EECS 470 Final Project

Tips and Common Questions

Department of Electrical Engineering and Computer Science
College of Engineering
University of Michigan

Thursday, September 25th, 2025

Overview

RTL Design Principles

- Backpressure
- Encoding
- Problem Decomposition

Advanced Feature Descriptions

- Early Tag Broadcast
- Early Branch Resolution
- GUI Debugger
- Simpler Features

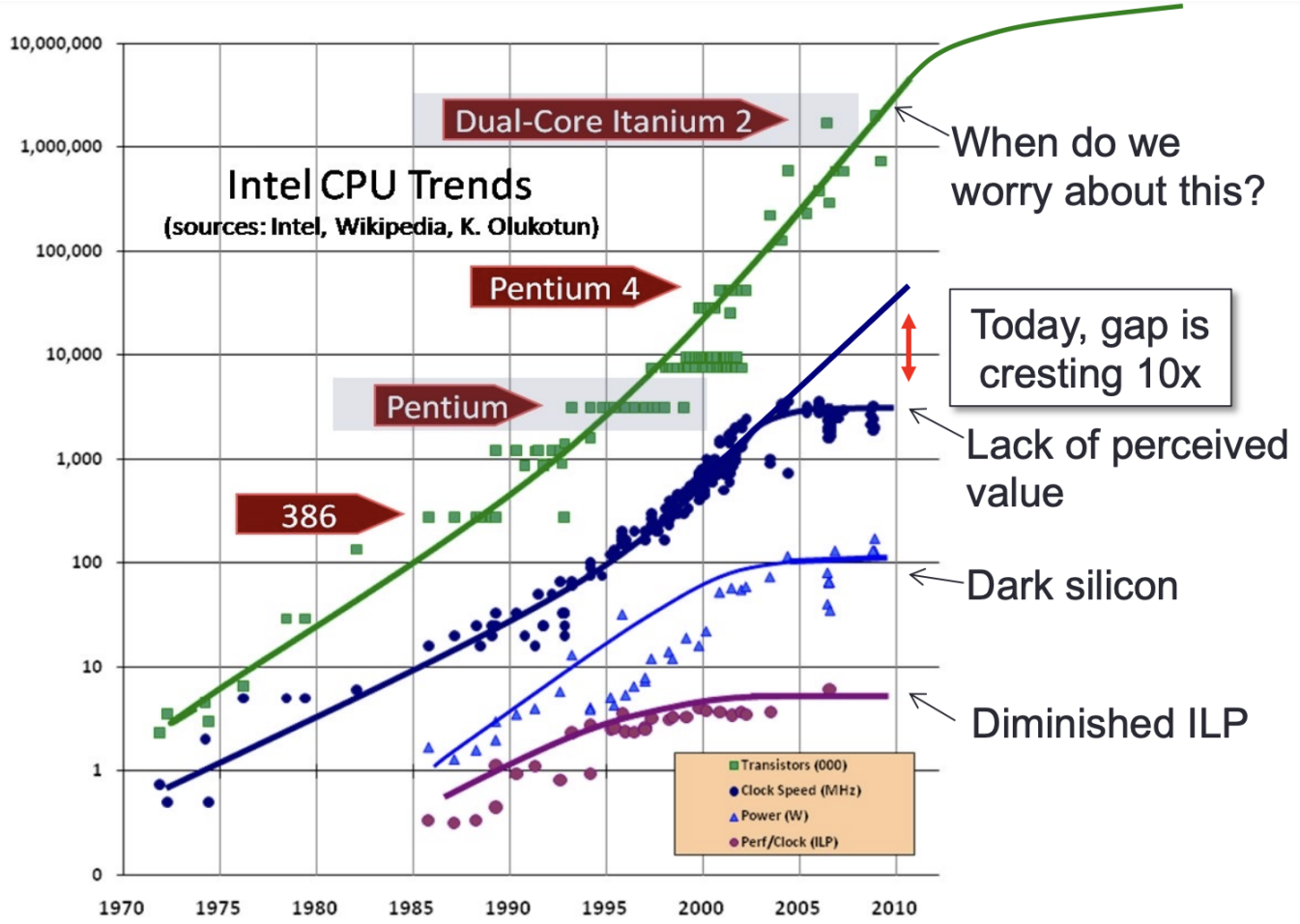
Implementation Tips

- Testing
- Branch Prediction
- CDB and Instruction Identification

RTL Design Principles

Overview

First, we'll go over some basics for good hardware design



Backpressure

Problem

- Bottlenecks in the system may cause stalls 🫠
  - CDB, caches, structural hazards
- Adding buffers everywhere is too expensive
  - Especially around functional units where real-estate is precious
  - Must be sized for worst case

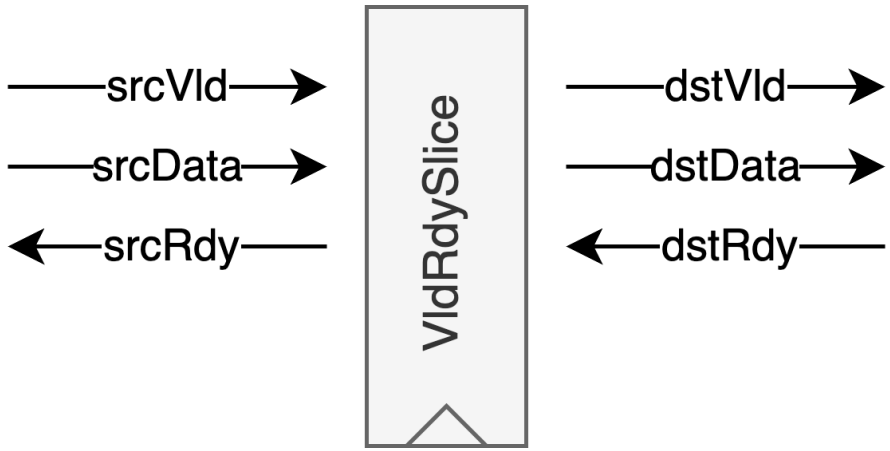
Solution

- Propagate stalls backwards through pipeline registers 🪱
- Data can't move forward if next stage is stalled
- Implement with a valid - ready handshake
  - Source asserts `valid` when it has data to send
  - Destination asserts `ready` when it can receive new data
  - Data only moves when both `valid` and `ready` are asserted

Backpressure Implementation

Functional Units

- Avoid a (very large) CDB buffer between functional units and CDB
  - Would have to be almost as large as the ROB to ensure correctness
- Instead, stall functional units when they don't get access to the CDB
- RS can't send data to stalled functional unit 😕



Encoding Transformations

Motivation

- Often you have to go between one-hot and encoded representations
  - Ex: `psel_gen` uses one-hot, but may feed a mux requiring an encoded value
- Certain interfaces work better with different representations

Binary2Onehot

```
out = '0;
for (int i = 0; i < OUT_BITS; i++) begin
    if (in == i) out[i] = 1'b1;
end
```

Onehot2Binary

```
for (int i = 0; i < IN_BITS; i++) begin
    if (in[i]) out = unsigned'(i);
end
```

Problem Decomposition

High Level

- Engineering is all about decomposing problems into simpler ones 🙌
  - ISA abstracts hardware $\leftrightarrow$ software
  - Memory interface abstracts underlying cache hierarchy 🥴
- Digital design is no different!

Common Hardware Design Problems

- **Buffering**: FIFOs, valid-ready slice
- **Selection/allocation**: priority selectors, muxes
- **Transformations**: bitwise operations, binary2onehot, onehot2binary

Problem Decomposition

Perks of Reuse

- Clean hierarchy 👑
- More readable code 📖
- Fewer modules to verify ✅
- . . . and therefore less work! 😇

Setting Yourself Up

- Write small modules that solve simple problems 🧰
- Make modules parametrizable 🧩
- Document interfaces 📄

Advanced Feature Descriptions

Now, we'll go over some more details some of the advanced features

- Early Tag Broadcast
- Early Branch Resolution
- GUI Debugger

Disclaimer!

The point values given here are only historical references! Actual point allocations may vary semester to semester.

Early Tag Broadcast

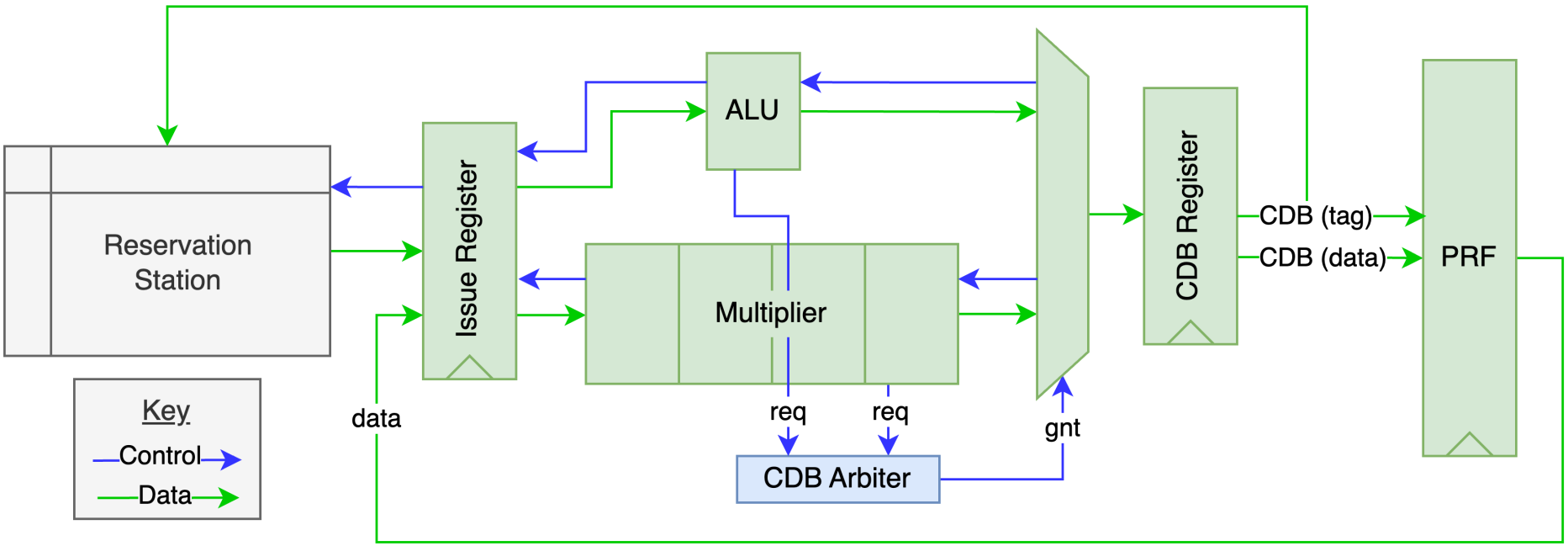
Problem

- Wakeup, select, and issue requires a long combinational path
  - CDB→update tags→mark as ready→select from ready instructions →route to issue register
- Need to split into 2 cycles or suffer a long clock period 😅
- If doing 2 cycles, there is a cycle gap between dependent instructions
  - Therefore max IPC of 0.5 😔

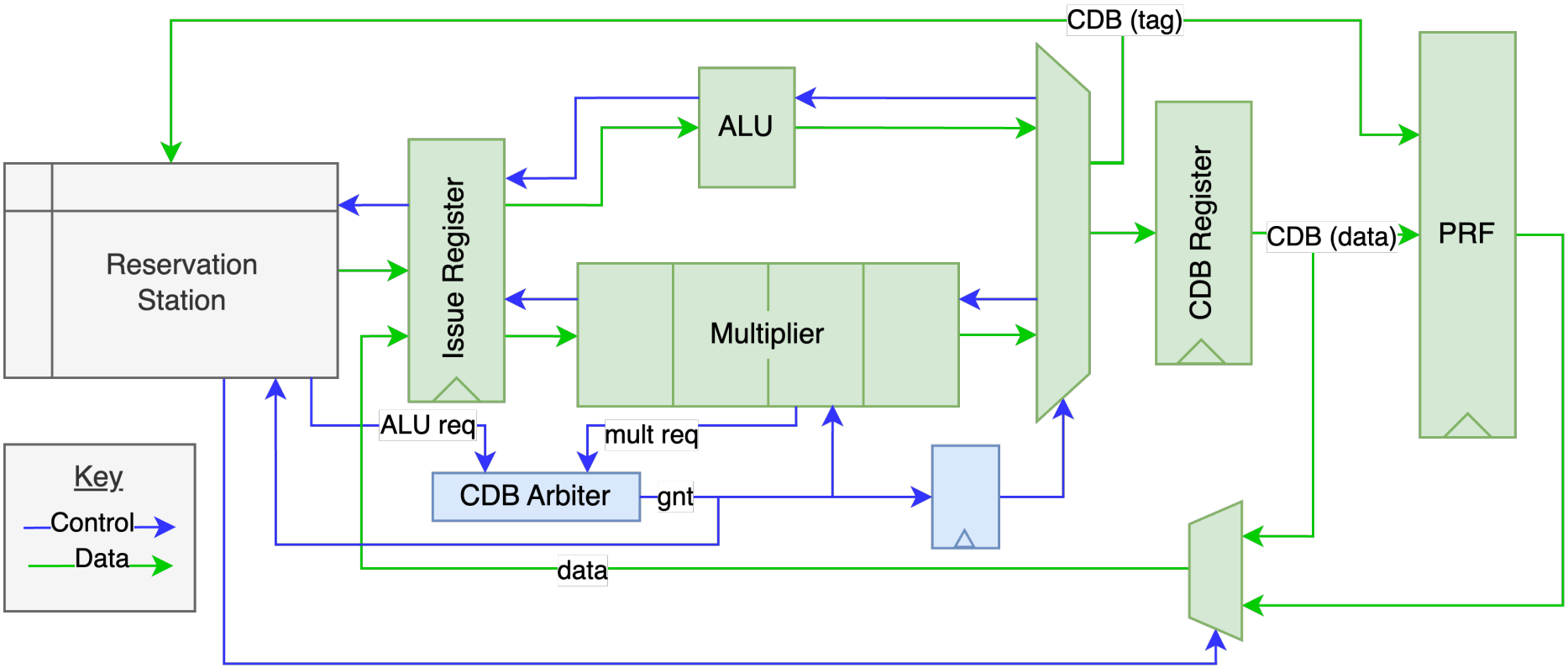
Solution

- Pipeline the issue stage by sending tag one cycle before data!
- Main impact: need to arbitrate for CDB one cycle earlier
  - For one cycle latency operations, CDB arbitration must be considered during issue

Non-ETB Architecture



ETB Architecture



Early Branch Resolution

Normal Approach

- Wait until a mispredicted branch hits the head of the ROB
- Recovery: clear everything (simple 😀)
- Issue: wastes cycles that could have been used for productive work 😒

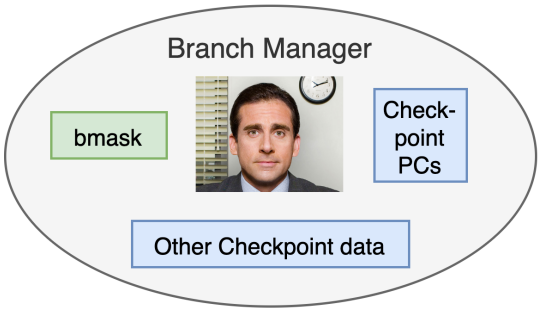
Optimization

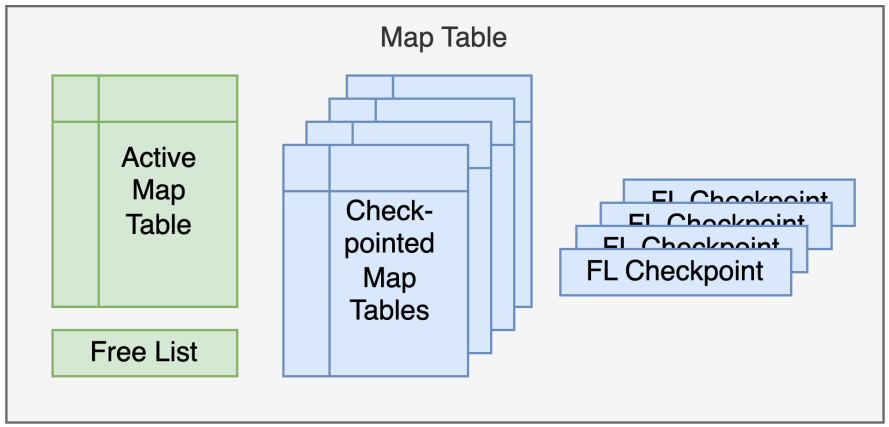
- Recover as soon as you know the result of a branch
- Difficulty: need to selectively squash just the instructions on the mispredicted path
- Requires taking checkpoints when branch is dispatched
- New structural hazard unlocked ⚠️: no more checkpoints remaining

EBR Implementation

Checkpoints

- Take snapshot of PC, map table, free list, store queue tail (if using)
  - Can accomplish without storing ROB tail
  - Cleaner to keep map table checkpoints in map table module
- Allocate bit in branch mask bit to dispatching branch
- Update checkpointed map table and free list as instructions retire
  - Recovery needs to reflect retirements after the branch dispatched
- No longer need architected map table





EBR Implementation

Recovery

1. Broadcast misprediction bmask and squash all instructions with a 1 at that bit
   - Search RS and all functional units
2. Restore ROB tail to the mispredicted branch
   - Can do this without ROB tail checkpoint if using PR allocated to branch to find it in the ROB
3. Restore map table and free list to checkpointed version (including updates from retired instructions)
4. Correct PC and branch predictor state
5. If applicable, clear instruction buffer

What about Fast Branch Recovery?

- Fast branch recovery refers to recovering from a misspeculation in one cycle instead of doing a serial rollback
- Groups pretty much always do fast branch recovery whether that's with or without early branch resolution

GUI Debugger

Motivation

- Debugging with waveforms does not scale
- Very helpful to visualize the structures in your design
  - RS, ROB, LSQ, caches
- Can dump everything to a .txt file . . .
- . . . but that can make unnecessarily huge files
  - which are difficult to navigate
  - And may crash the autograder 💀
- A good GUI debugger will save you lots of time later!

GUI Implementation

Back End

- Goal: run test programs like normal, but dump state for external use
- Options:
  - **`.vcd` or `.vpd` files:** good documentation, human readable
  - **`.fsdb` files:** used by verdi for waveforms
  - **VCS ucli interface:** control the simulation while it's running (less storage of state needed)
  - **Custom `.json`** written directly by testbench: most flexibility with how you parse files and easy integration with existing `json` libraries

For VCD or VPD files

- Use`$dumpvars` or `$vcdpluson` in the testbench to set which signals to dump 💩
- vpd format supports structs better
  - See reference sheet for more details

GUI Implementation

Front End

- If not using `ucli` interface, you can run your GUI offline (i.e. not on CAEN)
- Choose your favorite front-end development tool
  - React, JavaScript, Python, ncurses ...
- Seen students do some pretty amazing things 👀
  - Run on a remote server and forward through the web for live collaboration on the same program
  - Fully modular design that automatically adjusts to different design parameters
  - Highlighting signal changes between cycles

GUI Example


Chimp Debugger
D-Cache
I-Cache
Store Queue
Branch Predictor
ROB
RS
Inst Buffer
Physical Registers

Branch Prediction

Baseline

- For initial integration, most groups begin with a simple not-taken branch predictor
- However, this **does not** meet the baseline requirements
  - It doesn't have address prediction
- Need to implement at least a bimodal predictor and a BTB

Common Choices

- Gshare: (1 point): simple enough and may have decent performance
  - Speculative global history updates: updating global history on predictions and roll back state on a mispredict
- Tournament predictors (2 points): Gshare, pag, gag, 2-bit
- Other optimizations include a RAS, pipelined prediction, perceptron predictor, or whatever else you can think of

Memory

LSQ

- Store to load forwarding is a simple advanced feature (1 point)
  - Additional point if done at byte level
- Load speculation is far more complex (4+ points)
- See LSQ lab slides for more details

Caches

- Non-blocking (using MSHRs) = 1 point
- Fixed associatitivity = 0.5 points
- Parametrizable associativity = 1 point
- Victim cache = 1 point

Implementation Tips

Finally, it's time for some tips with your project implementation

- Testing
- Branch Prediction
- CDB Broadcasting
- Instruction Identification

Testing

Non-Fragile Tests

- Your test shouldn't fail if you change parameters 🙈
- This keeps tests usable as you adapt your design
- Tests are more likely to be correct if they're not hard-coded

Why Does Testing Take So Long?

- Expect to spend 60-90% of your time playing around with your testbench rather than implementing your module . . .
- Ensuring correctness is the hardest part of the design process
- Your errors will come from things you didn't think of 🙃

Verification Methods

Having a "Golden" Model

- Verify implementation against a reference model (in testbench or pre-existing one)
- Testbench model can use non-synthesizable constructs 🥳
  - Use this to design it differently than main implementation
- More useful and effective if model is an existing version (either hardware or software) that you know works
- **Pro**: Non-fragile, easy to check more outputs
- **Cons**: Lots of extra code, may be difficult to make golden model not share the same logic as your dut, need to verify the model as well

Verification Methods

Immediate Assertions

- Manually calculate the correct output and check that your module's output matches 🔍
- **Pro**: Simplest to implement
- **Con**: Very fragile

Embedded Assertions

- Encode properties within the design itself 👀
- **Pros**: Clean way to document functionality, can use to formally verify (and not have to write a testbench!), remains active when doing system-level testing
- **Cons**: Need to learn SystemVerilog assertion language

Branch Prediction

When To Predict

- Predict at fetch so you know the next PC to get
- If doing EBR, take checkpoint at dispatch

How To Predict

- Use BTB to see if it's a branch
  - Miss $\rightarrow$ this is not a taken branch we've seen before
  - Hit $\rightarrow$ this is a branch that we've seen before
- Using BTB means you can see if it's a branch without needing to get data from cache
- Alternatively, have a small "pre-decoder" after the data comes back from cache to check if opcode is a branch or jump

Updating Predictors

Global History

- More accurate: update speculatively, based on prediction
  - Ensures branches see deterministic history based on location in program
  - Until branch completes, previous state needs to be recoverable 
- Keep extra bits of history to enable recovery
  - # extra bits = # possible outstanding branches (minus 1)
  - To recover: right shift history by # of branches you need to undo
- . . . but if this is too complicated, you don't have to

Direction Predictors

- Better to update only when you know the outcome
  - Simplifies update logic

CDB Broadcasting

Restrictions

- We restrict CDB size since this is the main datapath bottleneck
  - In real life, these are large, long, high-fanout nets that are expensive
- Anything that writes data to a destination register needs to broadcast on the CDB to alert all relevant modules that data is ready

What about Branches and Stores?

- These don't have a destination register
- Can wire directly to (small) number of dependent modules
  - **Branches**: ROB, branch predictor
  - **Stores**: ROB

Identifying Instructions

Use Physical Registers!

- If you have `ROB_SZ + NUM_ARCH_REG` physical registers, you can give every in-flight instruction a unique physical register
  - Smoother, simpler dispatch logic 
- Use that physical register to uniquely identify instructions

Stores

- Usually identified by store queue index