_Execution();    break;
        case '4': LTP2_Execution();    break;
        case '5': GSH_Execution();    break;
        case '6': LVL2_Execution();    break;
        default: {
            printf("Error: wrong branch predictor selection.\n");
            exit(0);
        }
        }
        if (PCWrite && IDEX[1].Control.JumpReg) jump_instruction++;
        if (PCWrite) PC = MUX2(PC, EXMEM[0].ReadData1, IDEX[1].Control.JumpReg);
    }
}
```

Figure 56 - Execution Function

Figure 56 shows the source code of execution (EX) function. First, it checks that the valid bit is true and if it is, it operates. Next, it reads the register values considering forwarded data. These values are determined by the multiplexer that takes 3 input data which is controlled by forward control signals. Third, it changes the valid bit into true, so that EX/MEM latch can start operation. Then, it copies the control signals and wires values to the next latch from the previous stage. Fourth, it executes the predictor and fix the pattern history table, branch history table, global history register, or local history table due to the selected predictor which is determined by PREDICTOR variable. At the end, it executes the jump register operation.

5-5-4. Memory Access (MEM)

```
/* MEM (memory access) stage */
void MEM() {
    if (EXMEM[1].valid) {
        DM_MemoryAccess(EXMEM[1].ALUResult, 32, EXMEM[1].ReadData2, EXMEM[1].Control.MemRead, EXMEM[1].Control.MemWrite);
        MEMWB[0].valid = true; // change the valid bit into 1
        MEMWB_LAT_WIRE();
    }
}
```

Figure 57 - Memory Access Function

Figure 57 shows the source code of memory access (MEM) function. It operates when the valid bit from previous stage is true. If the memory access control signals are true, it accesses to the data memory and do load or store operation. After that, it changes the valid bit to allow MEM/WB latch to operate and wires the values to the next latch.

5-5-5. Write Back (WB)

```
/* WB (write back) stage */
void WB() {
    if (MEMWB[1].valid)
        RF_Write(MEMWB[1].rd, MUX2(MUX2(MEMWB[1].ALUResult, MEMWB[1].ReadData, MEMWB[1].Control.MemToReg), MEMWB[1].NPC, MEMWB[1].Control.JumpLink), MEMWB[1].Control.RegWrite);
}
```

Figure 58 - Write Back Function

Figure 58 shows the source code of write back (WB) function. It operates when the valid bit from previous stage is true. It chooses the write back value from the multiplexer and store it into the register file if the register write control signal is true.

6. Build Environment

- Build Environments:
  1. Linux environment – Vi editor, GCC complier
  2. Program is built by using the Makefile.

- Make Command:
  1. $make cpu -> build the execution program
  2. $make clean -> clean all the object files that builds main

- Execution Command
  1. ./cpu test_prog/{$program} 1 -> Execute the program with Always Not Taken branch predictor.
  2. ./cpu test_prog/{$program} 2 -> Execute the program with Always Taken branch predictor.
  3. ./cpu test_prog/{$program} 3 -> Execute the program with 1-bit Last Time branch predictor.
  4. ./cpu test_prog/{$program} 4 -> Execute the program with 2-bit Counter-Based branch predictor.
  5. ./cpu test_prog/{$program} 5 -> Execute the program with 2-Level Global branch predictor.
  6. ./cpu test_prog/{$program} 6 -> Execute the program with 2-Level Local branch predictor.

7. Results

Results are presented in execution command order presented in section 6, from left to right and top to bottom.

7-1. simple.bin

```
*****************result*********************
Final return value: 0
Number of executed instruction: 10
Number of executed memory access instruction: 2
Number of executed register operation instruction: 4
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
*********************************************
```

```
*****************result*********************
Final return value: 0
Number of executed instruction: 10
Number of executed memory access instruction: 2
Number of executed register operation instruction: 4
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
*********************************************
```

```
*****************result*********************
Final return value: 0
Number of executed instruction: 10
Number of executed memory access instruction: 2
Number of executed register operation instruction: 4
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
*********************************************
```

```
*****************result*********************
Final return value: 0
Number of executed instruction: 10
Number of executed memory access instruction: 2
Number of executed register operation instruction: 4
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
*********************************************
```

```
*****************result*********************
Final return value: 0
Number of executed instruction: 10
Number of executed memory access instruction: 2
Number of executed register operation instruction: 4
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
*********************************************
```

```
*****************result*********************
Final return value: 0
Number of executed instruction: 10
Number of executed memory access instruction: 2
Number of executed register operation instruction: 4
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
*********************************************
```

7-2. simple2.bin

```
*****************result*********************
Final return value: 100
Number of executed instruction: 12
Number of executed memory access instruction: 4
Number of executed register operation instruction: 6
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
********************************************
```

```
*****************result*********************
Final return value: 100
Number of executed instruction: 12
Number of executed memory access instruction: 4
Number of executed register operation instruction: 6
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
********************************************
```

```
*****************result*********************
Final return value: 100
Number of executed instruction: 12
Number of executed memory access instruction: 4
Number of executed register operation instruction: 6
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
********************************************
```

```
*****************result*********************
Final return value: 100
Number of executed instruction: 12
Number of executed memory access instruction: 4
Number of executed register operation instruction: 6
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
********************************************
```

```
*****************result*********************
Final return value: 100
Number of executed instruction: 12
Number of executed memory access instruction: 4
Number of executed register operation instruction: 6
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
********************************************
```

```
*****************result*********************
Final return value: 100
Number of executed instruction: 12
Number of executed memory access instruction: 4
Number of executed register operation instruction: 6
Number of executed taken branch instruction: 0
Number of predicted branch: 0
Number of not predicted branch: 0
Number of jump instruction: 1
********************************************
```

7-3. simple3.bin

```
*****************result*********************
Final return value: 5050
Number of executed instruction: 1535
Number of executed memory access instruction: 613
Number of executed register operation instruction: 714
Number of executed taken branch instruction: 102
Number of predicted branch: 1
Number of not predicted branch: 101
Number of jump instruction: 2
********************************************
```

```
*****************result*********************
Final return value: 5050
Number of executed instruction: 1337
Number of executed memory access instruction: 613
Number of executed register operation instruction: 714
Number of executed taken branch instruction: 102
Number of predicted branch: 101
Number of not predicted branch: 1
Number of jump instruction: 2
********************************************
```

```
*****************result*********************
Final return value: 5050
Number of executed instruction: 1337
Number of executed memory access instruction: 613
Number of executed register operation instruction: 714
Number of executed taken branch instruction: 102
Number of predicted branch: 100
Number of not predicted branch: 2
Number of jump instruction: 2
********************************************
```

```
*****************result*********************
Final return value: 5050
Number of executed instruction: 1337
Number of executed memory access instruction: 613
Number of executed register operation instruction: 714
Number of executed taken branch instruction: 102
Number of predicted branch: 100
Number of not predicted branch: 2
Number of jump instruction: 2
********************************************
```

```
*****************result*********************
Final return value: 5050
Number of executed instruction: 1349
Number of executed memory access instruction: 613
Number of executed register operation instruction: 714
Number of executed taken branch instruction: 102
Number of predicted branch: 94
Number of not predicted branch: 8
Number of jump instruction: 2
********************************************
```

```
*****************result*********************
Final return value: 5050
Number of executed instruction: 1349
Number of executed memory access instruction: 613
Number of executed register operation instruction: 714
Number of executed taken branch instruction: 102
Number of predicted branch: 94
Number of not predicted branch: 8
Number of jump instruction: 2
********************************************
```

7-4. simple4.bin

```
**********result**********
Final return value: 55
Number of executed instruction: 304
Number of executed memory access instruction: 100
Number of executed register operation instruction: 160
Number of executed taken branch instruction: 10
Number of predicted branch: 1
Number of not predicted branch: 9
Number of jump instruction: 22
**********
```

```
**********result**********
Final return value: 55
Number of executed instruction: 290
Number of executed memory access instruction: 100
Number of executed register operation instruction: 160
Number of executed taken branch instruction: 10
Number of predicted branch: 9
Number of not predicted branch: 1
Number of jump instruction: 22
**********
```

```
**********result**********
Final return value: 55
Number of executed instruction: 290
Number of executed memory access instruction: 100
Number of executed register operation instruction: 160
Number of executed taken branch instruction: 10
Number of predicted branch: 8
Number of not predicted branch: 2
Number of jump instruction: 22
**********
```

```
**********result**********
Final return value: 55
Number of executed instruction: 290
Number of executed memory access instruction: 100
Number of executed register operation instruction: 160
Number of executed taken branch instruction: 10
Number of predicted branch: 8
Number of not predicted branch: 2
Number of jump instruction: 22
**********
```

```
**********result**********
Final return value: 55
Number of executed instruction: 302
Number of executed memory access instruction: 100
Number of executed register operation instruction: 160
Number of executed taken branch instruction: 10
Number of predicted branch: 2
Number of not predicted branch: 8
Number of jump instruction: 22
**********
```

```
**********result**********
Final return value: 55
Number of executed instruction: 302
Number of executed memory access instruction: 100
Number of executed register operation instruction: 160
Number of executed taken branch instruction: 10
Number of predicted branch: 2
Number of not predicted branch: 8
Number of jump instruction: 22
**********
```

7-5. gcd.bin

```
**********result**********
Final return value: 1
Number of executed instruction: 1329
Number of executed memory access instruction: 486
Number of executed register operation instruction: 631
Number of executed taken branch instruction: 73
Number of predicted branch: 28
Number of not predicted branch: 45
Number of jump instruction: 103
**********
```

```
**********result**********
Final return value: 1
Number of executed instruction: 1299
Number of executed memory access instruction: 486
Number of executed register operation instruction: 631
Number of executed taken branch instruction: 73
Number of predicted branch: 45
Number of not predicted branch: 28
Number of jump instruction: 103
**********
```

```
**********result**********
Final return value: 1
Number of executed instruction: 1269
Number of executed memory access instruction: 486
Number of executed register operation instruction: 631
Number of executed taken branch instruction: 73
Number of predicted branch: 58
Number of not predicted branch: 15
Number of jump instruction: 103
**********
```

```
**********result**********
Final return value: 1
Number of executed instruction: 1269
Number of executed memory access instruction: 486
Number of executed register operation instruction: 631
Number of executed taken branch instruction: 73
Number of predicted branch: 58
Number of not predicted branch: 15
Number of jump instruction: 103
**********
```

```
**********result**********
Final return value: 1
Number of executed instruction: 1283
Number of executed memory access instruction: 486
Number of executed register operation instruction: 631
Number of executed taken branch instruction: 73
Number of predicted branch: 51
Number of not predicted branch: 22
Number of jump instruction: 103
**********
```

```
**********result**********
Final return value: 1
Number of executed instruction: 1285
Number of executed memory access instruction: 486
Number of executed register operation instruction: 631
Number of executed taken branch instruction: 73
Number of predicted branch: 50
Number of not predicted branch: 23
Number of jump instruction: 103
**********
```

7-6. fib.bin

```
******************result*********************
Final return value: 55
Number of executed instruction: 3280
Number of executed memory access instruction: 1095
Number of executed register operation instruction: 1638
Number of executed taken branch instruction: 109
Number of predicted branch: 55
Number of not predicted branch: 54
Number of jump instruction: 274
*********************************************
```

```
******************result*********************
Final return value: 55
Number of executed instruction: 3284
Number of executed memory access instruction: 1095
Number of executed register operation instruction: 1638
Number of executed taken branch instruction: 109
Number of predicted branch: 54
Number of not predicted branch: 55
Number of jump instruction: 274
*********************************************
```

```
******************result*********************
Final return value: 55
Number of executed instruction: 3256
Number of executed memory access instruction: 1095
Number of executed register operation instruction: 1638
Number of executed taken branch instruction: 109
Number of predicted branch: 67
Number of not predicted branch: 42
Number of jump instruction: 274
*********************************************
```

```
******************result*********************
Final return value: 55
Number of executed instruction: 3280
Number of executed memory access instruction: 1095
Number of executed register operation instruction: 1638
Number of executed taken branch instruction: 109
Number of predicted branch: 55
Number of not predicted branch: 54
Number of jump instruction: 274
*********************************************
```

```
******************result*********************
Final return value: 55
Number of executed instruction: 3232
Number of executed memory access instruction: 1095
Number of executed register operation instruction: 1638
Number of executed taken branch instruction: 109
Number of predicted branch: 79
Number of not predicted branch: 30
Number of jump instruction: 274
*********************************************
```

```
******************result*********************
Final return value: 55
Number of executed instruction: 3232
Number of executed memory access instruction: 1095
Number of executed register operation instruction: 1638
Number of executed taken branch instruction: 109
Number of predicted branch: 79
Number of not predicted branch: 30
Number of jump instruction: 274
*********************************************
```

7-7. input4.bin

```
******************result*********************
Final return value: 85
Number of executed instruction: 27430471
Number of executed memory access instruction: 7116606
Number of executed register operation instruction: 15240278
Number of executed taken branch instruction: 2029699
Number of predicted branch: 869
Number of not predicted branch: 2028830
Number of jump instruction: 104
*********************************************
```

```
******************result*********************
Final return value: 85
Number of executed instruction: 23374553
Number of executed memory access instruction: 7116606
Number of executed register operation instruction: 15240278
Number of executed taken branch instruction: 2029699
Number of predicted branch: 2028830
Number of not predicted branch: 868
Number of jump instruction: 104
*********************************************
```

```
******************result*********************
Final return value: 85
Number of executed instruction: 23376109
Number of executed memory access instruction: 7116606
Number of executed register operation instruction: 15240278
Number of executed taken branch instruction: 2029699
Number of predicted branch: 2028049
Number of not predicted branch: 1649
Number of jump instruction: 104
*********************************************
```

```
******************result*********************
Final return value: 85
Number of executed instruction: 23374685
Number of executed memory access instruction: 7116606
Number of executed register operation instruction: 15240278
Number of executed taken branch instruction: 2029699
Number of predicted branch: 2028761
Number of not predicted branch: 937
Number of jump instruction: 104
*********************************************
```

```
******************result*********************
Final return value: 85
Number of executed instruction: 23374667
Number of executed memory access instruction: 7116606
Number of executed register operation instruction: 15240278
Number of executed taken branch instruction: 2029699
Number of predicted branch: 2028770
Number of not predicted branch: 928
Number of jump instruction: 104
*********************************************
```

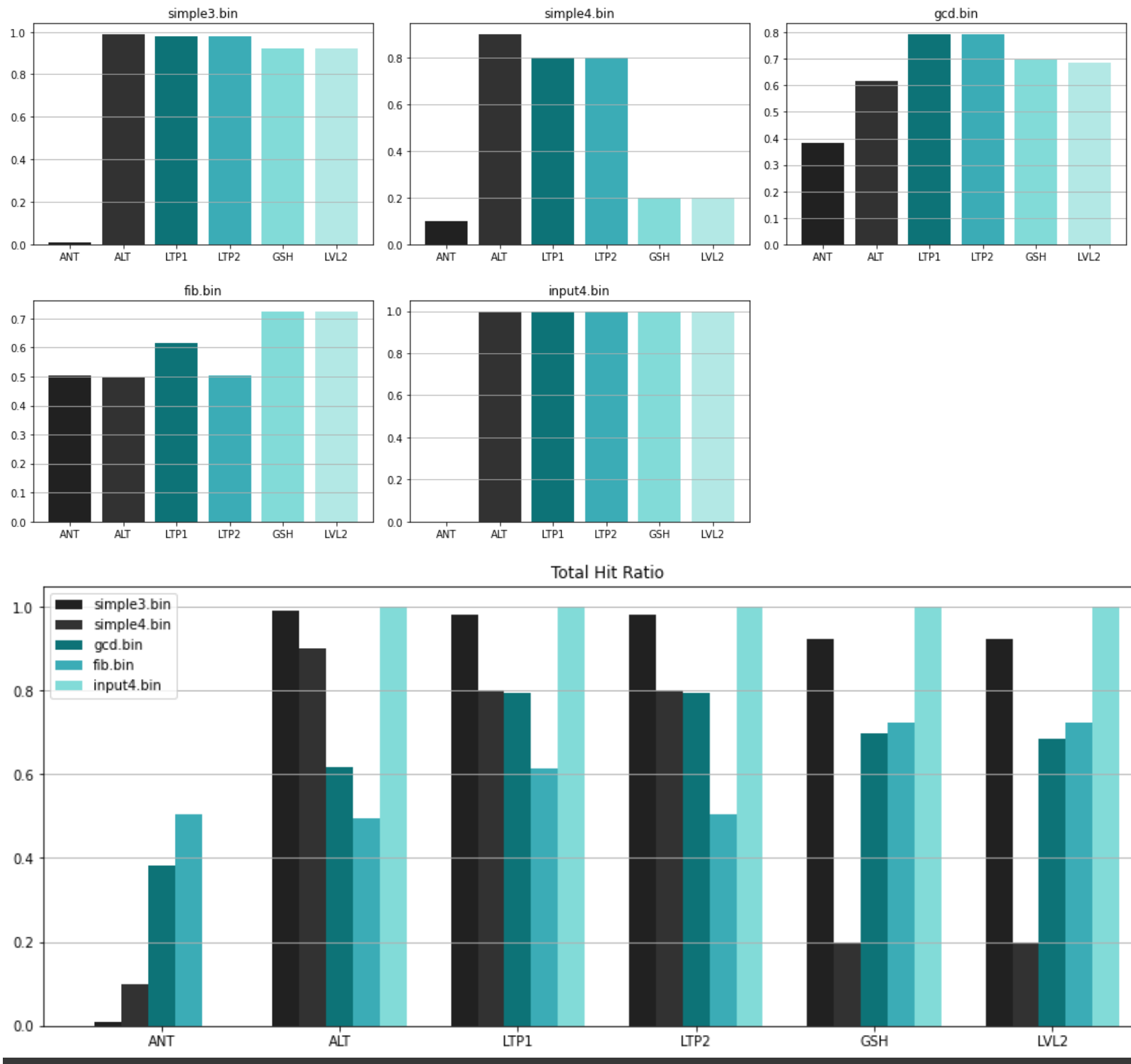
```
******************result*********************
Final return value: 85
Number of executed instruction: 23374635
Number of executed memory access instruction: 7116606
Number of executed register operation instruction: 15240278
Number of executed taken branch instruction: 2029699
Number of predicted branch: 2028786
Number of not predicted branch: 912
Number of jump instruction: 104
*********************************************
```

8. Evaluating the Pipelined Microarchitecture

8-1. Analysis

In short, we have built a pipelined MIPS simulator with the implementations above. In this section, we will analysis the branch predictor's performance, and comparison of performance between pipelined microarchitecture and single-cycle microarchitecture.

8-2. Branch Prediction Analysis



	simple3.bin(102)	simple4.bin(10)	gcd.bin(73)	fib.bin(109)	input4.bin(2029699)
ANT	0.98%	10.00%	38.35%	50.45%	0.04%
ALT	90.10%	90.00%	61.64%	49.54%	99.95%
LTP1	98.39%	80.00%	79.45%	61.46%	99.91%
LTP2	98.39%	80.00%	79.45%	50.45%	99.953%
GSH	92.15%	20.00%	69.86%	72.47%	99.954%
LVL2	92.15%	20.00%	68.49%	72.47%	99.955%

The figures above show the branch hit ratio of each branch predictor and their visualizations by the bar plots. From simple3.bin to input4.bin, let's check the branch hit ratio one by one. First is simple3.bin. It has a for-loop iteration that repeats 100 times. The reason that this program branches only one loop iteration, the predictors that use global or local history show slightly lower accuracy of the prediction than the predictors that do not use them. Also, the dynamic branch predictors show a better hit ratio than the static branch predictors. The second is simple4.bin. It repeats the operation by having a recursion 10 times. Due to the small number of loop iterations, the branch target buffer (BTB) cannot be fully trained. If the branch target buffer (BTB) is not fully trained for the prediction, the performance of dynamic branch predictors decreases. As a result, in simple4.bin, the static branch predictor shows higher prediction accuracy than the dynamic branch predictors. The third is gcd.bin. It also used recursion of the loop-iterations, but the logic is more complex than the simple4.bin. Also, the fluctuation of branch prediction hits is rapid. Therefore, the predictors that predict the branch condition with history bits show higher accuracy of prediction than the predictors that do not use it. However, the predictors that used global or local history show lower accuracy than the ones that do not use them. Fourth is fib.bin. This program uses recursion for loop-iterations and has more complex logic than the gcd.bin. The fluctuation of branch prediction hit is higher than gcd.bin and recording the branch results into the global-local history can be a powerful method to enhance the performance of the branch prediction. Therefore, branch predictors that use global or local history show better accuracy than the predictors that do not use them. Last is input4.bin. In this program, all branch predictors show high accuracy of branch prediction. Due to the large scale of the loop-iterations, it shows the ideal difference in the performances of each predictor. As a result, from always not taken to a 2-level local branch predictor, branch prediction hit ratio increases.

From the analysis of the branch predictors, we can find out that the performance of the branch predictors differs from program to program. For example, if the number of loop-iteration is small, then the branch prediction ratio of predictors that uses pattern history gets lower. However, if it is big, then the results go opposite. Also, if the iteration is affected by the former branch prediction, the predictors that use global and local history show higher branch prediction accuracy than the ones that do not use them. According to these results, an enhanced branch predictor does not always guarantee a higher performance of prediction.

8-3. Performance Comparison

Steps	IF	ID	EX	MEM	WB	Delay
Resources	Inst Mem	RF	ALU	Data Mem	RF	
R-type	200	50	100	X	50	400
I-type	200	50	100	X	50	400
LW	200	50	100	200	50	600
SW	200	50	100	200	X	550
Branch	200	50	100	X	X	350
Jump	200	X	100	X	X	200

Figure 59 - Instruction Delay Timetable

```
**********result**********
(1) Final return value: 85
(2) Number of executed instruction: 23372706
(3) Number of executed R type instruction: 5076368
(4) Number of executed I type instruction: 13219741
(5) Number of executed J type instruction: 103
(6) Number of executed memory access instruction: 7116606
(7) Number of executed taken branches instruction: 2029699
**********
```

Figure 60 - Result of input4.bin by Single-Cycle MIPS Simulator

```
**********result**********
Final return value: 85
Number of executed instruction: 23374635
Number of executed memory access instruction: 7116606
Number of executed register operation instruction: 15240278
Number of executed taken branch instruction: 2029699
Number of predicted branch: 2028786
Number of not predicted branch: 912
Number of jump instruction: 104
**********
```

Figure 61 - Result of input4.bin by Pipelined MIPS Simulator

Let's use the same assumption of hardware component accessing time from project 2, which is presented in Figure 59. In single-cycle MIPS microarchitecture, cycle per instruction (CPI) is strictly 1. Therefore, the cycle time is determined by the execution time of the slowest instruction, which is load instruction. According to the assumption above, the cycle time of single-cycle MIPS microarchitecture is 600ps. In pipelined MIPS microarchitecture, cycle per instruction (CPI) differs from the operation of the instructions. A single-cycle in single-cycle architecture is divided into 5 stages and instructions are executed in parallel. Therefore, the total number of cycles increase, but the clock time decreases to 200ps, which is the time of longest

hardware component accessing time in microarchitecture. Then, let's compare the performance of the single-cycle microarchitecture and pipelined microarchitecture due to the results generated from the input4.bin program.

- According to the result from Figure 60 and 61, we can calculate the total execution time of the input4.bin
  - Single-Cycle Microarchitecture: 600ps * 23372706 = 14,023,623,600ps
  - Pipelined Microarchitecture: 200ps * 23374635 = 4,674,927,000ps

From, the following result, we can figure out that the pipelined microarchitecture shows 2.99 (≈ 3) times higher performance than the single-cycle microarchitecture. In short, we can check that pipelined microarchitecture is a more efficient architecture than single-cycle microarchitecture. By applying this architecture to the processor, we can optimize the processor and increase its performance of it.

9. Conclusion

From sections 1 to 8, we have shown the concepts that are used to implement our pipelined MIPS simulator: former single-cycle microarchitecture, multi-cycle microarchitecture, pipelined microarchitecture, performance analysis of following architectures, data dependency handling methods such as stalling, score boarding, and forwarding, control dependency handling methods, and branch predictions concepts including compile time prediction and run time prediction. By applying these concepts, we implemented a real pipelined MIPS simulator that follows the flow of the MIPS data path. Then, we evaluated our simulator and the pipelined microarchitecture. By applying the useful concepts to the pipelined microarchitecture, such as data forwarding and various branch prediction methods, we optimized our pipelined architecture. Due to these optimizations, we enhanced the performance of the processor from the former microarchitecture, which is single-cycle microarchitecture. We have shown the results of the program execution which are generated by pipelined MIPS simulator and evaluated the branch prediction methods and compared the performance with single-cycle MIPS simulator. From this analysis, we found out that the enhanced branch predictor does not always guarantees the higher performance. Also, pipelined microarchitecture shows higher performance compared to single-cycle microarchitecture. In next project, we will add the additional implementation on this simulator, which is cache, to enhance the performance of the microarchitecture.

10. Citations

- *18-447 computer architecture - ECE: course page*. (n.d.). Retrieved May 12, 2022, from https://course.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=onur-447-spring15-lecture8-pipelining-ii-dependence-handling-afterlecture.pdf

- A pipeline latches. A valid bit from the previous stage is used to gate ... (n.d.). Retrieved May 15, 2022, from https://researchgate.net/figure/A-pipeline-latch-A-valid-bit-from-the-previous-stage-is-used-to-gate-the-clock-signal-A_fig1_220847451

- Chongstitvatana, P. (n.d.). Pipeline. Retrieved May 15, 2022, from https://www.cp.eng.chula.ac.th/~prabhas//teaching/ca/pipe.htm

- Constraining multi-cycle path in synthesis. VLSI Tutorials. (2020, June 21). Retrieved May 12, 2022, from https://vlsitutorials.com/constraining-multi-cycle-path-in-synthesis/

- EECC 550 Winter 2012 Home Page. (n.d.). Retrieved May 12, 2022, from http://meseec.ce.rit.edu/eecc550-winter2012/

- Hayes, J. (1978). Computer Architecture and organization. McGraw-Hill.

- Lee, C. (2022). (rep.). Project2. Seoul: LEE CHANGYOON.

- McGill University. (n.d.). Retrieved May 16, 2022, from http://www.info425.ece.mcgill.ca/tutorials/T05-Exceptions.pdf

- Multi-cycle MIPS processor - ETH Z. (n.d.). Retrieved May 12, 2022, from https://syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik_14/21_Architecture_MultiCycle.pdf

- WWW2.EECS.BERKELEY.EDU. (n.d.). Retrieved May 16, 2022, from https://www2.eecs.berkeley.edu/Pubs/TechRpts/1989/CSD-89-552.pdf

- Yoo, S. H. (n.d.). 10-data_dependency.

- Yoo, S. H. (n.d.). 11-control_dependence.

- Yoo, S. H. (n.d.). 12-branch_prediction.