CIS501: Branch Prediction

Shreyas S. Shivakumar

December 2, 2019

1 Lecture Notes

1. **Control Dependencies**

   We haven't talked about **control hazards** yet, i.e when we have some control flow instruction coming through the pipeline that dictates what instruction should follow it. It can be thought of as a dependency through the **Program Counter (PC)** since the **PC** dictates the order of instruction flow.

   Branch Prediction is how we will attempt to solve this problem. Branch Prediction cannot however fix all cases of failure related to control hazards.

2. **What are the challenges with Branch instructions?**

   - First, when an instruction is coming down the pipeline, we first need to figure out whether it is a **BR** instruction or not. This doesn't happen during the *Fetch Stage* but happens during the *Decode Stage*.
   - Second, we need to figure out which way the **BR** instruction should go, i.e should the processor move to the next sequential instruction or should it jump to the *target*. This usually involves some computation to figure out what the *target* address is, and thus involves the **ALU**. So this starts to illustrate that some of the information that we need is computed later into the pipeline.

     This gap between *when I need the information* and *when I have it* is the primary problem here and indicates that there will probably be some **stalling** involved.

- Lastly, building upon the first challenge - when the processor is handling a Branch instruction, it is usually Fetched and then moves onto the *Decode Stage*. But what should we be doing in the *Fetch Stage* next? Logically, we will have to fetch the next instruction in the sequence. But if the previous instruction was indeed a **Branch** instruction, we should be going to the *target* in the next cycle instead of the next sequential instruction. This is another fundamental problem in handling **Branch** instructions. *We don't yet know what the right thing to do is, but we have to do something!*. We can *not do anything* but that would then indicate a 3 cycle stall/gap, which isn't good for performance.

3. How do we then fix this problem?

We can just **guess**. In our previous approach of handling *data hazards*, we would try to identify a problem early on and make arrangements to avoid that problem or minimize it's effects on the system. However, with *control hazards*, we take a different approach because **waiting/stalling** is too expensive. We instead **guess**, and hope to make a **smart guess**. This does not guarantee that we are always correct, so when we are wrong we must ensure there are mechanisms in place to clean up behind us. This is the idea behind **Speculative Execution**.

4. Speculative Execution

- Execute before all parameters are known with certainty
- If your speculation is *correct*, you have avoided stalls and improved performance significantly
- If your speculation is *incorrect* (mis-speculation), you have to fix what you broke by flushing all incorrect instructions and undo all incorrect changes, going back to the original state

5. Control Speculation

This is the first type of speculation we will introduce. When you have a control flow instruction, you don't know where it is going, so the speculation is made on the control flow path.

- We want to guess whether it is a **Branch Instruction** to begin with.
- If it is a *Branch Instruction*, we want to guess whether it is **taken**.
- If it is *taken*, we want to guess what it's **target** is.

This ultimately results in guessing *where should I go next* or more simply *what is the next **PC** that I should be going to?*

According to this, what we've been doing so far is also technically *speculation*, except we guessed that the next **PC** would always be **PC + 4**.

6. At what stage should we perform Branch Prediction?

Ideally, the later we do it, the more accurate our prediction will be since we will have more and more information available to us as we go down the pipeline. But this is not necessarily good for performance since it will ensure that some cycles are lost in case the branch was actually taken.

1. During Decode:

Pros:

If we do it after the decoding operation, we have a lot more information - we know if it is indeed a *Branch Instruction* or a *Jump Instruction* etc. This will help quite a bit for example if it is a *Jump Instruction*, we know that it is always taken and we can update our PC accordingly. Even if it is a simple *Add Instruction*, it is useful because then we don't need to make any exotic guesses and we can just update the PC to the next sequential address.

Cons:

Even at it's best, this will introduce a one cycle penalty.

Cycle	1	2	3	4	5	6	7	8	9
BNEZ R3, target	F	D	X	M	W	-	-	-	-
target: ADD R5, R4 $\rightarrow$ R4	-	?	F	D	X	M	W	-	-

2. During Fetch:

How would this work?

Realistically, we don't really know anything about the instruction during the *Fetch Stage*. But we will design mechanisms that will make this possible.

7. Branch Recovery

How do we handle recovering from when a *Branch Instruction* is actually **taken**, i.e we figured this out in the *Execute Stage*? The instructions that are in the *Fetch Stage* and the *Decode Stage* are now **wrong**. These instructions need to be *flushed*; this can be done by replacing them with **no-ops**. Not too much damage has been done since these instructions haven't written anything to permanent state elements (register files, memory) yet. The *Program Counter* has however changed, but this is easier to fix. This exercise does introduce a **two cycle penalty** for any *Branch Instruction* that is **taken**.

For *example:* in **Lab 4** where we implement a Pipelined LC4 processor without *Branch Instructions*, we always predict PC = PC + 1, but when this is wrong, we need to perform branch recovery and introduce the respective stall cycles. This is done by just adding a MUX that can switch between a *no-op* signal and the regular instruction.

Speculation:

Cycle	1	2	3	4	5	6	7	8	9
ADDI R1, 1 → R3	F	D	X	M	W	-	-	-	-
BNEZ R3, targ	-	F	D	$\underline{\textbf{X}}$	M	W	-	-	-
STORE R6 → [R7, 4]	-	-	*F*	*D*	*X*	*M*	*W*	-	-
MUL R8, R9 → R10	-	-	-	*F*	*D*	*X*	*M*	*W*	-

We realize where we are actually heading in *Cycle 4*!

Recovery:

Cycle	1	2	3	4	5	6	7	8	9
ADDI R1, 1 → R3	F	D	X	M	W	-	-	-	-
BNEZ R3, targ	-	F	D	$\underline{\textbf{X}}$	M	W	-	-	-
~~STORE R6 → [R7, 4]~~	-	-	~~*F*~~	~~*D*~~	–	–	–	-	-
~~MUL R8, R9 → R10~~	-	-	-	~~*F*~~	–	–	–	–	-
targ: ADD R4, R5 → R4	-	-	-	-	F	D	X	M	W

8. **Branch Performance**

   Instructions are Branch: 20%, Load: 20%, Store: 10%, Other: 50%. 75% of *Branch Instructions* are taken. What is the CPI?

$$CPI = 1 + 20\% \times 75\% \times \mathbf{2}$$
$$CPI = 1 + 0.20 \times 0.75 \times \mathbf{2} = 1.3$$

   The **2** is because of the **2 cycle penalty** associated with branch mis-prediction.

   In this case, *Branch Instructions* will cause a 30% slow-down. We should be able to improve this if we do something a little less naïve than just assuming the *Branch* is **not taken** each time.

9. **Dynamic Branch Prediction**

   Here the hardware **guesses** the outcome and starts to *Fetch* from the guessed address. If this results in a mis-prediction, we can *flush* instructions as seen above. This hardware exists in the *Fetch Stage*, sitting concurrently with the *Program Counter* register and it's incrementing hardware.

   Dynamic Branch Prediction Performance:

   Instructions are Branch: 20%, Load: 20%, Store: 10%, Other: 50%. 75% of *Branch Instructions* are taken. **Branch targets are predicted with 95% accuracy**. What is the CPI?

$$CPI = 1 + 20\% \times 5\% \times \mathbf{2}$$
$$CPI = 1 + 0.20 \times 0.05 \times \mathbf{2} = 1.02$$

   In this case, *Branch Instructions* will cause a 2% slow-down. This is significantly better than the previous naïve approach.

10. **Dynamic Branch Prediction Components**

    (a) Is it a **branch**?

    (b) Is the branch **taken** or **not taken**?

    (c) If it is taken, **where** does it go?

We need to be able to answer all these three questions *speculatively*, and early on in the pipeline.

A structure called the **Branch Target Buffer (BTB)** will be used to answer part (a) and (c) of our problem above.

A **Direction Predictor** structure will be used to answer (b). Historically, part (b) is the most challenging part of this problem and is most often associated with literature on *dynamic branch prediction.*

Things to keep in mind while navigating through the mess that is *branch prediction*:

- Which instruction are we trying to make predictions about? The instruction **ahead** of us in the pipeline; and this is recursively applied.
- Our notion of PC must now account for this since there are 5 instructions in the pipeline and they all have *Program Counters.*

11. Branch Target Prediction

This will help us answer **(c) If a branch is taken, where does it go?**

Gist: *Learn from the past, predict the future.* Disclaimer: Learn is a bit too strong a word, and *memorize* is more in line with what actually occurs.

Branch Target Buffer (BTB):

- Working: *"The last time that Branch X was taken, it went to address Y; so the next time if address X is fetched, the next address to be fetched is address Y."*
- The BTB can be thought of as a *hash table / table* that uses a fixed number of bits from a *Program Counter* to index **target addresses**. For *example:*, if bits 2...9 (7 bits) are used as index bits, we can keep track of 2^7 target addresses. This also hints at why we don't use the entire PC as an index to the BTB - if you had 32-bits, that would result in 2^{32} target address - which would result in extremely high accuracy / precision of prediction, but require an unreasonable amount of hardware storage. Using 7-bits might not result in extremely high accuracy but this is ok since these are *predictions* anyway.

- *Aliasing* - two PCs with the same 7-bits are also possible, but this conflict is fine since we're just doing our best at making predictions anyway and we have no guarantees of perfect performance.

The *Branch Target Buffer (BTB)* does exist on the critical path and therefore adds a bit of extra overhead to the overall system.

With *static targets*, the BTB mechanism will perform well.

12. Branch Target Buffer Example

These screenshots were taken from the *http://comparchviz.com* tool.

The following *example* (Figure 1 - Figure 7) assumes a 1-bit *Branch Target Buffer* index, which means it has a capacity to learn 2^1 target addresses and can be indexed by PC index bits 00 and 01.

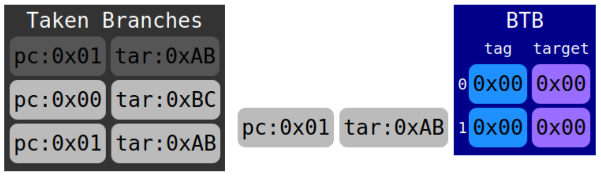

Figure 1: Branch Target Buffer is empty.

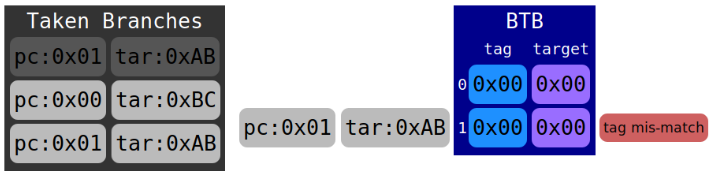

Figure 2: Tag-mismatch since the BTB was previously empty with tag bits 00.

13. What bits should you use as index bits and why do we discard the 2-bit least-significant bits?

The last 2 bits can be ignored because with MIPS, they are 32-bit instructions (PC is 32 bits) that are 4-bytes in size. You can throw away 2 bits because

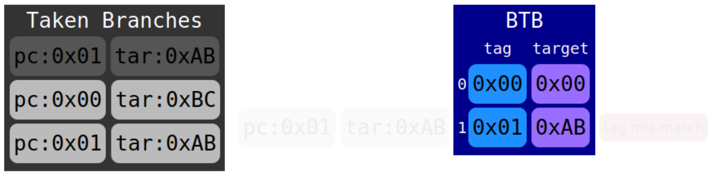

Figure 3: BTB is updated with previous *tag* and *target*.

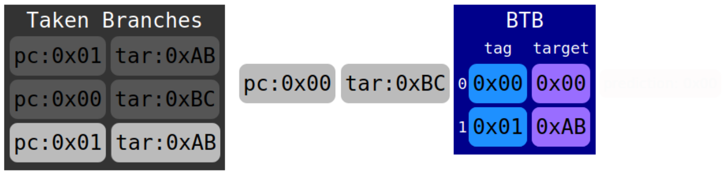

Figure 4: Next instruction arrives.

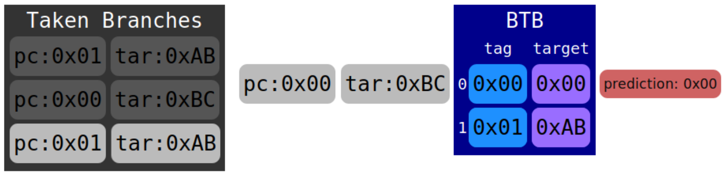

Figure 5: Mis-prediction because BTB is empty. Target is updated after this cycle.

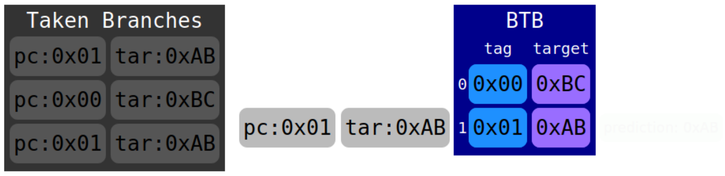

Figure 6: Next instruction arrives.

they will be the same (00) for all instructions since the instructions will all be incremented 4 bytes at a time. If you used these bits as BTB index bits, you would reduce your BTB target capacity implicitly.

We select lower order 7-bits instead of higher order 7-bits because the lower

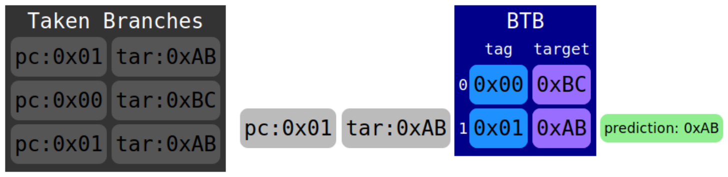

Figure 7: Correct branch prediction is made.

order bits change a lot more than the higher order bits during a program. This can also be thought of as picking bits that have the highest *entropy* and therefore can store the largest amount of information.

Is this the best way to do this hashing? No, there are definitely better and more complicated ways to design a hashing function that better utilizes it's has indices. But this is a good start.

14. What is the purpose of the TAG entry in the Branch Target Buffer (BTB)?

To understand the purpose of the **tag** entries, imagine a scenario where we did not have the *tag* entries. While running a program with many instructions, a situation will arise where the *Branch Target Buffer (BTB)* has been completely filled with entries (it is **saturated**). What this means for the following instructions is that every incoming instruction is treated as a taken branch and is mapped to a *target*, however this is definitely not the case.

To avoid this situation, an additional **tag field** is introduced which associates a given *Branch Target Buffer (BTB)* entry with a particular *Program Counter (PC)*, which is used to tag that particular branching instruction.

During the actual ***BTB*** **update**, this will look like:

$$BTB[PC].tag = PC$$
$$BTB[PC].target = \text{target of branch}$$

And at the ***BTB*** **prediction step** in the *Fetch Stage*, this will look like:

$$Predicted\ PC \ = (BTB[PC].tag == PC) \ ? \ BTB[PC].target \ : \ PC + 4$$

15. **Direct Targets and Indirect Targets**

(a) Why does this simple Branch Target Buffer (BTB) work?

Most control instructions use **direct targets**, i.e they branch to the same targets each time (*example:* JMP instructions). And the *BTB* does this really well.

(b) What if this isn't the case?

There are plenty of cases where control instructions use **indirect targets**:

- When you are branching to a value in a *Register*. There is no way to know this in the *Fetch Stage*.
- Dynamically Linked Functions (DLLs) - here the control flow will usually jump to a table in memory to find out where to go next.
- Virtual Functions (C++) - the control flow will usually branch to a location depending on the data type provided to the function (for *example*).
- Switch statements - difficult to handle but relatively uncommon
- Function returns - challenging because of the need to return to the caller; this varies heavily. This is difficult but we can use additional hardware called a **Return Address Stack (RAS)** to handle this.

16. **Return Address Stack (RAS)**

The mechanism is a fairly straightforward stack operation. During a function call the next PC, *(PC+4)*, is placed on top of the stack. A return instruction's target prediction is done by popping an element from the stack.

(a) How large is this stack?

It has a fixed capacity and hence will not work well for large amounts of instructions. But we can throw away the oldest instructions to ensure a relatively good accuracy for recent instructions. At the end of the day these are *predictions* anyway, so if it is *incorrect* for old instructions, it is alright.

(b) How do we know when to use the RAS?

There is usually another level of prediction that will associate a *Program Counter (PC)* with whether the **Return Address Stack (RAS)** is to be used or not. This will involve additional bits and hardware.

17. Branch Direction Prediction

We use a similar strategy from *branch target prediction* where we want to *learn from the past and predict the future*. Instead of learning a mapping from *Program Counter (PC)* to *Target*, here we learn a mapping from *Program Counter (PC)* to a single *Taken or Not Taken* bit.

Individual conditional branches are often very **biased**. Usually 90% branch one way or the other.

18. Bimodal Branch Predictor

This is the simplest form of *branch direction prediction*, which is similar in design to the *Branch Target Buffer (BTB)* except the **Branch History Table (BHT)** entry is a single bit for **Taken** or **Not Taken**. The following screenshots were taken from the *http://comparchviz.com* tool.

The following *example* assumes a **Branch History Table (BHT)** with **2** index bits. The **BHT** is initialized with all **Not Taken (N)**. The **3** *Program Counters (PCs)* in the system are 12, 23 and 12.

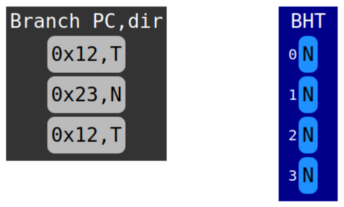

Figure 8: The Branch History Table (BHT) is initially set to all *Not Taken*

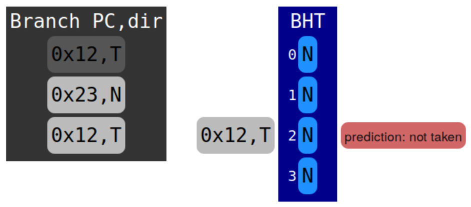

Figure 9: An instruction arrives but a false prediction is made

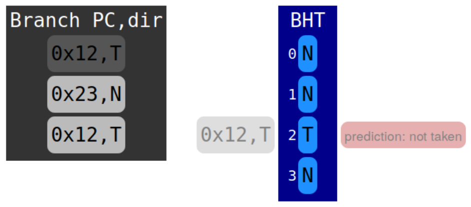

Figure 10: The table is then updated

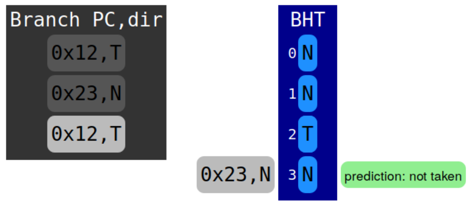

Figure 11: Another instruction arrives and a correct prediction is made

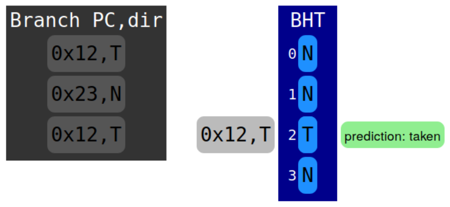

Figure 12: The first instruction arrives again and a correction prediction is made

19. Two-Bit Saturating Counter (2BC)

Failure of Bimodal Branch Predictor:

An obvious situation in which the bimodal branch predictor will fail is in nested loops such as the following:

```
for (i=0;i<100;i++)
    for (j=0;j<3;j++)
        ...
```

In the above situation, a bi-modal branch predictor will continuously switch between states and result in a low prediction accuracy.

The previous *Branch History Table (BHT)* based approach can be thought of as **changing it's mind too quickly**. We can introduce a level of **resistance** and force the predictor to *mis-predict twice before changing it's mind.* This is done by using a **2-bit saturating counter** that will essentially model a four-state finite state machine. The additional bits provide a larger memory.

$$(0, 1, 2, 3) = (N, n, t, T)$$

The state-transition $(state, prediction) \rightarrow (state)$ function can be thought of as follows:

$$\begin{gathered}(N, NT) \rightarrow N, \quad (T, NT) \rightarrow t \\ (N, T) \rightarrow n, \quad (T, T) \rightarrow T \\ (n, NT) \rightarrow N, \quad (t, NT) \rightarrow n \\ (n, T) \rightarrow t, \quad (t, T) \rightarrow T\end{gathered}$$

20. GShare History-Based Predictor

While the last two methods of *Bimodal Branch Predictor 1-bit* and *2-bit Saturating Counters* did a reasonable job at learning previous history, there are conditions such as below where learning history is insufficient. We need to instead learn to identify recurring patterns of instructions.

In the *example* below, the modulo three conditional branch is repeated twice as seen in *(i) and (ii)*, but they are separated by some unrelated, unpredictable set of instructions. In this case, the above two methods would not be able to effectively predict the right branch targets even though it has previously seen the same instructions.

```
for(i=0;i<10000;i++)
{
  if(i%3==0) {        — (i)
  ...
  }
  ...
  ... unrelated sequence ...
  ...
  if(i%3==0) {        — (ii)
  ...
  }
}
```

GShare maintains a **Branch History Register (BHR)** in addition to a **Branch History Table (BHT)**, which typically will use **2-bit Saturating Counters**.

How do we incorporate this idea of history and correlation into our predictions?

Instead of using just the *Program Counter (PC)* as the index into the *Branch History Table (BHT)*, use PC $\oplus$ BHR (xor operation) as the index to the BHT. This will result in more unique *keys / indexes* into the table.

We need to train the counter (BHT) and the BHR (*Branch History Register*) after each branch instruction.

21. GShare Example

Go to *http://comparchviz.com.*

22. Hybrid Tournament Predictor

A simple *bimodal branch predictor* is good for short term history dependent branches and is fast. Most static target branch prediction can be accomplished by this method. A *gshare correlated branch predictor* is good for branch prediction that requires pattern identification and correlated branches over time. *Can we combine the two?*

A **Hybrid Predictor** combines the above mentioned predictors and introduces a third component called the **chooser** which learns to predict which of the above two methods to use. This allows the correlated predictor to become smaller than previously defined and results in 90-95% accuracy.

23. Reducing Branch Penalty

Even a taken branch results in a **2 cycle** branch penalty. Can we decide a faster branching strategy that can decide at the *Decode Stage* and not the *Execute Stage* on instructions that require some test conditions?

What if we evaluate test conditions that are checking for *equality* or a *comparison* and we do this in the *Decode Stage*? This would reduce the branch penalty to **1 cycle** in these cases.

Performance:

Let's say that Branches: 20% and 75% of branches are taken. Out of these branches, 25% cannot be optimized with the method above and require an additional cycle.

$$CPI = 1 + \text{fast branch instructions} + \text{regular instructions}$$

$$CPI = 1 + (0.20 \times 0.75 \times 1) + (0.20 \times 0.25 \times 1) = 1.2$$

Branch Predictions are still a big problem even with all these optimizations. For typically large pipelines, mis-predictions cost the system 10+ cycles!

24. Predication

Instead of predicting which way we're going, why not go both ways?

Predication is an **alternate view** that attempts to get rid of branch prediction by rephrasing control flow instructions as conditional instructions. The condition is represented by a **predicate bit** and these instructions will either execute as normal or as *no-ops* depending on this predicate bit.

For control sequences such as an **if-then** statement, there will be no overhead and therefore there are gains in performance to be had.

However, in **if-then-else** statements, since there is only **one** useful path, this might not be very effective.

Predicated instructions are useful only if the number of cycles saved is more than the mis-prediction penalty if we were using regular branch predictors.

2 Textbook Notes

1. **Control Hazards**

   Control hazards occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding / bypassing is against data hazards.

2. **Fast Branching**

   Many branch instructions rely on simple tests (*equality or sign, for example*) and such tests do not require a full **ALU** and can be done independently with a few gates. For these instructions, we can reduce the cost by one cycle.

   When a more complex branch decision is required, a separate instruction that uses an ALU to perform a comparison is used.

3. **Drawback of 1-bit Bimodal Branch Predictor**

   Even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken.

   Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

   There will be a mis-prediction at the entry to this loop and one at the end, resulting in 80% accuracy even though the branch is taken 90% of the time. We should be able to do better than this? **Yes, use 2-bit saturating counter.**

4. **2-bit Saturating Counter Branch Predictor**

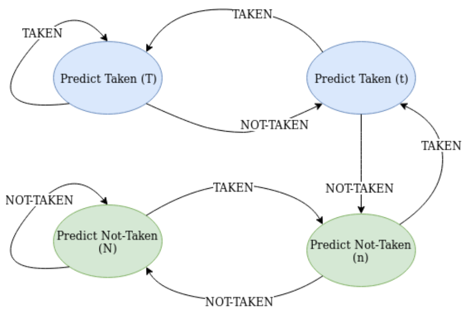

Figure 13: Finite State Machine for 2-bit Saturating Counter

3 Exam Questions

1. **List one source of slowdown that increases as pipeline depth increases**

   Branch mis-prediction penalty

2. **A bimodal branch predictor augmented with a branch history register is known as what kind of predictor?**

   GShare Branch Predictor

3. **What does BTB stand for?**

   Branch Target Buffer

4. **What are the three steps / questions involved with Branch Prediction?**

   (a) Is this instruction a branch?

   (b) If it is a branch, is it taken or not taken?

   (c) If it is taken, where is it going?

5. **Your pipelined processor in Lab 4 implements speculative branch prediction (true / false)?**

   True, because even just predicting $PC = PC + 1$ is a form of speculation.