Computer Architecture Final Project Report

Yun-Chun Chen
b03901148
Department of Electrical Engineering
National Taiwan University

Yu-Chuan Chuang
b03901142
Department of Electrical Engineering
National Taiwan University

Bo-Wen Chen
b03901036
Department of Electrical Engineering
National Taiwan University

***Abstract*—We design a pipelined MIPS along with multiple branch predictors and different L1 and L2 caches to enhance the performance. In this report, we will firstly introduce the design philosophy of our pipelined MIPS, secondly, we will discuss the branch predictors in detail and analyze the pros and cons of different predictors, compare the experimental results, and draw some conclusions. Lastly, we will elaborate the design philosophy of the L1 and L2 caches. We conduct some experiments and compare the results of different combinations of the L1 and L2 caches.**

I. Baseline

We have designed a single-cycle MIPS in HW3. To reduce cycle length and enhance the performance of single-cycle MIPS, we design a MIPS which is embedded with pipelines and is able to execute all the instructions listed in Table 1. After finishing the design, we test our pipelined MIPS which combines the instruction cache, data cache, and the memory we designed before together to sort Fibonacci series written under assembly language and evaluate the performance.

TABLE I
Required Instruction Sets For Pipelined MIPS

Name	Descrpition	Name	Descrpition
ADD	Addition	SRL	Shift right logical
ADDI	Addition immediate with sign-extension	SLT	Set less than
SUB	Subtraction	SLTI	Set less than variable
AND	Boolean logic operation	BEQ	Branch on equal
ANDI	Boolean logic operation	J	Unconditionally jump
OR	Boolean logic operation	JAL	Unconditionally jump and link
ORI	Boolean logic operation	JR	Unconditionally jump to address in $31
XOR	Boolean logic operation	JALR	Jump and link
XORI	Boolean logic operation	LW	Load word from data memory
NOR	Boolean logic operation	SW	Store word from data memory
SLL	Shift left logical	NOP	No operation
SRA	Shift right arithmetic		

II. Design philosophy for the pipelined MIPS

Fig. 1 presents the whole picture of the architecture of the pipelined MIPS. In this architecture, there are 5 stages, IF, ID, EX, MEM, and WB, and 4 four big registers: IF/ID, ID/EX, EX/MEM, and MEM/WB which are designed as a buffer to store the value from the previous stage and output the value to the next stage. As we can observe from the diagram, each stage is assigned to perform different tasks. In IF stage, there is a program counter (PC) which provides the address for instruction fetching. The instruction will be decoded in ID stage and, in the meantime, the register will output the corresponding data according to the decoded instruction. Afterwards, the control unit will output the control signals to the subsequent stages. In EX stage, the ALU control unit decodes the instruction and outputs ALU control signal to ALU unit. Based on the ALU control signal, ALU unit will perform the corresponding arithmetic operation accordingly. In MEM stage, the instruction can access the data cache to obtain or store values. Lastly, in WB stage, the value passed from MEM stage will be written back to register depending on the WB signal. By separating the architecture into the above-mentioned stages, the pipelined MIPS will not only perform multiple instructions simultaneously but can enhance the efficiency by maximizing the utilization of every single stage.

Although the 5-stage architecture can execute all the listed instructions, there are still many cases we have to take into concern. For example, if the instruction is jal, we must decide the jump address for the next instruction and store PC + 4 into register $31. On the other hand, in pipelined MIPS, there are two hazards that might occur: ”Branch Hazard” and ”Data Hazard”. In order to handle the hazard issue, we need to incorporate other modules into the original pipelined MIPS: the forwarding unit and the hazard detection unit. In the following sections, we will explain how to tackle these issues by incorporating the additional modules in detail.

A. J type

There are 4 different functions that are all categorized as a J type function: jump, jal, jr, and jalr. ”Jump” is to jump to the address indicated by the jump instruction. Jal also works in the same way as ”Jump”, but meanwhile, the circuit needs to store the value ”PC + 4” into register $31. The jump address of Jal, however, only depends on the address stored in register $31 whereas jalr is function combination of jal and jr.

The purple lines depicted in Fig. 1 indicate how we handle J type instructions. When the instruction is recognized as J type, it will be decoded in ID stage where

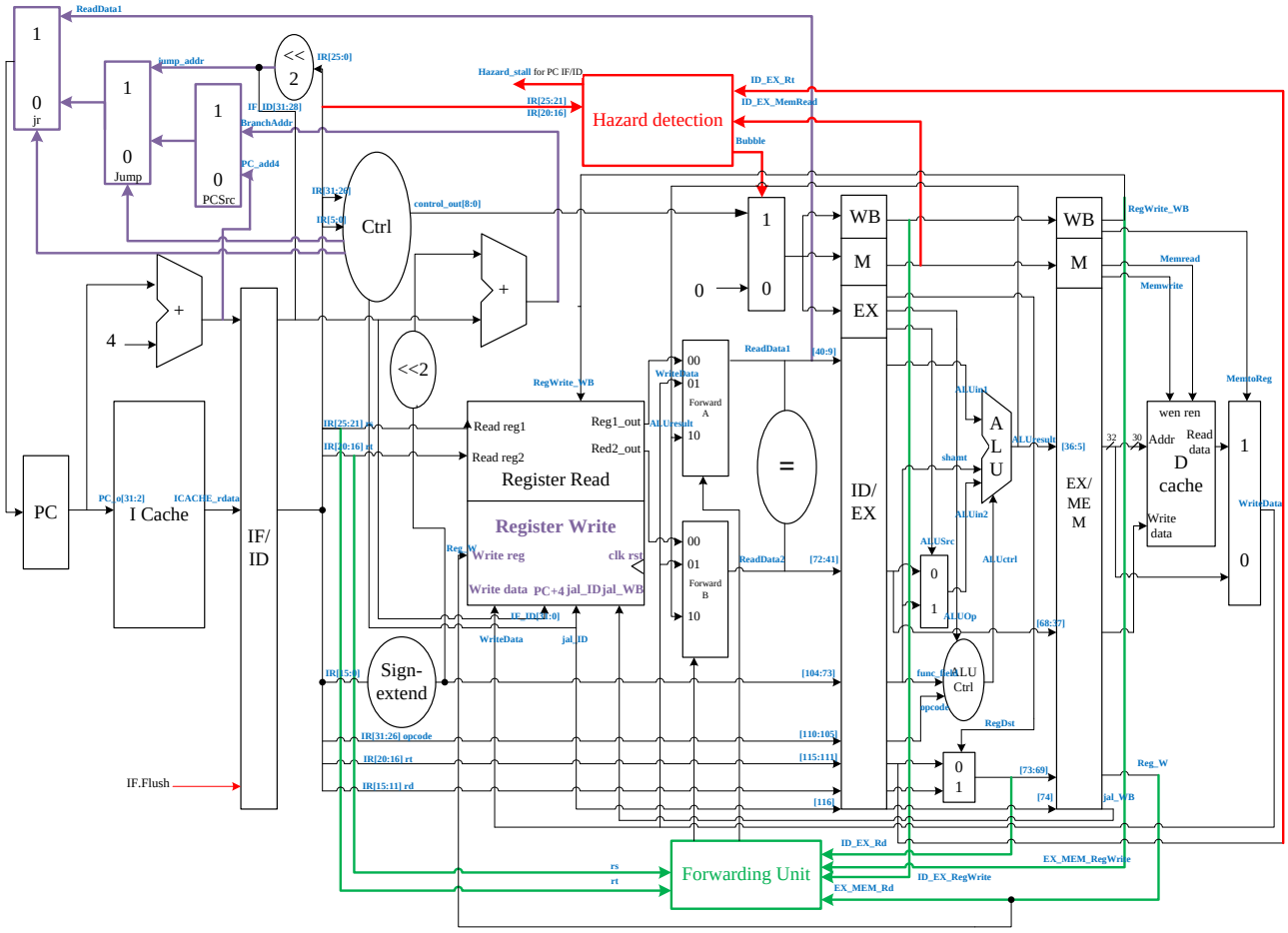

Fig. 1. Circuit Architecture of the Pipelined MIPS

the jump address will be calculated in the meantime. Control unit outputs control signals jump and jr which serve as the selected signals for multiplexers colored purple. After ID stage finishes decoding the J type instructions, the multiplexers will pass the jump address to the program counter. At the same time, since J type instruction is decoded in ID stage, IF stage will fetch the new address which is the address of J type instruction + 4 and is also the wrong address. Therefore, after ID stage passes the jump address to the program counter, we have to flush the undesired value stored in the IF/ID register.

So far we have explained how to handle jump and jr instructions, now we have to deal with jal instruction. The difference between jal and jump is that jal not only has to jump but store the address PC + 4 into the register $31 and under this scenario, there is another thing that needs to be taken into concern. In order to avoid structure hazard which will take place when more than 1 instructions collide at the same stage, we manually adjust the the length of each instruction such that the instruction has to pass every stage even though some instructions may do nothing in some stages such as store word (sw) instruction will do nothing in WB stage. If we apply this mechanism to all the instructions, jal instruction will have to pass every stage but the address PC + 4 will not be stored into register $31 until the instruction arrives WB stage. If the following instruction is jr, jr will fetch the wrong address stored in register $31 because the address PC + 4 carried by jal, which needs to be stored in register $31, is still in EX stage at this moment.

To resolve this issue, we divide the instruction register write into two parts, one is designed for jal instruction in particular and the other one is designated for other instructions which have to write the value back to the corresponding register such as add. Under this adjustment, when ID stage decodes the instruction such as jal, in the next cycle, the address PC + 4 will be directly written into register $31 meaning that the address does not need wait until the address is passed to the WB stage. In addition, if the following instruction is jr, the program counter will fetch the right address stored in register $31 accordingly.

(1) presents the pseudo code for register write. The first if statement is for jal instruction. If the jal signal

```
always@( posedge clk) begin
    if (jal_ID)
        register[31] <= PC+4 ;

    if (RegWrite_WB && jal_WB != 1)
        register[Reg_W] <= WriteData ;
end                                                    (1)
```

Fig. 2. Pseudo Code for Register Write

passed from the control unit in ID stage is 1, after the next cycle, PC + 4 will be written back to register $31 immediately. In the second if statement, when the register write signal coming from the WB stage is 1 and jal signal is 0, the data in WB stage will be written into register. The reason why we have to add "jal_WB != 1" in the statement is because the control unit will output "register write = 1" for every jal instruction. In order to prevent jal from writing back to register twice, we add this statement "jal_WB != 1" to tell MIPS when jal instruction is operating in WB stage, PC + 4 does not have to be written back again.

B. Forwarding Unit

In pipelined MIPS, data hazard will occur when instructions operating in different stages of pipelines modify the data that has dependency with others. There are two different stages that may result in data hazard: ID stage and EX stage.

In EX stage, ALU unit performs arithmetic operation on two values in ID stage.

```
add $t3,$t1,$t2
add $t4,$t3,$t1                                        (2)
```

Take equation (2) for an example, the first instruction calculates $t1 + $t2 and stores the result into $t3. The second instruction calculates $t3 + $t1 and stores the result in $t4. Unfortunately, when the second instruction gets the value stored in $t3 in ID stage, the new value of $t3 calculated by the first instruction has not written back to the corresponding register yet. This is a simple demonstration of data hazard.

In ID stage, we make branch decision. When beq instruction operates in ID stage, MIPS will compare two values and decide whether beq is true.

```
add $t3,$t1,$t2
beq $t4,$t3,100                                        (3)
```

Take equation (3) for an example, the first instruction computes $t1 + $t2 and store the result into $t3. The second instruction compares the value stored in $t4 and $t3. However, when beq operates in ID stage, it will fetch the wrong value stored in $t3 since the new value of $t3 hasn't written back yet. This is another example that illustrates the occurrence of data hazard.

To handle data hazard, we have to store the result back to the corresponding register after the calculation. Therefore, we design a forwarding unit which is destined to store the result back to the corresponding register after the calculation. Forwarding unit will detect rs, rt, and rd in ID, EX, and MEM stages. When data hazard takes place, forwarding unit will directly forward the new value to the previous stages. Although both ID and EX stage have the possibility to encounter data hazard, we don't need to incorporate an additional forwarding unit meaning that we only need 1 forwarding unit. This is because when we detect data hazard happens between ID and EX or ID and MEM, forwarding unit will forward the newest data to the ID stage. After that, the data hazard issue will no longer take place since the value has been updated in the ID stage.

```
always@(*) begin
  if(EX_MEM_RegWrite && EX_MEM_Rd == IF_ID_Rs
&& !(ID_EX_RegWrite&&ID_EX_Rd==IF_ID_Rs))
    ForwardA = 2'b01;
  else
  if(ID_EX_RegWrite&&ID_EX_Rd==IF_ID_Rs)
    ForwardA = 2'10;
  else
    ForwardA = 2'b00;
end                                                    (4)
```

Fig. 3. Pseudo Code for Forwarding Unit

(4) roughly presents the pseudo code for forwarding unit. The first if statement means that there is a data hazard between ID stage and MEM stage while, at the same time, there is no data hazard that happens between ID stage and EX stage. Forwarding unit will forward the new value in MEM stage to ID stage. When data hazard occurs in EX and MEM stage simultaneously, EX stage will possess the newest value than that operated in the MEM stage. Therefore, if this case happens, the second if statement will be true and the forwarding unit will forward the newest value computed in the EX stage to the ID stage.

C. Hazard detection unit

There is a special case for data hazard. Take (5) for an example, when the load word instruction operates in the EX stage while the add instruction operates in the ID stage, addition requires the newest value stored in $t3. When forwarding unit detects data hazard, it will forward the value stored in $t3 in EX stage; however, the load word instruction will not obtain the newest value until it arrives MEM stage. Consequently, we need to insert a bubble between the load word instruction and the add instruction to make sure that the load word instruction can read the right value from data cache in

MEM stage. That is reason why we have to perform a hazard detection.

```
lw $t3,0($t1)
add $t4,$t3,$t1                    (5)
```

Besides, when hazard detection unit inserts a bubble, it has to stall all the instructions operating in the IF stage and the ID stage. (6) presents the pseudo code for the hazard detection unit.

```
always@(*) begin
 if(ID_EX_MemRead && (ID_EX_Rt=IF_ID_Rs/Rt)
     Stall  = 1;
     Bubble = 1;
 else
     Stall  = 0;
     Bubble = 0;
end                                (6)
```

D. Area and Simulation Time

After we pass the test bench in RTL level, we start to synthesize the MIPS circuit and run post simulation. Table 2 lists the area and the total simulation time of MIPS. We keep decreasing the value of set cycle until the time slack reaches negative.

TABLE II
AREA AND TOTAL SIMULATION TIME OF MIPS

Set cycle (ns)	Area (μ m^2)	Time (ns)	Cycle length (ns)	AT
10	255804	19983	9.5	5011816332
8	252277	14824	6.9	3739754248
5	244624	10937	5.1	2675452688
4	279376	9540	4.3	2665247040
3.3	281113	7576	3.37	2129712088

It is worthy to notice that the relationship between the set cycle and area. Fig. 3 depicts the relationship between set cycle and area. We can see that the area keeps shrinking before set cycle reaches 6. However, when keep decreasing the value of set cycle, area will increase gradually. This phenomenon is because before set cycle is equal to 6, the synthesis tool does not use the optimal way to synthesize MIPS. Therefore, when keep decreasing set cycle, synthesis tool will find a better way of synthesizing with less area to synthesize MIPS. However, after set cycle reaches a critical point, if we want MIPS to have a lesser cycle length, synthesis tool will optimize the cycle length but will result in an even larger area. That is why area will become larger in the last part. Although getting a shorter cycle length, we have to sacrifice area. In general, the value of AxT still becomes smaller in the very beginning of the decreasing process of the cycle length.

We synthesize the pipelined MIPS and 2 caches together. From HW5, the area of the cache is about 67739

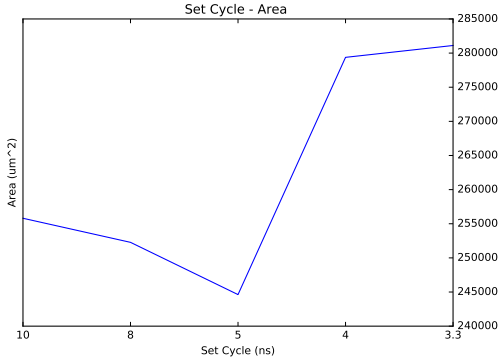

Fig. 4. Relationship between Set Cycle and Area

(m$\hat{2}$). The area of 2 cache is approximately the same as that in pipeline MIPS. This phenomenon indicates that a good design for a cache is as important as MIPS.

E. Some Possible Improvements

Although the MIPS we designed is functionally complete, there are still some possible improvements that we can do to enhance the performance.

Firstly, from section B. and section C., it shows that when MIPS meets J type instruction and beq instruction, it needs to waste 1 cycle to flush IF/ID register. When the test bench grows larger in size, we can imagine that MIPS will waste a lot of cycles for flushing, which will result in a low efficiency. As a result, we hope that J type instructions can be operated in IF stage so that MIPS won't have to perform jump instruction until it reaches ID stage and, by doing so, we can save 1 cycle in each occurrence. For branch instruction, we hope that branch can be predicted in ID stage to reduce the wasted cycle. We will make a clear explanation on how to do it in the extension: Branch Prediction section.

Secondly, since data cache contains only 32 words, it is easy to occur a read miss or a write miss, which will waste a lot of cycles waiting to get the value from the memory or write the value back to the memory. Therefore, we add another bigger sized cache in data cache to reduce the read/write miss rate to enhance the performance of MIPS. We will make a detailed explanation on how to do it in the extension: L2 Cache section.

F. Cache

We use the same cache as that in HW3. Table 3 illustrates the specification of the cache.

III. BRANCH PREDICTION UNIT

The architecture we designed so far has a drawback whenever a beq instruction is encountered. To mitigate this issue, we design different types of branch prediction unit and incorporate them into the pipelined MIPS architecture. Branch prediction unit is used to guess

TABLE III
SPECIFICATION OF CACHE

Characteristic	Specification
Cache size	32 words
Block size	4 words
Way number	Direct mapped
Writing policy	Write back

which way a branch will go before this is known for sure. The purpose of the branch prediction unit is to improve the flow in the instruction pipeline especially when the instruction pipeline is designed to be very long in length. In practice, branch prediction unit plays a crucial role in enhancing the performance of the pipelined architectures. We implement 3 different kind of branch prediction units and the details will be discussed in the following sections.

A. Circuit Design

The circuit diagram of the branch prediction unit is depicted in Fig. 4. There are 2 things need to be mentioned. Firstly, we place our branch prediction unit in the instruction fetch stage (IF stage) for early prediction. Branch prediction unit will output a signal named "BrPred" which serves as the selection signal for the right multiplexer of the program counter illustrated in Fig. 4. If the BrPred signal is 1, the multiplexer will select BranchAddress for program counter, which means that the branch is predicted to be taken. On the contrary, if the BrPred signal is 0, the multiplexer will select PC + 4 for the program counter meaning that the branch is predicted not to be taken. After the beq instruction enters ID stage, the comparator module will decide whether the beq should be taken or not. If the branch prediction is wrong, the comparator will send a signal PredWrong to the left multiplexer illustrated in Fig. 4. This multiplexer will select the right address to the program counter. Meanwhile, the comparator will send a flush signal to IF/ID stage to flush the mis-predicted instruction in the IF/ID register.

Secondly, Fig. 5 shows that we place the J type multiplexers in the IF stage instead of that placed in the ID stage which is shown in Fig. 1. The advantage is that when the instruction is jump, the jump address is directly calculated in the ID stage and will be given to the program counter. Therefore, compare with the architecture presented in Fig. 1, this design can save 1 cycle for the jump instruction. However, the jr instruction cannot reduce the wasted cycle in this design since it still need to wait for the value stored in register $31 in ID stage.

In the later sections, we will describe the design philosophy of the branch prediction unit in detail and elaborate the pros and cons with respect to each of them.

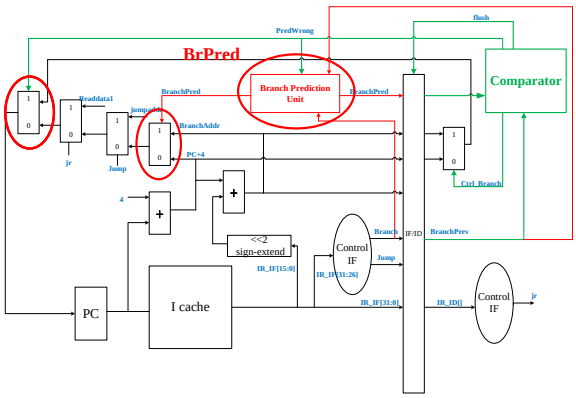

Fig. 5. Architecture of the Branch Predictor

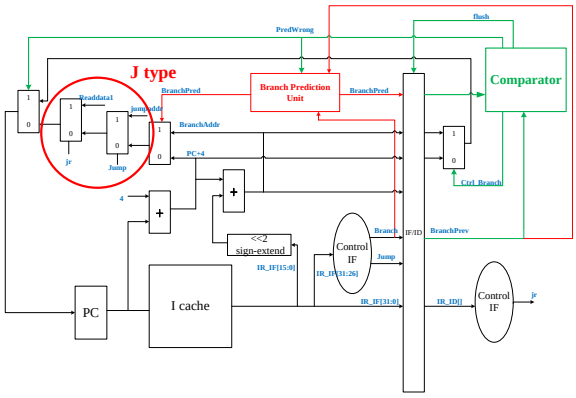

Fig. 6. Architecture of the Branch Predictor

B. 1-bit Predictor

As mentioned in the textbook, a 1-bit predictor is the simplest but most intuitive one in designer's perspective when branch prediction is taken into consideration. The transition state table is given in Fig. 7. It is obvious that there are only 2 states contained in the transition state table. To decide the next state, we only need to consider the current state and whether the branch will be taken or not.

C. Pros and Cons

Although 1-bit predictor is easy to design and implement, the short coming is that when encountering inner loop branches it will mis-predicted twice. Besides, there still exists a disadvantage that the decision of next state only depends on the current state and whether the branch will be taken or not, which implies that under this settlement, next state will be dependent on the current state only and independent of the other previous states. The experimental results will be illustrated in the

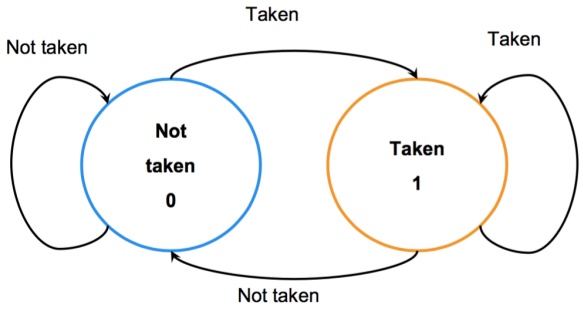

Fig. 7. Transition State Table

following section, and at that moment, we will compare 1-bit predictor with other branch predictors as a whole.

D. 2-bit Predictor

2-bit predictor is a more advanced branch prediction unit. As illustrated in the textbook, 2-bit predictor has 4 different transition states shown in Fig. 8. We name each of the state as "strongly taken", "weakly taken", "weakly not taken", and "strongly not taken" corresponding to the 4 states depicted in the transition state table. To decide the next state, not only do we need to consider our current state, since there are 4 different states that can be chosen, we also need to consider the branch will be taken or not. From a designer's perspective, designing a 2-bit branch prediction unit is a more complex task compared to the 1-bit predictor. However, experimental results demonstrate that the performance of the 2-bit predictor is at least or even beyond the 1-bit predictor.

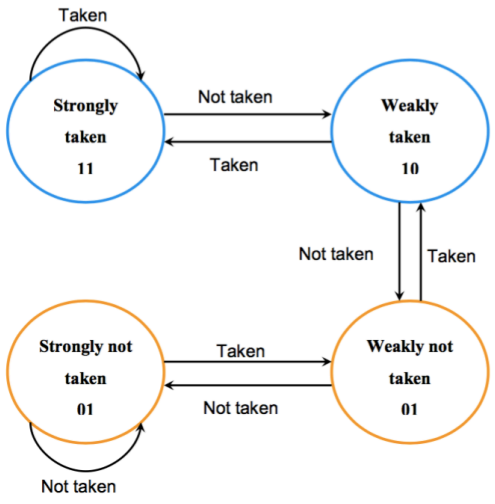

Fig. 8. Transition State Table

E. Pros and Cons

2-bit predictor is designed to overcome the inner loop branches issue. From theoretical point of view, when encountering inner loop branches, it will diminish the probability of mis-predicting twice. However, there is still a disadvantage same as that mentioned in the 1-bit predictor, the dependency of the next state on dependent on the current state and independent of the other previous states. The experimental results will be illustrated in the following section. We will jointly compare the 2-bit predictor with other branch predictors.

F. 2-level Adaptive Predictor

An adaptive predictor is a more sophisticated architecture that takes the past records into consideration. To elaborate the working principle of the adaptive predictor in detail, we take the 2-level adaptive predictor for example. A 2-bit 2-level adaptive predictor contains a 2-bit history register which records the true condition of the past 2 branches. Namely, from the 2-bit history register, we can easily observe the past 2 branch conditions. If the 2-bit history register records 11, this means that the 2 previous branch conditions are both truly taken. Besides, for the 2-level adaptive predictor, there are 4 ($2^2 = 4$) individual transition state tables that are mutually independent. Each of the transition state table contains a 2-bit predictor which records the state condition of different history records. For example, if 10 is recorded in the history register, branch prediction will depend on the corresponding 2-bit predictor according to the history state. Under this settings, different history tables will correspond to different 2-bit predictors which is quite different from the above-mentioned 2-bit predictor which there only exists one transition state table and the transition state table is shared with all conditions.

G. Pros and Cons

Adaptive predictor is designed to consider the history records and predicts the next state by the current state, history records, and whether the branch will be taken or not. From theoretical point, we know that the performance of the adaptive predictor will at least or even better than the 2-bit predictor.

IV. Analysis

We conduct 3 different experiments, record the results, and use python to plot the corresponding curve for visualization. In the following experiments, we would like to know the performance of different branch predictors with respect to different given parameters.

A. Experiment 1

There are 3 adjustable parameters that can generate different kinds of test benches to test the performance of the designed branch predictors. In experiment 1, the parameter "b" and "c" is fixed to 5 while parameter "a" serves as the dependent variable. Parameter "a" serves as the index of the for loop. The greater the "a" is, the more times the loop will have to be executed. The experimental result is shown in the following figure.

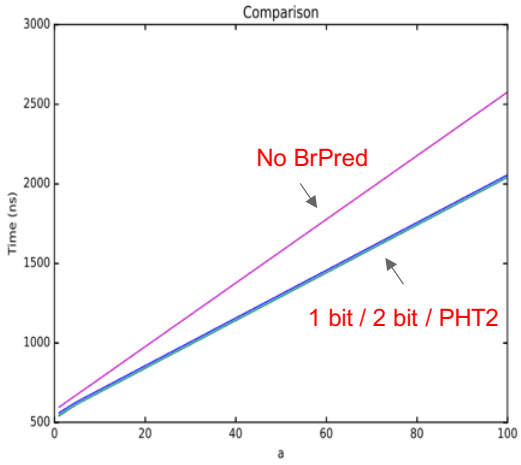

Fig. 9. Comparison of Different Branch Predictors

As we can see that, ”No BrPred” performs the worst while the other predictors perform at the same level. The reason for this phenomenon is that whenever ”No BrPred” encounters a beq command, it will waste a cycle due to the flush operation. The curves of the other predictors will almost overlap is because these predictors will only predict wrong in the begin and the end of the loop. In this case, we can conclude that the variation of ”a” will not result in different performances of different predictors.

B. Experiment 2

In experiment 2, the parameter ”a” and ”c” is fixed to 5 while parameter ”b” serves as the dependent variable. The variation of ”b” will result in the number of the repetition of two consecutive beq commands. That is, the greater of the parameter ”b” is, the greater the number of two consecutive beq commands will be. The experimental result is illustrated in the following figure.

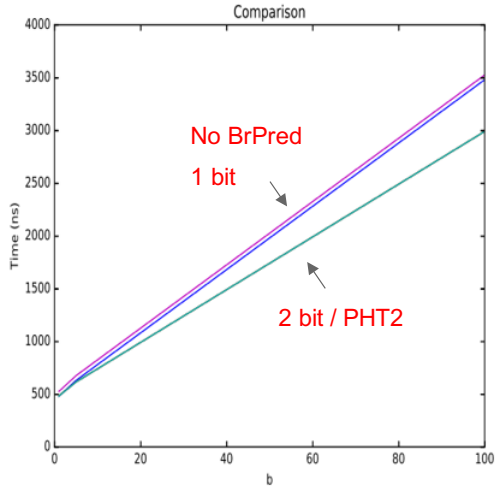

Fig. 10. Comparison of Different Branch Predictors

From the experimental result, we can see that ”No BrPred” performs the worst, 1-bit predictor is slightly better than ”No BrPred”, and the 2-bit and 2-level adaptive predictor perform the best. The reason that the curve corresponding to the ”No BrPred” predictor is parallel to that corresponding to 1-bit predictor is that whenever ”No BrPred” meets beq, it will waste 1 cycle for the flush operation. On the other hand, under this test bench, whenever 1-bit predictor encounters 2 consecutive beq commands, it will mis-predict both beqs. 2 mis-predictions for 1-bit is the reason that the curve of it will be parallel to that of ”No BrPred”. The small margin between the two curves is due to the effect of parameter ”a” and ”c”. The reason that the curves of 2-bit predictor and PHT2 will overlap is because 2-bit predictor will predicts right in the first beq but predicts wrong in the second beq. On the other hand, PHT2 predicts right in both beqs, but to get a right history state for the upcoming beq, we need to insert a NOP since updating history states needs 2 cycles. Therefore, even though PHT2 predicts right in each of the beq, there is a stall between the first and the second beq commands. A mis-prediction in 2-bit predictor and a stall in PHT2 can be viewed as a cancellation and this is why the curves of 2-bit and PHT2 will overlap.

C. Experiment 3

In experiment 3, the parameter ”a” and ”b” is fixed to 5 while parameter ”c” serves as the dependent variable. The variation of parameter ”c” will lead to different times of repetition of the beq command. Namely, the greater the parameter ”c” is, the more repetition of beq command will be. The experimental result is depicted in the following figure.

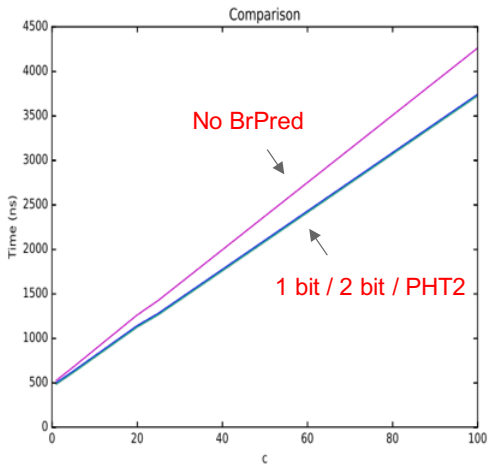

Fig. 11. Comparison of Different Branch Predictors

As we can see that, ”No BrPred” performs the worst while the other predictors perform at the same level. The reason of this phenomenon is same as that elaborated in experiment 1.

V. Result

Table 4 and Table 5 show the execution cycles for each MIPS circuit with different branch prediction units under test bench I_mem_hasHazard and I_mem_BrPred. We can observe that MIPS incorporated with the branch prediction unit reduces approximately 9% 10% cycles compares to that without the branch prediction unit. Table 6 presents the approximate area of each BPU. We also can observe that the area of MIPS with PHT2 is larger than the others since PHT2 has 2 additional registers, one for the history state, and the other for the pattern history table.

TABLE IV
Execution Cycles under I_mem_hasHazard

I_mem_hasHazard	
BPU	**Execution Cycles**
no	2144.5
1-bit	1954.5
2-bit	1939.5
PHT2	1942.5

TABLE V
Execution Cycles under I_mem_BrPred

I_mem_BrPred	
BPU	**Execution Cycles**
no	155.5
1-bit	141.5
2-bit	138.5
PHT2	138.5

TABLE VI
Area of BPUs under set cycle = 3.5 (ns)

Area	
BPU	**Total Cell Area(μm^2)**
1-bit	285400
2-bit	285433
PHT2	286086

VI. Conclusion

From the above experimental results and taking the area into consideration, we can sum up that 2-bit predictor is the optimal choice under the given test bench. As the professor mentioned in class, we conclude that ”Simplicity favors complexity”.

VII. Cache

L2 cache is used as the transition buffer between L1 cache and the main memory. In order to shorten the miss penalty for read/write miss, we add a L2 cache into our circuit. First, we want to analyze the performance of different way numbers of L1 cache.

A. Different Way Numbers of L1 Cache

We compare the performance of L1 cache based on direct-mapped with 2-way set associative. After the experiment, we find out that L1 cache has better performance with 2-way set associative cache which is shown in Fig. 12. We can observe that two lines representing cache with 2-way set associative have lower simulation time compared to the direct-mapped, especially for L1 instruction cache with 32 words (black solid line at the bottom depicted in Fig. 13), which has smaller miss rate compared to direct-map. However, we also discover that L1 instruction cache with 16 words has greater miss rate with 2-way set associative compared to direct-map (red solid line and blue solid line depicted in Fig. 13). We think that this phenomenon is due to the size of the function block in the instruction sets. When we use L1 instruction cache with 32 words which fully covers all of the instructions in certain function blocks, we will obtain a smaller miss rate. On the other hand, when we use L1 instruction cache with 16 words which is small in size, it will only cover a portion of the entire instructions in certain function blocks, while adding 2-way set associative will decrease the total number of entry for the L1 cache. Therefore, this phenomenon will increase the miss rate when comparing to the L1 cache with direct-mapped. In conclusion, we find out that L1 cache with larger size and multiple way numbers set associative has better performance. For the remaining test, we use 2-way L1 cache with different settings.

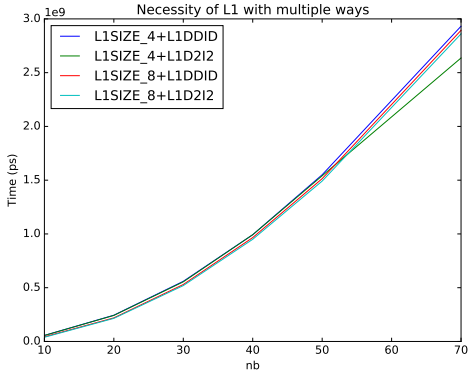

Fig. 12. Necessity of L1 with Different Way Numbers

B. Existence of L2 cache

We find out that after adding L2 instruction cache into our circuit, we will have a better performance for fetching instruction data. The performance of processor only embedded with a L1 cache (solid blue line shown in Fig. 14) is worse than adding a L2 instruction cache with any size when not dealing with too much data (nb¡50). However, while the amount of data increases, the effect and benefit of adding L2 instruction cache

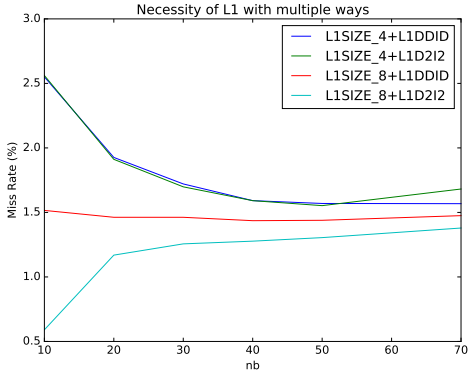

Fig. 13. Necessity of L1 with Different Way Numbers

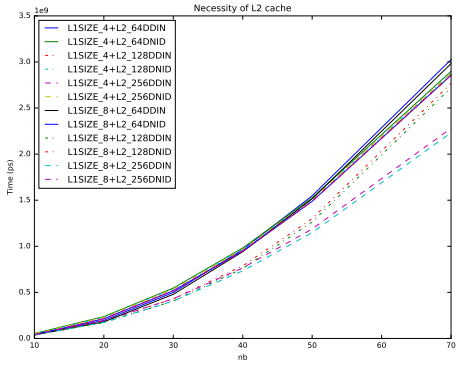

Fig. 14. Necessity of L2 Cache

is overwhelmed by too many data cache miss when dealing with sequential data. Since the total number of instructions for our test bench is 76, when using L1 cache with size of 32 words which almost covers half of the instruction data set, whole function block in assembly code. Thus, adding L2 instruction cache will not enhance the performance (solid purple line is stuck with other lines shown in Fig. 14). On the other hand, while using L2 data cache with 256 words which covers the entire necessary data will achieve a better performance than not using L2 data cache. The performance after adding L2 data cache with size of 64 words is worse than that with no L2 data cache (dotted line in Fig. 15). We think that the non-ideal result is due to the characteristic of our test bench which computes and stores the Fibonacci series first then performs bubble sorting on the Fibonacci series which is generated by processor. The whole operations are sequential and the size of L2 cache is not large enough to contain all the necessary data. Therefore, we will not obtain any benefit from L2 cache where data needs to be used at least twice to utilize the effect of L2 cache. What's worse, we have to wait 1 additional cycle for L2 cache to determine what operation is needed to perform next.

C. L2 Cache with Different Way Number Set Associative

We fix the size of L1 and L2 cache and then compare the performance of multiple way number set associative. We discover that the performance of multiple way number set associative of L2 data cache doesn't have obvious enhancement on performance (blue solid line in Fig 16). However, we can still observe that the L2 data cache with 2-way number set associative (green solid line in Fig. 16) is best for almost all test cases. As we have already mentioned, the test bench for our experiment is composed of operations of fetching and manipulating sequential data, thus, if L2 data cache does not contain the entire necessary data which will be used in operations, data will only come in and out of L2

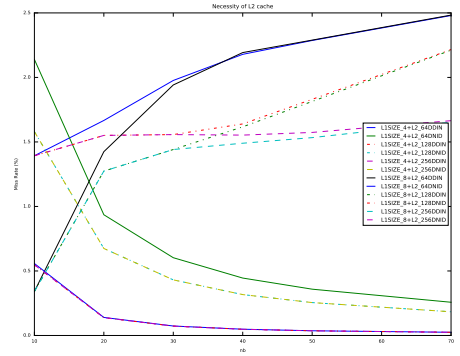

Fig. 15. Necessity of L2 Cache

cache without utilize the effect of L2 data cache. On the other hand, we obtain better performance on L2 instruction cache even with the size is smaller than the number of total instructions. Since the instructions do not operate sequentially, there are jumps within our test bench, we can utilize the effect of multiple way number set associative on L2 instruction cache (Fig. 17).

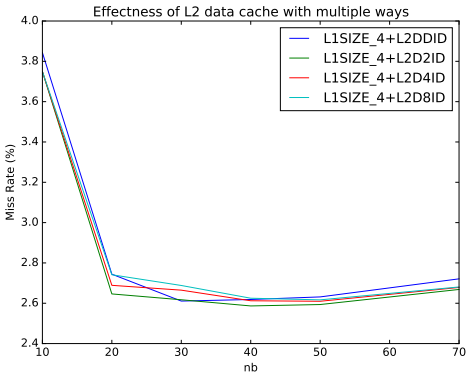

Fig. 16. Effectiveness of L2 Data Cache with Different Way Numbers

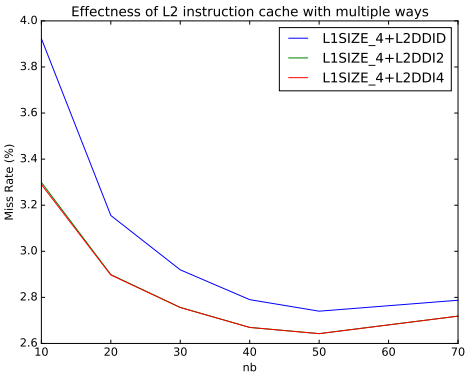

Fig. 17. Effectiveness of L2 Instruction Cache with Different Way Numbers

D. L1 and L2 Cache with Different Size Ratios

We discover that L1 and L2 cache with different size ratios does not really matter for our data cache because of the sequential property we mentioned in the previous sections for our test bench. The key point for determining the performance is based on the size of the L2 cache. With larger size of cache, we have better performance. (Fig. 18 and Fig. 19)

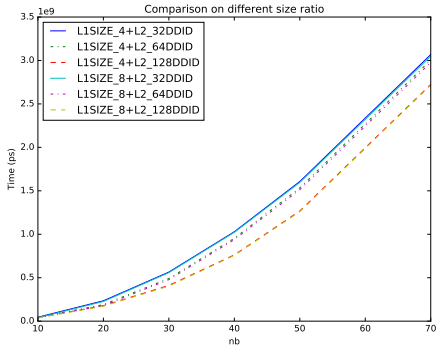

Fig. 18. Comparison on Different Size Ratios

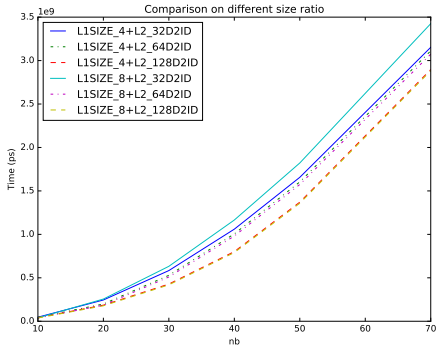

Fig. 19. Comparison on Different Size Ratios