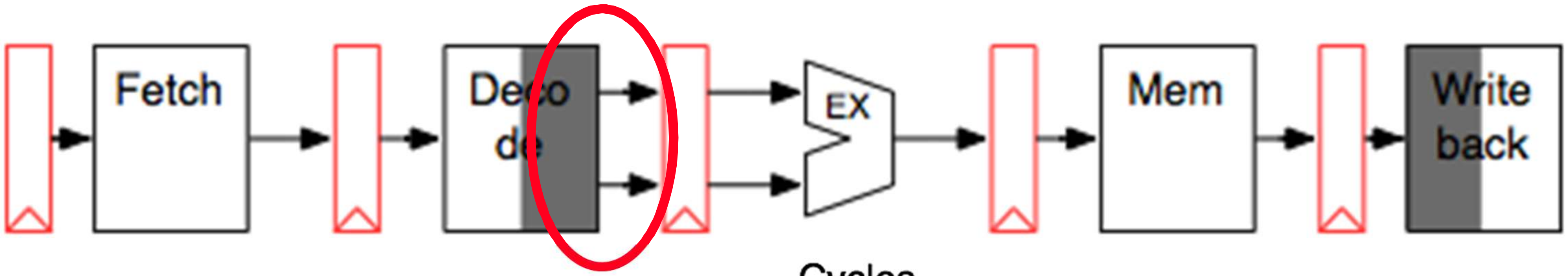
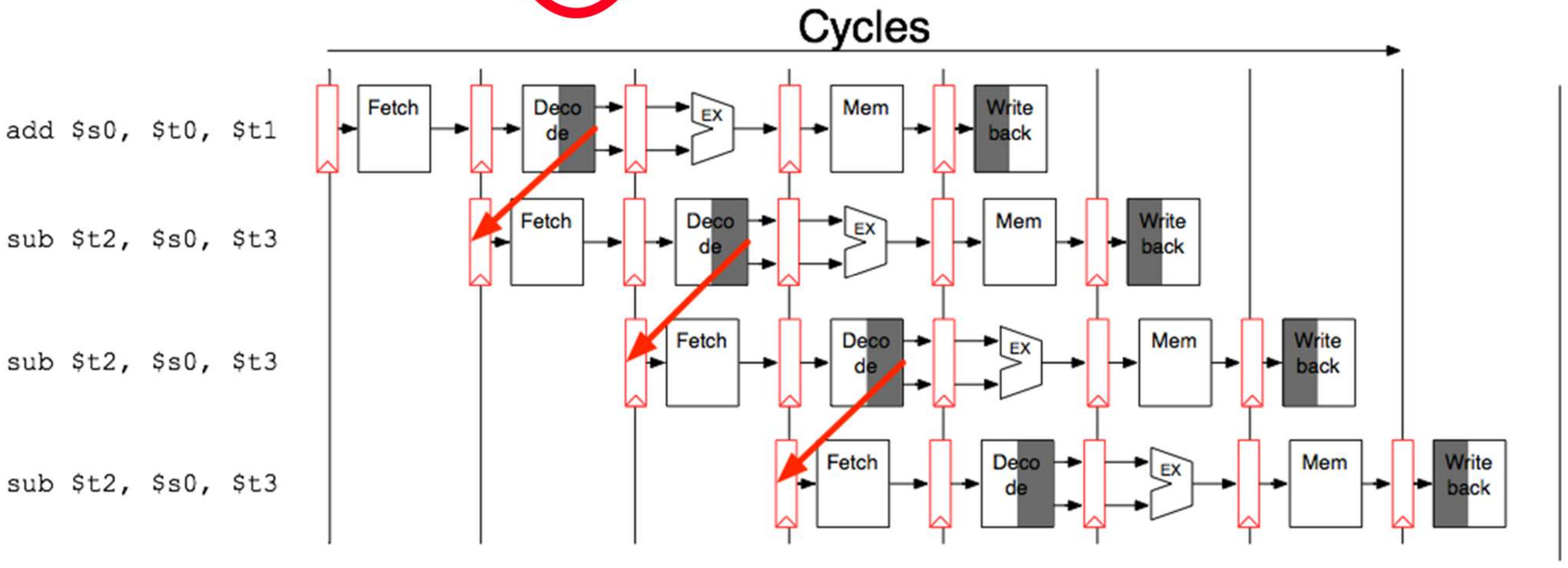
Key Points: Control Hazards

- Control hazards occur when we don't know what the next instruction is
- Caused by branches and jumps.
- Strategies for dealing with them
  - Stall
  - Guess!
    - Leads to speculation
    - Flushing the pipeline
    - Strategies for making better guesses
- Understand the difference between stall and flush

Computing the PC Normally

- Non-branch instruction
  - PC = PC + 4
- When is PC ready?

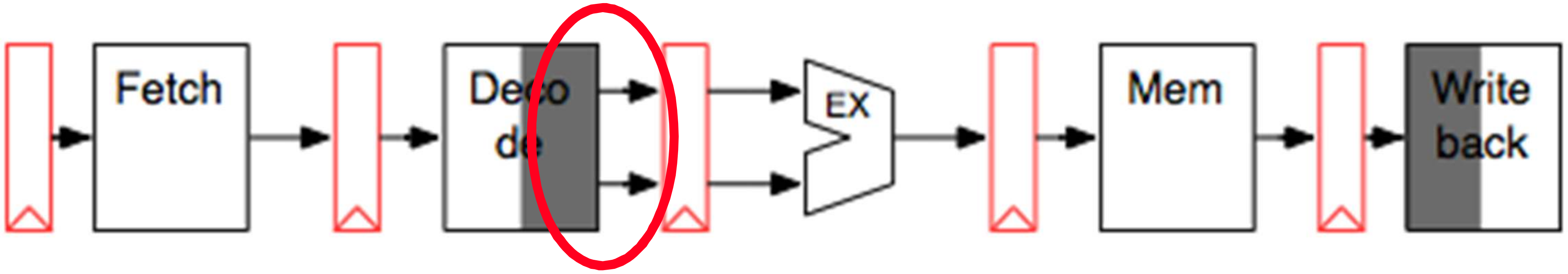




Fixing the Ubiquitous Control Hazard

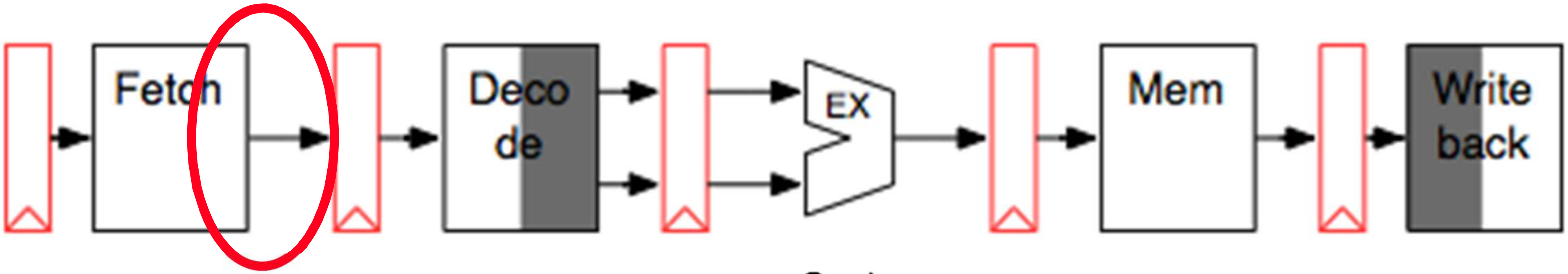
- We need to know if an instruction is a branch in the fetch stage!
- How can we accomplish this?

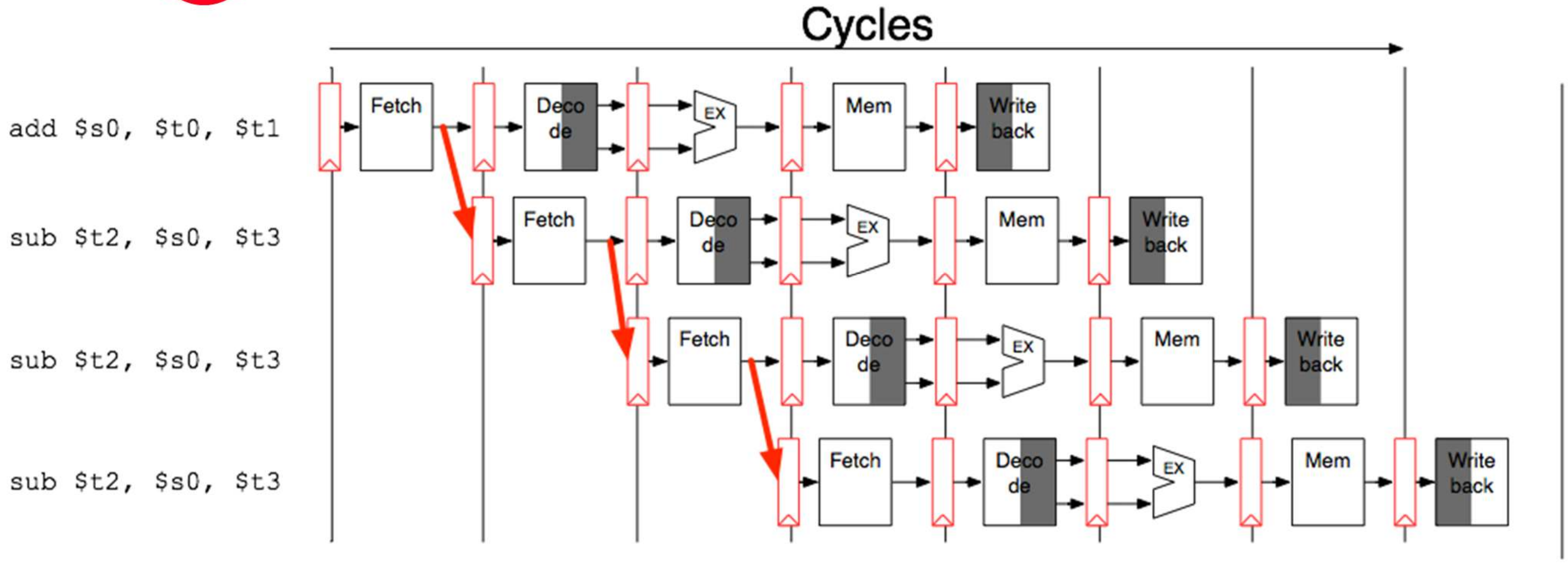
Solution: Partially decode the instruction in fetch (or even when you bring it into the I-Cache). You just need to know if it's a branch, a jump, or something else.



Computing the PC Normally

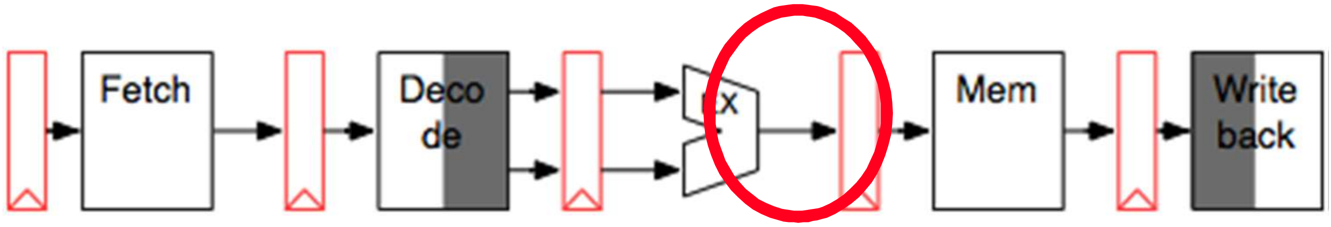
- Pre-decode in the fetch unit.
  - PC = PC + 4
- The PC is ready for the next fetch cycle.

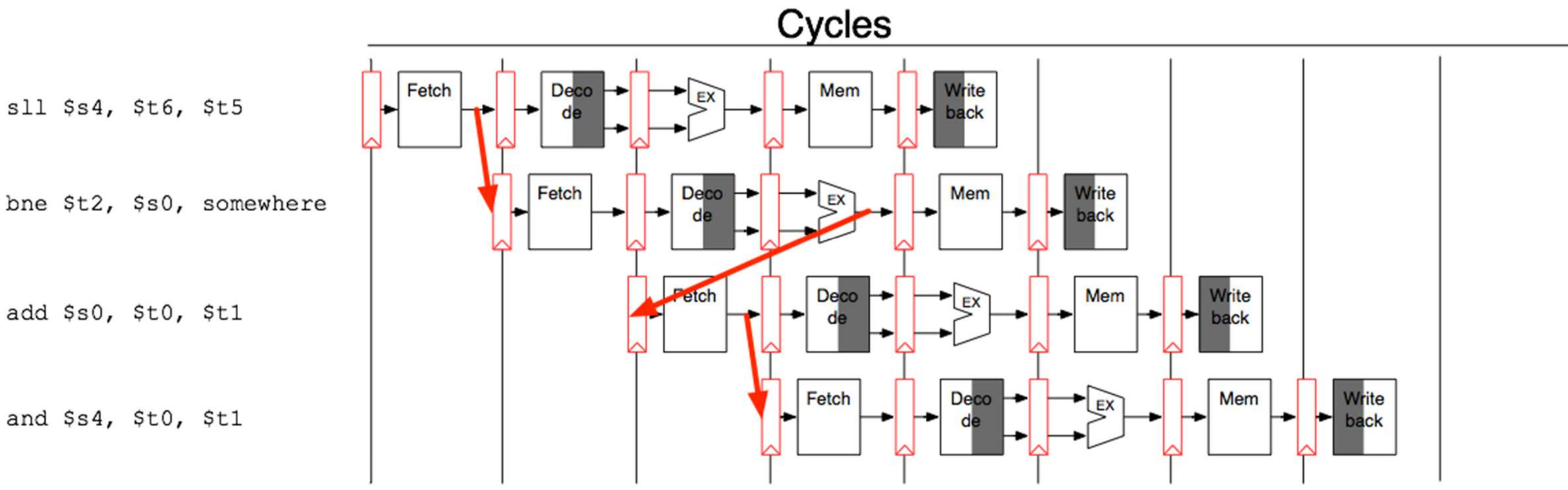




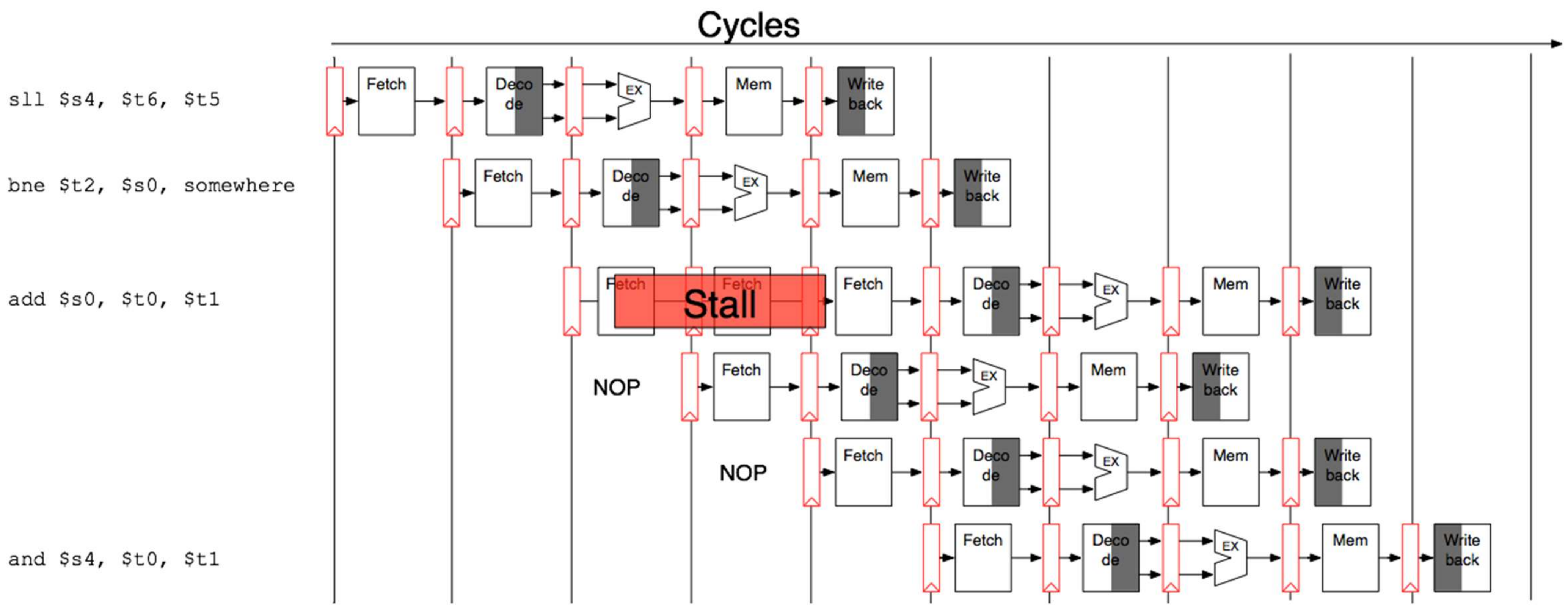
Computing the PC for Branches

- Branch instructions
  - bne $s1, $s2, offset
  - if ($s1 != $s2) { PC = PC + offset} else {PC = PC + 4;}
- When is the value ready?





Dealing with Branches: Option 0 -- stall



- What does this do to our CPI?

Option 1: The compiler

- Use “branch delay” slots.
- The next N instructions after a branch are *always* executed
- How big is N?
  - For jumps?
  - For branches?
- Good
  - Simple hardware
- Bad
  - N cannot change.

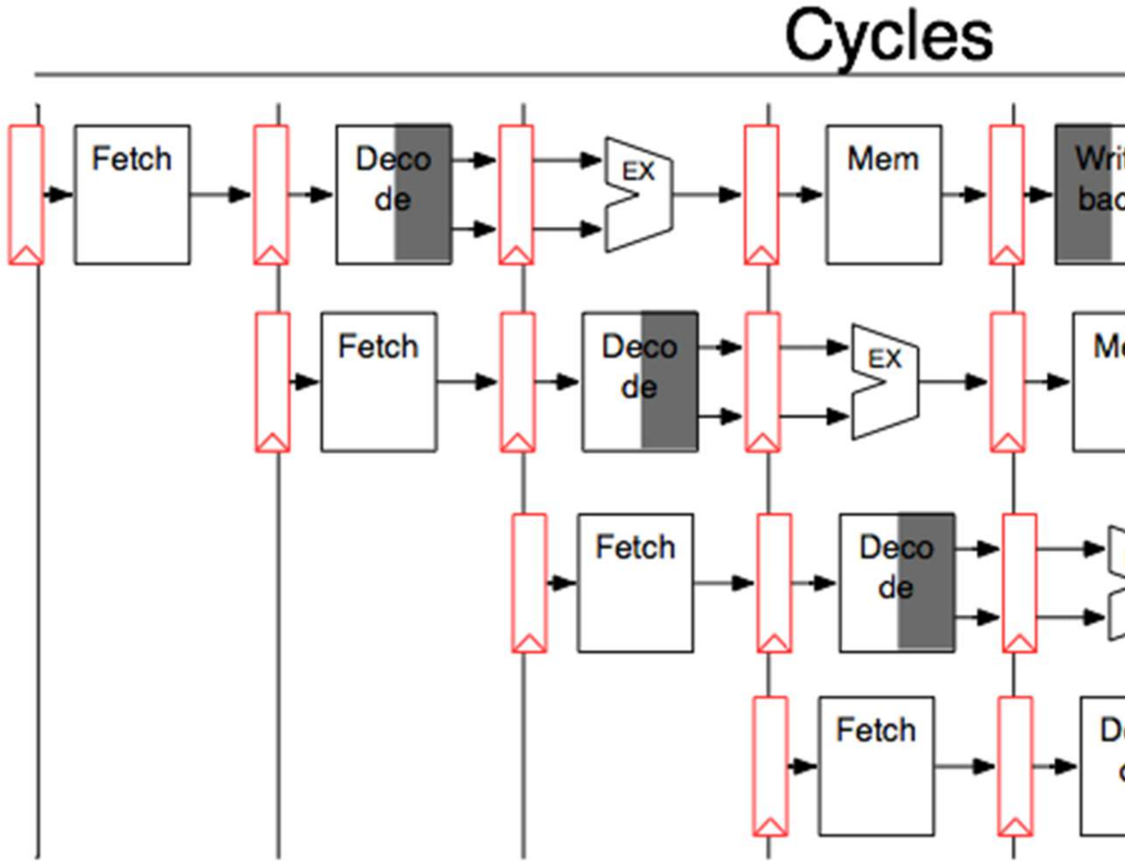
Delay slots.

Taken

```
bne $t2, $s0, somewhere
```

Branch Delay

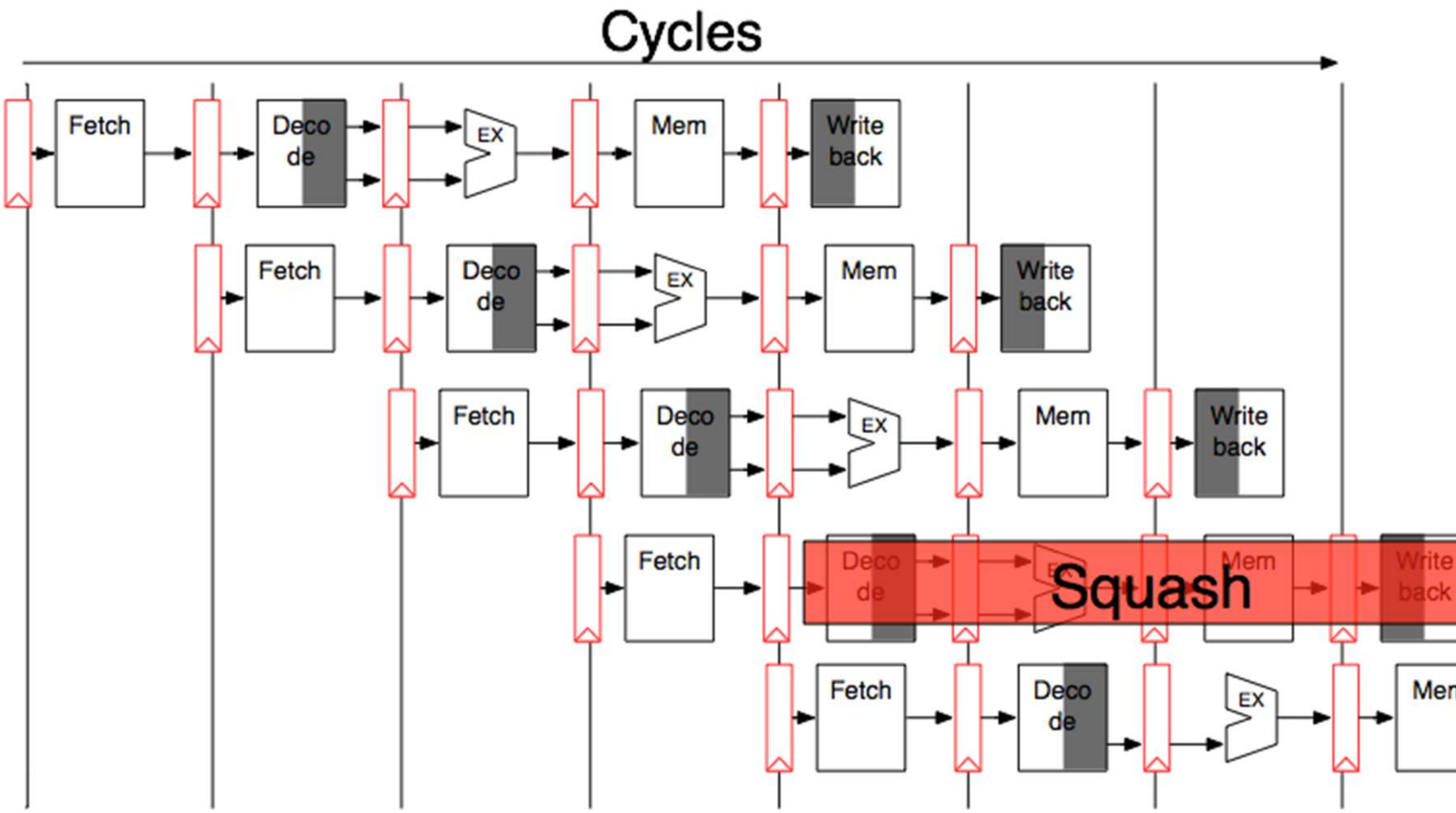
```
add $t2, $s4, $t1

add $s0, $t0, $t1

 ...
 somewhere:
 sub $t2, $s0, $t3
```



Option 2: Simple Prediction

- Can a processor tell the future?
- For non-taken branches, the new PC is ready immediately.
- Let's just assume the branch is not taken
- Also called "branch prediction" or "control speculation"
- What if we are wrong?
- Branch prediction vocabulary
  - Prediction -- a guess about whether a branch will be taken or not taken
  - Misprediction -- a prediction that turns out to be incorrect.
  - Misprediction rate -- fraction of predictions that are incorrect.

Predict Not-taken



```
bne $t2, $s0, foo

add $s4, $t6, $t5

bne $t2, $s4, else

and $s4, $t0, $t1

...
else:
sub $t2, $s0, $t3
```

- We start the add, and then, when we discover the branch outcome, we *squash* it.
  - Also called “flushing the pipeline”

Simple “static” Prediction

- “static” means before run time
- Many prediction schemes are possible
- Predict taken
  - Pros? Loops are commons
- Predict not-taken
  - Pros? Not all branches are for loops.
- Backward taken/Forward not taken
  - The best of both worlds!
  - Most loops have have a backward branch at the bottom, those will predict taken
  - Others (non-loop) branches will be not-taken.

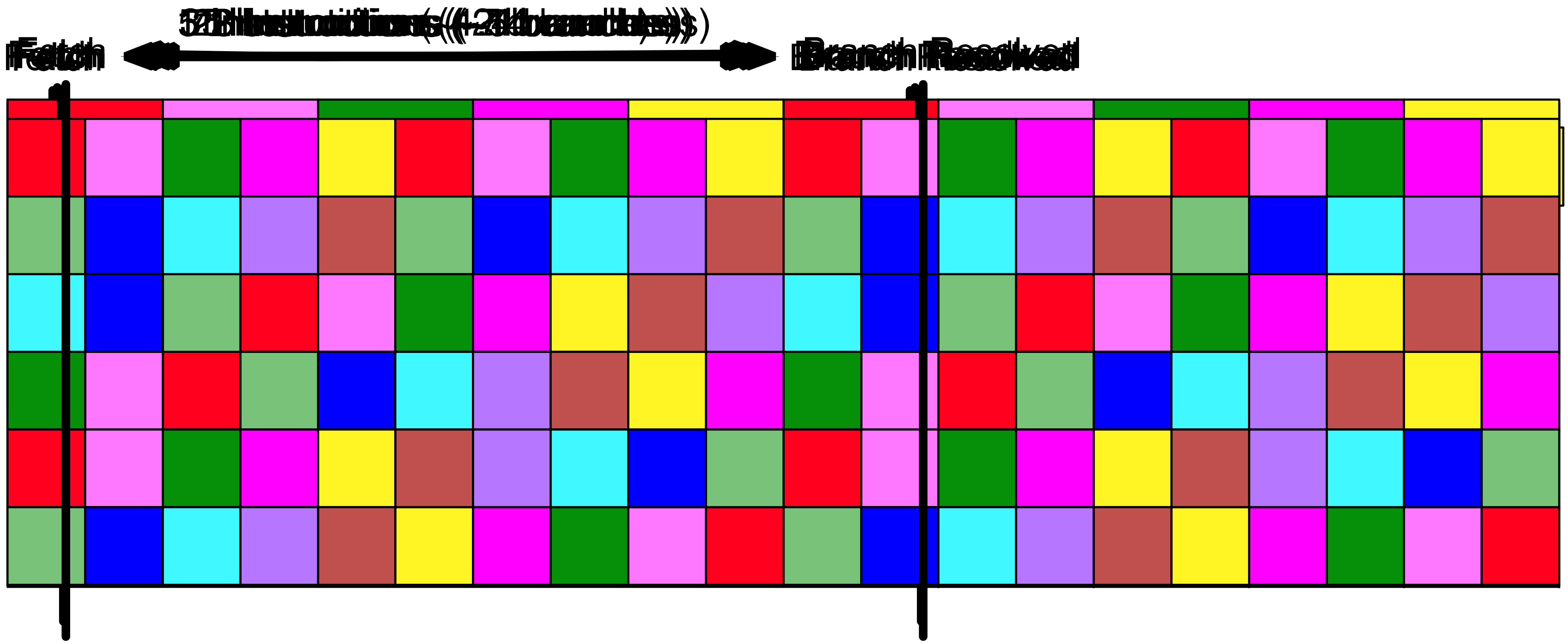
The Branch Misprediction Penalty

- The number of cycle between fetch and branch resolution is called the “branch delay penalty”
  - It is the number of instruction that get flushed on a misprediction.
  - It is the number of extra cycles the branch gets charged (i.e., the CPI for mispredicted branches goes up by the penalty for)

The Importance of Pipeline depth

- There are two important parameters of the pipeline that determine the impact of branches on performance
  - Branch decode time -- how many cycles does it take to identify a branch (in our case, this is less than 1)
  - Branch resolution time -- cycles until the real branch outcome is known (in our case, this is 2 cycles)

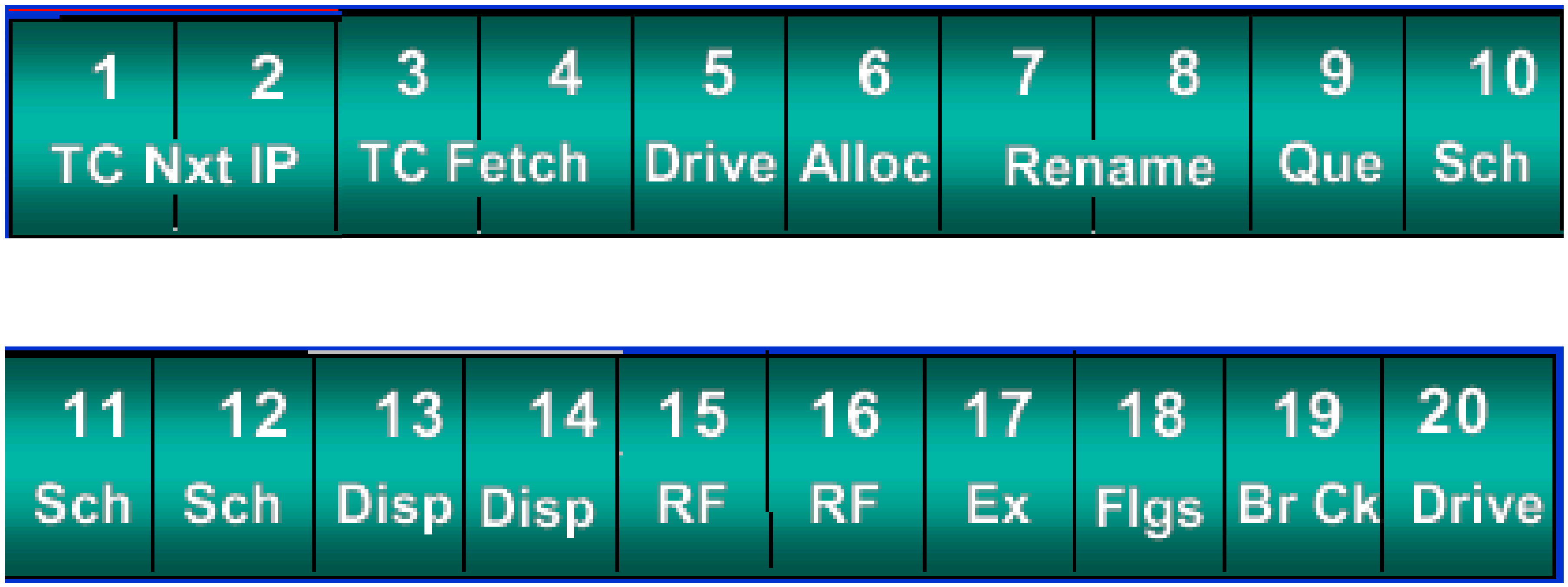
BTFNT is not nearly good enough!



14 branches @ 80% accuracy = .8^14 =4.3%
14 branches @ 90% accuracy = .9^14 =22%
14 branches @ 95% accuracy = .95^14 =49%
14 branches @ 99% accuracy = .99^14 =86%

Pentium 4 pipeline

- Branches take 19 cycles to resolve
- Identifying a branch takes 4 cycles.
- Stalling is not an option.
- 80% branch prediction accuracy is also not an option.
- Not quite as bad now, but BP is still very important.
- Wait, it gets worse!!!!



Dynamic Branch Prediction

- Long pipes demand higher accuracy than static schemes can deliver.
- Instead of making the the guess once (i.e. statically), make it every time we see the branch.
- Many ways to predict dynamically
  - We will focus on predicting future behavior based on past behavior

Predictable control

- Use previous branch behavior to predict future branch behavior.
- When is branch behavior predictable?

Predictable control

- Use previous branch behavior to predict future branch behavior.
- When is branch behavior predictable?
  - Loops -- for(i = 0; i < 10; i++) {} 9 taken branches, 1 not-taken branch. All 10 are pretty predictable.
  - Run-time constants
    - Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}.
    - The branch is always taken or not taken.
  - Corollated control
    - a = 10; b = <something usually larger than a >
    - if (a > 10) {}
    - if (b > 10) {}
  - Function calls
    - LibraryFunction() -- Converts to a jr (jump register) instruction, but it's always the same.
    - BaseClass * t;   // t is usually a of sub class, SubClass
    - t->SomeVirtualFunction() // will usually call the same function

Dynamic Predictor 1: The Simplest Thing

- Predict that this branch will go the same way as the previous branch did.
- Pros?

Dead simple. Keep a bit in the fetch stage that is the direction of the last branch. Works ok for simple loops. The compiler might be able to arrange things to make it work better.

- Cons?

An unpredictable branch in a loop will mess everything up. It can’t tell the difference between branches.

Accuracy of 1-bit counter

- Consider the following code:

```
i = 0;
do {
    if( i % 3 != 0)   // Branch Y, taken if i % 3 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch X
```

What is the prediction accuracy of branch Y using 1-bit predictors (if all counters start with 0/not taken). Choose the most close one.

A. 0%
[✓ B. 33%]
C. 67%
D. 100%

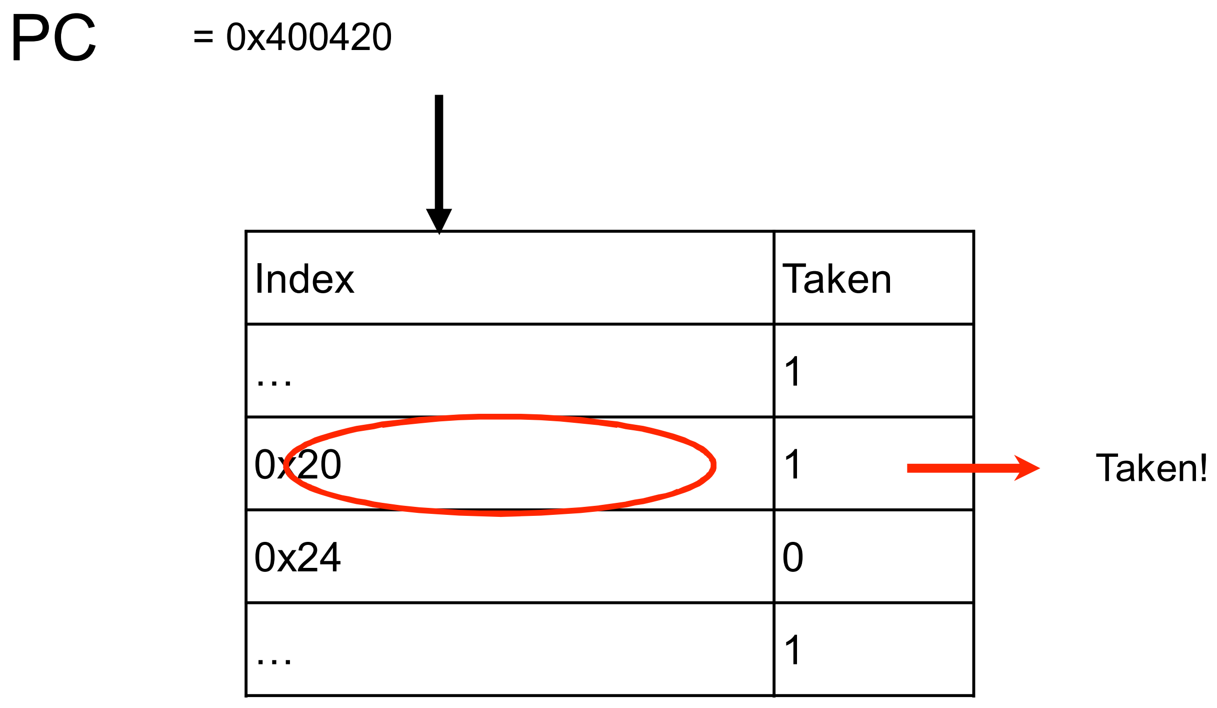
i	branch	Last branch (x) bit	Actual (y)
0	Y	T	T
1	Y	T	NT
2	Y	T	NT
3	Y	T	T
4	Y	T	NT
5	Y	T	NT
6	Y	T	T
7	Y	T	NT

The 1-bit Predictor

- Predict this branch will go the same way as the result of the last time this branch executed
  - 1 for taken, 0 for not takens

How big should this table be?

What about conflicts?



Index	Taken
…	1
0x20	1
0x24	0
…	1

Simple 1-bit Predictor

Accuracy of 1-bit counter

- Consider the following code:

```
i = 0;
do {
    if( i % 3 != 0)   // Branch Y, taken if i % 3 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch X
```

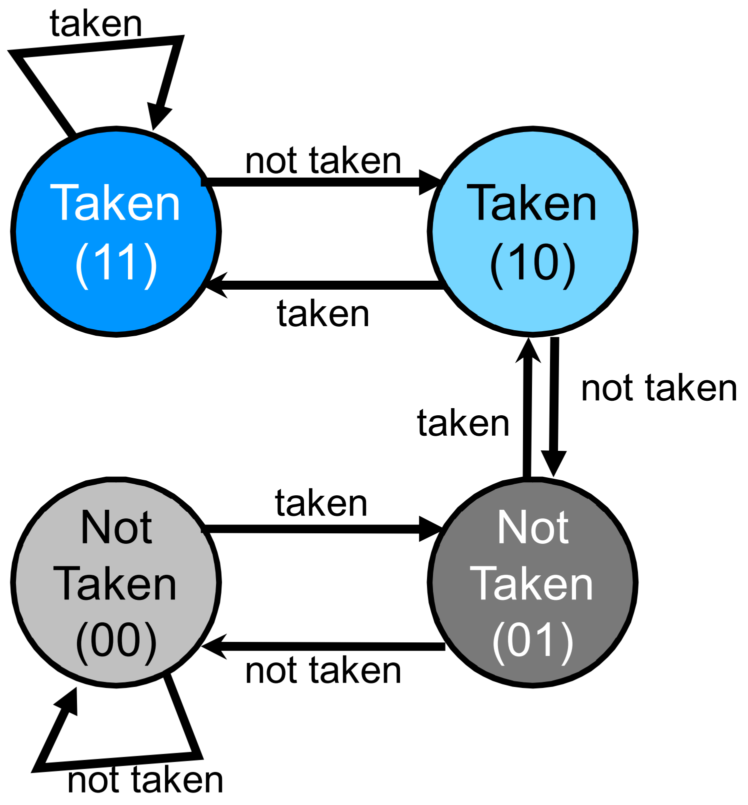
What is the prediction accuracy of branch Y using 1-bit predictors (if all counters start with 0/not taken). Choose the most close one.
Assume unlimited BTB entries.

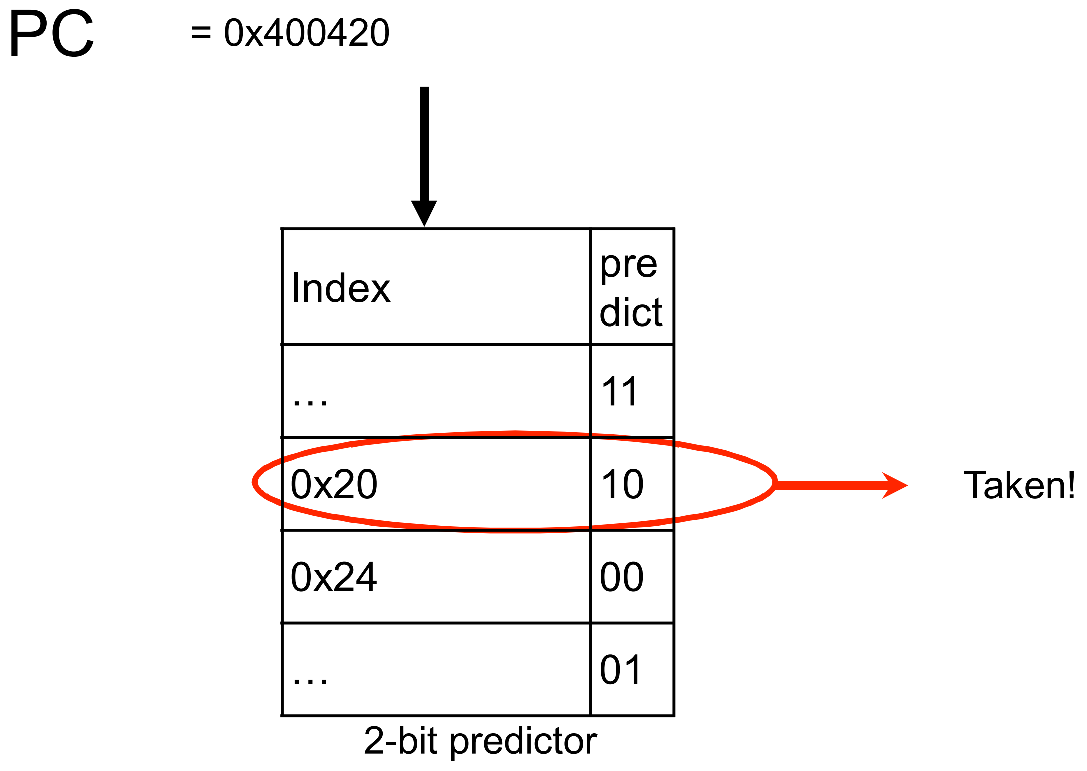
A. 0%
[✓ B. 33%]
C. 67%
D. 100%

i	branch	predict	actual
0	Y	NT	T
1	Y	T	NT
2	Y	NT	NT
3	Y	NT	T
4	Y	T	NT
5	Y	NT	NT
6	Y	NT	T
7	Y	T	NT

2-bit counter

- A 2-bit counter for each branch
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4



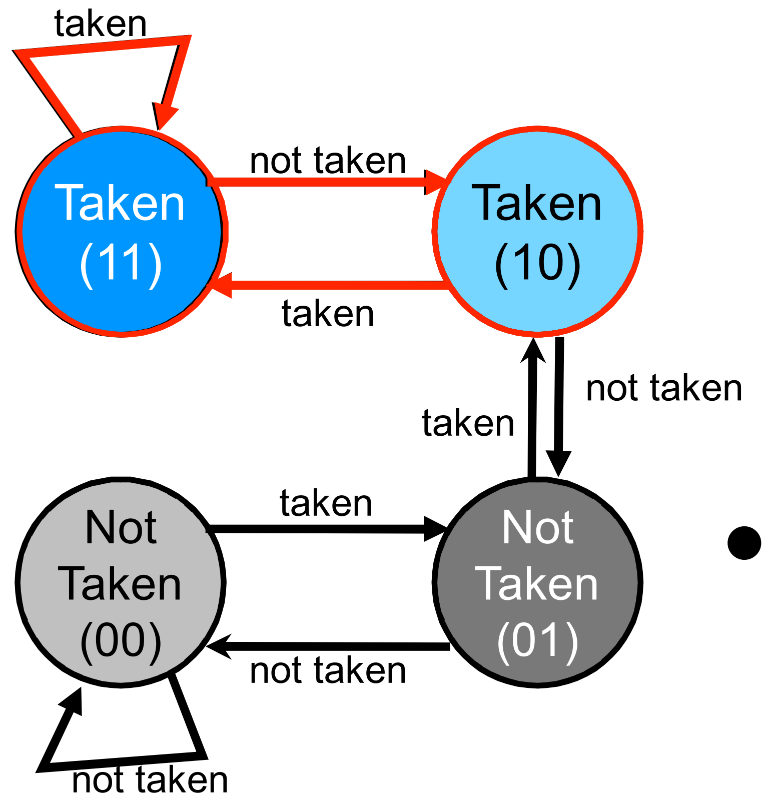


Index	pre dict
…	11
0x20	10
0x24	00
…	01

2-bit predictor

Performance of 2-bit counter

- 2-bit state machine for each branch



```
for(i = 0; i < 10; i++)
{
        sum += a[i];
}
```

90% prediction rate!

- Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?
  - 1+20%*(1-90%)*2 = 1.04

Accuracy of 2-bit counter

- Consider the following code:

```
i = 0;
do {
    if( i % 3 != 0)  // Branch Y, taken if i % 3 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch X
```

What is the prediction accuracy of branch Y using 2-bit predictors (if all counters start with 00). Choose the closest one. Assume unlimited BTB entries.

A. 0%
B. 33%
C. 67%
D. 100%

i	branch	state	predict	actual
0	Y	00	NT	T
1	Y	01	NT	NT
2	Y	00	NT	NT
3	Y	00	NT	T
4	Y	01	NT	NT
5	Y	00	NT	NT
6	Y	00	NT	T
7	Y	01	NT	NT

Make the prediction better

- Consider the following code:

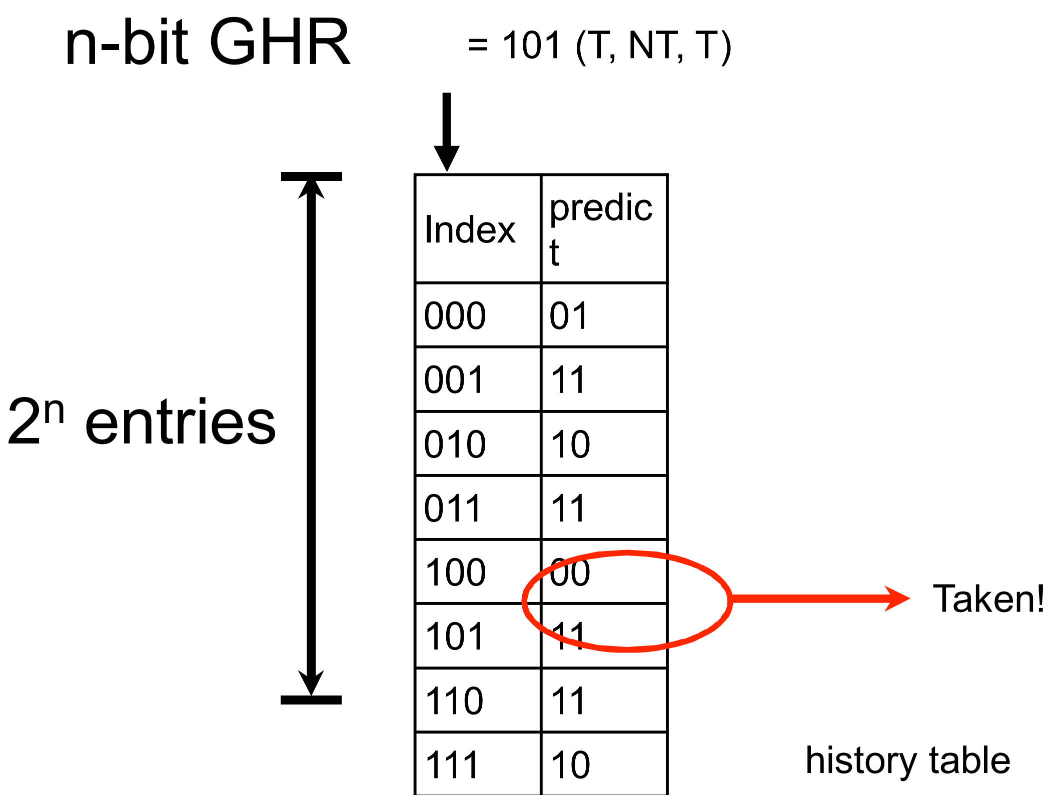
```
i = 0;
do {
    if( i % 3 != 0)   // Branch Y,
taken if i % 3 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch X
```

Can we capture the pattern?

i	branch	result
0	Y	T
0	X	T
1	Y	NT
1	X	T
2	Y	NT
2	X	T
3	Y	T
3	X	T
4	Y	NT
4	X	T
5	Y	NT
5	X	T
6	Y	T
6	X	T
7	Y	NT

Predict using history

- Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.
- Each entry in the history table has its own counter.



Index	predic t
000	01
001	11
010	10
011	11
100	00
101	11
110	11
111	10

Performance of global history predictor

- Consider the following code:

```
i = 0;
do {
    if( i % 3 != 0)   // Branch Y,
taken if i % 3 == 0
       a[i] *= 2;
    a[i] += i;
// Branch Y
} while ( ++i < 100) // Branch X
```

Assume that we start with a 4-bit GHR= 0, all counters are 10.

i	?	GHR	BHT	prediction	actual	New BHT
0	Y	0000	10	T	T	11
0	X	0001	10	T	T	11
1	Y	0011	10	T	NT	01
1	X	0110	10	T	T	11
2	Y	1101	10	T	NT	01
2	X	1010	10	T	T	11
3	Y	0101	10	T	T	11
3	X	1011	10	T	T	11
4	Y	0111	10	T	NT	01
4	X	1110	10	T	T	11
5	Y	1101	01	NT	NT	00
5	X	1010	11	T	T	11
6	Y	0101	11	T	T	11
6	X	1011	11	T	T	11
7	Y	0111	01	NT	NT	00
7	X	1110	11	T	T	11
8	Y	1101	00	NT	NT	00
8	X	1010	11	T	T	11
9	Y	0101	11	T	T	11
9	X	1011	11	T	T	11
10	Y	0111	00	NT	NT	00

Nearly perfect after this

Accuracy of global history predictor

- Consider the following code:

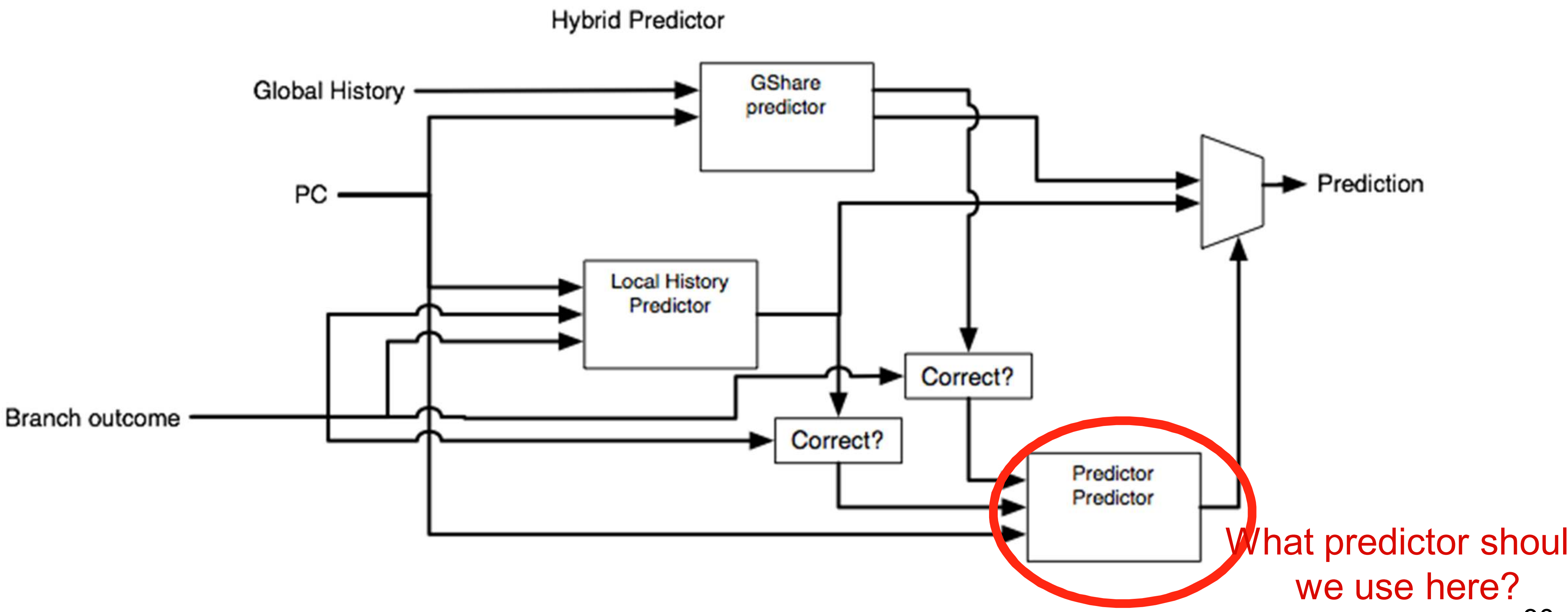
```
sum = 0;
i = 0;
do {
     if(i % 2 == 0)     // Branch Y, taken if i % 2 != 0
         sum+=a[i];
} while ( ++i < 100) // Branch X
```

Which of predictor performs the best?

A. Predict always taken
B. Predict alway not-taken
C. 1-bit predictor
D. 2-bit predictor
[✓ E. 4-bit global history with 2-bit counters]

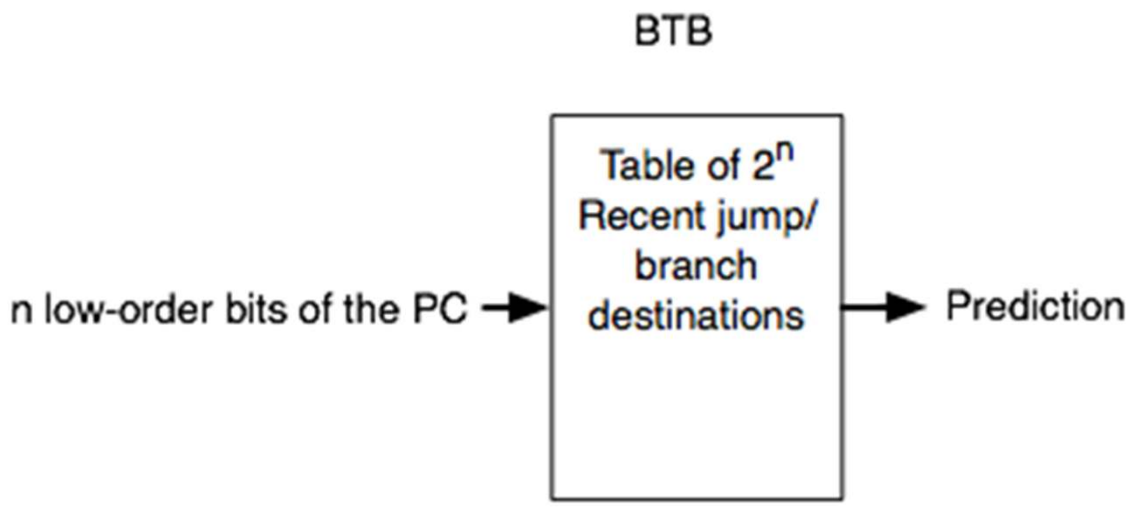
Other Ways of Identifying Branches

- How do we get the best of all possible worlds?
- Build them all, and have a predictor to decide which one to use on a given branch -- The Hybrid (or Tournament) Predictor
  - 2-bit predictor now has different states
  - Strongly prefer GShare, weakly prefer Gshare, weakly prefer local, strongly prefer local.



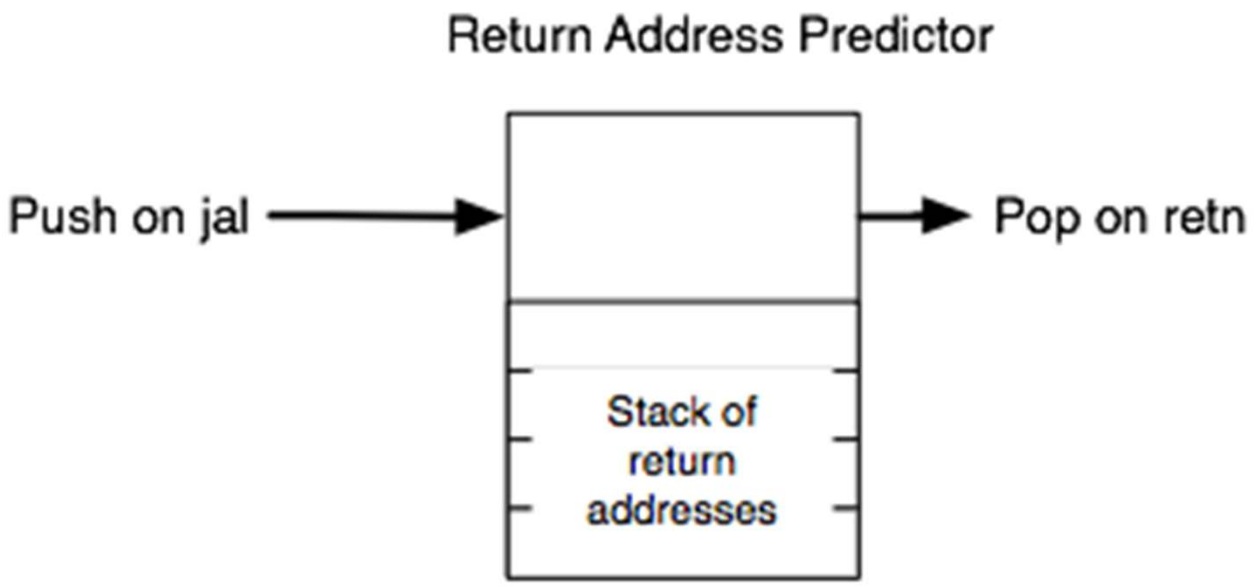
Predicting Function Invocations

- Branch Target Buffers (BTB)
  - Use a table, indexed by PC, that stores the last target of the jump.
  - When you fetch a jump, start executing at the address in the BTB.
  - Update the BTB when you find out the correct destination.
- The BTB is useful for predicting function calls and jump instructions (and some other things, as we will see shortly.)


BTB
Table of 2^n Recent jump/ branch destinations
n low-order bits of the PC
Prediction

Predicting Returns

- Function call returns are hard to predict
  - For every call site, the return address is different
  - The BTB will do a poor job, since it’s based on PC
- Instead, maintain a “return stack predictor”
  - Keep a stack of return targets
  - `jal` pushes $ra onto the stack
  - Fetch predicts the target for `retn` instruction by popping an address off the stack.
  - Doesn’t work in MIPS, because there is no return instruction.


Return Address Predictor
Push on jal
Pop on retn
Stack of return addresses