Speculative Execution

Wei Wang

Optional Text Book Chapters

- “Computer Architecture: A Quantitative Approach,” Appendix C.2 and C.3

Road Map

- Overview of Branch Prediction
- Branch Prediction Algorithms
- Advanced Branch Prediction
- Memory Disambiguation
- Security Implications of Speculative Execution

Overview of Branch Prediction

Control Hazards

- Mostly caused by branches
- Key to performance in modern microprocessors.
- Solutions:
  - Stalling the pipeline
  - Assume not taken
  - Branch delay slot
  - None of these solutions really address the problem

Branch Prediction

- The best solution to branch-induced control hazards is Branch Prediction.
- For a branch instruction, Branch prediction predicts
  - Whether the branch is taken or not
  - The address of the branch target if the branch is taken
- The prediction is then used to fetch next instruction.
  - If prediction is correct, we can completely avoid stalls. Hence improving performance.

Branch Prediction cont'd

- Branch prediction is not perfect
  - If the prediction result is wrong, pipeline has to be flushed.
- Almost every design decision changes if we have "perfect" rather than realistic branch prediction.

An Example of Branch Prediction

- Consider the following the C code and its corresponding assembly codes

```
If (R1 == 2)
     R1 = 0;
else
     R1 = 2;

R2 = 3;
```

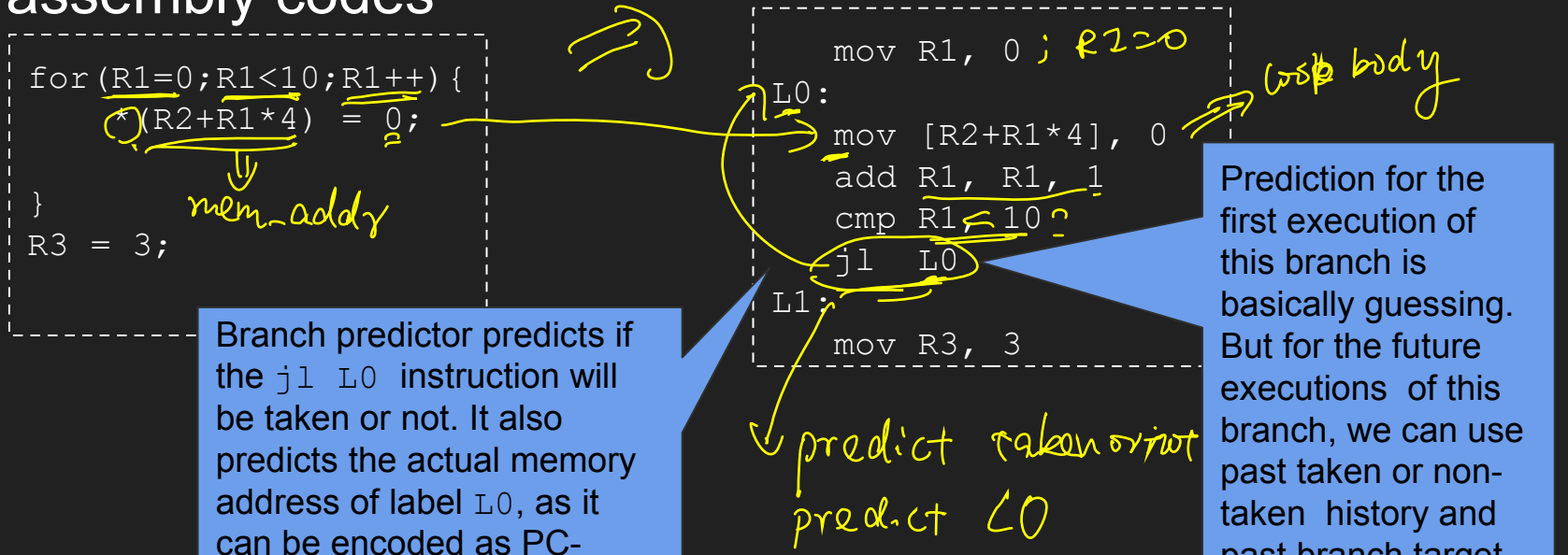
```
     cmp R1, 2
     jne L1
     mov R1, 0
     jmp L2
L1:
     mov R1, 2
L2:
     mov R2, 3
```

Branch predictor predicts if the `jne L1` instruction will be taken or not. It also predicts the actual memory address of label `L1`, as it can be encoded as PC-relative address.

Prediction for this branch is essentially a guess, as there is very limited information to aid us in this prediction.

Another Example of Branch Prediction

- Consider the following the loop and its corresponding assembly codes



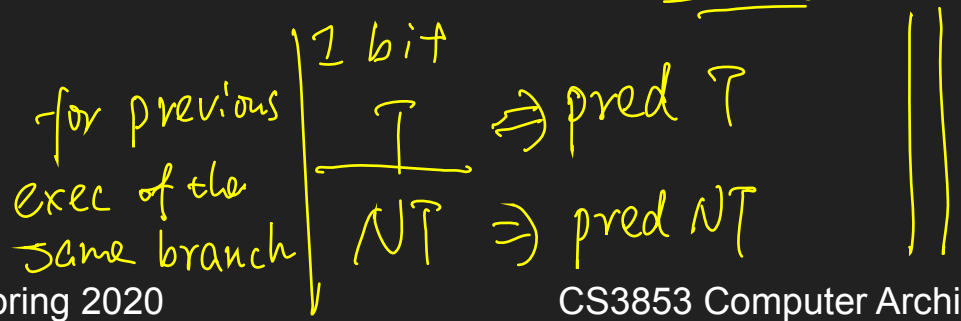
Branch Prediction Algorithms

Dynamic Hardware Branch Prediction

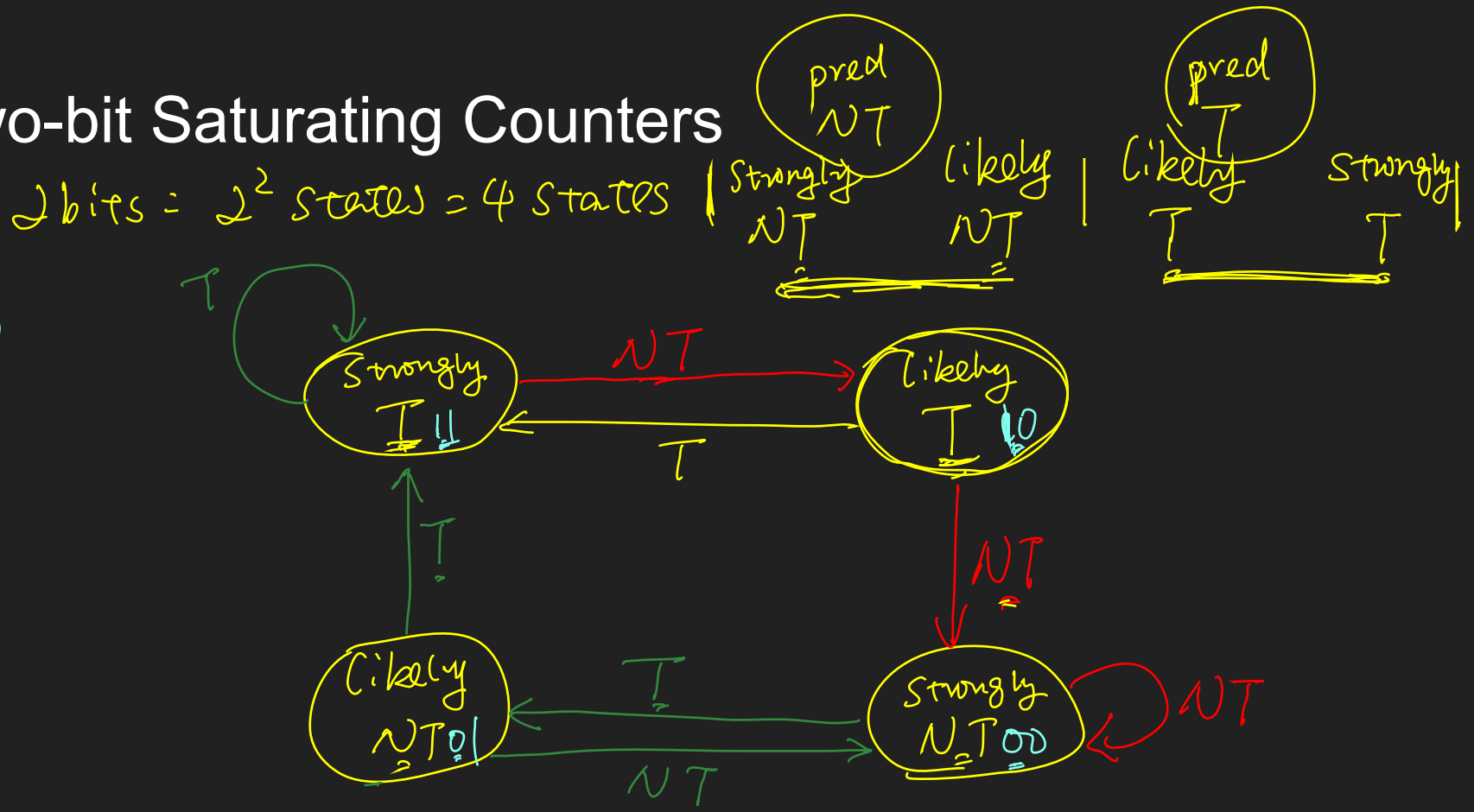
- Branch behavior is monitored during program execution
  - History data can influence prediction of future executions of the branch instruction
- Branches instruction execution has two tasks/predictions
  - Condition evaluation (taken or not-taken)
  - Target address calculation (where to go when taken)
- Target prediction also applies to unconditional branches
- Branch Direction Prediction: 3 levels of complexity
  - Branch history tables, Two-level tables, hybrid predictors

Branch Direction Prediction (taken or not ?)

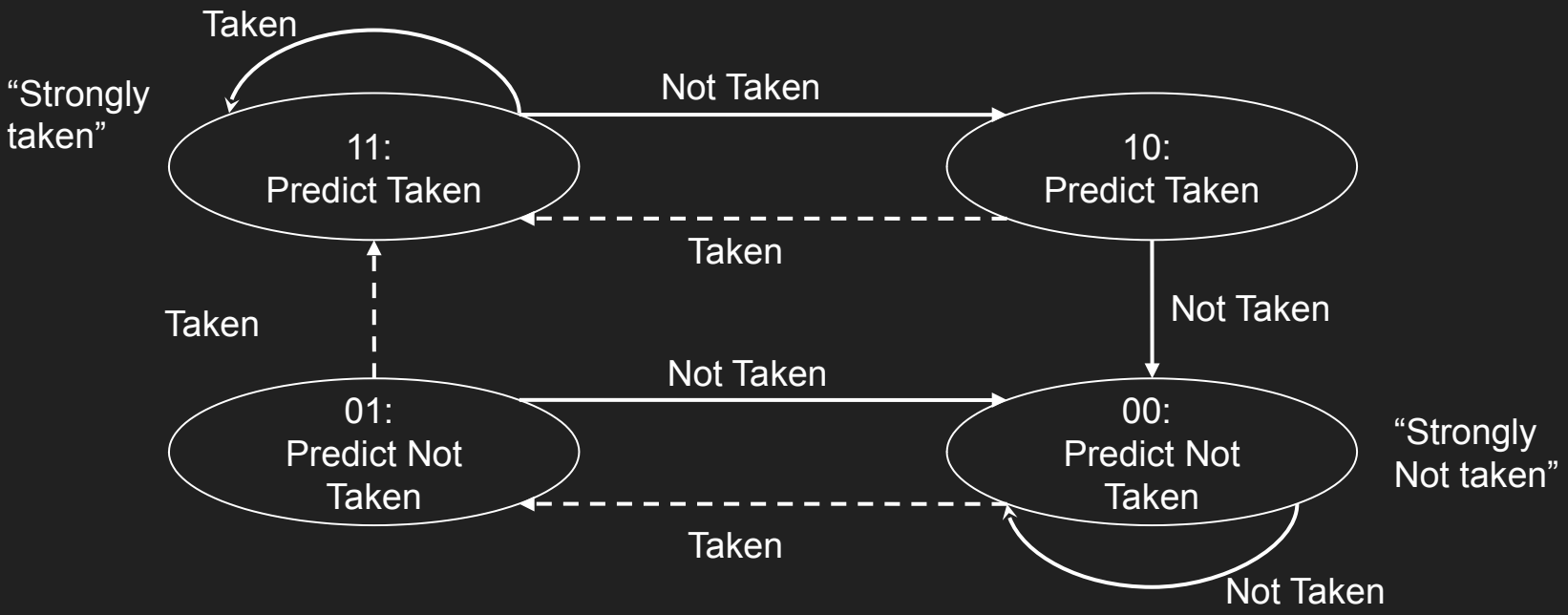
- Basic idea: Hope that future behavior of the branch is correlated to past behavior. This idea works well for,
  - Loops
  - Error-checking conditionals
- For a single branch PC
  - Simplest possible idea: Keep 1 bit around to indicate taken or not-taken
  - 2nd simplest idea: Keep 2 bits around, have a bit more history



Two-bit Saturating Counters

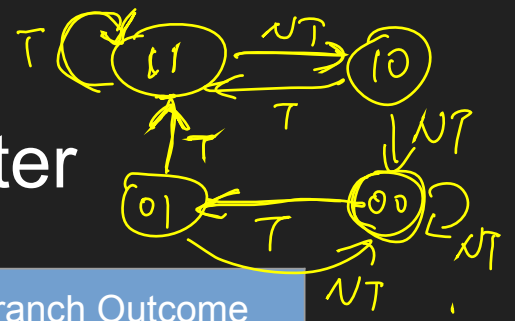


Two-bit Saturating Counters



- 2-bit history means prediction mut miss twice before change.
- N-bit predictors are possible, but after 2-bits not much benefit.

An Example of Two-bit Saturating Counter



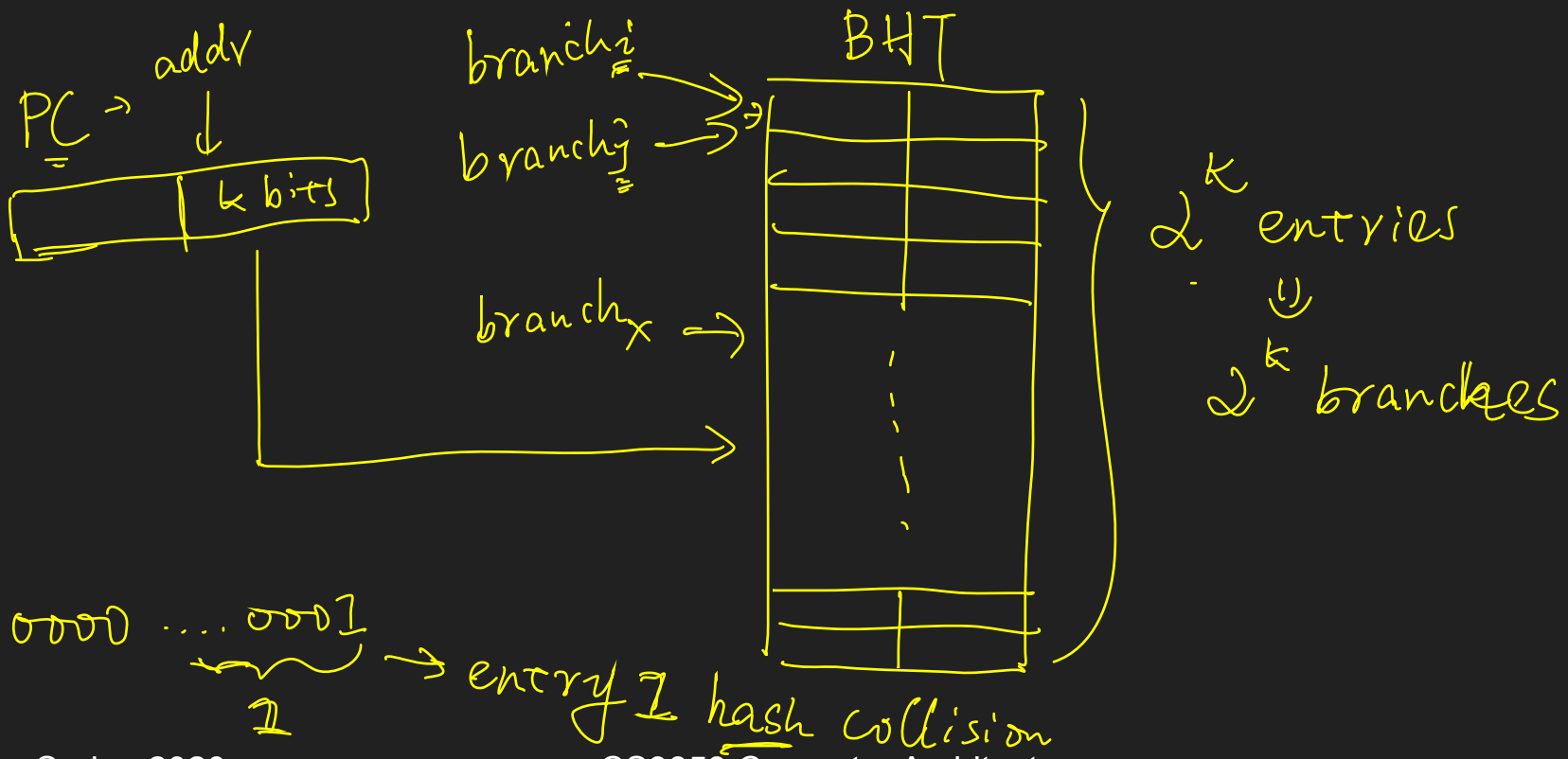
State	Prediction	Actual Branch Outcome
00	NOT TAKEN	Taken
01	NOT TAKEN	Taken
11	Taken	Taken
11	Taken	Taken
11	Taken	Taken
11	Taken	Taken
11	Taken	Not Taken
10	Taken	

An Example of Two-bit Saturating Counter

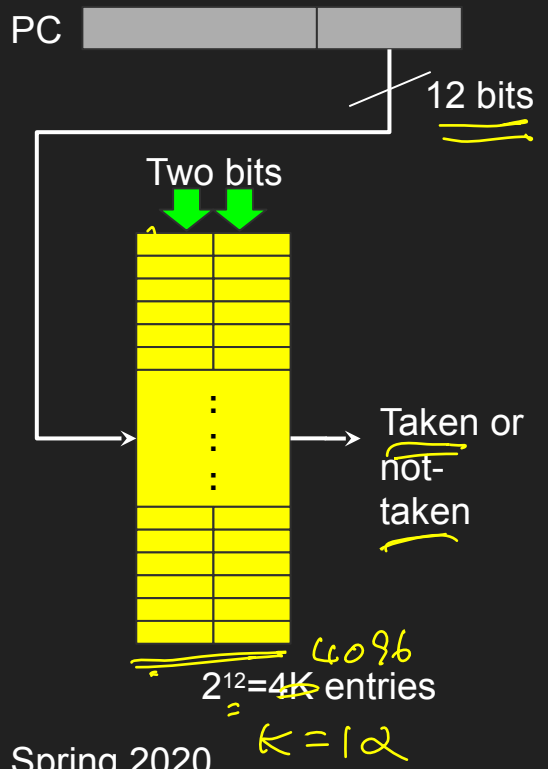
State	Prediction	Actual Branch Outcome
00	Not Taken	Taken
01	Not Taken	Taken
11	Taken	Taken
11	Taken	Taken
11	Taken	Taken
11	Taken	Taken
11	Taken	Not Taken

3 mispredictions, 4 correct predictions, accuracy is 4/7.

Branch Prediction Buffer (branch history table, BHT)



Branch Prediction Buffer (branch history table, BHT)



- Small memory indexed with lower bits of the branch instruction's address
  - Why the low bits?
  - Branches sharing the same lower bits share the BHT entry, thus interference.
- Implementation of BHT
  - 2-bits attached to each block in the Instruction Cache
    - requires separate memory accessed during IF phase
  - Caveats: Cannot separately size I-Cache and BHT

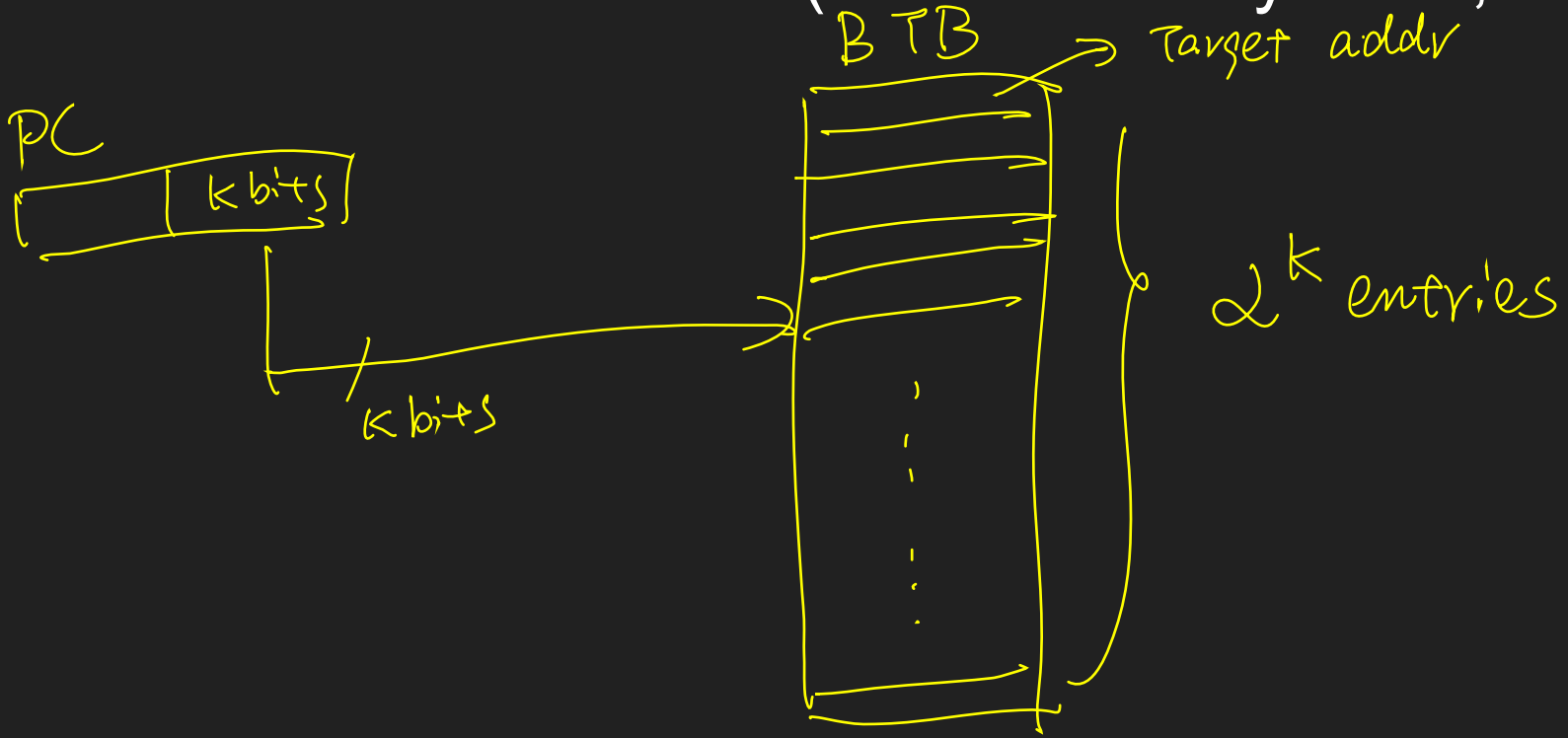
Branch Target Prediction

- Besides predicting branch taken or not, branch targets also needs to be predicted.
- Branch Target Buffer (BTB) is the storage used to store the branch target address from previous execution of the branch.
  - This is essentially a cached design.
  - If predicted taken, the cached target address is extracted as predicted branch target.

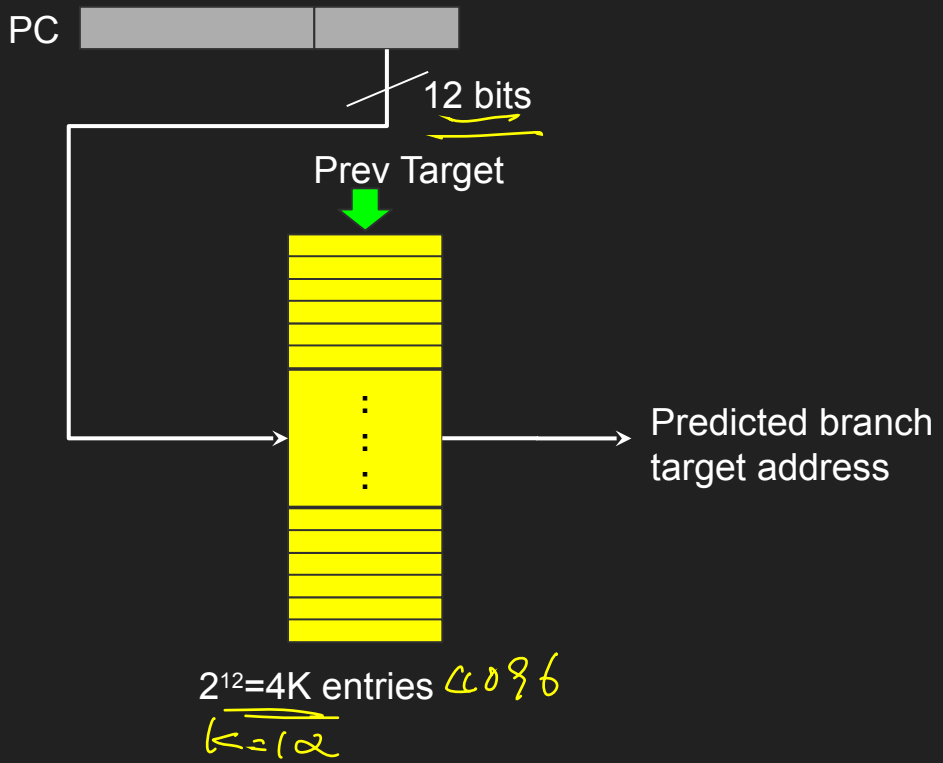
BTB (cached)

Branch Target Buffer (BTB)

~~Branch Prediction Buffer (branch history table, BHT)~~



A Example of Branch Target Buffer

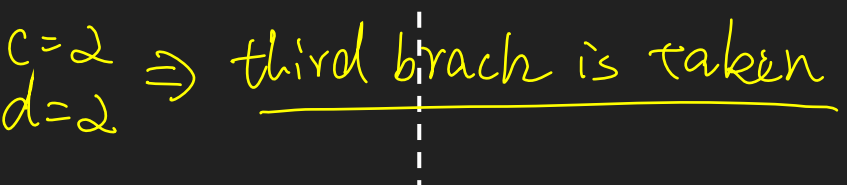


Advanced Branch Predictions

Correlated Branches

- Consider the following code:

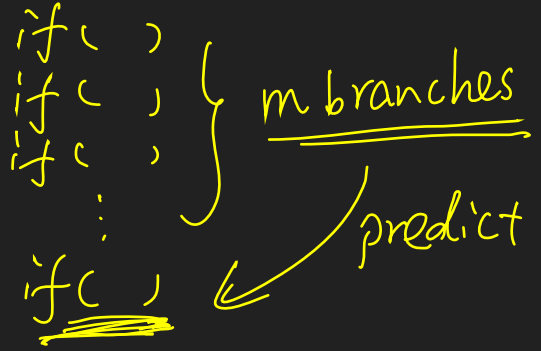
```
if (a == 1)
    c = 2;
if (b == 1);
    d = 2;
if (c == d)
    e = 2;
```



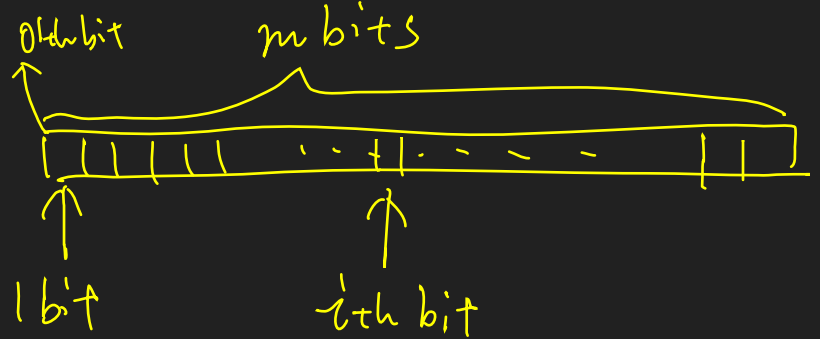
- The if-statements in the above code are correlated.
  - If the first two branches are taken then the third branch will definitely be taken.

Correlating Branch Prediction

- Correlating branch prediction is based on the observation that the outcomes of some branches are related to other branches.
  - Correlating branch predictor predicts if a branch instruction is taken or not based on the outcomes of m branch instructions before it.
  - For example, in the previous code, the third branch should be predicted taken if the previous two branches are taken.



m-bit Global Branch History Register (GBHR)



record the history of the ith branch:

1: Taken

0: Not taken

m = 2 ⇒

2-bit GBHR

0 or 1

move

shift

1: if ( ) T

2: if ( ) NT

3: if ( ) T

4: if ( )

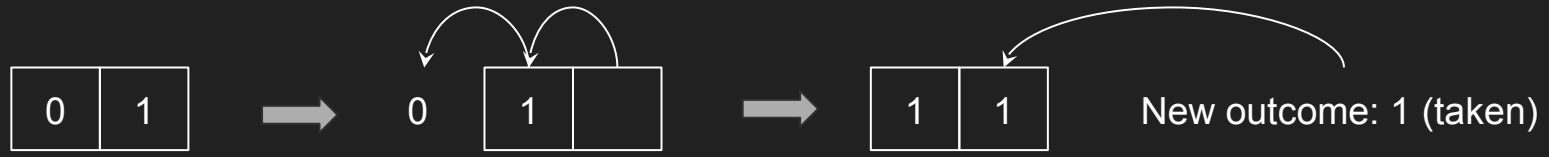
predict outcome

m-bit Global Branch History Register

- A typical m-bit correlating Global Branch History register (GBHR) stores the outcomes of m branches right before the branch-to-be-predicted.
- A m-bit shift register is used to record the outcomes of m previous branches.
  - For example, the follow two-bit register is used to record the outcomes of two branches preceding the branch-to-be predicted. 1 for taken, and 0 for not taken. So the following register records a non taken and a taken registers.

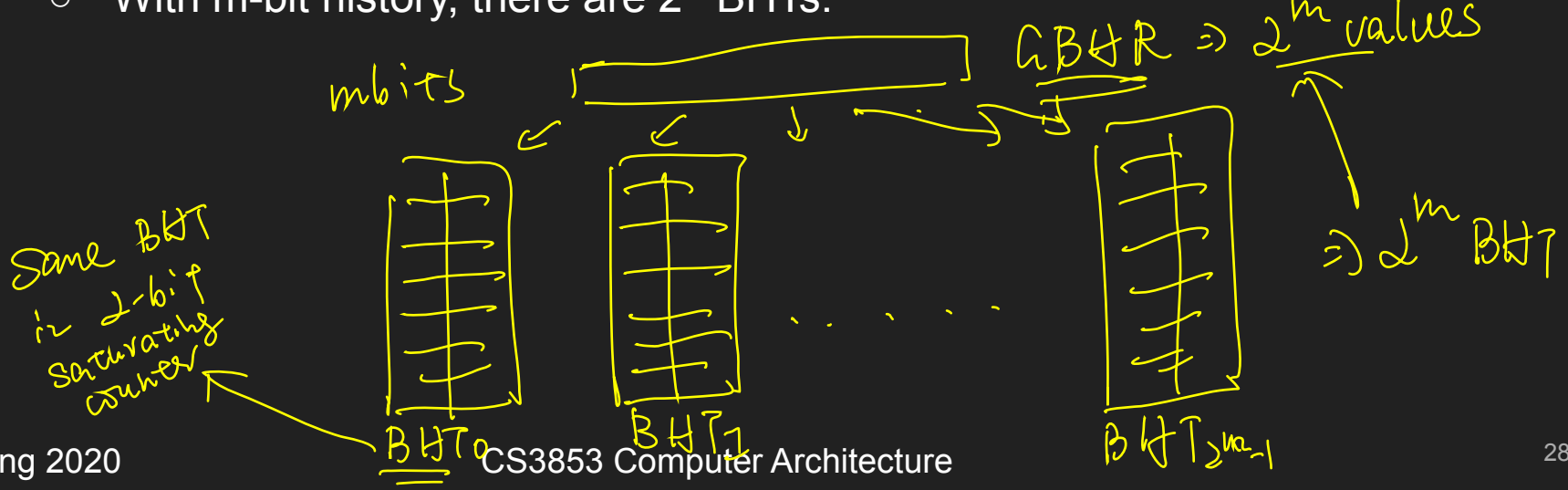
m-bit Global Branch History Register cont'd

- A m-bit shift register is used to record the outcomes of previous branches.
  - When a new branch outcome is acquire, the shift register shifts the highest bit out and store the new outcome in its lowest bit
  - For example, if a new branch outcome is taken, the previous two-bit register is changed to:



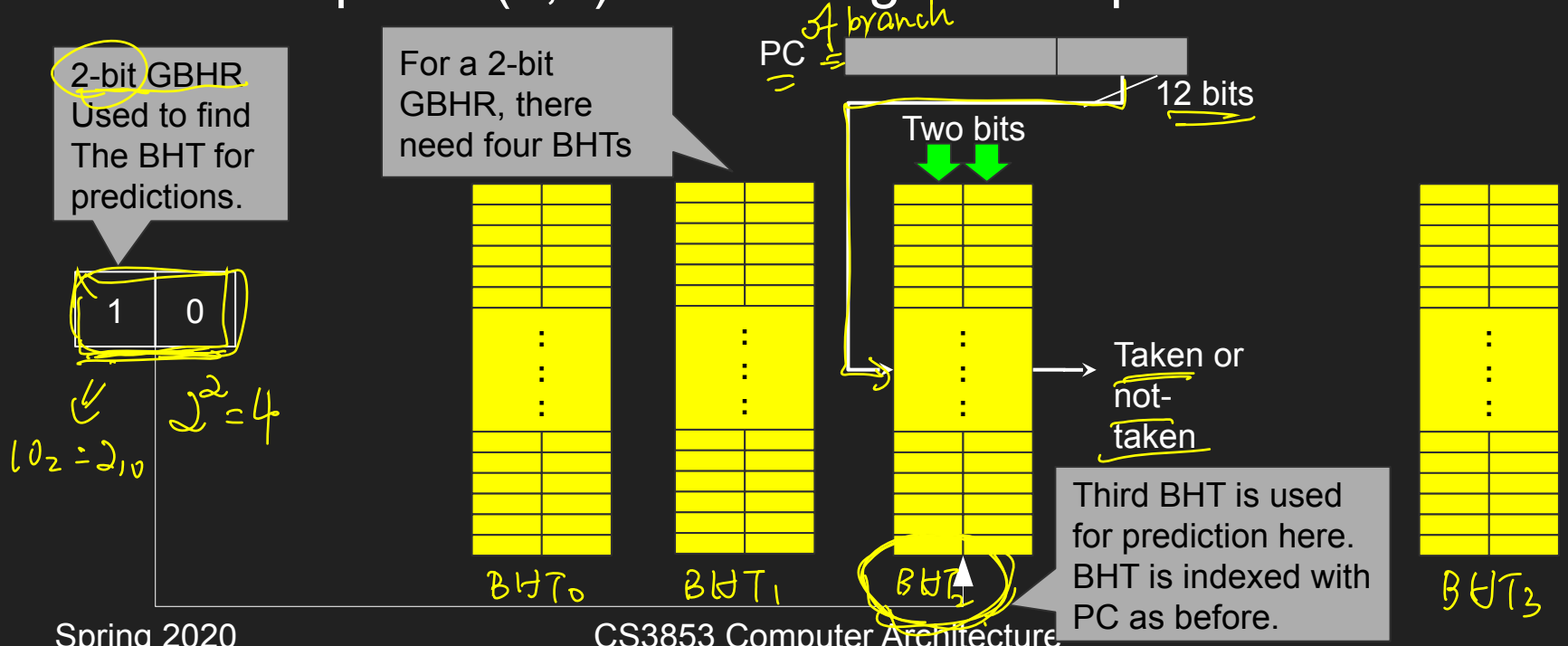
(m, n)-Correlating Branch Predictor

- A (m, n)-Correlating Branch Predictor combines the m-bit GBH register with several n-bit saturated-counter-based branch history table (BHT).
  - With m-bit history, there are 2^m BHTs.



(m, n)-Correlating Branch Predictor cont'd

- A example of (2,2)-correlating branch predictor



Tournament Predictor

correlating predictor better

- In real programs, some branches are related to other branches, while other branches are completed independent. → 2-bit saturating counter better
  - Therefore, sometimes correlating predictor is better, sometimes simple predictor is better.
- Tournament branch predictor combines both predictors and dynamically choose the best one

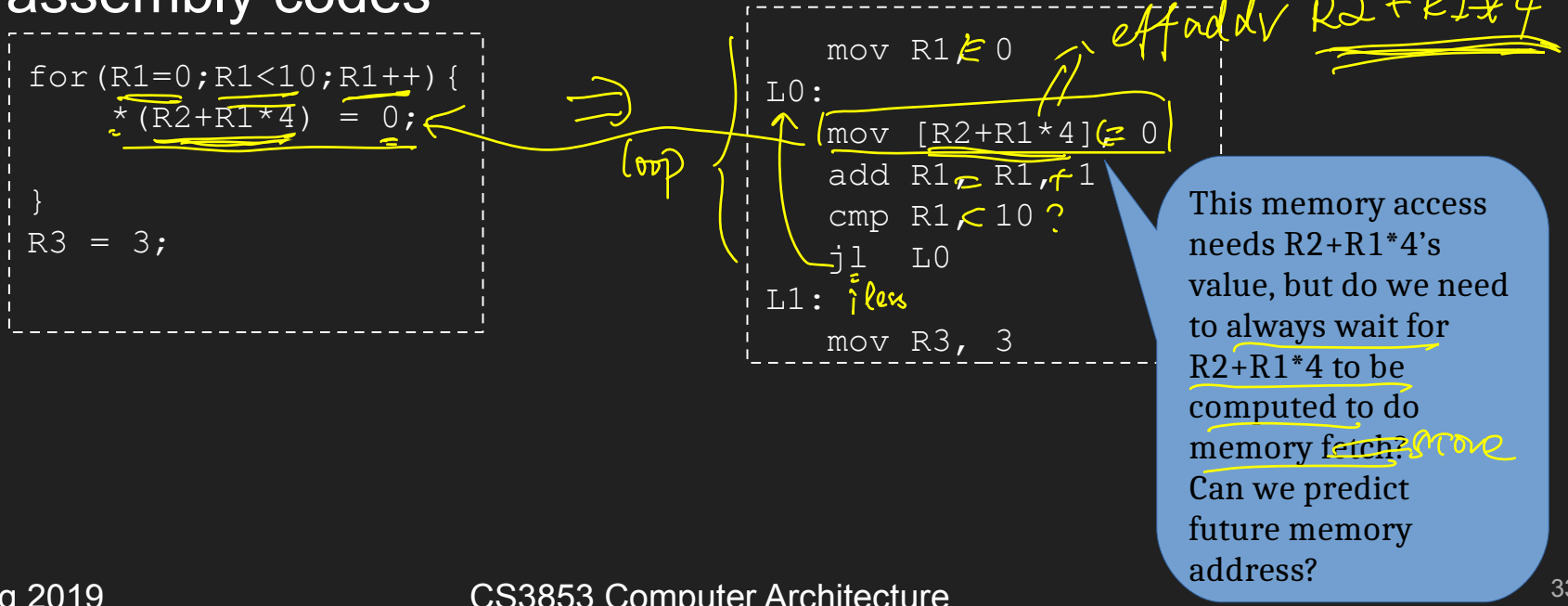
Tournament Predictor cont'd

- Local, per-branch prediction, accessed by the PC
- Correlating prediction based on the last m branches, assessed by the global history
- Indicator of which had been the best predictor for this branch
- 2-bit counter: increase for one, decrease for the other

Memory Disambiguation

An Example of Memory Disambiguation

- Consider the following the loop and its corresponding assembly codes



An Example of Memory Disambiguation cont'd

- Consider the following the loop and its corresponding assembly codes

```
for(R1=0;R1<10;R1++){
    *(R2+R1*4) = 0;

}
R3 = 3;
```

*(R2+0) = 0 +4B
*(R2+4) = 0 +4B
*(R2+8) = 0
⋮
*(R2+36) = 0

```
    mov R1, 0
L0:
    mov [R2+R1*4], 0
    add R1, R1, 1
    cmp R1, 10
    jl  L0
L1:
    mov R3, 3
```

predict the itv of this mov instr access next 4B

Clearly, this mem read instruction access the memory with a stride of 4 bytes. The CPU can easily observed the stride and predicts it.

Memory Disambiguation

- Memory Disambiguation is a technique to execute memory access instructions speculatively by predicting the memory address of that access.
- The benefit of memory disambiguation
  - Issue memory accesses as early as possible, allowing more overlapping of memory accesses and computation of eff addr
  - Allow more out-of-order instruction execution by speculatively assuming no data dependencies.
    - Usually, memory disambiguation is only in OoO processors.

mov [R2], 0
mov R4, [R3]
depend? relies on R2 ≠ R3? if R2 ≠ R3 execute in parallel

Memory Disambiguation: Incorrect Speculation

- For read memory accesses
  - Never really load the data to real register.
  - Loads are always buffered in temporary registers until the memory loads are known to be safe.
- For write memory accesses
  - Never really write the results to memory speculatively.
  - Writes are always buffered in temporary (but close-to-memory) registers (e.g., write buffers) until the memory writes are known to be safe.

Memory Disambiguation: Incorrect Speculation cont'd

- Wrong data dependencies
  - Pipeline have to flushed and wrong results have to be discarded

Security Implications

Speculative Execution and Security

- "Speculative execution is an optimization technique where a computer system performs some task that may not be needed or should not be executed."
  - Branch prediction is a type of speculative execution.
- Speculative execution is extensive in modern processors since they are crucial for performance.
- If was until the second half of 2017 that we learned the security impact from speculative execution, by Spectre and Meltdown vulnerabilities.

The Spectre Vulnerability

- Consider the following pseudocode:

```
if (do security check and pass){
    Access_sensitive_data()
}
```

speculatively exec

- Normally, the sensitive data can only be accessed if the security check is successful. However, with speculative execution the sensitive data may be accessed before the we know if the security check is passed or not.
  - When the sensitive data is read into cache, a separate side-channel attack is used to access this data in the cache.

The Spectre Vulnerability cont'd

- Consider the following pseudocode:

```
if (do security check and pass){
    Access_sensitive_data()
}
```

another branch similar PC (12 bits) to train the BHT to always for the security check branch

pred to taken

- The attacker can train the branch predictor to always predicted true for this if-branch.
  - Recall the branch predictor only uses the low bits of the PC.
  - The attacker can find another branch instruction who's PC has the same lower bits as this if-branch. The attacker can then train the branch predictor to predict "taken" by letting this 2nd branch instruction be taken for several times.

The Meltdown Vulnerability

- Similarly to Spectre, except the security check is performed by the hardware.
  - When accessing OS kernel's data, CPU will check if current user has access to kernel's data.
- However, the data access is speculatively executed as the same time as the hardware security check. Hence, the data may be leaked in the cache.

Solutions to Meltdown and Spectre

- Software solutions usually have a high performance cost.
  - Afterall, the whole reason to do speculative execution is for better performance.
- Hardware solutions are elusive.
  - Chip manufacturers have proposed/released some solutions for certain variants of Meltdown and Spectre. However, there is no general solution.
  - Hardware solutions are mostly likely come with a performance penalty.
    - If Dennard scaling is still alive, we may be in a much better situation.

Acknowledgement

- This lecture is partially based on the slides from Dr. David Brooks and Dr. Susan Eggers.