Lesson 4

Branches

Branches account for 20% of all instructions, so mispredicting then can have a major impact on processor performance. Understanding and improving branch prediction will greatly enhance performance.

Branches in a Pipeline

If a branch is not taken the PC just advances to the next instruction.
If the branch is taken the immediate value is added to the PC to advance to the new destination.
It is not known until the end of the ALU stage whether the branch is taken or not. (ALU stage = Execute stage)
It is better to make a prediction, then the processor is at least right some of the time.

Branch Prediction Requirements

Branch prediction must be based on the available information, which is the PC. Using only the PC, the processor must try and correctly guess if the instruction is a taken branch and if so, what is the new PC.

CPI w/ ideal pipelining/perfect prediction

Mispredictions PER instruction = (what % of all instructions are branches) * (what % of branch instructions are mispredicted)

Branch Prediction Accuracy

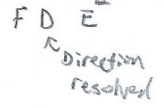
CPI = 1 + (Mispredictions / Instruc) * (Penalty / Misprediction)
Mispredictions/Instruction is dependent on the predictor accuracy
Penalty/Misprediction is dependent on the size of the pipeline

The deeper the pipeline the more important it is to have an accurate branch predictor. (b/c higher penalty)

Performance with Not-Taken Prediction

Refuse-to-predict waits to determine if the instruction is a branch
- Every branch costs 3 cycles → 2 cycles to determine it is a branch taken + 1 cycle to find out the target
- Non-branches cost 2 cycles → 2 cycles to determine it is not taken

A not-taken prediction just fetches the next instruction.
- Every branch costs 1 or 3 cycles (1 the branch is not taken, 3 if it is taken)
- Non-branches cost 1 cycle



correct prediction

incorrect prediction

Every processor will perform some type of prediction

Predict Not Taken

The Not-taken predictor is the simplest predictor, just increment the PC.
20% of all instructions are branches, 60% of branches are taken.
So Predict Not-Taken is correct 88% of the time and incorrect 12% of the time

(20%)(60%) = 12% of all instructions are "taken branches", so 100-12 = 88% of all instructions are NOT "taken branch" instructions

Why Do We Need Better Branch Prediction?

The larger the pipeline, the greater the penalty, especially when the branches are detected later in the pipeline and there are multiple instructions per cycle.

There is a lot of waste from a misprediction, especially if it is a deep and/or wide pipeline.

Better Prediction - How?

All that is known about the instruction is the PC. A better prediction would require more knowledge about the instruction. Since this is not possible another method of improving prediction is needed.
Better prediction can come from using the history of the instruction to make a better prediction.

Branch Target Buffer (BTB)

The Branch target buffer holds the target PC, it is indexed by PC number.
BTB Steps

1. At Fetch the processor has the PC of an instruction
2. Looks in the BTB for this PC number
3. The processor reads out the prediction for the next PC.
4. This predicted PC is then compared with the actual PC that is generated later in the pipeline. (execute stage or something)
5. If the BTB prediction and the actual PC are the same -- this is a correct prediction. If the two are not the same -- this is a misprediction, and the correct PC is stored in the BTB.

Realistic BTB

The BTB needs to have a 1 cycle latency, so it needs to be small.
The BTB needs to hold only the instructions that are likely to be executed soon.
Use the LSB for indexing to the BTB.

Direction Prediction

Use a Branch History Table (BHT) to first determine if the branch is taken or not taken.
Steps for BHT

1. Use the LSB of the PC to index the BHT.
2. The simplest BHT will have 1 bit.
   0 = branch not-taken.
   1 = branch taken.
3. If BHT = 0, just increment the PC
   If BHT = 1, look up the new PC in the BTB

Since the BHT stores just 1 bit, it can store the history of a lot of instructions.
The BTB is only accessed when the BHT says the instruction is a taken branch.

Problems with 1 bit Predictors

The 1 bit predictor is very accurate, it works well for large loops with many iterations. This is because these branches tend to be always (or almost always) taken or not-taken.

1-bit predictors do not do well when the branch is not so predictable.
Each anomalous behavior with result in two mispredictions, so if the behavior changes a lot there will be many mispredictions.

2-Bit Predictors (2BP or 2BC)
A 2-bit predictor can reduce the number of mispredictions when there is a change in the branch prediction.
The two bits in the 2BP play the following roles:

MSB - the prediction, taken or not-taken
LSB - the conviction bit, sure, or not sure
00 - not-taken, strong
01 - not-taken, weak
10 - taken, weak
11 - taken, strong

A single anomaly will cost 1 misprediction, a change in behavior will cost 2 mispredictions.

Predictor Present State	Actual Branch Event	Predictor Next State
00	not taken	00
00	taken	01
01	not taken	00
01	taken	10
10	not taken	01
10	taken	11
11	not taken	10
11	taken	11

NT ↓

T ↑

2-Bit Predictor Initialization
If the 2BP starts in a weak state (01 or 10) , this will lead to zero mispredictions if correct and only one misprediction if wrong.
If the branch is alternating between taken, not-taken, starting in a weak state will lead to more mispredictions. This condition is slightly less likely to happen.
Most predictors just start in the 00 state because it is the easiest to initialize.

"worst case" sequence
↳ A "good" predictor is one where its "worst case sequence" is very rare / unlikely to occur.

Every predictor has a sequence that will result in every prediction being wrong.

1BP → 2BP
More bits in a predictor leads to more cost, without greatly improving the predictor accuracy.

History Based Predictors
Using history of the branch, the alternating patterns predictions can be learned.

1-Bit History with 2-Bit Counters
Steps for BHT with history:

1. Look up the BHT index with the PC
2. The BHT will have a 1-bit history, a 2-bit counter for history = 0, and a 2-bit counter for when the history = 1.
3. Look at the history bit. If the history bit = 0, use the first counter, if history bit = 1 use the second counter

2-Bit History Predictor
The pattern NNT mispredicts ⅓ of the time when a 1-bit history predictor is used.
Using a 2-bit predictor will solve this problem.
If a 2-bit history predictor is used the BHT now has 2 bits for history, and four 2-bit counters.

History Based Predictors with Shared Counters
The cost of adding counters to each entry in the BHT can become prohibitive. Sharing counters between entries is a good alternative since most of the counters are not used.

An N-bit History Predictor will predict all patterns of length <= N+1 → 2-bit History Predictor can predict patterns ≤ 3
An N-bit History Predictor will cost $N+2*2^N$ per entry, with most of the counters wasted.

2 for the past 2 outcomes + 1 for the current outcome

Use a Pattern History Table (PST) to reduce the cost of the predictor.
A pattern of 2 uses two counters, which means two entries in the BHT.
A pattern of 16 uses 16 counters (16 entries in the BHT).

PShare
PShare = Private history, shared counters - good for small loops and predictable short patterns.

GShare = Global history and shared counters - good for correlated branches.

GShare or PShare
Use both in a processor.

Tournament Predictors
Using two different predictors requires choosing one, but which one?
A tournament predictor can predict which predictor has the better prediction.

GShare	PShare	MetaPredictor
Correct	Correct	no change —
Correct	Incorrect	count down ↓
Incorrect	Correct	count up ↑
Incorrect	Incorrect	no change —

Hierarchical Predictors

While a tournament predictor combines to good predictors, a hierarchical predictor combines 1 good and one okay predictor.

In a tournament predictor both predictors are updated with every prediction. With a hierarchical predictor the good predictor is only updated when the okay predictor is incorrect.

Return Address Stack Predictor

If a branch is taken, the return address must be predicted. There are different types of branches:

Conditional branches: the BTB will work for predicting RAS

Unconditional branches: the BTB will work for predicting RAS

Function Returns: use a RAS predictor

RAS - a small hardware stack that is dedicated to storing the return addresses for functions.

When the RAS is full, use the wrap around the approach for replacement.

The return instruction needs to be identified before decoding. There are two ways to do this, using a predictor or using predecoding from the prefetch.

How do we know its a RET

Use a predictor or predecoding to determine when an instruction is a RET.

Summary

Wednesday, February 12, 2020 6:36 PM

We've learned that it's always better to predict than to not predict.

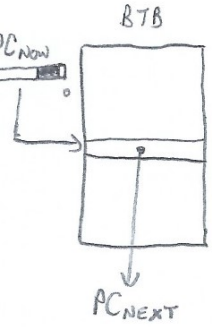
- By refusing to predict, we are choosing to ALWAYS waiting for EVERY instruction to be decoded (at least 2 cycles because 1st stage is fetch and 2nd stage is decode) to figure out if it is a branch instruction or not, and then if it is a branch instruction, waiting an additional cycle (execute stage) to figure out the address of the branched instruction. This is terrible CPI.
- Refusing to predict is choosing the "worst case scenario" to occur "all the time", whereas predicting will give us a mix of the "ideal" scenario and the "worst case scenario", which is much better than "all the time".

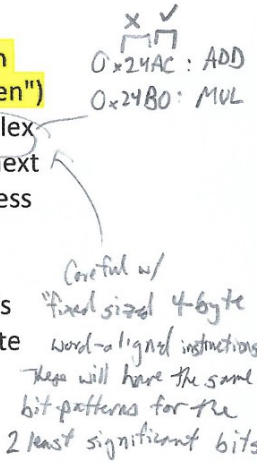
Branch Prediction involves making two predictions:

- "Direction" - Is this instruction a "branch taken" instruction or not?
- "Target" - If this instruction is a branch taken, what is the next instruction address that we should immediately go to next?

So we have multiple ways to do Branch Prediction:

- There's the "**Predict Not-Taken**" Predictor, which is a simple, no-history predictor which simply automatically assumes that every instruction is NOT a branch taken instruction and simply increments the PC. (This simply combines the prediction of "direction" and "target" into one prediction)
- There's the "**Branch Target Buffer/BTB**" Predictor, which is the simplest "history" predictor there is. It uses a table where, given an address of an instruction, what it "predicts" to be the address of the next instruction. As the processor executes, it will reference this table to predict a "next address" for a given instruction address, and then when the instruction reaches the stage of the pipeline where the next instruction address is actually known, it will then (if the predicted value and the actual value don't match) update the table entry with the actual next instruction address.
  - The BTB makes a "target" prediction, which technically can also be used to make a "direction" decision as well (if the predicted "target" is simply the following instruction in memory, then we're prediction that the "direction" of the current instruction is "not taken")
  - The BTB needs to be small in order to resolve "next" addresses quickly, so we can multiplex "input addresses" using their least significant bits to index into the BTB. So instructions next to each other spatially/address-wise will end up using different BTB entries and lead to less frequent BTB entry conflicts.
    - In the case of a conflict, an instruction would index into the same BTB entry as another instruction, meaning that it would get the wrong "next" address, but that's okay because that just also counts as a misprediction, and we will eventually update the entry the this instructions "actual next address".
      - However, since BTB entry conflicts are also mispredictions, then we should avoid them as much as possible too.





BTB Summary

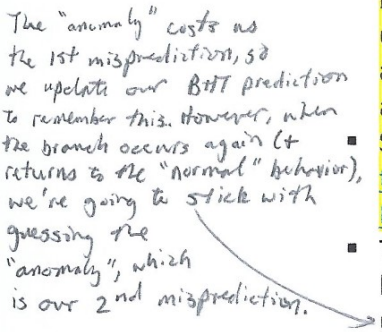
The "Branch Target Buffer" is exactly what it sounds like. It only tracks what the "target" (aka address of the next instruction) of a potential branch might be.

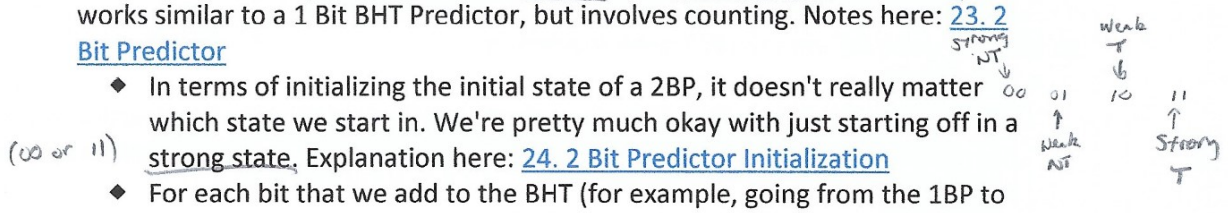
- In CONJUNCTION with the BTB, we can use a "**Branch History Table/BHT**" **(aka 1 Bit Predictor)**. Similar to the BTB, it's a history based predictor, but the difference is that it only tries to predict/remember if the current instruction (given its address) was a taken branch

DIRECTION PREDICTOR

instruction or not. Remember, the BTB tries to predict/remember (given an instruction address), what the next address is (either simply incremented or a branch to another address).

- We'd only need 1 bit for each BHT entry (0 = not a taken branch, while 1 = taken branch), so our BHT can have more entries than the BTB.
- With the **BHT (aka 1 Bit Predictor)**, we ONLY need to use the BTB if the BHT predicts a 1. Otherwise (0 = not a taken branch), we don't need to use the BTB and can simply increment the PC. This greatly reduces the use of the BTB, meaning we don't need to use the scarce BTB entries to remember simple "the next address is just the current address incremented". This leads to LESS BTB conflicts and therefore LESS "next address" mispredictions.
- So "when you have both the BHT and the BTB, the BTB is accessed only if the BHT says that this is a taken branch. If this is not a taken branch, then we can just increment the PC".
- The **1 bit BHT predictor** is simple and works well for simple, reoccurring patterns of branch decisions, but when a single anomaly in the repeating pattern occurs, it costs us 2 mispredictions. Also, when the pattern is more complex than a reoccurring pattern (for example, frequently switching between a reoccurring patterns), the 1 bit BHT predictor starts to fail. Notes here: 22. Problems With 1 Bit Predictions (BHT)
  - A **2 Bit Predictor/2 Bit Counter (2BP/2BC)** behaves similarly well as the 1 Bit BHT Predictor, but does not mispredict twice whenever there is an anomaly. It works similar to a 1 Bit BHT Predictor, but involves counting. Notes here: 23. 2 Bit Predictor
    - In terms of initializing the initial state of a 2BP, it doesn't really matter which state we start in. We're pretty much okay with just starting off in a strong state. Explanation here: 24. 2 Bit Predictor Initialization
    - For each bit that we add to the BHT (for example, going from the 1BP to 2BP), we're making it more difficult for an anomaly (or anomaly pattern) to convince the predictor to change its prediction pattern, but each additional bit also doubles the cost of the predictor. It's sort of rare for these types of branch decision patterns to occur, so it becomes less worth it to add more bits to our predictor (for example, not really worth to have a 4 bit predictor). Notes here: 26. 1BP, 2BP
  - However, neither the 1 Bit or 2 Bit (BHT) Predictors work well when dealing with alternating patterns of branch decisions (for example: NT, T, NT, T, NT, etc.) For this, we'd need a "**History Based Predictor**"





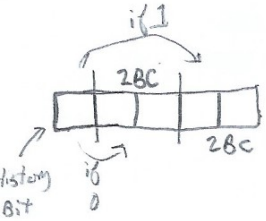
Remember, simple N-Bit (BHT) Predictors/Counters can only deal with **reoccurring branch decisions with the occasional "anomaly"**. More bits allows us to handle larger anomalies, but these large anomaly situations are rare, so it's not really worth it to keep doubling the cost of the counter for each bit that we add. However, these N-Bit (BHT) Predictors/Counters can't deal with **alternating patterns of branch decisions**. For that, we'll need a History Based Predictor.

Branch History Table: Using an instruction's address to predict whether the instruction is a TAKEN branch or not:

- BHT predictors are used to predict a "direction" (is the instruction a branch taken instruction or not)
- A simple 1 Bit Branch Predictor (1BP)
- 2 Bit Branch Predictor (2BP/2BC)

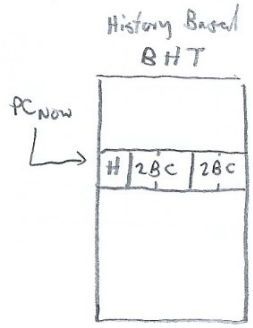
↙ Used to deal w/ alternating patterns of branch decisions (T, NT, T, NT, etc.)

- A History Based BHT
  - 1-Bit History Predictor
    - This is a BHT where the entries are 5 bits each (Notes here: 28. 1 Bit History With 2 Bit Count)
      - 1 Bit is the "History Bit"
      - 2 Bits are used as a counter (like the 2BP/2BC) for when the "History Bit" is 0
      - 2 Bits are used as a counter (like the 2BP/2BC) for when the "History Bit" is 1
    - So in this 1 Bit History Predictor, the History Bit keeps track of what the previous outcome was, and based on that history, we have a separate 2-Bit Predictor/Counter that will be used to predict what it believes the next outcome will be (for that previous outcome).



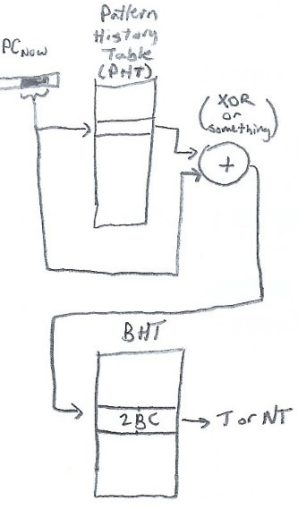
"last time we were at this address, we got a 0 (NT), so based on that, we'll use our 1st 2BC to tell us what it thinks the next outcome will be"

  - N-Bit History Predictor
    - We can increase the number of bits that we use to track a history, where N is the number of History Bits and when put together, each unique combination of those N bits (2^N of them) represents a specific pattern of previous outcomes.
    - An N-Bit History Predictor can Predict Patterns with Length <= N + 1
    - However plain N-Bit History Predictors can be costly:
      - Increasing the number of History Bits in a History Based BHT leads to an exponential increase in cost, because N History Bits means we need 2^N 2-Bit Counters (which each take up 2 bits). So N-Bit History Predictor Cost = N + 2 * 2^N
      - As we include more 2-Bit Counters in our N-Bit History Predictor, for any given entry (which is indexed by the PC branch instruction address), most of the 2BCs for that entry are unused. For a single branch, only specific history patterns will be used, and the rest of them will go unused. So our N-Bit History Predictor costs more as N increases, but also more of that cost is unused and wasted.



UPDATING THE HISTORY-BASED BHT
- Use history bit again to figure out what 2BC to update.
- Update 2BC w/ outcome (if T, increment. If NT, decrement)
- Then take the outcome and left-shift it into the HISTORY BIT(S). (if T, shift a "1") (if NT, shift a "0")

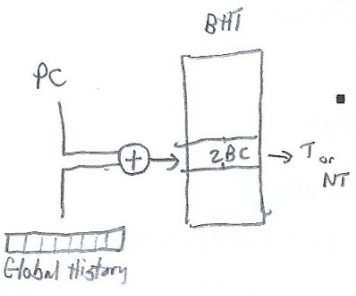
  - **History Predictor with Shared Counters** (aka **PSHARE**)
    - We can reduce the costly waste of having to keep unused 2BCs in each entry by separating out the 2BCs into a separate table and sharing (aka multiplexing) them.
    - This approach now has two separate tables:
      - Pattern History Table (PHT)
        - This is a simple table where each entry ONLY keeps track of the history bits.
      - "Branch History Table" (not to be confused with the other, general BHTs above)
        - This is another table where each entry is a 2BC.
    - History-Based Predictor with Shared Counters to Reduce Wasted 2BC
    - How Do History Based Predictors with Shared Counters Work?
    - Although its possible for different instruction addresses to map/use the same 2BC now, it happens very rarely if we make our "BHT" table very large (aka have a lot of 2BC entries), which is very doable because since each entry is only 2-bits in size, our entries are cheap and we can have a lot.
    - Also known as PSHARE (each branch instruction has its own Private histories but **SHARE** counters)
    - PSHARE is good for short iteration loops or branches that have alternating branch sequences



UPDATING PSHARE
1) Update the 2BC with the outcome
2) Left-shift the outcome to the PHT entry

The PHT has a single history entry per instruction address, but the same address combined with different histories (from the SAME PHT ENTRY) will give us different 2BCs!

  - **Global History Predictor with Shared Counters** (aka **GSHARE**)
    - GSHARE is a similar but simpler version of PSHARE, except that there is not "Pattern History Table", but instead there is one single, global History Entry that ALL branch outcomes are pushed into. More notes here: How GSHARE Works
    - This is good for "correlated branches", which you can find an example of here: Example of Correlated Branches
      - Basically, two separate but side by side if statements (if A = X) followed by (if A != X). We can correlate the outcome of one branch with the outcome of the

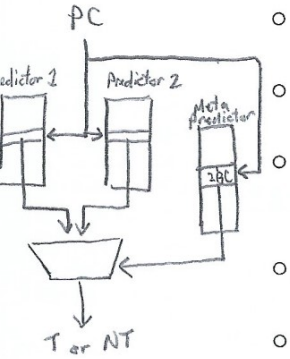


other branch. So if we know one outcome, we can predict the other outcome.
    - Look at how GSHARE and PSHARE are used in an example here: 38. Quiz: PShare vs GShare Quiz
    - Compared to PSHARE, GSHARE is able to predict correlated branches (which PSHARE cannot do), but often requires a longer history than PSHARE. However, this is not that big of a deal, since with PSHARE each entry has its own private history while with GSHARE, there is just a single, global history (so it isn't very costly to make it longer).
  - Modern Processors use BOTH PSHARE & GSHARE History Based Predictors

Methods to Combine Different "Direction" Predictors

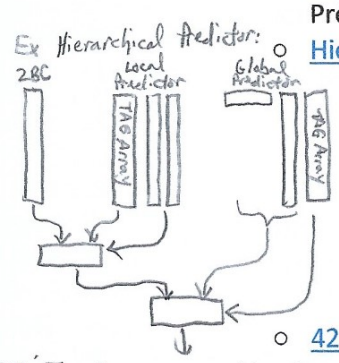
Since different Predictors have different strengths and weaknesses, why pick and choose only one when you can use multiple? There are different methods of "combining" (or "simultaneously using") different branch predictors:

- Tournament Predictor
  - Tournament Predictors are used when you want to combine two "good" predictors that are uniquely good at difference scenarios.
  - When using two predictors in a "Tournament Predictor" style, Both Predictors will make a Prediction, and use a Meta-Predictor to decide which Prediction to use
  - The Meta-Predictor is just another array of 2BC's, where each entry/2BC makes a prediction on which of the 2 predictors are more likely to be correct **for a given branch instruction**. Notes: How a Meta-Predictor Works
  - You update the two separate predictors like you would normally, but you update the Meta-Predictor entry based on Which Predictor was Correct
  - With a Tournament Predictor, you're basically doing 2 "good" predictions for every branch just to use one of them. And each of the "good" predictors costs you a lot for each entry.
    - So the Tournament Predictor can be highly accurate, but that accuracy comes at a cost of using and maintaining two (or more) costly "good" predictors.



Predictor 1 correct?	Predictor 2 correct?	Ch… & n… (2B…
✓	✓	–
✓	×	↓
×	✓	↑
×	×	–

- Hierarchical Predictor
  - Hierarchical Predictor are used when you want to combine a "good" predictor with an "okay" predictor.
  - Hierarchical Predictor Usually Better than Tournament Predictor because while Tournament Predictors are highly accurate, they're costly to use.
  - Hierarchical Predictor = Optimizing for the Common Case (simple branch prediction)
    - Its usually the case where most branches don't have fancy outcome patterns or need a lot of resources to predict their outcome with high accuracy. Only a few branches actually need those heavy predictor resources. Therefore, we can optimize for the common case by usually using a cheap, "okay" predictor that can still perform well for the simple branch patterns while saving the rest of our resources for a more "costly" but "good" predictor for the more complex branches that have more intricate outcome patterns.
  - 42. Hierarchical Predictor Example



The Tag Array will indicate, for a given address, if that entry is FOR that address (if the tag matches). If the tag doesn't match, then we shouldn't use this predictor & move down the hierarchy.

- Here's a good example on how you can combine several different predictors with **nested** combinations of "combined" predictors: 43. Quiz: Multi Predictor Quiz

· 2BP works fine for 95% of instructions
· PSHARE works fine for some 95% + additional 2%
· GSHARE works fine for same 95% + the other 3%

The overall predictor is a hierarchical predictor that chooses b/w the 2BP & a tournament predictor, which then chooses b/w the PSHARE + GSHARE predictors.

"Target" Predictors - BTB & RAS

- The "target" predictors (assuming that the current instruction is a "taken" branch instruction, what is the address of the next instruction that we need to execute?) that we mostly learned about is the BTB.
  - Notes: 44. Return Address Stack (RAS)
- However, the BTB doesn't work well for returning from Function Calls. For that we need a Return Address Stack (RAS)
  - The RAS is a fast hardware "stack" that we push the return address on a function call and

then pop off the return address on a "return" instruction.

- This is a separate stack from the software call stack because this is a hardware structure on the actually processor. We implement this as a hardware structure because we use this for processor prediction, so it needs to be very fast (~1 cycle).
- The speed requirement also limits the size of the structure, so we're bound to run out of space on deeply nested function calls. When we run out of space, we can either:
  - Stop pushing to the stack and preserve what already exists on there.
  - Or wrap around and overwrite the existing entries on the stack.
    - Better to "overwrite/wrap around" than to "stop pushing" when RAS is full
    - Neither is perfect, but "Wrap Around" gives us less mispredictions