COMP60001/COMP70086
Advanced Computer Architecture
Chapter 3

Part 2: Branch *Target* Prediction

October 2026

Paul H J Kelly

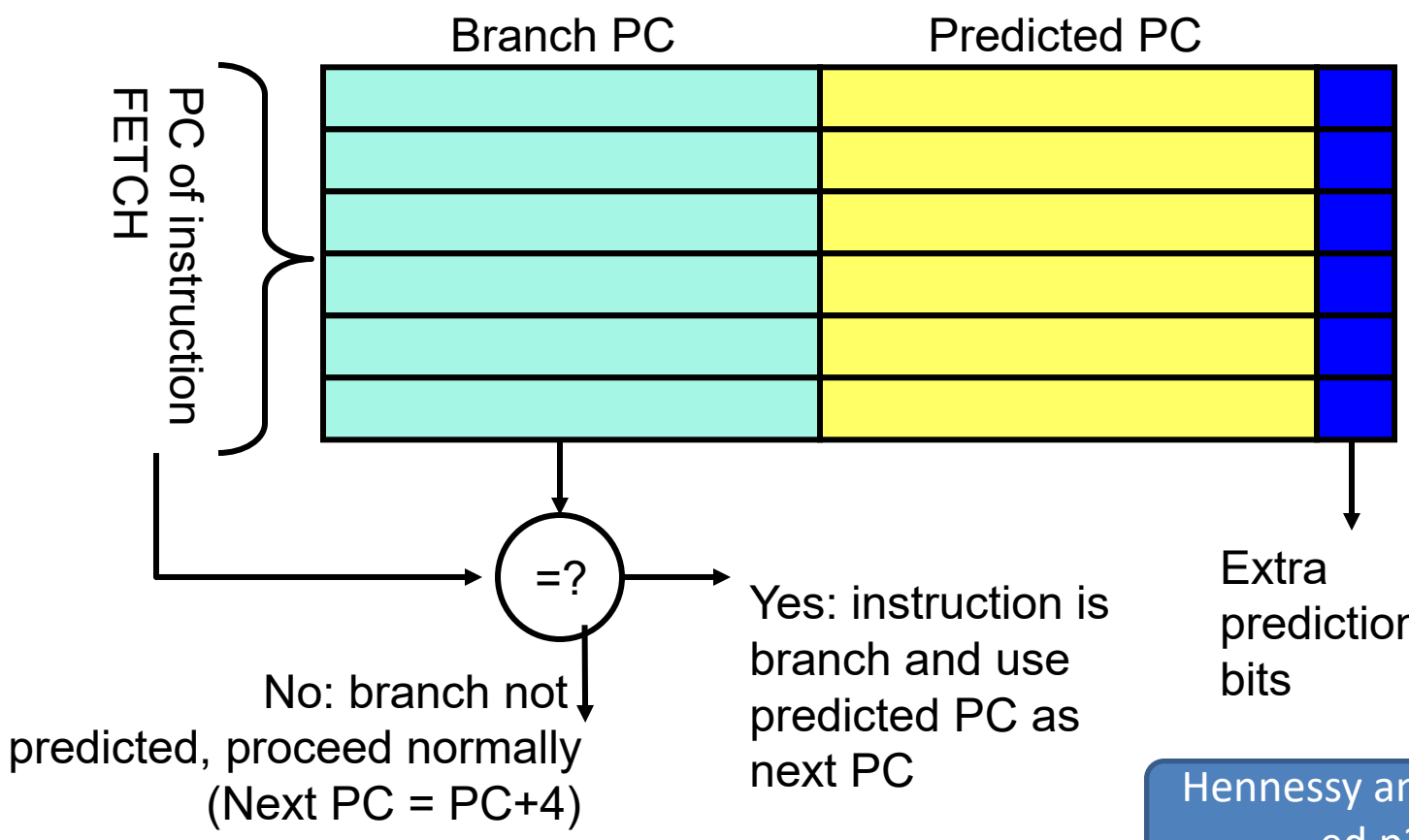
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (4-6th eds), and on the lecture slides of David Patterson's Berkeley course (CS252)*

Branch Prediction - context

- What we really want from 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 (BTB)

- 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

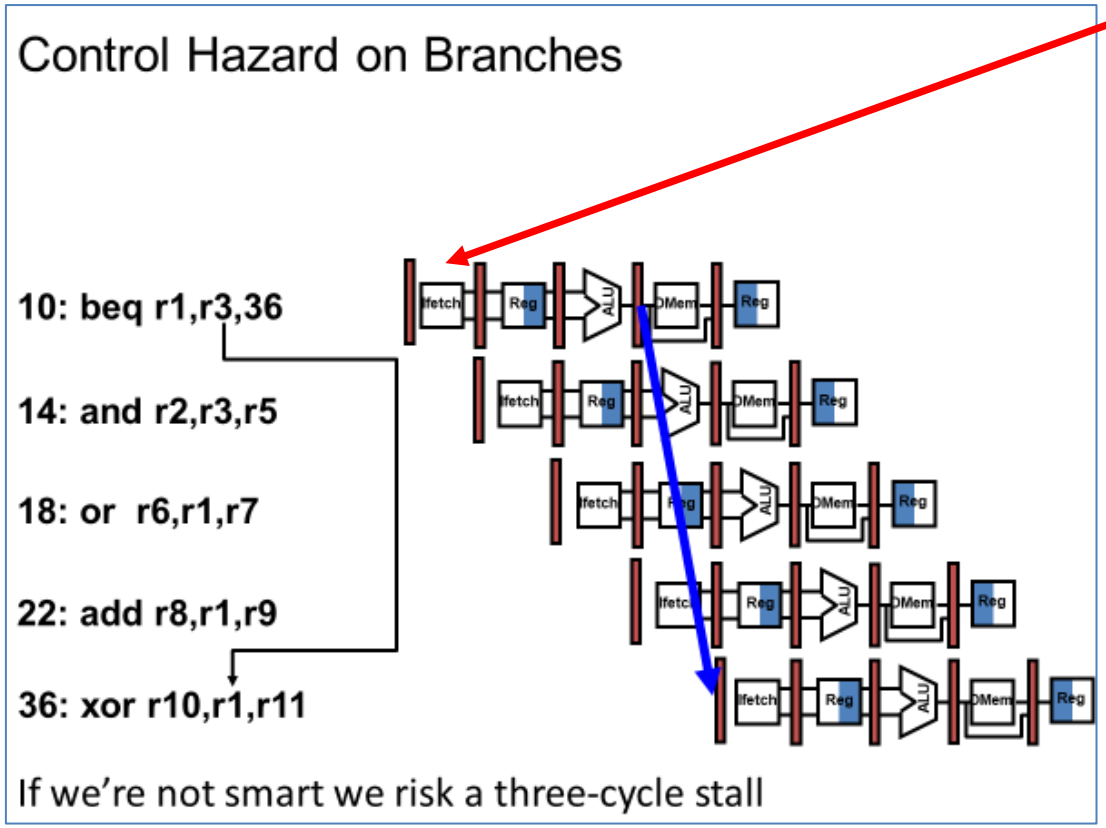


Hennessy and Patterson 6th ed p228-232

Branch **target** prediction: BTBs

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

- This doesn't have to be true!



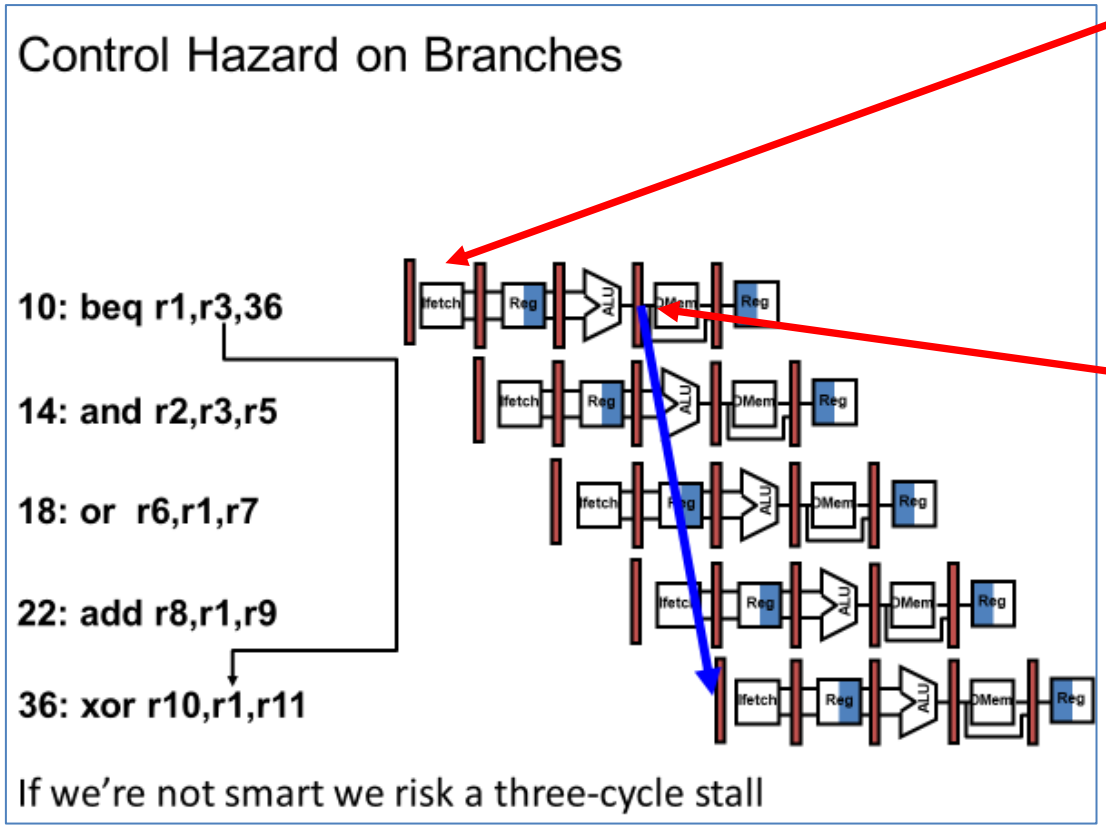
In parallel with every ifetch

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 that current instruction is a branch instruction"

- This doesn't have to be true!



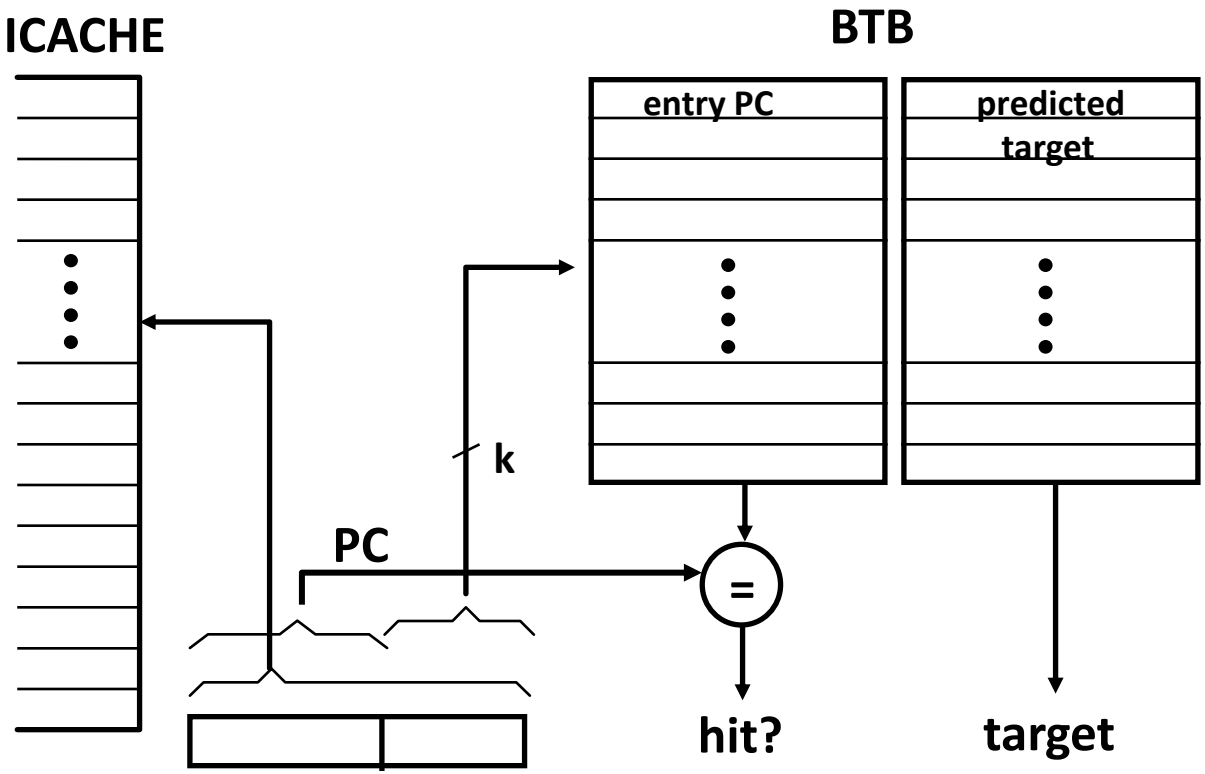
In parallel with every ifetch

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

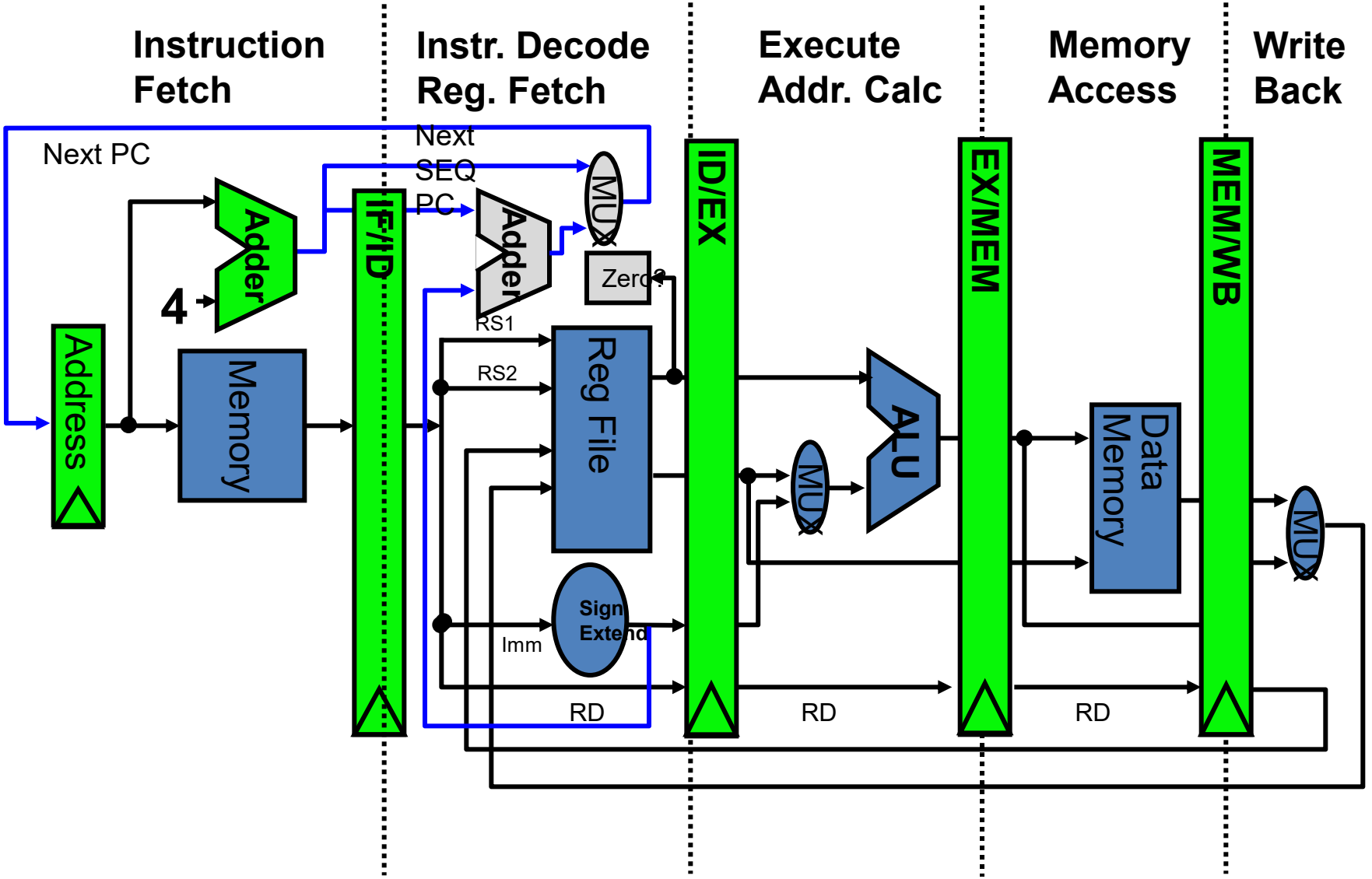
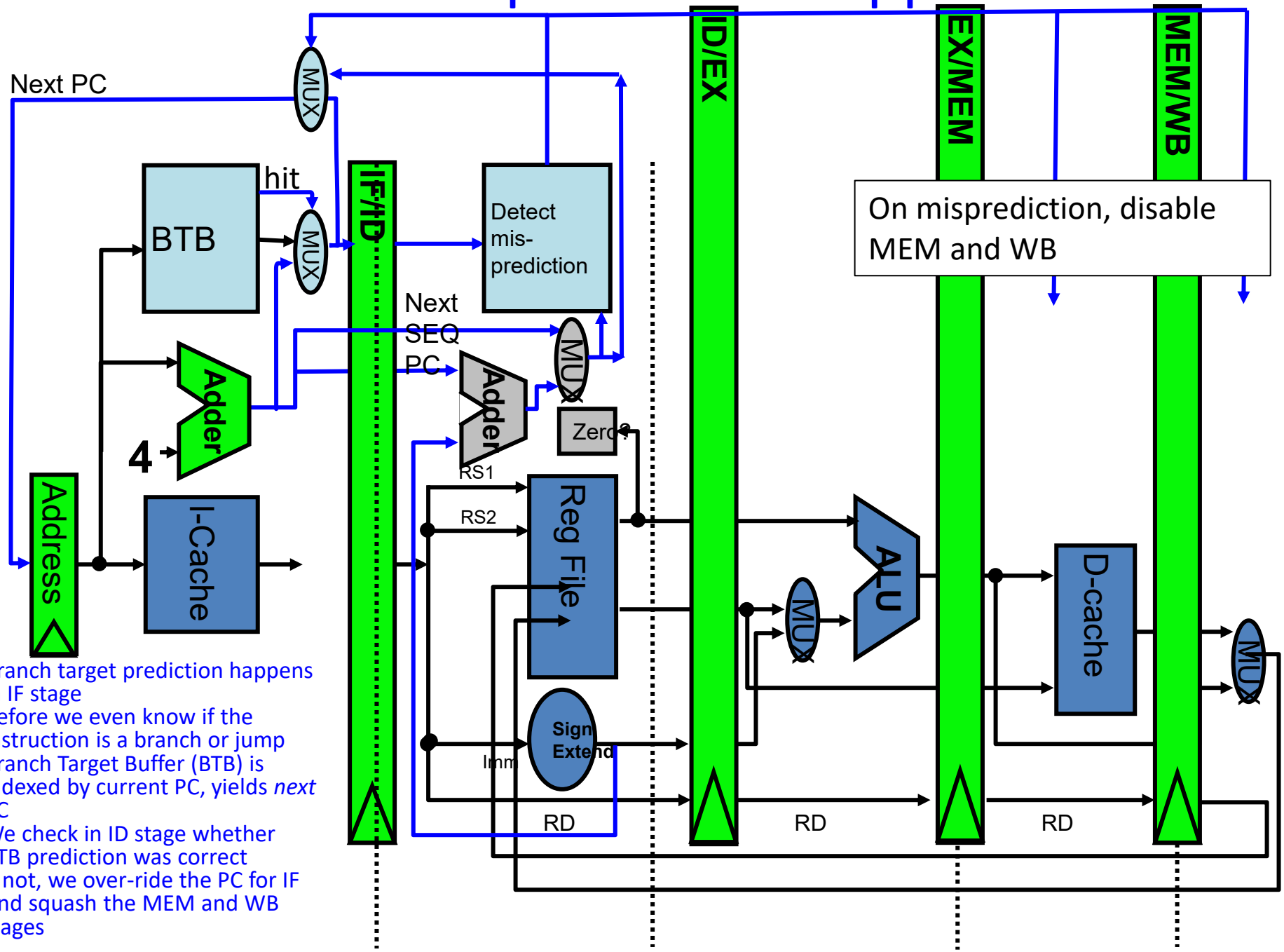


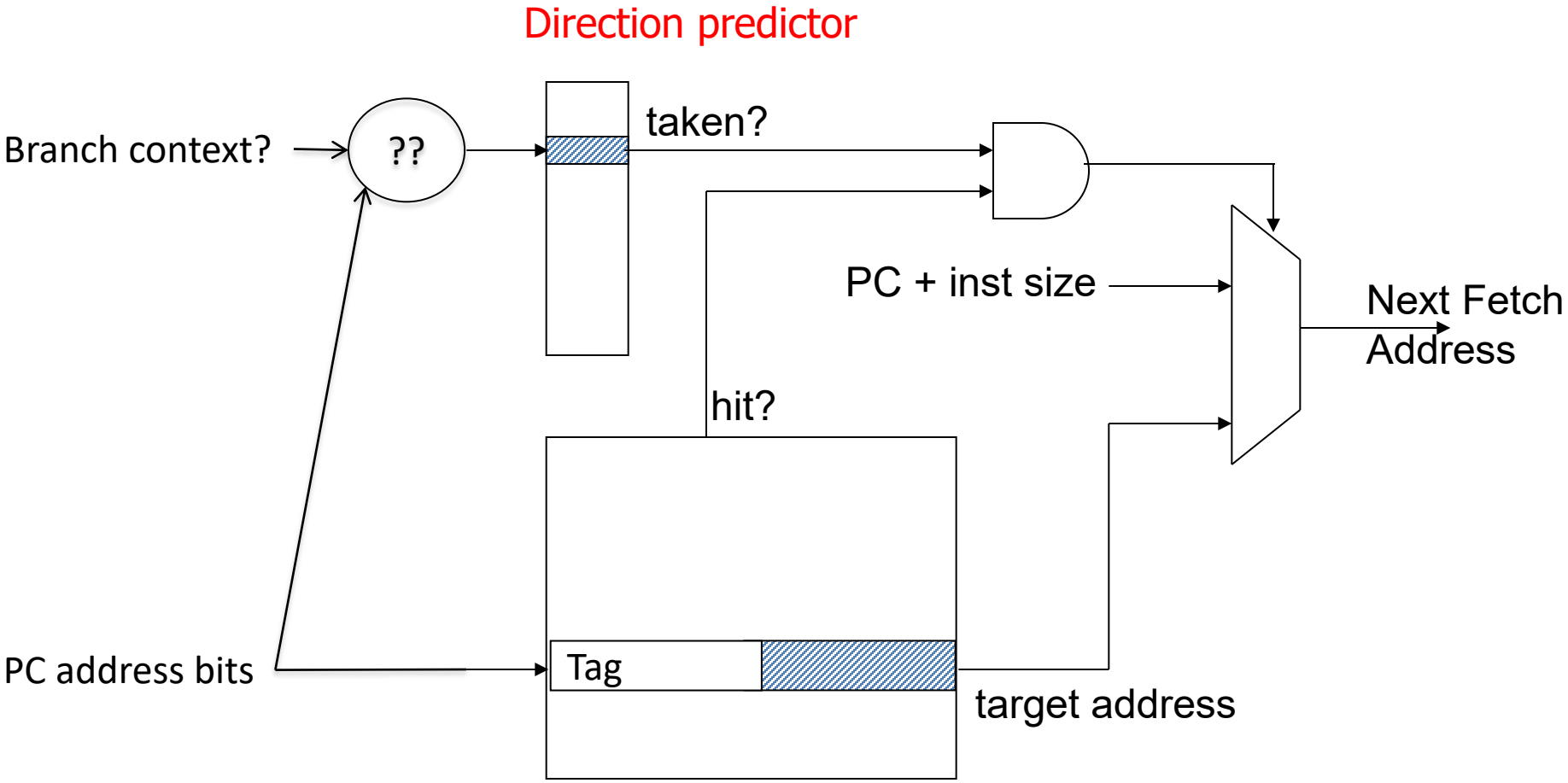
Figure 3.22, page 163, CA:AQA 2/e

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

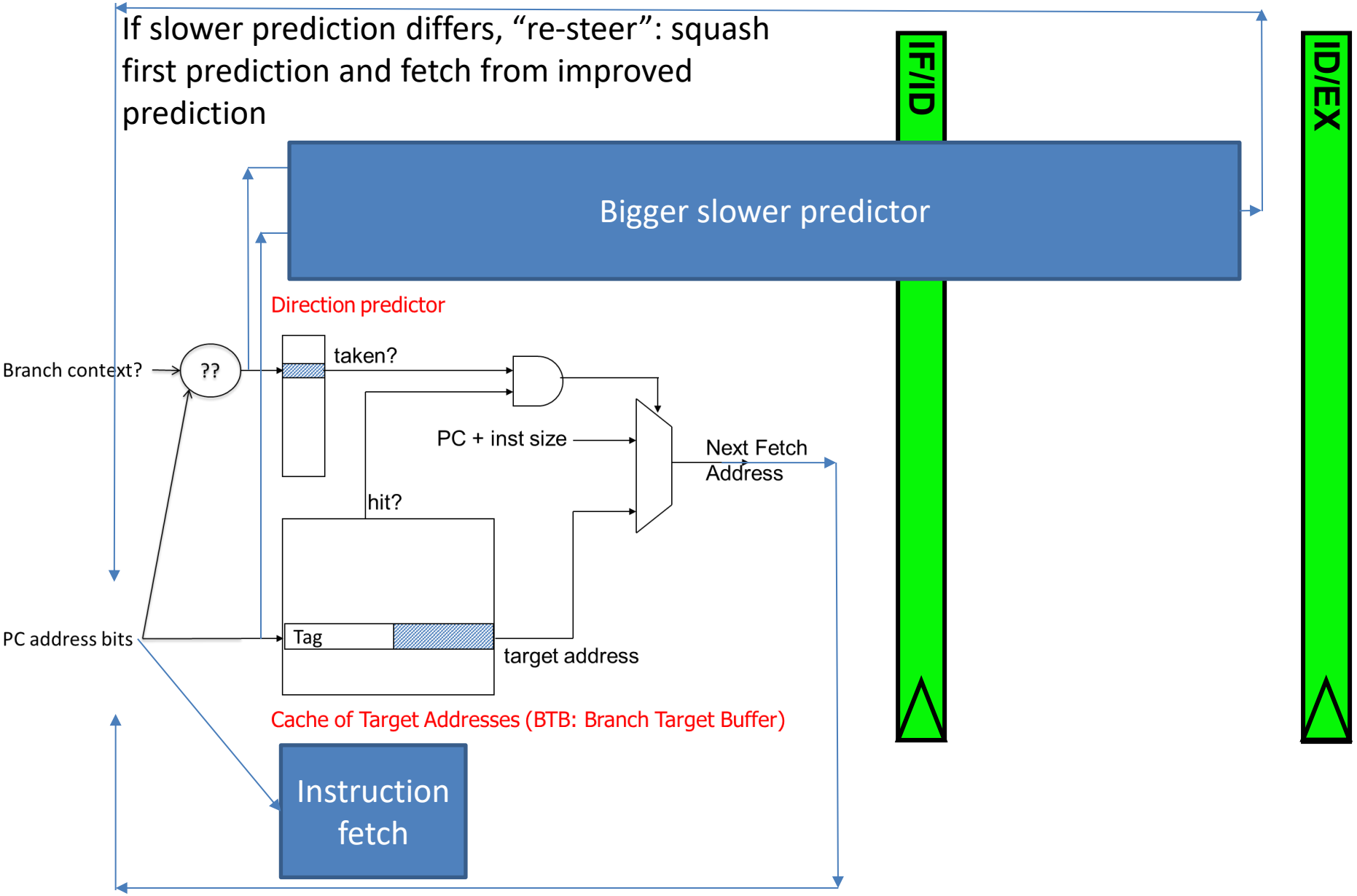
Combining BTB with direction prediction



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

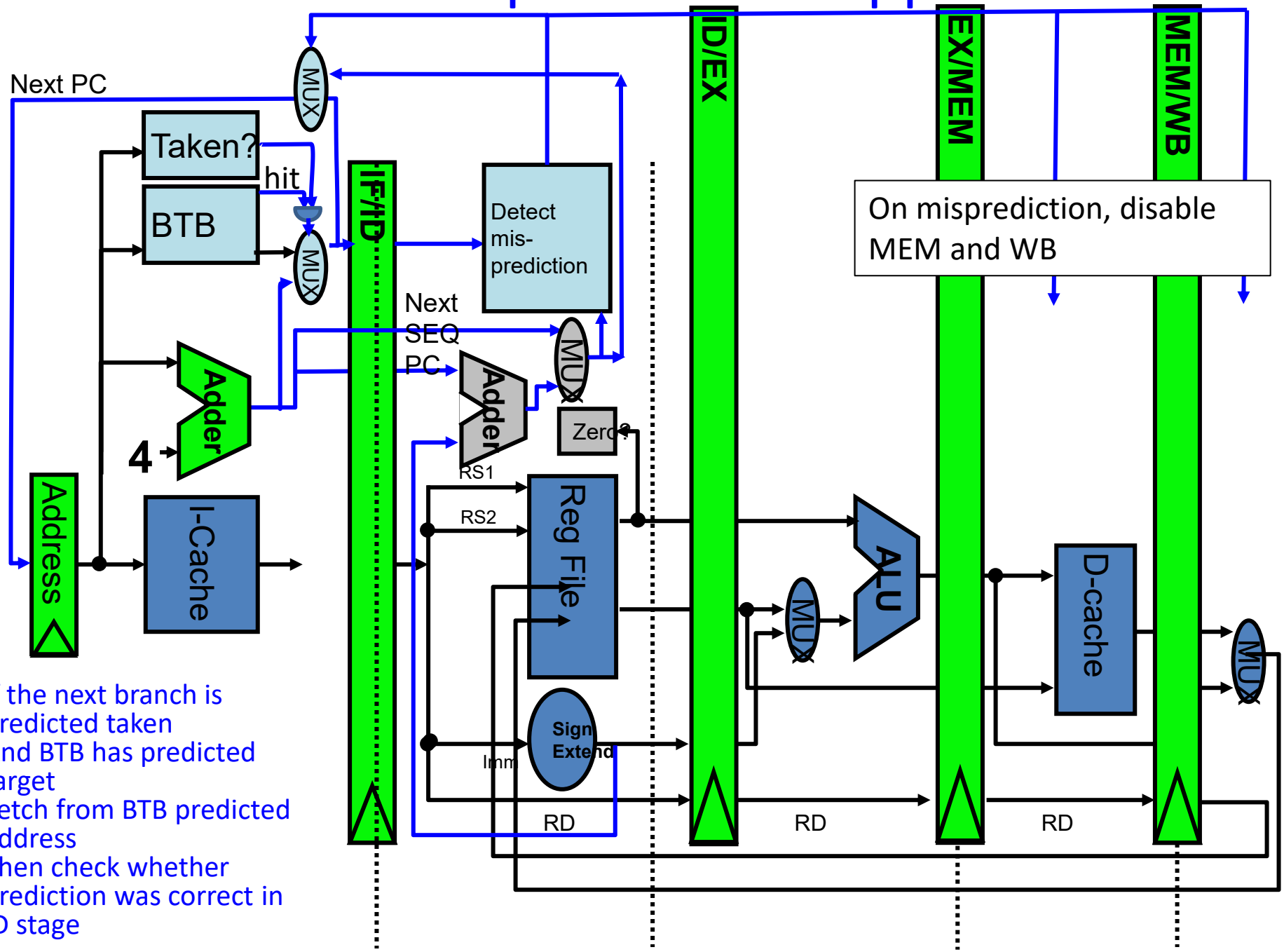
Credit: Onur Mutlu

Combining fast simple predictor with slower bigger predictor



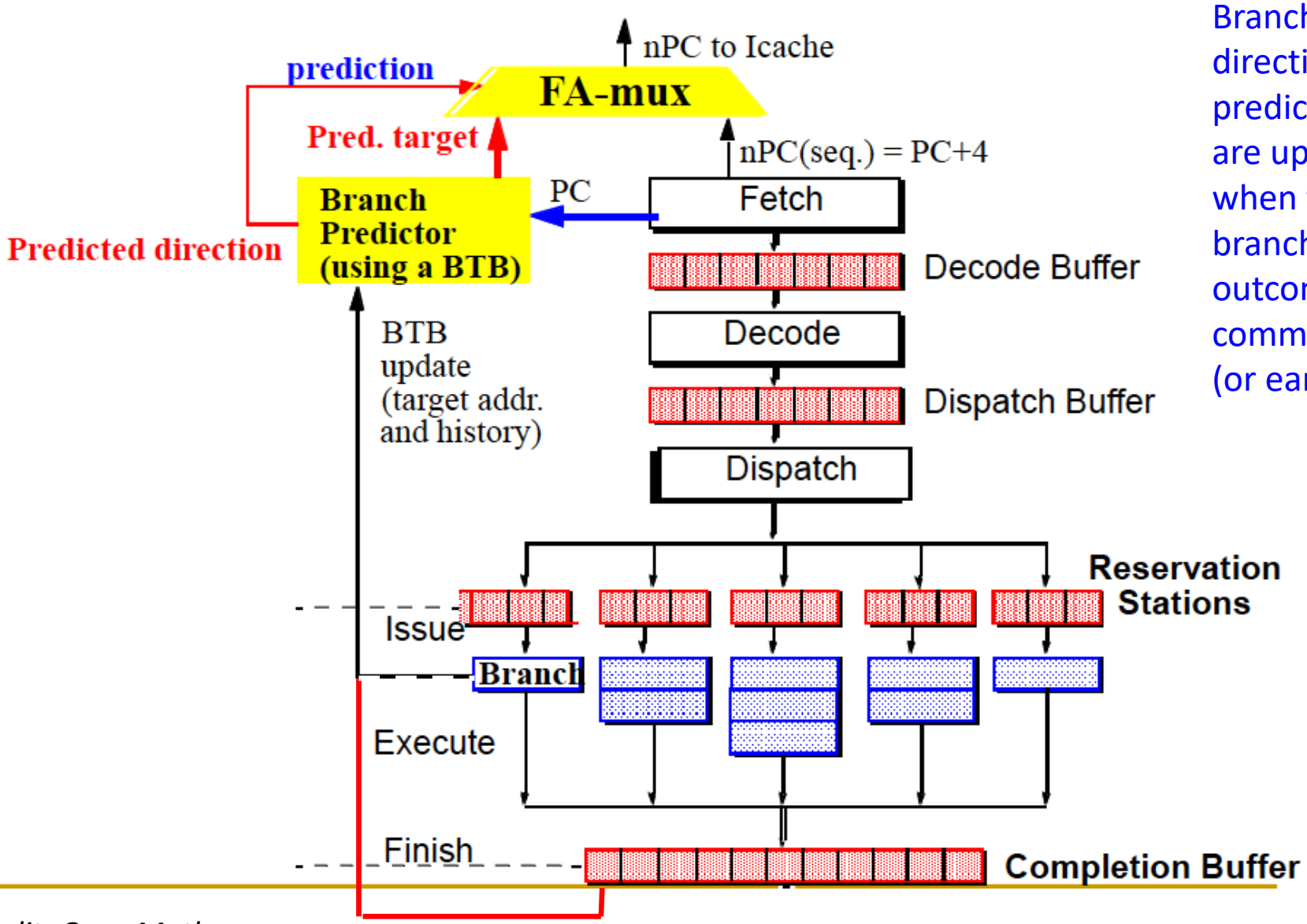
(What if branch is predicted-taken but BTB miss?)

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

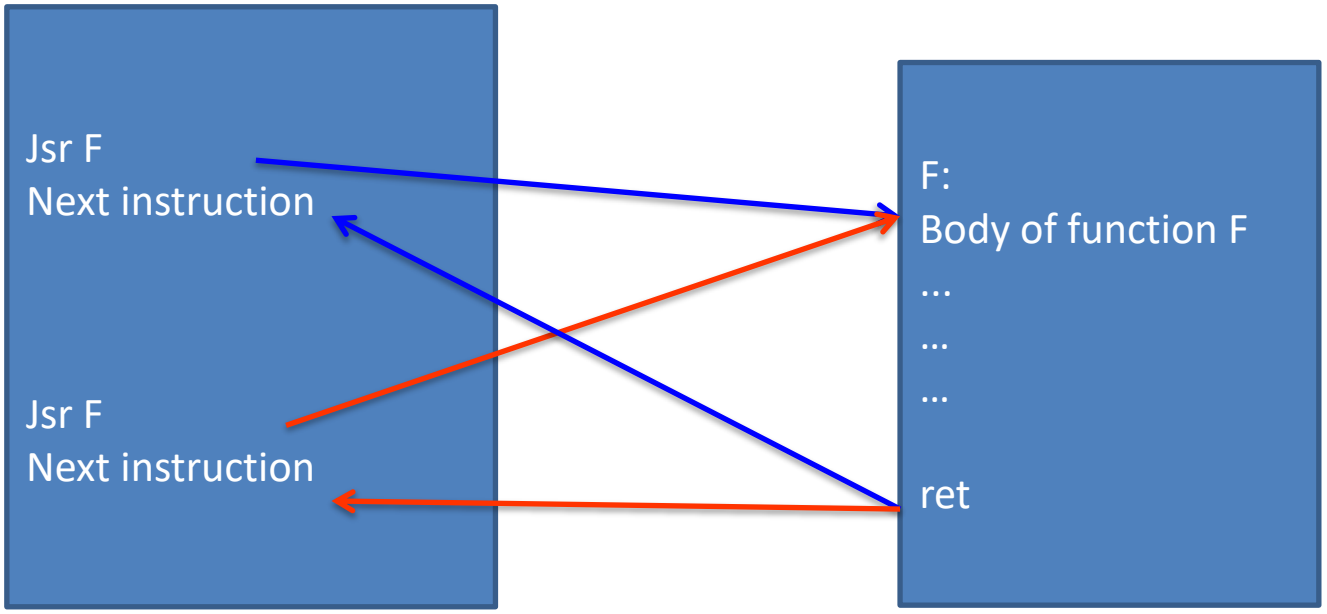
Updating the branch prediction



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

Credit: Onur Mutlu

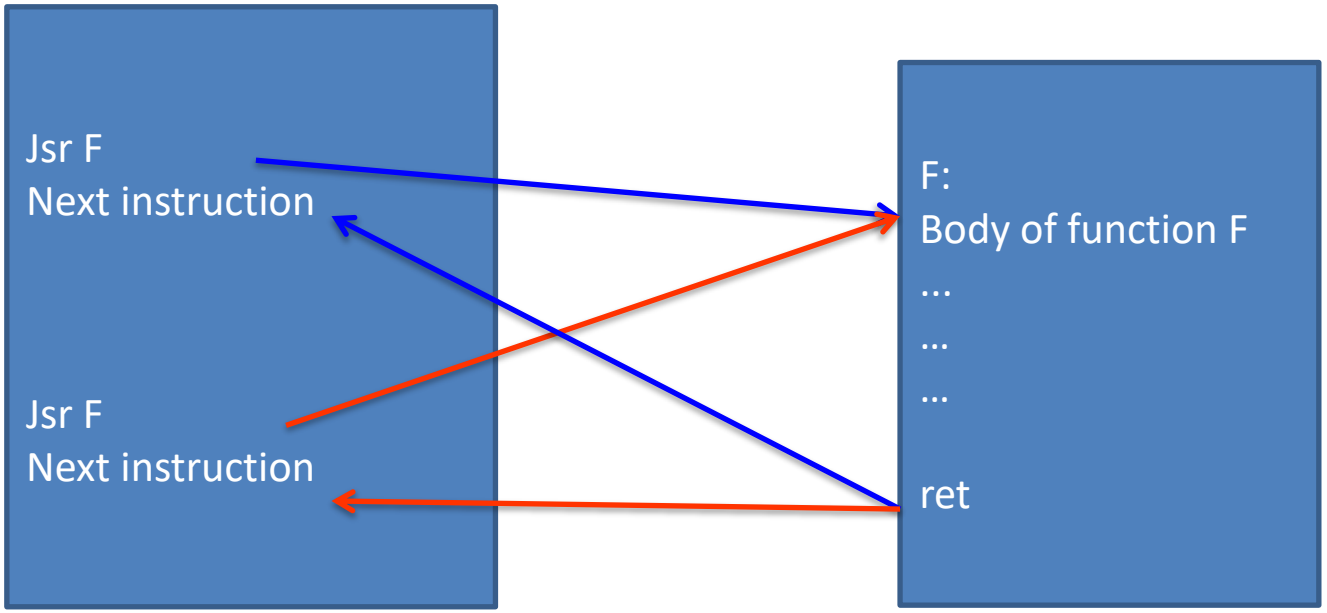
Return addresses



- 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

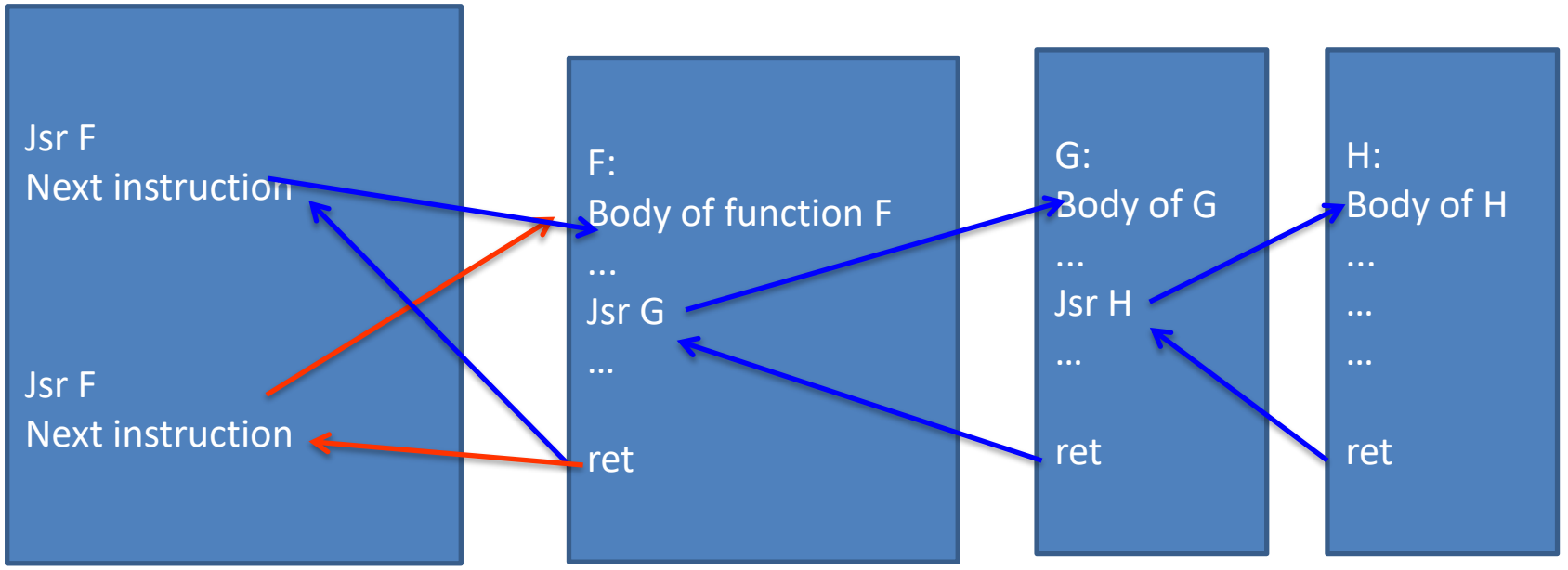
Return addresses



- 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

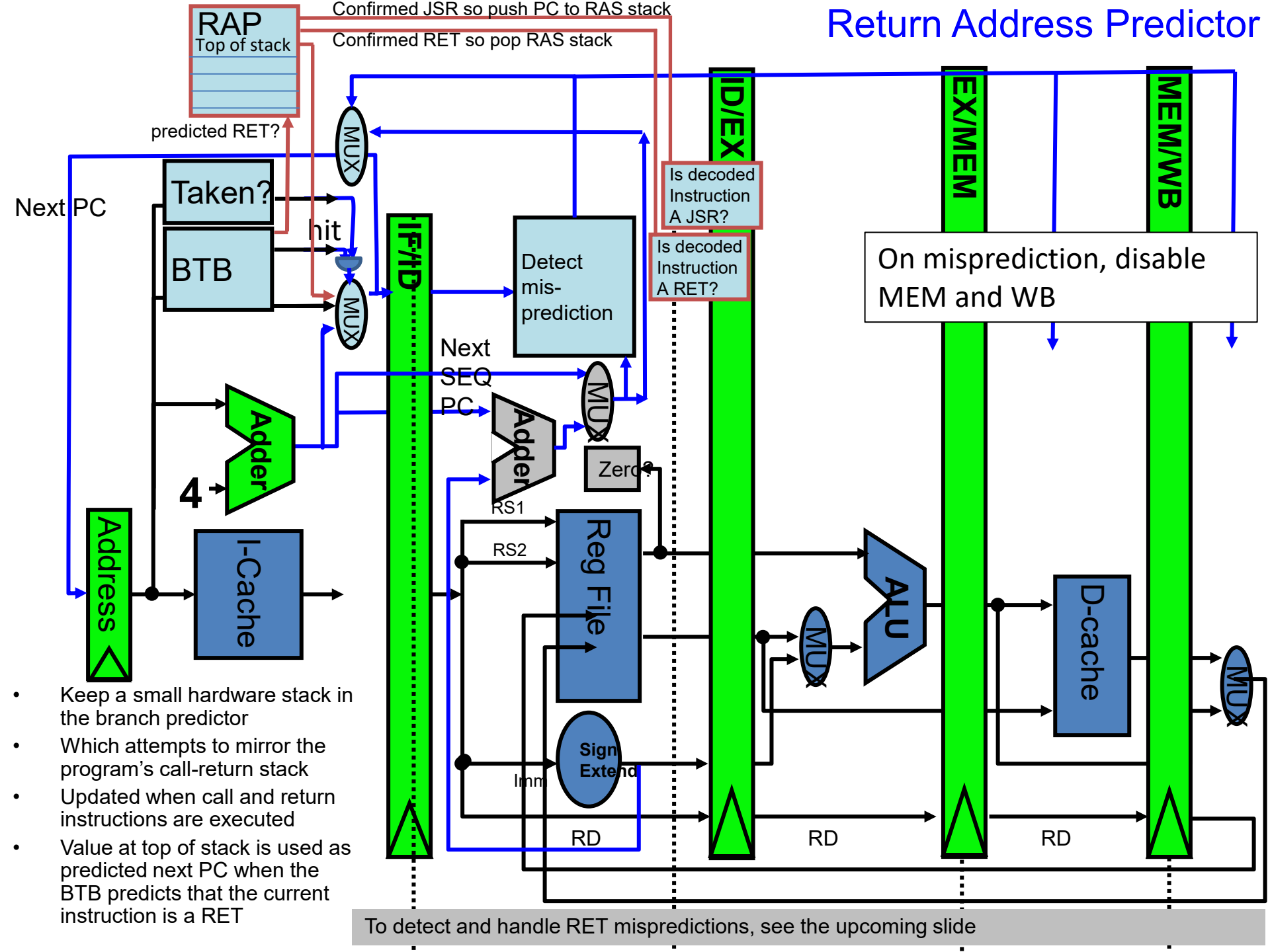
Return addresses



- 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

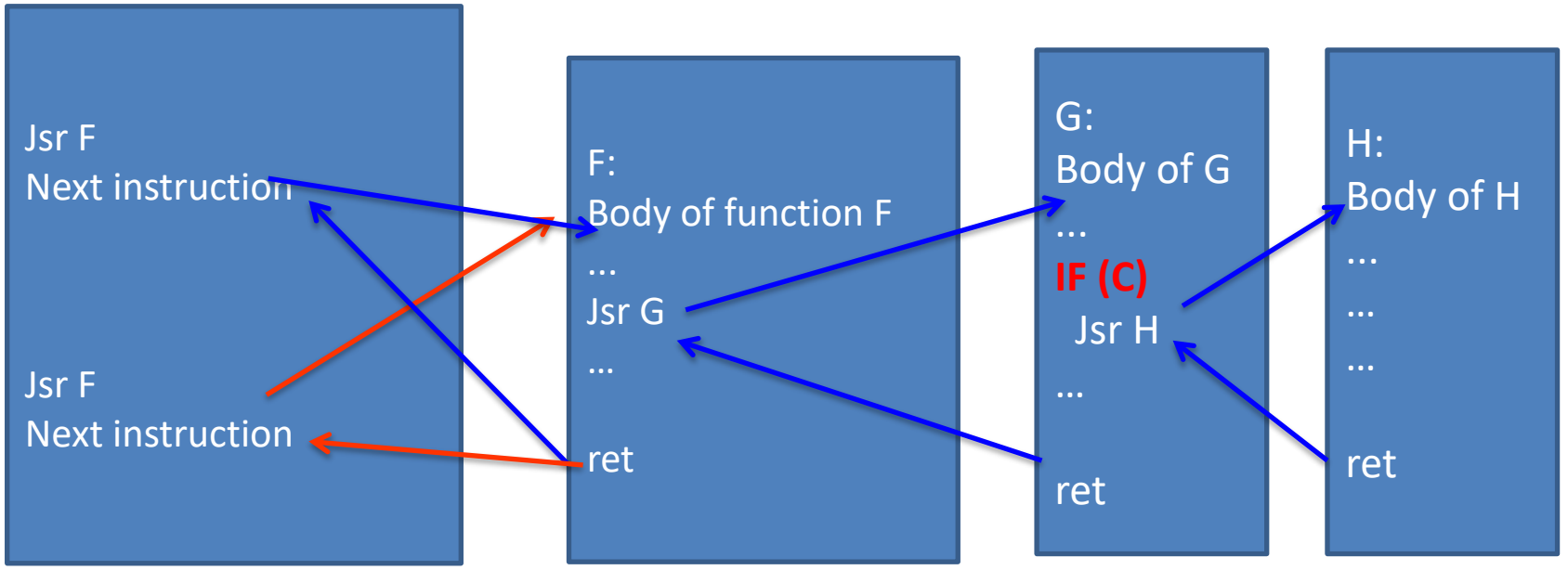
Return Address Predictor

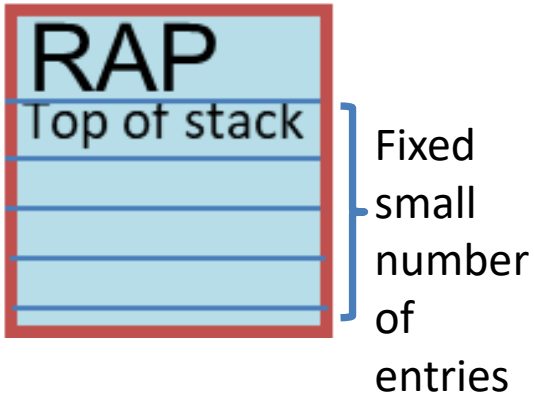


- Keep a small hardware stack in the branch predictor
- Which attempts to mirror the program's call-return stack
- Updated when call and return instructions are executed
- Value at top of stack is used as predicted next PC when the BTB predicts that the current instruction is a RET

To detect and handle RET mispredictions, see the upcoming slide

Return Address Predictor - mispredictions



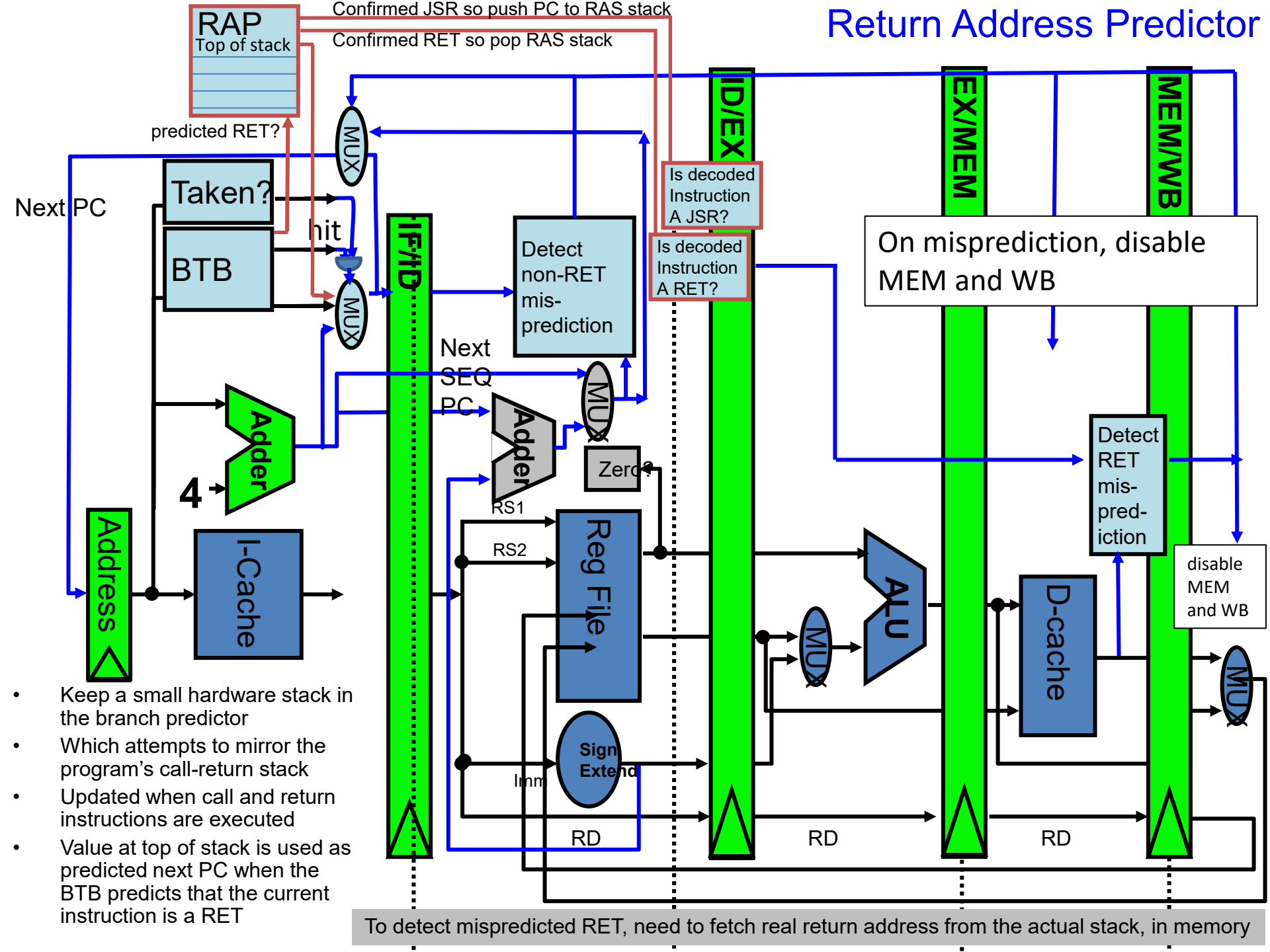


- What happens if the call stack is deeper than the RAP's stack?
  - On return, the RAP's stack will be empty!
- Why might the prediction from the RAP be wrong?
  - Maybe the return address was overwritten
  - Maybe the stack pointer was changed
  - Maybe because we switched to another thread

Detecting RET mispredictions

- The Return Address Predictor might mispredict
- So after making the prediction, we have to actually read the real return address, from memory, and check that it matches the address the RAP predicted
- So we don't actually know for sure that the prediction was right until:
  - After the MEM stage in an ISA where RET uses the stack (as in x86)
  - Or earlier in ISAs where the return instruction uses a register to hold the return address (MIPS, ARM, RISCV)

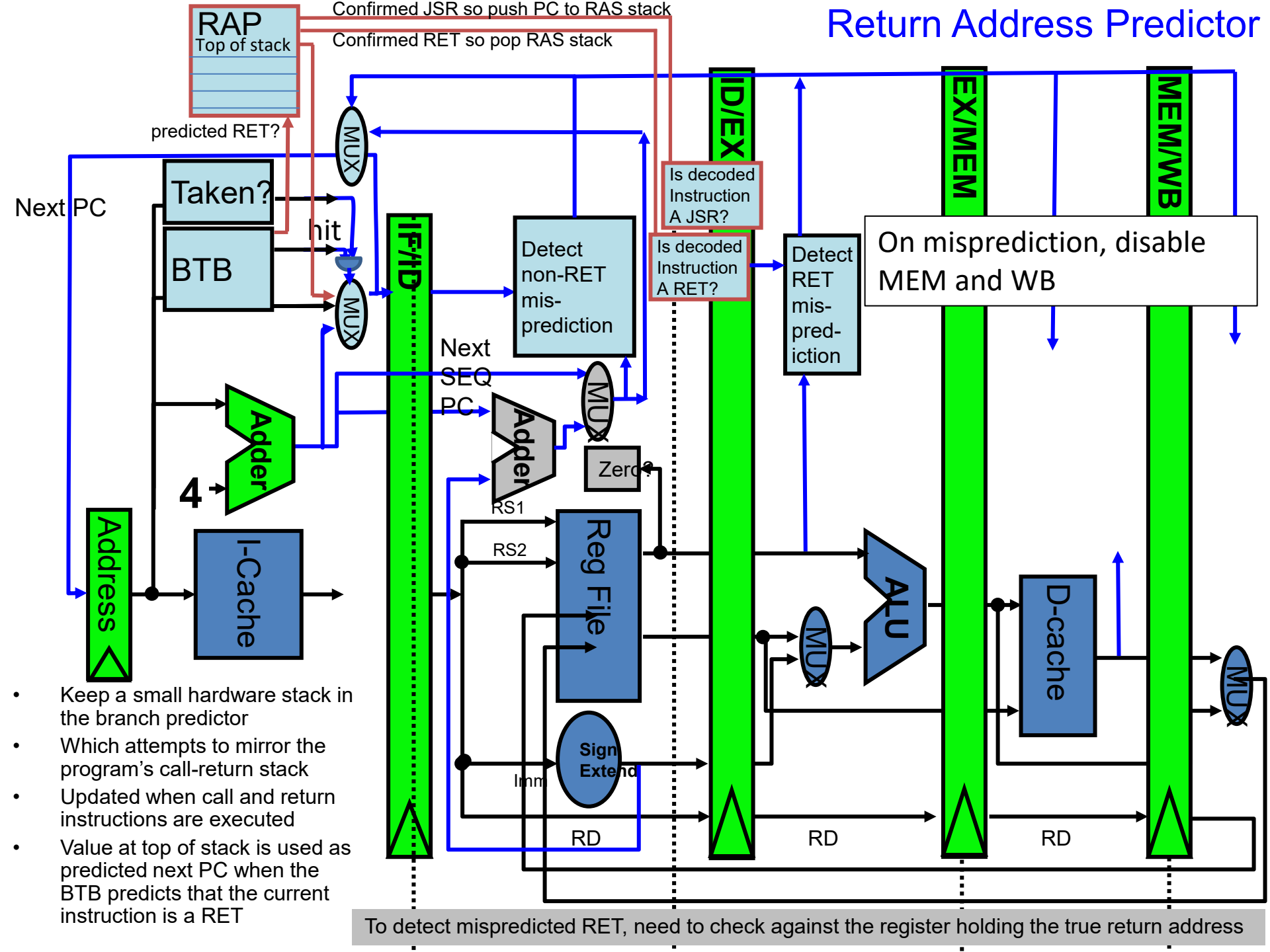
Return Address Predictor



- 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

To detect mispredicted RET, need to fetch real return address from the actual stack, in memory

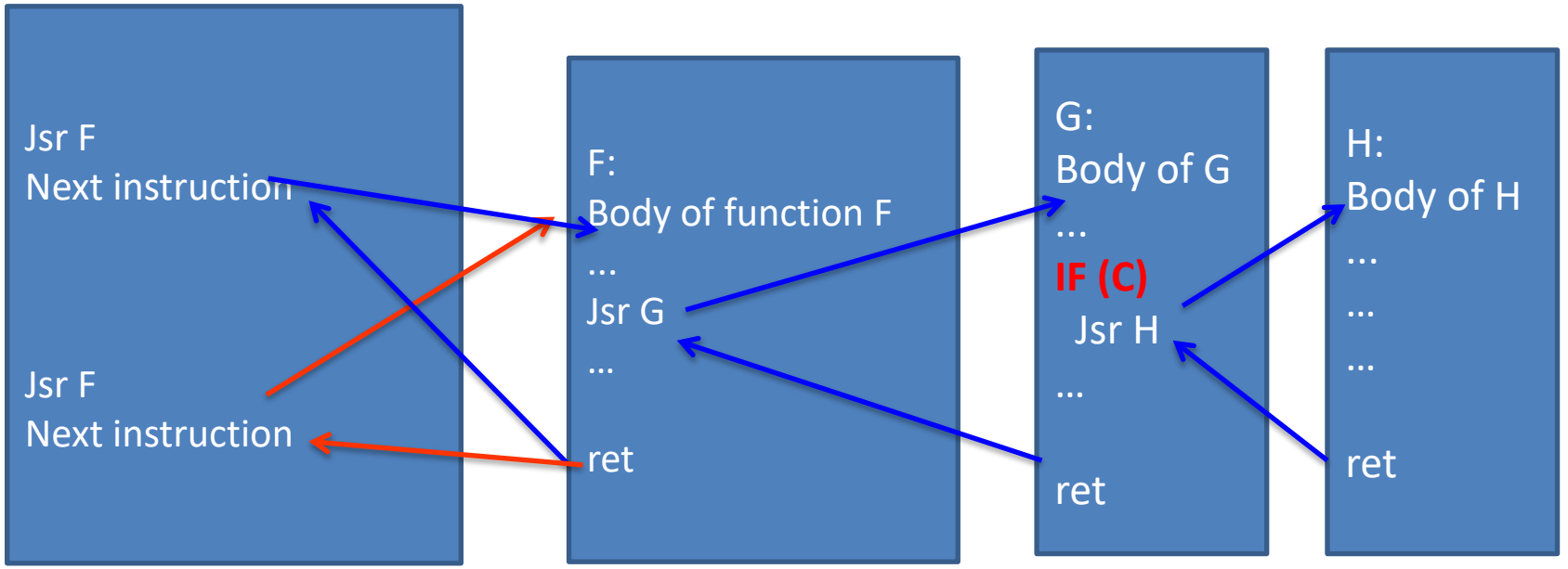
Return Address Predictor



- 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

To detect mispredicted RET, need to check against the register holding the true return address

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
    - Sebastian Kim & Alberto Ros, Effective Context-Sensitive Memory Dependence Prediction (HPCA24)
    - 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

Example: Branch prediction in AMD Jaguar

- See https://www.realworldtech.com/jaguar/2/
- Particularly interesting for coverage of branch prediction for a processor that fetches and executes multiple instructions per cycle

- "When a branch is detected, the IP address of the fetch window indexes into the Branch Target Buffer (BTB), which is coupled to the L1I. The BTB is a two level structure; the L1 is optimized for sparse branches and the L2 handles dense branches. The L1 BTB is conceptually part of the instruction cache; it tracks two branches for every 64B line (1024 entries total) and can simultaneously predict both branches with only a single cycle penalty for taken branches. The L2 BTB is allocated dynamically and tracks an extra 2 branches per 8B region and also contains 1024 entries. The L2 BTB is slower and makes a single prediction per cycle, with a two cycle penalty for the first dense branch prediction and only a single cycle for any subsequent prediction. The BTB design saves power by only engaging the L2 when code actually has 3 or more branches per cache line, exploiting branch density to reduce power.
- Conditional near branches are implicitly predicted as not-taken, which saves space in the BTB. Once such a branch is taken, it is set to always taken in the BTB. Should the always taken branch subsequently fall through, it switches to a dynamic neural network predictor using 26-bits of global history.
- Another BTB optimization is that the L1 and L2 BTBs only predict target addresses for direct branches that are in the same 4KB page as the IP of the fetch window. A 32-entry out-of-page target array handles branch targets with up to 256MB of displacement for the L1 BTB. Sparse branch targets with >256MB of displacement, and dense branches with out-of-page targets are resolved by the branch target address calculator with a four cycle penalty.
- Near calls and the associated returns are predicted by a 16 entry Return Address Stack (RAS). The RAS can recover from most forms of misspeculation without corrupting the predictions. For cases that cannot be recovered, the RAS is invalidated to avoid mispredictions.
- Indirect branches with multiple targets are predicted using the IP address and 26-bits of global history to index into the 512-entry indirect branch target array. There is an extra 3 cycle penalty for any indirect branch predictions, but indirect branches with a single target and 256MB or less displacement are tracked through the lower latency out-of-target array.
- If a cache line is only being used for instructions, then the branch information in the L1 BTB is compressed and stored in the ECC bits of the L2 cache when the line is evicted and can be reloaded. The information is lost if the cache line is hit by a store, or is evicted to main memory. L1I misses trigger a 64B fetch request to the L2, and also prefetch one or two additional cache lines.
- Once the fetch address has been determined, the 32B of instructions from the L1I are sent to the Instruction Byte Buffer (IBB), which acts as a decoupling queue between the fetch and decoding stages. The IBB entries are 16B each, so a fetch will typically fill two at a time, and Jaguar has 16 entries, versus 12 for Bobcat. A small loop buffer tracks four recent 32B fetches and can bypass the instruction cache lookup mechanism to save power."

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

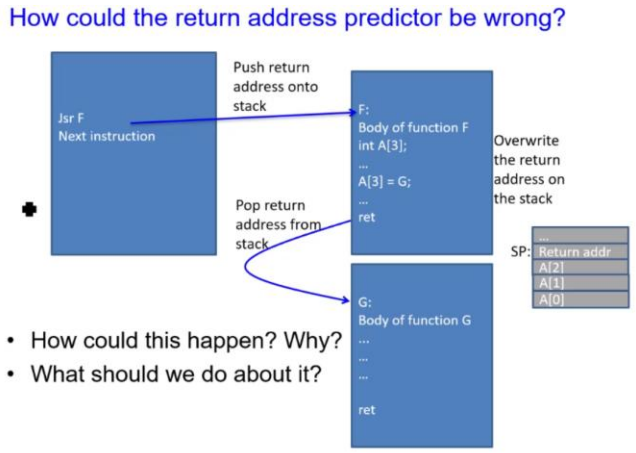
Student question: better predictions for indirect branches

- 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

Q:

- "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?"



A:

- 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 how 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

Student question: Branch History in multithreaded cores

Q: Let's say our CPU uses gselect(m, n) and so has a Branch History Register and Branch History Tables. But we want to use thread interleaving in the pipeline to reduce stalls (Instruction order of: T1, T2, T1, T2, ...). Will the BHT still work or are there modifications that need to be made in hardware to support this?

If the PC for T1 and the PC for T2 both index into the BHR and BHTs then I can see a lot of overwriting happening leading to many mispredictions.

(Question is based on content in Ch03 Part 1 Branch Direction Prediction.)

A: In a multi-threaded CPU core (using either fine-grained multithreading (FGMT) or simultaneous multithreading (SMT)) we need to be careful about what resources are separate, and what resources are shared.

The Branch History Register stores the sequence of taken/no-taken branch outcomes from recent branches. The BHR is used for branch prediction because this is sometimes predictive - a previous branch's direction is correlated with a later branch's direction. This is clearly a property that only holds within a single thread.

So we definitely need a separate BHR for each thread.

The branch history tables could still contain predictions from two different threads. [similarly, the L1 data and instruction caches contain cached data from both threads].

This means that one thread could overwrite branch prediction information "belonging" to the other thread. Actually this could even help - if the two threads are part of the same application, they might constructively learn from each other's branch outcomes. But it might just hurt.

So what we do see is that there is crosstalk: the behaviour of one thread might affect the behaviour of another thread.

In fact one thread might infer something about what the other thread is doing.

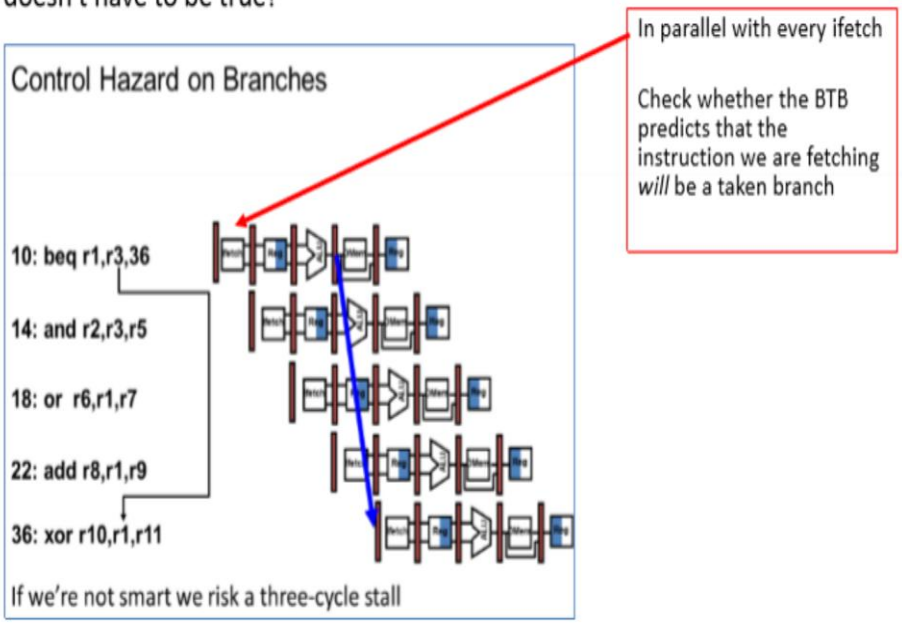
As we will see in a week or two, this creates a potential security problem. Although we will dive into this kind of thing later, you might like to have a look at this, for example: Taming STIBP [LWN.net] (https://lwn.net/Articles/773118/).

If you're truly determined to dive deeper still, you might enjoy seeing that the topic is causing continuing pain, with this bug report from 11 Oct 2023: Linux Patched For A New AMD Zen 4 CPU Bug - Erratum #1485 – Phoronix (https://www.phoronix.com/forums/forum/hardware/processors-memory/1414611-linux-patched-for-a-new-amd-zen-4-cpu-bug-erratum-1485)

Student question: BTB and direction prediction

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!



I'm just going back through lectures to summarise and noticed this - doesnt the direction predictor do this, and BTB is only for when we have a predicted taken branch?

- You're right - the direction predictor should also be used.
- The tricky thing is that you want the prediction before you even know whether the instruction even is a branch or jump.
- Note that the instruction might be a conditional branch; it might also be a jump, indirect jump, call, virtual call, or return. It might also not be a control-flow instruction at all.
- So
  - you index the BTB, you get a predicted next instruction; you also get a tag - you check the tag to see if it matches this particular instruction.
  - If you do get a BTB tag hit, you know it's a control-flow instruction.
  - In parallel with the BTB access, you index the direction predictor; if the direction predictor says the branch is predicted taken, you use the BTB's prediction for the branch target - if it says not-taken, you use PC+1 regardless.
  - If the direction predictor has no prediction for this PC address, you use the BTB prediction, because the instruction is an unconditional jump/call.
- This is complicated by the fact that the BTB is indexed by the PC address only, but the direction predictor is probably indexed by a combination of PC address and branch history. And the direction predictor might not have tags (to save space, to allow a larger number of predictions within the same transistor/energy budget).
- So I was simplifying!

Student question: delayed commit & return address prediction

- I am unsure about how the RAP should be updated. The slides mention that updating the RAP at commit time could mean we may not produce a return address prediction in time, and if we update the RAP before commit, we would have a similar issue as updating the BTB before commit.

- In what cases would the RAP not be able to produce a prediction in time if we update it at commit?

- Consider a timeline like this - where we have some code in a function F that calls a function G, which does some stuff then returns to F:

```
F1
call G
G1
G2
ret
F2
```

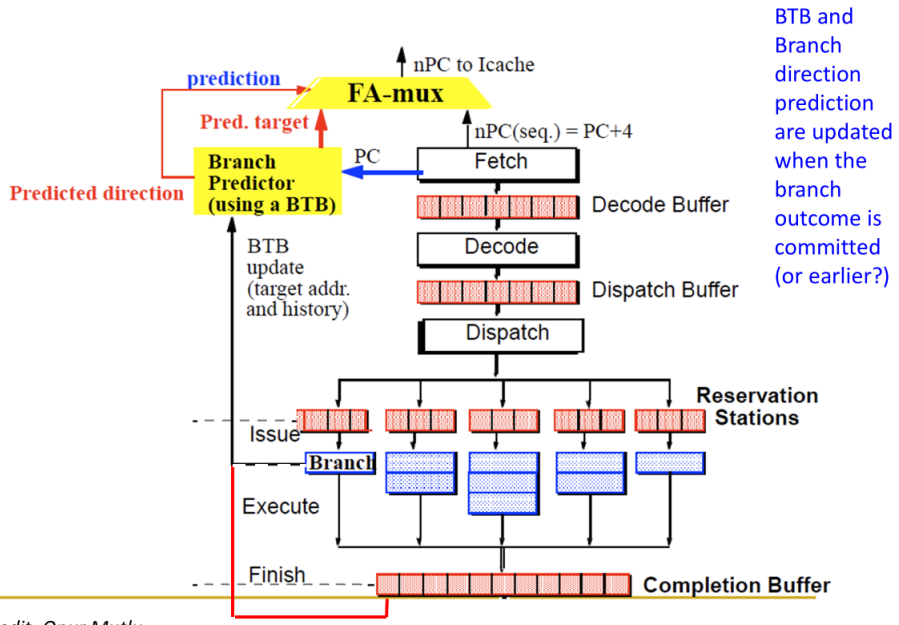
- So we need the RAP to predict where the "ret" will go.

- If we don't update the RAP until commit, we would need the "call" to be committed before the branch prediction for the "ret" is made.

- If the call is not yet committed at the point where we do the return, we will use the wrong (earlier) entry from the top of the RAP's stack.

- So what people actually do is to have a copy of the RAP for the speculated path (you could think of this as register renaming, applied to the RAP's state). So now we can update the RAP at issue time. But if we discover a misprediction, we revert the RAP to the state it had at the point of the mispredicted branch.

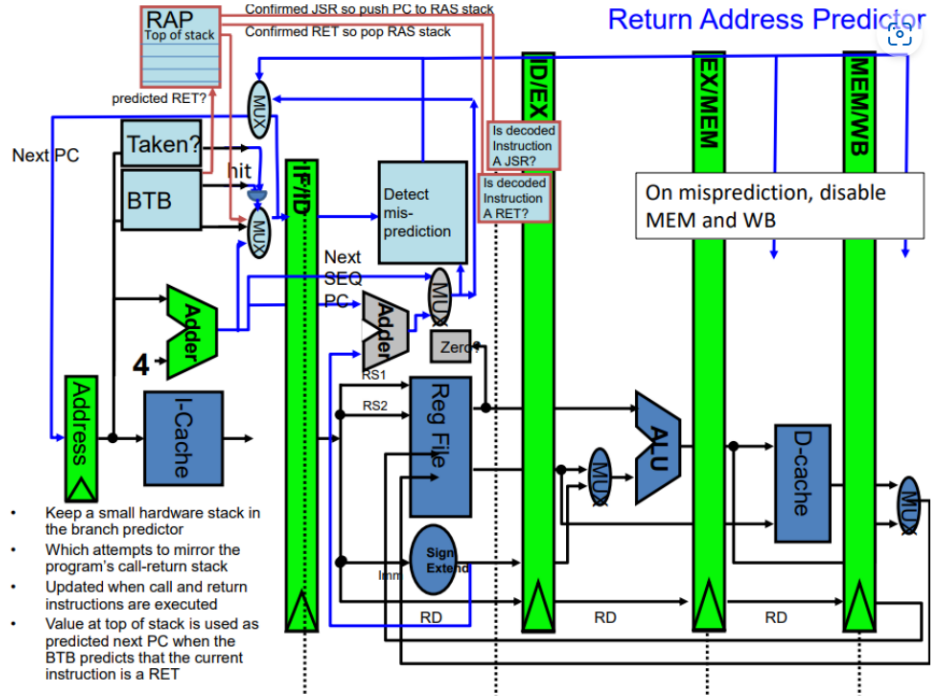
Updating the branch prediction



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

Credit: Onur Mutlu

Student question #2 on delayed commit & return address prediction



In the lectures it was mentioned that RAP can only be updated once we know that branch misprediction has not occurred, which confuses me.

From what I understand, when a RET instruction is in the decode stage, the "*detect mis-prediction*" block checks whether the instruction fetched **after** the RET is a misprediction. In which case, why would the difference in instruction being executed **after** the RET, change our decision to "pop" a return address from the stack?

The issue is not whether the return instruction is a misprediction (although it might be).

The issue is that we might execute a call, or a ret, conditionally - that is, there is some other conditional branch that determines whether the call/ret is executed.

Consider for example:

```
1 F: BEQ R1, L
2    CALL G
3 L: RET
4
5 G: ...
6    ... (for many instructions)
7    ...
8    RET
```

In this example, the function F conditionally calls a function G then returns. Suppose the BEQ is mispredicted as not-taken. So we speculatively jump to G (and push the address after that call to G, ie instruction 3, onto the RAP.

After a while we discover the misprediction (perhaps before line 7). So we rollback to line 1, and correctly branch to line 3, where we encounter the RET. At this point, we pick up the address at the top of the RAP - which is the wrong return address (it's the one that was pushed when we speculatively executed line 2).

So now we regret speculatively updating the RAP.

Of course if our branch prediction had been *correct*, we would be happy to have updated the RAP.

So, ideally, what we want is to clone the RAP each time we encounter a speculated conditional branch.

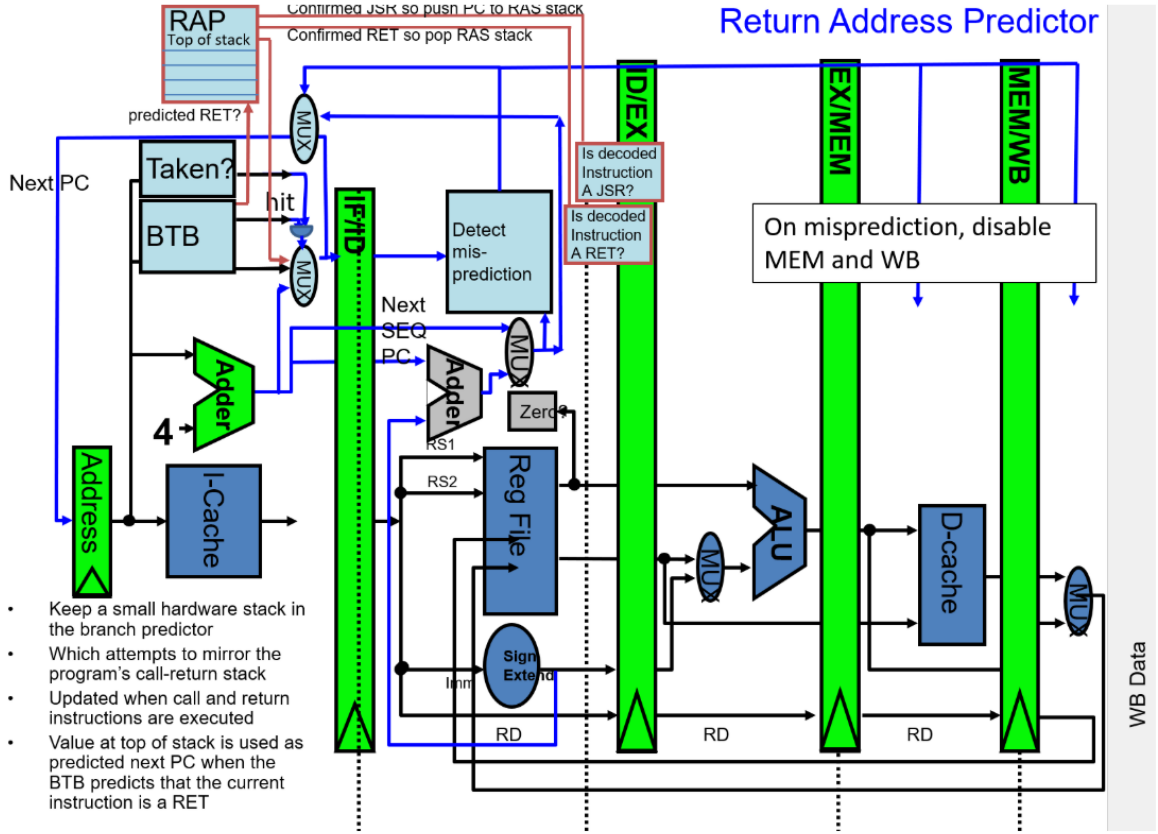
My explanation above is intended to make sense with a more complicated pipeline than the one in the slide that you quote - in a very simple pipeline, some streamlining should be possible.

I *think* that in the pipeline on the slide, we should be OK unless a CALL jumps to a RET instruction, as there is a cycle latency to update the RAP.

Student question: how do we know when to query the return-address predictor?

Q: In the Return Address Predictor (RAP) diagram, which hardware block actually determines that the current instruction is a RET ? Is it the BTB, the RAP itself, or some earlier decode logic?

A: The relevant slide is, I think, this one:



You *could* fetch and decode the RET, discover that it's a RET, and query the RAP.

But that would deliver the next address too late.

Recall that the job of the BTB is to tell us whether the next instruction is a branch/jump, and if so, what it's predicted target is.

So we extend the BTB to tell us if the next instruction is a RET. If it is predicted to be a RET, we query the RAP.

- You might also enjoy https://docs.kernel.org/admin-guide/hw-vuln/srso.html (after we have covered speculation side channel vulnerabilities)

Student question: How could the return-address predictor be wrong?

Q: Also, why is the RAP considered a predictor? From the slides it looks like a small hardware stack mirroring the call stack, not something that "guesses" like a direction predictor

A:

We push the return address onto the RAP when we encounter a CALL

We pop it from the RAP when we encounter a RET

But wait – isn't that what the CALL and RET instructions *do*?

No: the ISA specifies that the CALL instruction saves the return address on the stack actually stack, in the RAM*

But the RAP is an internal hardware structure that "mirrors" the real stack that is in RAM.

So most of the time they hold the same data

But sometimes they don't.

Examples:

- A function overwrites its return address on the stack using a store instruction. So when we return, the RAP predicts that we should return to the caller, but the real stack says otherwise
- You might change the stack pointer completely, for example because we are switching to another thread.
- The RAP has limited capacity – so we might encounter a RET but find that the RAP's stack is empty. The return address is safely on the real stack.

- See the discussion of retpolines in Ch5

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

- You might wonder whether branch predictors are already as good as can be. See
  - C. Lin and S. J. Tarsa, "Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions," in 2019 IEEE International Symposium on Workload Characterization (IISWC), Orlando, FL, USA, 2019, pp. 228-238, doi: 10.1109/IISWC47752.2019.9042108. https://arxiv.org/abs/1906.08170