EECS 370 - Lecture 15

Control Hazards

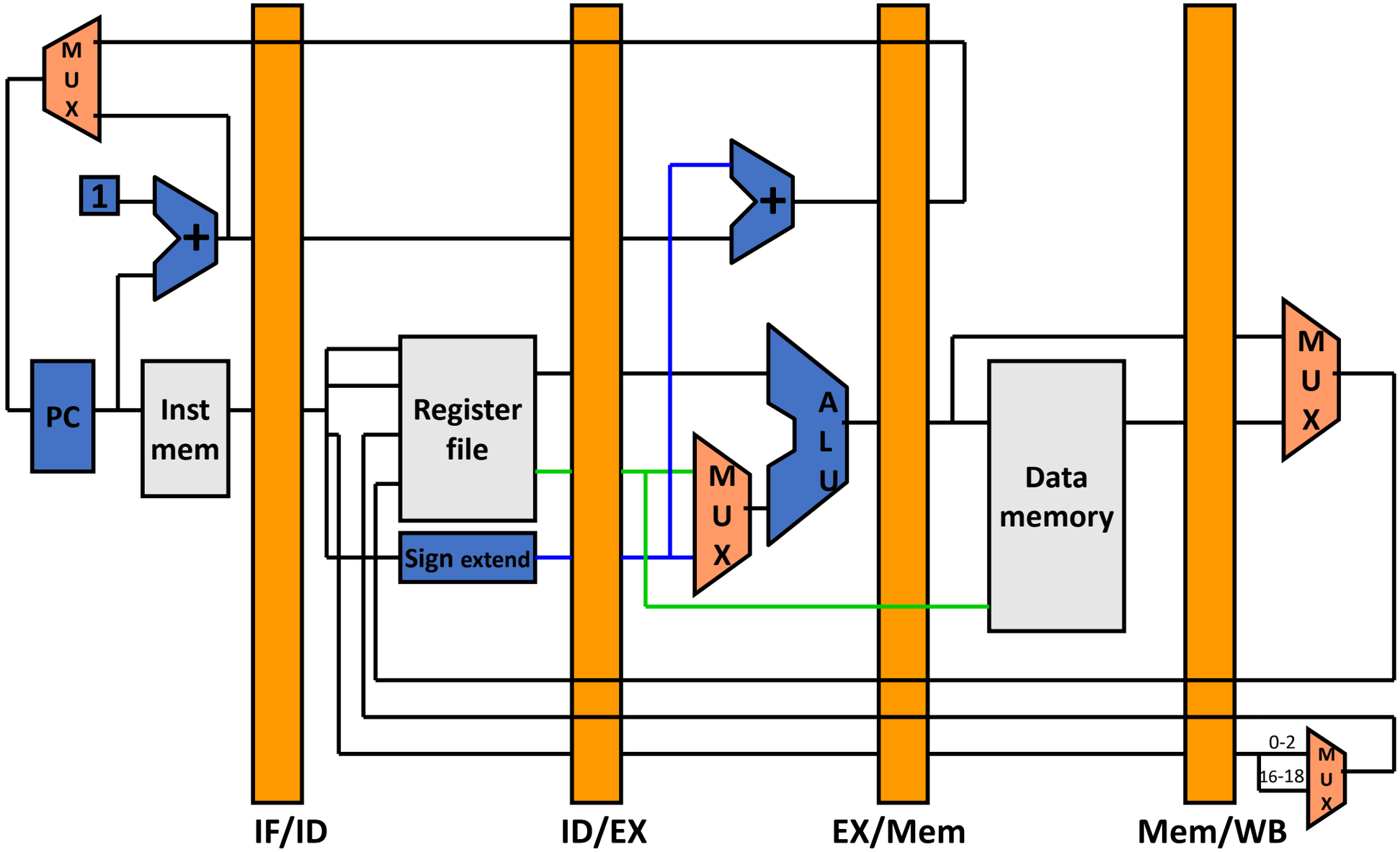
Announcements

- 2L deadline pushed until Tuesday after break
  - (I wouldn't recommend putting it off until you get back… post-break brain usually takes a couple days to get back in the swing of things)
- Exam Thursday after break
  - Sample exams on website
- **Review lecture on Tuesday, no lecture Thursday**

Resources

- Pipelining not clicking? Try playing with the "Pipeline Simulator" under "Resources" on the website
  - https://vhosts.eecs.umich.edu/370simulators/pipeline/simulator.html
  - Several pre-written programs you can step through to understand what's going on
  - Note that the project pipeline is slightly different

Review: New Datapath



Other issues

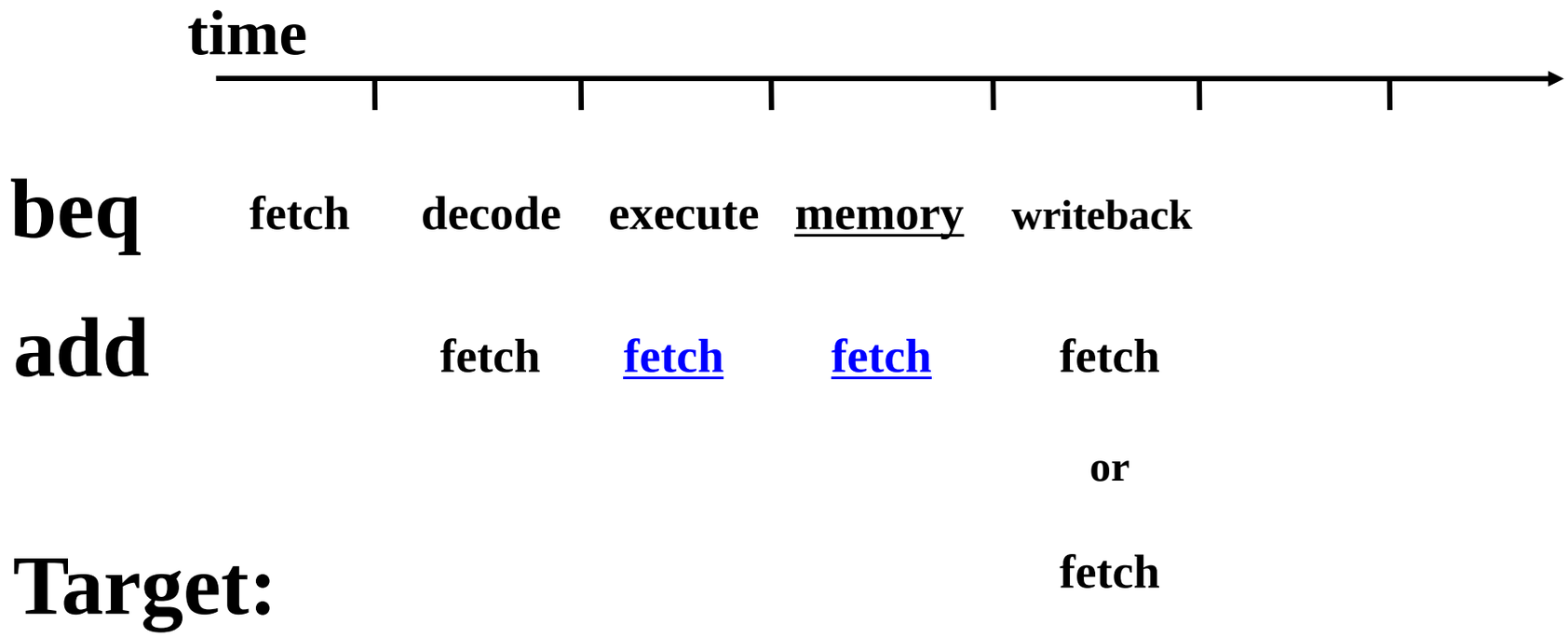
- What other instruction(s) have we been ignoring so far??
- Branches!! (Let's not worry about jumps yet)
- Sequence for BEQ:
  - Fetch: read instruction from memory
  - Decode: read source operands from registers
  - Execute: calculate target address and test for equality
  - Memory: **Send target to PC** if test is equal
  - Writeback: nothing
  - **Branch Outcomes**
    - Not Taken
      - PC = PC + 1
    - Taken
      - PC = Branch Target Address

Detect and Stall

- Detection
  - Wait until decode
  - Check if opcode == beq or jalr
- Stall
  - Keep current instruction in fetch
  - Insert noops
  - Pass noop to decode stage, not execute!

Control Hazards

beq 1 1 10
add 3 4 5



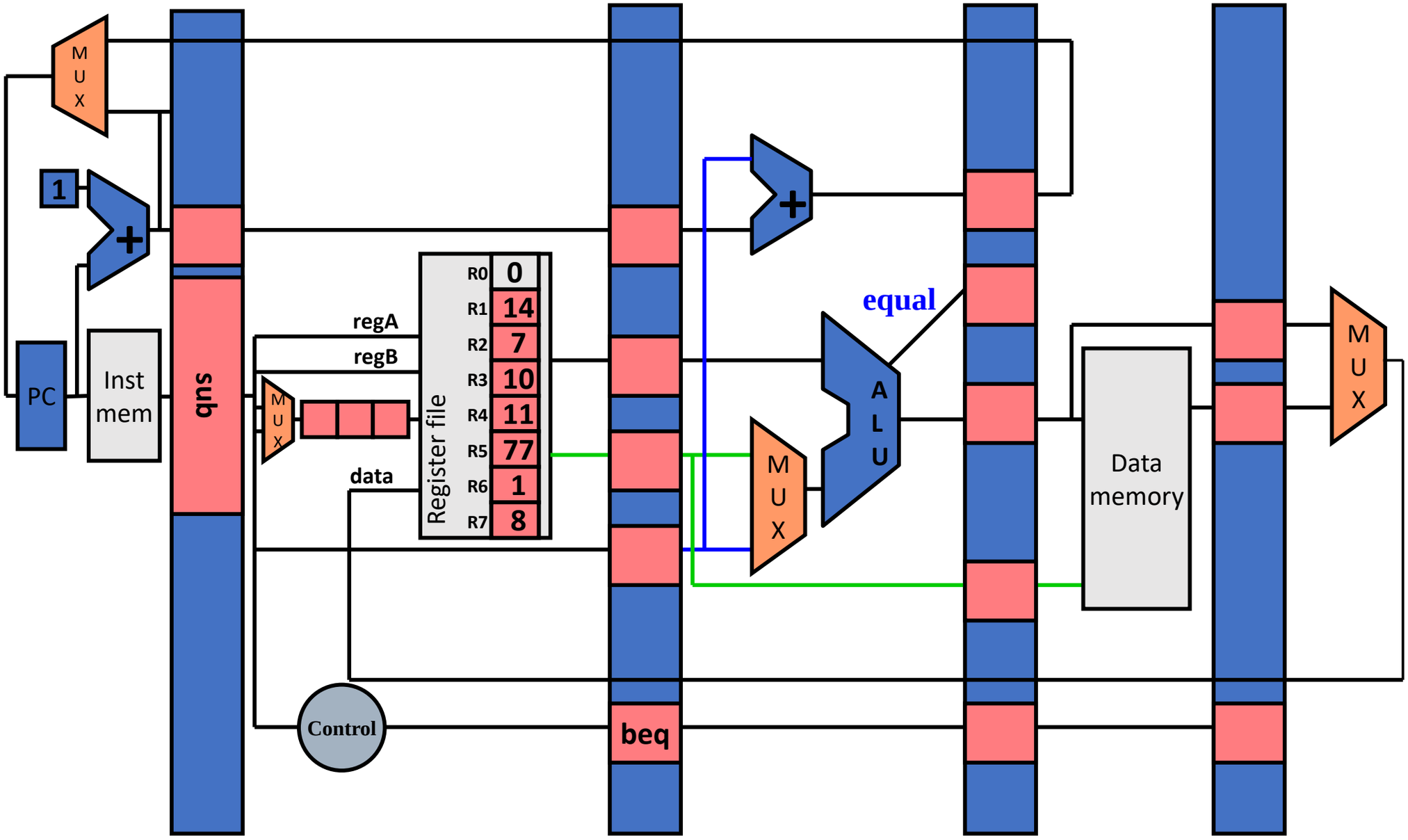
Problems with Detect and Stall

- CPI increases every time a branch is detected!
- Is that necessary? Not always!
  - Branch not always taken
  - Let's assume it is NOT taken…
    - In this case, we can ignore the beq (treat it like a noop)
    - Keep fetching PC + 1
  - What if we're wrong?
  - OK, as long as we do not COMPLETE any instruction we mistakenly execute
  - I.e. DON'T write values to register file or memory

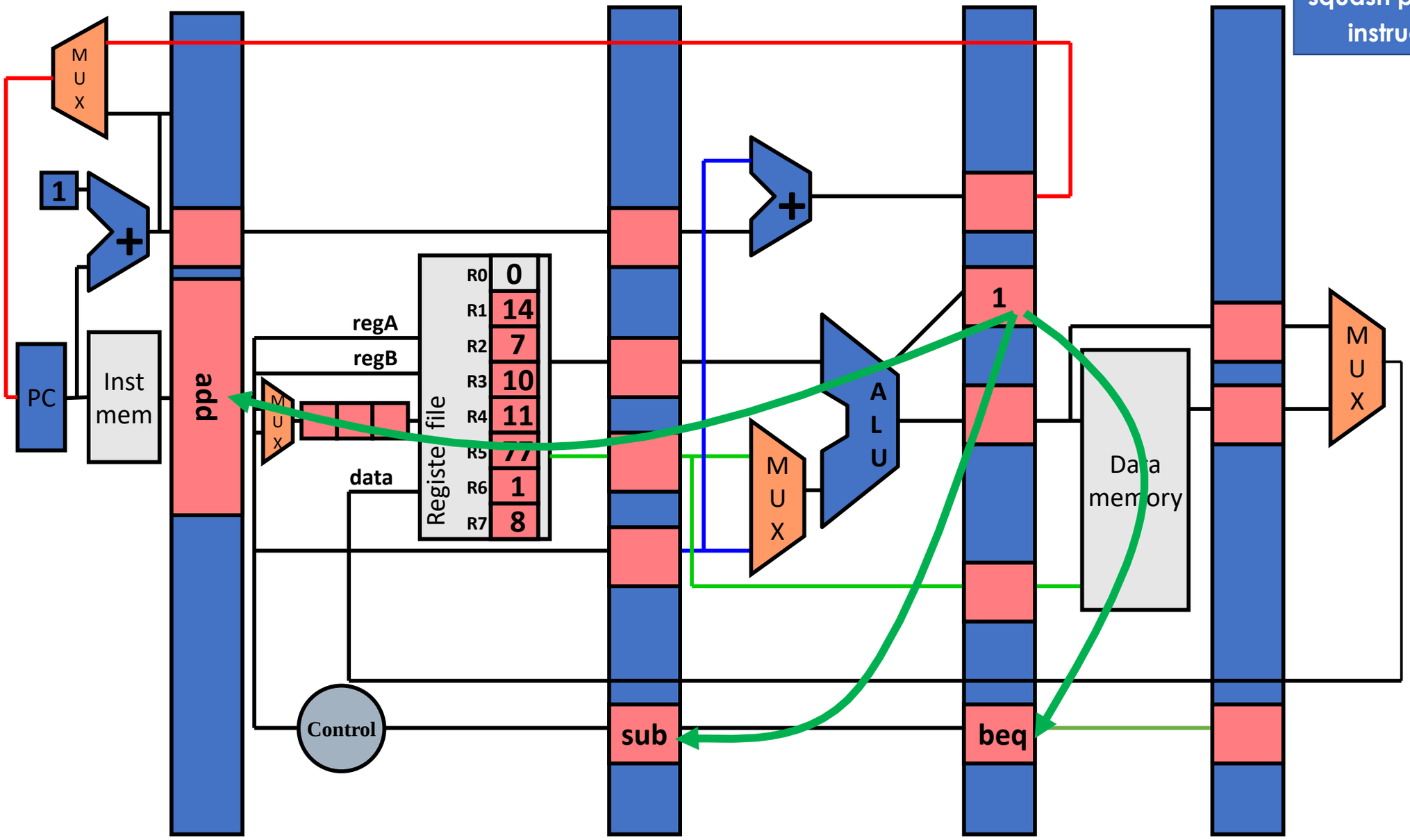
Speculate and Squash

- Speculate: assume not equal
  - Keep fetching from PC+1 until we know that the branch is really taken
- Squash: stop bad instructions if taken
  - Send a noop to Decode, Execute, and Memory
  - Sent target address to PC

Resolve branch in execute stage



Send target to PC in Mem stage, squash previous 3 instructions



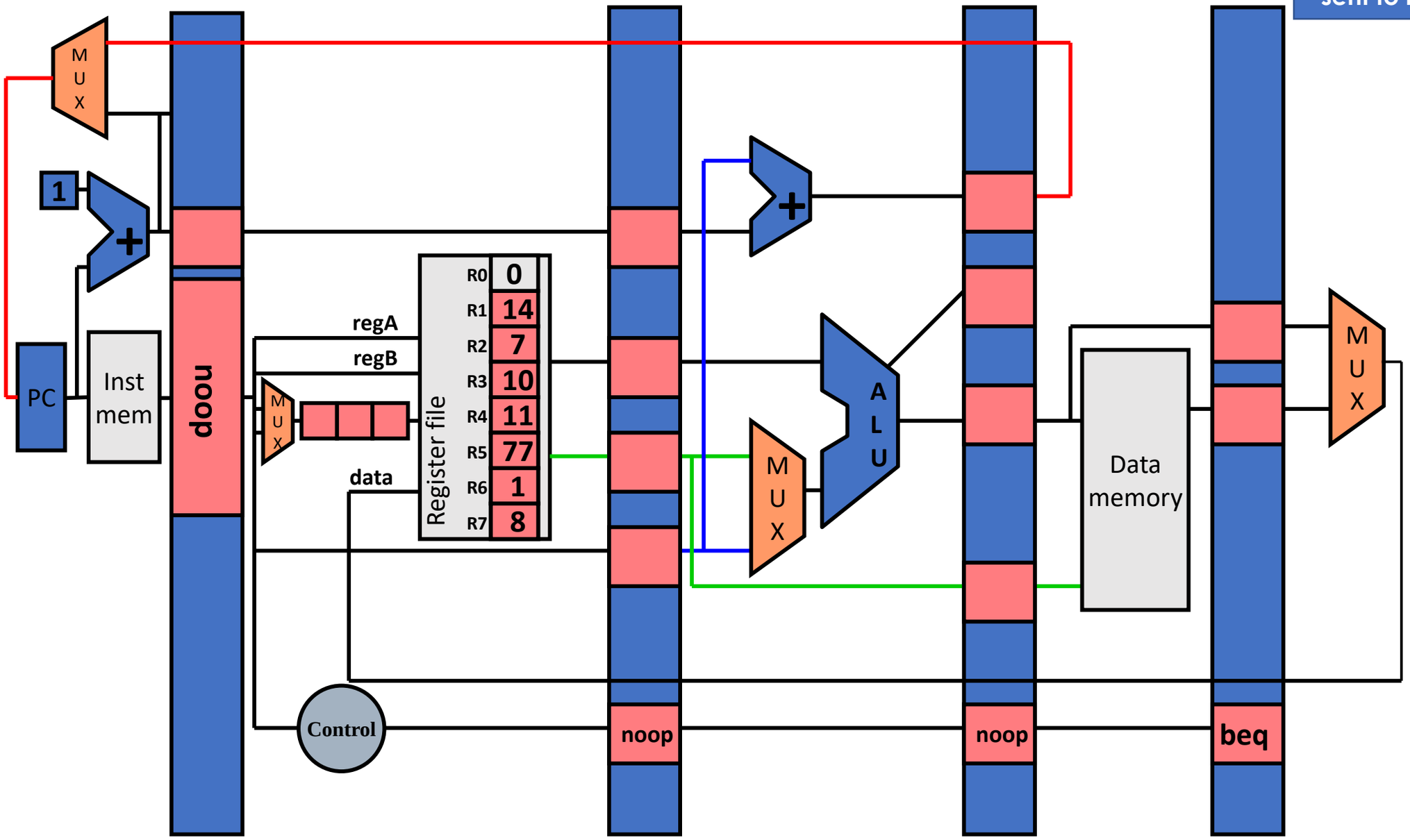
IF/ ID

ID/ EX

EX/ Mem

Mem/ WB

3 noops inserted, correct PC being sent to memory



What can go wrong?

- Data hazards: since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if is about to be written.

- Control hazards: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?

- Exceptions: How do you handle exceptions in a pipelined processor with 5 instructions in flight?

Exceptions

- Exception: when something unexpected happens during program execution.
  - Example: divide by zero.
  - The situation is more complex than the hardware can handle
    - So the hardware branches to a function, an “exception handler” which is code to try to deal with the problem.
- The exact way to set up such an exception handler will vary by ISA.
  - With C on x86 you would use <signal.h> functions to handle the “SIGFPE” signal.
  - There is a pretty good Hackaday article on this if you want to learn more.

Exceptions and Pipelining

- The hardware branches to the “exception handler”
  - This means that any instruction which can “throw” an exception *could* be a branch.
  - Throwing an exception should be rare (“exceptional”)

- So we would treat it much like a branch we predicted as “not taken”
  - Squash instructions behind it and then branch
  - It will introduce stalls, but since it should be rare, we don’t worry about it.
    - “Make the common case fast”.

Classic performance problem

- Program with following instruction breakdown:

lw	10%
sw	15%
beq	25%
R-type	50%

- Speculate “always not-taken” and squash. 80% of branches not-taken
- Full forwarding to execute stage. 20% of loads stall for 1 cycle
- What is the CPI of the program?
- What is the total execution time if cycle time is 100MHz?

Classic performance problem

- Program with following instruction breakdown:

lw	10%
sw	15%
beq	25%
R-type	50%

- Speculate “always not-taken” and squash. 80% of branches not-taken
- Full forwarding to execute stage. 20% of loads stall for 1 cycle
- What is the CPI of the program?
- What is the total execution time if cycle time is 100MHz?

CPI = 1 + 0.10 (loads) * 0.20 (load use stall)*1

+ 0.25 (branch) * 0.20 (miss rate)*3

CPI = 1 + 0.02 + 0.15 = 1.17

Time = 1.17 * 10ns =11.7ns per instruction

Classic performance problem (cont.)

- Assume branches are resolved at Execute?
  - What is the CPI?
  - What happens to cycle time?
  - What is the total execution time?

Classic performance problem (cont.)

- Assume branches are resolved at Execute?
  - What is the CPI?
  - What happens to cycle time?

CPI = 1 + 0.10 (loads) *0.20 (load use stall)*1
+ 0.25 (branch) * 0.20 (miss rate)*2
CPI = 1 + 0.02 + 0.1 = 1.12

Performance with deeper pipelines

- Assume the setup of the previous problem.
- What if we have a 10 stage pipeline?
  - Instructions are fetched at stage 1.
  - Register file is read at stage 3.
  - Execution begins at stage 5.
  - Branches are resolved at stage 7.
  - Memory access is complete in stage 9.
- What's the CPI of the program?
- If the clock rate was doubled by doubling the pipeline depth, is performance also doubled?

Performance with deeper pipelines

- Assume the setup of the previous problem.
- What if we have a 10 stage pipeline?
  - Instructions are fetched at stage 1.
  - Register file is read at stage 3.
  - Execution begins at stage 5.
  - Branches are resolved at stage 7.
  - Memory access is complete in stage 9.
- What's the CPI of the program?
- If the clock rate was doubled by doubling the pipeline depth, is performance also doubled?

CPI = 1 + 0.10 (loads) *0.20 (load use stall)*4 + 0.25 (branch) * 0.20 (N stalls)*6

CPI = 1 + 0.08 + 0.30 = 1.38

Time = 1.38 * 5ns = 6.9 ns per instruction

Can We Improve Branch Performance?

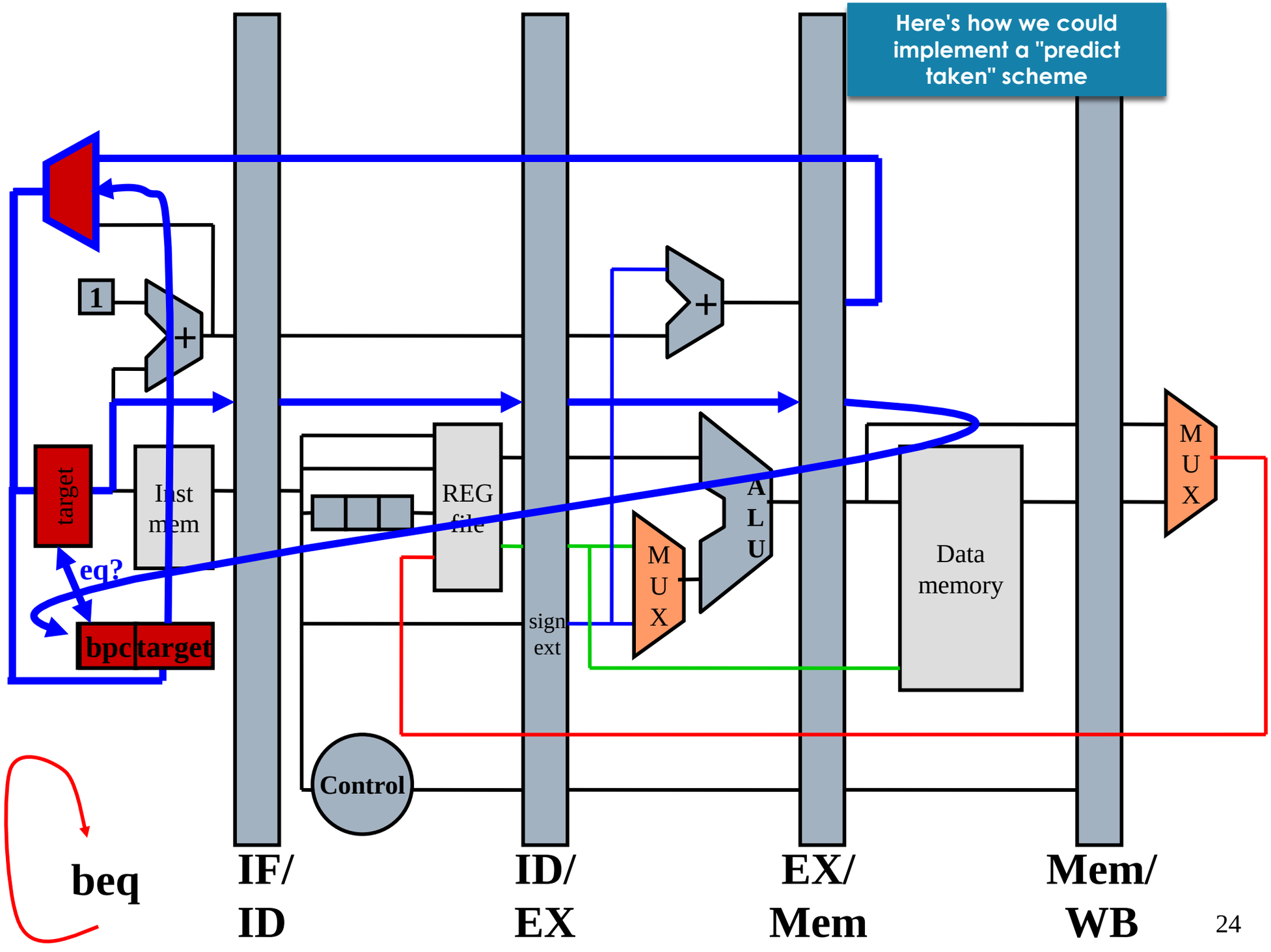
- CPI increases every time a branch is taken!
  - About 50%-66% of time
- Is that necessary?

- **No!** We can try to predict when branch is taken
  - But we would need to send target PC to memory before decoding branch
  - How do we:
    1. Know an instruction is a branch before decoding?
    2. Reliably guess whether it should be taken?
    3. Figure out the target PC before executing the branch?

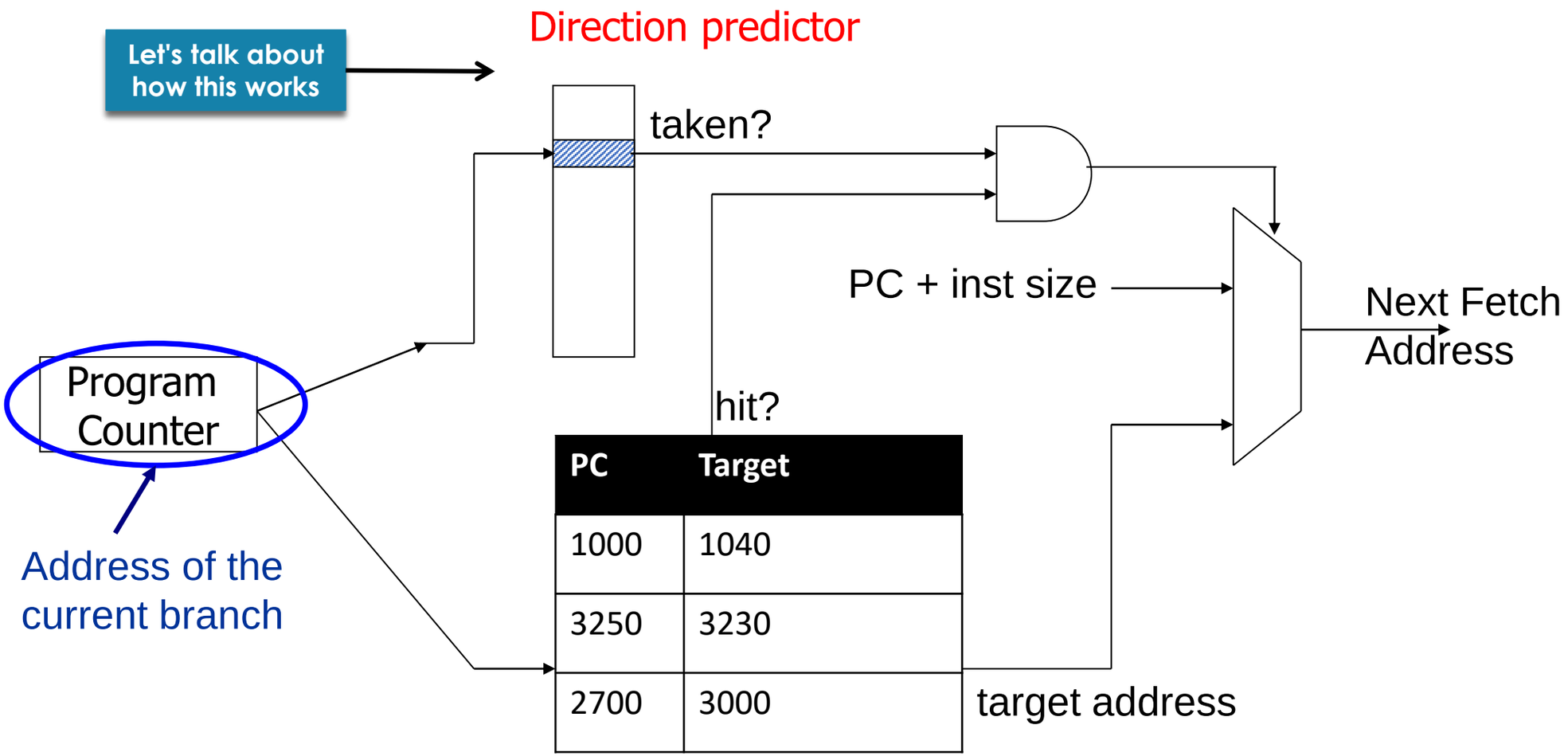
Sometimes predict taken?

- When fetching an instruction, need to predict 3 things:
  1. Whether the fetched instruction is a branch
  2. Branch direction (if conditional)
  3. Branch target address (if direction is taken)

- Observation: Target address remains the same for conditional branch across multiple executions
  - Idea: store the target address of branch once we execute it, along with PC of instruction
  - Called Branch Target Buffer (BTB)


Here's how we could implement a "predict taken" scheme
1
+
target
Inst mem
eq?
bpc
target
REG file
sign ext
Control
+
M U X
A L U
Data memory
M U X
beq
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

Sometimes predict taken?



"Cache" of Target Addresses (BTB: Branch Target Buffer)

Branch Direction Prediction

- "Branch direction" refers to whether the branch was taken or not
- Two methods for predicting direction:
  - Static - We predict once during compilation, and that prediction never changes
  - Dynamic - We predict (potentially) many times during execution, and the prediction may change over time

- *Static vs dynamic strategies are a very common topic in computer architecture*

Branch Direction Prediction (Static)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the “not-taken” path

- Always taken
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC

- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

Branch Direction Prediction (Dynamic)

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    TTTTTTTTTTNNNNNNNNNN → 90% accuracy

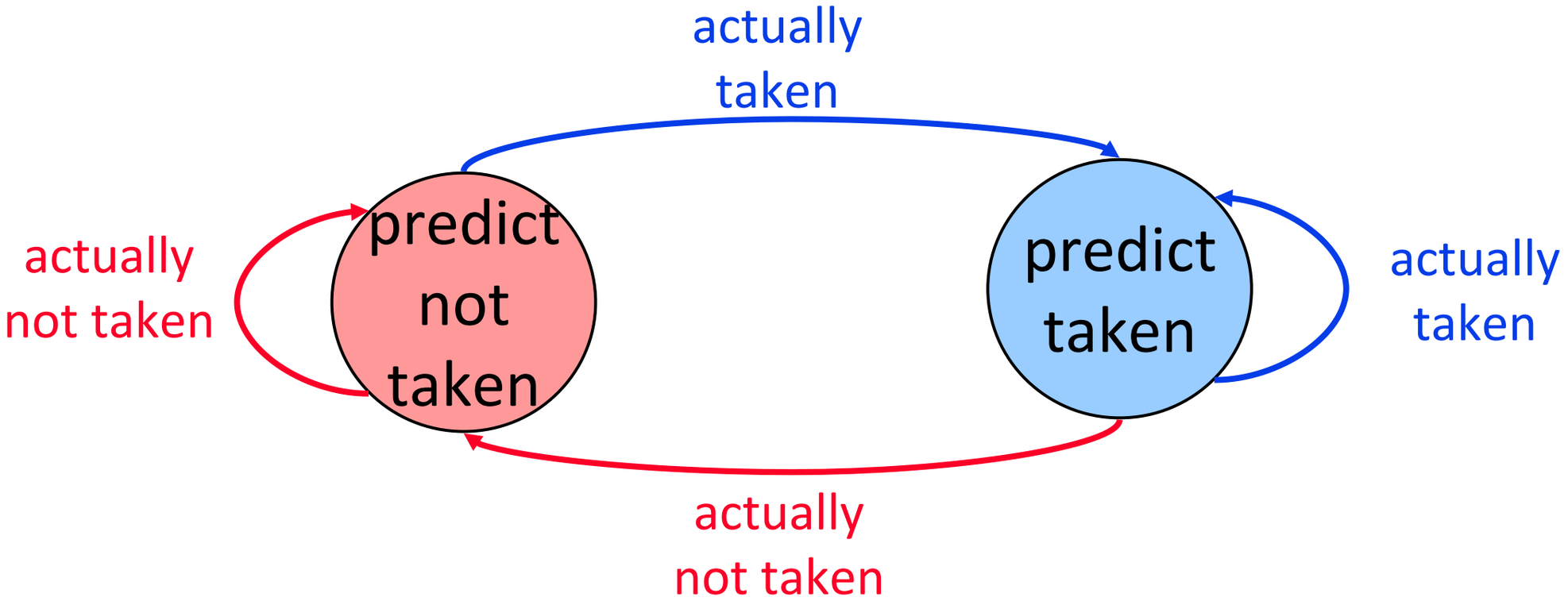
- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large number of iterations

-- Loop branches for loops will small number of iterations

TNTNTNTNTNTNTNTNTNTN → 0% accuracy

State Machine for Last-Time Prediction

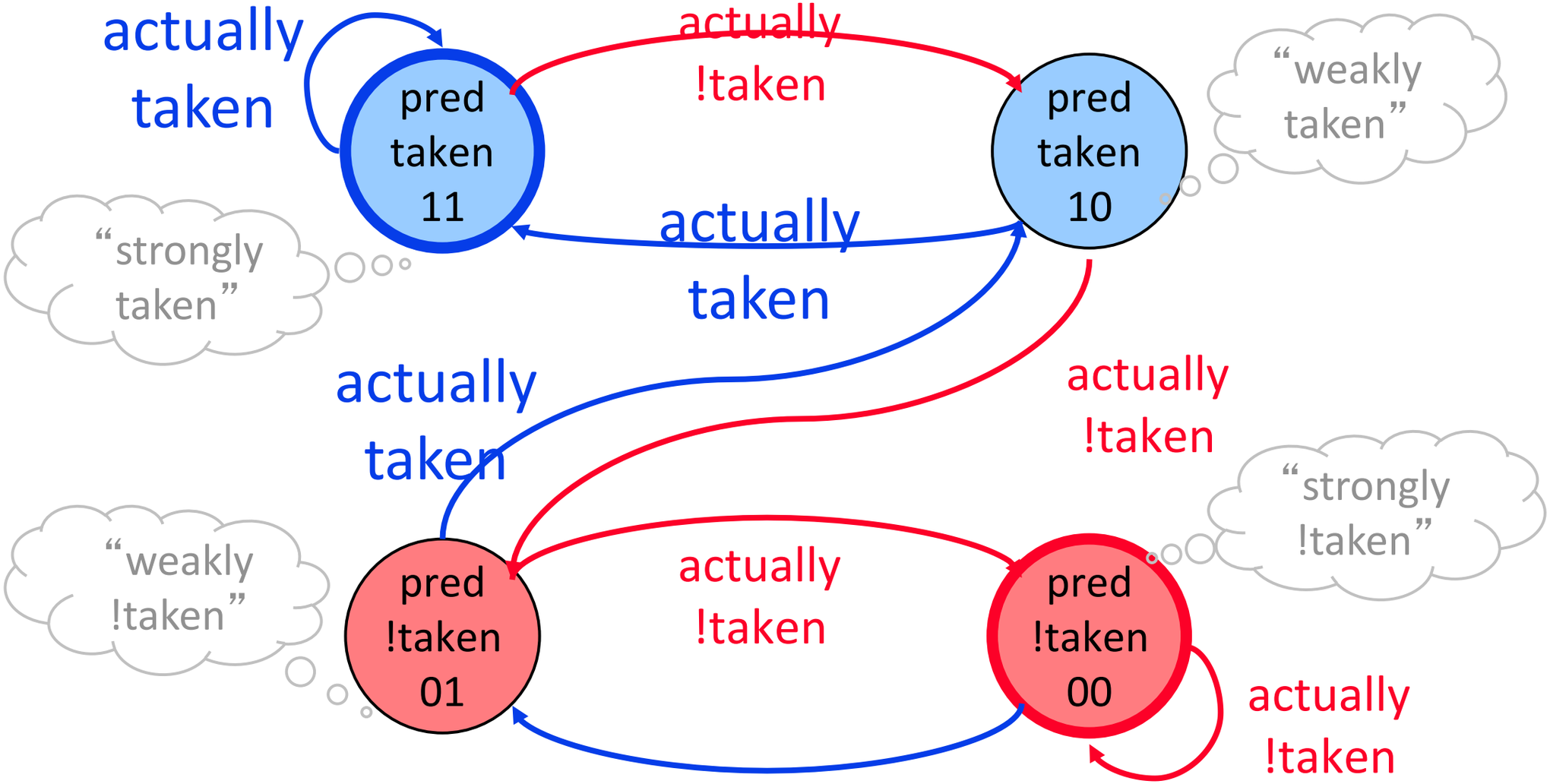


Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - Even though the branch may be mostly taken or mostly not taken

- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each

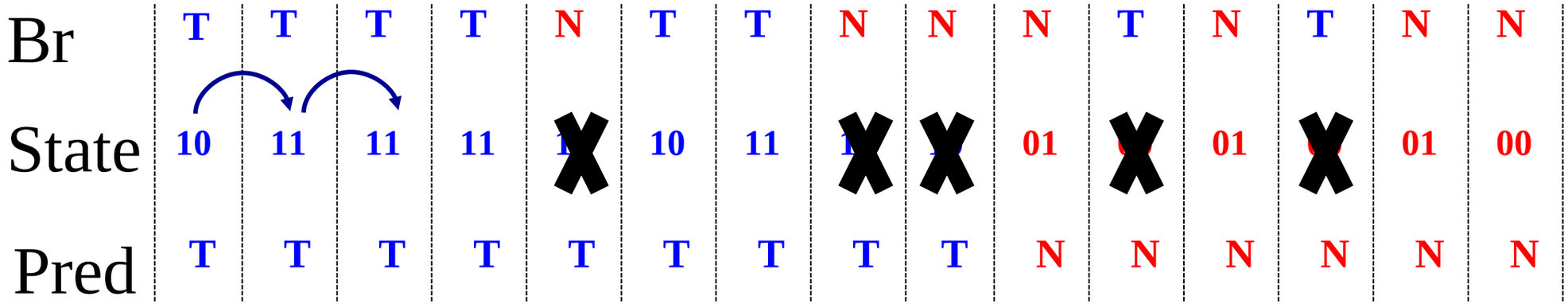
State Machine for 2-bit Saturating Counter



Two-Bit Counter Based Prediction

Poll: How many branches do we get wrong?

- What's the prediction accuracy of a branch with the following sequence of taken/not taken outcomes:
  - T T T T N T T N N N T N T N N



Can We Do Better?

- Absolutely… take 470
  - Tons of sophisticated branch predictor designs

- I've worked on a few that found their way into some Chromebooks!

Remember this Example from Lecture 1?

- We know understand why sorting improves the inner-loop so much
  - The branch predictor is better at guessing what's gonna happen when data is sorted!

```
for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;
std::sort(data, data + arraySize);

// Test
clock_t start = clock();
long long sum = 0;
// Primary loop
for (unsigned c = 0; c < arraySize; ++c)
{
    if (data[c] >= 128)
        sum += data[c];
}

double elapsedTime =
  static_cast<double>(clock() - start);
```

Branch Prediction

- Predict not taken: ~50% accurate
- Predict backward taken: ~65% accurate
- Predict same as last time: ~80% accurate

- Realistic designs: ~96% accurate

Next time

- More on Pipeline Peformance
  - What if we change number of stages?
- Into to caches
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah