

Multi Cycle MIPS Simulator

Nathaniel Vallen

32221415

Global SW Convergence

Free Days: 5 \rightarrow 0 (all used on this one)

Introduction:

This project is an imitation of a simple multi cycle MIPS processor that takes in a single binary file as an input. The binary file consists of MIPS core instructions with a few exceptions which will be specified in a later part. A multi cycle machine can be seen as a significant upgrade from its former version, the single cycle machine. With that said, it shares a lot of similar properties with a single cycle machine. The general stages undergone are basically the same, namely:

1. Instruction Fetch (IF)
2. Instruction Decode and Register Operand Fetch (ID/RF)
3. Execute/Evaluate Memory Address (EX/AG)
4. Memory Operand Fetch (MEM)
5. Store/Writeback Result (WB)

One main difference that the multi cycle processor has is a new hardware known as a latch. The function of a latch is to store and pass the architectural states at different stages during execution, which can be used to shorten the execution time of a program. While a single cycle machine is already fully capable of performing various calculations and executing different instructions without any noteworthy issue, the execution speed of a single cycle machine is still far from optimal. This is the main improvement focus of the multi cycle machine. Before diving deeper into the detailed process, it is best to first define what are the contributing factors to a CPU's execution time. To put it simply, CPU's execution time can be formulated as follows:

$$CPU\ Time = Clock\ Cycles \times Clock\ Cycle\ Time$$

Where clock cycles can be further defined as:

$$Clock\ Cycles = Instruction\ Count \times CPI$$

CPI (cycle per instruction), as the name suggests, is the number of cycles passed to process a single instruction. With this consideration in mind, we can expand the clock cycle in the formula of CPU time into:

$$CPU\ Time = Instruction\ Count \times CPI \times Clock\ Cycle\ Time$$

In a single cycle machine, the number of CPI is always equal to one due to the nature of the machine processing a single instruction every cycle. While this may sound like a good feature, this property also costs a large amount of clock cycle time. This is the case because every instruction is made to fit in one cycle, meaning that the clock cycle time needs to be stretched to

adjust to the most time-consuming instruction (load word/LW). On the other hand, a multi cycle machine's CPI may vary depending on the instruction while providing a shorter clock cycle time. The multi cycle machine can provide shorter clock cycle time because instead of setting the cycle time to the worst case processing time, the machine adjusts it to the time required to finish a single stage. This tradeoff allows optimization in CPU execution time as multi cycle machines process multiple instructions at the same time instead of finishing one before the other. This method of using instruction-level parallelism is called pipelining. A well-known analogy for this is the laundry analogy, which can be described as follows:

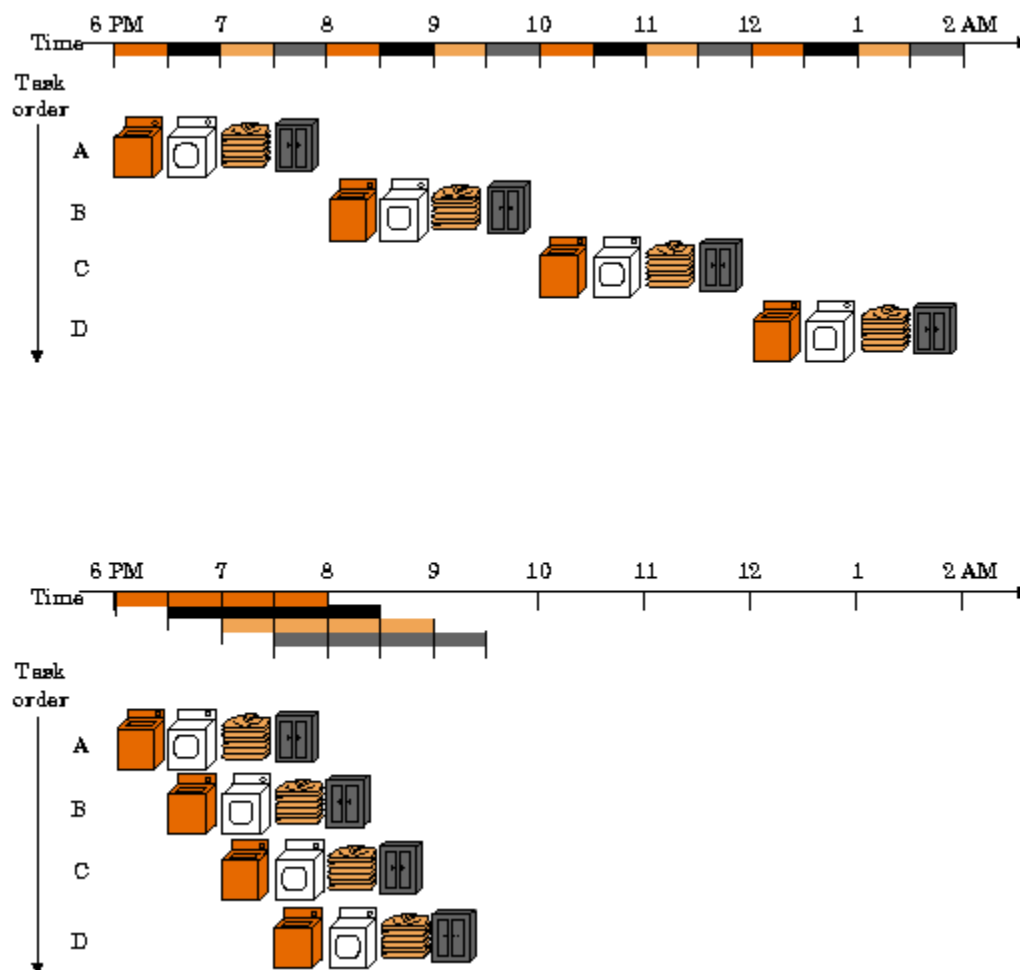


Figure 1. Laundry analogy for multi cycle pipelining

In this example, we see instruction execution as a laundry work. Four stages in total are required to complete a batch of laundry. The first (top) figure shows the sequential completion of four batches of laundry without pipelining. In short, a batch of laundry can only start getting processed when there is currently no laundry being processed in any of the four stages. This is how a single cycle machine processes input instructions. The second figure shows the completion of the same number of laundry batches, but with pipelining. This means that we no

longer have to wait for a batch of laundry to finish before starting to work on the next one. As long as the equipment related to the laundry work stage is idle, we can use that equipment to process a new batch. This leads to a significant reduction in work time. From the pictures above, we can see that pipelining can bring down the time needed to complete four batches from 8 hours to only 3 and a half hours. This is how a multi cycle machine works with instructions.

However, it is important to keep in mind that this example is made with several underlying assumptions. The first assumption is that each stage of the execution takes the same amount of time to complete. In reality, some stages may consume more or less time compared to the others. If this happens, the time reduction pipelining can offer may not be as big of a cut as presented in the previous example due to the presence of waiting time. To understand this, we can use the previous laundry analogy, but this time, stage two of the cycle takes twice as much time to finish in comparison to the other stages. Assuming this is the case, then we can illustrate the process as follows:

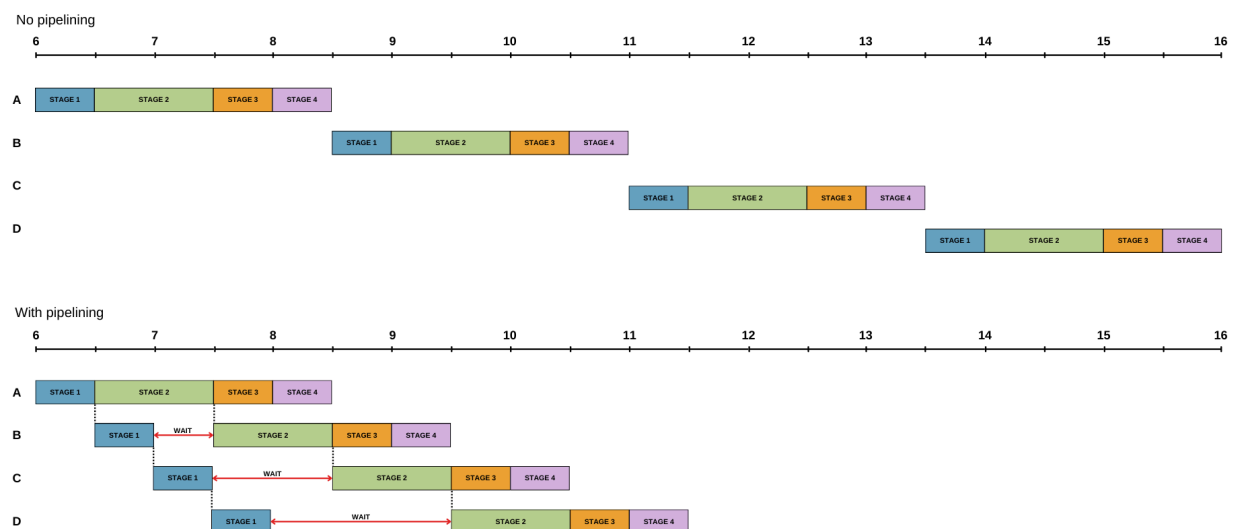


Figure 2. Rough performance comparison between single cycle and multi cycle

Let's compare the two examples we have seen. In the first example, pipelining has reduced the execution time from 8 hours to 3.5 hours. We can roughly calculate that the execution time got reduced by 56.25%. Meanwhile in the second example, pipelining has brought down the execution time from 10 hours to 5.5 hours, resulting in approximately 45% reduced time, which is a step down from the initial 56.25%. This hints that in a series of instruction executions where one stage takes longer processing time than the others, the benefit of pipelining may also seem to be less efficient.

Secondly, in the first example, we also ignored another important factor called data dependency. This means that we assume that the output of each stage does not affect any other stages. Data dependency can be classified into three groups, namely:

1. Flow Dependence (Read after Write/RAW)
2. Anti Dependence (Write after Read/WAR)
3. Output Dependence (Write after Write/WAW)

Each of these dependencies often take place depending on the order of instructions. Several strategies are commonly used to overcome these dependencies. The simplest way to do so is by stalling dependent instructions. This means that when an instruction is detected as an instruction that depends on the output of an instruction before it in ID stage, this instruction will be rendered invalid for the time being until the instruction it is dependent on is completely executed (until WB stage). During this period of time, both IF and ID stage will also be frozen (i.e. no new instructions shall be fetched until the dependency is resolved), leaving EX, MEM, and WB stage to proceed normally.

Another method of resolving data dependencies is by implementing data forwarding or bypassing. The first step is similar to stalling, which is to detect the presence of dependency in a specific instruction. However, further distinctions should be made in this phase. In order to do so, a new hardware called a forwarding unit is introduced. The forwarding unit takes in forwarded values and instruction information (registers used, control signals activated) to detect dependencies, locate its source (MEM or WB stage), and forward required information.

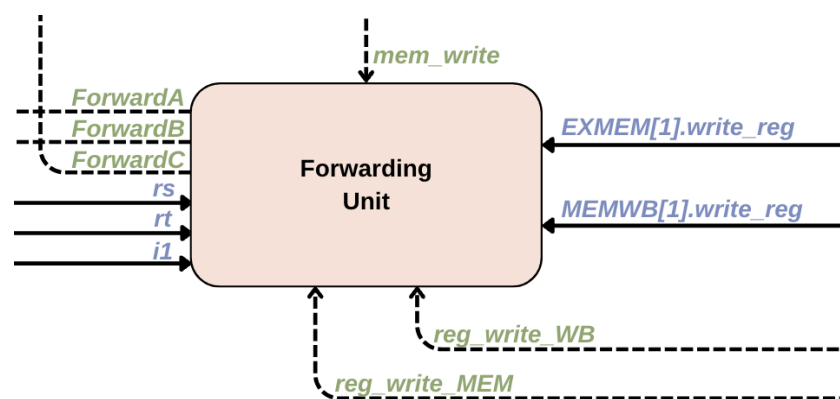


Figure 3. Forwarding unit datapath

While stall may seem much simpler in comparison to data forwarding, it is not the most efficient method to use as stalling is tantamount to significantly increasing the number of cycles, hence inflating the CPU time. With data forwarding, we no longer need to wait until the dependent instruction finishes writing back. Additionally, the simple stalling scheme is proven to

be lacking when faced with some specific circumstances. For example, in the case of output dependency, an improved version of scoreboarding should be used instead of a simple scoreboarding. A simple scoreboarding method utilizes register number, data, and validity to organize which register should be stalled and for how long it should be stalled for. However, in the improved scoreboarding scheme, a new variable is introduced, which is “tag”. When output dependencies take place, a tag is used to indicate which instruction’s register value should be used for a specific instruction’s calculation. In addition, a reservation station can also be applied to serve as a temporary “waiting room” for stalled instructions.

Another factor we ignored in our first example is control dependency. In a single cycle processor, the next value which should be fetched by the PC is calculated and passed in the EX stage. While this is entirely not an issue in a single cycle perspective, in multi cycle, the next value of PC should be determined in the IF stage in order to keep fetching new instructions every cycle. This means that the value of the PC should be ready before the newly fetched instruction passes through the EX stage. Just like data dependency, control dependency also has several solutions. In order to always have the next PC value ready without much delaying, branch prediction is usually the fitting choice. To be exact, branch prediction has to answer the following questions:

1. Is the instruction a branch instruction?
2. Should the branch be taken?
3. What is the branch target address?

There are several ways we can provide answers to these questions by making use of branch predictions. Different types of branch predictions can bring varying prediction accuracy results. The main goal is naturally to aim for the highest average accuracy.

The first type of branch prediction is known as a static branch prediction. Static branch predictions always result in a constant branch outcome prediction, or in other words, it has no direction prediction. For example, an “always taken” branch prediction assumes that every branch’s conditions are true, and the opposite is true for an “always not taken” branch. Another example is “backward taken, forward not taken” (BTFN) branch prediction, which is a result after some considerations that most backward branches (loop/iterations) are taken. Apart from these, other static branch predictions include profile-based prediction and program analysis-based prediction. The second type of branch prediction is called dynamic branch prediction. Unlike static branch predictions, dynamic branch predictions utilizes direction prediction schemes. Before diving deeper into the specifics, dynamic branch predictions require a hardware called a BTB (branch target buffer), and in some cases, a PHT (pattern history table). Generally, a BTB has three columns consisting of PC value, target address, and history branch (HB). After a branch instruction is detected (and mispredicted in the first time), an entry will be

written in the BTB, listing the branch instruction's PC, the target address if the branch is to be taken, and an indicator whether the branch condition was fulfilled or not.

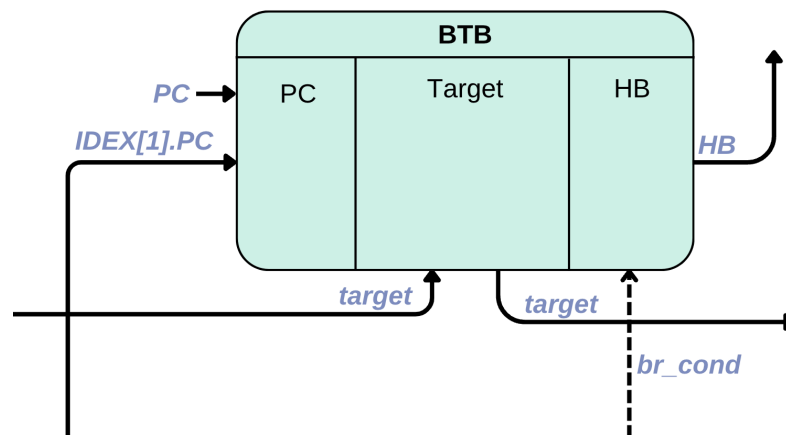


Figure 4. BTB datapath

The most unsophisticated idea of dynamic branch prediction is a last time predictor. This predictor stores a single bit per branch in BTB to specify the branch's direction last time it is executed. While the implementation is simple, this method will always miss the start and the end of a loop. This is due to the nature of branch conditions switching to its opposite state at the start and the end of a loop (to indicate the beginning and the end). Naturally, this makes last time predictor a decent solution for programs with small loops (i.e. loops consisting of small number of iterations), and a poor choice for the opposite case. On top of that, the last time predictor also has a weakness of changing states too quickly, which promotes the depletion of prediction accuracy.

Another idea of a dynamic branch predictor is a 2-bit counter. This method promotes higher accuracy by adding one more bit, meaning that the state will not change to the opposite state right after a single misprediction. We can also implement a saturating counter and/or a hysteresis to further raise the average prediction accuracy. While these are good ideas for a dynamic branch predictor with sufficient accuracy, there is another idea which is also implemented in this project. This scheme is called the global history predictor. A global history predictor utilizes a BTB, BHR, and a PHT in its process of fetching the next predicted address. A BHR (branch history register) holds the last few outcomes of a branch instruction. In this program, the BHR keeps record of the last two outcomes, though in real life implementations, more than two outcomes can be recorded (four, etc.). The value of BHR serves as an index of another array called PHT. PHT contains the predicted branch outcomes given some input (BHR). So, PHT alongside BHR can be used to produce a branch prediction based on the history of previous branch instructions. This, paired with the BTB is what we call a two-level global history predictor.

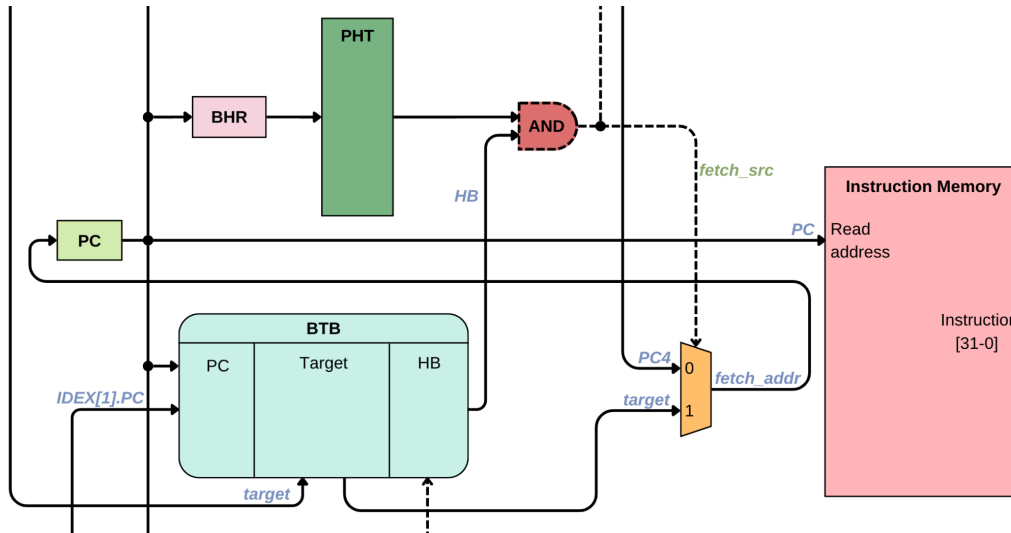


Figure 5. Two-level global history predictor datapath

Important Concepts & Considerations:

- I. The following MIPS core instructions are excluded from this program:
 - A. Load Byte Unsigned (LBU) [24]
 - B. Load Halfword Unsigned (LHU) [25]
 - C. Load Linked (LL) [30]
 - D. Store Byte (SB) [28]
 - E. Store Halfword (SH) [29]
 - F. Store Conditional (SC) [38]
- II. The full datapath of this program can be found [here](#) (the image is horizontally too long, hence is unable to be presented properly in this document).
- III. Below are the details of each variable in the program:

Name	Type	Definition	Additional Notes
pc	int	Value of the pointer pointing to the current instruction in the instruction memory	
pc4	int	Value of pc + 4	
pc8	int	Value of pc + 8	

inst	int	32 bits representing an instruction	
opcode	u_char	The first 6 bits of an instruction that serves as a unique indicator of the instruction	
shamt	u_char	5 bits of an R-type instruction [10:6] that indicates the shift amount	
func	u_char	6 bits of an R-type instruction [5:0] that indicates the specific operation of an R-type instruction	
imm_	u_short	16 bits of an I-type instruction [15:0] that indicates an immediate value	
s_imm	int	32-bit sign extended immediate	
ext_imm	int	32-bit zero extended immediate	
address	int	26 bits of a J-type instruction [25:0] that indicates the address	
j_addr	int	Address to be taken in a jump situation	
br_addr	int	Address to be taken in a branch situation	
target	int	Next value of PC	
rs	u_char	Index of the source register	
rt	u_char	Index of the target register	
rd	u_char	Index of the destination register	
i1	u_char	Index of read register 1	
i2	u_char	Index of read register 2	
d1	int	Value of register i1 (read data 1)	
d2	int	Value of register i2 (read data 2)	
v1	int	Chosen value to be used	Always equal to d1
v2	int	Chosen value to be used	d2 or shamt
write_reg	int	Index of register to update	

write_data	int	Value to update register writeReg	
ALU_in1	int	ALU's input	
ALU_in2	int	ALU's input	
ALU_out	int	ALU's output	
set_cond	u_char	Set condition	
brValCond	bool	Branch value condition	Value-wise condition to take a branch
br_cond	bool	Condition to take branch	Confirmation to take branch by checking brValCond and if either isBeq or isBne is true In practice (the program), brValCond and brCond are merged into one as simply "brCond"
mem_out	int	Output of the memory (value from the read address; read data)	
mem_to_reg	int	Control signal that controls whether writeData should originate from memOut	
mem_write	int	Control signal that indicates that the instruction will write a certain value in the memory	
mem_read	int	Control signal that indicates that the instruction will read a certain value in the memory	
reg_dst	int	Control signal that determines the writeReg	
reg_write	int	Control signal that indicates that the instruction will write a certain value in a	

		register	
ALU_op	char	Control signal that sets the operation done in the ALU	
ALU_src	int	Control signal that determines the SrcInput	
src_input	int	ALU's input based on ALUSrc	
ForwardA	int	Control signal that controls whether ALUIn1 should originate from the forwarded value from MEM stage or WB stage	
ForwardB	int	Control signal that controls whether ALUIn2 should originate from the forwarded writeReg value from MEM stage or WB stage	
ForwardC	int	Control signal that controls whether v2 should originate from the forwarded writeReg value from MEM stage or WB stage	
isJr	bool	Control signal that indicates that the instruction is a JR instruction	
isShift	bool	Control signal that indicates that the instruction involves shifting	
isSet	bool	Control signal that indicates that the instruction involves setting a value less than another value	
isJal	bool	Control signal that indicates that the instruction is a JAL instruction	
isJump	bool	Control signal that indicates that the instruction involves jumping	
isBeq	bool	Control signal that indicates that the instruction is a BEQ instruction	
isBne	bool	Control signal that indicates that the instruction is a BNE instruction	
isLui	bool	Control signal that indicates that the	

		instruction is a LUI instruction	
cycle_count	u_int	Number of cycles passed	
inst_count	u_int	Number of instructions executed	
r_count	u_int	Number of R-type instructions executed	
i_count	u_int	Number of I-type instructions executed	
j_count	u_int	Number of J-type instructions executed	
br_count	u_int	Number of branches taken	
mem_count	u_int	Number of memory accesses performed	
reg	struct_register	An array of 32 registers (0~31) to hold values	
inst	struct_instruction	An instruction (to be broken down in decode)	
forwarding_unit	struct_forwarding_Unit	A unit to organize data forwarding	
BTB	struct_BTBT	An array of BTB entries	Value limits are set to the limits of an u_char (0~255)
mem	int	An array to hold values	Maximum size is 0x400000
IFID	struct_ifid	A latch between IF and ID stage	
IDEX	struct_idex	A latch between ID and EX stage	
EXMEM	struct_exmem	A latch between EX and MEM stage	
MEMWB	struct_memwb	A latch between MEM and WB stage	
BHR	u_char	An array that holds the last 2 outcomes of a branch instruction	
PHT	u_char	An array of predicted branch condition outcomes	

branch_outcomes	bool	An array that holds the last 10 branch condition outcomes	
BTB_out	bool	The result of whether a branch is taken or not according to BTB	
BTB_last_entry	u_int	The latest entry in BTB	
match_index	u_int	The index of BTB that matches a certain PC value	
notFound	bool	An indicator that a matching PC is not found in BTB	
fetch_src	u_char	A control signal that controls whether the next fetch address should be taken from (PC + 4) or a specific jump/branch target	
fetch_addr	int	The next PC value	
stall	bool	An indicator if the pipeline is currently stalled	
halt_IF	bool	An indicator that IF stage is halted	
halt_ID	bool	An indicator that ID stage is halted	
halt_EX	bool	An indicator that EX stage is halted	
halt_MEM	bool	An indicator that MEM stage is halted	
cycles_after_stall	u_char	Cycles passed after pipeline stall is enforced	
excess_cycles	int	Cycles after PC = -1	

IV. In the [datapath](#), the process of calculating br_cond is presented in detail, making it seem a little bit more complicated. However, essentially, it is a simple “if” condition.

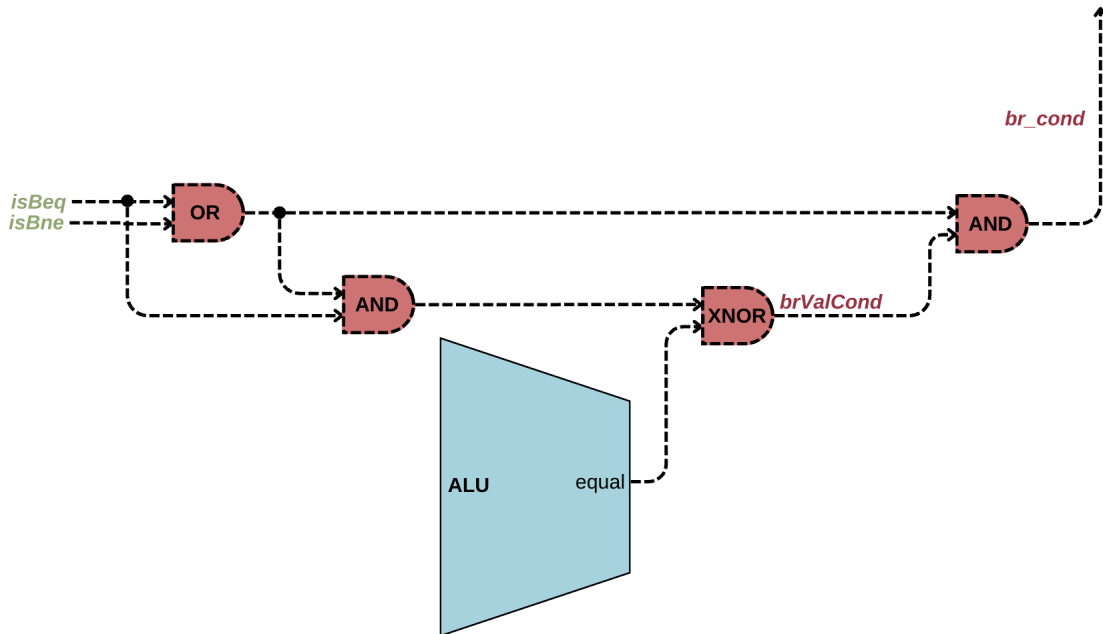


Figure 6. The hidden datapath to determine the value of *br_cond*

Here is the detailed breakdown of the process (from left to right):

- Check whether the instruction is a branch instruction → *isBeq* OR *isBne*
- Check whether the instruction is BEQ or BNE → *isBeq* AND (*isBeq* OR *isBne*)
- Check whether the branch condition is met → {*isBeq* AND (*isBeq* OR *isBne*)} XNOR *isEqual*
- Check whether the branch condition is met and confirm that there indeed is a branch instruction → (*isBeq* OR *isBne*) AND [{*isBeq* AND (*isBeq* OR *isBne*)} XNOR *isEqual*]

- V. The output of the program is printed in a log file instead of the terminal to show varying outputs (see “Screen Captures” section).

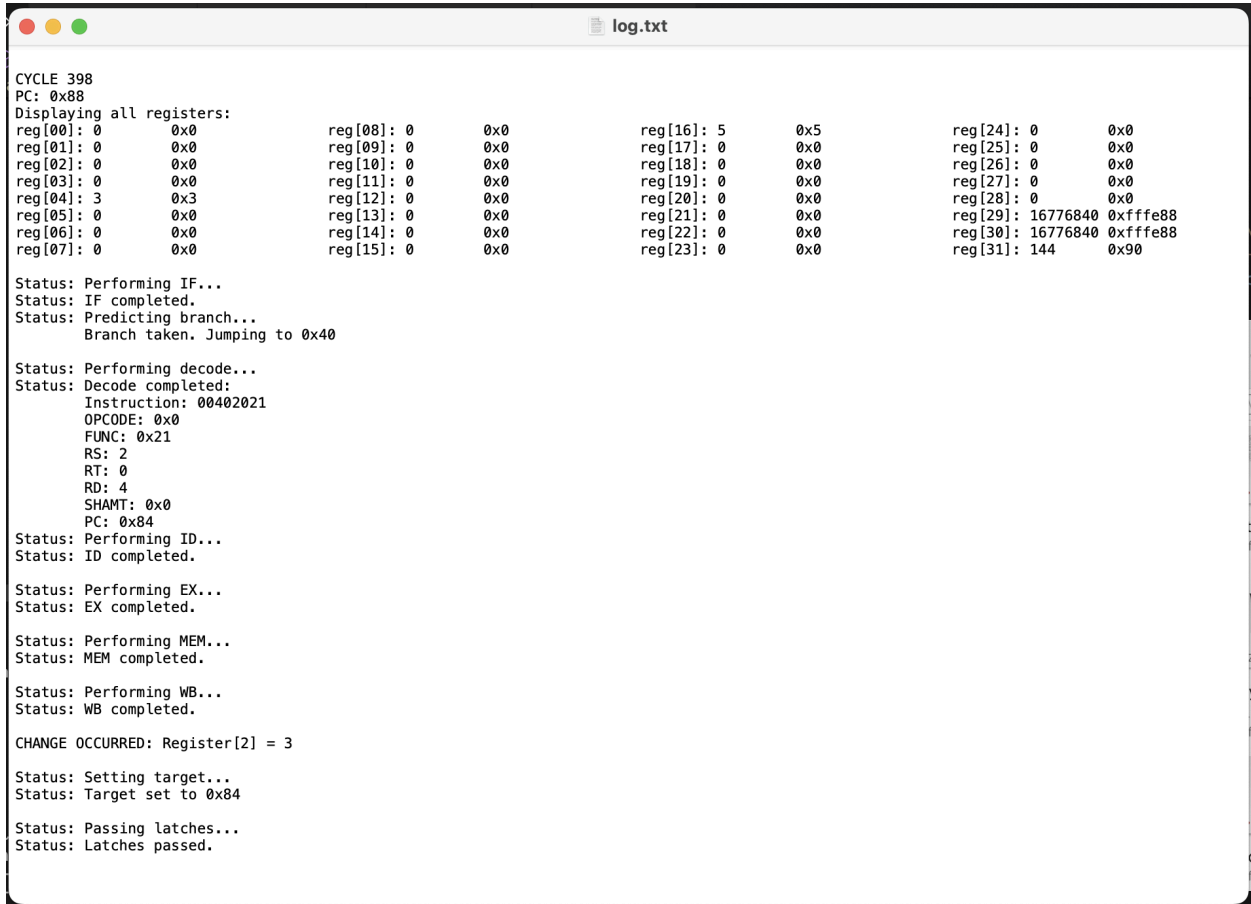
Unique Features

- Implementation of BHR and PHT for two-level global branch prediction.
- Detailed output display of each stage/status.

Build Configurations & Environment

The code in this project was written in C programming language using Visual Studio Code IDE.

Screen Captures



```
CYCLE 398
PC: 0x88
Displaying all registers:
reg[00]: 0      0x0      reg[08]: 0      0x0      reg[16]: 5      0x5      reg[24]: 0      0x0
reg[01]: 0      0x0      reg[09]: 0      0x0      reg[17]: 0      0x0      reg[25]: 0      0x0
reg[02]: 0      0x0      reg[10]: 0      0x0      reg[18]: 0      0x0      reg[26]: 0      0x0
reg[03]: 0      0x0      reg[11]: 0      0x0      reg[19]: 0      0x0      reg[27]: 0      0x0
reg[04]: 3      0x3      reg[12]: 0      0x0      reg[20]: 0      0x0      reg[28]: 0      0x0
reg[05]: 0      0x0      reg[13]: 0      0x0      reg[21]: 0      0x0      reg[29]: 16776840 0xfffe88
reg[06]: 0      0x0      reg[14]: 0      0x0      reg[22]: 0      0x0      reg[30]: 16776840 0xfffe88
reg[07]: 0      0x0      reg[15]: 0      0x0      reg[23]: 0      0x0      reg[31]: 144      0x90

Status: Performing IF...
Status: IF completed.
Status: Predicting branch...
Branch taken. Jumping to 0x40

Status: Performing decode...
Status: Decode completed:
Instruction: 00402021
OPCODE: 0x0
FUNC: 0x21
RS: 2
RT: 0
RD: 4
SHAMT: 0x0
PC: 0x84
Status: Performing ID...
Status: ID completed.

Status: Performing EX...
Status: EX completed.

Status: Performing MEM...
Status: MEM completed.

Status: Performing WB...
Status: WB completed.

CHANGE OCCURRED: Register[2] = 3

Status: Setting target...
Status: Target set to 0x84

Status: Passing latches...
Status: Latches passed.
```

Figure 7. Random output screenshot [fib.bin]

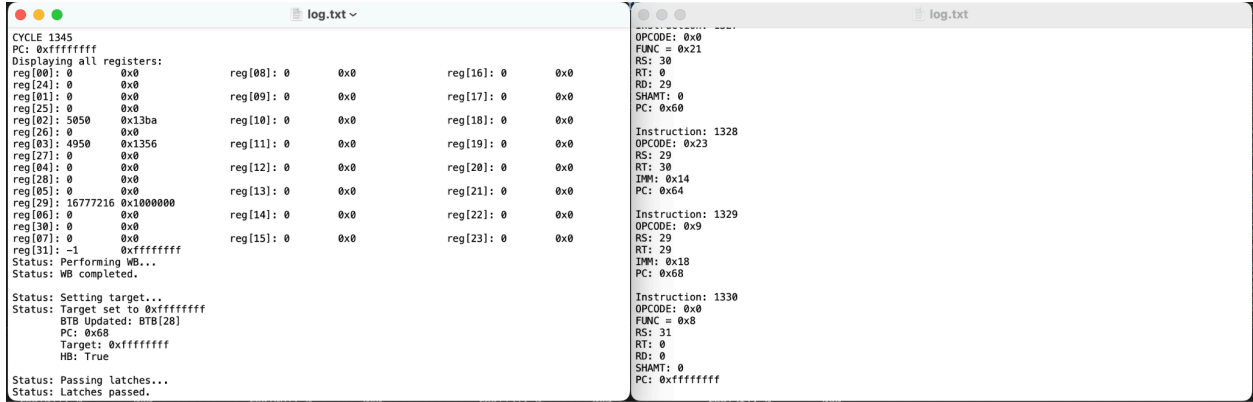


Figure 8. Comparison between multi cycle (left) and single cycle (right) performance [simple3.bin]

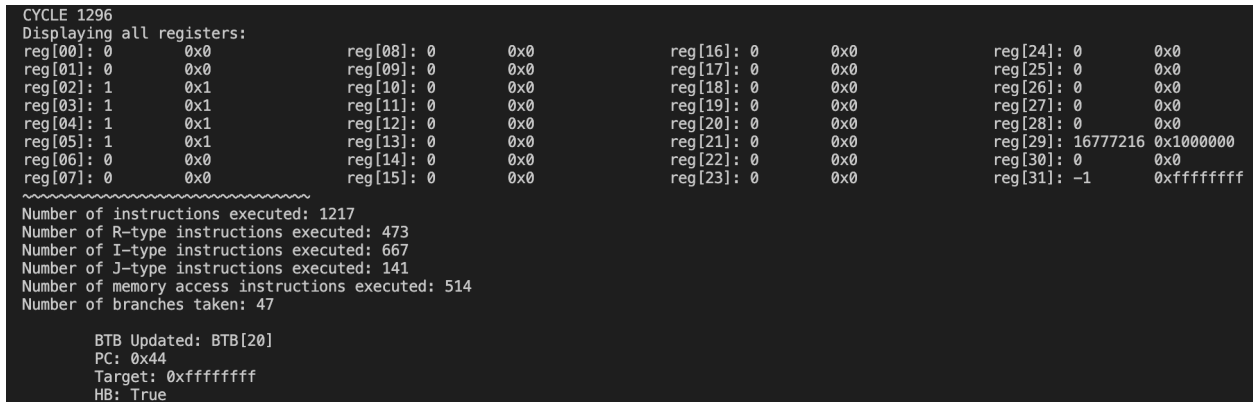


Figure 9. Output on terminal (showing instruction details and BTB entry update) [gcd.bin]


```
log.txt
CYCLE 27
PC: 0x7c
Displaying all registers:
reg[00]: 0      0x0      reg[08]: 0      0x0      reg[16]: 0      0x0      reg[24]: 0      0x0
reg[01]: 0      0x0      reg[09]: 0      0x0      reg[17]: 0      0x0      reg[25]: 0      0x0
reg[02]: 10     0xa      reg[10]: 0      0x0      reg[18]: 0      0x0      reg[26]: 0      0x0
reg[03]: 10     0xa      reg[11]: 0      0x0      reg[19]: 0      0x0      reg[27]: 0      0x0
reg[04]: 10     0xa      reg[12]: 0      0x0      reg[20]: 0      0x0      reg[28]: 0      0x0
reg[05]: 0      0x0      reg[13]: 0      0x0      reg[21]: 0      0x0      reg[29]: 16777144 0xfffffb8
reg[06]: 0      0x0      reg[14]: 0      0x0      reg[22]: 0      0x0      reg[30]: 16777144 0xfffffb8
reg[07]: 0      0x0      reg[15]: 0      0x0      reg[23]: 0      0x0      reg[31]: 28      0x1c

Status: Performing IF...
Status: IF completed.
Status: Predicting branch...

Status: Performing decode...
Status: Decode completed:
Instruction: 00000000
OPCODE: 0x0
FUNC: 0x0
RS: 0
RT: 0
RD: 0
SHAMT: 0x0
PC: 0x78
Status: Performing ID...
Status: ID completed.

Status: Performing EX...
Status: EX completed.

Status: Performing MEM...
Status: MEM completed.

Status: Performing WB...
Status: WB completed.

CHANGE OCCURRED: Register[2] = 9

Status: Setting target...
Status: Target set to 0x34
BTB Updated: BTB[2]
PC: 0x74
Target: 0x34
HB: True
Status: Branch mispredicted. Jumping back to 0x74
Status: IF flushed.
Status: ID flushed.

Status: Passing latches...
Status: Latches passed.
```

Figure 10. Output of a cycle with a mispredicted branch [simple4.bin]