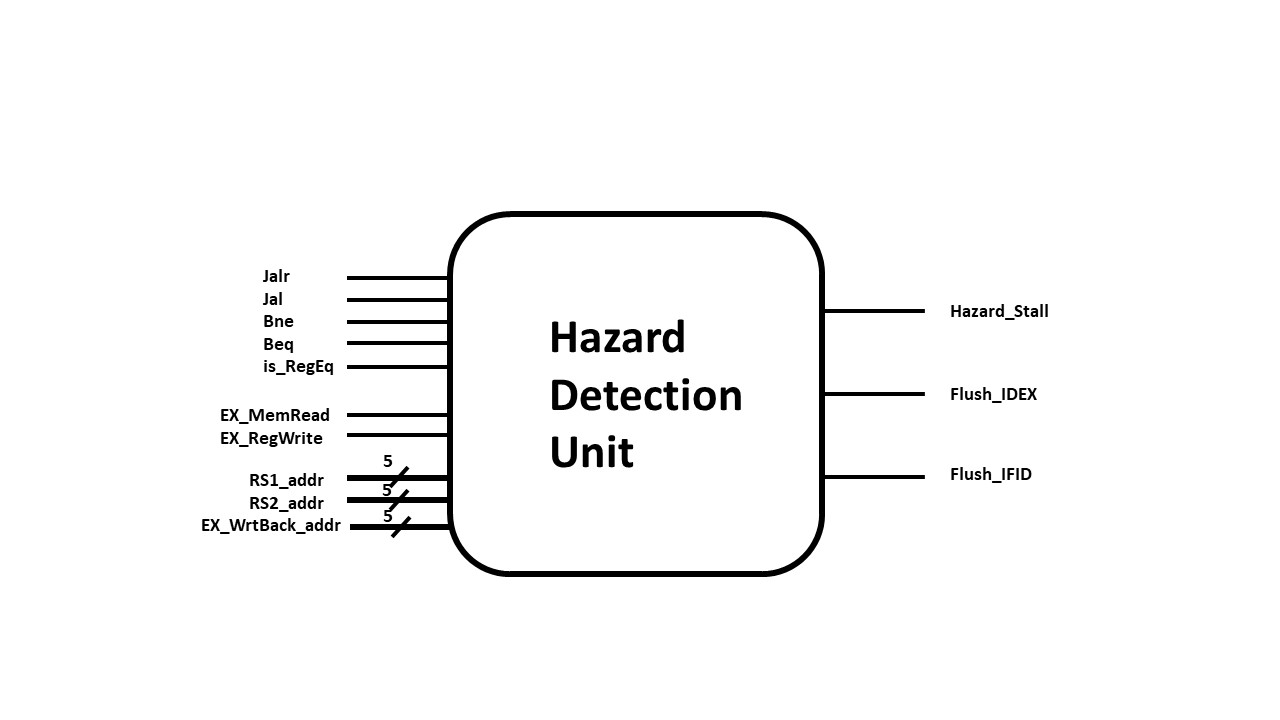
**1082 Digital System Design Final Project Report**

**Pipelined RISC-V Design**

**B06901149 電機三 周光照**

1. **Baseline Implementation**

* **Hazard detection unit:**

**Input:** Jalr, Jal, Bne, Beq, isRegEq, EX\_MemRead, EX\_RegWrite, RS1\_addr, RS2\_addr, EX\_WrtBack\_addr.

**Output:** Hazard\_Stall, Flush\_IDEX, Flush\_IFID.

**Features:**

According to our knowledge in this course, the piplelined structure inherently possesses the load-use hazard and the branch hazard. These issues were covered by a hazard detection unit. During the process of trial and error, we also discovered some potential hazards which will cause malfunctions under pipelined implementation. For instance, since there is the jalr instruction, we need data in the register block in ID stage to address the next PC. However, when the instruction in EX stage will write the new value back to the register block later, and meanwhile it has the same Rd address as the Rs1 address or the Rs2 address in ID stage, the updated data cannot arrive in time for the request of jalr (beq and bne also have this probability). This could be treated in the same manner as we overcome the load-use hazard. The goal of this block unit is to solve these hazards which may not simply be solved by the forwarding unit.

Additionally, when the instruction in ID stage needs data in the register that has the same address as the write-back Rd address in WB stage, it will also cause a hazard. Initially, we want to stall a cycle to separate orders of the read operation and the write operation, which is the method we use for the load-use hazard and others. Nevertheless, it will make a problem as we realize the branch prediction unit, which will be discussed later in Extension part. Therefore, we eventually discarded this idea.

**Design Methodology:**

The output Flush\_IFID depends on whether the instruction in ID stage is branch or jump. If so, we flush IF/ID register block in order to delete the wrong instruction due to delayed arrival of the correct PC. In case of load-use hazard and similar hazards coming from branch or jump, we compare EX\_WrtBack\_addr with RS1\_addr or RS2\_addr (RS1\_addr only when the hazard comes from jalr), and check whether later the instruction in EX stage will really store data back to the register. Then we stall PC and IF/ID register block in addition to flushing the ID/EX register block. Furthermore, we also add the condition that EX\_WrtBack\_addr must not be zero to prevent unnecessary stall because x0 cannot be rewritten.

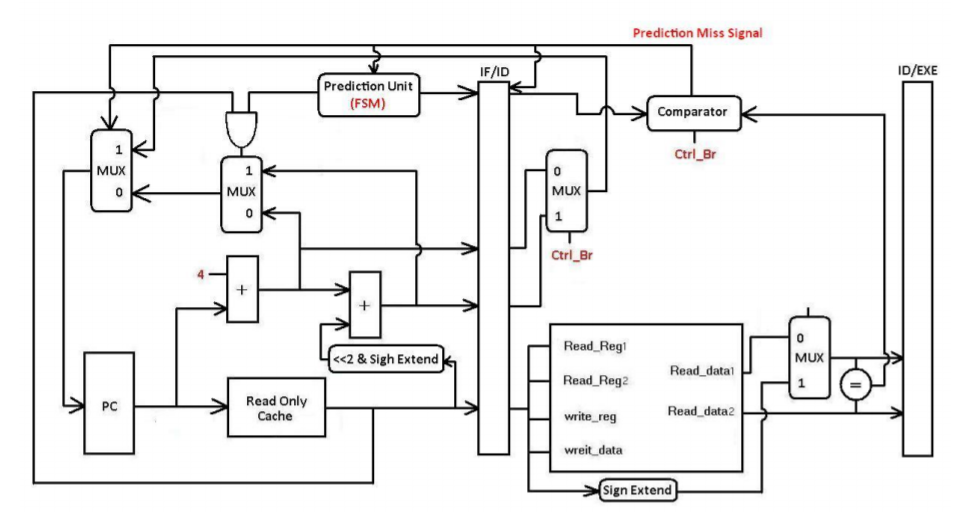
1. **Extension Exploration**

* **Branch prediction mechanism**

**Preface:**

As learned from the course, there are always-taken, always-not-taken, 1-bit prediction, and 2-bit prediction mechanisms that could be used to implement the branch prediction function. We shall not explain them here in detail again. We will focus on the comparisons among these types of methods to see their performance and discuss the results so as to discover the hidden problems we have faced with. Finally, we will introduce some modified mechanisms to achieve better efficiency according to the given test patterns.

**Structure:**

The structure of the BPU is just similar with the structure given in the supplement material with some adjustment. We will append the figure as follows to have a brief look, and we will not focus on it to proceed to more important issues as soon as possible.

**Obstacle:**

1. **Write-read hazard problem:**

Originally, we stall a cycle when there is a confliction between write-back address in WB stage and read address in ID stage. However, this will generate a problem when using hasHazard test pattern to run the program with BPU. This is due to a sequence of instructions in the bubble sort:

// BubbleInLoop : lw x13 x11 0x000

// lw x14 x11 0x004

. . .

. . .

. . .

// SwapExit : addi x11 x11 0x004

// bne x11 x12 BubbleInLoop

The execution orders starting from SwapExit would be: addi → nop (see hazard detection unit) → bne → lw (stall a cycle due to write-read hazard) → lw → … (when branch is true and BPU makes a correct prediction). Then register x11 will have a write-read hazard. This will make the CHIP with BPU runs slower than the CHIP without BPU because we almost always need to stall an extra cycle every time we enter the loop even though BPU has made a right prediction. To solve this problem, we modify our baseline part to forward the write-back data back to the output port of the register-read. As a result, we avoid stalling a cycle and the CHIP with BPU can finally have a chance to achieve the better performance than the CHIP without BPU.

1. **Branch-taken sequence problem:**

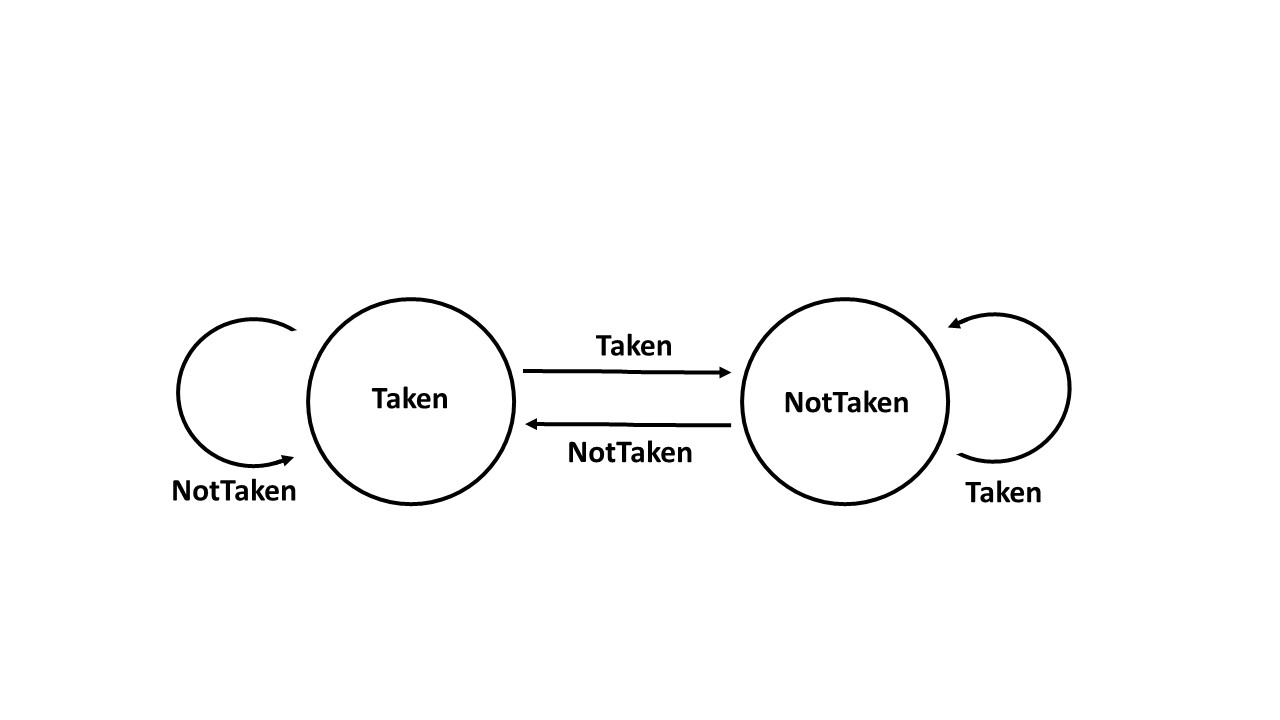
In the bubble sort of hasHazard test pattern, a large part of the branch sequence is TAKEN, NOTTAKEN, TAKEN, NOTTAKEN, TAKEN, . . . repeatedly. This is totally against the idea when we design the state transitions of 1-bit predictor and 2-bit predictor. As we can see, 1-bit predictor and 2-bit predictor just have worse performance than the baseline CHIP.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Without BPU | Always-not-taken | Always-taken | 1-bit predictor | 2-bit predictor |
| The total execution time(ns) | 20595 | 20595 | 20485 | 21505 | 20645 |

\* Cycle time = 10ns.

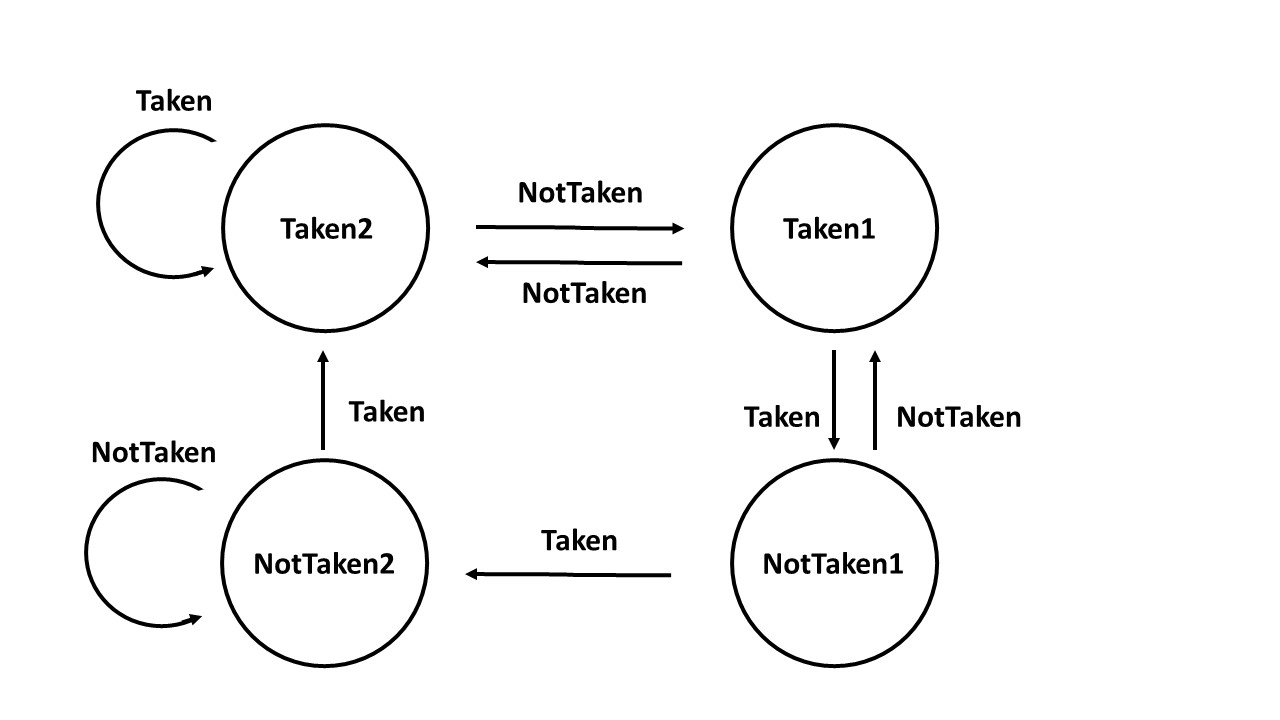
\* The execution times of Without BPU and Always-not-taken are equal, this is because each time a branch instruction comes, both of them will stall a cycle if the branch condition is true and continue if the branch condition is false.

To solve this problem, we introduce two new state transition mechanisms. Although they are far from intuition, they are quite suitable for hasHazard test pattern.

1. **Reverse 1-bit predictor**
2. **Features:**

It is just a simple idea that reverses the transition of the original 1-bit state graph. If the branch pattern meets the regularity described above, this predictor can well-predicted the branch control behavior. However, there still exist for loops in the test pattern, which is beyond the prediction ability of this type of predictor. Even so, this predictor still achieves outstanding performance when using hasHazard test pattern.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Without BPU | Always-not-taken | Always-taken | 1-bit predictor | 2-bit predictor | Reverse 1-bit predictor |
| The total execution time(ns) | 20595 | 20595 | 20485 | 21505 | 20645 | 19565 |

1. **2-bit predictor v2.0**
2. **Features:**

This state transition is the upgrade of the reverse 1-bit predictor. The initial state is Taken2. The right-handed side of the graph (Taken1 and NotTaken1) is the way the reverse 1-bit predictor changes its state. However, we know the test pattern in hasHazard has for loops that will always take the branch behavior, so we need a mechanism for the state to change back to Taken2 if the predictor senses continual branch-taken pattern. After that, it can stay at Taken2 until there is a NotTaken pattern. This is the reason why NotTaken2 points back to Taken2 when Taken is true. This state graph can achieve really high accuracy specific to this testing data.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Without BPU | Always-not-taken | Always-taken | 1-bit predictor | 2-bit predictor | Reverse 1-bit predictor | 2-bit predictor v2.0 |
| The total execution time(ns) | 20595 | 20595 | 20485 | 21505 | 20645 | 19565 | 19475 |

Finally, we give the following table of accuracy rates of each type of predictor (just approximation).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Always-not-taken | Always-taken | 1-bit predictor | 2-bit predictor | Reverse 1-bit predictor | 2-bit predictor v2.0 |
| The accuracy rate (using hasHazard pattern) | 41.4% | 60% | 10% | 41% | 83% | 87% |

**Exploration of changing test pattern:**

From now on, we know the concepts of each type of branch predictors and their pros and cons relative to the test pattern. Now we use several BrPred test patterns by changing parameters a, b, and c to see whether the results justify our concepts.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | The execution time(ns) | | | | | |
| Always-not-taken | Always-taken | 1-bit predictor | 2-bit predictor | Reverse 1-bit predictor | 2-bit predictor v2.0 |
| a = 5  b = 20  c = 35 | 3735 | 3445 | 3605 | 3635 | 3765 | 3445 |
| a = 10  b = 20  c = 30 | 3785 | 3595 | 3705 | 3735 | 3865 | 3605 |
| a = 25  b = 20  c = 15 | 3935 | 4045 | 4005 | 4035 | 4165 | 4045 |
| a = 13  b = 8  c = 39 | 3455 | 3205 | 3165 | 3195 | 3565 | 3205 |
| a = 5  b = 40  c = 15 | 4335 | 4245 | 4605 | 4635 | 4365 | 4245 |
| a = 2  b = 52  c = 6 | 4665 | 4635 | 5145 | 4645 | 4665 | 4645 |

We can conclude that when there is more proportion of always-branch patterns, always-taken will be better than always-not-taken. Furthermore, 2-bit v2.0 has bad performance when there are more never-branch patterns. This is due to the fact that never-branch patterns are executed first, but we have our initial state at Taken2, which the predictor cannot break away from two Taken states unless interleaving patterns come. Therefore, it will always predict wrongly until the interleaving patterns arrive.

If we increase the proportion of interleaving patterns, 1-bit predictor and 2-bit predictor will perform poorly, justifying our concept that both of the predictor cannot well-predict the interleaving branch behavior. But reverse 1-bit and 2-bit v2.0 can maintain comparatively good performance. The reason that reverse 1-bit and 2-bit v2.0 cannot outperform always-taken and always-not-taken is due to the fact that there are two continuous branch patterns in the interleaving loop. When the first branch pattern is checked in ID stage, the second branch pattern is already in IF stage, so the second pattern cannot use the new next state to predictor its behavior since the new next state needs to wait for the next cycle to be updated.

Finally, by changing the number of interleaving patterns, we see that 2-bit v2.0 is more stable than reverse 1-bit as long as there are not too many never-branch patterns.

This is reasonable as we use the more complicated mechanism in 2-bit v2.0.

\* The execution times shown above do not use the most updated version of the chip. We just use them to compare the performance among different predictors.

**Summary:**

In short, we have described several problems we met during the implementation of the BPU, and suggested some potential solutions. We also discuss the relationship between test patterns and the predictor. However, we found that there is hard to create a global solution that is suitable for all patterns we met. How practical these solutions are depends on the probabilities of patterns happened in reality.