EECS 470

Branches:
Address prediction and recovery
(And interrupt recovery too.)

Lecture 6 – 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:

- Combining Branch Predictors, S. McFarling, WRL Technical Note TN-36, June 1993.
  - On the website
  - Part of HW2.
  - Expect a (likely brief) question on the midterm.

Last time:

- Started on branch predictors
  - Branch prediction consists of
    - Branch taken predictor
    - Address predictor
    - Mis-predict recovery.
  - Discussed direction predictors and started looking at some variations

Today

- Predictors
  - Bimodal
  - Local history predictor
  - Global history predictor
  - Gshare
  - Tournament
    - Do detailed example
- Start on the “P6 scheme”

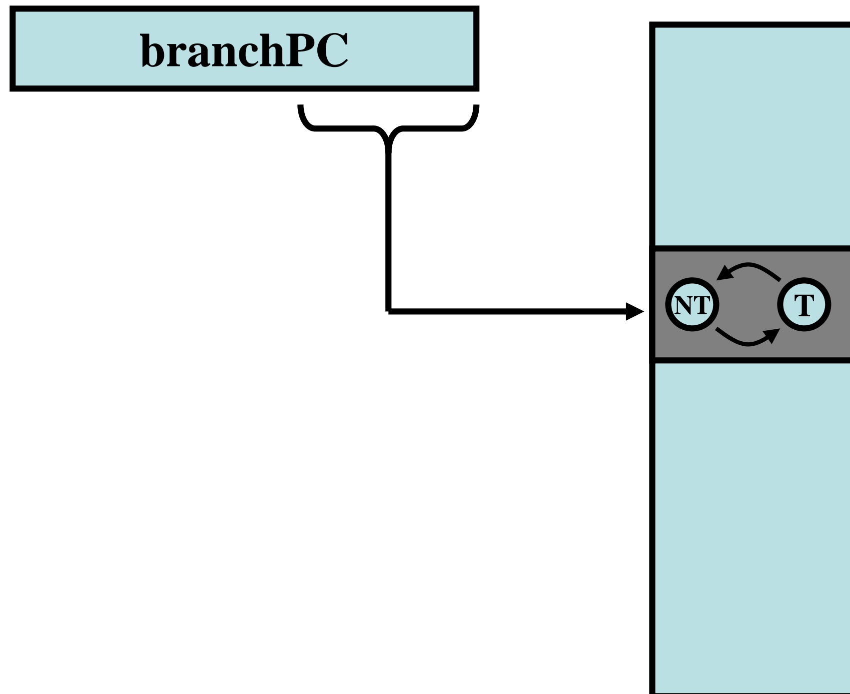
What are the limitations of Tomasulo's Algorithm?

- Branches are a pain.

- Instructions that *might* throw an exception are a pain.

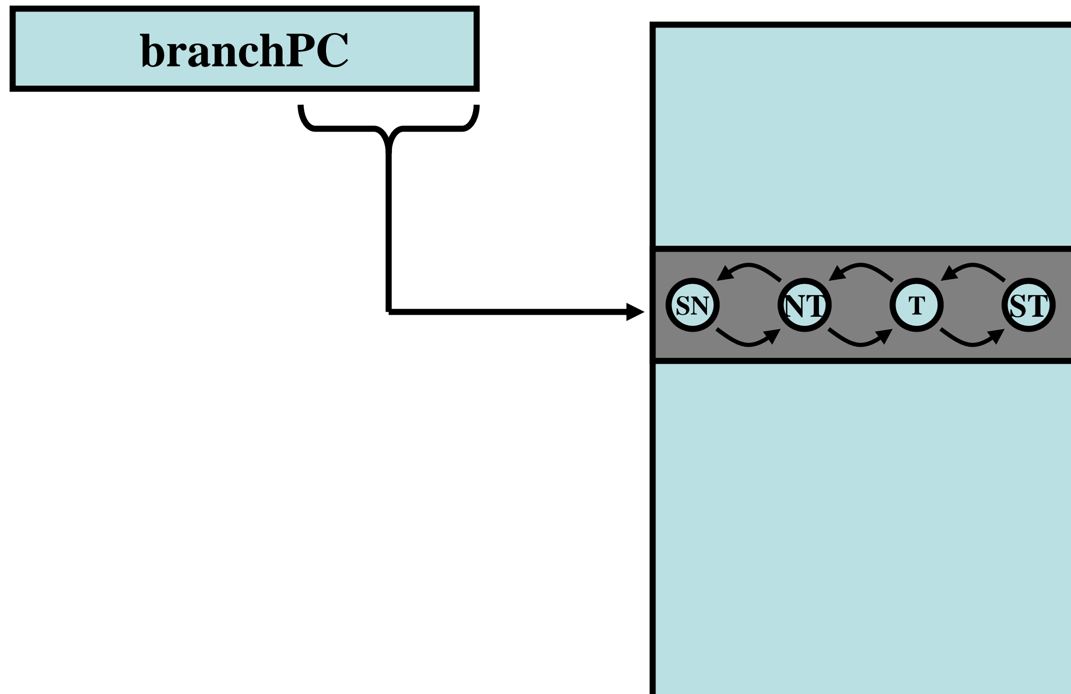
Using history—bimodal

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



Using history—bimodal

- 2-bit history (direction predictor)



Using History Patterns

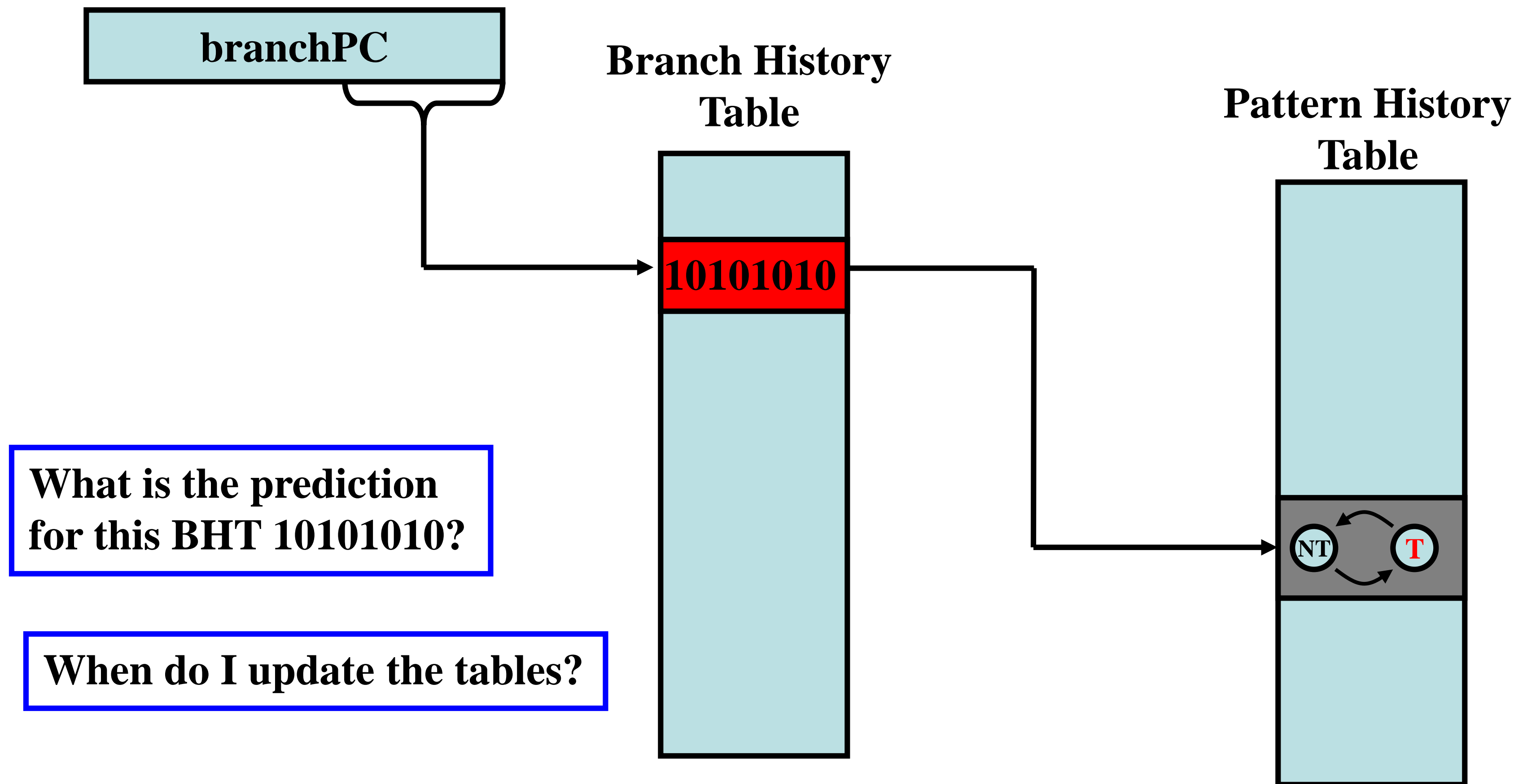
~80 percent of branches are either heavily TAKEN or heavily NOT-TAKEN

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

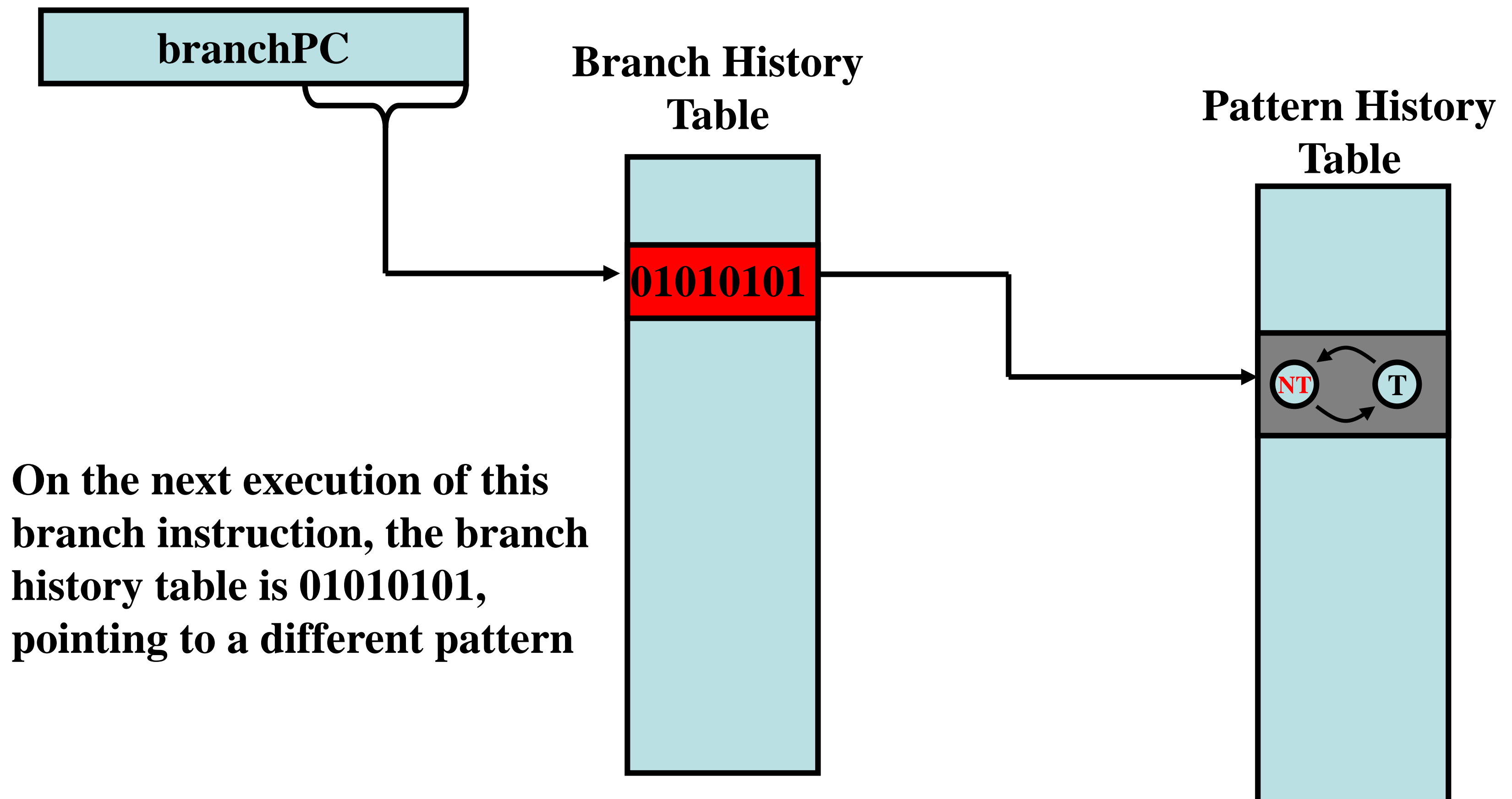
Example: gcc has a branch that flips each time

T(1) NT(0) 10101010101010101010101010101010101010

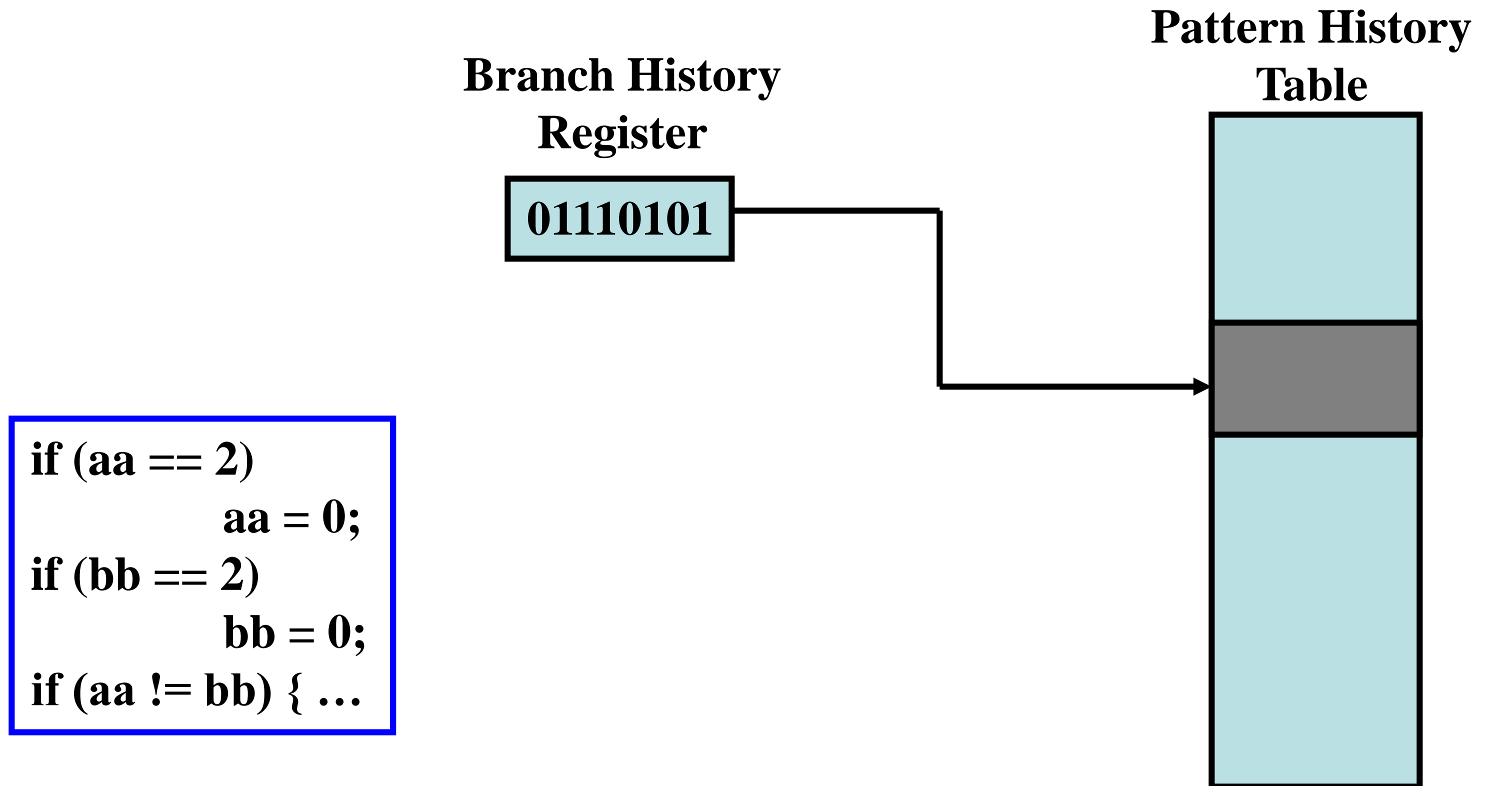
Local history



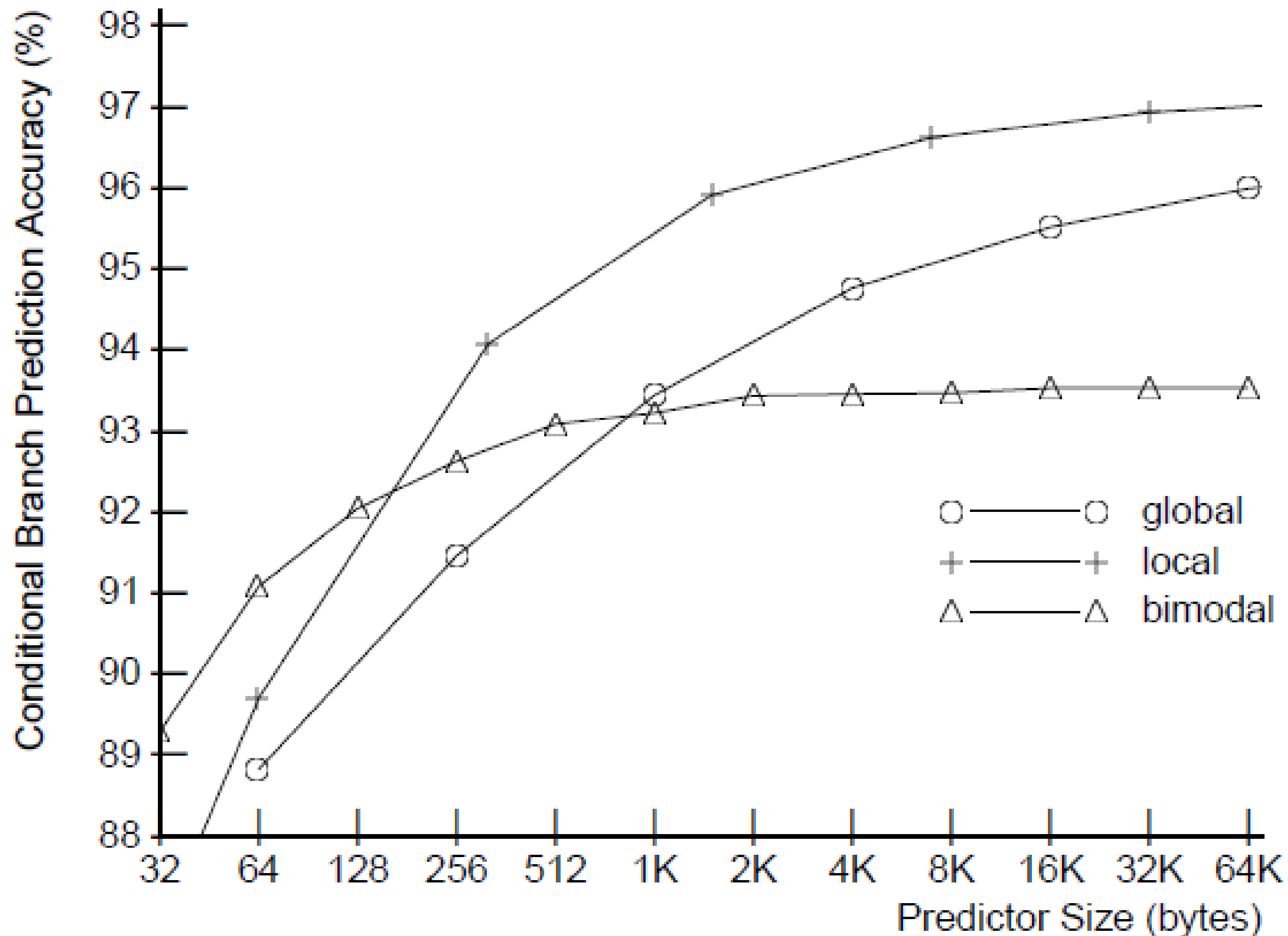
Local history



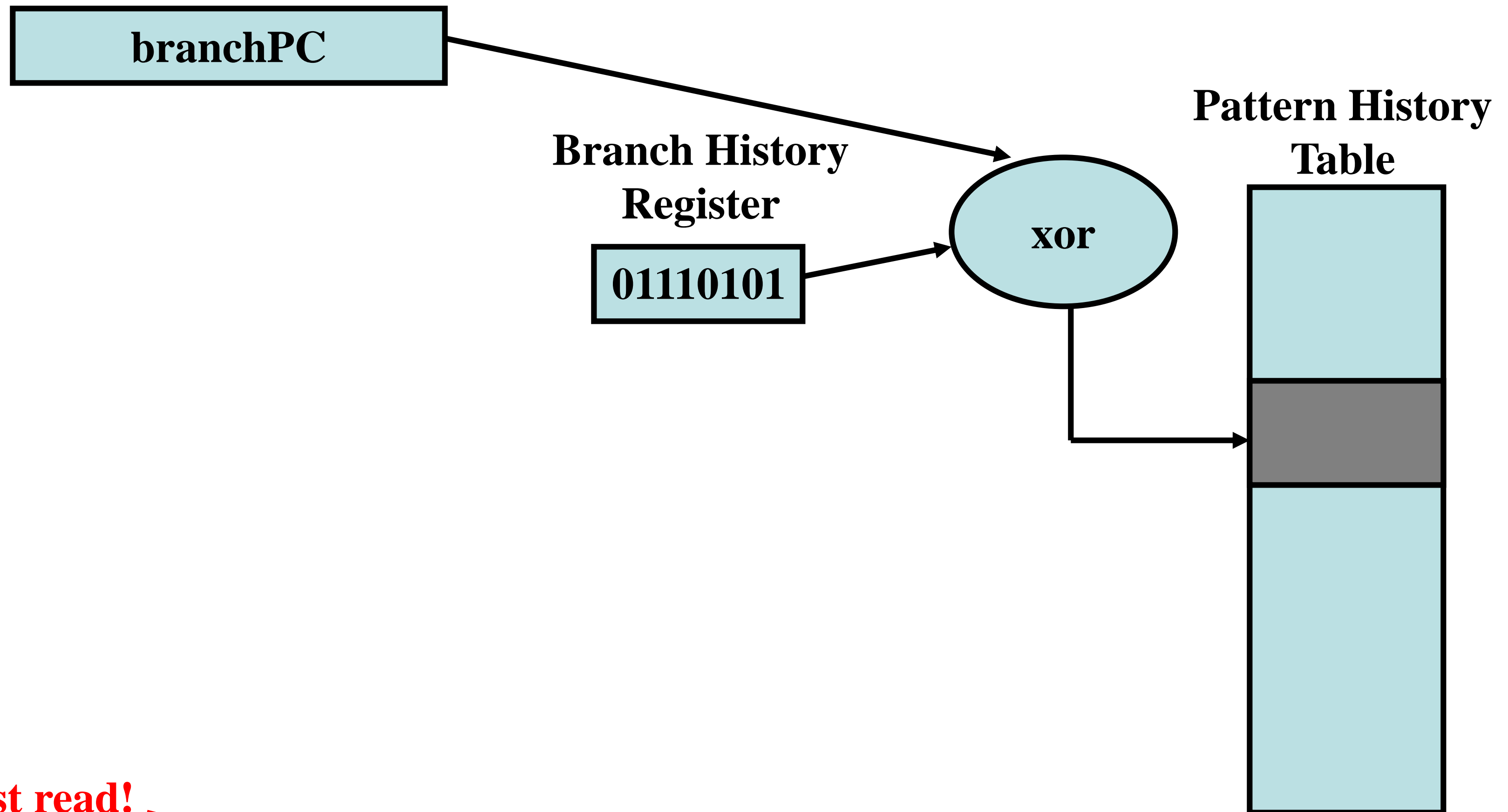
Global history



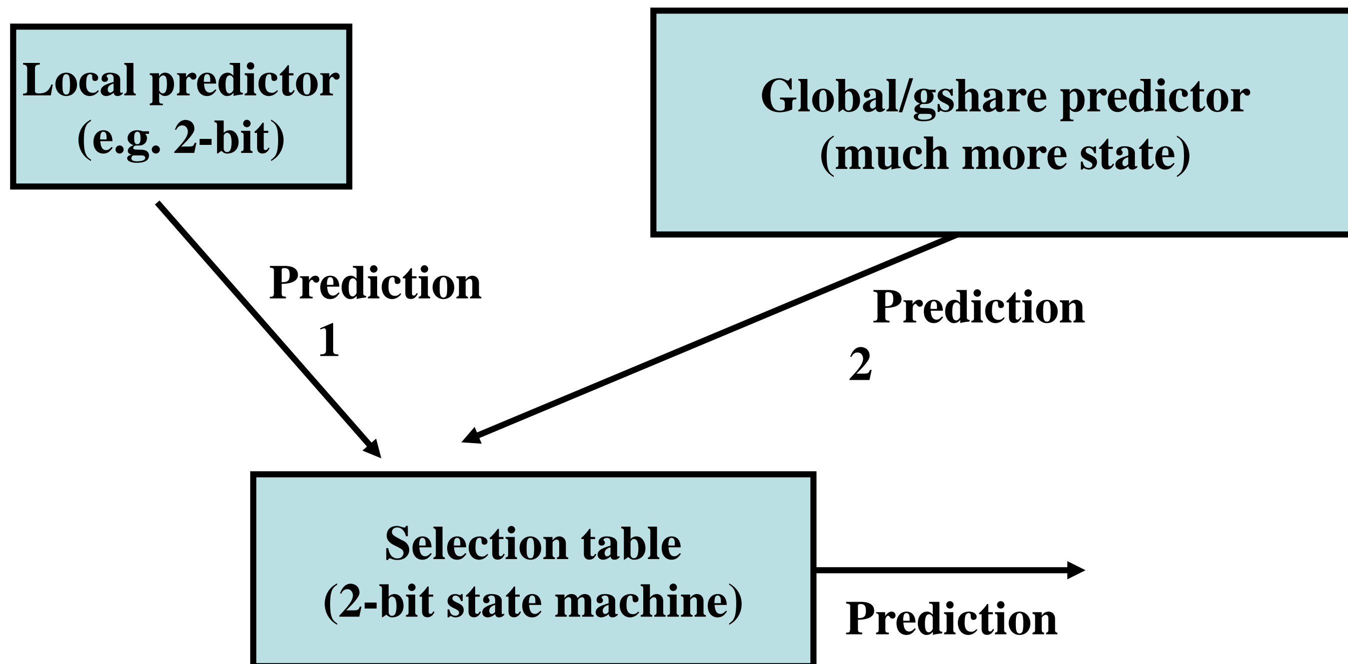
Relative performance (Spec '89)



Gshare predictor



Hybrid predictors

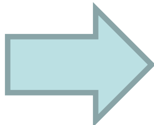


How do you select which predictor to use?
How do you update the various predictor/selector?

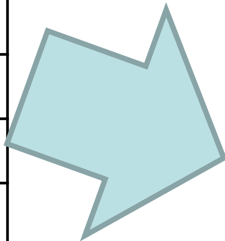
“Trivial” example: Tournament Branch Predictor

- 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

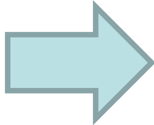


Local predictor 2nd level table (PHT) 00=NT, 11=T	
History	*Pred. state*
0	00
1	11
2	10
3	00
4	01
5	01
6	11
7	11

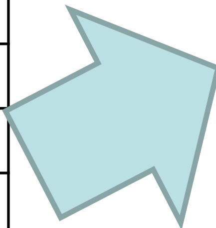


Tournament selector 00=local, 11=global	
ADR[4:2]	*Pred. state*
0	00
1	01
2	00
3	10
4	11
5	00
6	11
7	10

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
```

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

Address Prediction

BTB
(Chapter 3.9)

- 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.

- What branches will a BTB struggle with?
  - How to address that?

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!!!

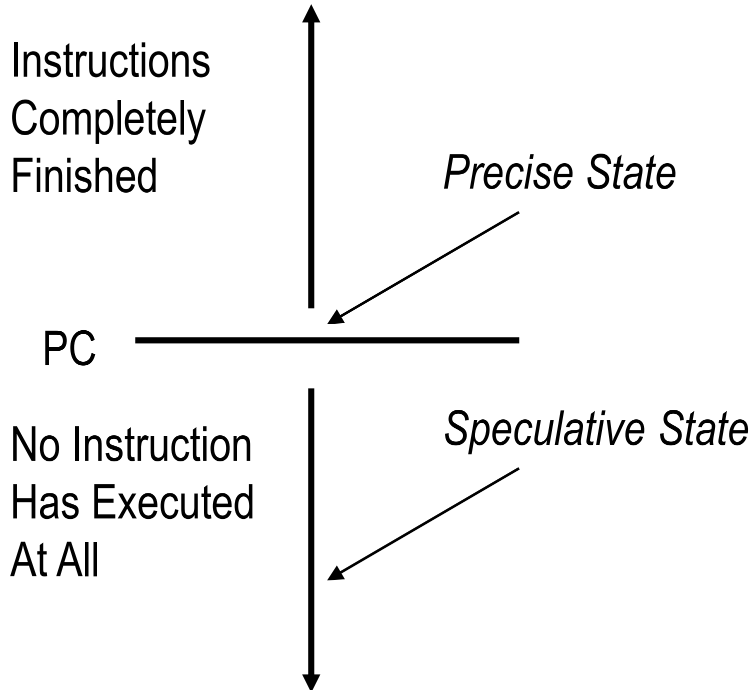
Interrupts

- 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

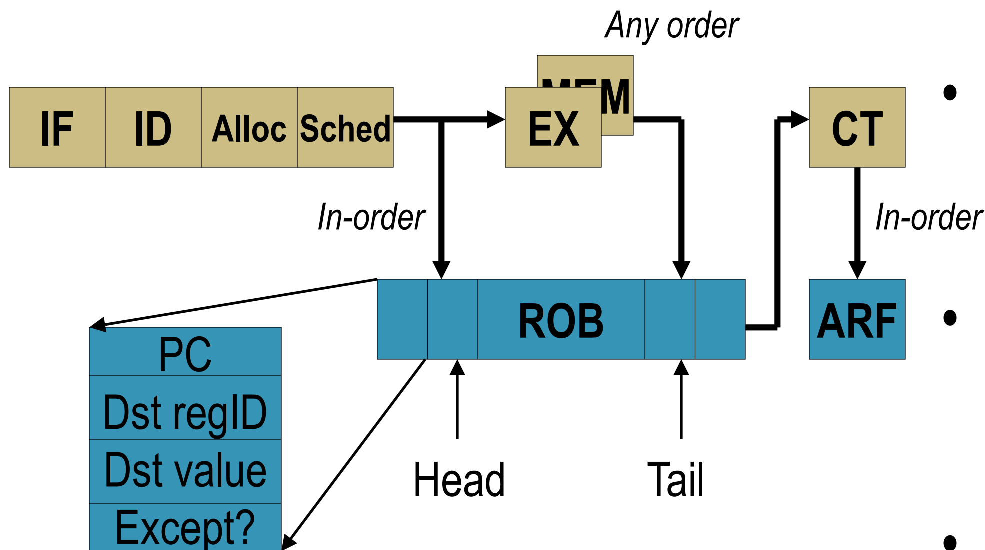
Exception Type	Sync/Async	Maskable?	Restartable?
I/O request	Async	Yes	Yes
System call	Sync	No	Yes
Breakpoint	Sync	Yes	Yes
Overflow	Sync	Yes	Yes
Page fault	Sync	No	Yes
Misaligned access	Sync	No	Yes
Memory Protect	Sync	No	Yes
Machine Check	Async/Sync	No	No
Power failure	Async	No	No

Precise Interrupts



- 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

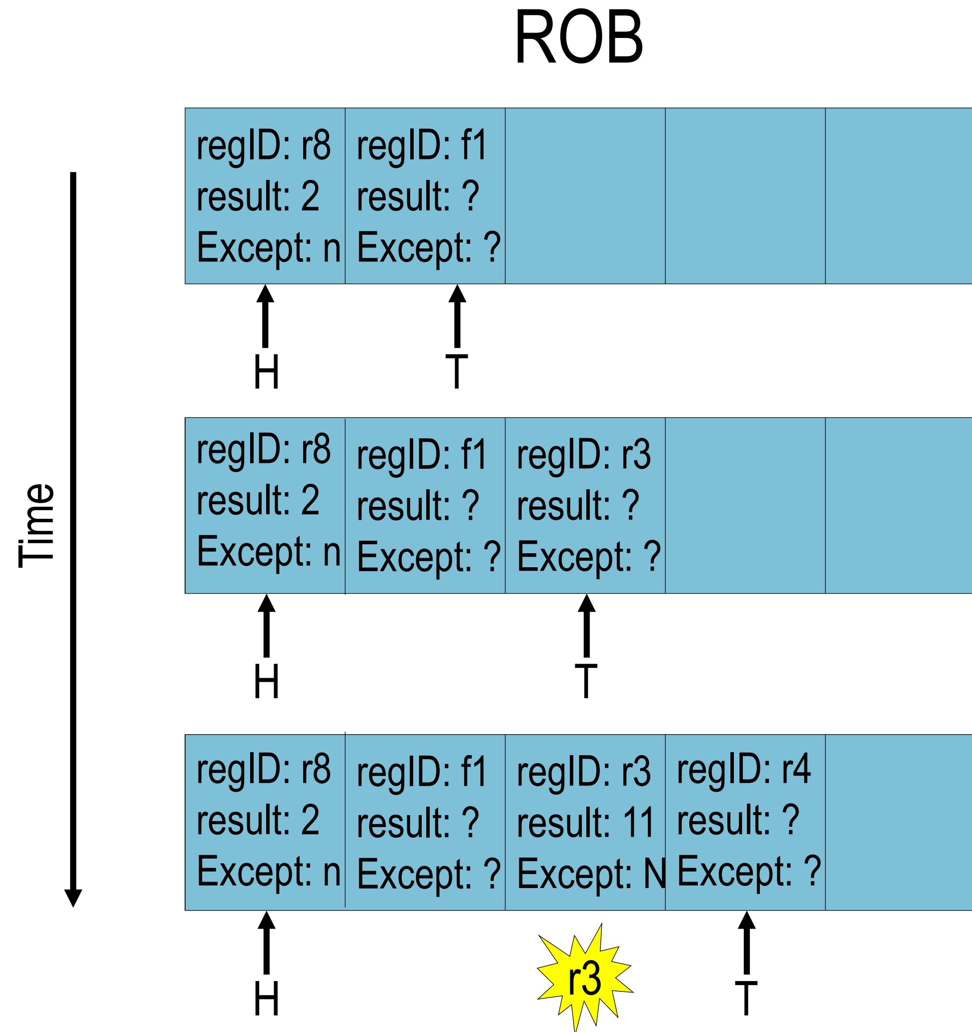
Reorder Buffer Example

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



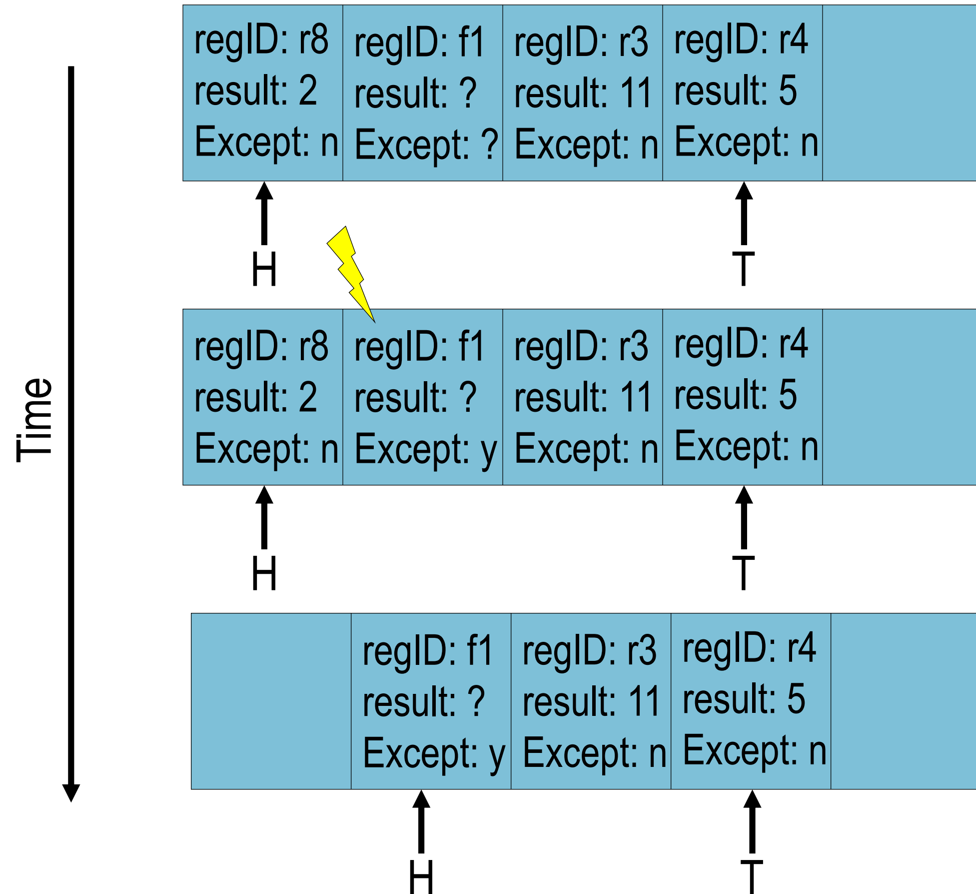
Reorder Buffer Example

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



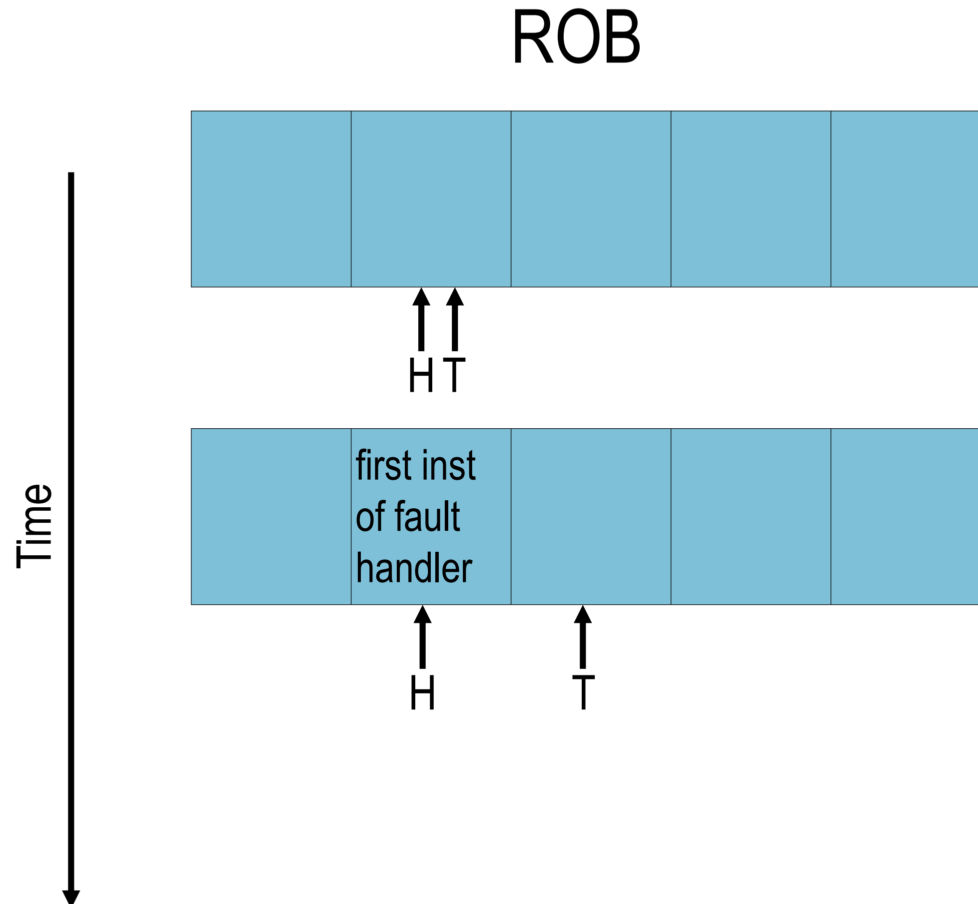
Reorder Buffer Example

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?

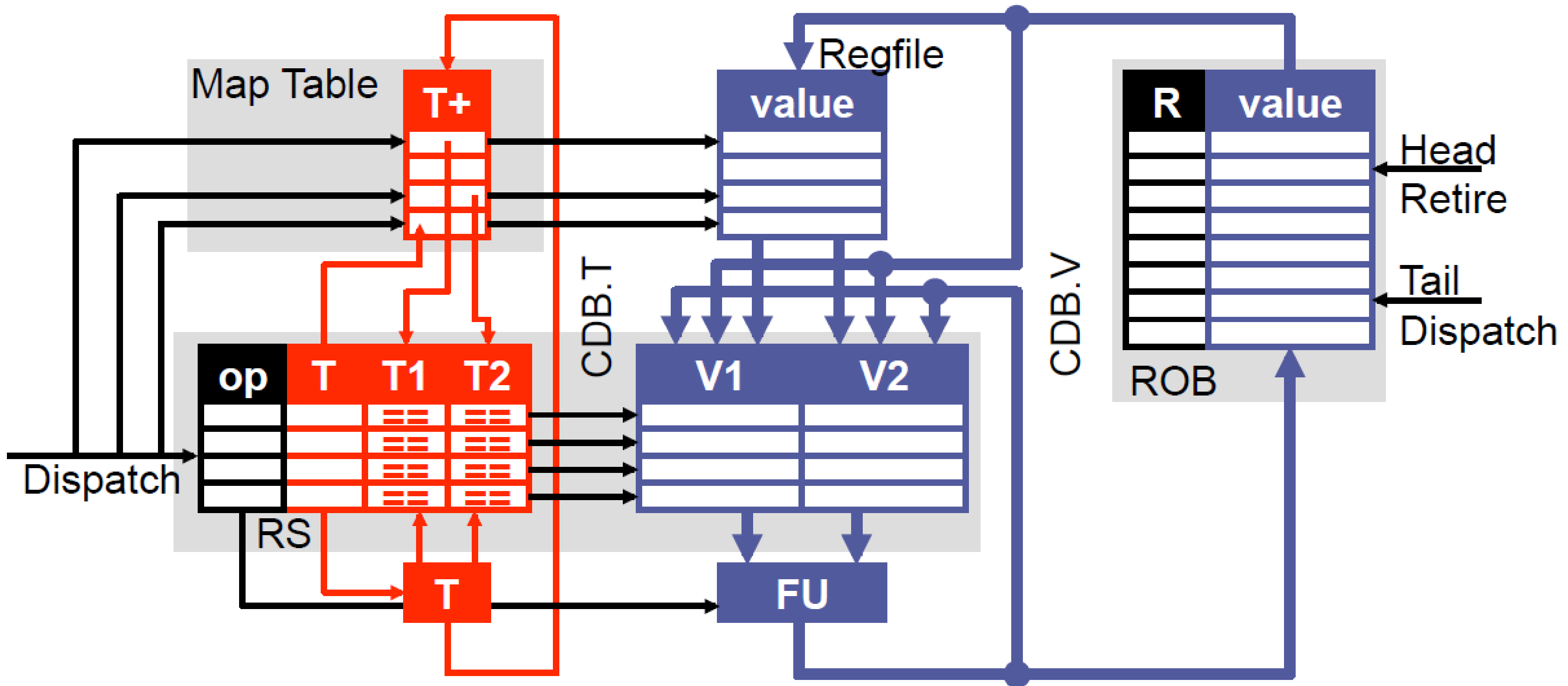
And while we're at it…

- 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.

Adding a Reorder Buffer



Tomasulo Data Structures
(Timing Free Example)

CDB	
T	V

Map Table	
Reg	Tag
r0	
r1	
r2	
r3	
r4	

Reservation Stations (RS)								
T	FU	busy	op	RoB	T1	T2	V1	V2
1								
2								
3								
4								
5								

ARF	
Reg	V
r0	
r1	
r2	
r3	
r4	

Instruction
r0=r1*r2
r1=r2*r3
Branch if r1=0
r0=r1+r1
r2=r2+1

Reorder Buffer (RoB)							
RoB Number	0	1	2	3	4	5	6
Dest. Reg.							
Value							

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?

More review questions

1. What is the purpose of the RoB?
2. Why do we have both a RoB and a RS?
   - Yes, that was pretty much on the last page…
3. Misprediction
   a) When to we resolve a mis-prediction?
   b) What happens to the main structures (RS, RoB, ARF, Rename Table) when we mispredict?
4. What is the whole purpose of OoO execution?

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**→**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→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)