Branch Hazards

or

"Which way did she go?"

Control Dependence

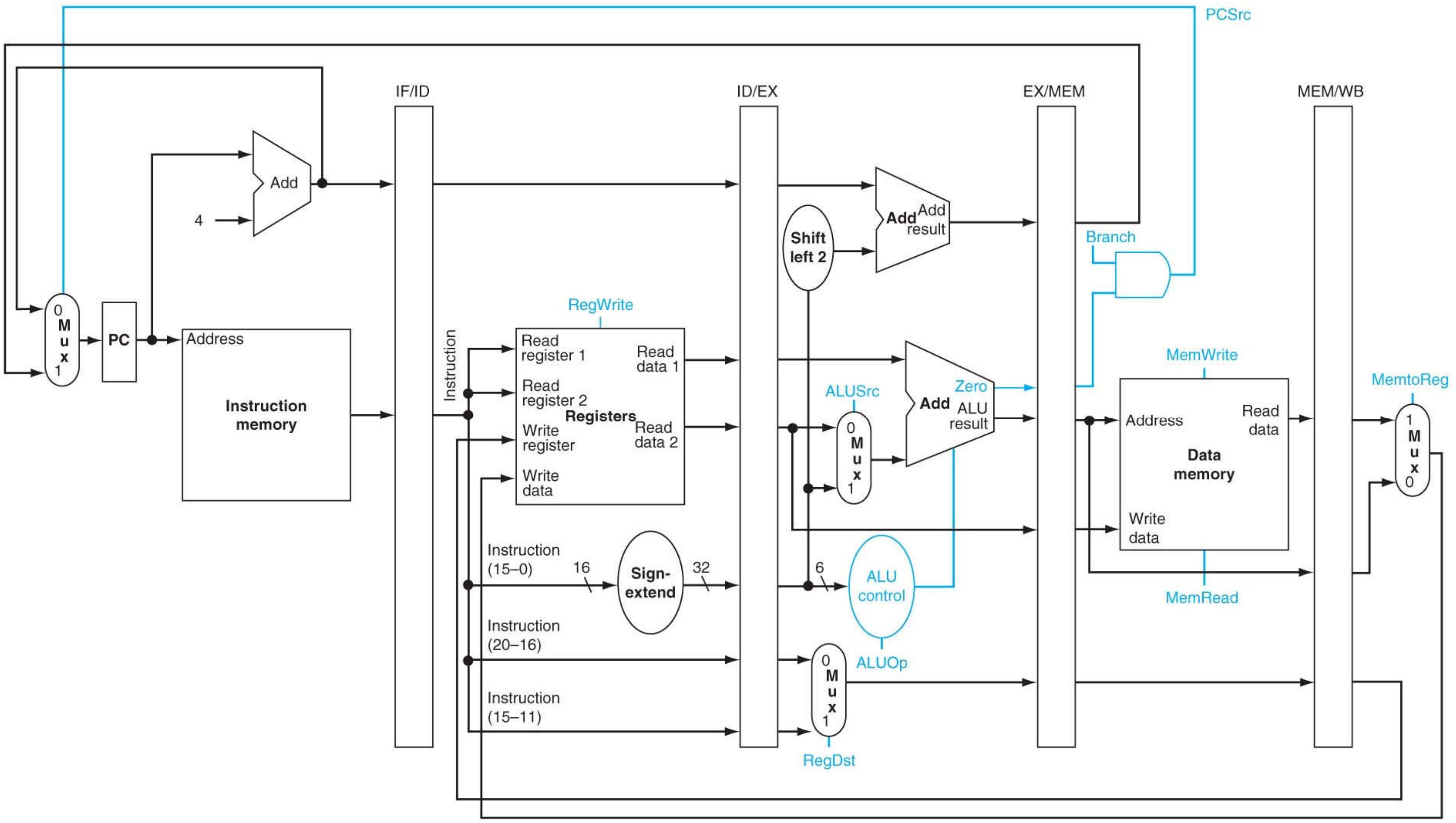
- Just as an instruction will be dependent on other instructions to provide its operands (*dependence*), it will also be dependent on other instructions to determine whether it gets executed or not (*dependence* or ____________ *dependence*).
- Control dependences are particularly critical with ______________ branches.

```
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
...                          somewhere: or $10, $5, $2
                                        add $12, $11, $9
                                        ...
```

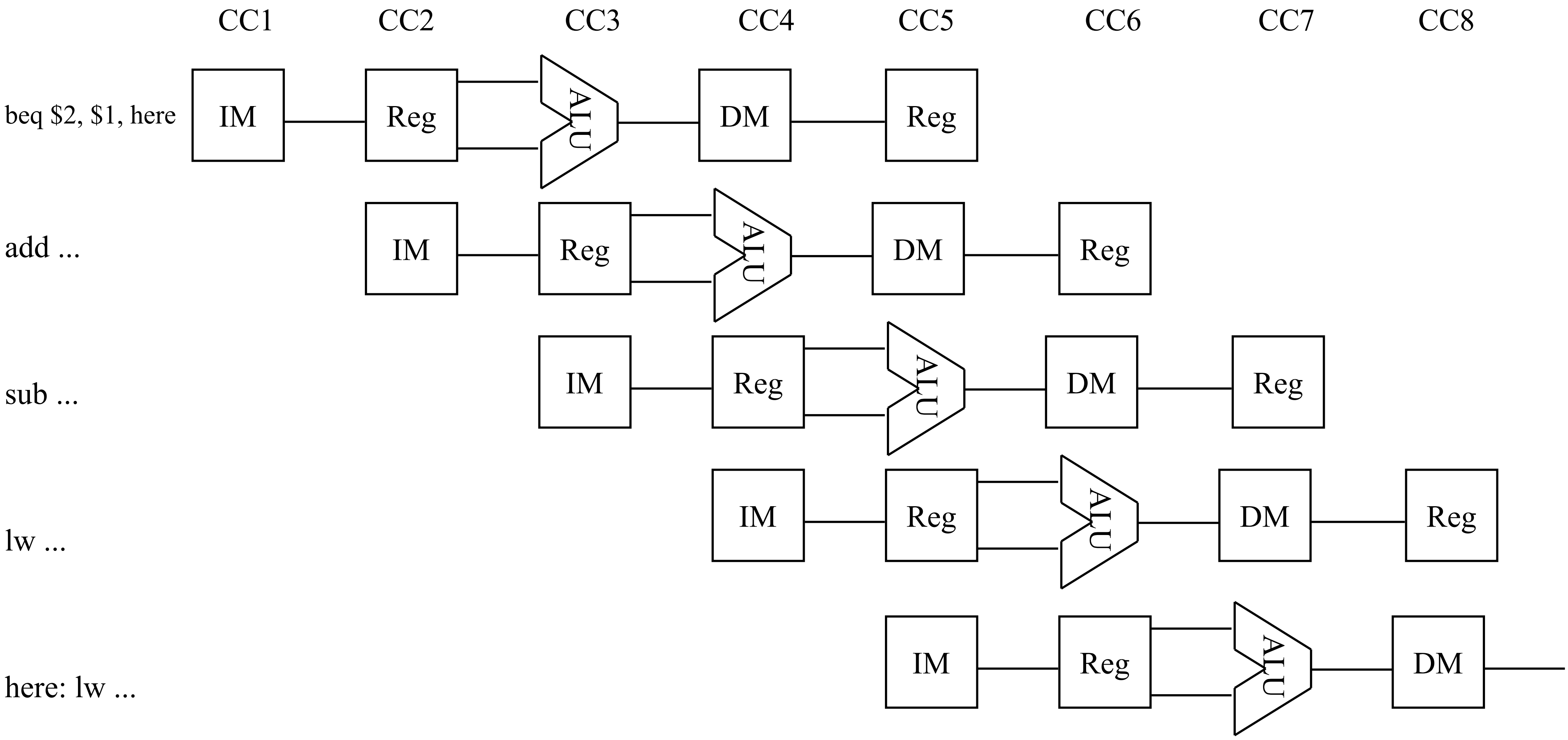
Branch Hazards

- Branch dependences can result in branch hazards (when they are too close to be handled correctly in the pipeline).

Branch Hazards



Branch Hazards



CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8

beq $2, $1, here: IM — Reg — ALU — DM — Reg

add ...: IM — Reg — ALU — DM — Reg

sub ...: IM — Reg — ALU — DM — Reg

lw ...: IM — Reg — ALU — DM — Reg

here: lw ...: IM — Reg — ALU — DM

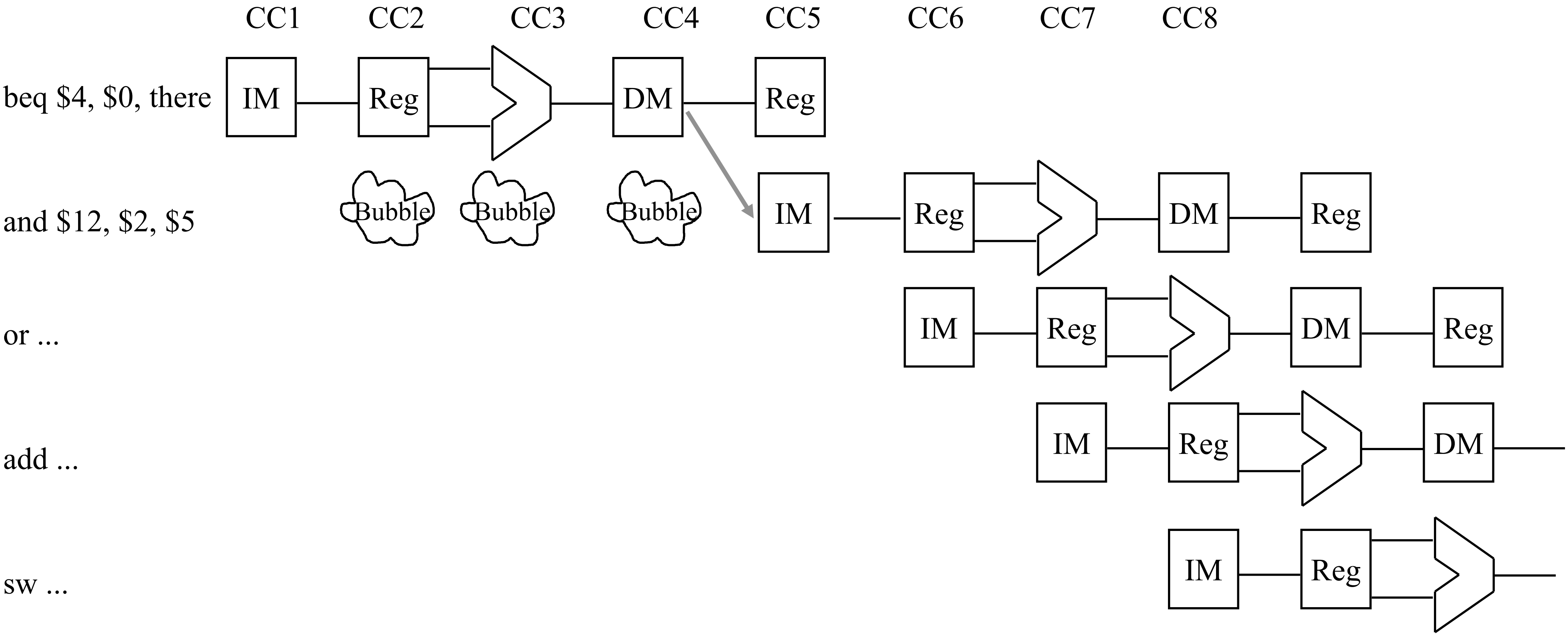
Dealing With Branch Hazards

- Hardware
  - stall until you know which direction
  - reduce hazard through earlier computation of branch direction
  - guess which direction
    - assume not taken (easiest)
    - more educated guess based on history (requires that you know it is a branch before it is even decoded!)

Dealing With Branch Hazards

- Hardware
  - stall until you know which direction
  - reduce hazard through earlier computation of branch direction
  - guess which direction
    - assume not taken (easiest)
    - more educated guess based on history (requires that you know it is a branch before it is even decoded!)
- Hardware/Software/ISA
  - nops, or instructions that get executed either way (delayed branch).

Stalling for Branch Hazards



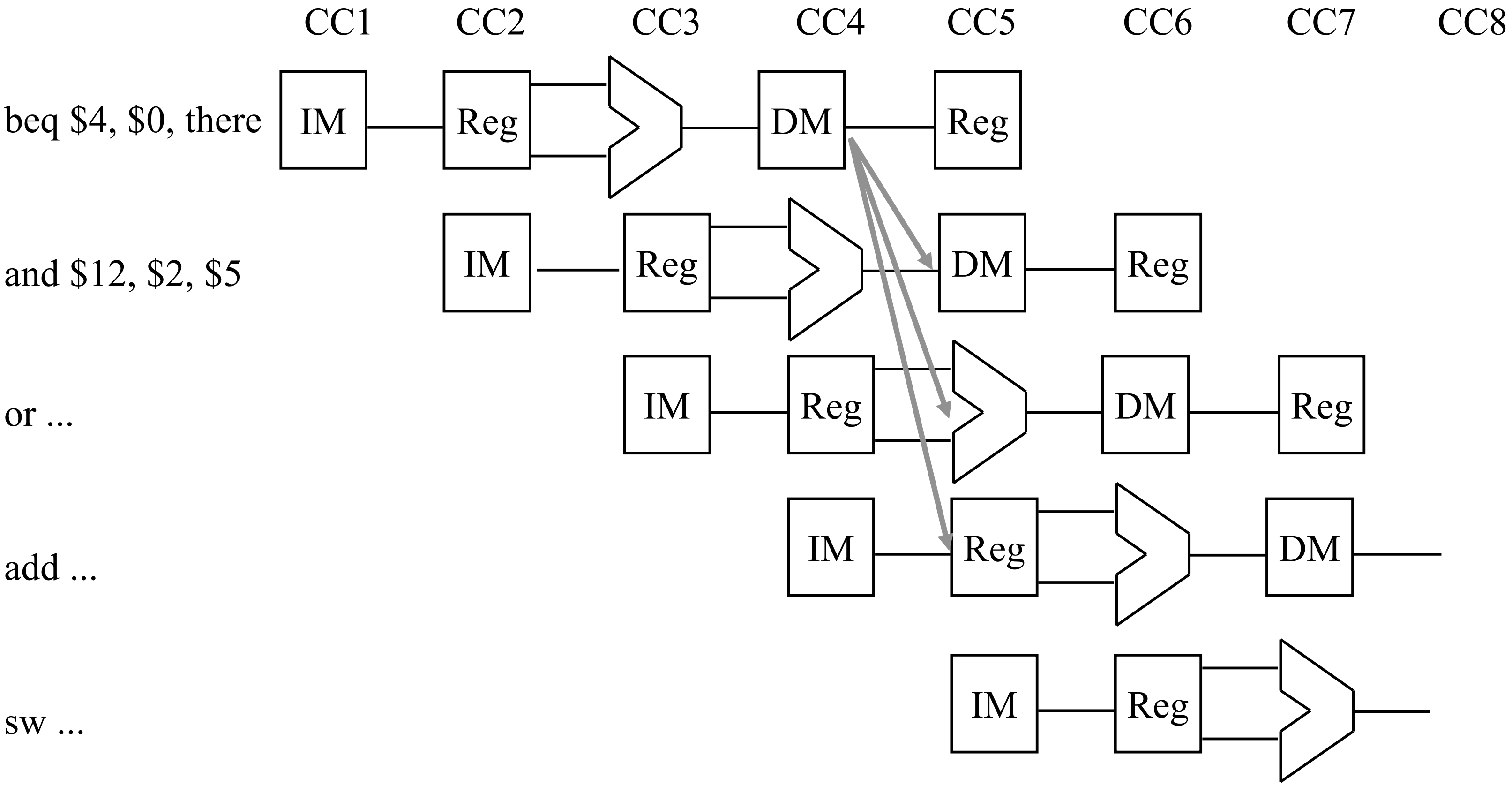
Stalling for Branch Hazards

- Seems wasteful, particularly when the branch isn't taken.
- Makes all branches cost 4 cycles.

- Also, requires you know that it's a branch before you decode it (several of our solutions have this problem)

Assume Branch *Not Taken*

- works pretty well when you’re right



CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8

beq $4, $0, there — IM, Reg, DM, Reg

and $12, $2, $5 — IM, Reg, DM, Reg

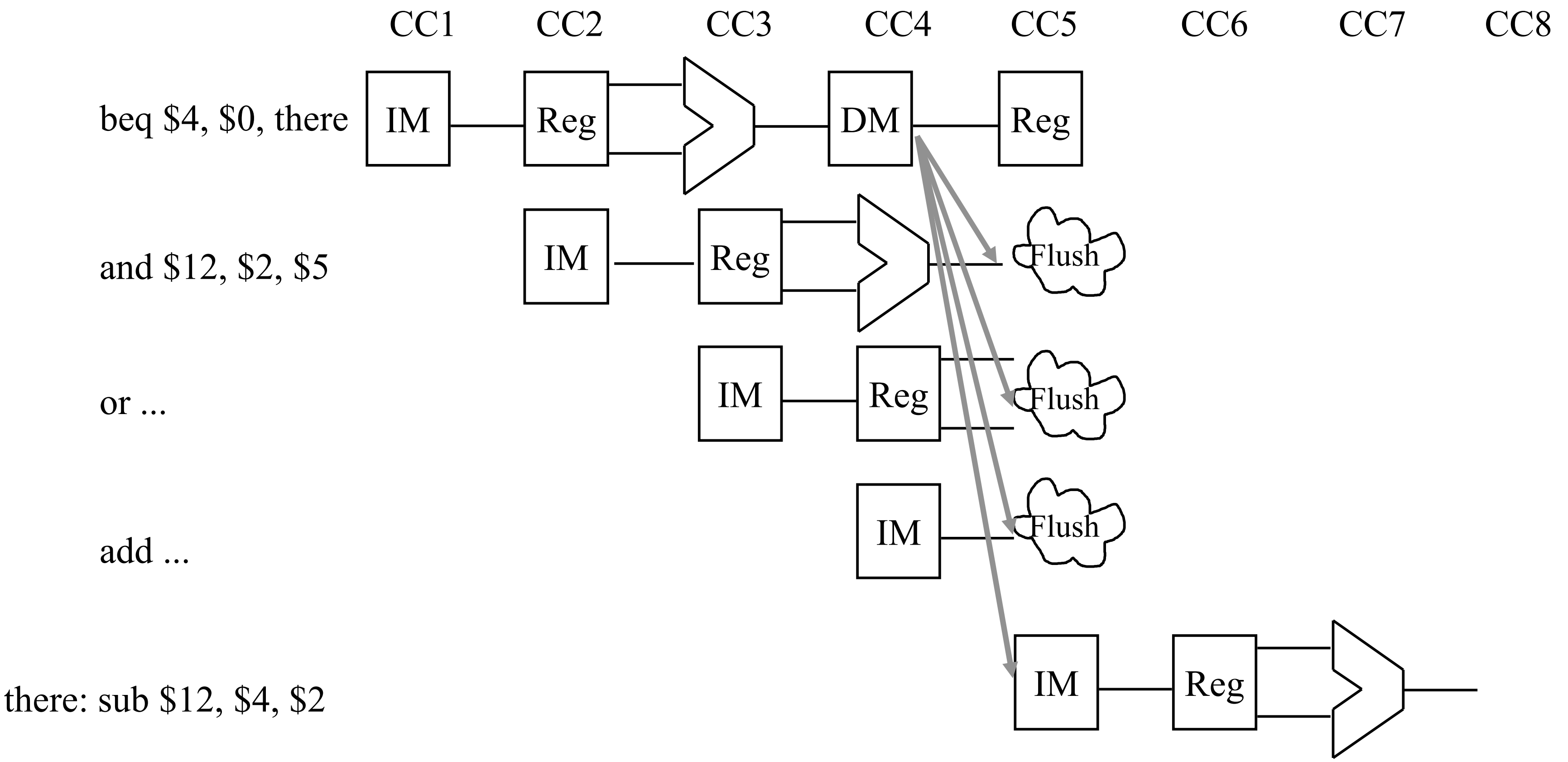
or ... — IM, Reg, DM, Reg

add ... — IM, Reg, DM

sw ... — IM, Reg

Assume Branch *Not Taken*

- same performance as stalling when you’re wrong



CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8

beq $4, $0, there

and $12, $2, $5

or ...

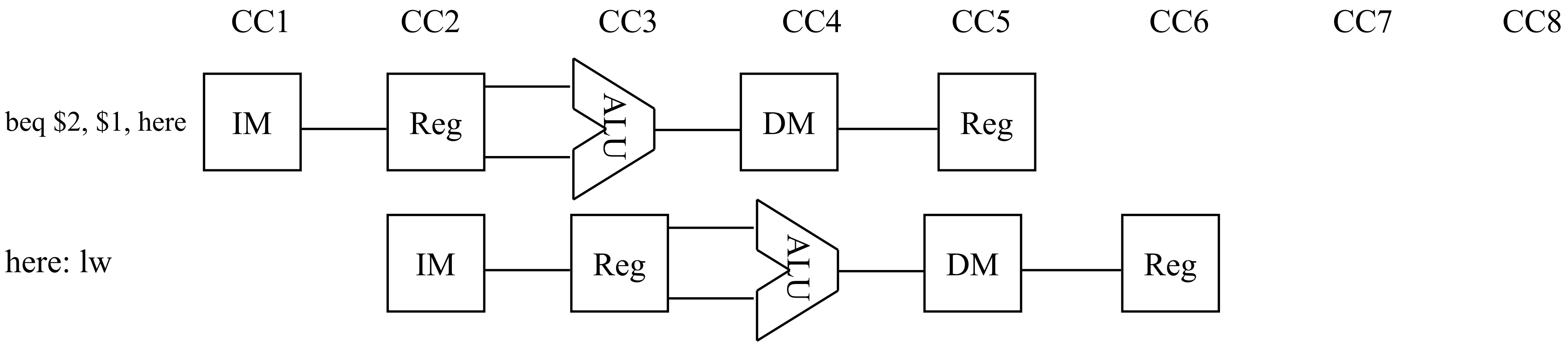
add ...

there: sub $12, $4, $2

Assume Branch *Not Taken*

- Performance depends on percentage of time you guess right.
- Flushing an instruction means to prevent it from changing any permanent state (registers, memory, PC).
  - sounds a lot like a bubble...
  - But notice that we need to be able to insert those bubbles later in the pipeline

Branch Hazards – Predicting Taken?

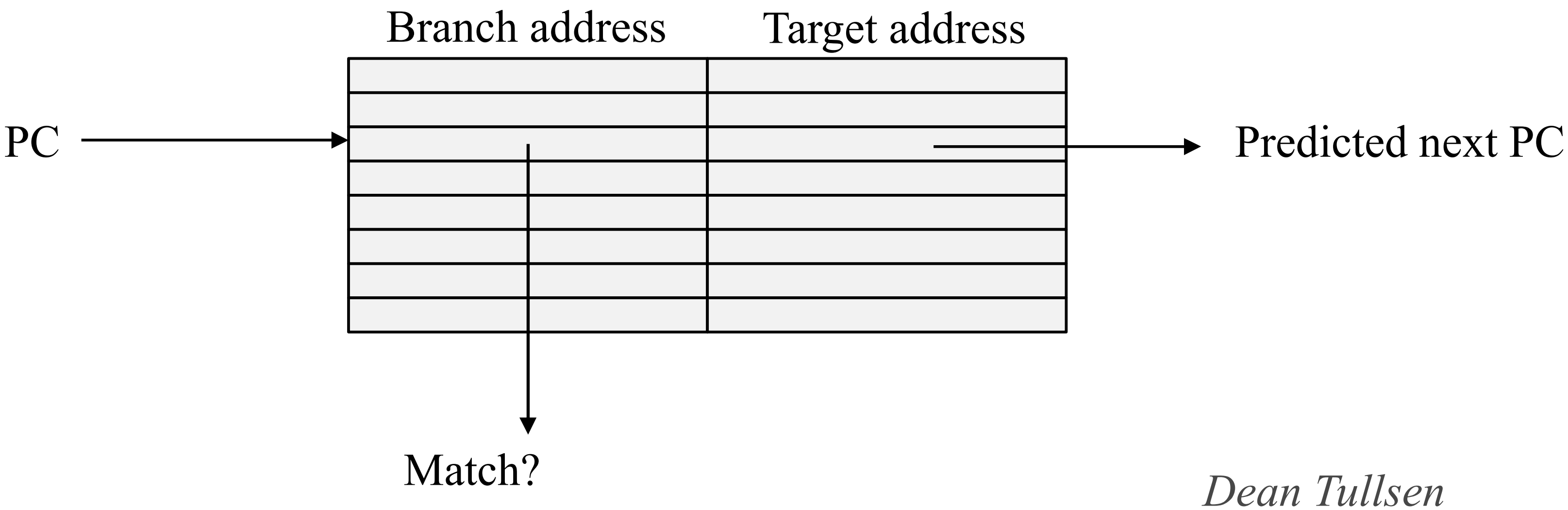


Required information to predict Taken:

1. Whether an instruction is a branch (before decode)
2. The target of the branch
3. The outcome of the branch condition

Branch Target Buffer

- Keeps track of the PCs of recently seen branches and their targets.
- Consult during Fetch (in parallel with Instruction Memory read) to determine:
  - Is this a branch?
  - If so, what is the target

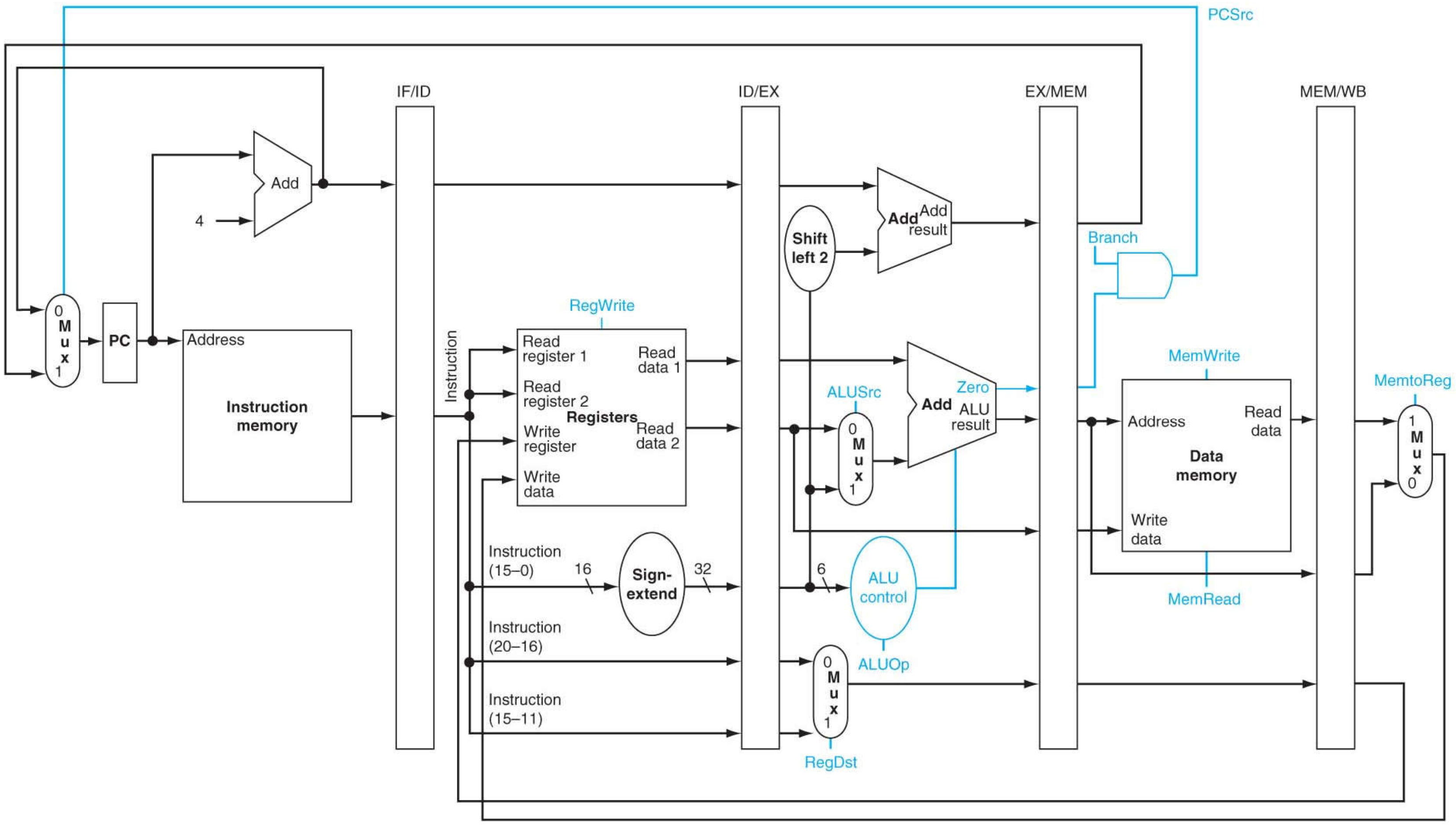


So, just to set the stage…

MIPS R2000

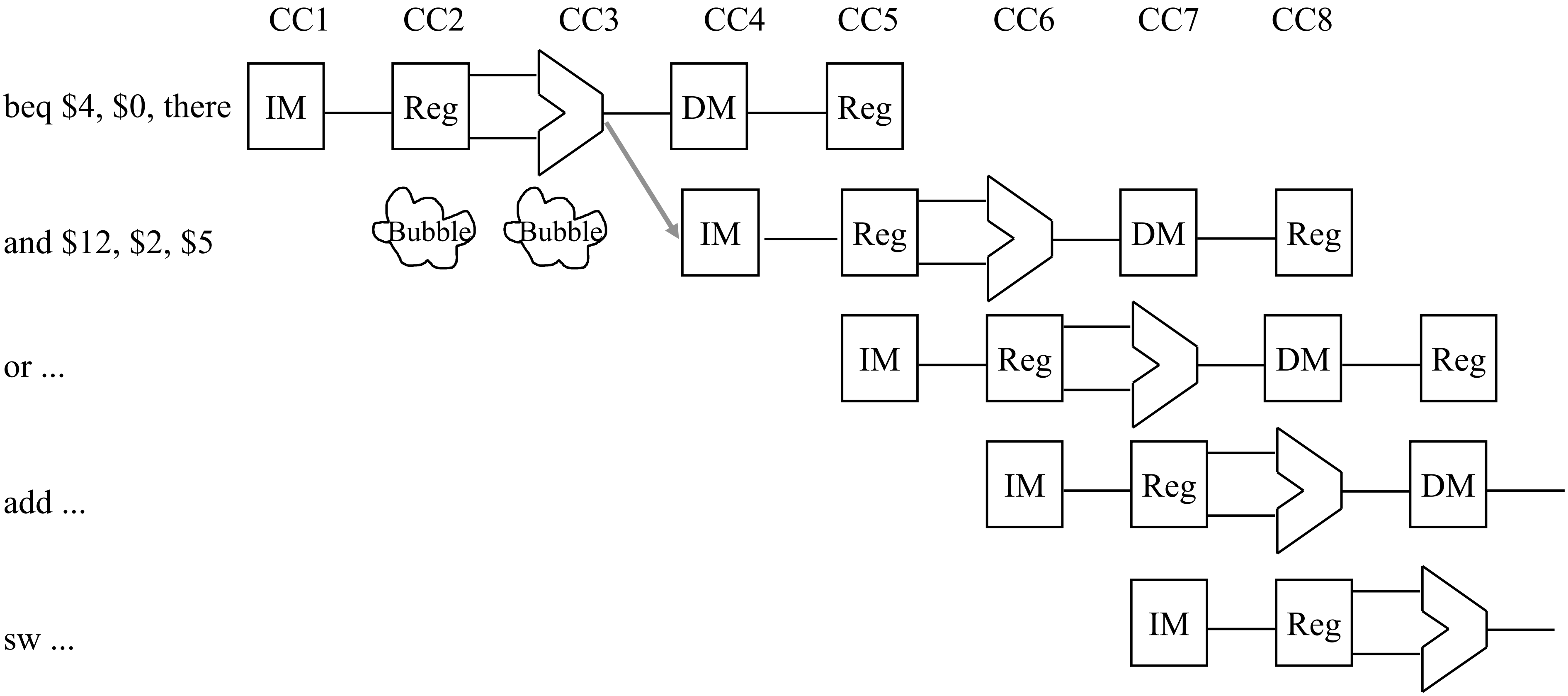
- We are going to walk through the solutions they used for the initial commercial MIPS pipeline.
- Just be aware, many of those do not transfer to modern architectures.
- But this will lay the foundation for modern techniques, which we'll look at in some detail.

Reducing the Branch Delay

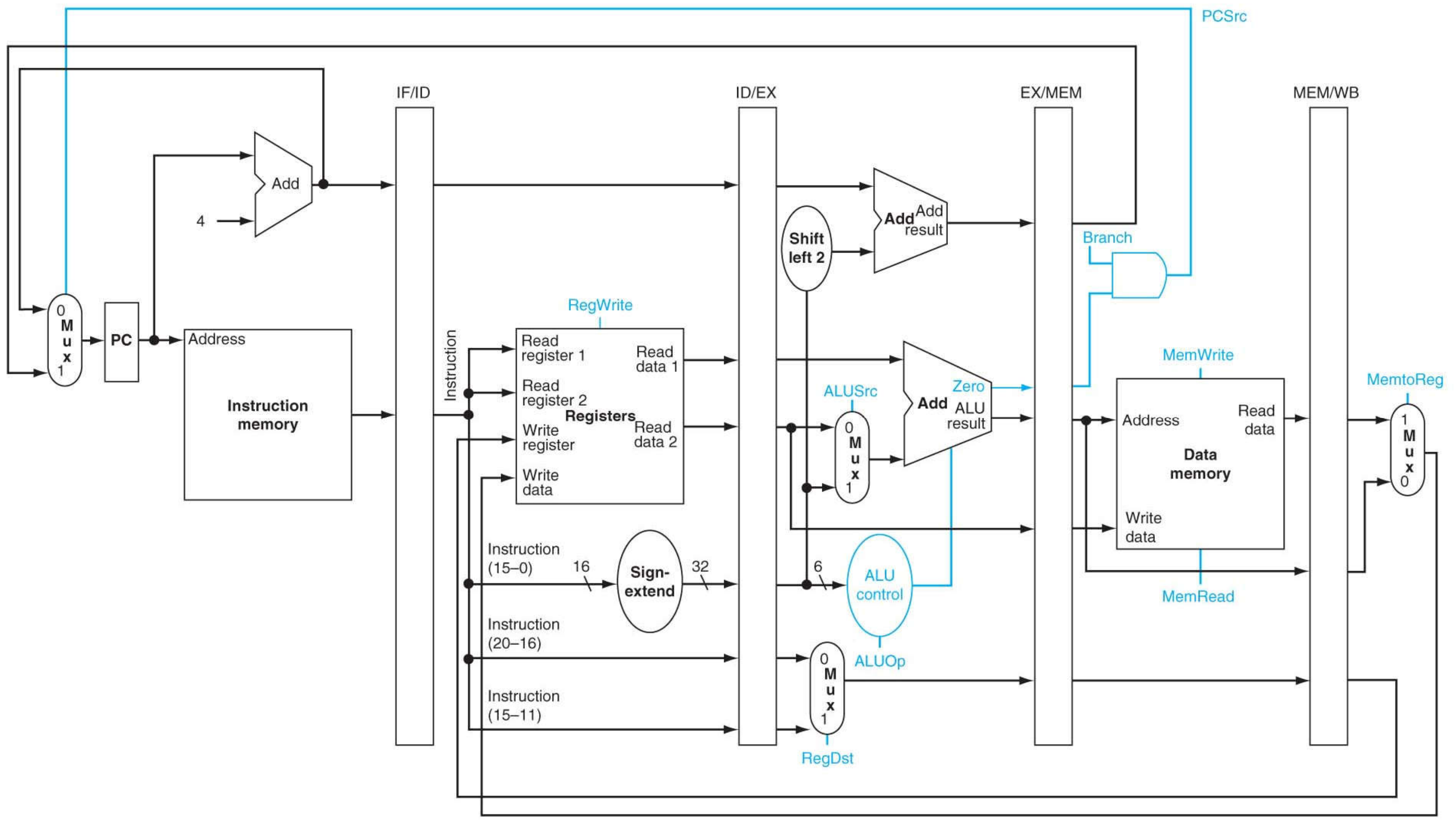


- can easily get to 2-cycle stall

Stalling for Branch Hazards

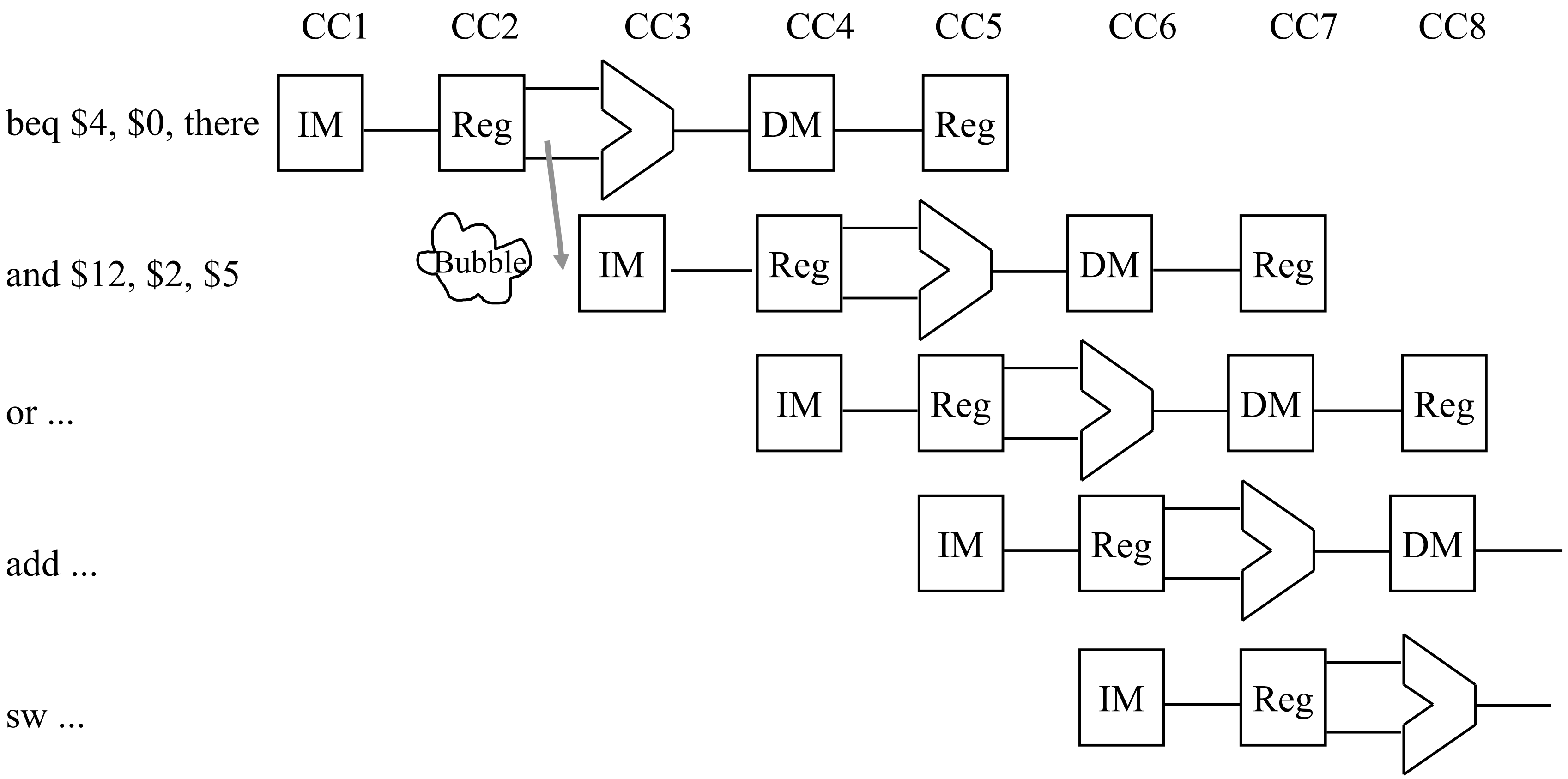


Reducing the Branch Delay

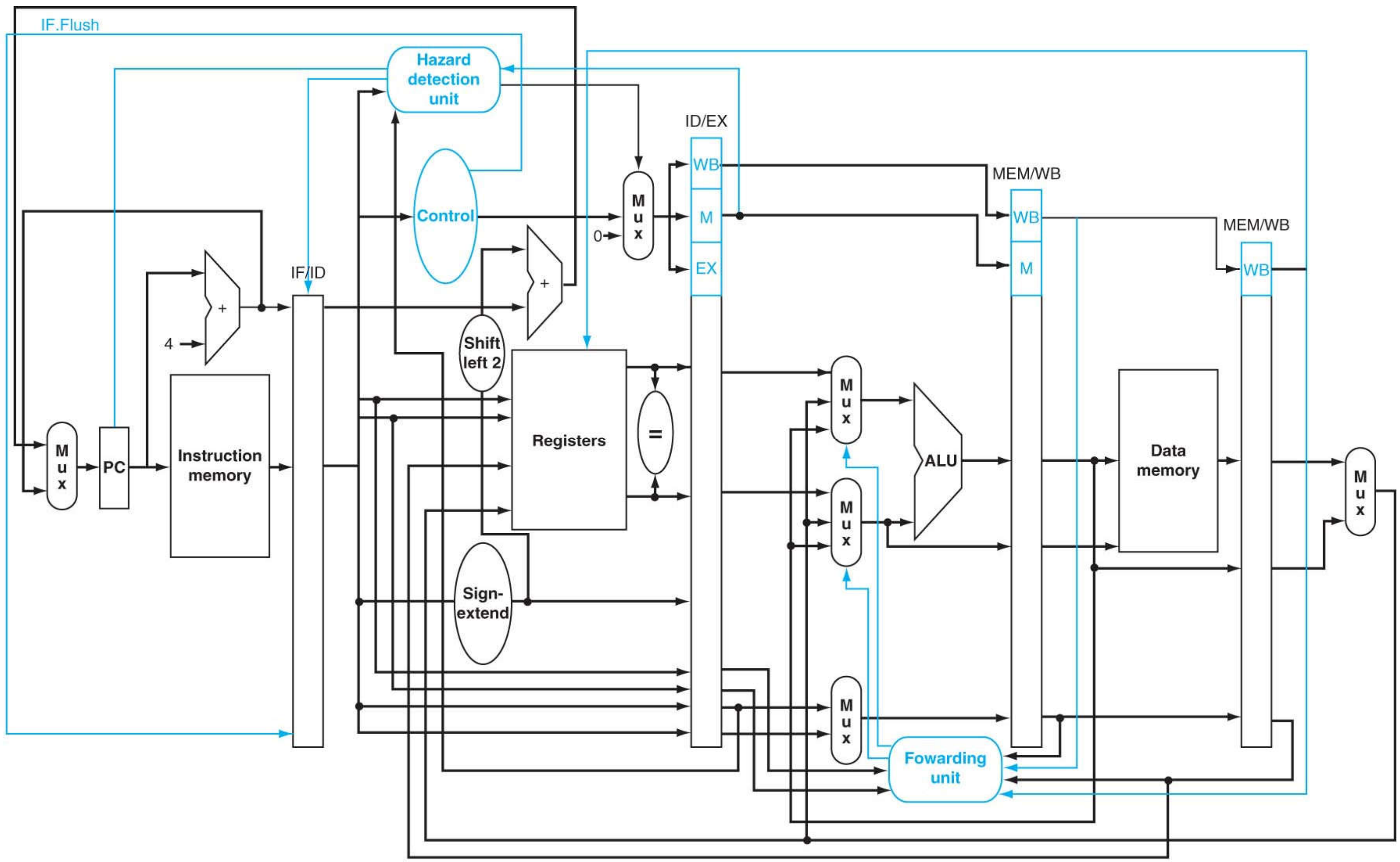


- Harder, but possible, to get to 1-cycle stall

Stalling for Branch Hazards



The Pipeline with flushing for taken branches

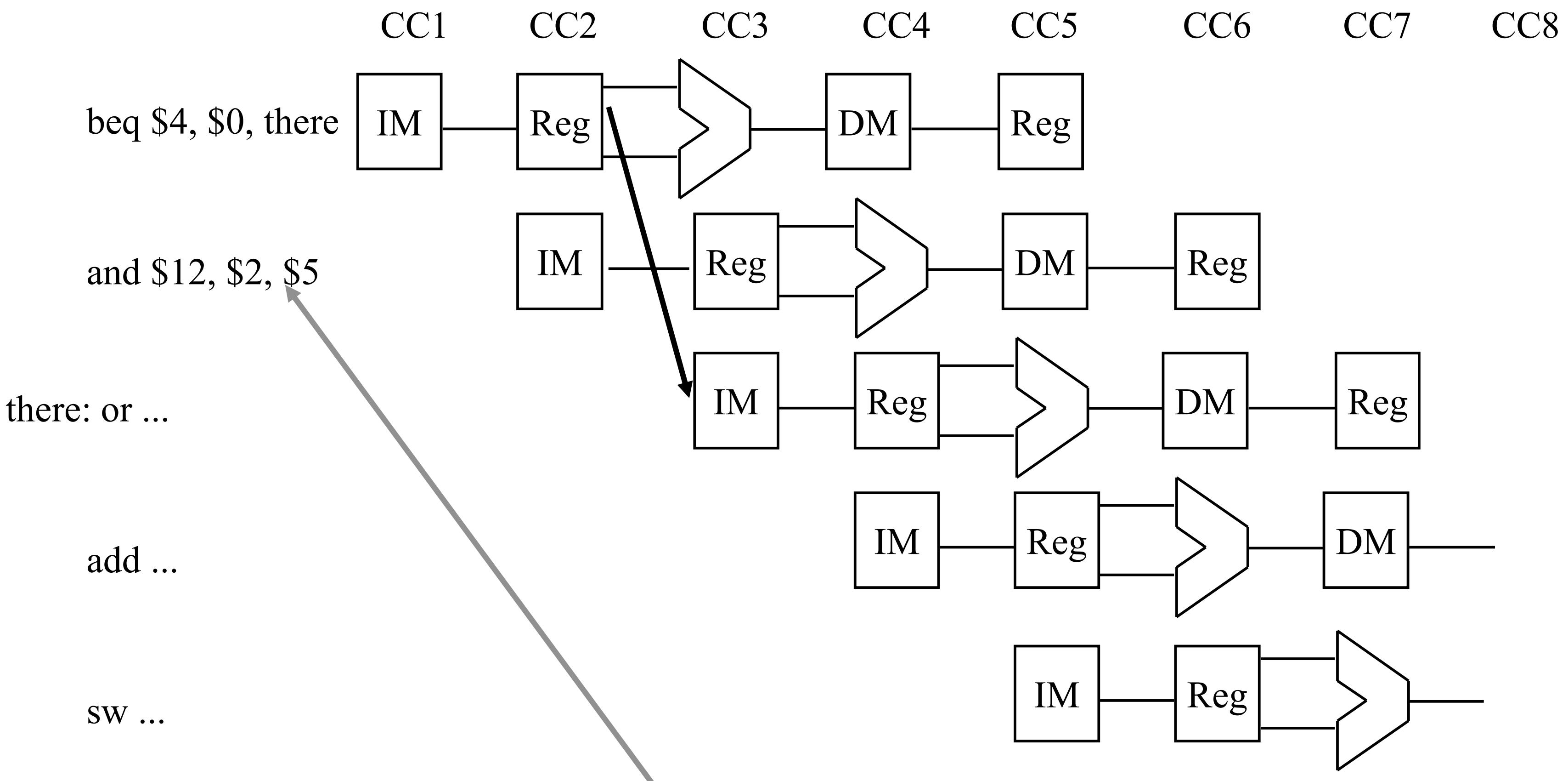


- Notice the IF/ID flush line added.
- Not our final design yet, so don’t need to study this too hard.
- MIPS ISA choice -> beq, blt, …

Eliminating the Branch Stall

- There’s no rule that says we have to see the effect of the branch immediately. Why not wait an extra instruction before branching?
- The original SPARC and MIPS processors each used a single *branch delay slot* to eliminate single-cycle stalls after branches.
- The instruction after a conditional branch is *always executed* in those machines, regardless of whether the branch is taken or not!

Branch Delay Slot



beq $4, $0, there

and $12, $2, $5

there: or ...

add ...

sw ...

Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.

Filling the branch delay slot

- The branch delay slot is only useful if you can find something to put there.
- If you can't find anything, you must put a *nop* to insure correctness.
- Where do we find instructions to fill the branch delay slot?

  –

  –

  –

Filling the branch delay slot

```
1  add  $5, $3, $7
2  add  $9, $1, $3
3  sub  $6, $1, $4
4  and  $7, $8, $2
5  beq $6, $7, there
   nop   /* branch delay slot */
6  add  $9, $1, $4
7  sub  $2, $9, $5
   ...
   there:
8  mult $2, $10, $11
   …
```

- Which instructions could be used to replace the nop?

Branch Delay Slots

- This works great for this implementation of the architecture, but becomes a permanent part of the ISA.
- What about the MIPS R10000, which has a 5-cycle branch penalty, and executes 4 instructions per cycle??
- What about the Pentium 4, which has a 21-cycle branch penalty and executes up to 3 instructions per cycle???

Early resolution of branch + branch delay slot

- Worked well for MIPS R2000 (the 5-stage pipeline MIPS)
- Early resolution doesn't scale well to modern architectures
  - Better to always have execute happen in execute
  - Forwarding into branch instruction?
- Branch delay slot
  - Doesn't solve the problem in modern pipelines
  - Still in ISA, so have to make it work even though it doesn't provide any significant advantage.
  - Violates important general principal – (unless you really only want a single generation of your product) do not expose current technology limitations to the ISA!!

Okay, then...

- What do we do in modern architectures???

Branch Prediction

- Always assuming the branch is not taken is a crude form of *branch prediction*.

Branch Prediction

- Always assuming the branch is not taken is a crude form of *branch prediction*.
- What about loops that are *taken* 95% of the time?
  - we would like the option of assuming *not taken* for some branches, and *taken* for others, depending on ???

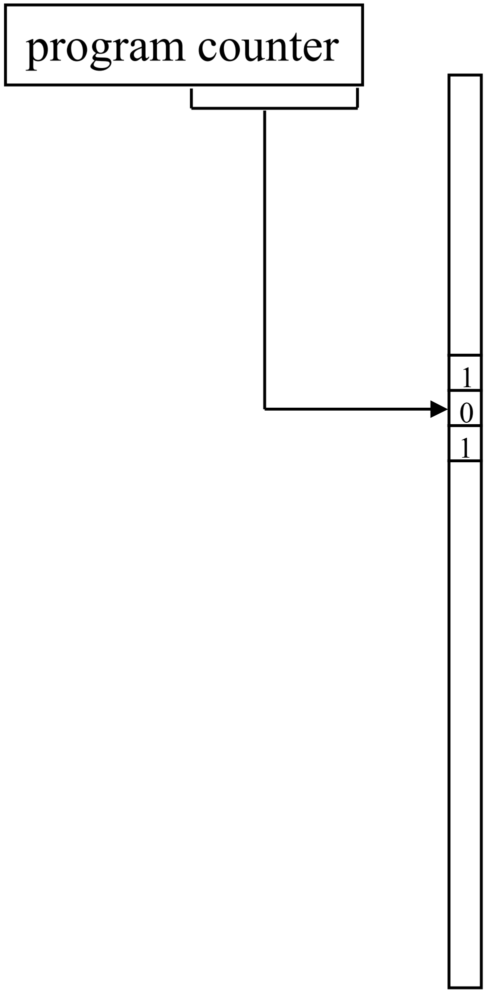
Branch Prediction

- Historically, two broad classes of branch predictors:
- *Static predictors* – for branch B, always make the same prediction.
- *Dynamic predictors* – for branch B, make a new prediction every time the branch is fetched.
- Tradeoffs?
- Modern CPUs all have sophisticated *dynamic* branch prediction.

Dynamic Branch Prediction

- What information is available to make an intelligent prediction?

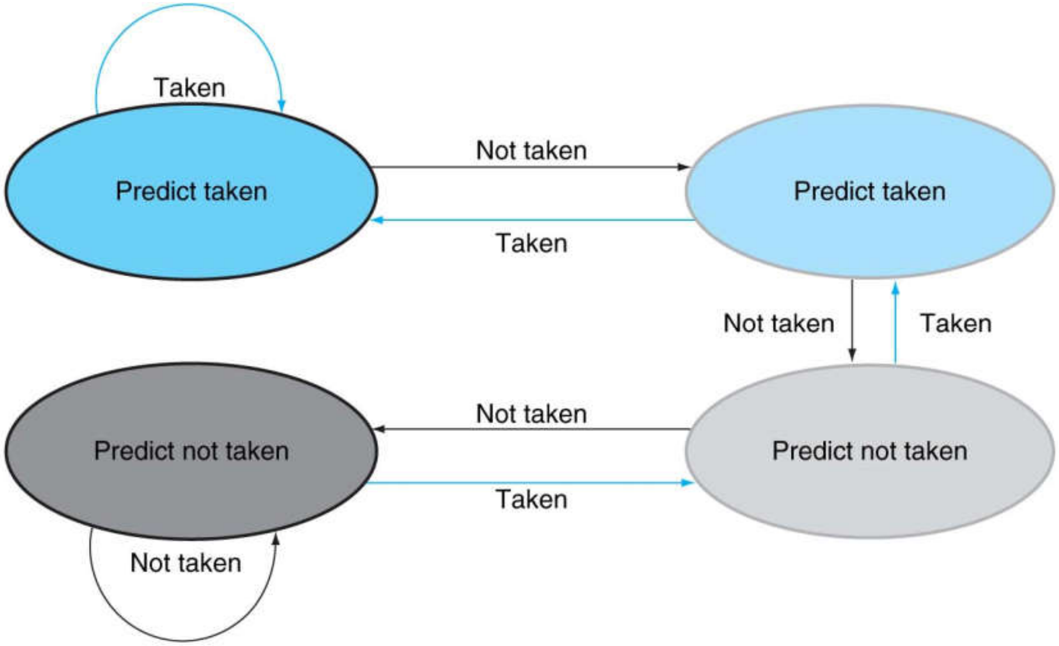
Branch Prediction



```
for (i=0;i<10;i++) {
...
...
}
```

```
...
...
add  $i, $i, #1
beq $i, #10, loop
```

Two-bit predictors give better loop prediction



```
for (i=0;i<10;i++) {
...
...
}
```

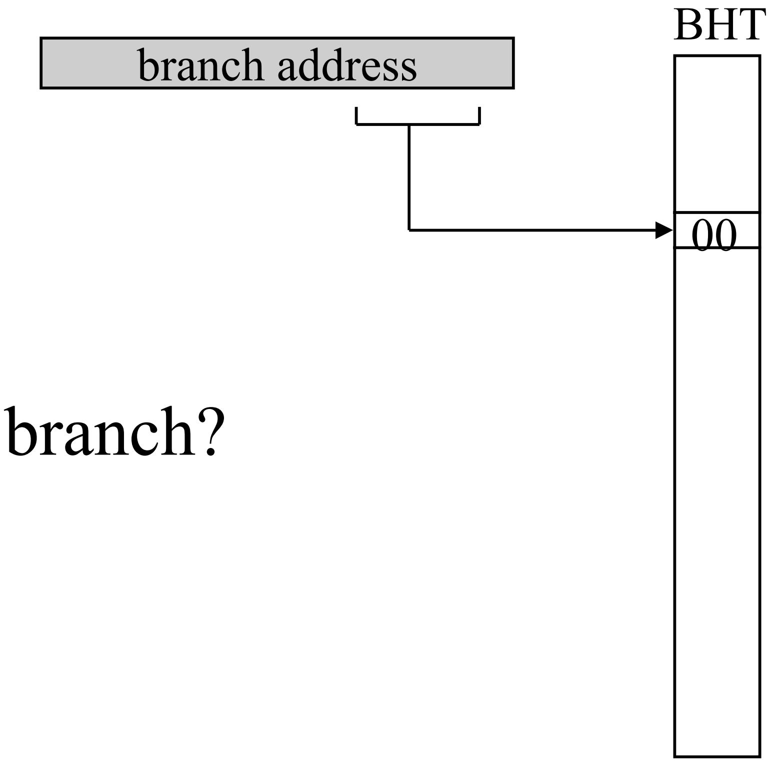
↓

```
...
...
add  $i, $i, #1
beq $i, #10, loop
```

This state machine also referred to as a *saturating counter* – it counts down (on *not takens*) to 00 or up (on *takens*) to 11, but does not wrap around.

Branch History Table (*bimodal* predictor)

- has limited size
- 2 bits by N (e.g. 4K)
- uses low bits of branch address to choose entry

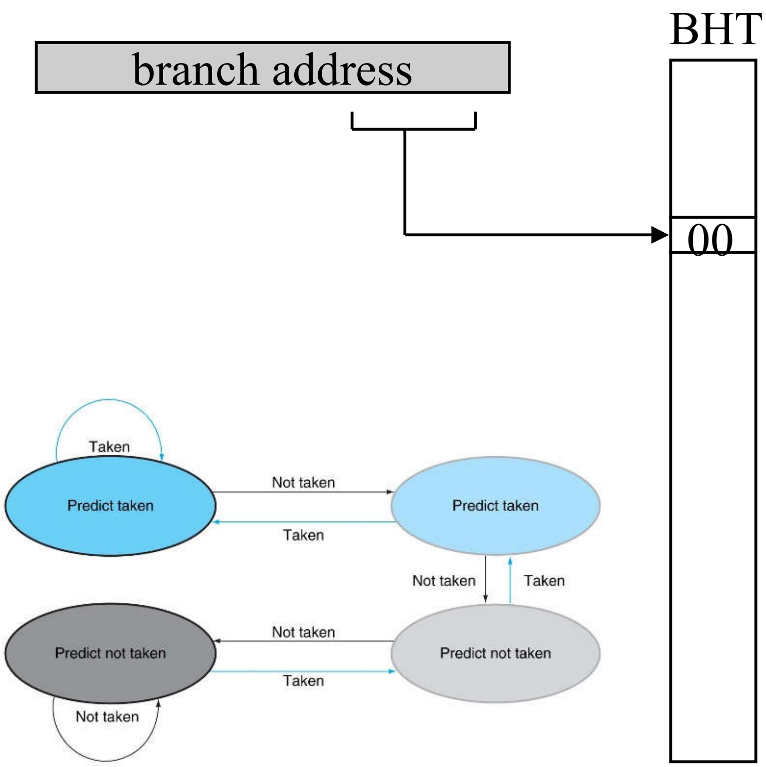


- what about even/odd branch?

bimodal predictor

- For the following loop, what will be the prediction accuracy of the bimodal predictor for the conditional branch that closes the loop?

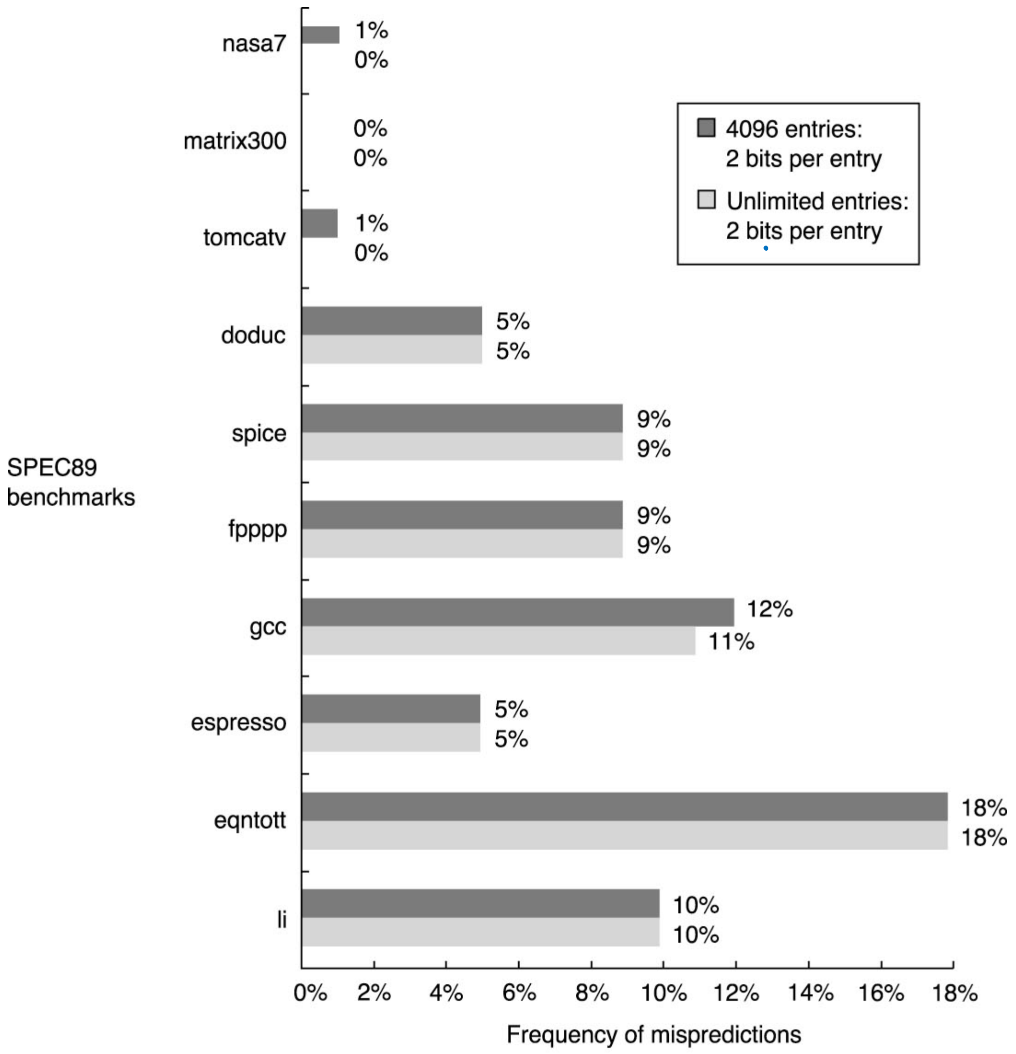
```
        for (i=0; i< 2; i++)  //two iterations per loop
                { z = …  }
```



(let’s make this a freeform answer poll question)

Selection	Accuracy
A	100%
B	50%
C	0%
D	Maybe 0%, maybe 50%
E	other

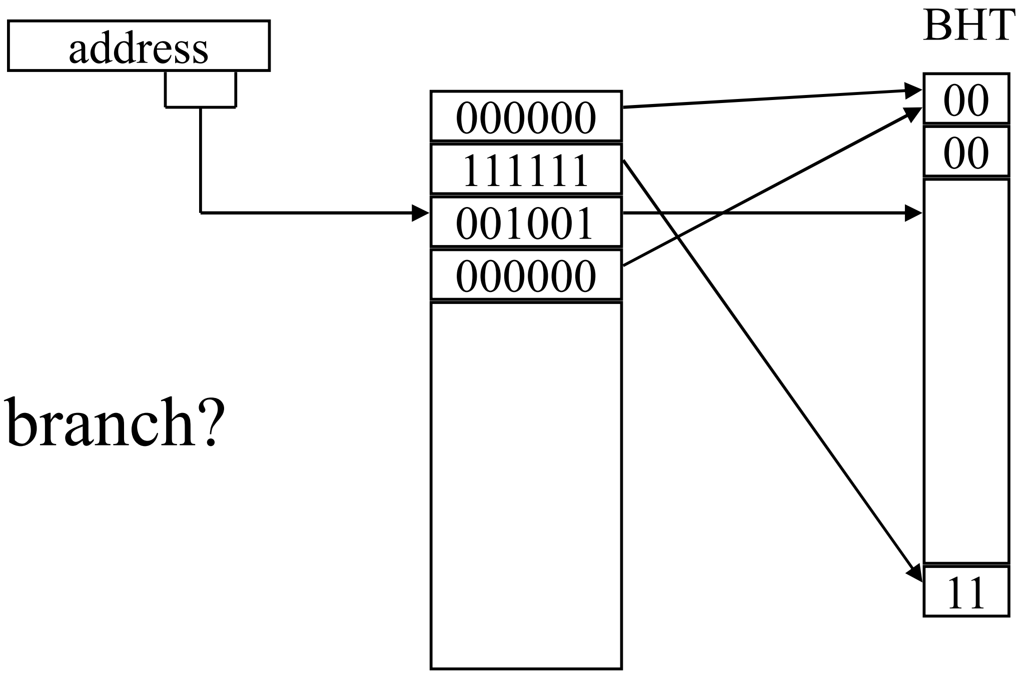
2-bit bimodal prediction accuracy



Is this good enough?

Can We Do Better?

- Can we get more information dynamically than just the recent bias of this branch?
- We can look at patterns (2-level *local* predictor) for a particular branch.
  - last eight branches 00100100, then it is a good guess that the next one is "1" (taken)



- even/odd branch?

Can We Do Better?

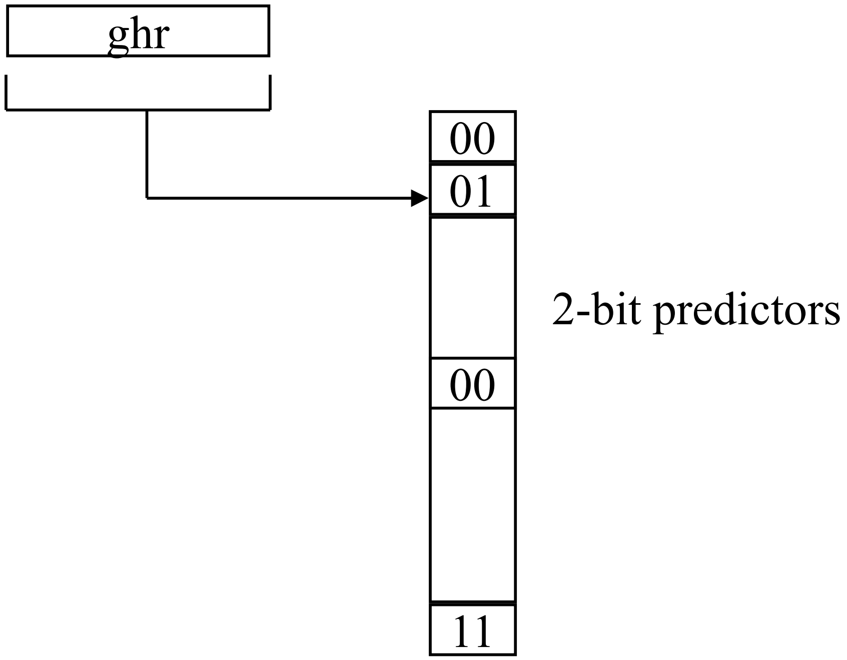
- *Correlating Branch Predictors* also look at other branches for clues

```
if (i == 0)
  ...
if (i > 7)
  ...
```

- Typically use two indexes
  - Global history register --> history of last m branches (e.g., 0100011)
  - branch address

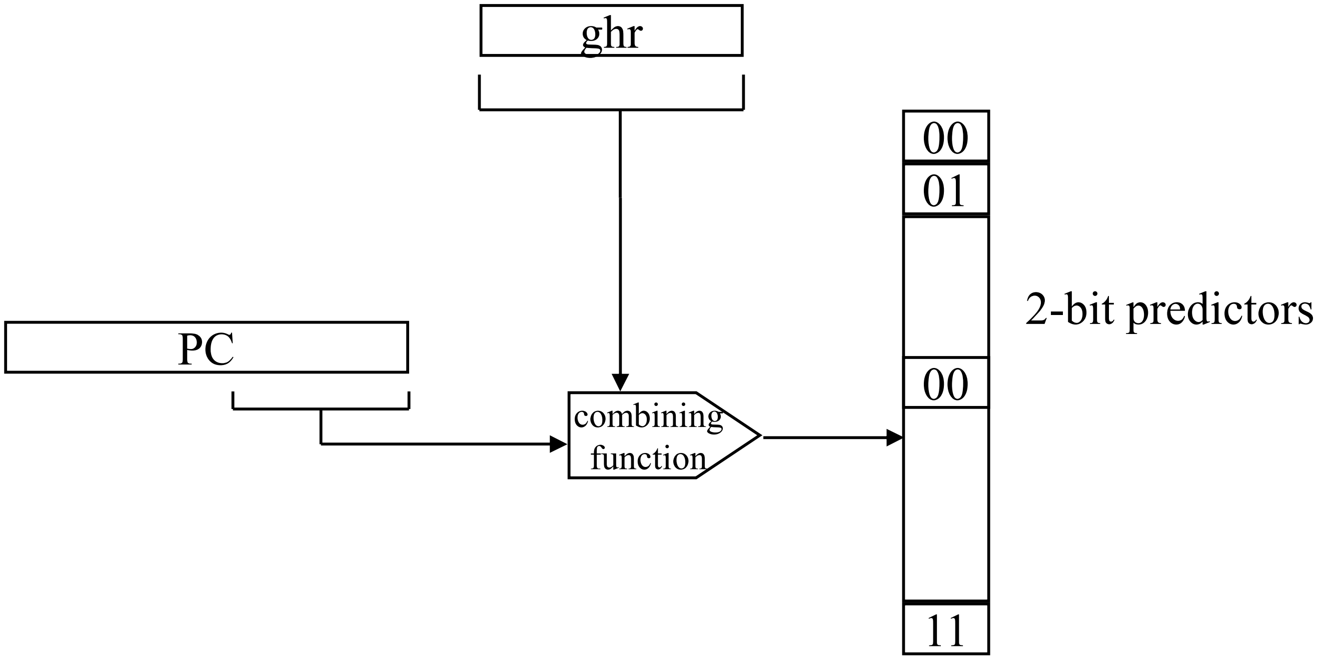
Correlating Branch Predictors

- The *global history register (ghr)* is a shift register that records the last *n* branches (of any address) encountered by the processor.



Two-level correlating branch predictors

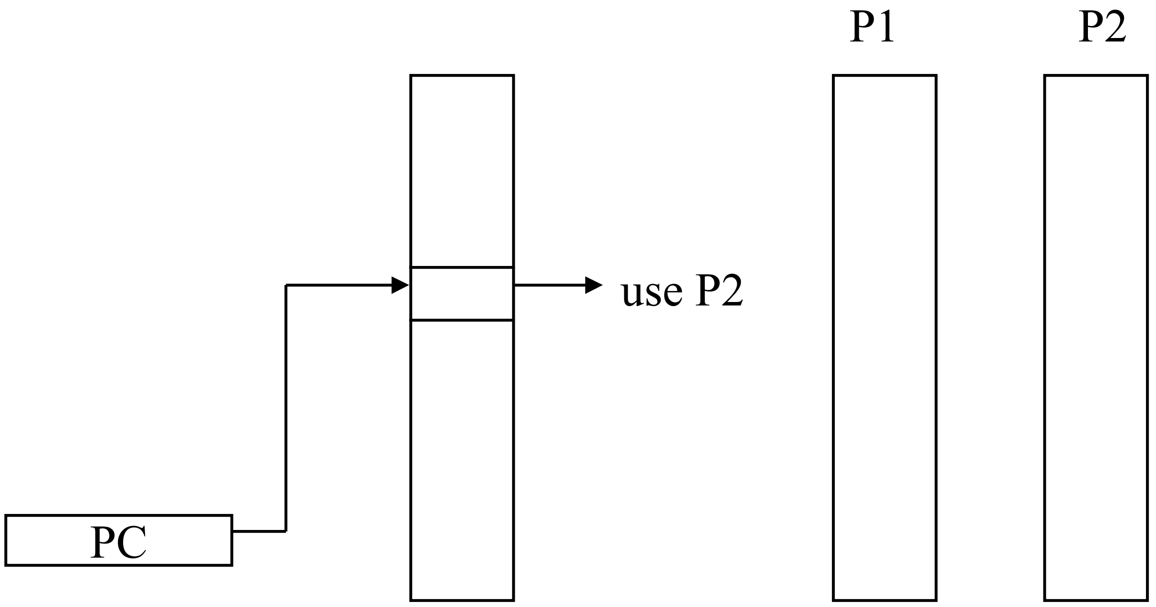
- Can use both the PC address and the GHR



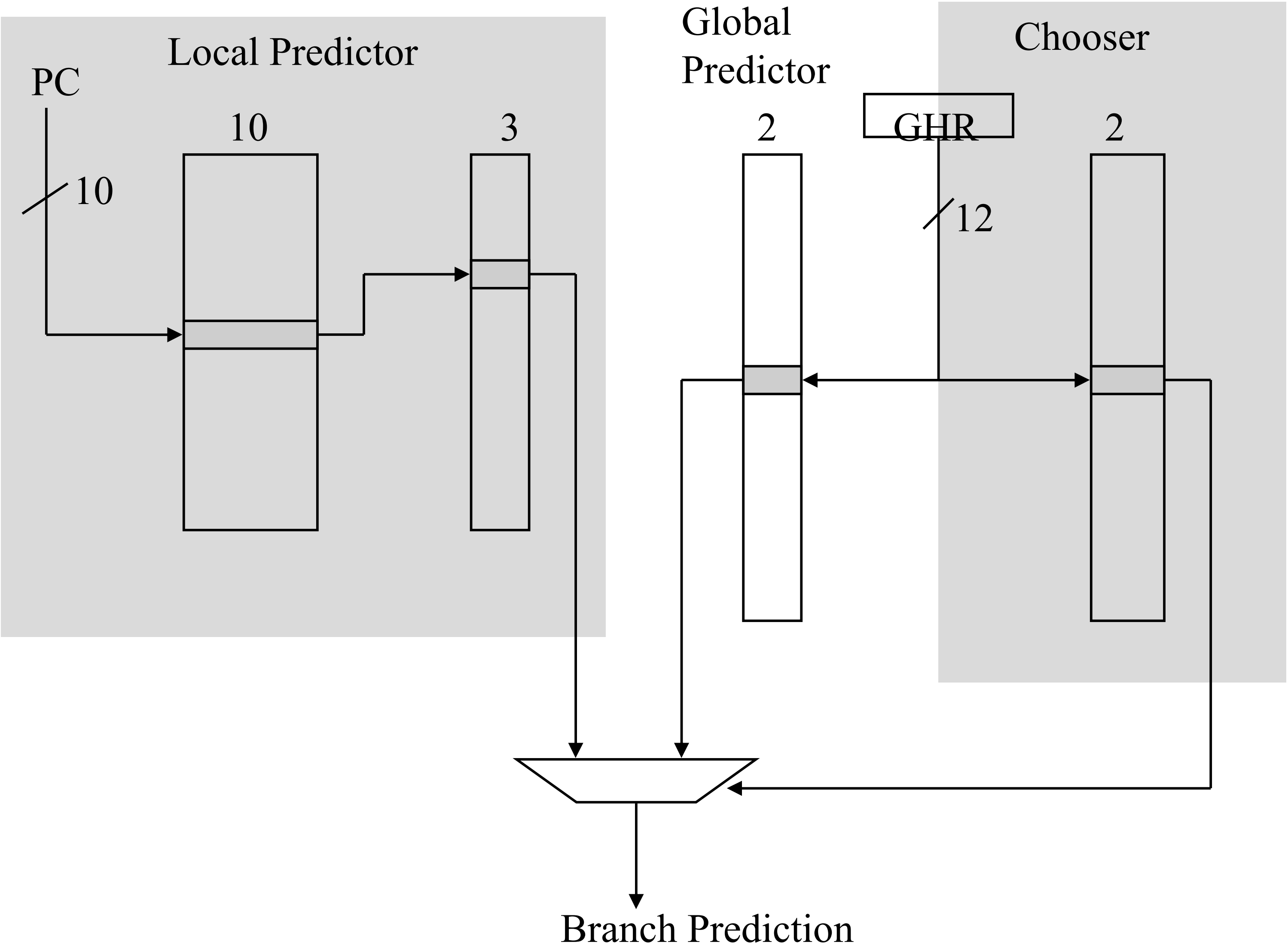
- Most common – ***gshare***: use xor as the combining function.

Are we happy yet????

- *Combining branch predictors* use multiple schemes and a voter to decide which one typically does better for *that* branch.



Compaq/Digital Alpha 21264



Aliasing in Branch Predictors

- Branch predictors will always be of finite size, while code size is relatively unlimited.

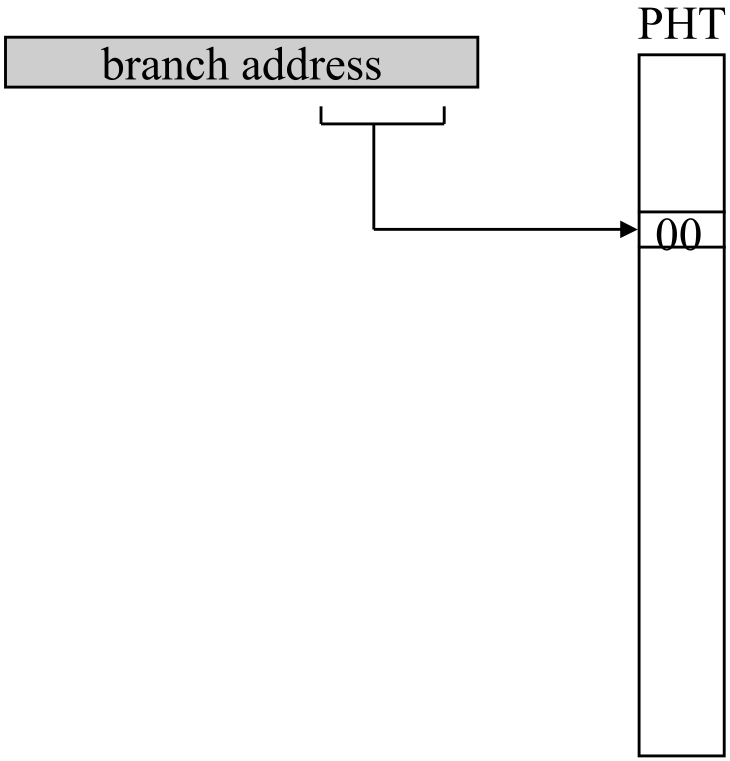
Aliasing in Branch Predictors

- Branch predictors will always be of finite size, while code size is relatively unlimited.
- What happens when (in the common case) there are more branches than entries in the branch predictor?

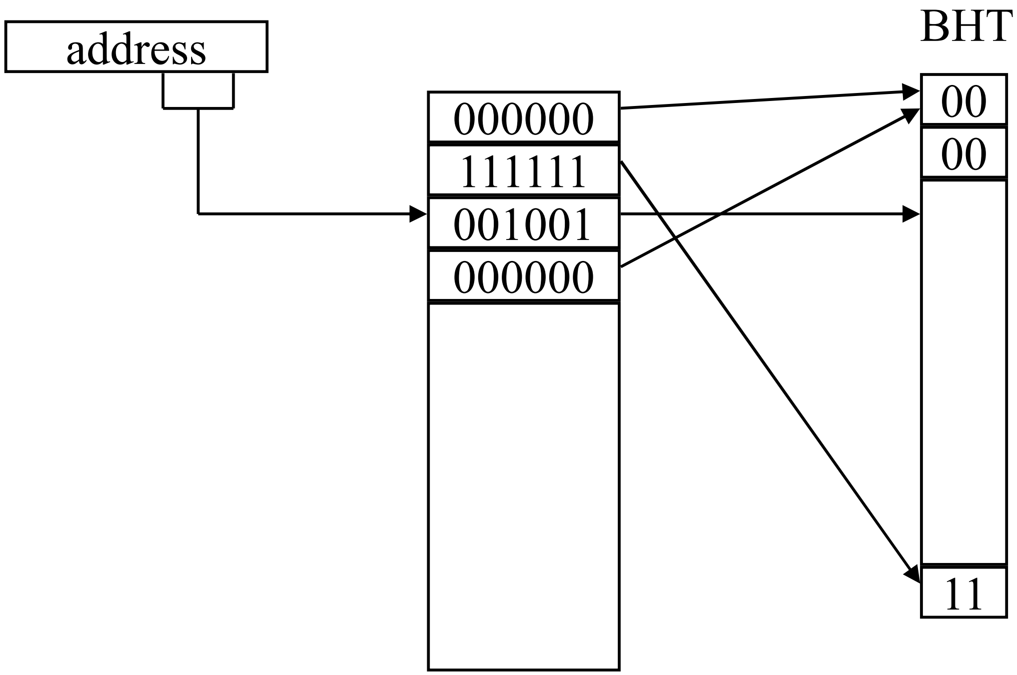
Aliasing in Branch Predictors

- Branch predictors will always be of finite size, while code size is relatively unlimited.
- What happens when (in the common case) there are more branches than entries in the branch predictor?
- We call these conflicts *aliasing*.
- We can have negative aliasing (when biases are different) or neutral aliasing (biases same). Positive aliasing is unlikely.

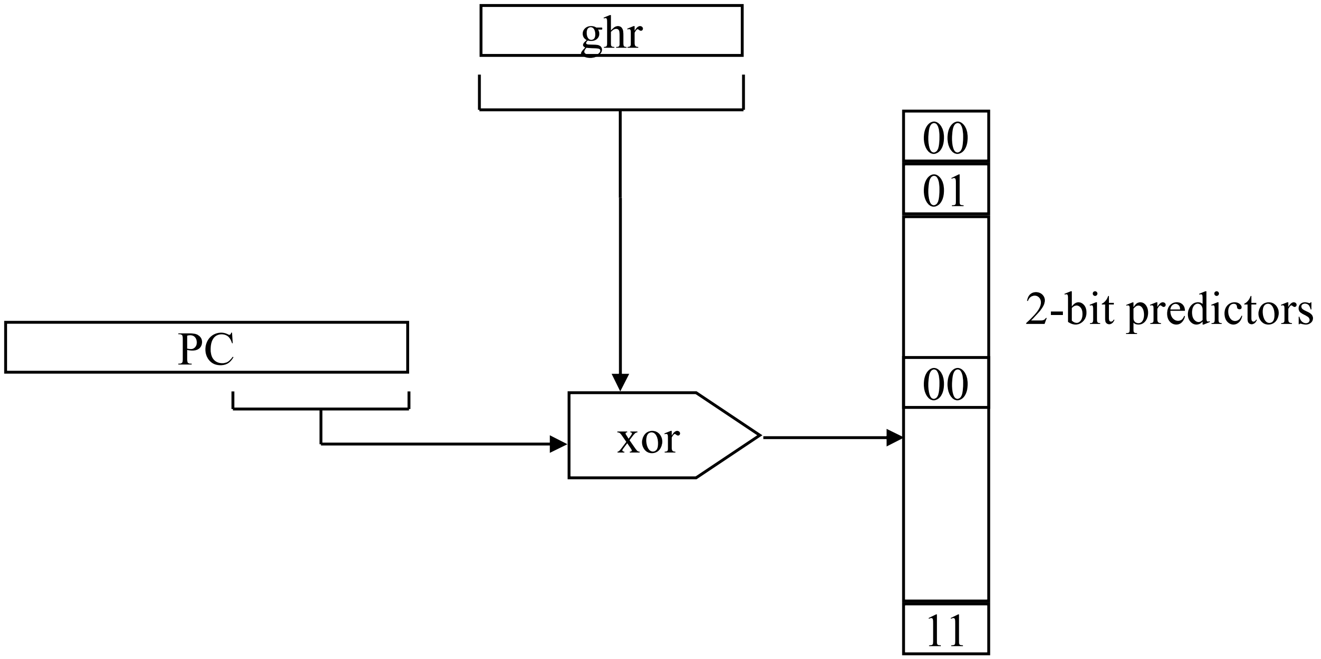
Bimodal aliasing



Local Predictor Aliasing



Gshare aliasing



Branch Prediction

- Latest branch predictors significantly more sophisticated, using more advanced correlating techniques, larger structures, and in some cases using AI techniques.
- Remember from earlier….
  - Presupposes what two pieces of information are available at *fetch time*?
    - ▪
    - ▪
  - Branch Target Buffer supplies this information.

MIPS R2000

- Okay, at this point, we’ve essentially recreated the MIPS R2000 (ignoring the branch prediction slides – that all came later).
- R2000
  - 5 stage pipeline
  - Forwarding wherever possible
  - 1 cycle load-use hazard
  - Branches resolved in ID stage
  - Branch Delay Slot incorporated in the ISA.

- So what kind of performance should we expect?

Pipeline performance

```
loop:   lw $15, 1000($2)
        add $16, $15, $12
        lw $18, 1004($2)
        add $19, $18, $12
        beq $19, $0, loop:
        nop
```

What is the steady-state CPI of this code? Assume branch taken many times. Assume 5-stage pipeline, forwarding, early branch resolution, BDS. (always assume this architecture if not given the details)

Can we improve this?

Putting it all together.

For a given program on our 5-stage MIPS pipeline processor:

20% of insts are loads, 50% of instructions following a load are arithmetic instructions depending on the load

20% of instructions are branches. We manage to fill 80% of the branch delay slots with useful instructions.

What is the CPI of your program?

Given our 5-stage MIPS pipeline – what is the steady state CPI for the following code? Assume the branch is taken thousands of times.

```
Loop:  lw r1, 0 (r2)
       add r2, r3, r4
       sub r5, r1, r2
       beq r5, $zero, Loop
       nop
```

IF = 200ps
ID = 100ps
EX = 150ps
M = 200ps
WB = 100ps

Hardware engineers determine these to be the execution times per stage of the MIPS 5-stage pipeline processor. Consider splitting IF and M into 2 stages each. (So IF1 IF2 and M1 M2.) For this code (assume branch usually taken):

```
Loop:  lw r1, 0 (r2)
       add r2, r3, r4
       sub r5, r1, r2
       beq r5, $zero, Loop
       nop
```

What would be the impact of the new 7-stage pipeline compared to the original 5-stage MIPS pipeline.. Assume the pipeline has forwarding where necessary, predicts branch not taken, and resolves branches in ID.

7-stage Pipeline

```
Loop:  lw r1, 0 (r2)
       add r2, r3, r4
       sub r5, r1, r2
       beq r5, $zero, Loop
       nop
```

Control Hazards -- Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.
- Control hazards are detected in hardware.
- We can reduce the impact of control hazards through:
  - early detection of branch address and condition
  - ***branch prediction***
  - branch delay slots