332
Advanced Computer Architecture
Chapter 4

Part 2: Branch *Target* Prediction

October 2023

Paul H J Kelly

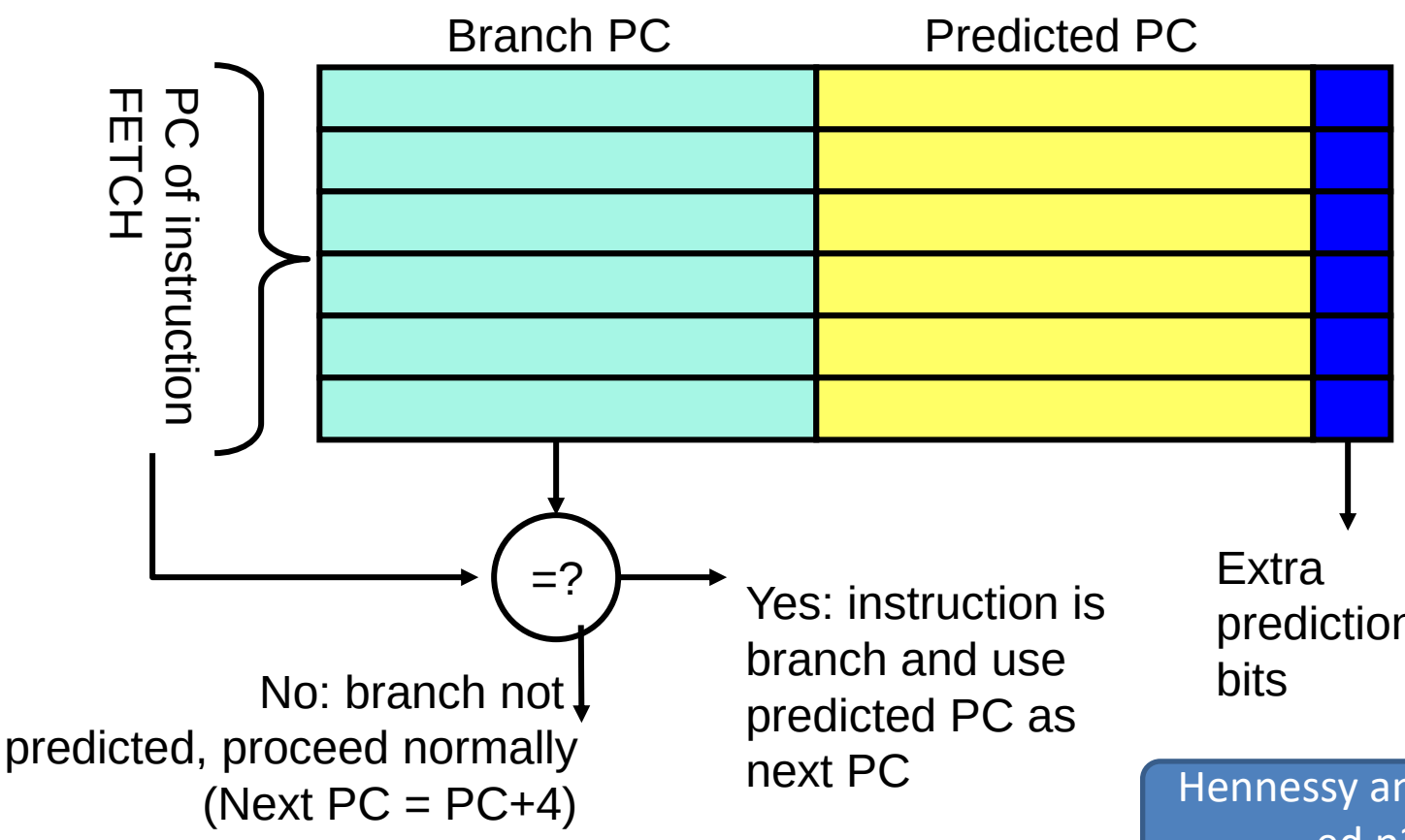
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (4-6th eds), and on the lecture slides of David Patterson's Berkeley course (CS252)*

Branch Prediction - context

- If we have a branch predictor….
  - **We want to fetch the correct (predicted) next instruction without any stalls**
  - **We need the prediction before the preceding instruction has been decoded**
  - We need to predict conditional branches
    - Direction prediction
  - And indirect branches
    - **Target prediction**

Branch Target Buffer

- Need address at same time as prediction
- Especially for indirect branches and virtual method calls
- Note that we must check for branch match, since can't use wrong branch address



BTB is

indexed with low-order PC address bits,

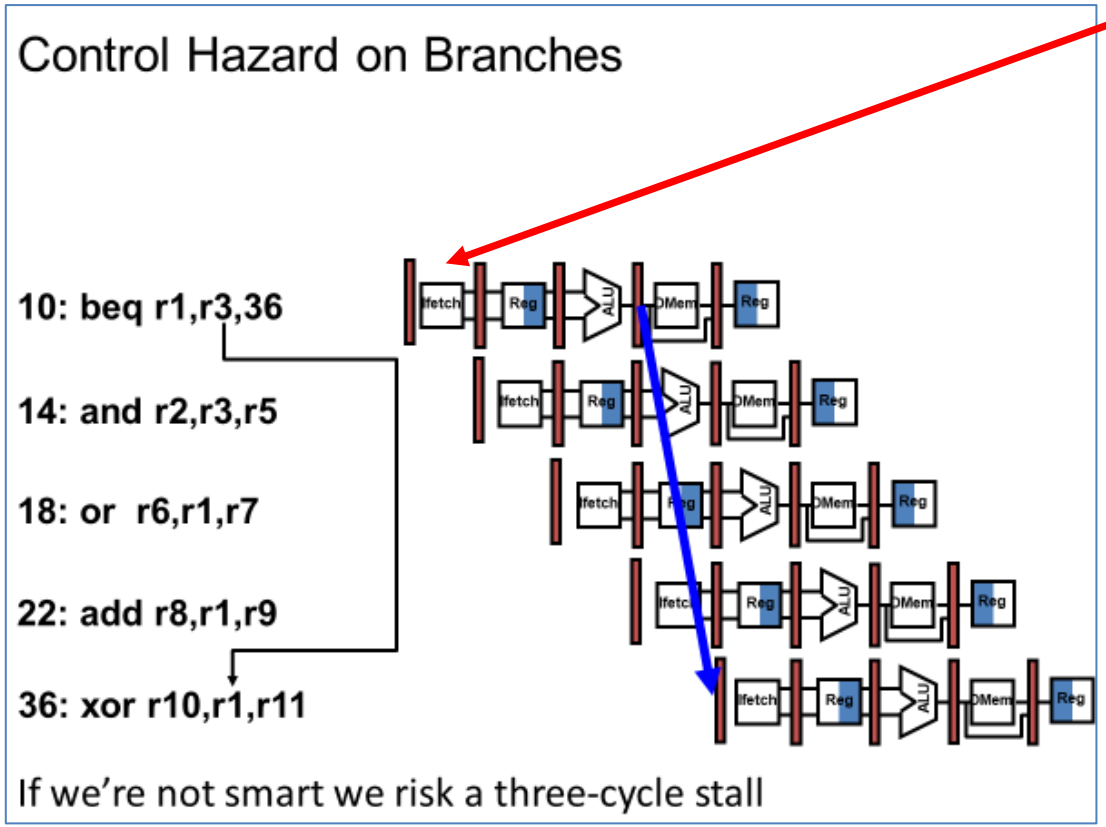
tagged with high-order bits

Hennessy and Patterson 6th ed p228-232

Branch **target** prediction: BTBs

- re: "In order to predict a branch, we need to know that current instruction is branch instruction"

- This doesn't have to be true!



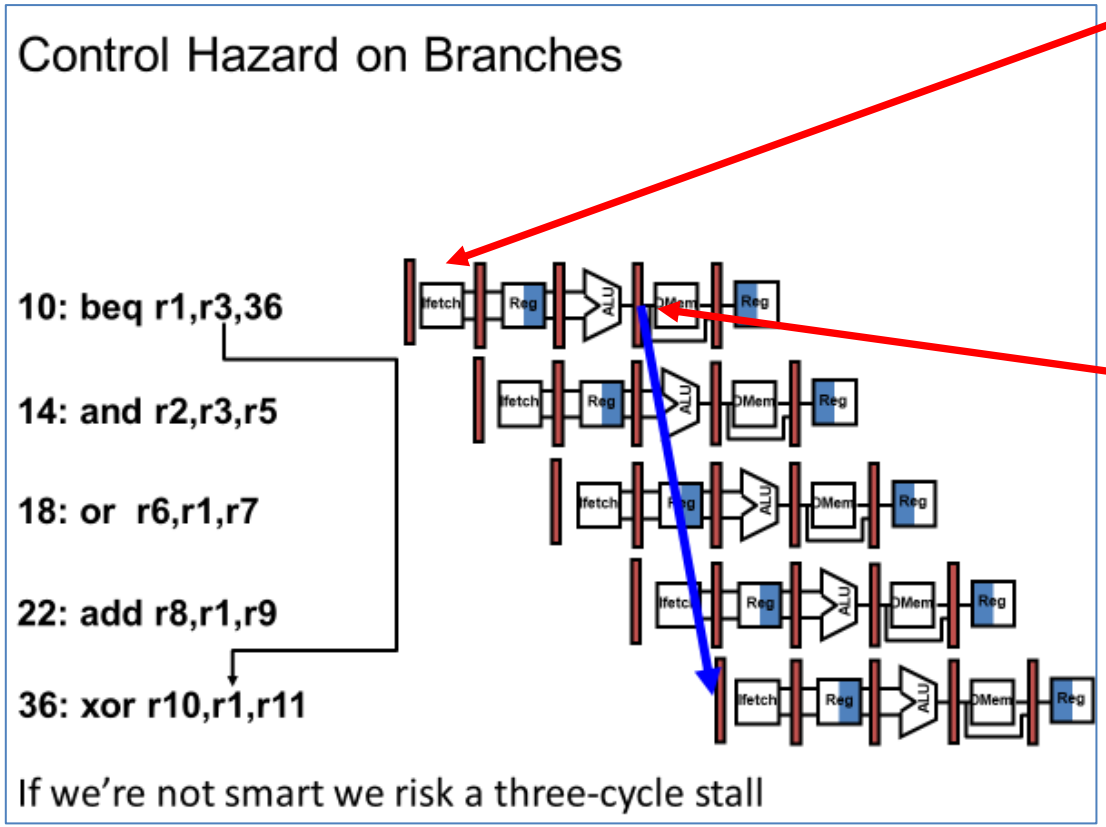
In parallel with every ifetch

Check whether the BTB predicts that the instruction we are fetching *will* be a taken branch

Branch **target** prediction: BTBs

- re: "In order to predict a branch, we need to know that current instruction is branch instruction"

- This doesn't have to be true!



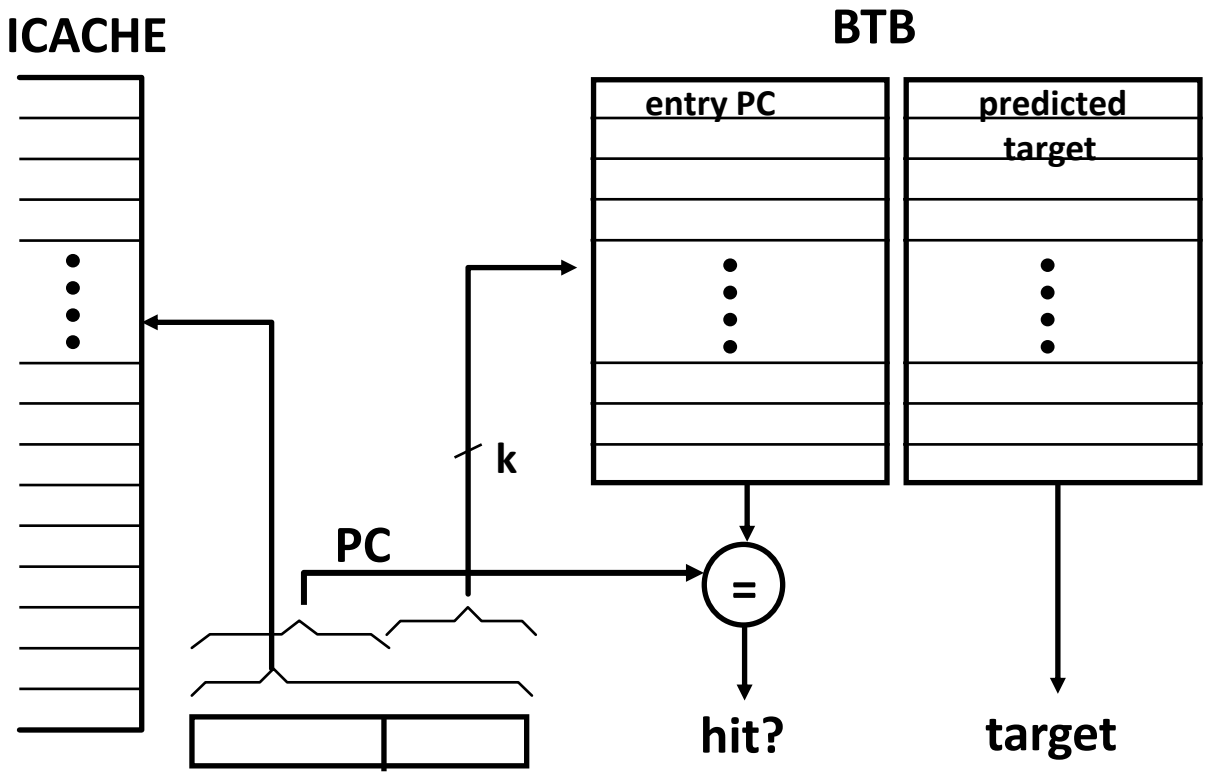
In parallel with every ifetch

Check whether the BTB predicts that the instruction we are fetching *will* be a taken branch

When a **taken** branch is committed, we update the BTB with the branch's target address (and with the tag of the address of the branch instruction).

Branch Target Buffer (BTB)

- Cache of branch target addresses accessed in parallel with the I-cache in the fetch stage
- Updated only by taken branches (the direction-predictor determines whether BTB is used)
- If BTB hit and the instruction is a predicted-taken branch
  - target from the BTB is used as fetch address in the next cycle
- If BTB miss or the instruction is a predicted-not-taken branch
  - PC+N is used as the next fetch address in the next cycle



BTB is

indexed with low-order PC address bits,

tagged with high-order bits

(Note: we could use an n-way set-associative design here)

Credit: Onur Mutlu

Target prediction: recall the 5-stage MIPS pipeline

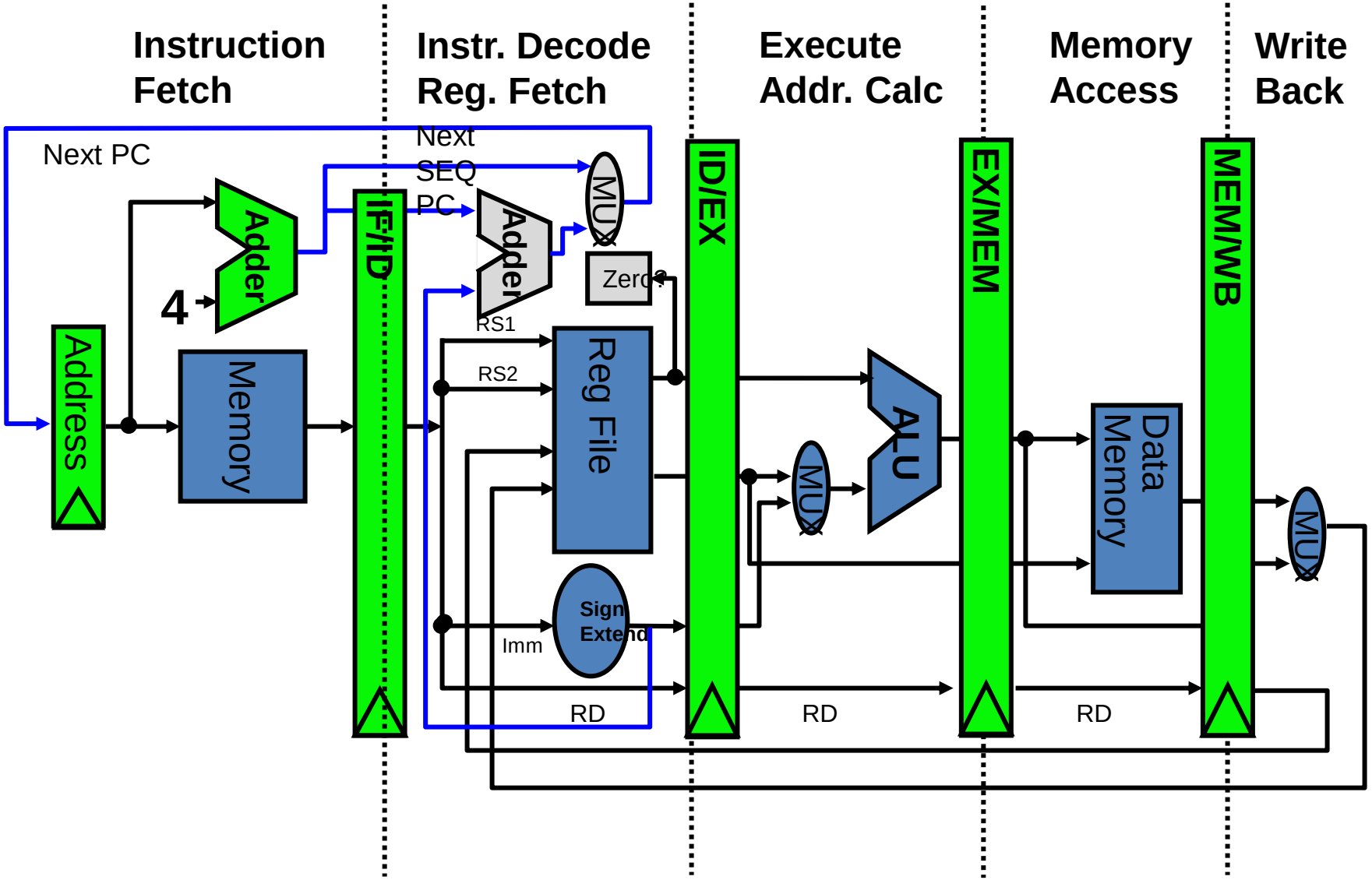
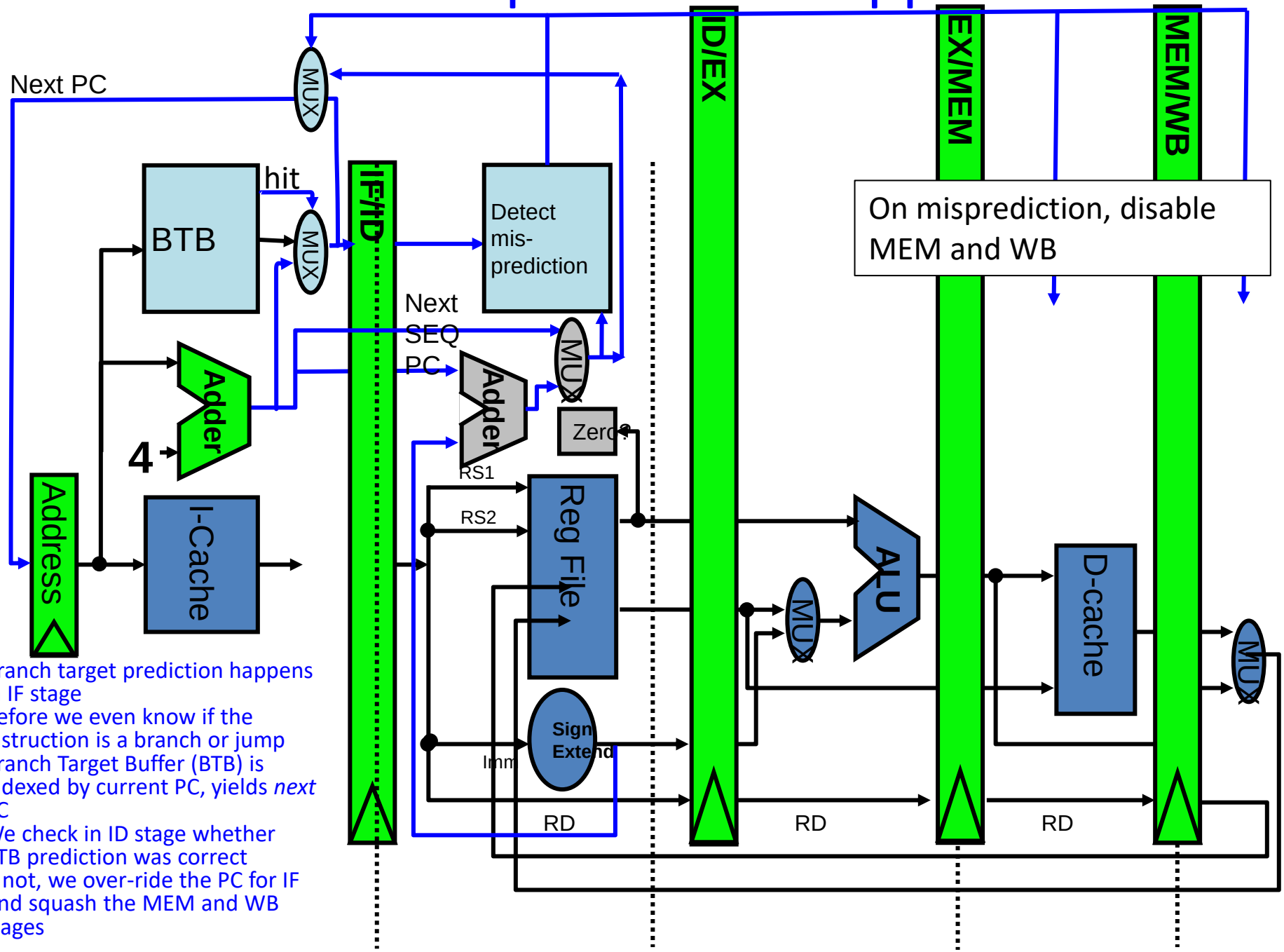


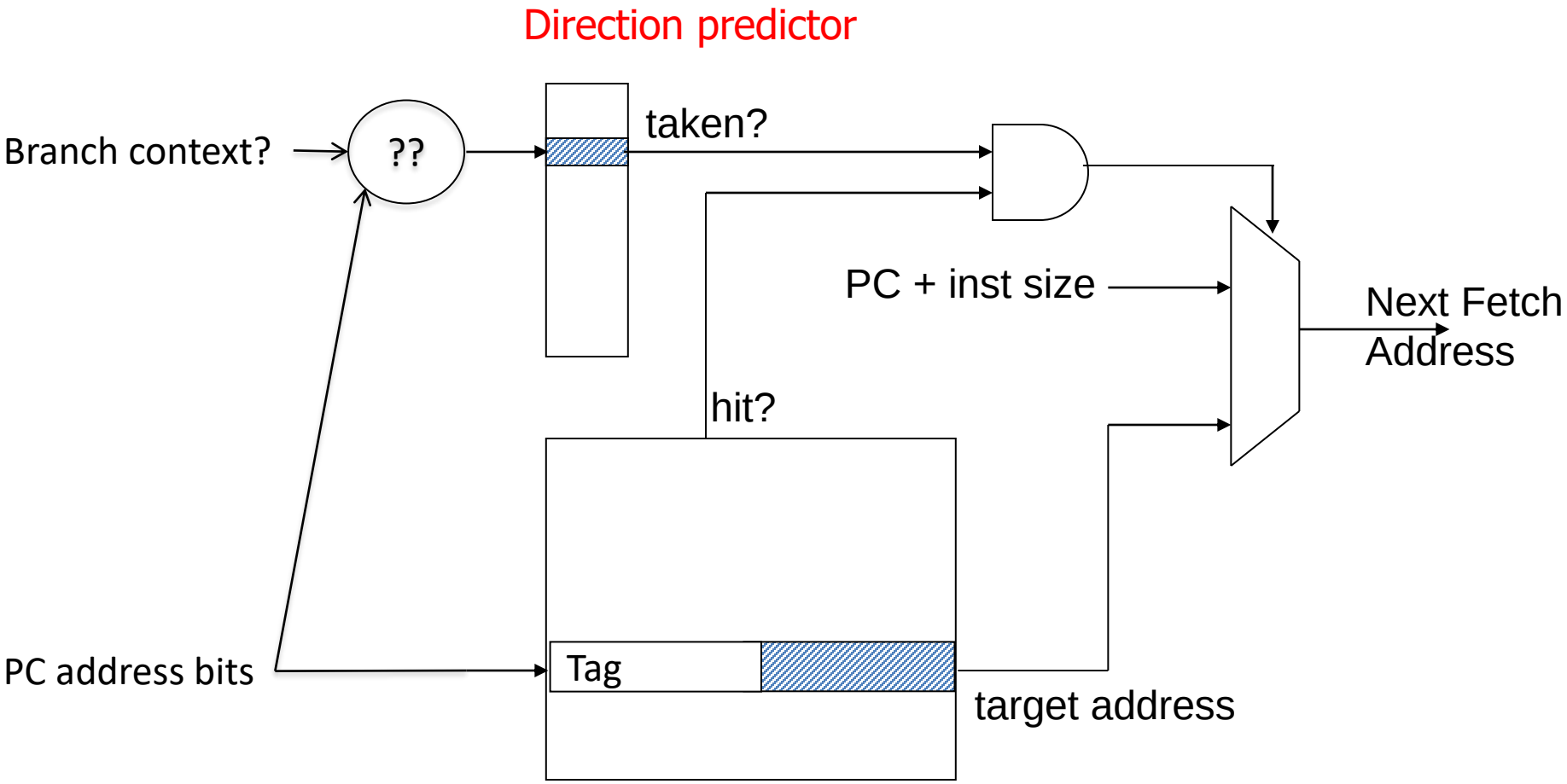
Figure 3.22, page 163, CA:AQA 2/e

Where does branch prediction happen?



- Branch target prediction happens in IF stage
- Before we even know if the instruction is a branch or jump
- Branch Target Buffer (BTB) is indexed by current PC, yields *next* PC
- We check in ID stage whether BTB prediction was correct
- If not, we over-ride the PC for IF
- And squash the MEM and WB stages

Combining BTB with direction Prediction

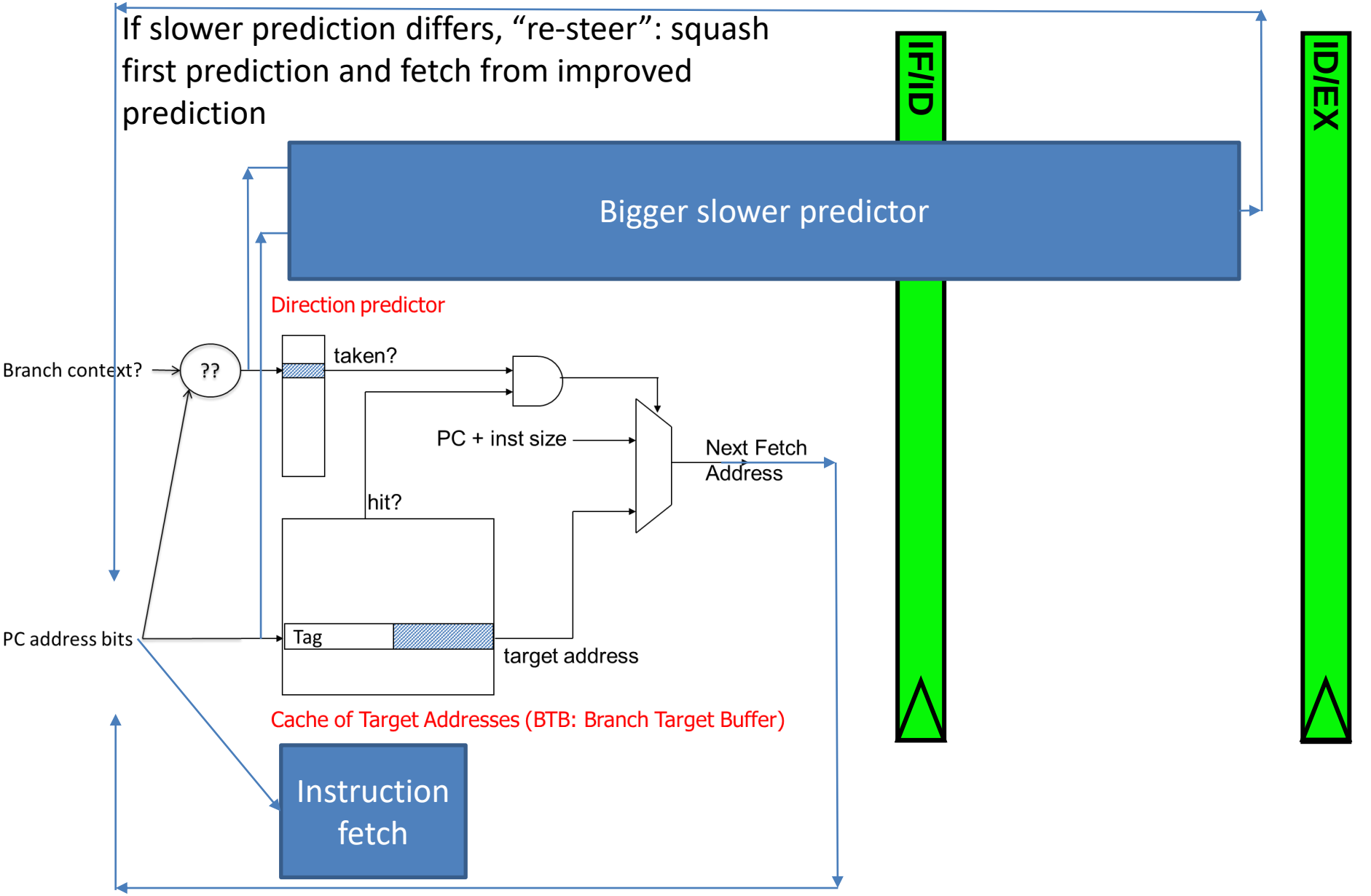


Cache of Target Addresses (BTB: Branch Target Buffer)

(What if branch is predicted-taken but BTB miss?)

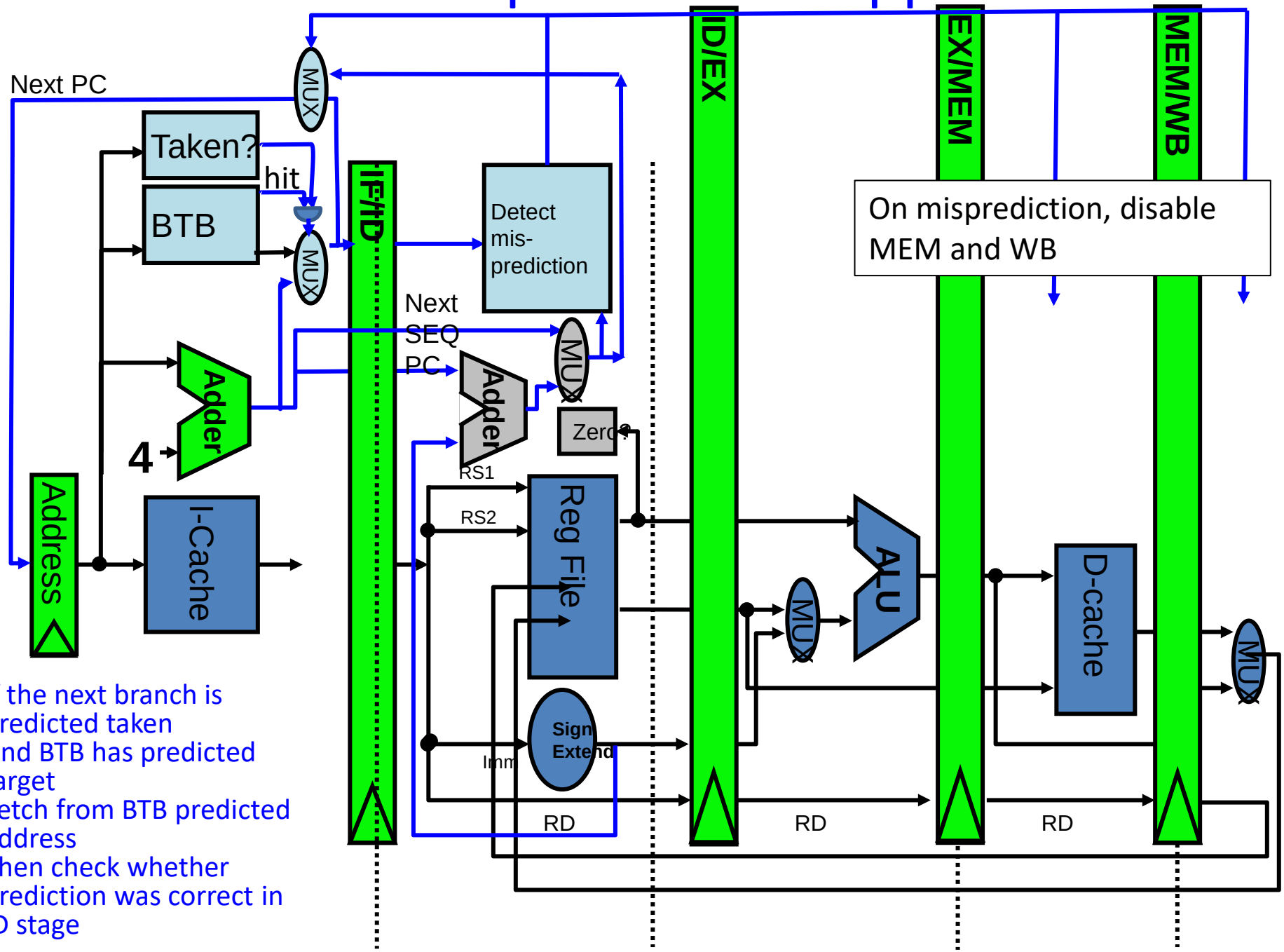
Credit: Onur Mutlu

Combining fast simple predictor with slower bigger predictor



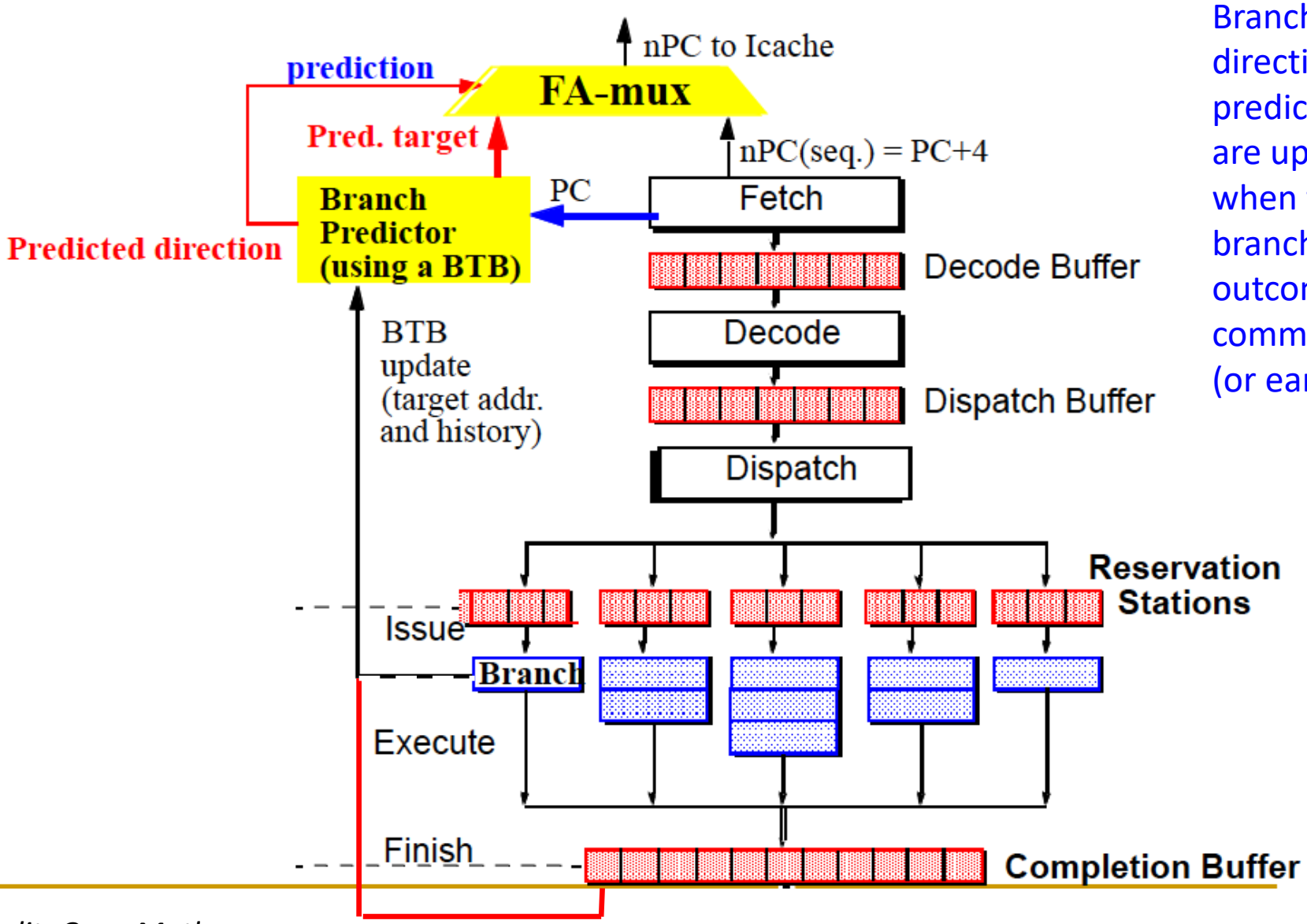
(What if branch is predicted-taken but BTB miss?)

Where does branch prediction happen?



- If the next branch is predicted taken
- And BTB has predicted target
- Fetch from BTB predicted address
- Then check whether prediction was correct in ID stage

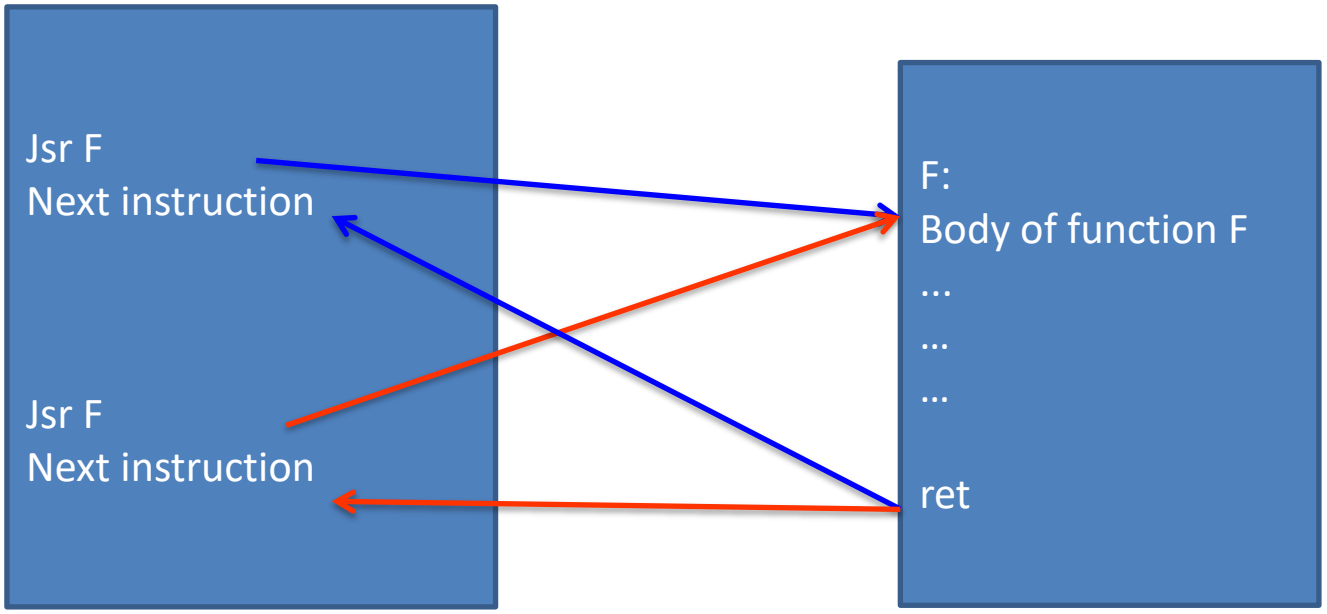
Updating the branch prediction



BTB and Branch direction prediction are updated when the branch outcome is committed (or earlier?)

Credit: Onur Mutlu

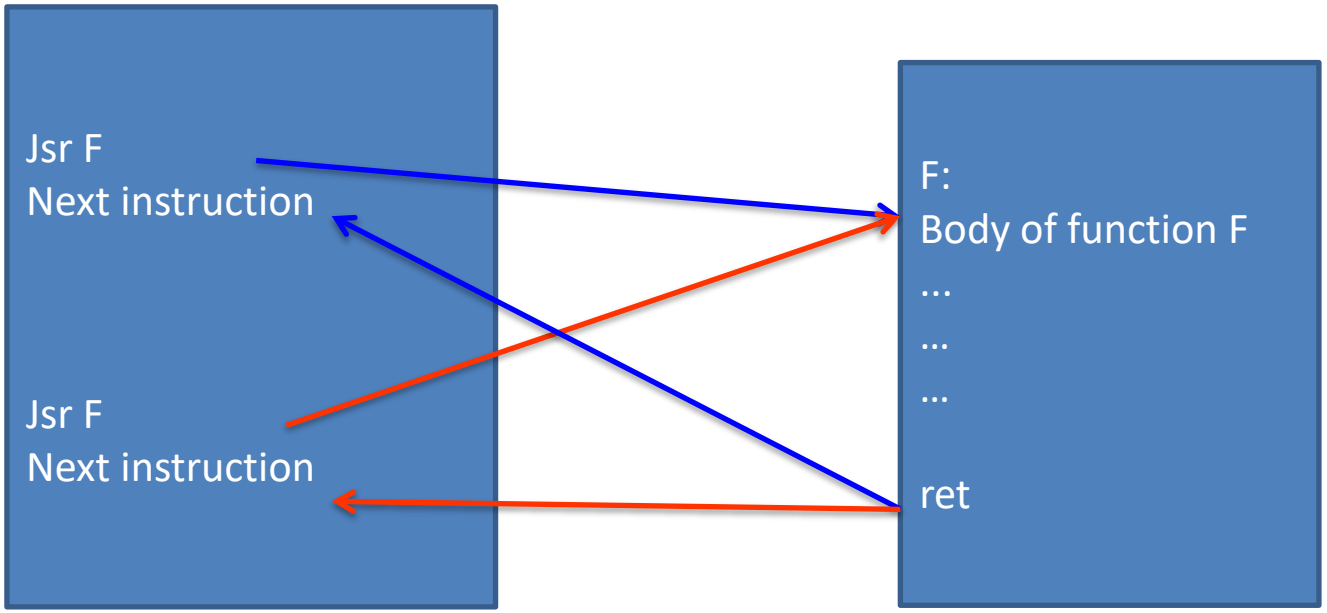
Return addresses



- A function might be called from different places
- In each case it must return to the right place
- Address of next instruction must be saved and restored

Hennessy and Patterson 6th ed p232-233

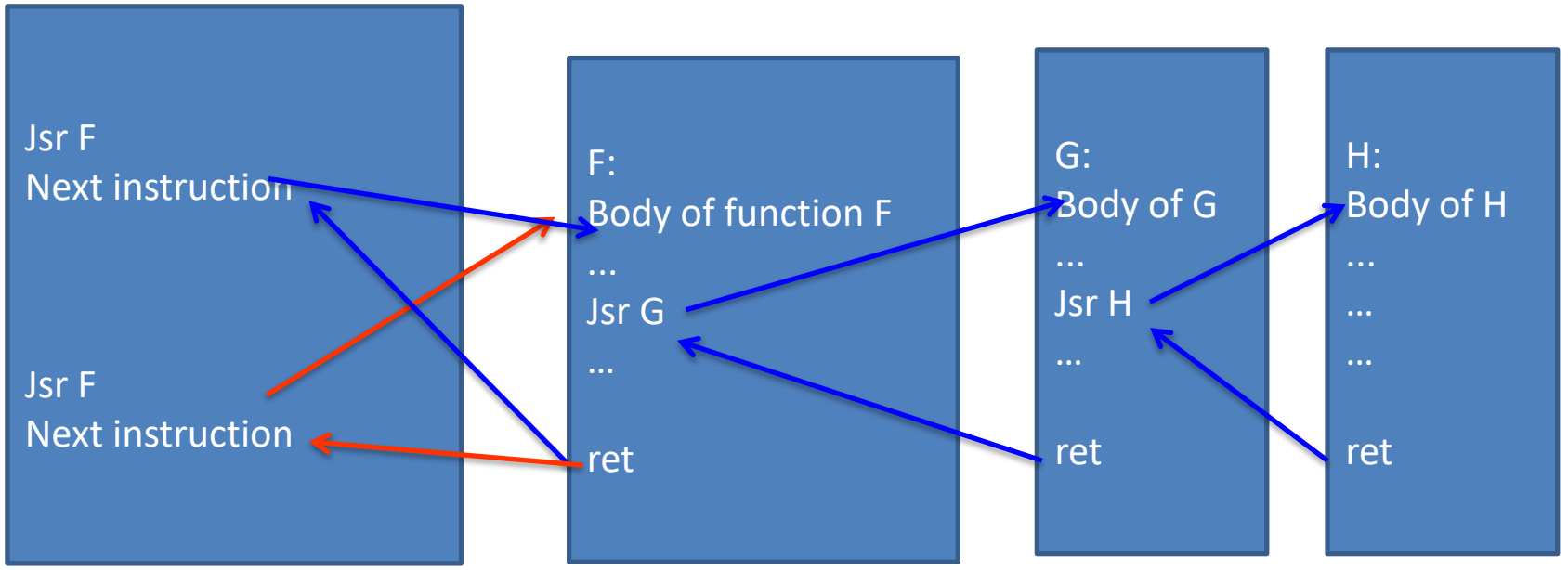
Return addresses



- jsr must save return address somewhere
- On x86 jsr pushes return address onto stack
- ret jumps to the address on the top of the stack

- On MIPS, “jal F” (jump-and-link) jumps to F, and stashes the current PC in a special register $ra.
- Function returns with an indirect jump “jr $ra”
- If the function body has other calls, compiler must push $ra to the stack

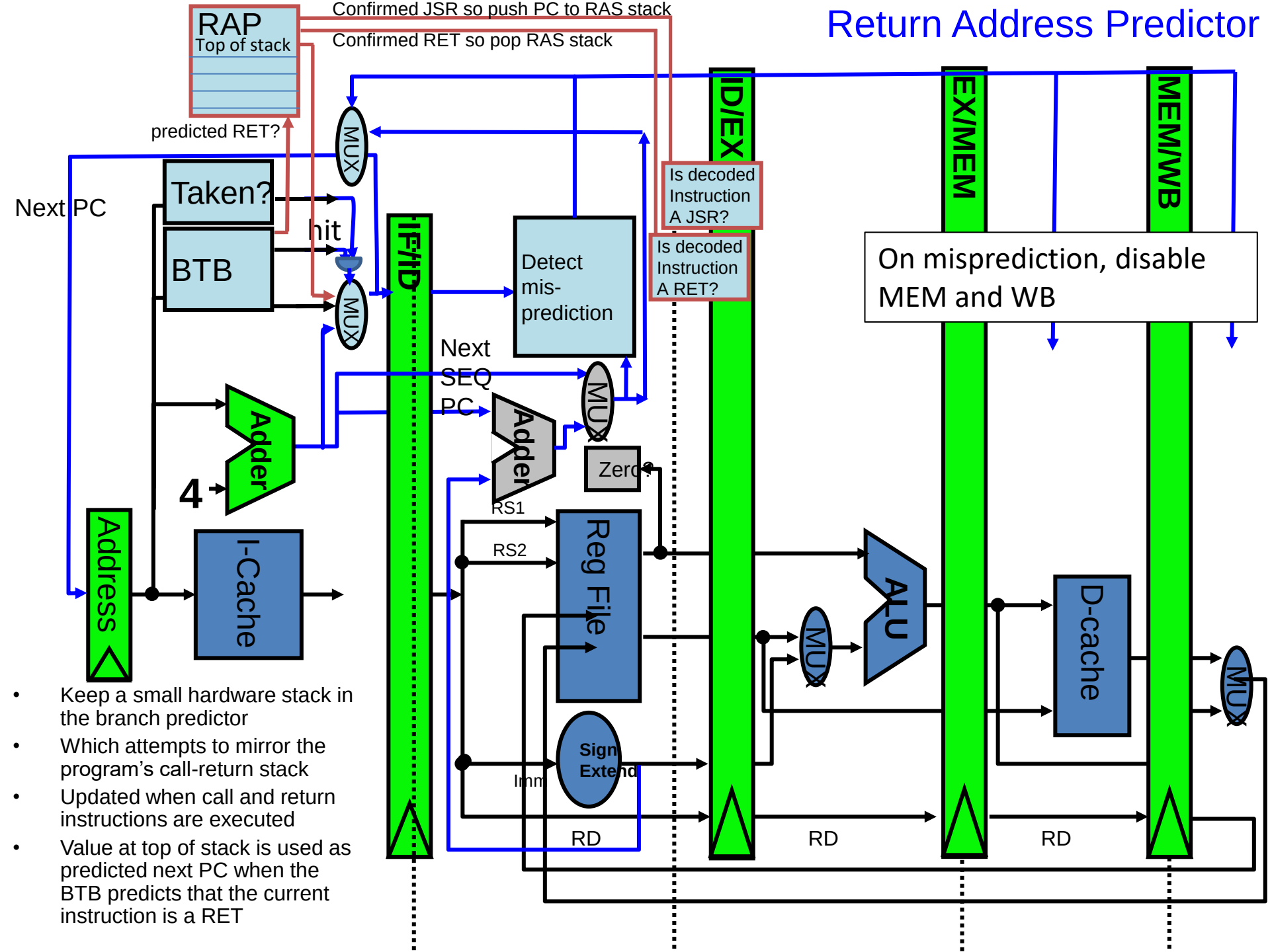
Return addresses



- Return addresses form a stack (even if they are stored in registers)
- They *should* be easy to predict!

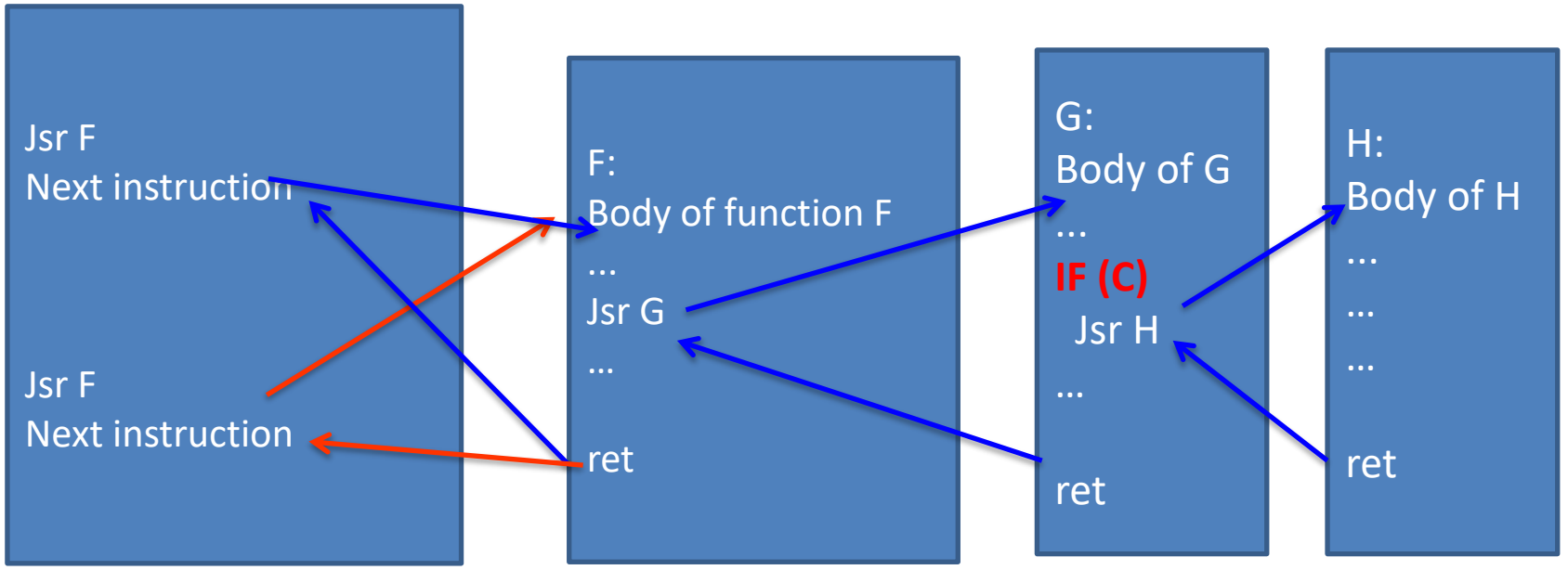
- We need to add *another* branch target predictor
- That maintains a hardware stack of return addresses
- Presumably a small stack

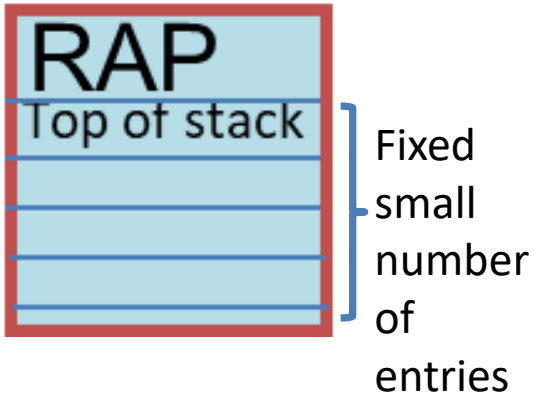
Return Address Predictor



- Keep a small hardware stack in the branch predictor
- Which attempts to mirror the program's call-return stack
- Updated when call and return instructions are executed
- Value at top of stack is used as predicted next PC when the BTB predicts that the current instruction is a RET

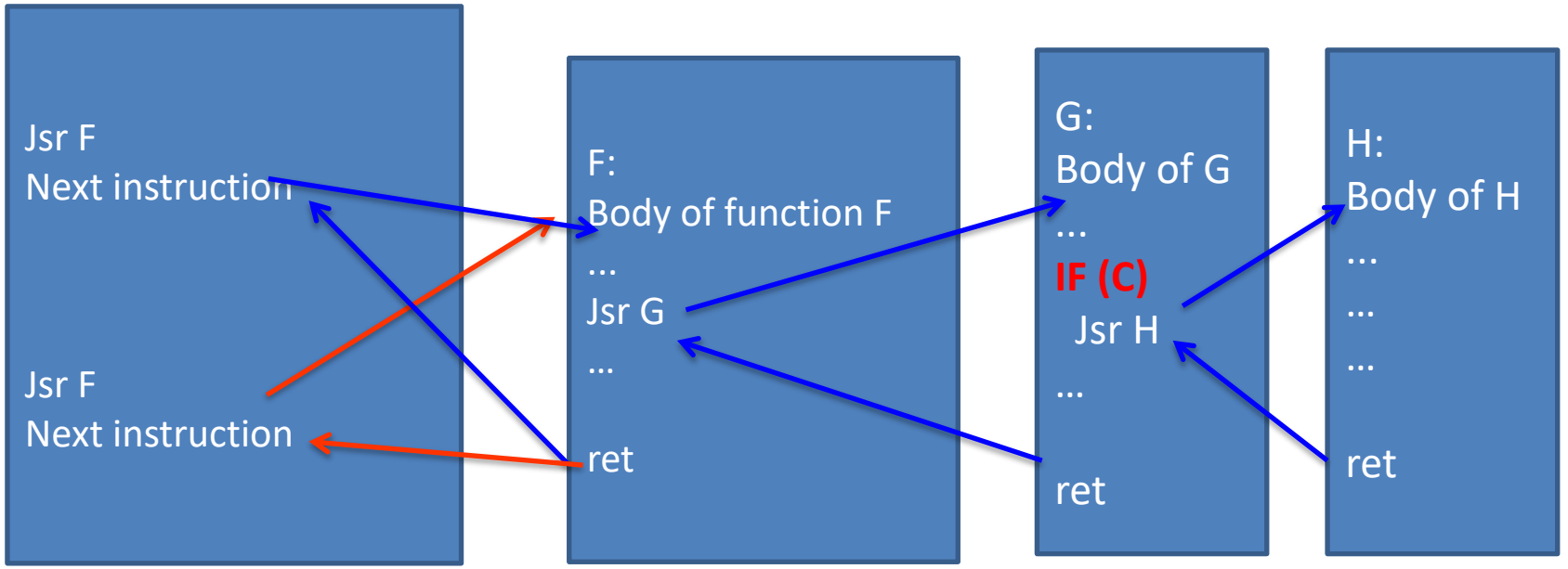
Return Address Predictor - mispredictions





- What happens if the call stack is deeper than the RAP's stack?
  - On return, the RAP's stack will be empty!
- Why might the prediction from the RAP be wrong?
  - Maybe the return address was overwritten
  - Maybe the stack pointer was changed
  - Maybe because we switched to another thread

Return addresses



- Q: when should the RAS be updated?
- The BTB is updated when a branch is *committed*
- But if we wait for commit to update the RAS, we might not have a prediction for the return from H
- Or: if we mispredict that the conditional "**IF(C)**" is true
  - We might have the wrong RAS prediction for the return from G

Branch prediction and multi-issue

- In a processor that fetches, issues and dispatches multiple instructions per cycle…..
- What if we encounter two (or more) branches in one issue “packet”?

Branch prediction and multi-issue

- In a processor that fetches, issues and dispatches multiple instructions per cycle…..
- What if we encounter two (or more) branches in one issue “packet”?

- **But all the BTB needs is to predict the next instruction to fetch – it doesn’t matter which branch is responsible**

- **Commonly, a bigger slower branch predictor may later *re-steer* the processor if it has a better prediction that should over-ride the BTB**

Dynamic Branch Prediction Summary

- Prediction seems essential (?)
- Two questions: branch ***takenness***, branch ***target***

Takenness:

- Branch History Table: 2 bits for loop accuracy
  - Saturating counter (bimodal) scheme handles highly-biased branches well
  - Some applications have highly dynamic branches
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches
  - Or different executions of same branches
- Tournament Predictor: try two or more competitive solutions and pick between them
- Predicated Execution can reduce number of branches, number of mispredicted branches

***Target*:**

- **Branch Target Buffer: include branch address & prediction**
- **BTB update**
- **Return address stack for prediction of indirect jump**

This lecture

Beyond:

- Prediction mechanisms have many applications beyond branch prediction:
  - Way prediction, prefetching, store-to-load forwarding, value prediction, etc
    - George Z. Chrysos and Joel S. Emer. 1998. Memory dependence prediction using store sets. ISCA98
  - Predictors can increase performance, but make it *harder* to optimize programs

Branch prediction resources

- Design tradeoffs for the Alpha EV8 Conditional Branch Predictor (André Seznec, Stephen Felix, Venkata Krishnan, Yiannakis Sazeides)
  - SMT: 4 threads, wide-issue superscalar processor, 8-way issue, 512 registers (cancelled June 2001 when Alpha dropped)
  - Paper: http://citeseer.ist.psu.edu/seznec02design.html
  - Talk: http://ce.et.tudelft.nl/cecoll/slides/PresDelft0803.ppt
- Branch prediction in the Pentium family (Agner Fog)
  - Reverse engineering Pentium branch predictors using direct access to BTB
  - http://www.x86.org/articles/branch/branchprediction.htm
- Championship Branch Prediction Competition (CBP), organised by the Journal of Instruction-level Parallelism
  - http://www.jilp.org/cbp/
- **The CBP-1 winning entry: TAgged GEometric history length predictor (TAGE):** for each branch, maintain a predictor for what history length (from a geometric progression) works best.
  - http://www.irisa.fr/caps/people/seznec/JILP-COTTAGE.pdf

Example: Branch prediction in Intel Atom, Silvermont and Knights Landing

- two-level adaptive predictor with a global history table,
- Branch history register has 12 bits
- The pattern history table on the Atom has 4096 entries and is shared between threads
- The branch target buffer has 128 entries, organized as 4 ways by 32 sets
  - (size on Silvermont unknown, but probably bigger, and not shared between threads)
- Unconditional jumps make no entry in the global history table, but always-taken and nevertaken branches do
- Silvermont has branch prediction both at the fetch stage and at the later decode stage in the pipeline, where the latter can correct errors in the former
- No special predictor for loops (as there is for some other Intel CPUs)
  - Loops are predicted in the same way as other branches
- Penalty for mispredicting a branch is 11-13 clock cycles.
- It often occurs that a branch has a correct entry in the pattern history table, but no entry in the branch target buffer, which is much smaller:
  - If a branch is correctly predicted as taken, but no target can be predicted because of a missing BTB entry, then the penalty will be approximately 7 clock cycles.
- Pattern prediction evident for indirect branches on Knights Landing but not on Silvermont.
  - Indirect branches are predicted to go to the same target as last time on Silvermont
- Return stack buffer with 8 entries on the Atom and 16 entries on Silvermont and Knights Landing

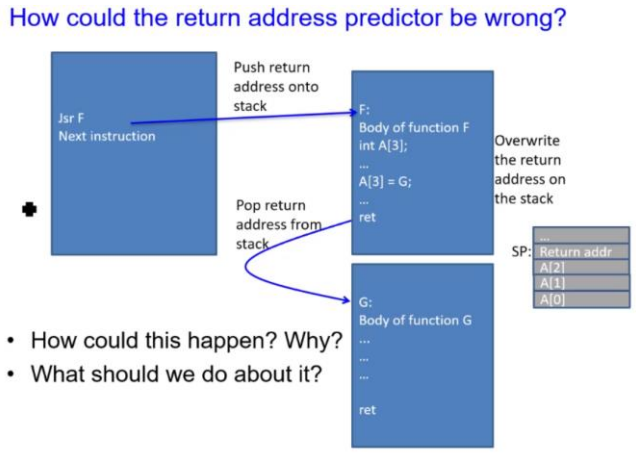
"The microarchitecture of Intel, AMD and VIA CPUs An optimization guide for assembly programmers and compiler makers" http://www.agner.org/optimize/microarchitecture.pdf

Student question: better predictions for indirect branches

- As you say, a BTB should give you a prediction for an indirect branch.
- However it might not be a very good one - the killer app is polymorphic calls in object-oriented languages (virtual calls where the target object has a different type on different invocations).
- For that we need to add global history to the branch target prediction. We did not cover this in the lectures.
- This paper evaluates three alternative schemes:
- Dharmawan, Tubagus & Jeyachandra, E & Rahmadhani, Andri. (2016). Techniques to Improve Indirect Branch Prediction. 10.13140/RG.2.2.24350.02884.
- The state of the art is perhaps represented by this article in the same ISCA2020 "Industry" track:
- The IBM z15 High Frequency Mainframe Branch Predictor (computer.org) (section VI], pg 35-6). Basically they use the branch history to index a special BTB (actually they expand the branch history concept to include a couple of bits of the PC address of each taken branch in the history).

Student question: return address predictor stack consistency

- “Hi, I don't quite understand the difference between the RAP stack and the main memory stack in the following example.
- When the main memory stack is overwritten by A[3] = G, is the RAP stack overwritten as well? If not, then the RAP prediction will be correct right?”



- The RAP stack is a small hardware unit which tries to mirror what the return address stack should look like in memory.
- However there are various reasons why it might not actually reflect the real stack.
- We discussed for example:
  - F might change the SP register
  - F might overwrite the return address, for example through a buffer overrun as shown in the slide above.
  - We might have an inconsistent RAP due to the misprediction of some other branch - as we discussed at length in the class. This could happen if the RAP is updated speculatively - while the real stack is updated only when memory writes are committed (eg the memory write resulting from a jsr).

Followup on class discussion: buffer overrun vulnerabilities

- In yesterday's class we touched upon buffer over-run vulnerabilities.
- As I mentioned, this is a big deal and it's the root cause for many many cyberattacks.
- As was discussed in the class, there are some mitigations. One that we talked about was the use of a "canary" word, adjacent to each return address on the stack. This idea (and the general problem) is introduced in this nice paper:

  StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks
  https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf
- There are more sophisticated techniques, for example Shadow Stacks:

  Stack Shield: https://www.angelfire.com/sk/stackshield/info.html
- More secure mechanisms are the focus for a lot of current research and development; see Control-flow integrity - Wikipedia .
- Arguably the heart of the problem is the design of the C programming language, which lacks bounds checking on the use of arrays and pointers - this is what let's a buffer overrun happen in the first place. Indeed C allows a pointer to get separated from the array into which it is supposed to point - making checking hard.
- I actually published a bounds checking scheme back in 1997. To get an idea of how big the field has become, check out the citations to our paper:

  https://scholar.google.com/citations?view_op=view_citation&hl=en&user=4YyGhBUAAAAJ&citation_for_view=4YyGhBUAAAAJ:u-x6o8ySG0sC
- The latest hot topic in this space is "capabilities":

  CHERI: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/
- Buffer overruns should be old news. Sadly not. Check out this article:

  The Battle for the World's Most Powerful Cyberweapon - The New York Times:
  https://www.nytimes.com/2022/01/28/magazine/nso-group-israel-spyware.html
- This entertaining article describes how the attack actually worked:

  Analyzing Pegasus Spyware's Zero-Click iPhone Exploit ForcedEntry:
  https://www.trendmicro.com/en_us/research/21/i/analyzing-pegasus-spywares-zero-click-iphone-exploit-forcedentry.html
- And yes, at its heart, it's a buffer overrun.

Student question:

"**What is branch folding?“**

See ARM1136JF-S and ARM1136J-S Technical Reference Manual r1p3

How can we avoid even having to execute branches at all?

I'm not sure ARM do it but here's how I think about it:

- Idea: instead of just the branch target address, stash the branch target *instruction* in the BTB
- Skip IF stage for next instruction
- Effective CPI for branch is zero
- Could stash target instruction for both taken and not-taken cases to reduce misprediction delay

Beyond the lectures

- An interesting step beyond the ideas presented in the lecture is to incorporate branch prediction into instruction prefetching.

- The search term is "branch predictor directed prefetch".

- For example this is used in ARM's Neoverse N1:

  https://ieeexplore.ieee.org/document/8986666 (page 3)

- For an example of academic work in this space, see

  https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9408197 “Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective”, Yasuo Ishii et al.

- see also this IBM patent:

  https://patents.google.com/patent/US6560693B1/en US6560693B1 - Branch history guided instruction/data prefetching