

Dynamic branch prediction

- Until now, we have focused on overcoming data hazards.
- However, control hazards contribute greatly to reduced CPI, especially as pipelines become longer
- More evident for machines that issue multiple instructions per cycle (CPI < 1)
 - Branches arrive n times more frequently in a n -issue machine.
 - Latency of resolving a branch does not decrease
 - ⇒ CPI is more significantly affected than it is for a single-issue machine

Effect of branch prediction on performance

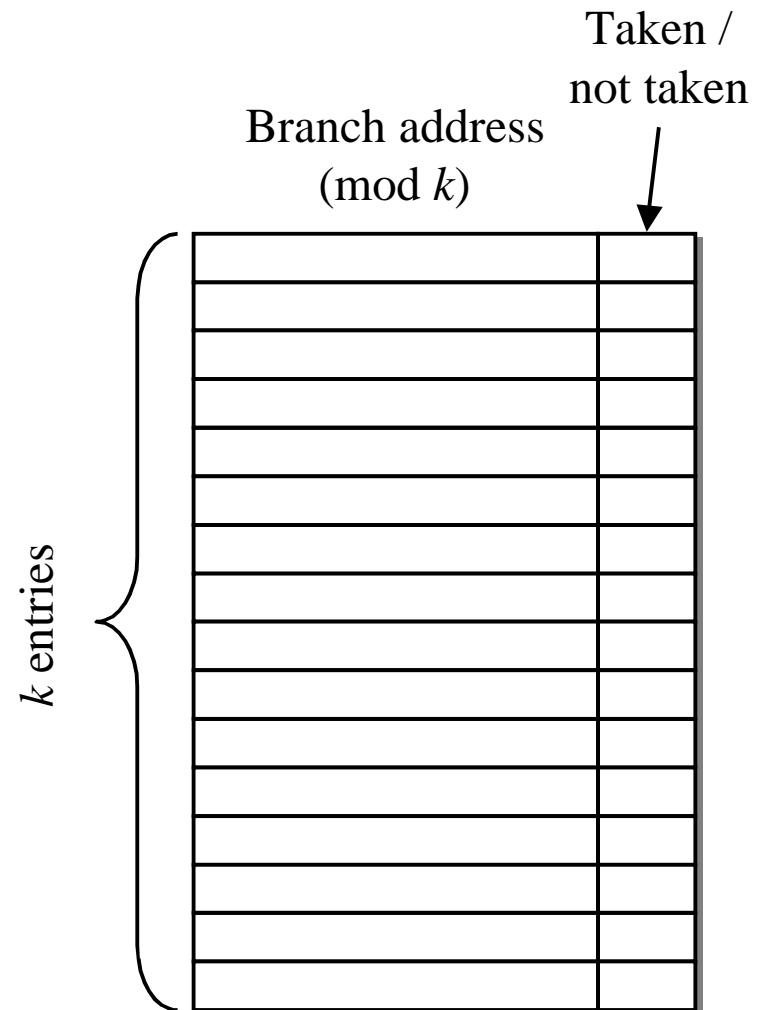
- Static vs. dynamic prediction
 - Static prediction: all decisions are made at compile time
 - ⇒ This does not allow the prediction scheme to adapt to program behavior that changes over time
- Effects of prediction on performance:
 - Accuracy
 - Accuracy of a branch prediction scheme impacts CPU performance
 - A scheme that is not accurate may make CPU performance worse than it would be without prediction
 - Latency: two orthogonal aspects to performance
 - Branch may be **taken** or **not taken**
 - Branch may be **correctly predicted** or **incorrectly predicted**
 - ⇒ Up to four different latencies for a single branch instruction

Predicting a branch's result

- The simplest thing to do with a branch is to predict whether or not it is taken
 - Helps in pipelines where the branch delay is longer than the time it takes to compute the possible target PCs
 - Most pipelines can calculate branch destination quickly!
 - By saving the decision time, the CPU can branch sooner
- This scheme does NOT help with the DLX
 - Branch decision and target PC are computed in ID, assuming there is no hazard on the register tested
 - Only helps when branch decision is calculated *after* branch target

Branch prediction buffer

- Keep a buffer (cache) indexed by the lower portion of the address of the branch instruction
 - Include bit(s) to indicate whether or not the branch was recently taken or not
 - If the prediction is incorrect, the prediction bit is inverted and stored back
- Branch direction could be incorrect because
 - Branch mispredicted
 - Instruction mismatch
 - ⇒ Either way, the worst outcome is paying the full branch latency



Improving prediction accuracy

- Sample code

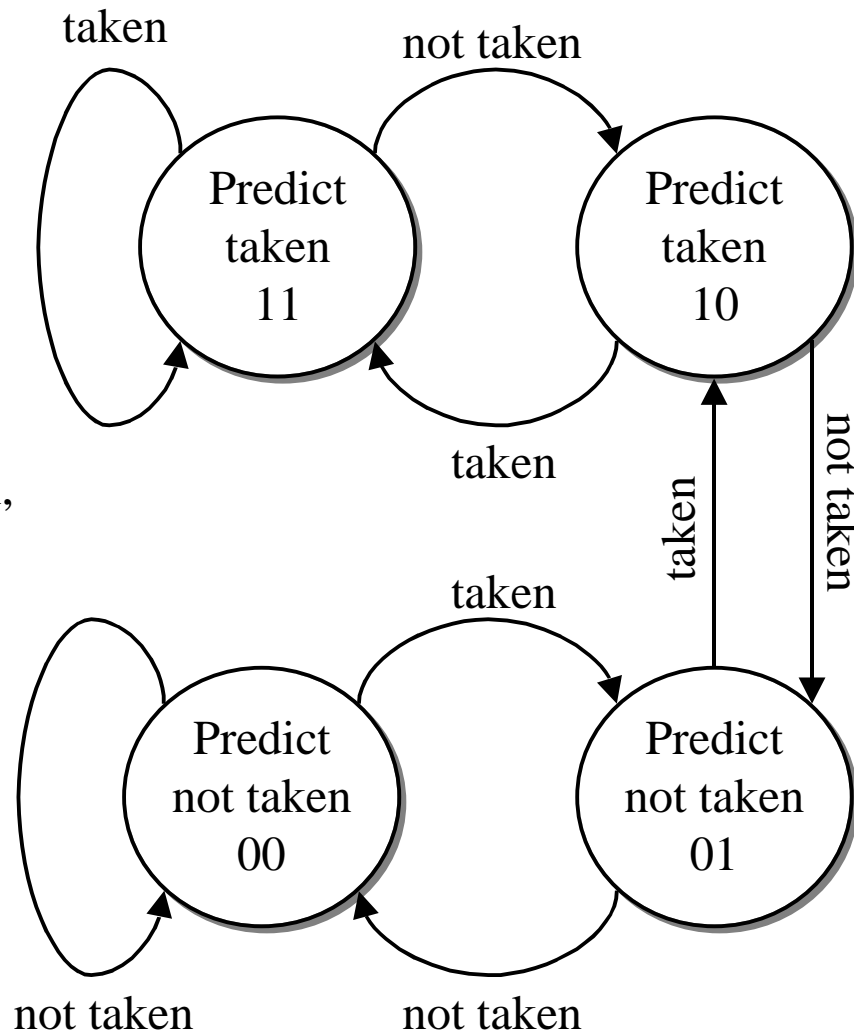
```
for (j = 0; j < 10; j++) {  
  for (k = 0; k < 10; k++) {  
    // stuff  
  } // Predict this branch  
}
```

- Problem

- If branch is *almost always* taken, this scheme will likely predict incorrectly twice
- Mispredicts when $j==0, k==10$
- Mispredicts when $j==1, k==0$

- Solution

⇒ Use 2-bit predictor!



Multi-bit predictors

- 2-bit predictor scheme
 - Allows the accuracy of the predictor to approach the taken branch frequency (i.e. 90% for highly regular branches)
 - Implements “forgiveness” for a single misprediction
- n -bit predictors
 - Keep an n -bit saturating counter for each branch
 - Increment it on branch taken
 - Decrement it on branch not taken
 - If the counter is greater than or equal to half its maximum value, predict the branch as taken
 - This can be done for any n
- $n=2$ performs almost as well as larger values for n
 - ⇒ Use $n=2$ because it requires less hardware!

Location of the branch prediction buffer

- “Special cache”
 - Accessed during IF (with the PC)
 - Prediction bits used during ID if the instruction is decoded as a branch
- Instruction cache
 - Requires more space (the instruction cache is usually much larger than the “special cache”)
 - Reduces the likelihood that “conflicts” occurs between different branches
- Accuracy of branch prediction
 - Misprediction rates range from **1%** to **18%** (using a 4K entry branch prediction buffer)
 - Static rates are around **30%** for many programs

Improving accuracy: correlated predictions

- The accuracy of our predictor is critical to exploiting more ILP
- How can we improve accuracy?
 - Increasing the size of the cache does not help (much)
 - Increasing the number of bits beyond 2 does not help (much)
- Consider the behavior of “surrounding” branches?
 - Works particularly well if there are common “paths” through code that require several branches, as in the following code:


```
if (aa == 2) // B1
    aa = 0;
if (bb == 2) // B2
    bb = 0;
if (aa != bb) ... // B3
```
 - B3 is correlated with B1 and B2
 - ⇒ If both *if* statements are **TRUE**, then (aa != bb) is **FALSE**

Correlated branch predictors

```

if (d == 0)
    d = 1;
if (d == 1)
    ...

```



Assume **d** is held in R3

```

BNEZ R3, label1 ; Branch b1
ADDI R3, R0, #1
label1:
SUBI R1, R3, #1
BNEZ R1, label2 ; Branch b2
...
label2:

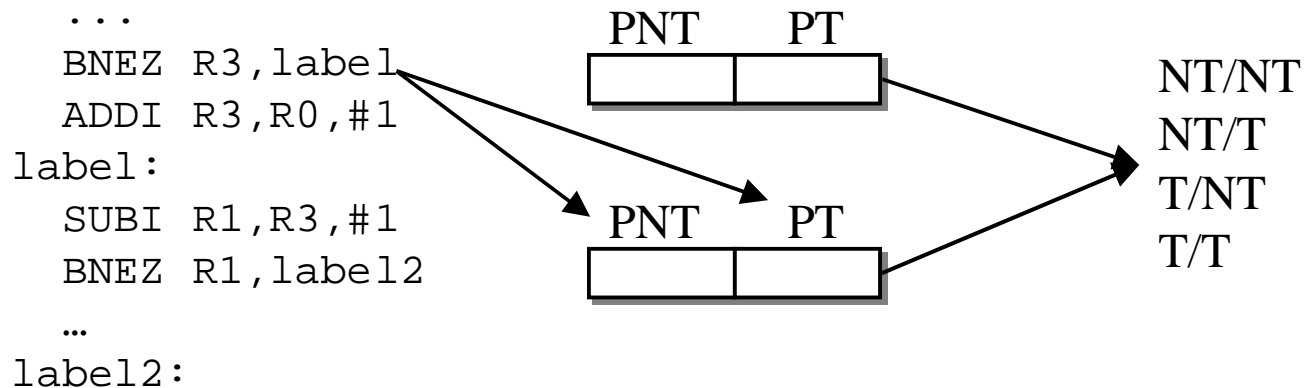
```

Initial value of d	d==0	b1	Value of d before b2	d==1	b2
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	No	Taken

- If b1 is **not taken**, then b2 will also be **not taken**
 \Rightarrow A *correlating predictor* can take advantage of this

Correlated branch prediction

- Use two-level predictors
 - Keep track of the behavior of *previous* branches
 - Use history to predict the behavior of the *current* branch
- Implement this by assigning two bits to each branch instruction
 - One bit predicts the direction of the current branch if the previous branch was not taken (PNT)
 - One bit predicts the direction of the current branch if the previous branch was taken (PT)



How do two-level predictors help?

- Assume the value of **d** alternates between 2 and 0 in a loop
 - The correct prediction of **b2** shows the advantage of correlating predictors
 - The correct prediction of **b1** is due to the choice of *d*, since there is no obvious correlation

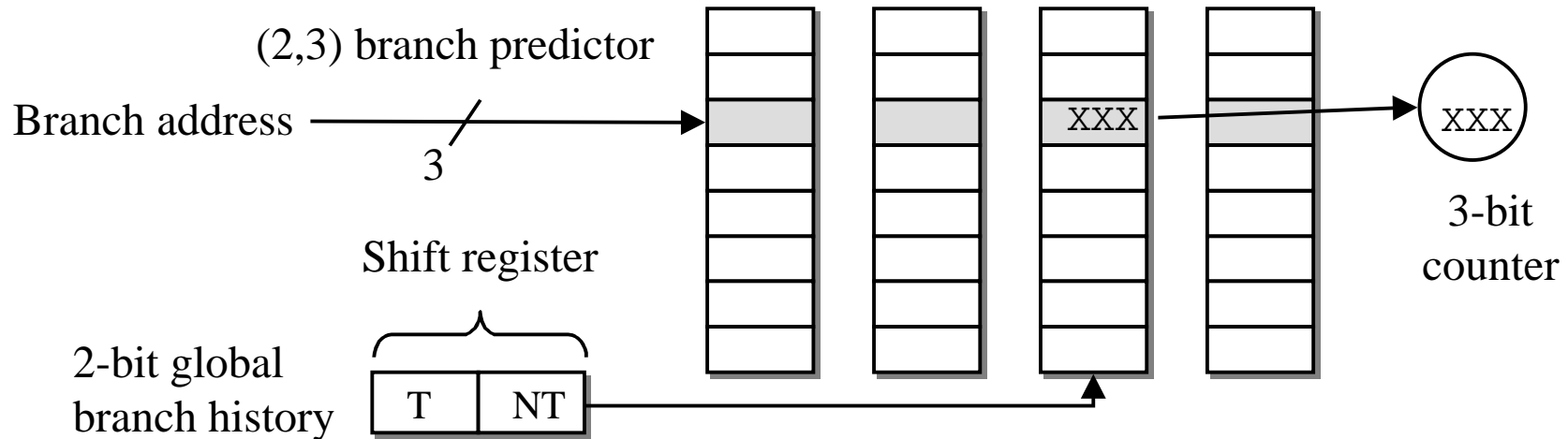
```

if (d == 0) // Branch b1
    d = 1;
if (d == 1) // Branch b2

```

	d==?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
Simple branch prediction	2	NT	<u>T</u> → T	T	NT	→ <u>T</u> → T	T
	0	T	<u>NT</u> → NT	NT	T	→ <u>NT</u> → NT	NT
	2	NT	<u>T</u> → T	T	NT	→ <u>T</u> → T	T
	0	T	<u>NT</u> → NT	NT	T	→ <u>NT</u> → NT	NT
Correlated branch prediction	2	NT/NT	<u>T</u>	T/NT	NT/NT	<u>T</u>	NT/T
	0	T/NT	NT	T/NT	NT/T	NT	NT/T
	2	T/NT	T	T/NT	NT/T	T	NT/T
	0	T/NT	NT	T/NT	NT/T	NT	NT/T

(m,n) branch predictors



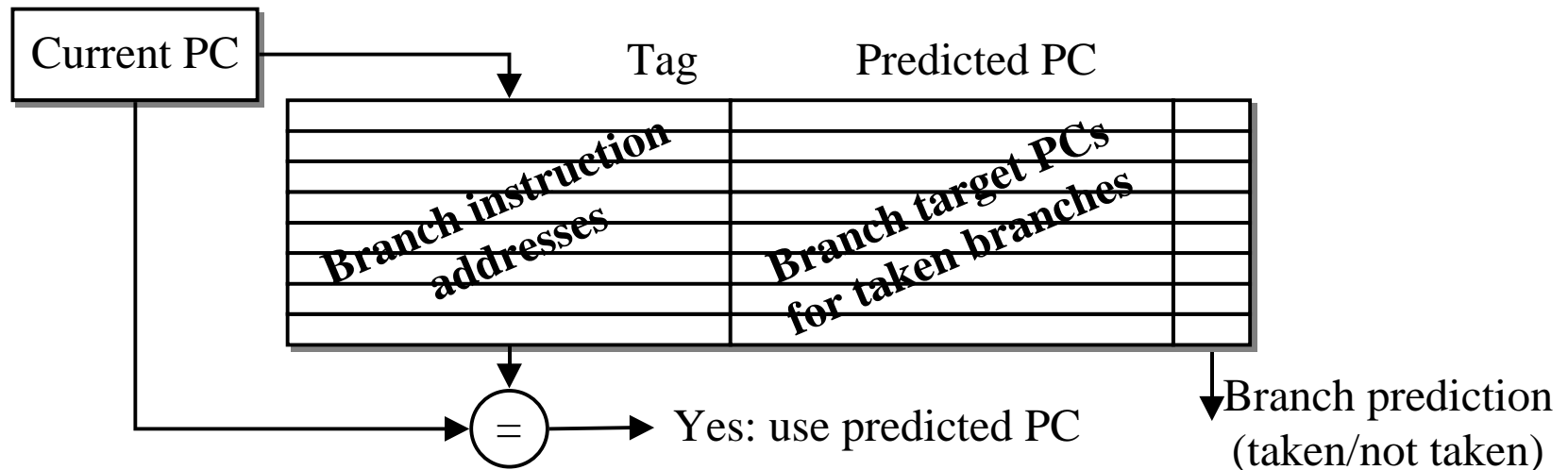
- (m,n) predictors use the behavior of the last m branches to choose from one of 2^m branch predictors, each of which is an n -bit predictor
 - \Rightarrow Results in better prediction rates than conventional n -bit prediction because it allows several “contexts”
- Global Branch History can be implemented using a shift register that shifts in the branch behavior (**not taken** or **taken**) when the branch is executed
- Since the branch prediction buffer is **NOT** a cache, there's no *guarantee* that the predictions correspond to the “correct” branch instruction

Branch target buffers

- Branch predictors help predict whether a branch is taken
 - CPU needs to know which address to fetch from ASAP in order to reduce stalls even further, ideally to 0
 - Must do this even before the CPU knows the instruction is a branch
- ⇒ Use *branch target buffer (BTB)* (also called *branch target cache*)
- A branch target buffer is very similar to a cache
 - Indexed exactly like a cache!
 - BTB must include a tag to catch collisions in the table
 - “Value” in the cache is the address of the next instruction, not the contents of the memory location

Branch target buffer operation

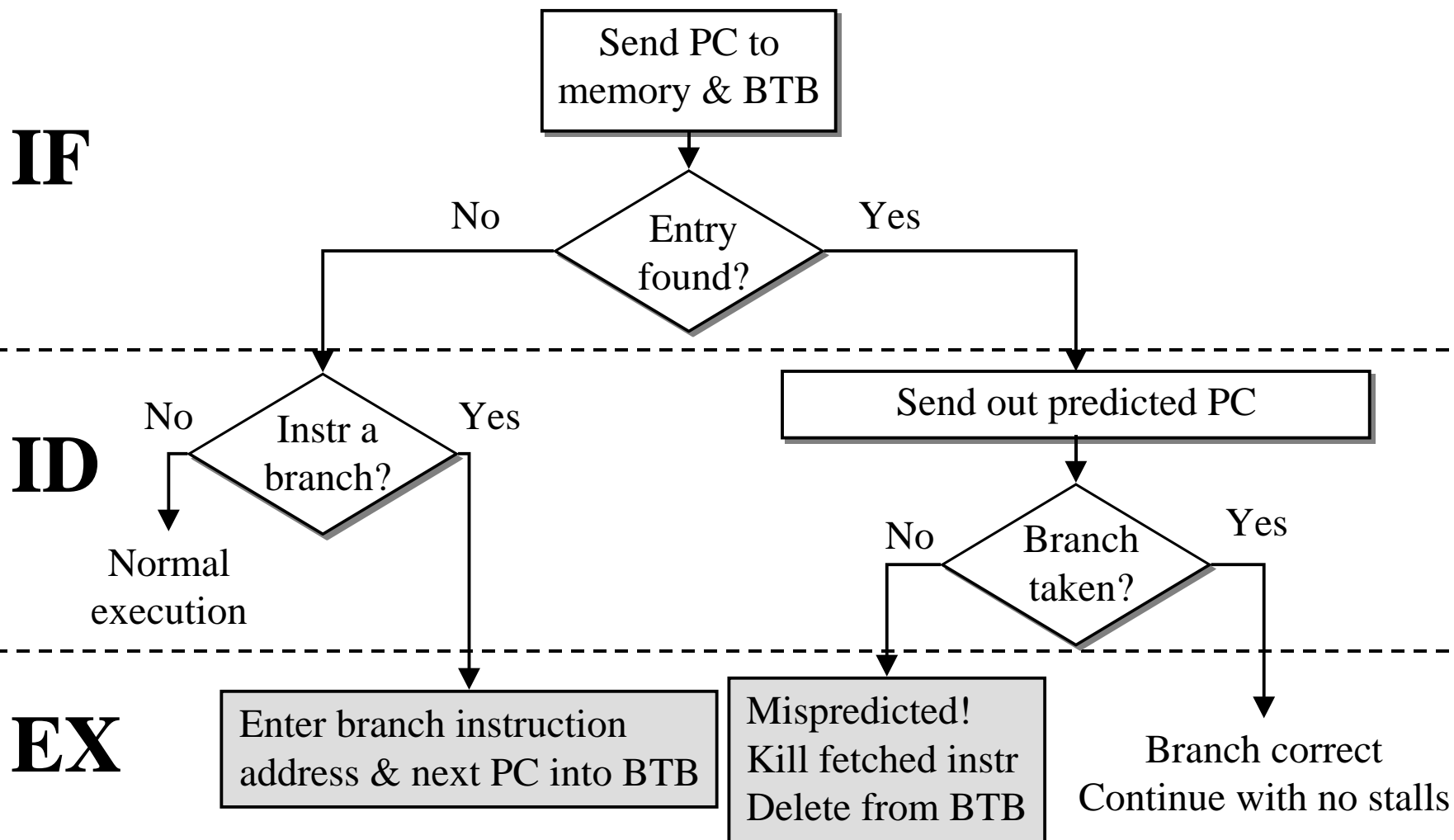
- If a hit occurs in the BTB, the CPU fetches the next instruction from the address stored in the BTB, and not $PC + 4$
 - This occurs by the end of IF!
 - CPU must compare the entire address (unlike prediction buffers)
- On an incorrect match (current instruction is NOT a branch instruction)
 - Slow things down because the predicted PC is always non-sequential by definition (and therefore, incorrect)



Adding prediction to BTBs

- Add 2 bits of prediction (the purpose of the last field in the previous figure)
 - By definition, the branch is predicted taken!
 - It has an entry in the BTB
 - Even happens if the predictor indicates that it should NOT be taken
 - In this case, it is better to have separate buffers for prediction and predicted PCs (which can be different sizes)
 - A “not taken” in the prediction buffer will override an entry in the BTB
- More complex prediction mechanisms such as (m,n) predictors can also be used in this way

Interaction of prediction & BTBs



Branch folding

- Instead of storing just the branch address, the BTB can store the actual instruction as well
 - Return the new instruction from the cache rather than just the new address
 - The branch “disappears” since it is replaced with the instruction given by its target address
 - ⇒ The branch instruction does NOT require any execution cycles!
- If it’s a conditional branch, we will still have to make sure the condition is satisfied
- Branch folding works well for
 - Unconditional branches
 - Conditional branches where the condition is easy to test (including condition codes)

Limits to branch prediction

- Misprediction rate: limits branch prediction benefits
 - If it's too high, there's too little benefit to justify the added hardware
- Misprediction penalties: also important!
 - If these are no worse than the standard penalties for missed static prediction, dynamic prediction is a win
 - What if *dynamic* misprediction penalties are worse than *static* misprediction penalties?
 - ⇒ *Static* prediction might actually outperform *dynamic* prediction even though it has a worse misprediction rate

Multiple issue: superscalar & VLIW

- Prior techniques reduce ideal CPI to as close to 1 as possible
- To reduce CPI below 1, the CPU must be capable of issuing more than one instruction per cycle
 - Superscalar: CPU tries to issue more than one instruction per cycle to keep all of the functional units busy
 - May be limits (i.e., no more than one memory instruction per cycle, no more than one branch per cycle)
 - Use **static & dynamic** scheduling to issue as many as possible
 - VLIW: fixed number of instructions per clock cycle
 - Similar to cramming (for example) four “simple” instructions into a single 128-bit instruction (one per functional unit)
 - **Statically** scheduled by the compiler

Superscalar DLX hardware

- Ensure that there are no data and structural hazards between instructions issued together
 - The easiest way to accomplish this is to allow dual issue of one integer instruction (ALU, load/store) and one floating point instruction
- Hardware requirements
 - Instruction alignment
 - Require that instruction pairs be 64-bit aligned, and that the integer instruction be first
 - Relaxing this requirement would increase the complexity of detecting hazards and thus the cost of the hardware
 - Arithmetic units & pipelines
 - CPU must have sufficient FP hardware to support one issue/clock
⇒ Requires pipelined FP units or multiple FP units (or both)

Superscalar DLX hardware

- Interactions between integer and FP
 - FP and integer are largely independent
 - Integer instructions such as FP loads and stores as well as movement between integer and FP registers can cause problems
 - Creates contention for the FP register ports
 - Creates RAW hazards between integer FP loads/stores and FP ALU instructions
 - Handle FP register contention by adding an extra port to the FP register file for memory operations
 - ⇒ Detect the case in which an FP ALU instruction is issued in the same cycle as the load that fetches a source operand for it (RAW hazard)

Superscalar DLX data & control hazards

- Simple DLX pipeline: loads had a latency of one clock cycle
- Superscalar pipeline: the result of a load cannot be used on the same clock or the next clock cycle
- Hazards impose a penalty measured in cycles, not instructions
 - The next **3** instructions cannot use the result without a stall!
 - The same is true for branch delays
- More ambitious compiler or hardware scheduling techniques and more complex instruction decoding for branches are needed
- If the CPU is not able to get a useful instruction in both of the two slots, the CPI increases and approaches 1

Static scheduling on a superscalar DLX

```

Loop: LD    F0, 0(R1)
      ADDD  F4, F0, F2
      SD    0(R1), F4
      SUBI  R1, R1, #8
      BNEZ  R1, Loop
    
```

	Integer instruction	FP instruction	Clock cycle
Loop:	LD F0, 0(R1)	-	1
	LD F6, -8(R1)	-	2
	LD F10, -16(R1)	ADDD F4, F0, F2	3
	LD F14, -24(R1)	ADDD F8, F6, F2	4
	LD F18, -32(R1)	ADDD F12, F10, F2	5
	SD 0(R1), F4	ADDD F16, F14, F2	6
	SD -8(R1), F8	ADDD F20, F18, F2	7
	SD -16(R1), F12	-	8
	SUBI R1, R1, #40	-	9
	SD 16(R1), F16	-	10
	BNEZ R1, Loop	-	11
	SD 8(R1), F20	-	12

1 cycle latency (indicated by a dashed arrow from cycle 1 to cycle 3)

2 cycle latency (indicated by a dashed arrow from cycle 6 to cycle 8)

- Unrolled loop 5 times; average 2.4 cycles per iteration

Dynamic scheduling on a superscalar DLX

- Dynamic scheduling can improve on these results to an even greater extent
 - The CPU can dual issue instructions with dependencies and serialize them later using hazard detection logic
 - Additional hardware can reduce delays through the elimination of WAR and WAW hazards and memory disambiguation
 - Similar to Tomasulo's approach
- Dynamic scheduling
 - Allows the CPU to keep the functional units busy as often as possible
 - Permits the CPU to run well on code that was not scheduled for superscalar execution

Dynamic superscalar CPUs today

- Modern CPUs may have
 - 2+ integer ALUs
 - Load/store (memory) unit
 - Branch unit
- CPU attempts to keep each functional unit busy
 - Extensive dynamic scheduling to work around many RAW hazards
 - Integer instructions can now have RAW hazards!
 - Lots of dynamic reordering to keep the units busy
 - FP/integer conflicts often less of an issue: not much FP computation
- Branch delays are a huge problem
 - 2 cycle delay is up to 11 lost instructions for 4-way issue (3 in the same cycle, 4 each in following cycles)