EECS 470

Finish Tomasulo

Branches:
Address prediction and recovery

Lecture 5 – Winter 2024

Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch of Carnegie Mellon University, Purdue University, University of Michigan, University of Pennsylvania, and University of Wisconsin.

Announcements:

- Programming assignment #2
  - Due Tuesday 1/30
- HW #2
  - Due Friday 2/22
- Reading
  - Book: 3.1, 3.3-3.6, 3.8
  - **Combining Branch Predictors**, S. McFarling, WRL Technical Note TN-36, June 1993.
    - On the website.

Last time:

- Started in on Tomasulo’s algorithm.

Today

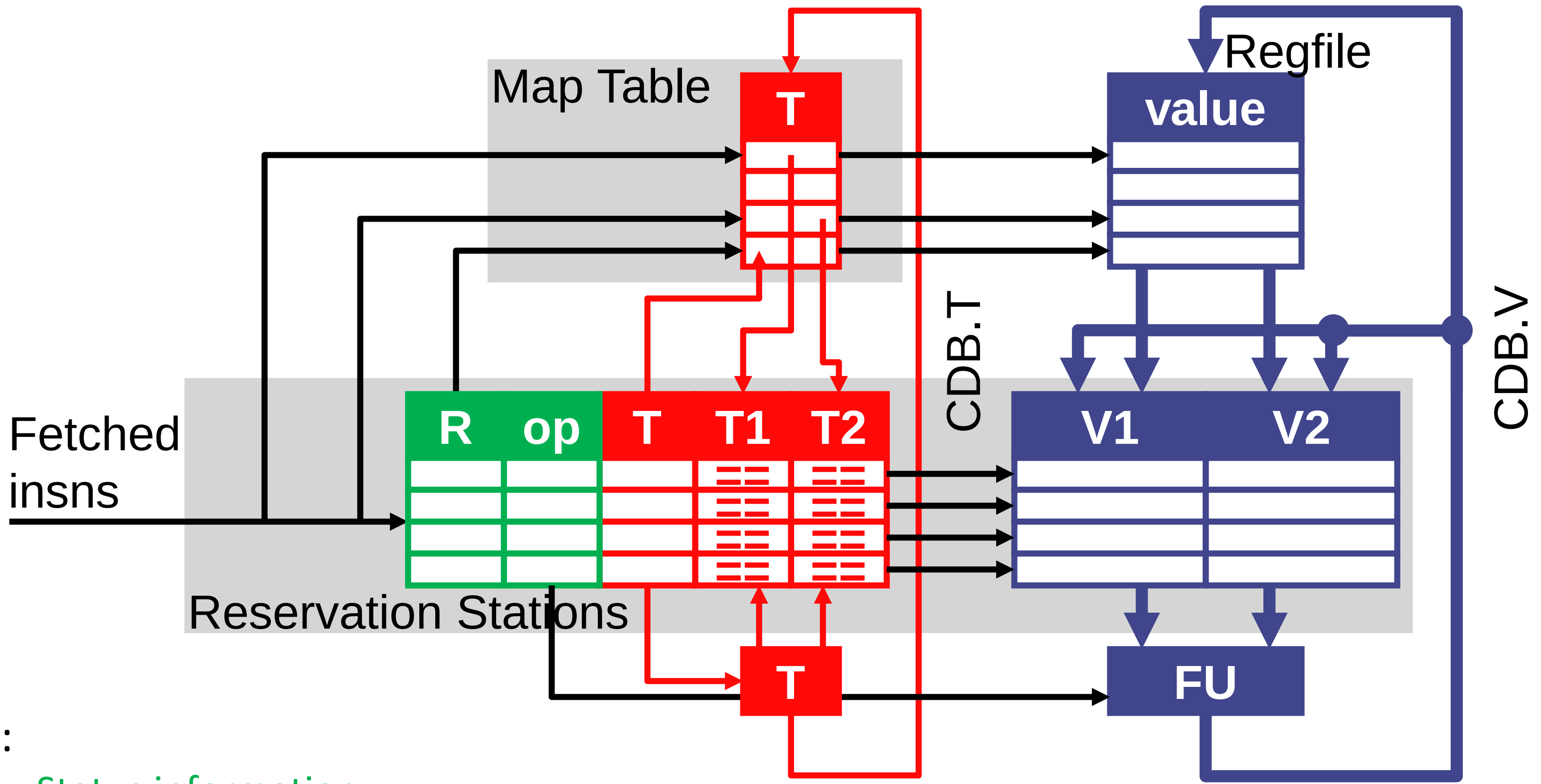
- Some deep thoughts on Tomasulo’s
- Branch prediction consists of
  - Branch taken predictor
  - Address predictor
  - Mispredict recovery.
- Also interrupts become relevant
  - “Recovery” is fairly similar…

But first…
Brehob’s Verilog ***rules****

- Always blocks
  - In always_comb blocks, use *blocking* assignments.
  - In always_ff blocks, use *non-blocking* assignments.
    - Do not mix blocking and non-blocking assignments in the same always block.
  - Do not make assignments to the same variable in more than one always block.
  - Make sure that all paths through a combinational always block assign all variables.
  - Sync. resets.
- Generic:
  - Correctly use logical and bitwise operators.
  - Avoid extra text that confuses
    - X=A?1’b1:1’b0; //big ick.
    - if(X==1’b1)   //ick but sometimes okay.

* Most of these are style/clarity issues. Some are required. In the real world there are good reasons to break nearly all of these. But in 470, please follow them. When we grade for style we’ll be looking at these issues (among other things!)

Simple Tomasulo Data Structures



- RS:
  - Status information
    - R: Destination Register
    - op: Operand (add, etc.)
  - Tags
    - T1, T2: source operand tags
  - Values
    - V1, V2: source operand values
- Map table (also RAT: Register Alias Table)
  - Maps registers to tags
- Regfile (also ARF: Architected Register File)
  - Holds value of register if no value in RS

Tomasulo Data Structures
(Timing Free Example)

CDB	
T	V

Map Table	
Reg	T
r0	
r1	
r2	
r3	
r4	

Reservation Stations								
T	FU	busy	R	op	T1	T2	V1	V2
1								
2								
3								
4								
5								

ARF	
Reg	V
r0	
r1	
r2	
r3	
r4	

Instruction
r0=r1*r2
r1=r2*r3
r2=r4+1
r1=r1+r1

Tomasulo Data Structures
(Timing Free Example #2)

CDB	
T	V

Map Table	
Reg	Tag
r0	
r1	
r2	
r3	
r4	

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1								
2								
3								
4								
5								

ARF	
Reg	V
r0	
r1	
r2	
r3	
r4	

Instruction
r0=r1*r2
r1=r0*r3
r0=r4+1
r1=r1+r0

Note:

- Tuesday’s slides had a detailed example with timing.
  - Just for reference, some of you may find it useful.

Can We Add Superscalar?

- Dynamic scheduling and multiple issue are orthogonal
  - E.g., Pentium4: dynamically scheduled 5-way superscalar
  - Two dimensions
    - **N**: superscalar width (number of parallel operations)
    - **W**: window size (number of reservation stations)

- What do we need for an **N**-by-**W** Tomasulo?
  - RS: **N** tag/value w-ports (D), **N** value r-ports (S), **2N** tag CAMs (W)
  - Select logic: **W**→**N** priority encoder (S)
  - MT: **2N** r-ports (D), **N** w-ports (D)
  - RF: **2N** r-ports (D), **N** w-ports (W)
  - CDB: **N** (W)
  - Which are the expensive pieces?

Superscalar Select Logic

- Superscalar select logic: W→N priority encoder
  - – Somewhat complicated (N^2 logW)
  - Can simplify using different RS designs
- **Split design**
  - Divide RS into N banks: 1 per FU?
  - Implement N separate W/N→1 encoders
  - + Simpler: N * logW/N
  - – Less scheduling flexibility
- **FIFO design** [Palacharla+]
  - Can issue only head of each RS bank
  - + Simpler: no select logic at all
  - – Less scheduling flexibility (but surprisingly not that bad)

Dynamic Scheduling Summary

- Dynamic scheduling: out-of-order execution
  - Higher pipeline/FU utilization, improved performance
  - Easier and more effective in hardware than software
    - + More storage locations than architectural registers
    - + Dynamic handling of cache misses
- Instruction buffer: multiple F/D latches
  - Implements large scheduling scope + “passing” functionality
  - Split decode into in-order dispatch and out-of-order issue
    - Stall vs. wait
- Dynamic scheduling algorithms
  - Scoreboard: no register renaming, limited out-of-order
  - Tomasulo: copy-based register renaming, full out-of-order

Are we done?

- When can Tomasulo go wrong?
  - Branches
    - What if a branch finishes after older instructions (things behind the branch) finish?
  - Exceptions!!
    - No way to figure out relative order of instructions in RS

And… a bit of terminology

- Issue can be thought of as a two-stage process: “wakeup” and “select”.
  - When the RS figures out it has it’s data and is ready to run it is said to have “woken up” and the process of doing so is called ***wakeup***
    - But there may be a structural hazard—no EX unit available for a given RS
      - When?
  - Thus, in addition to be woken up, and RS needs to be selected before it can go to the execute unit (EX stage).
    - This process is called ***select***

Questions

- What are we “renaming” to?
- Why are branches a challenge?
  - What are my options on how to handle them?
- What are some other names for the map table?
- Could you explain when to update the RAT again?
  - Why?

Branch mispredict

- In this original version of Tomasulo's algorithm, branches are a big problem.
  - Unless we don't speculate past branches, we allow instructions to speculatively modify architectural state.
    - That's a really bad idea—we have no recovery mechanism.
    - Think about the 5-state pipeline.
  - Tomasulo's answer was to not let instructions be dispatched until all branches in front of them have resolved.
    - Branches are about ~15% (1 in 7 or so) of all instructions.
      - That really limits us.
- Let's first discuss how to predict.

Parts of the predictor

- Direction Predictor
  - For conditional branches
    - Predicts whether the branch will be taken
  - Examples:
    - Always taken; backwards taken
- Address Predictor
  - Predicts the target address (use if predicted taken)
  - Examples:
    - BTB; Return Address Stack; Precomputed Branch
- Recovery logic

Example gzip:

- gzip: loop branch A@ 0x1200098d8

- Executed: 1359575 times
- Taken: 1359565 times
- Not-taken: 10 times
- % time taken: 99% - 100%

Easy to predict (direction and address)

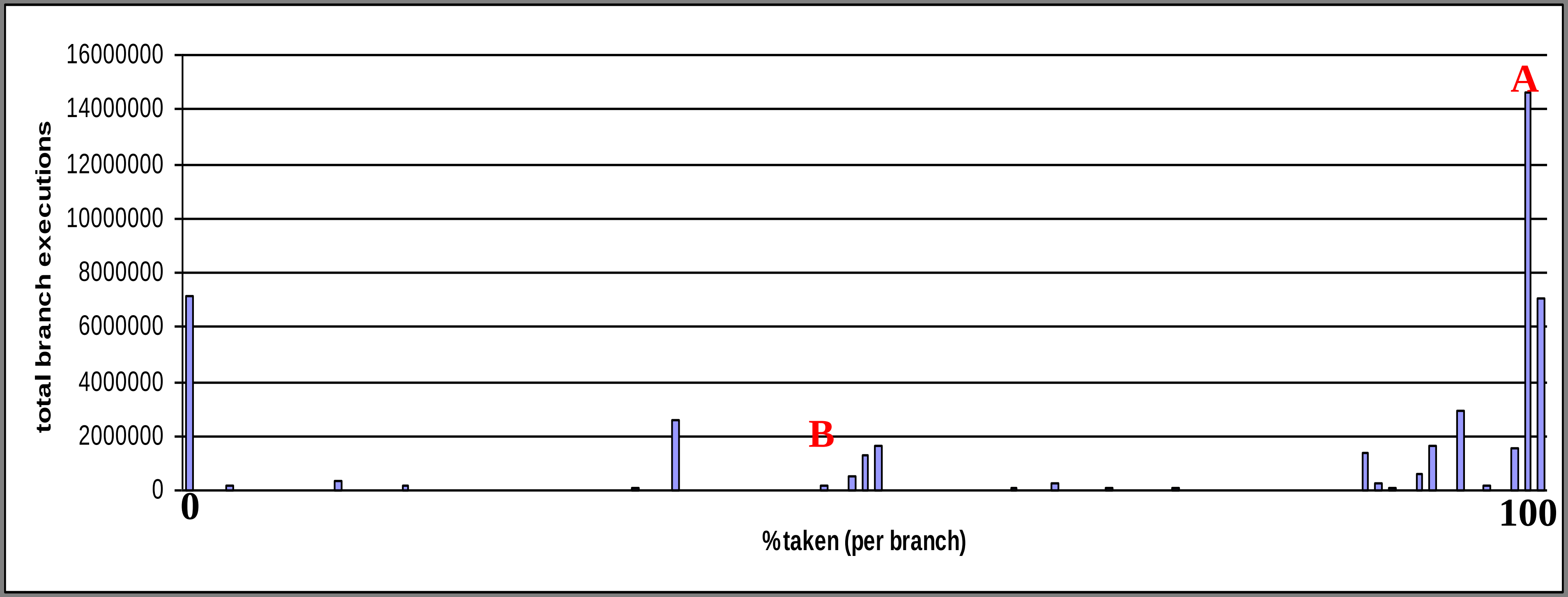
Example gzip:

- gzip: if branch B@ 0x12000fa04

- Executed: 151409 times
- Taken: 71480 times
- Not-taken: 79929 times
- % time taken: ~49%

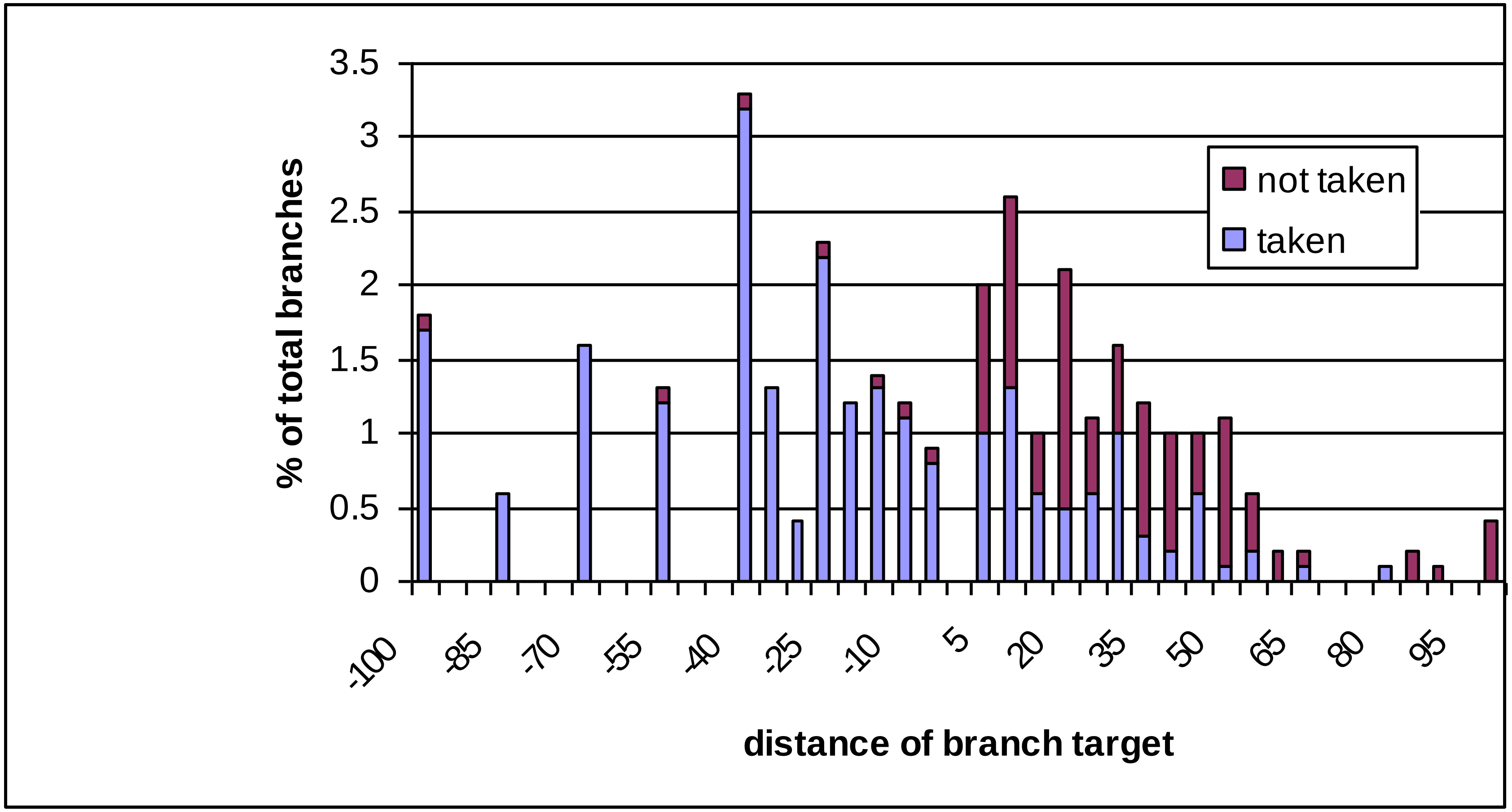
Easy to predict? (maybe not/ maybe dynamically)

Example: gzip



Direction prediction: always taken
Accuracy: ~73 %

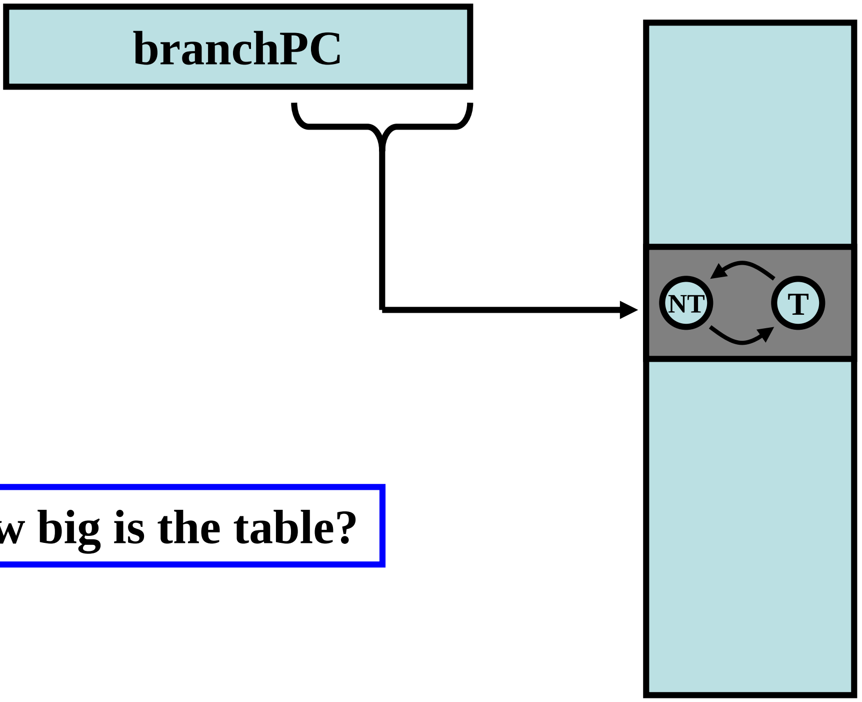
Branch Backwards



Most backward branches are heavily TAKEN
Forward branches slightly more likely to be NOT-TAKEN

Using history

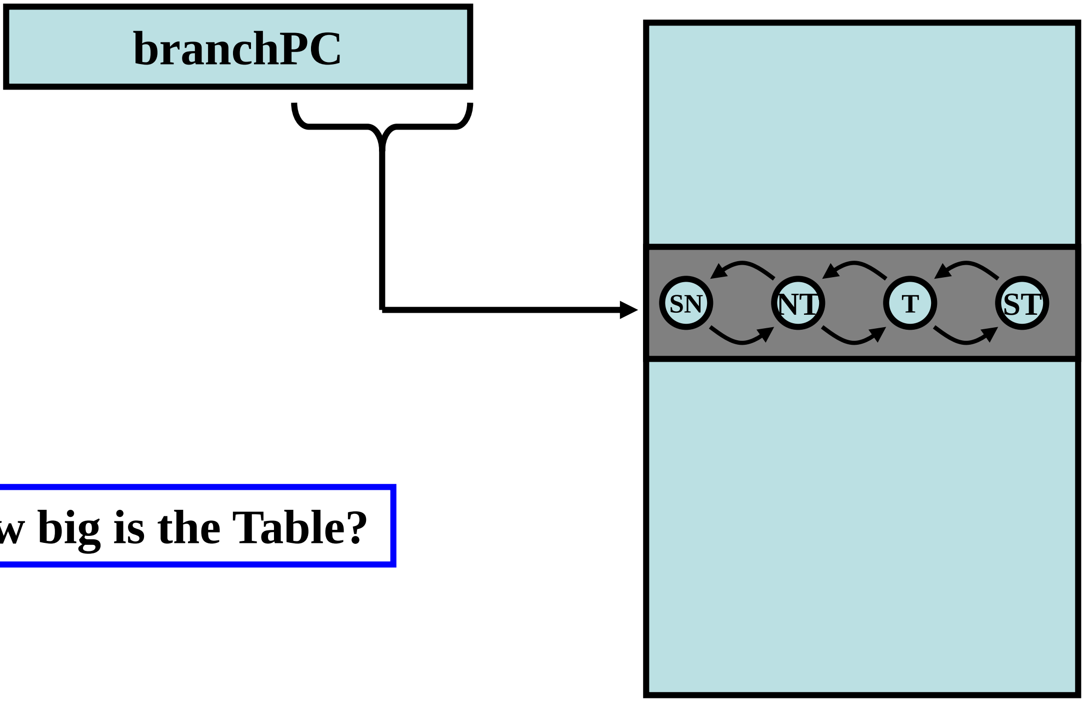
- 1-bit history (direction predictor)
  - Remember the last direction for a branch



How big is the table?

Using history

- 2-bit history (direction predictor)



How big is the Table?

Using History Patterns

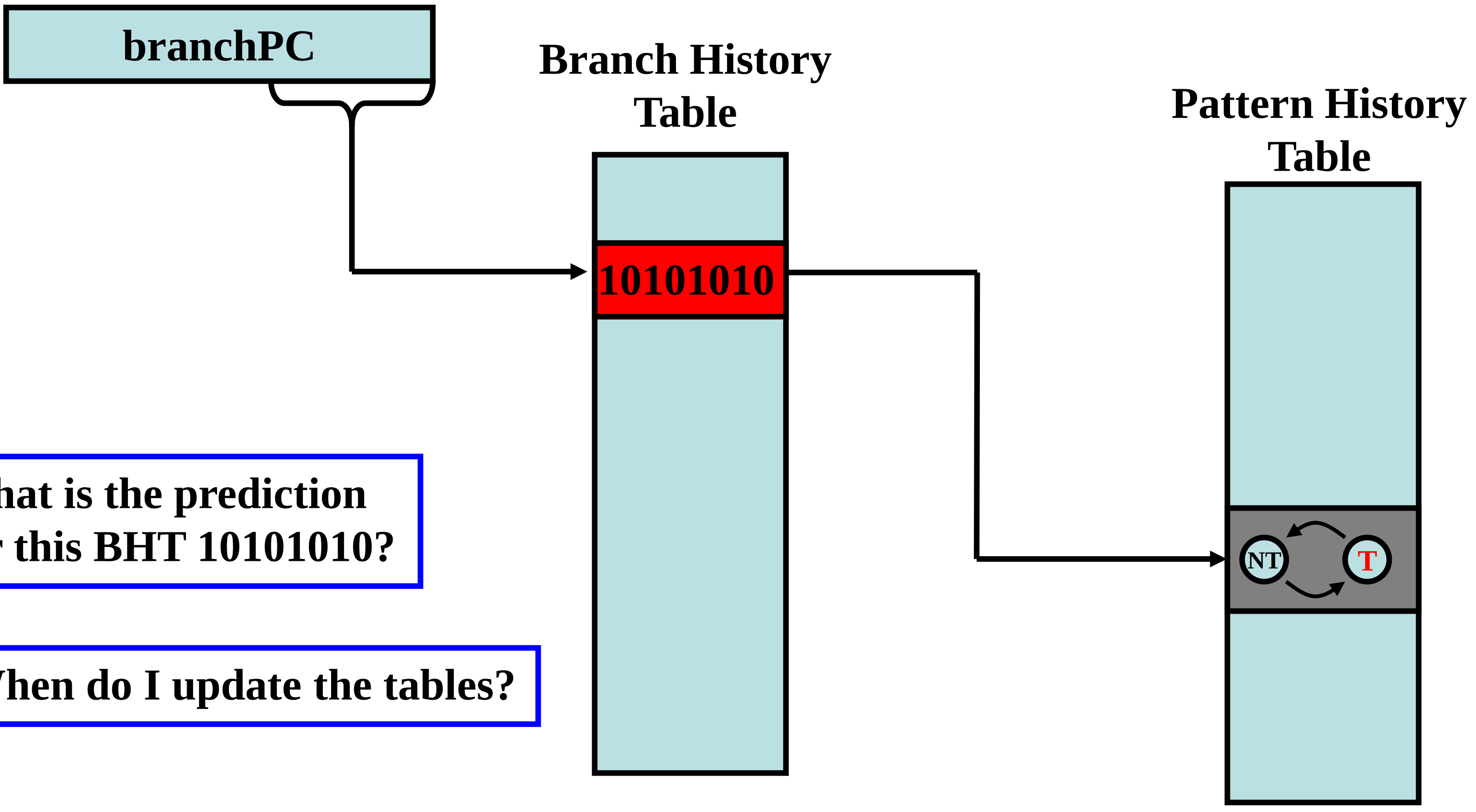
~80 percent of branches are either heavily TAKEN or heavily NOT-TAKEN

For the other 20%, we need to look a patterns of reference to see if they are predictable using a more complex predictor

Example: gcc has a branch that flips each time

T(1) NT(0)   1010101010101010101010101010101010101010

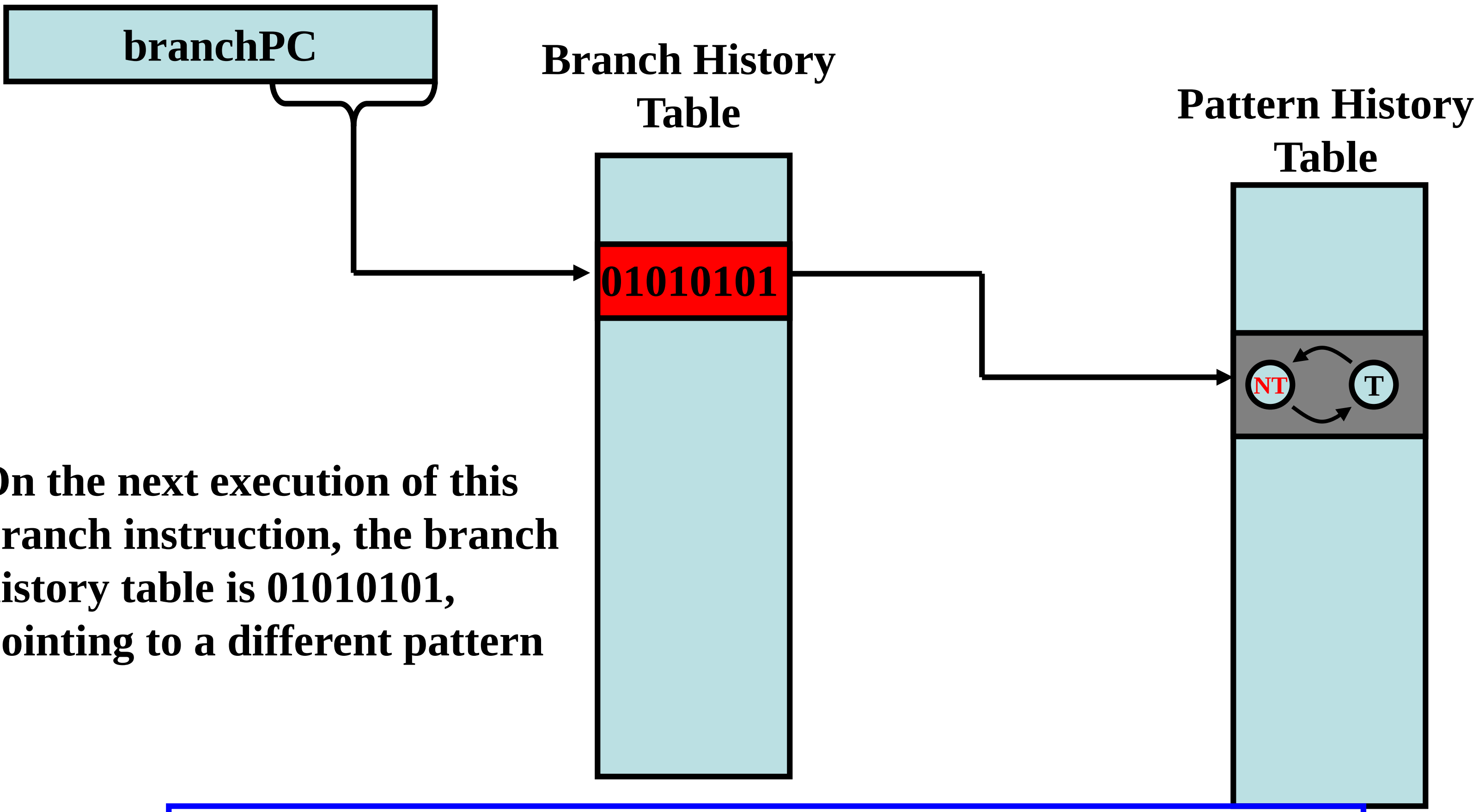
Local history



What is the prediction for this BHT 10101010?

When do I update the tables?

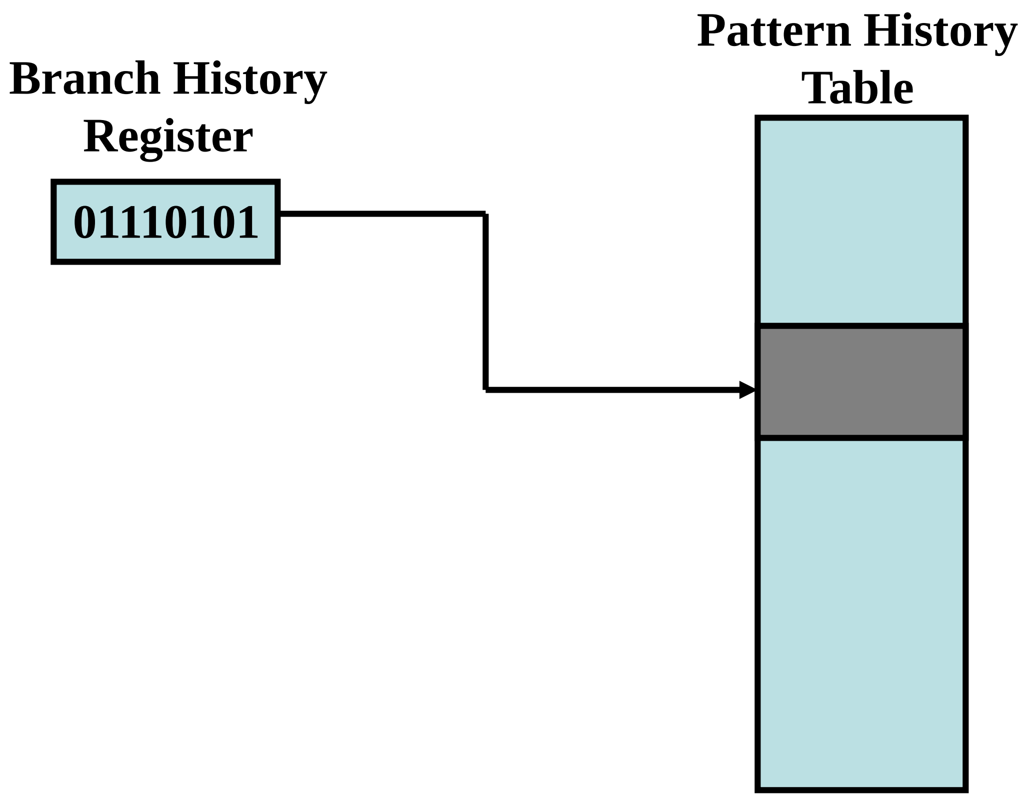
Local history



On the next execution of this branch instruction, the branch history table is 01010101, pointing to a different pattern

What is the accuracy of a flip/flop branch 0101010101010…?

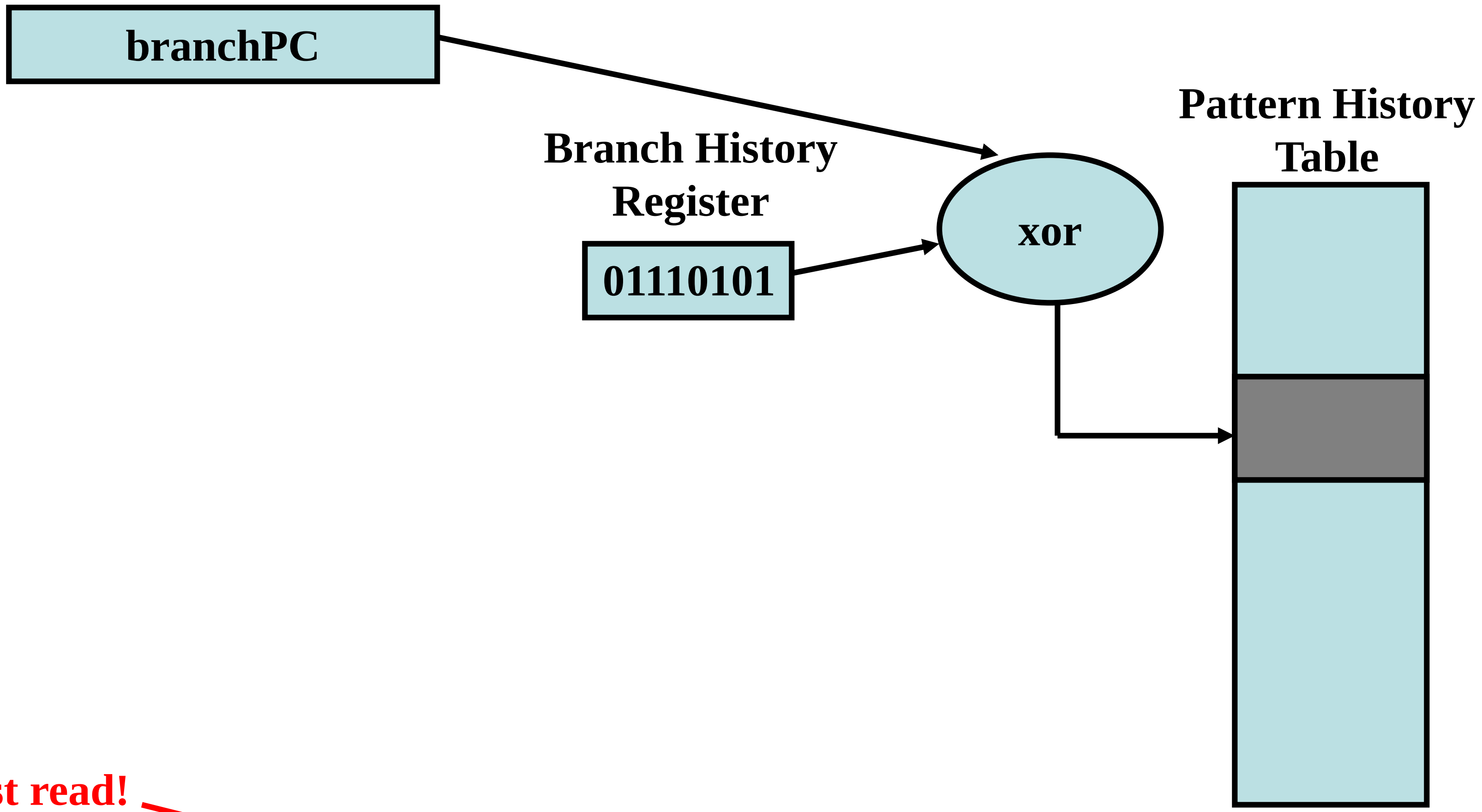
Global history



```
if (aa == 2)
        aa = 0;
if (bb == 2)
        bb = 0;
if (aa != bb) { …
```

How can branches interfere with each other?

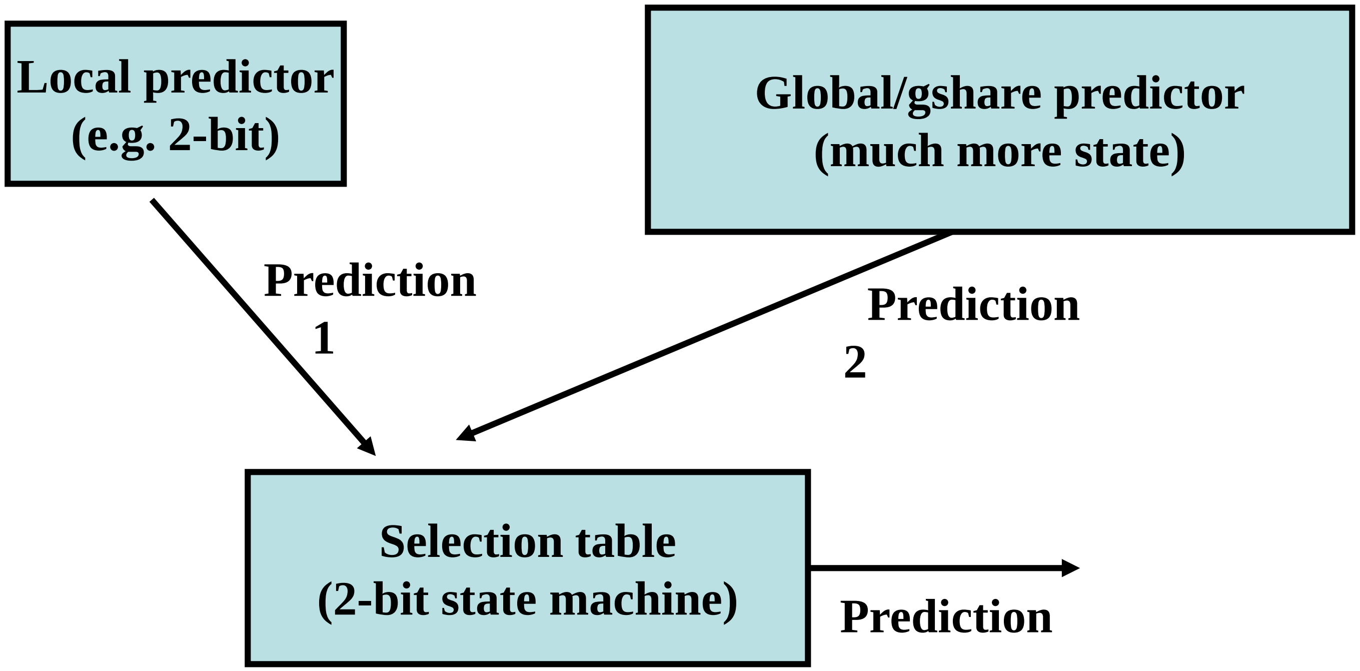
Gshare predictor



Must read!

Ref: Combining Branch Predictors

Hybrid predictors



How do you select which predictor to use?
How do you update the various predictor/selector?

Overriding Predictors

- Big predictors are slow, but more accurate
- Use a single cycle predictor in fetch
- Start the multi-cycle predictor
  - When it completes, compare it to the fast prediction.
    - If same, do nothing
    - If different, assume the slow predictor is right and flush pipline.
- Advantage: reduced branch penalty for those branches mispredicted by the fast predictor and correctly predicted by the slow predictor

“Trivial” example: Tournament Branch Predictor

- Local
  - 8-entry 3-bit local history table indexed by PC
  - 8-entry 2-bit up/down counter indexed by local history
- Global
  - 8-entry 2-bit up/down counter indexed by global history
- Tournament
  - 8-entry 2-bit up/down counter indexed by PC

Local predictor 1st level table (BHT) 0=NT, 1=T

ADR[4:2]	*History*
0	001
1	101
2	100
3	110
4	110
5	001
6	111
7	101

Local predictor 2nd level table (PHT) 00=NT, 11=T

History	*Pred. state*
0	00
1	11
2	10
3	00
4	01
5	01
6	11
7	11

Tournament selector 00=local, 11=global

ADR[4:2]	*Pred. state*
0	00
1	01
2	00
3	10
4	11
5	00
6	11
7	10

Branch History Register

Global predictor table 00=NT, 11=T

History	*Pred. state*
0	11
1	10
2	00
3	00
4	00
5	11
6	11
7	00

Tournament selector 00=local, 11=global	
ADR[4:2]	*Pred. state*
0	00
1	01
2	00
3	10
4	11
5	00
6	11
7	10

Local predictor 1st level table (BHT) 0=NT, 1=T	
ADR[4:2]	*History*
0	001
1	101
2	100
3	110
4	110
5	001
6	111
7	101

Local predictor 2nd level table (PHT) 00=NT, 11=T	
History	*Pred. state*
0	00
1	11
2	10
3	00
4	01
5	01
6	11
7	11

Global predictor table 00=NT, 11=T	
History	*Pred. state*
0	11
1	10
2	00
3	00
4	00
5	11
6	11
7	00

- r1=2, r2=6, r3=10, r4=12, r5=4
- Address of joe =0x100 and each instruction is 4 bytes.
- Branch History Register = 110

```
joe:     add r1 r2 r3
         beq r3 r4 next
         bgt r2 r3  skip // if r2>r3 branch
         lw r6 4(r5)
         add r6 r8 r8
skip:    add r5 r2 r2
         bne r4 r5 joe
next:    noop
```

General speculation

- Control speculation
  - “I think this branch will go to address 90004”
- Data speculation
  - “I’ll guess the result of the load will be zero”
- Memory conflict speculation
  - “I don’t think this load conflicts with any proceeding store.”
- Error speculation
  - “I don’t think there were any errors in this calculation”

Speculation in general

- Need to be 100% sure on final correctness!
  - So need a recovery mechanism
  - Must make forward progress!
- Want to speed up overall performance
  - So recovery cost should be low or ***expected*** rate of occurrence should be low.
  - There can be a real trade-off on *accuracy*, *cost of recovery*, and *speedup when correct.*
- Should keep the worst case in mind…

Address Prediction

BTB

(Chapter3.5)

- Branch Target Buffer
  - Addresses predictor
  - Lots of variations
- Keep the target of “likely taken” branches in a buffer
  - With each branch, associate the expected target.

- BTB indexed by current PC
  - If entry is in BTB fetch target address next
- Generally set associative (too slow as FA)
- Often qualified by branch taken predictor

Branch PC	Target address
0x05360AF0	0x05360000
…	…
…	…
…	…
…	…
…	…

So…

- BTB lets you predict target address during the ***fetch*** of the branch!
- If BTB gets a miss, pretty much stuck with not-taken as a prediction
  - So limits prediction accuracy.
- Can use BTB as a predictor.
  - If it is there, predict taken.
- Replacement is an issue
  - LRU seems reasonable, but only really want branches that are taken at least a fair amount.

Branch Recovery

Pipeline recovery is pretty simple

- Squash and restart fetch with right address
  - Just have to be sure that nothing has “committed” its state yet.
- In our 5-stage pipe, state is only committed during MEM (for stores) and WB (for registers)

Tomasulo’s

- Recovery seems really hard
  - What if instructions after the branch finish after we find that the branch was wrong?
    - This could happen. Imagine

      ```
      R1=MEM[R2+0]
      BEQ R1, R3 DONE ← Predicted not taken
      R4=R5+R6
      ```
  - So we have to not speculate on branches or not let anything pass a branch
    - Which is really the same thing.
    - Branches become serializing instructions.
      - Note that can be executing some things before and after the branch once branch resolves.

What we need is:

- Some way to not commit instructions until all branches before it are committed.
  - Just like in the pipeline, something could have finished execution, but not updated anything “real” yet.

Interrupt!!!

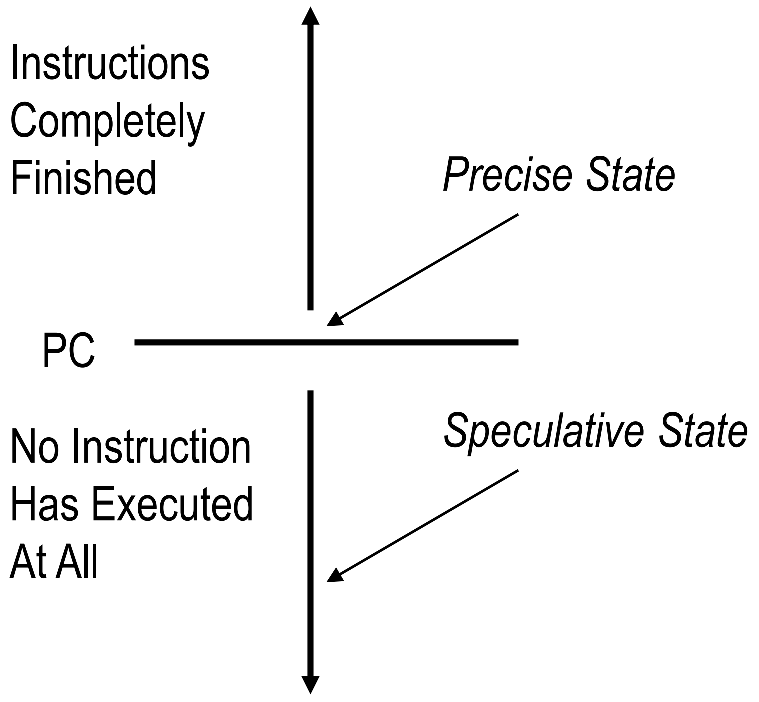
Interrupts

- These have a similar problem.
  - If we can execute out-of-order a “slower” instruction might not generate an interrupt until an instruction in front of it has finished.
- This sounds like the end of out-of-order execution
  - I mean, if we can’t finish out-of-order, isn’t this pointless?

Exceptions and Interrupts

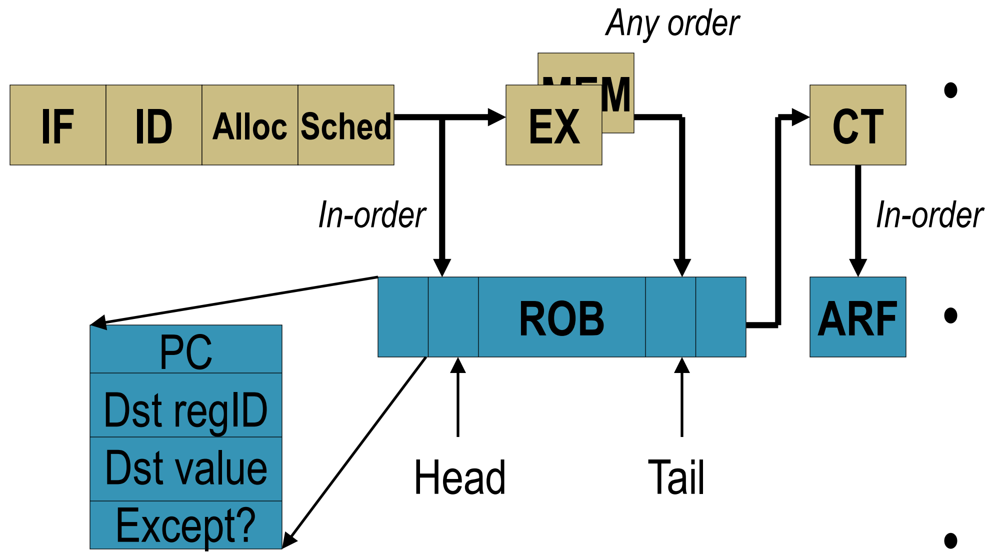
Exception Type	Sync/Async	Maskable?	Restartable?
I/O request	Async	Yes	Yes
System call	Sync	No	Yes
Breakpoint	Sync	Yes	Yes
Overflow	Sync	Yes	Yes
Page fault	Sync	No	Yes
Misaligned access	Sync	No	Yes
Memory Protect	Sync	No	Yes
Machine Check	Async/Sync	No	No
Power failure	Async	No	No

Precise Interrupts


Instructions Completely Finished
Precise State
PC
Speculative State
No Instruction Has Executed At All

- Implementation approaches
  - Don't
    - E.g., Cray-1
  - Buffer speculative results
    - E.g., P4, Alpha 21264
    - History buffer
    - Future file/Reorder buffer

Precise Interrupts via the Reorder Buffer



- Reorder Buffer (ROB)
  - Circular queue of spec state
  - May contain multiple definitions of *same* register

- @ **Alloc**
  - Allocate result storage at Tail
- @ **Sched**
  - Get inputs (ROB T-to-H then ARF)
  - Wait until all inputs ready
- @ **WB**
  - Write results/fault to ROB
  - Indicate result is ready
- @ **CT**
  - Wait until inst @ Head is done
  - If fault, initiate handler
  - Else, write results to ARF
  - Deallocate entry from ROB

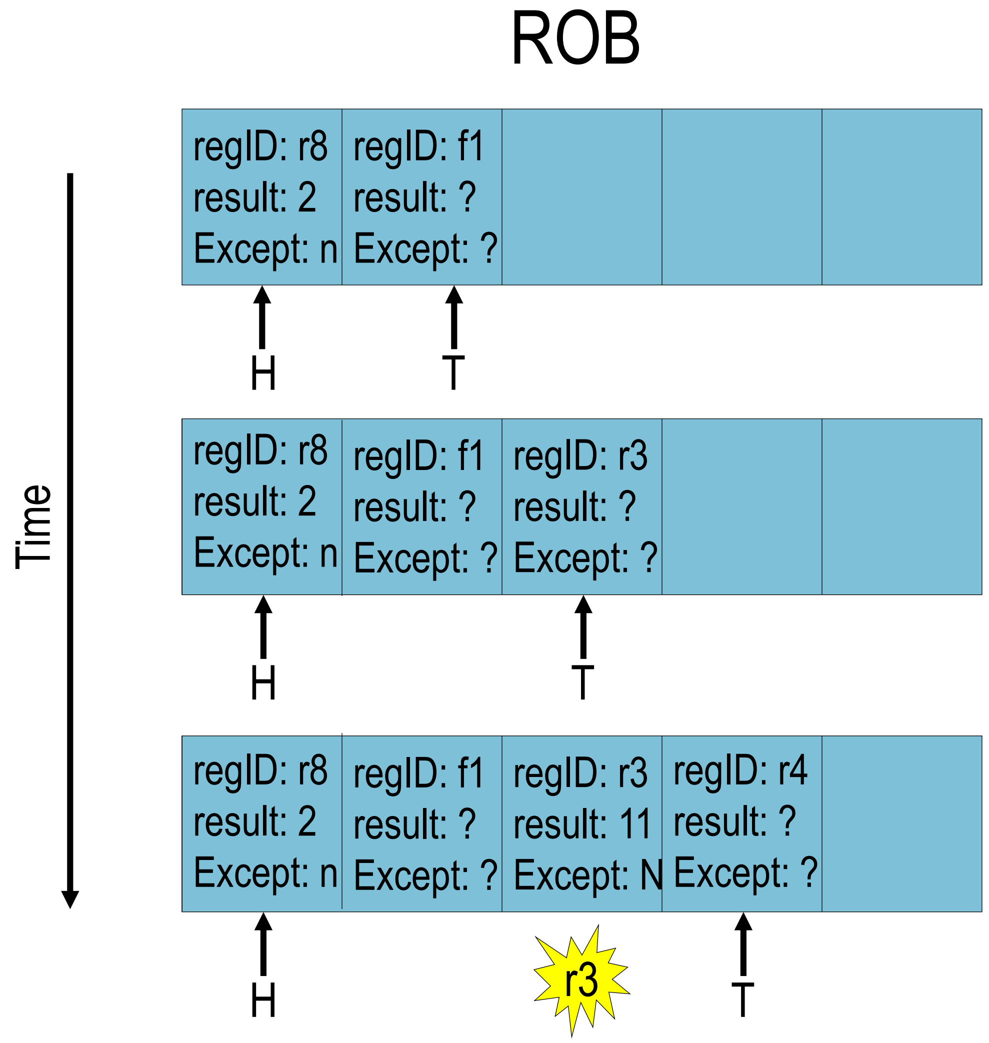
Reorder Buffer Example

Code Sequence

f1 = f2 / f3
r3 = r2 + r3
r4 = r3 – r2

Initial Conditions

- reorder buffer empty
- f2 = 3.0
- f3 = 2.0
- r2 = 6
- r3 = 5



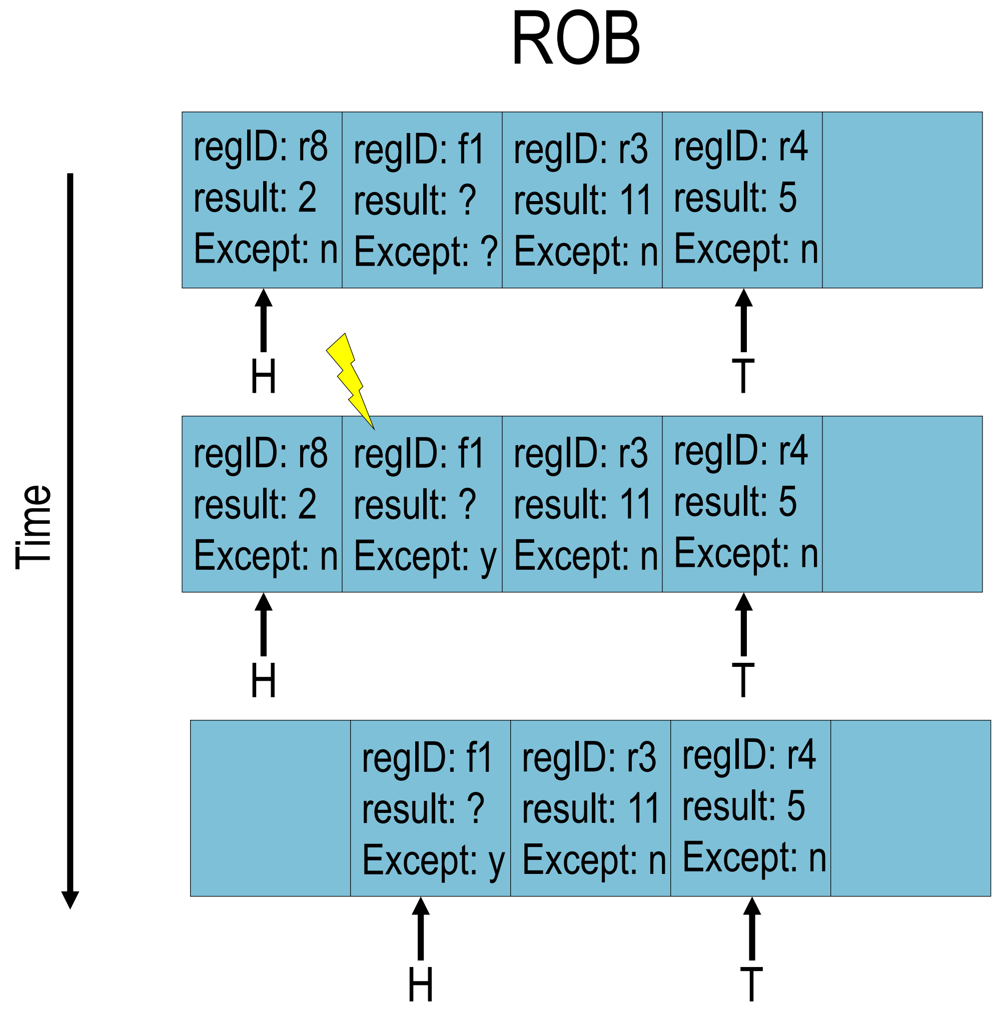
Reorder Buffer Example

<u>Code Sequence</u>

f1 = f2 / f3
r3 = r2 + r3
r4 = r3 – r2

Initial Conditions

- reorder buffer empty
- f2 = 3.0
- f3 = 2.0
- r2 = 6
- r3 = 5



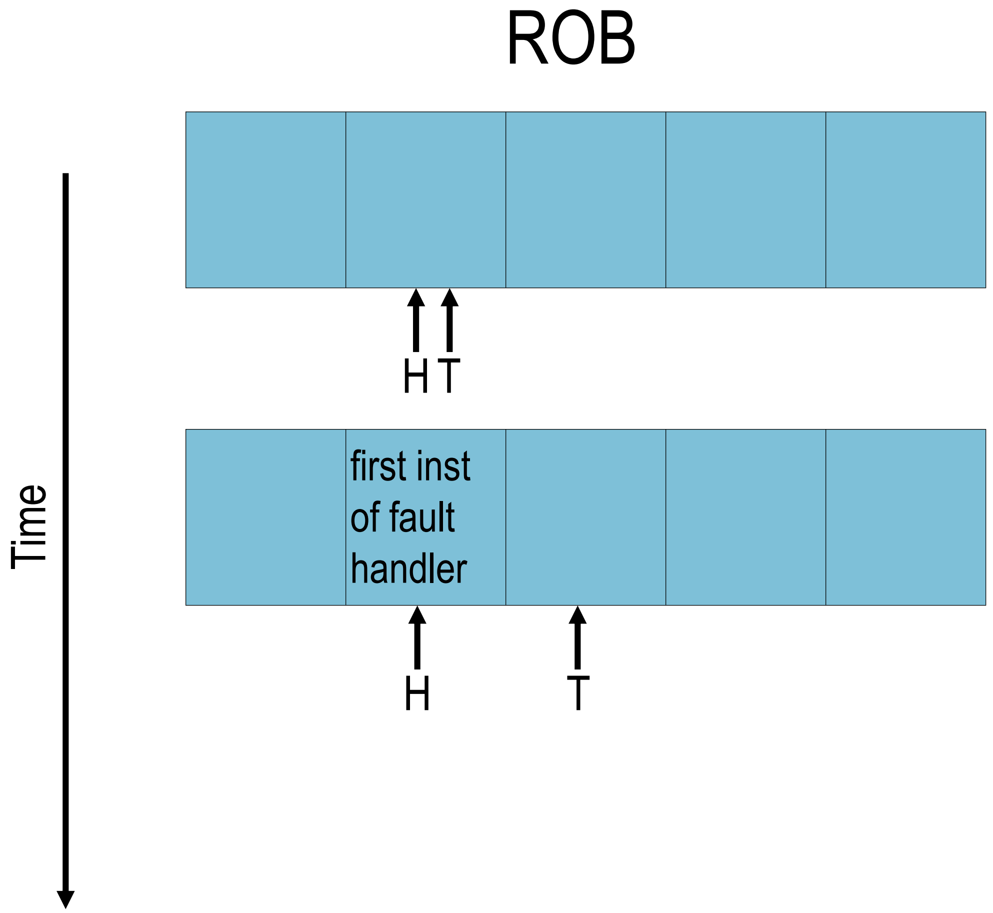
Reorder Buffer Example

Code Sequence

f1 = f2 / f3
r3 = r2 + r3
r4 = r3 – r2

Initial Conditions

- reorder buffer empty
- f2 = 3.0
- f3 = 2.0
- r2 = 6
- r3 = 5



There is more complexity here

- Rename table needs to be cleared
  - Everything is in the ARF
  - Really do need to finish everything which was before the faulting instruction in program order.
- What about branches?
  - Would need to drain everything before the branch.
    - Why not just squash everything that follows it?

And while we’re at it…

- Does the ROB replace the RS?
  - Is this a good thing? Bad thing?

ROB

- ROB
  - ROB is an *in-order* queue where instructions are placed.
  - Instructions *complete* (retire) in-order
  - Instructions still *execute* out-of-order
  - Still use RS
    - Instructions are issued to RS and ROB at the same time
    - Rename is to ROB entry, not RS.
    - When *execute* done instruction leaves RS
  - Only when all instructions in before it in program order are done does the instruction retire.

Review Questions

- Could we make this work without a RS?
  - If so, why do we use it?
- Why is it important to retire in order?
- Why must branches wait until retirement before they announce their mispredict?
  - Any other ways to do this?

Can We Add Superscalar?

- Dynamic scheduling and multiple issue are orthogonal
  - E.g., Pentium4: dynamically scheduled 5-way superscalar
  - Two dimensions
    - **N**: superscalar width (number of parallel operations)
    - **W**: (number of reservation stations)

- What do we need for an **N**-by-**W** Tomasulo?
  - RS: **N** tag/value w-ports (D), **N** value r-ports (S), **2N** tag CAMs (W)
  - Select logic: **W**$\rightarrow$**N** priority encoder (S)
  - MT: **2N** read-ports (D), **N** write-ports (D)
  - RF: **2N** read-ports (D), **N** write-ports (W)
  - CDB: **N** (W)
  - Which are the expensive pieces?

Superscalar Select Logic

- Superscalar select logic: W$\rightarrow$N priority encoder
  - Somewhat complicated (N^2 logW)
  - Can simplify using different RS designs
- **Split design**
  - Divide RS into N banks: 1 per FU?
  - Implement N separate W/N$\rightarrow$1 encoders
  - + Simpler: N * logW/N
  - Less scheduling flexibility
- **FIFO design** [Palacharla+]
  - Can issue only head of each RS bank
  - + Simpler: no select logic at all
  - Less scheduling flexibility (but surprisingly not that bad)