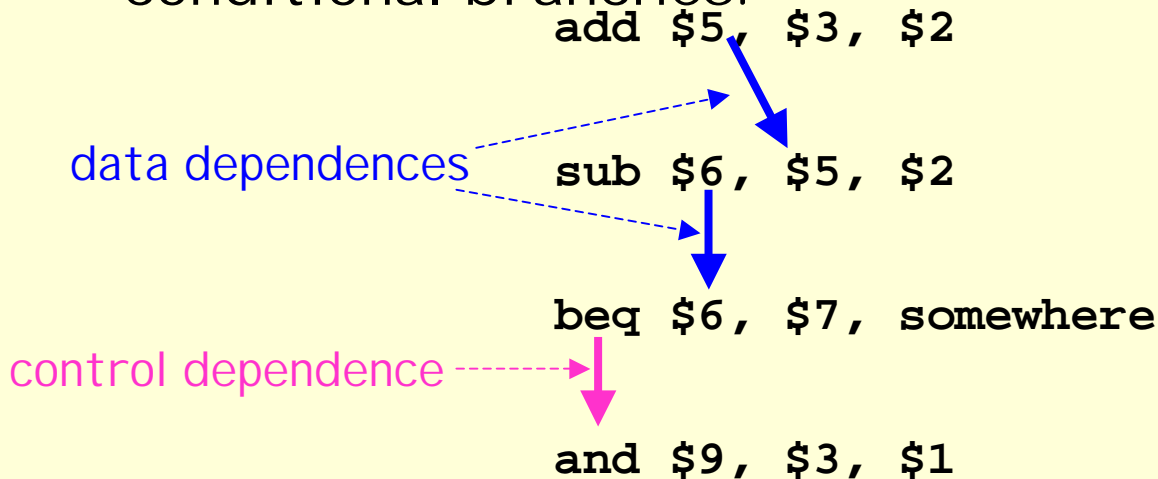


Branch Hazards in the Pipelined Processor

Dependences

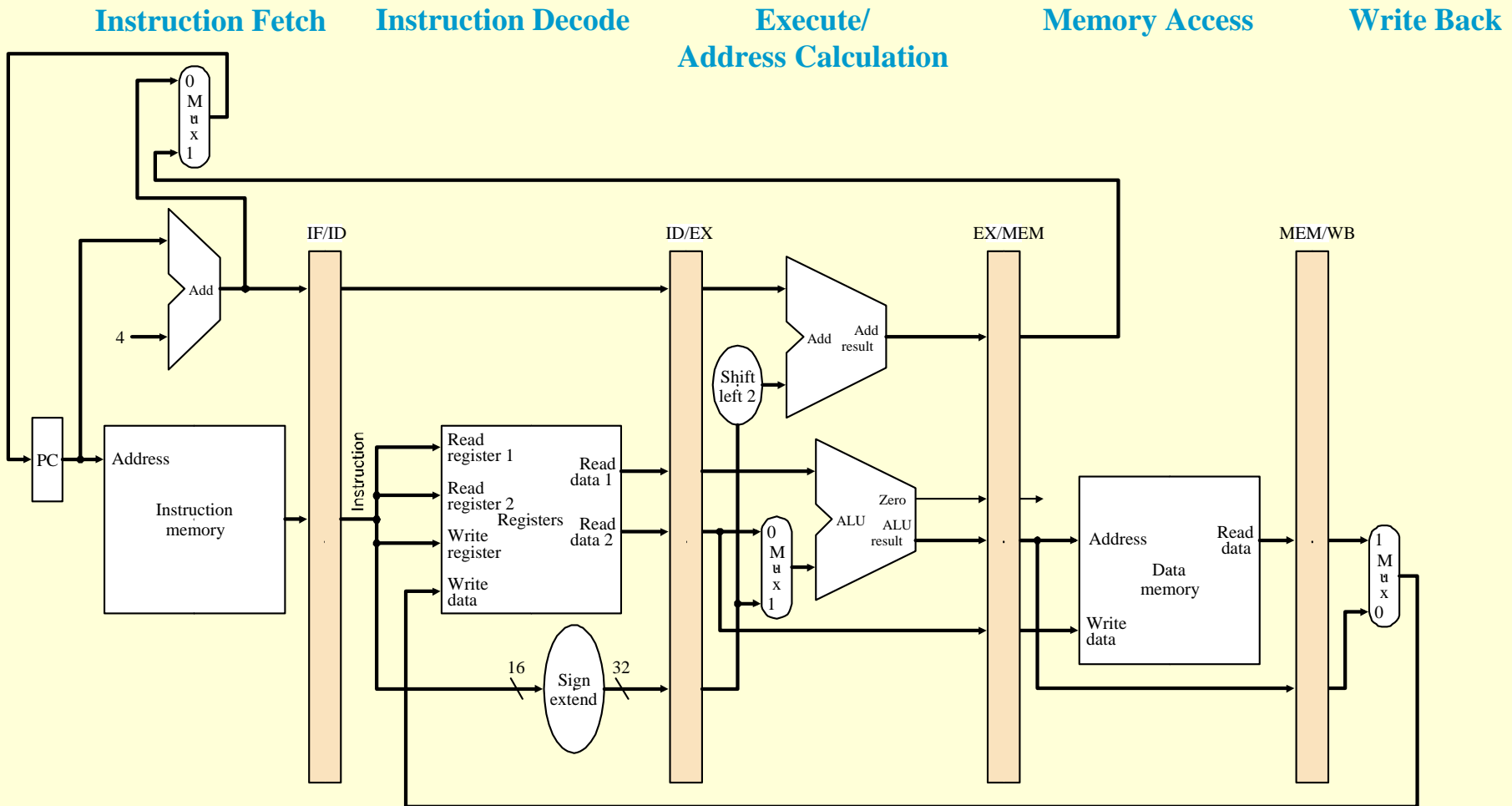
- Data dependence: one instruction is dependent on another instruction to provide its operands.
- Control dependence (aka branch dependences): one instructions determines whether another gets executed or not.
- Control dependences are particularly critical with conditional branches.



Branch Hazards

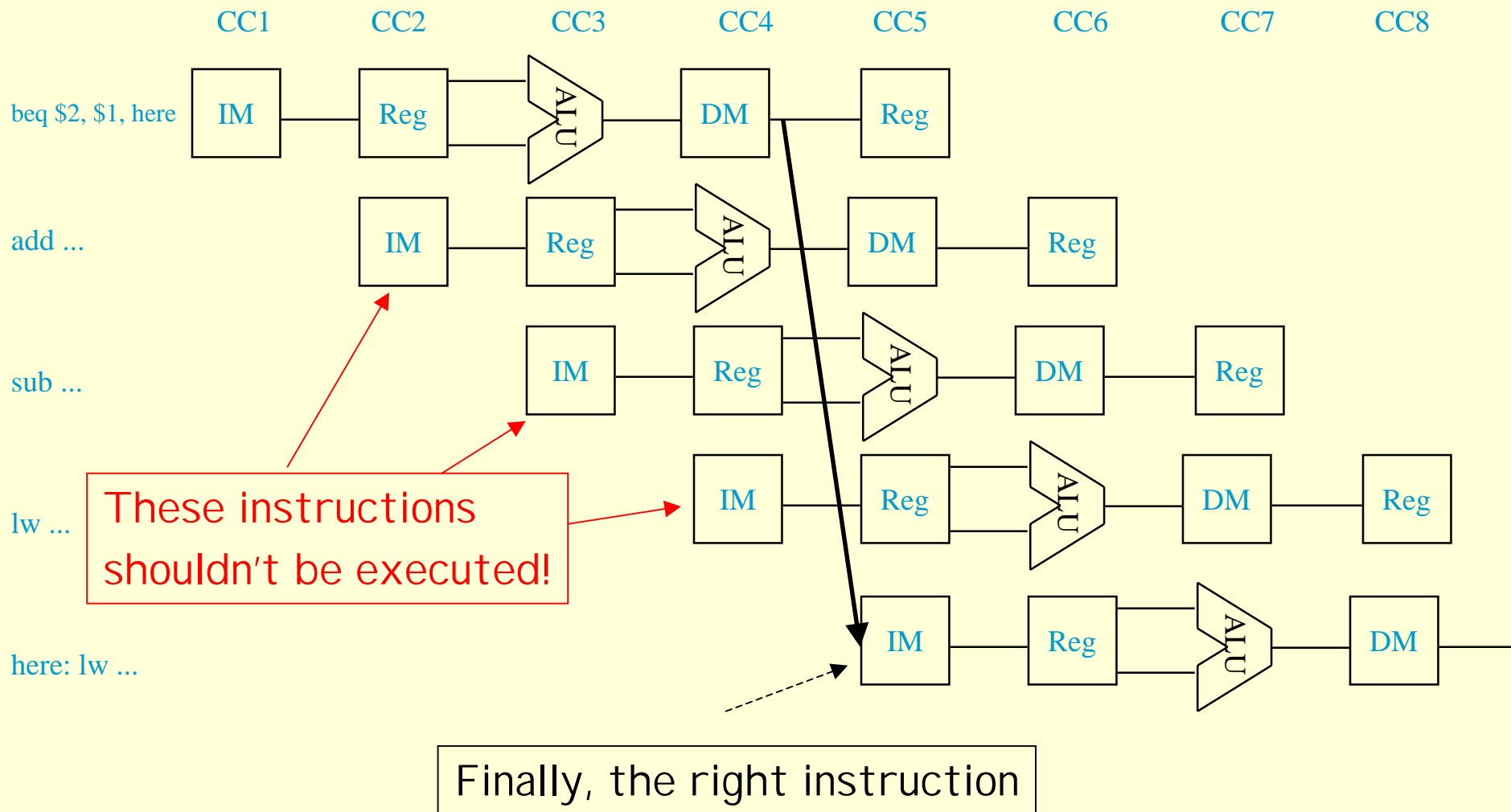
- Branch dependences can result in branch hazards (aka control hazards) when they are too close to be handled correctly in the pipeline.

When are branches resolved?



Branch target address is put in PC during Mem stage.
Correct instruction is fetched during branch's WB stage.

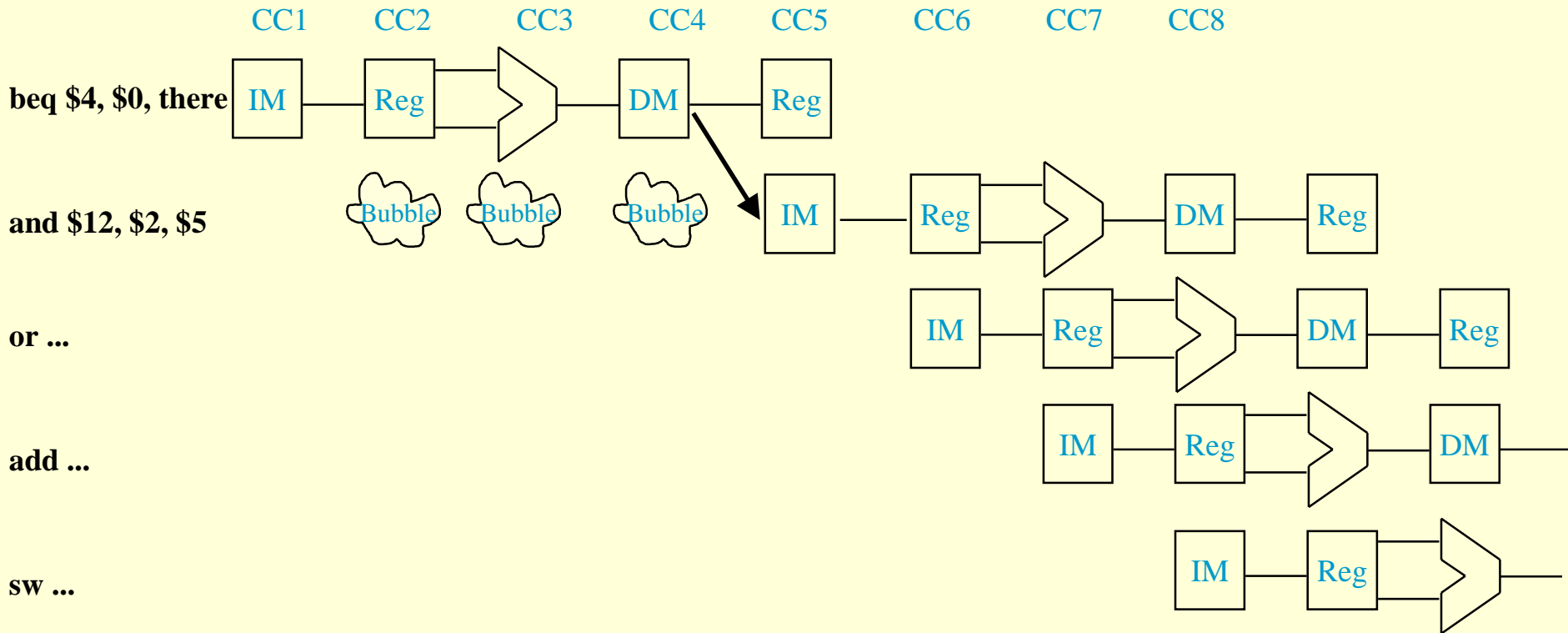
Branch Hazards



Dealing With Branch Hazards

- Software solution
 - insert no-ops (I don't think any processors do this)
- Hardware solutions
 - stall until you know which direction branch goes
 - guess which direction, start executing chosen path (but be prepared to undo any mistakes!)
 - static branch prediction: base guess on instruction type
 - dynamic branch prediction: base guess on execution history
 - reduce the branch delay
- Software/hardware solution
 - delayed branch: Always execute instruction after branch.
 - Compiler puts something useful (or a no-op) there.

Stalling for Branch Hazards

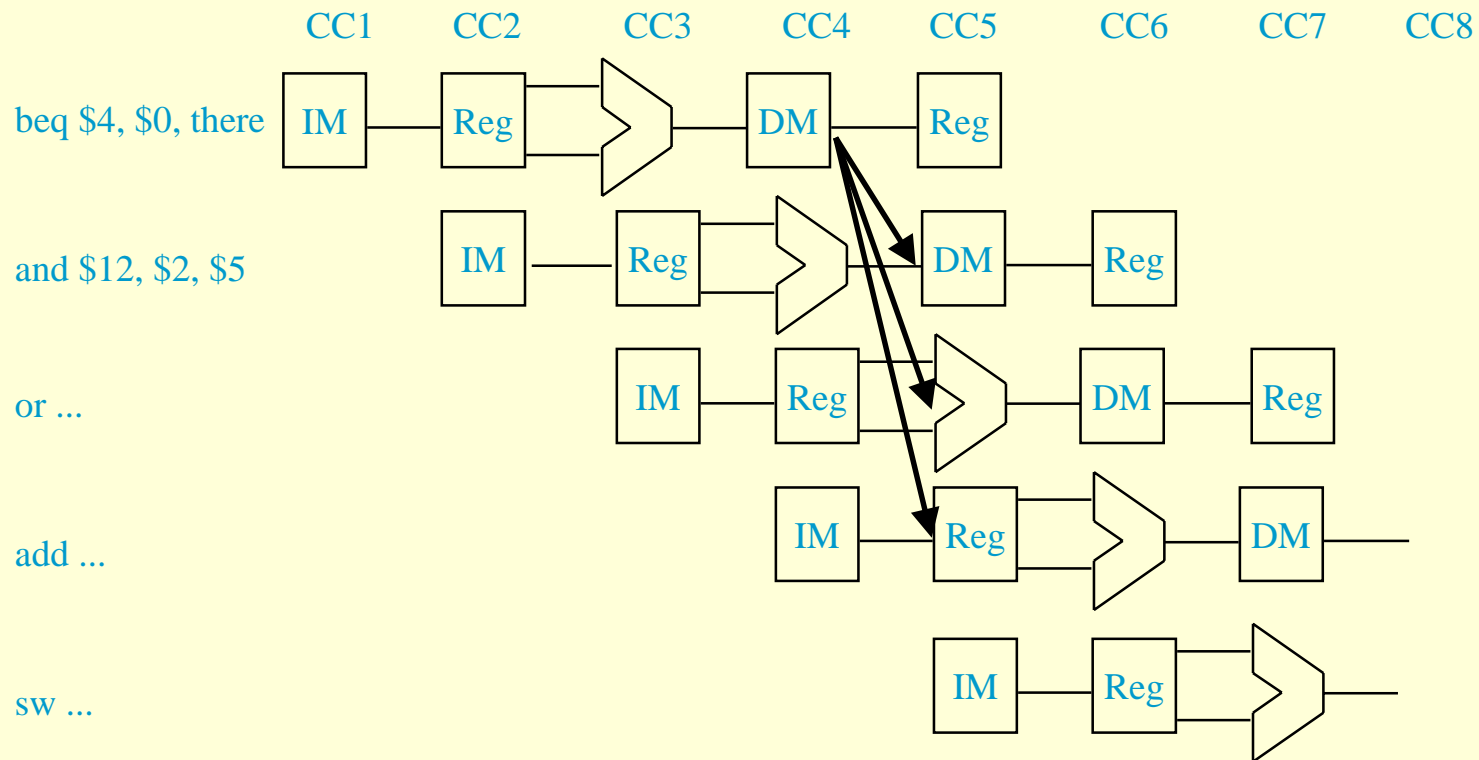


Stalling for Branch Hazards

- All branches waste 3 cycles.
 - Seems wasteful, particularly when the branch isn't taken.
- It's better to guess whether branch will be taken
 - Easiest guess is "branch isn't taken"

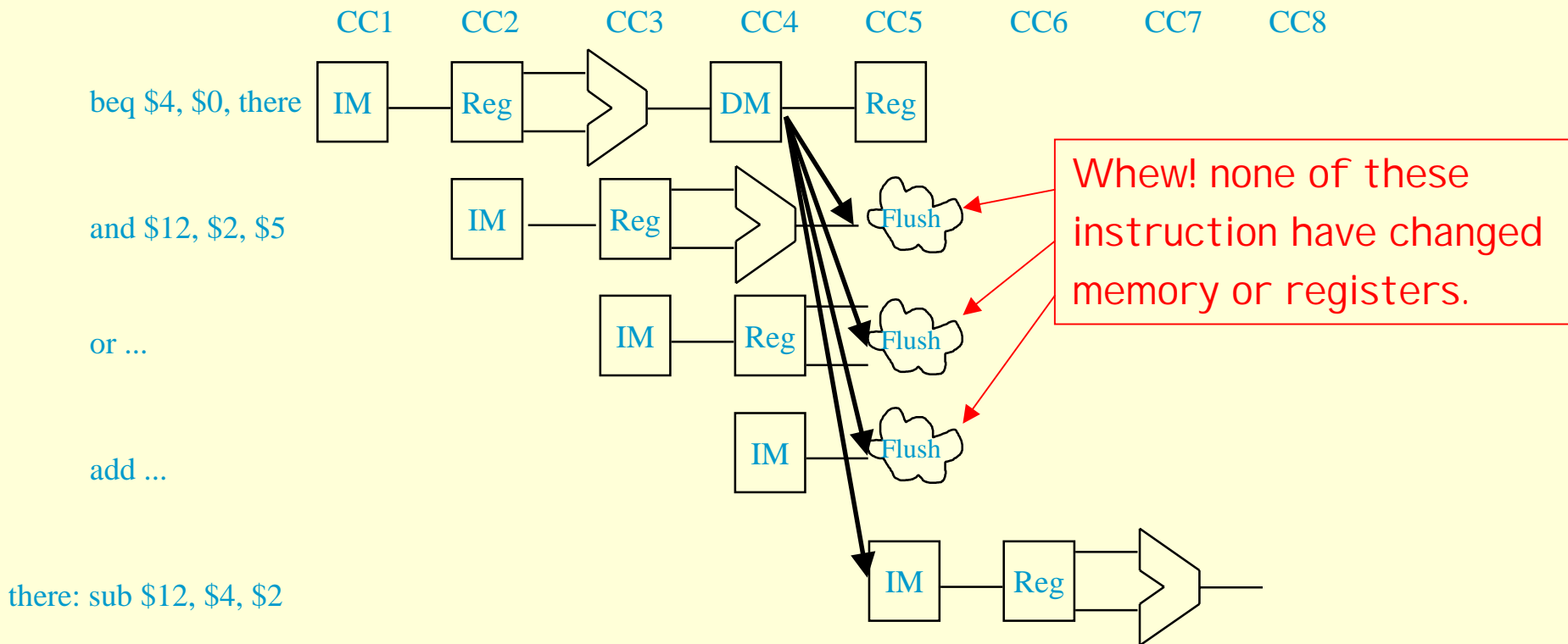
Assume Branch *Not Taken*

- works pretty well when you're right – no wasted cycles



Assume Branch *Not Taken*

- same performance as stalling when you're wrong

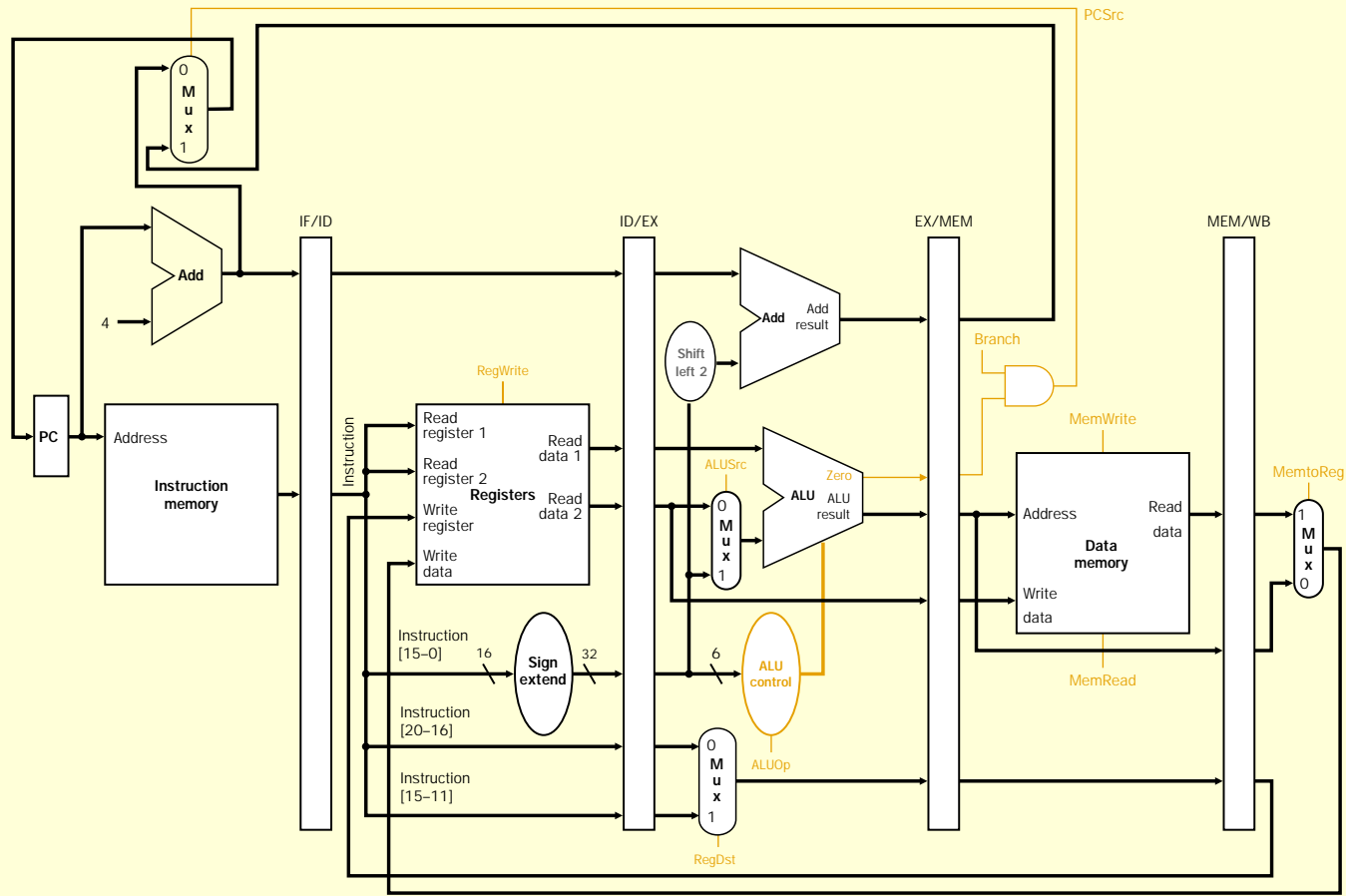


Some other static strategies

1. Assume *backwards* branch is always taken, *forward* branch never is
 - “backwards” = negative displacement field
 - loops (which branch backwards) are usually executed multiple times.
 - “if-then-else” often takes the “then” (no branch) clause.
2. Compiler makes educated guess
 - sets “predict taken/not taken” bit in instruction

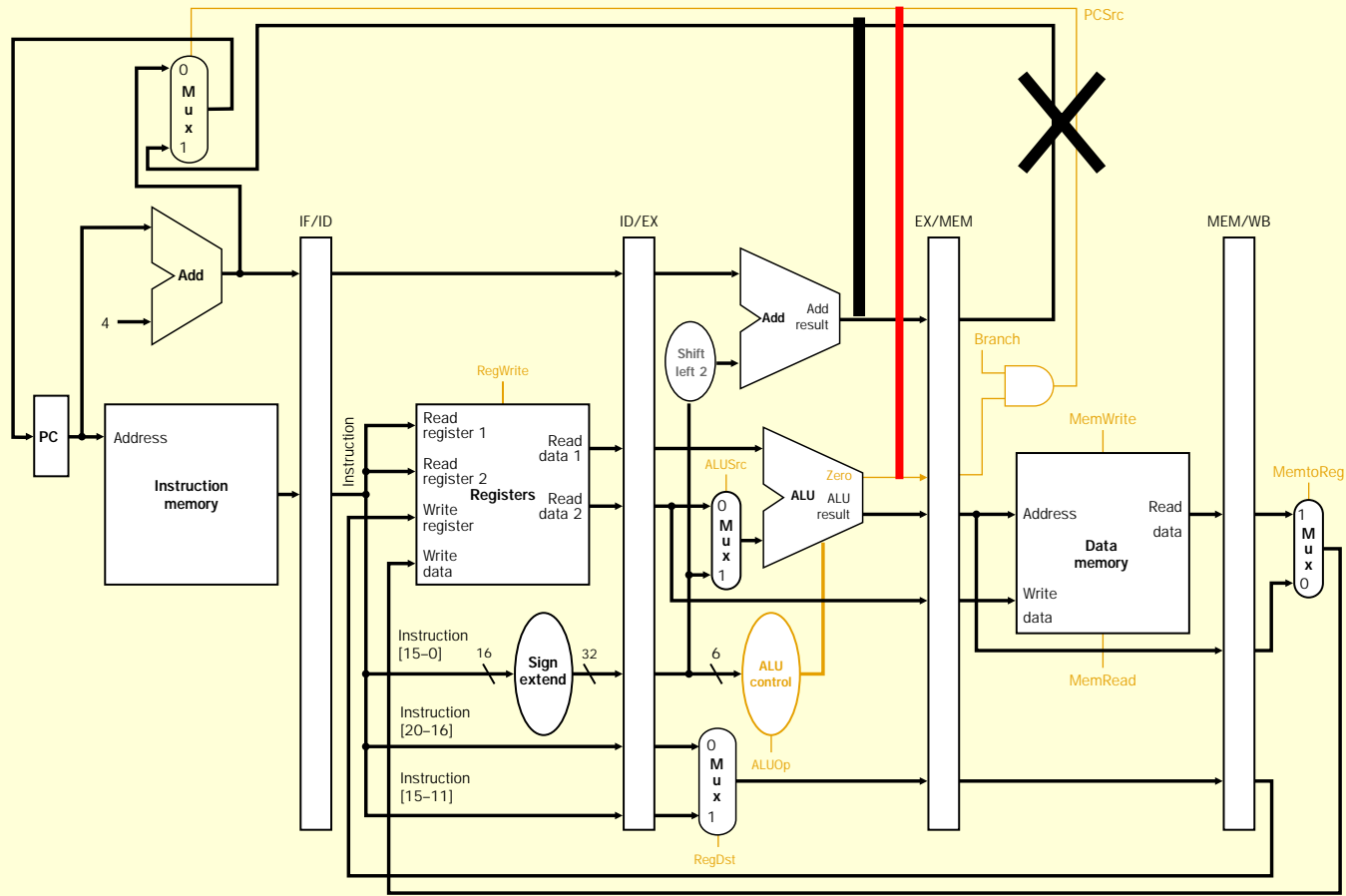
Reducing the Branch Delay

it's easy to reduce stall to 2-cycles

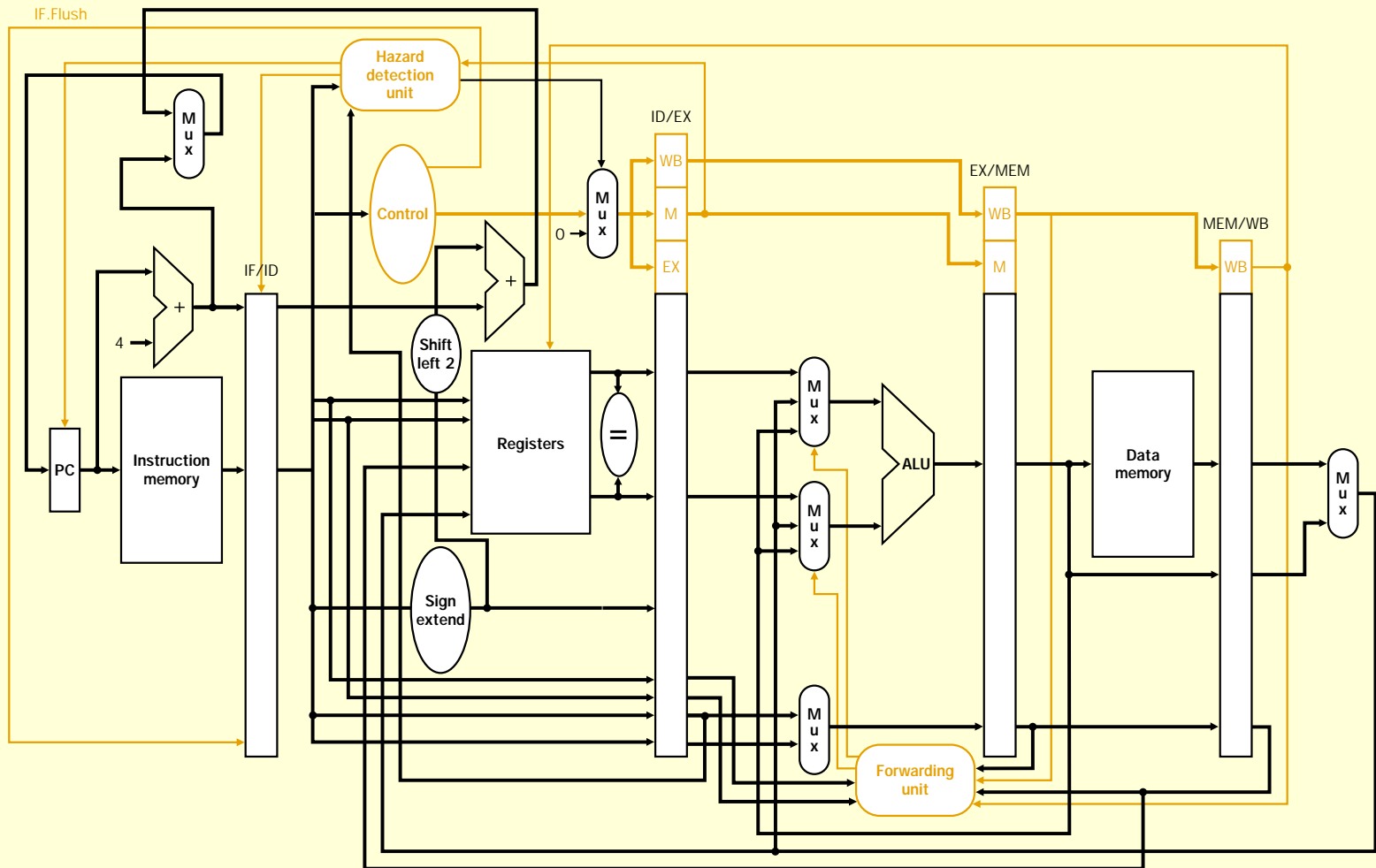


Reducing the Branch Delay

it's easy to reduce stall to 2-cycles

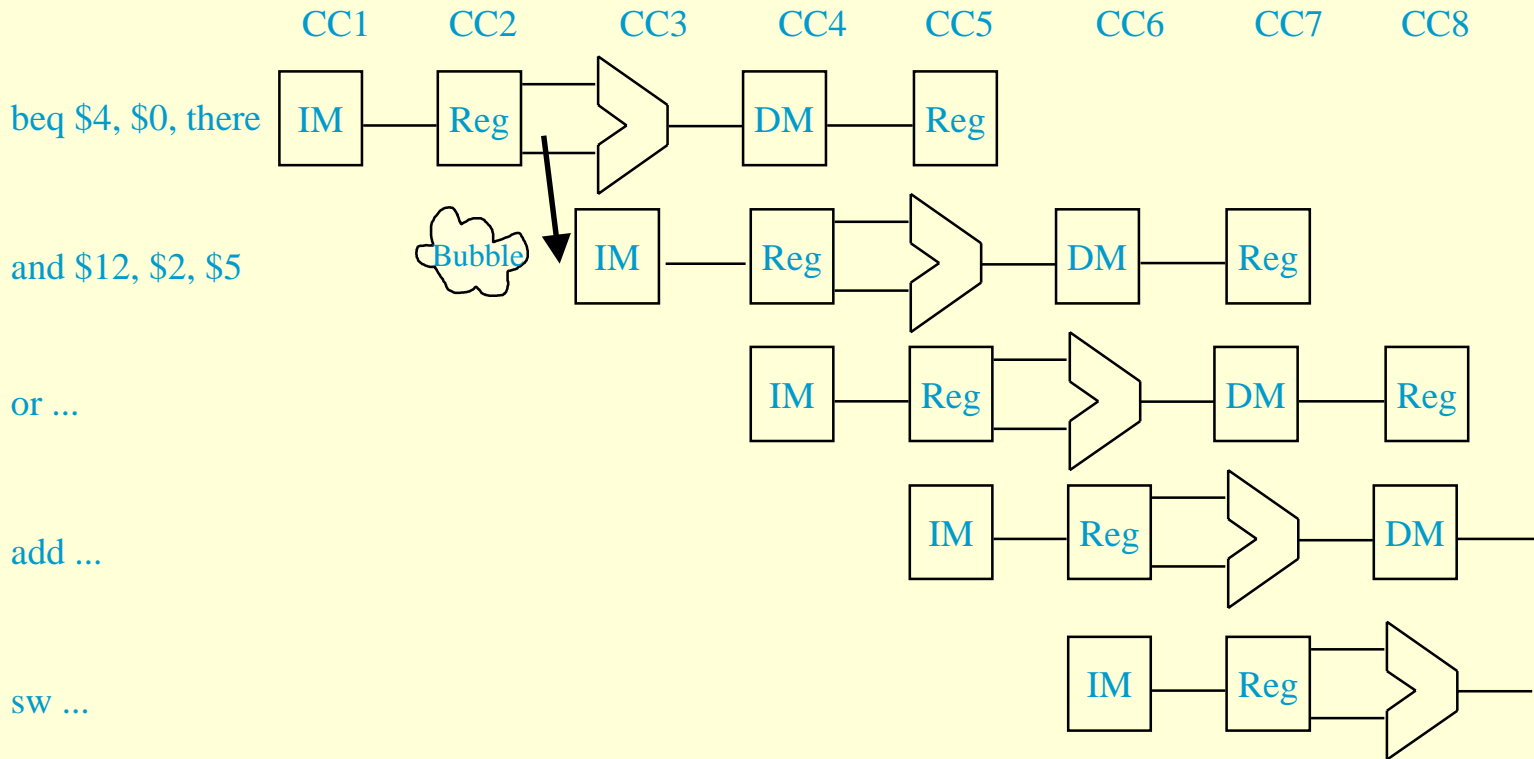


One-cycle branch misprediction penalty



- Target computation & equality check in ID phase.
 - This figure also shows flushing hardware.

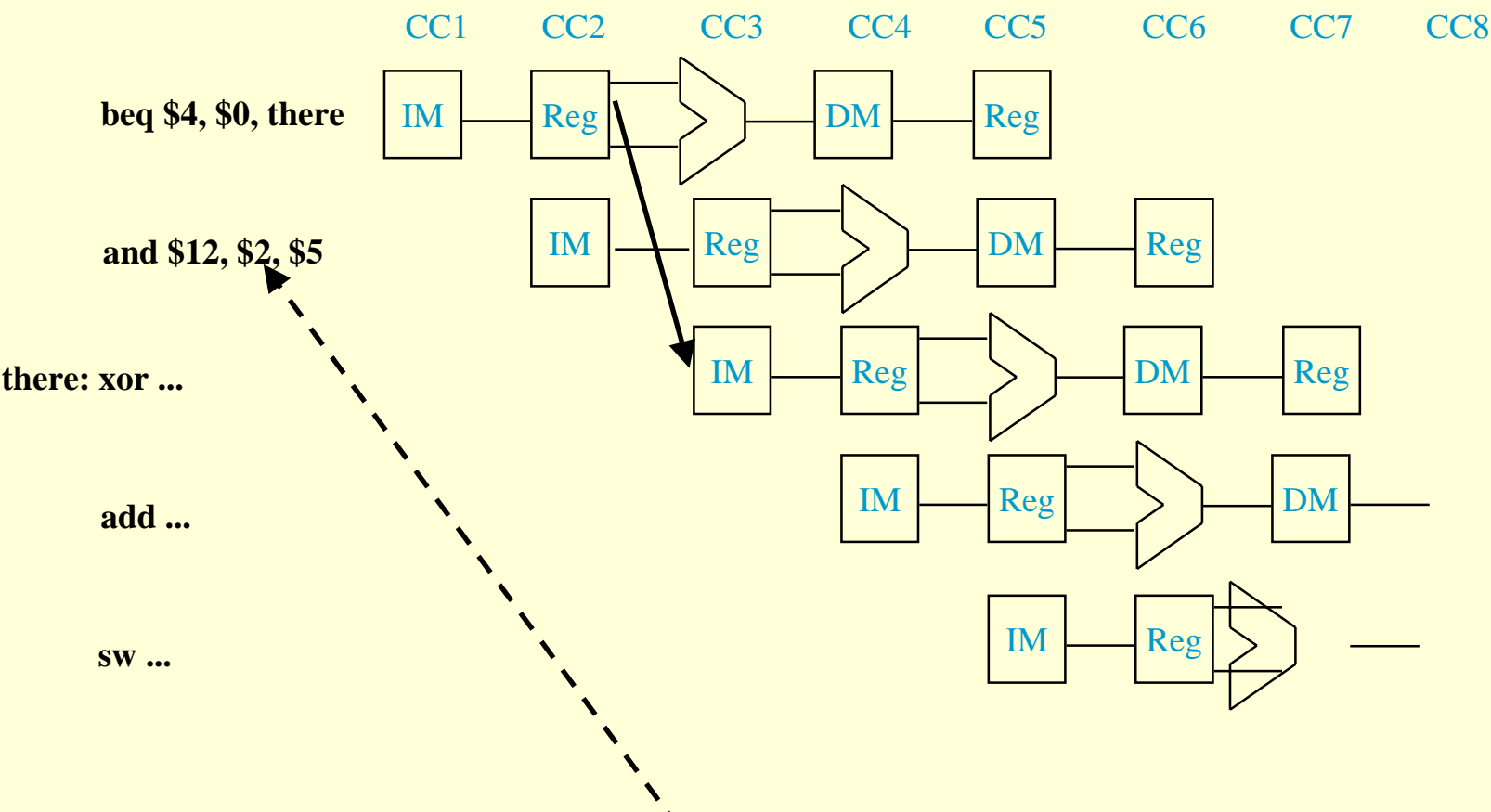
Stalling for Branch Hazards with branching in I D stage



Eliminating the Branch Stall

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

Branch Delay Slot



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

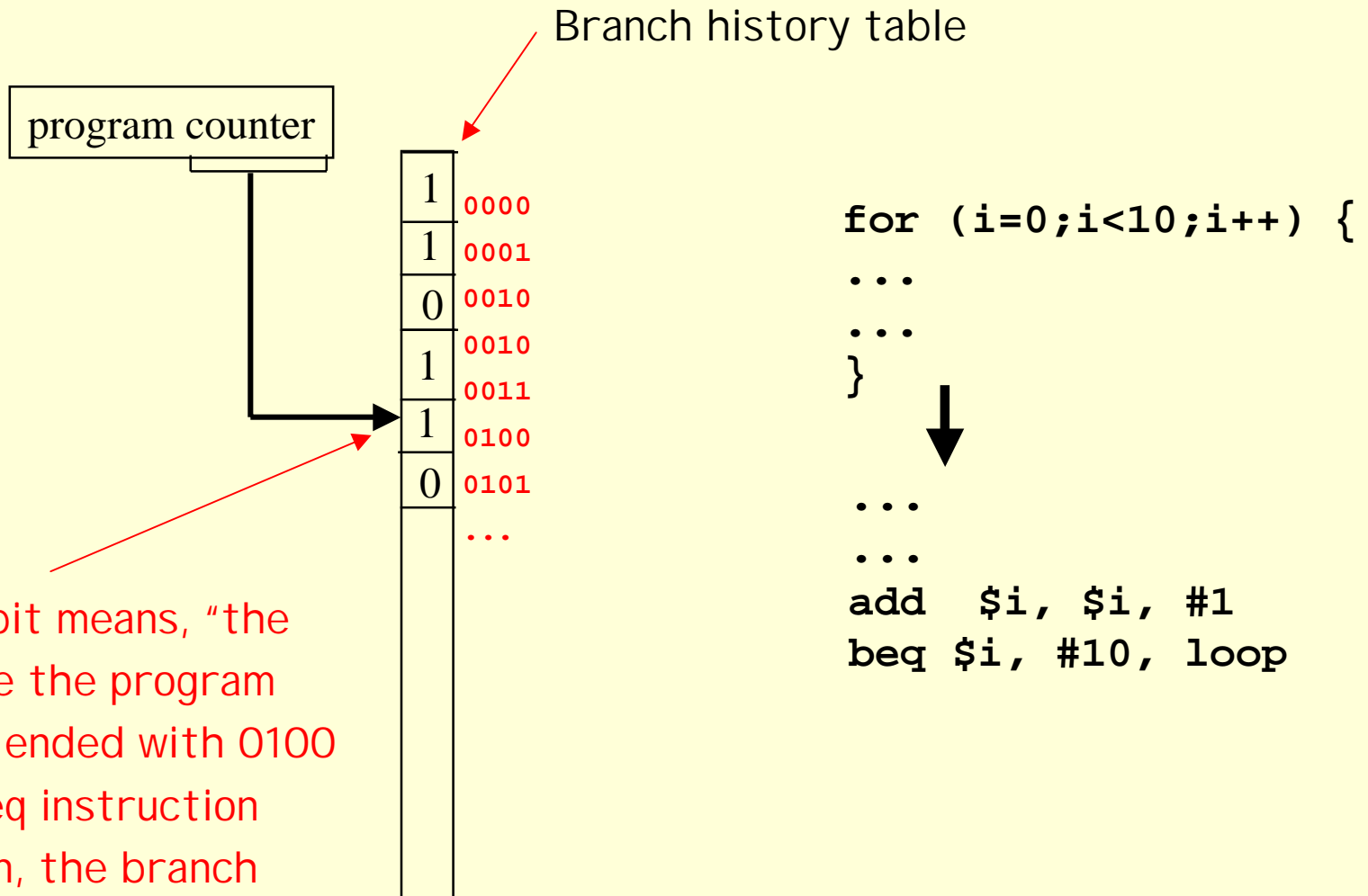
Filling the branch delay slot

- The branch delay slot is only useful if you can find something to put there.
 - Need earlier instruction that doesn't affect the branch
- If you can't find anything, you must put a *nop* to insure correctness.
- Worked well for early RISC machines.
 - Doesn't help recent processors much
 - E.g. MIPS R10000, has a 5-cycle branch penalty, and executes 4 instructions per cycle.
- Meanwhile, delayed branch is a permanent part of the ISA.

Branch Prediction

- Static branch prediction isn't good enough when mispredicted branches waste 10 or 20 instructions .
- Dynamic branch prediction keeps a brief history of what happened at each branch.

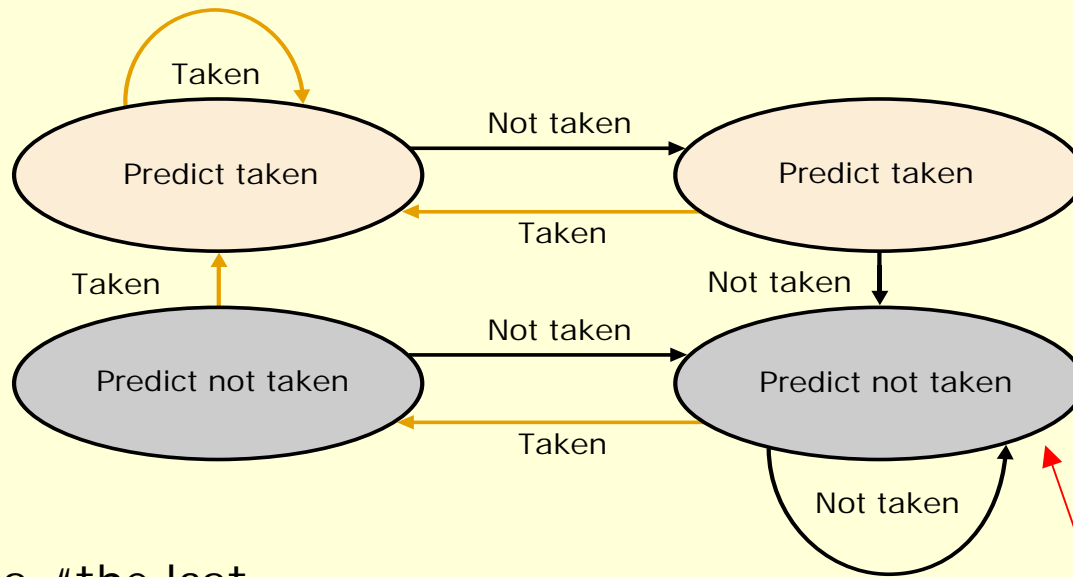
Branch Prediction



This '1' bit means, "the last time the program counter ended with 0100 and a beq instruction was seen, the branch was taken." Hardware guesses it will be taken again.

Two-bit predictors are even better

(Branch prediction is a hot research topic)



this state means, "the last two branches at this location were taken."

This one means, "the last two branches at this location were *not* taken."

Branch Hazards -- Key Points

- Branch (or control) hazards arise because we must fetch the next instruction before we know if we are branching or not.
- Branch hazards are detected in hardware.
- We can reduce the impact of branch hazards through:
 - computing branch target and testing early
 - branch delay slots
 - branch prediction – static or dynamic

Computer of the Day

- 1963: Seymour Cray's CDC 6600
 - First supercomputer. 10 MHz clock. (Individual transistors!)
 - Also first Register-Register (i.e. Load-Store) ISA machine.
 - 10 multicycle functional units in the "Central" processor
 - float + (4 cycle), 2 float x 's (10 cyc), float divide (29 cyc), assorted boolean & integer units (most 3 cyc), branch (9 cyc)
 - Unrelated instructions can be executed concurrently.
 - 10 "Peripheral & Control" processors for I/O
 - 60-bit words, 15-bit 3-address instructions (also has 30-bit inst's)
 - 60-bit general registers, plus 18-bit address & index regs
 - 8 word instruction cache (no data cache)
 - 28 or fewer instructions in loop for peak speed
 - Programmer's goal - provably optimal code

Quiz #2

- You did well ...
 - Top quartile: 34 (out of 40)
 - Median: 31.5
 - Third quartile: 27
- I still grade on a curve ... but average is about “B”
- Nobody got #6 right!
 - Yes, you can eliminate a MUX on the register write port
 - Yes, you need a MUX on the second register read port
 - But how do you set this MUX on the 2nd cycle?
 - If you choose “**rt**”, then you can’t execute R-type in 4 cycles.
 - If you choose “**rd**”, then you can’t execute **beq** in 3 cycles.
 - If you make it depend on instruction, you slow down control !!