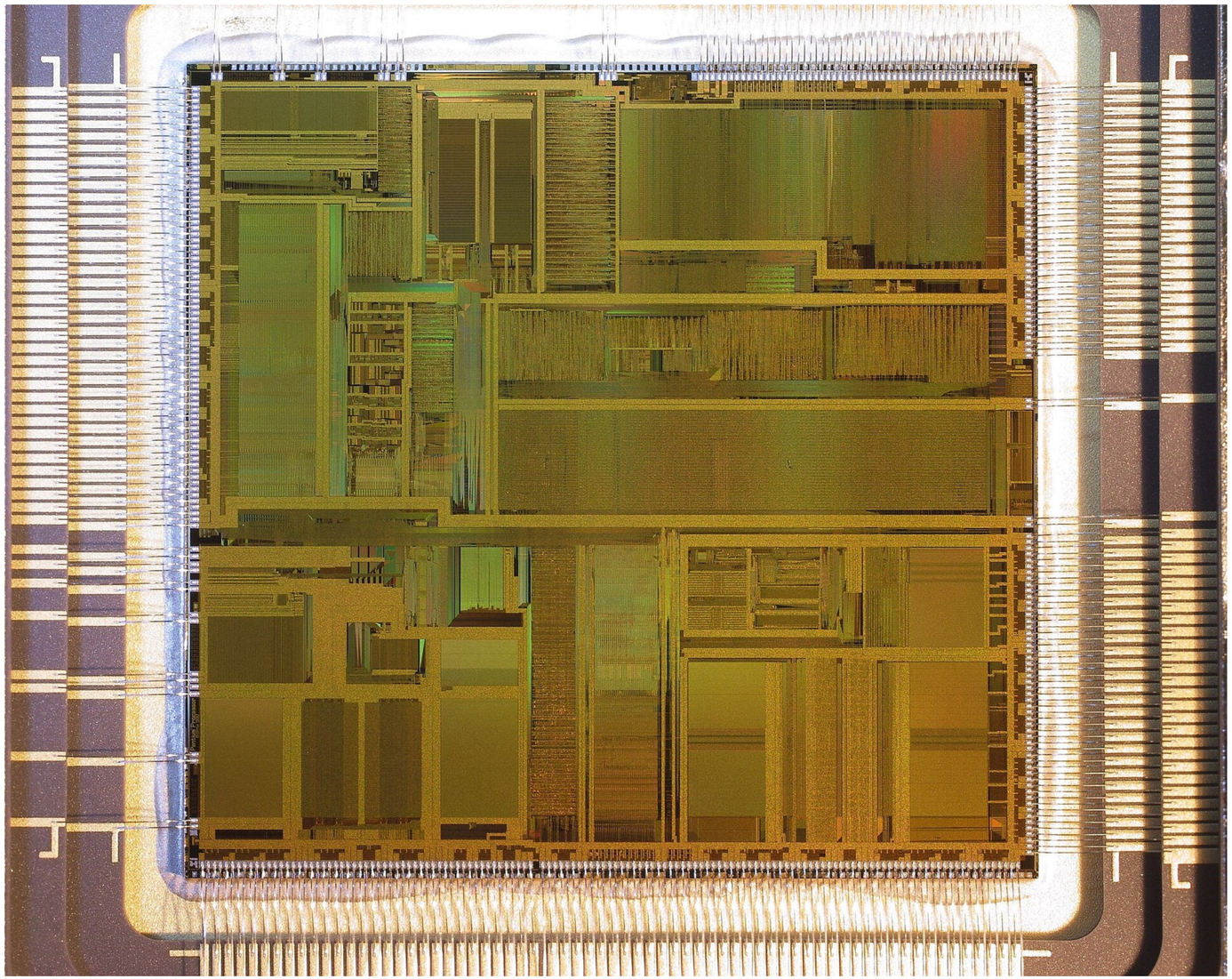
Introducing “Transient Execution”

Homework 3 released (due Nov 21)!



Intel Pentium (P5) from 1989

Two parallel pipelines to handle multiple instructions at the same time!

First processor chip with dynamic branch prediction

Image credit: https://en.wikipedia.org/wiki/Pentium_(original)

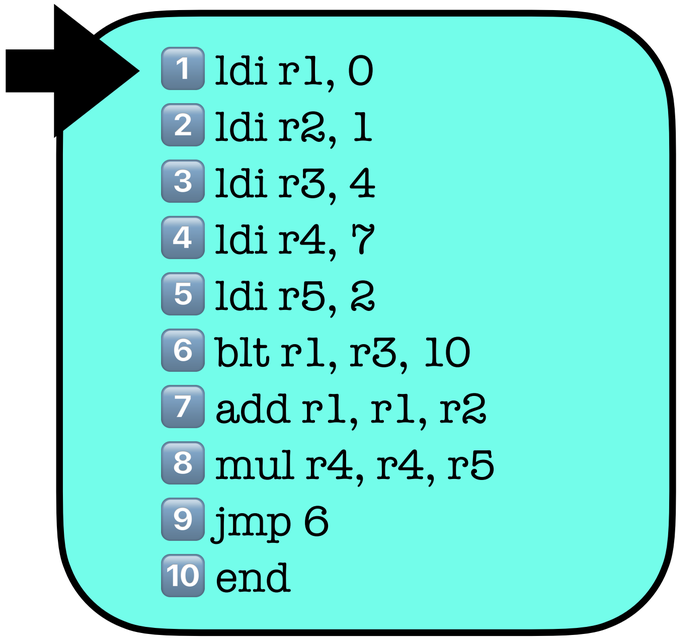
Outline

- Strategies to work around control hazards
- Pipelining with predicted PCs
- Introducing the branch-target buffer

Control Hazards

- If it takes several cycles to know what the appropriate next program counter value should be, then it may be the case that our processor executes instructions that are incorrect relative to the expected program behavior
- Executing instructions on the incorrect side of a branch is called a *control hazard* as it will lead to incorrect instructions in the pipeline
- If our processor implements a hazard checking unit, then the unit must also check to see if incorrect instructions are in the pipeline due to control hazards and appropriately stall/bubble the stages

Strategies to Workaround Control Hazards



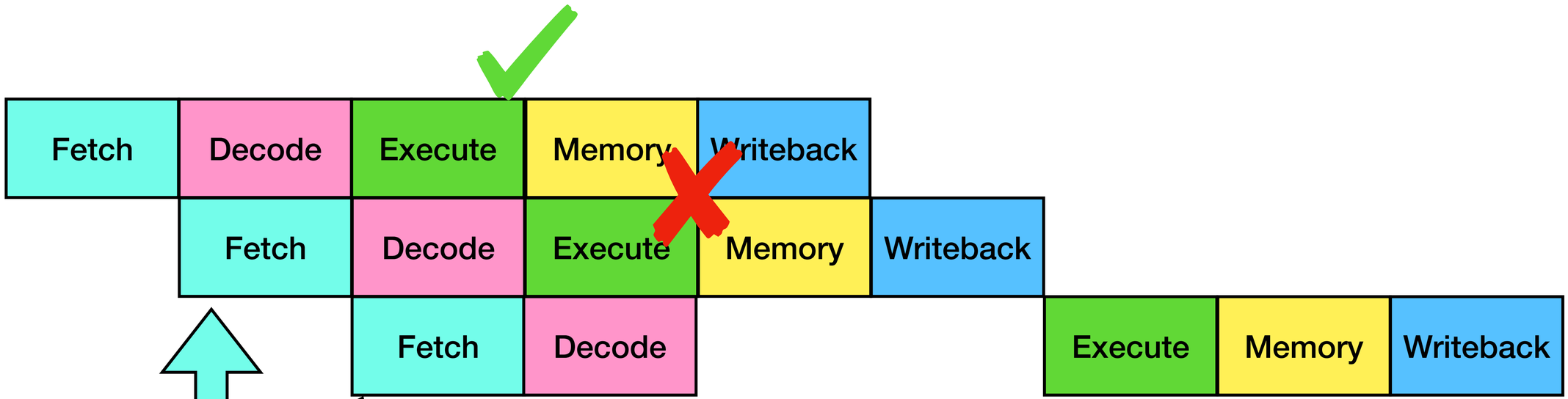
We don’t know the answer!

We could wait or guess!

What to Fetch Next

- Seeing as we will not know the *true value* of the PC until after the branch is evaluated (which could be as late as after Execute), it may seem as though we have to delay our next fetch until we know the PC
- Alternatively, we can try optimistically making an *assumption* or *prediction* about what the next PC will be and correct ourselves later if we were wrong (we will talk about how to do this next week)
- For example, we could *assume* that the instruction is always either not a branch or that the branch is *not taken* whenever we reach a control instruction
- Alternatively, we could track certain behaviors to try to *predict* whether the branch was or was not taken ➡️ we will come up with strategies to implement branch predictors on Wednesday!

Pipelining the Branching Data Path (attempt 4)



Optimistically fetch next instruction right away!

When the branch target is incorrectly computed, then we need to rely on the hazard checking logic

If we can correctly predict the branch target with high probability, then we can avoid control hazards!

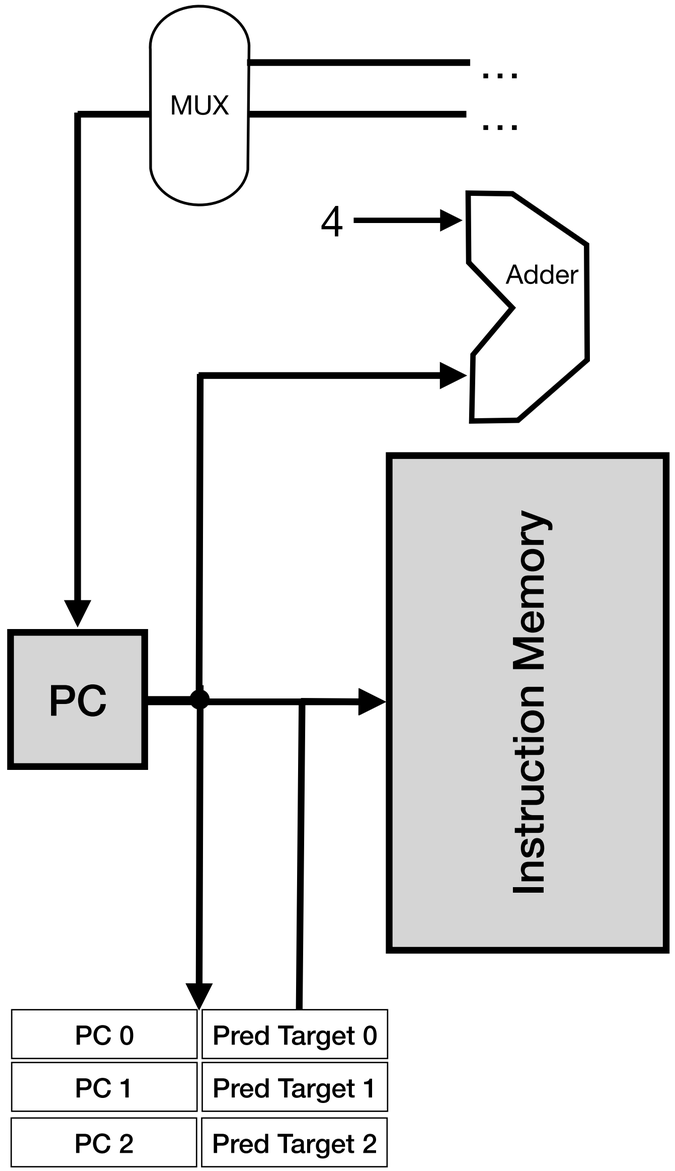
Our processor needs to have a mechanism to roll back state into the last known correct version of the registers!

Chat with your neighbor(s)!

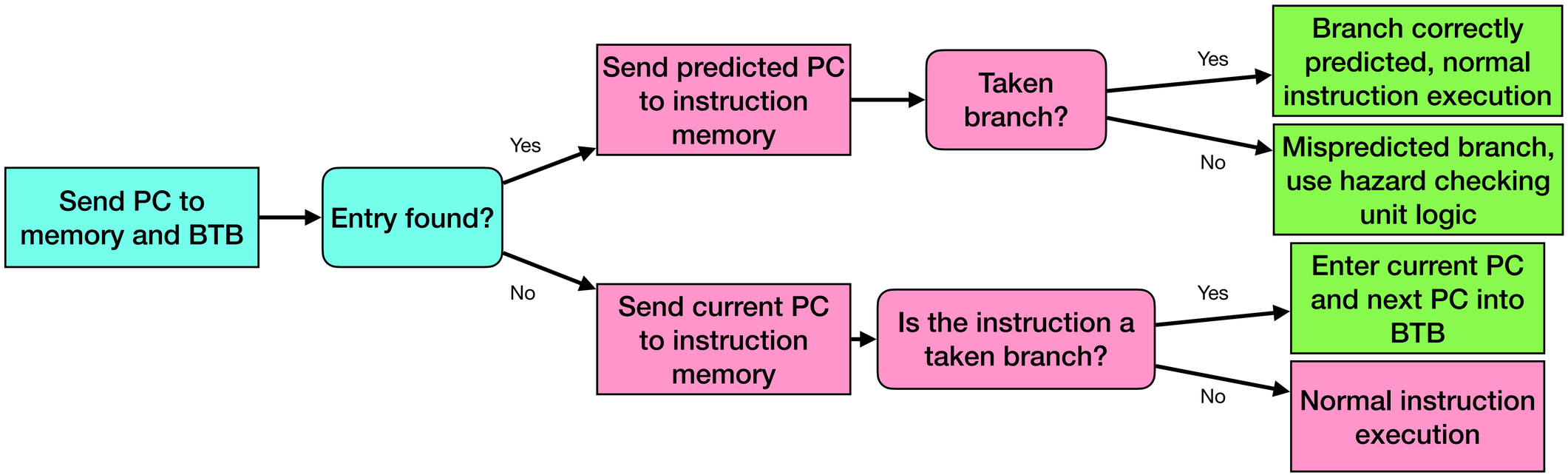
Our goal is to optimistically fetch the next instruction from instruction memory without knowing the next PC. Speculate as to how you may extend the processor data path to account for this unknown next value.

Branch-Target Buffers

- Ultimately, our goal is to minimize the *branch delay* between instructions. If the next PC is correct or the target is known, then our delay is zero ➡️ use dynamic prediction!
- We can do this by maintaining a table of recently used PCs to track what our guesses for the next PC values should be
- This can be referred to as a “branch prediction cache” or a *branch-target buffer* (BTB) and allows the processor to have a new PC value *before* the instruction is decoded
- If no such BTB entry for the current PC exists, then use the current PC as the next value to fetch from the



Data Path Pipeline with BTB



Chat with your neighbor(s)!

Our BTB is essentially a cache for branch instruction targets. Do all hits to the BTB provide the same benefit? Why or why not?

Branch Target Buffers and Instruction Memory

- Recall, instruction memory does not necessarily fetch instructions in a well-defined amount of time
  - ➡️ if the processor issues a fetch for an instruction at `0xff00` then `0xff40`, the memory system may respond with `0xff40` first if it was an instruction cache hit
- We *could* start executing `0xff40` before `0xff00` and later correcting ourselves if a dependence existed between these instructions ➡️ this processor "pre-processing" is called *runahead*
- Implementing runahead uses same mechanism as a control hazard! We are "executing the unknown"

Takeaways

- The processor can add components to its data path that *are not explicitly* related to an individual instruction in the instruction set
- The BTB serves as an auxiliary component that helps predict the next PC for any branch
- We can leverage “execution of the unknown” to speculatively reduce the amount of time spent waiting for sequential program order