SFU

CMPT 450/750: Computer Architecture
Fall 2024

Branch Prediction & Precise Interrupts

Alaa Alameldeen & Arrvindh Shriraman

Why Branch Prediction? Branch Penalty

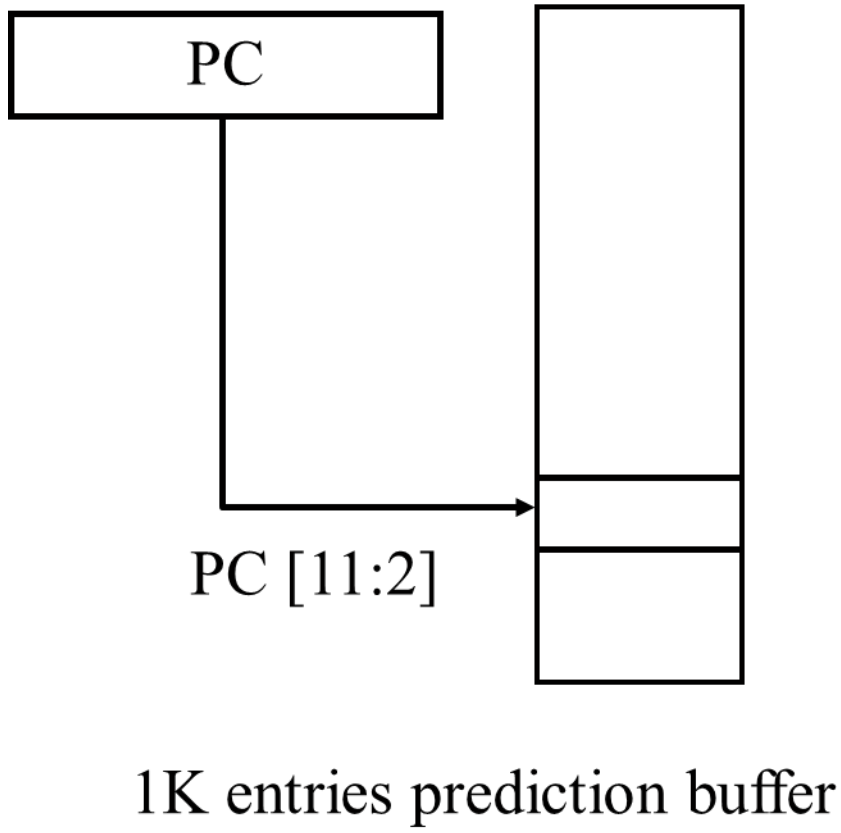
- **Example: Processor has a 15-stage 6-wide pipeline. An incorrectly predicted branch leads to pipeline flush (15-cycle penalty). Program retires an average of 4 instructions per cycle if all branches are predicted correctly. Program has 1 million instructions including 100,000 conditional branches.**
- **Comparing different branch prediction scenarios:**
  - Perfect BP: IPC = 4.00 Execution Time = 1,000,000/4 = 250,000 cycles
  - 90% BP accuracy: 1/10 branches incorrectly predicted
    - IPC = 1,000,000/(250,000 + 0.1x100,000x15) = 2.5 (60% slower)
  - 95% BP accuracy: 1/20 branches incorrectly predicted
    - IPC = 1,000,000/(250,000 + 0.05x100,000x15) = 3.08 (30% slower)
  - 99% BP accuracy: 1/100 branches incorrectly predicted
    - IPC = 1,000,000/(250,000 + 0.01x100,000x15) = 3.77 (6% slower)
  - No BP: Fetch stalled until branch is resolved (4 pipeline stages)
    - IPC = 1,000,000/(250,000 + 100,000x4) = 1.53 (160% slower)
  - 70% BP accuracy: 3/10 branches incorrectly predicted
    - IPC = 1,000,000/(250,000 + 0.3x100,000x15) = 1.43 (180% slower)

Reducing Branch Cost with Hardware Prediction

- **Branch prediction basics:**
  - ➢We need to predict conditional branch outcome to select the address for next instruction fetch
    - ❑PC + 4 (assuming each instruction is 4B like in RISC architectures)
    - ❑Or branch *target* address
  - ➢Also we need to quickly determine the branch *target* address
    - ❑Direct branches
    - ❑Register indirect branches
    - ❑Returns

Predicting Conditional Branch Outcomes

- **Static branch prediction: Use profile information (from previous runs, or from compiler) to predict the same outcome for each branch**
  - Disadvantage: Many branches aren't highly biased, leading to high misprediction rate
- **Simplest dynamic branch prediction scheme uses a *branch-prediction buffer* or *branch history table***
  - Small memory indexed by the lower portion of the branch address
  - Stores previous branch outcomes to predict next outcome
  - Table is not *tagged*: Prediction may have been put in the entry by a different branch (Aliasing)



Predicting Conditional Branch Outcomes

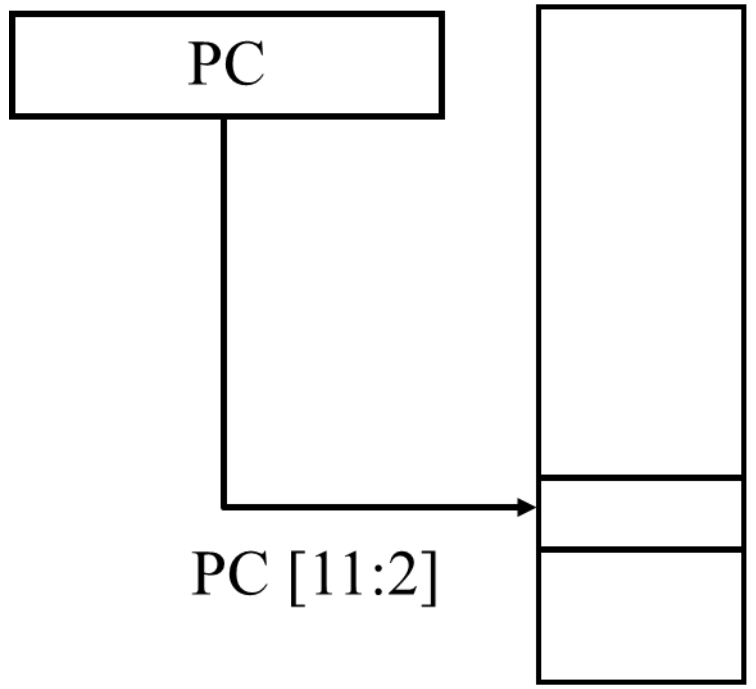
- **1-bit prediction buffer stores the last executed branch outcome, and uses it to predict the next outcome**
  - ➢If bit = 1, branch is predicted taken
  - ➢If bit = 0, branch is predicted not-taken
- **A simple 1-bit scheme may not perform well**
  - ➢Example: Below is a series of branch outcomes and corresponding predictions

outcomes	1111011110111101
predictions	1111011110111110
mispredictions	111**10**111**10**111**10**

(60% accurate)

Predicting Conditional Branch Outcomes: Bimodal Predictor

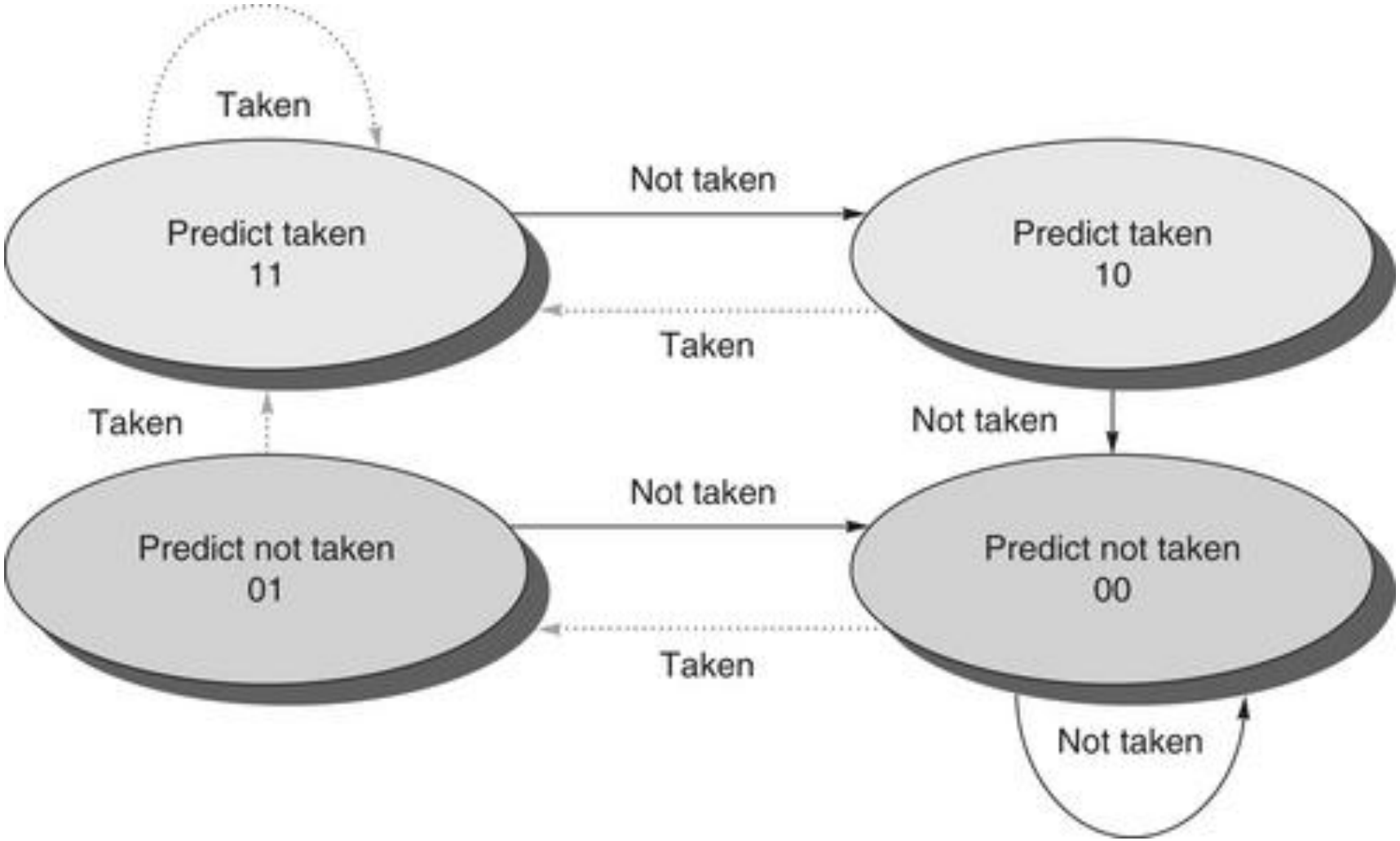
- **2-bit saturating counter often used**
  - ➢Branch taken ==> increment state
    - ❑Max state “11” stays at “11” when incremented
  - ➢Branch not-taken ==> decrement state
    - ❑Min state “00” stays at “00” when decremented
  - ➢“11” and “10” are predict taken states
  - ➢“00” and “01” are predict not-taken states



1K entries prediction buffer

2-bit Saturating Counter State Machine

ARCH Figure C.18



Note: Some advanced predictors use perceptrons instead of 2-bit saturating counters to predict future branches

Predicting Conditional Branch Outcomes

- **Assuming initial state to be "11", i.e., 3, branch outcomes and corresponding predictions (bimodal predictor) are:**

outcomes	1111011110111101
BP state	3333233332333332
predictions	1111111111111111
mispredictions	1111**1**1111**1**1111**1**1
	(80% accurate)

- **Key idea for conditional branch predictors: Use branch outcomes to "train" predictor; then use predictor to predict future outcomes**

Correlating Branch Predictors

- **2-bit prediction schemes use the recent behavior of a single branch to predict the future behavior of that branch**
- **Behavior of longer sequence of branch execution *history* often provides more accurate prediction outcome**
- **Behavior of *other* branches rather than just the branch we are trying to predict is sometimes important**
  - Because outcomes of different branches often correlate
  - Global branch history
- **For some branches, prior history execution of the same branch is important**
  - Because of loops
  - Local branch history

Correlating Branch Predictors: Code Example

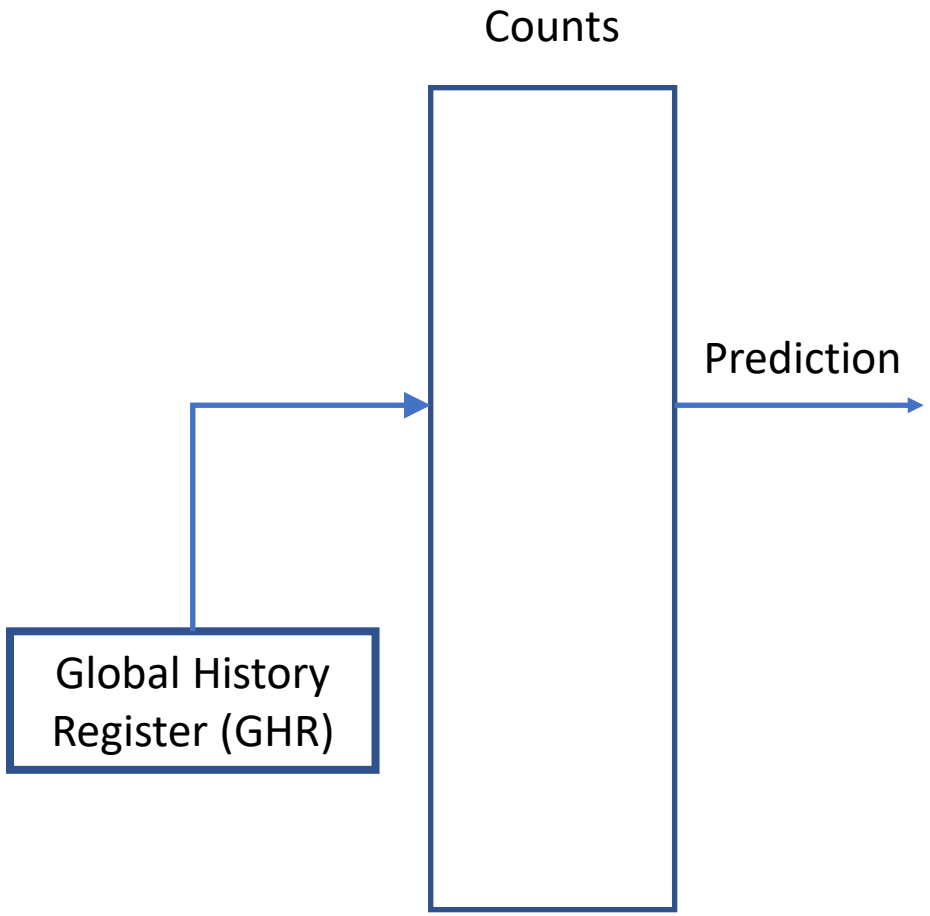
```
if (aa == 2)
  aa = 0;
if (bb == 2)
  bb = 0;
if (aa != bb) {
```

```
        DADDIU   R3,R1,#-2
        BNEZ     R3,L1          ;branch b1   (aa!=2)
        DADD     R1,R0,R0       ;aa=0
L1:     DADDIU   R3,R2,#-2
        BNEZ     R3,L2          ;branch b2   (bb!=2)
        DADD     R2,R0,R0       ;bb=0
L2:     DSUBU    R3,R1,R2       ;R3=aa-bb
        BEQZ     R3,L3          ;branch b3   (aa==bb)
```

Outcome depends on b1 and b2

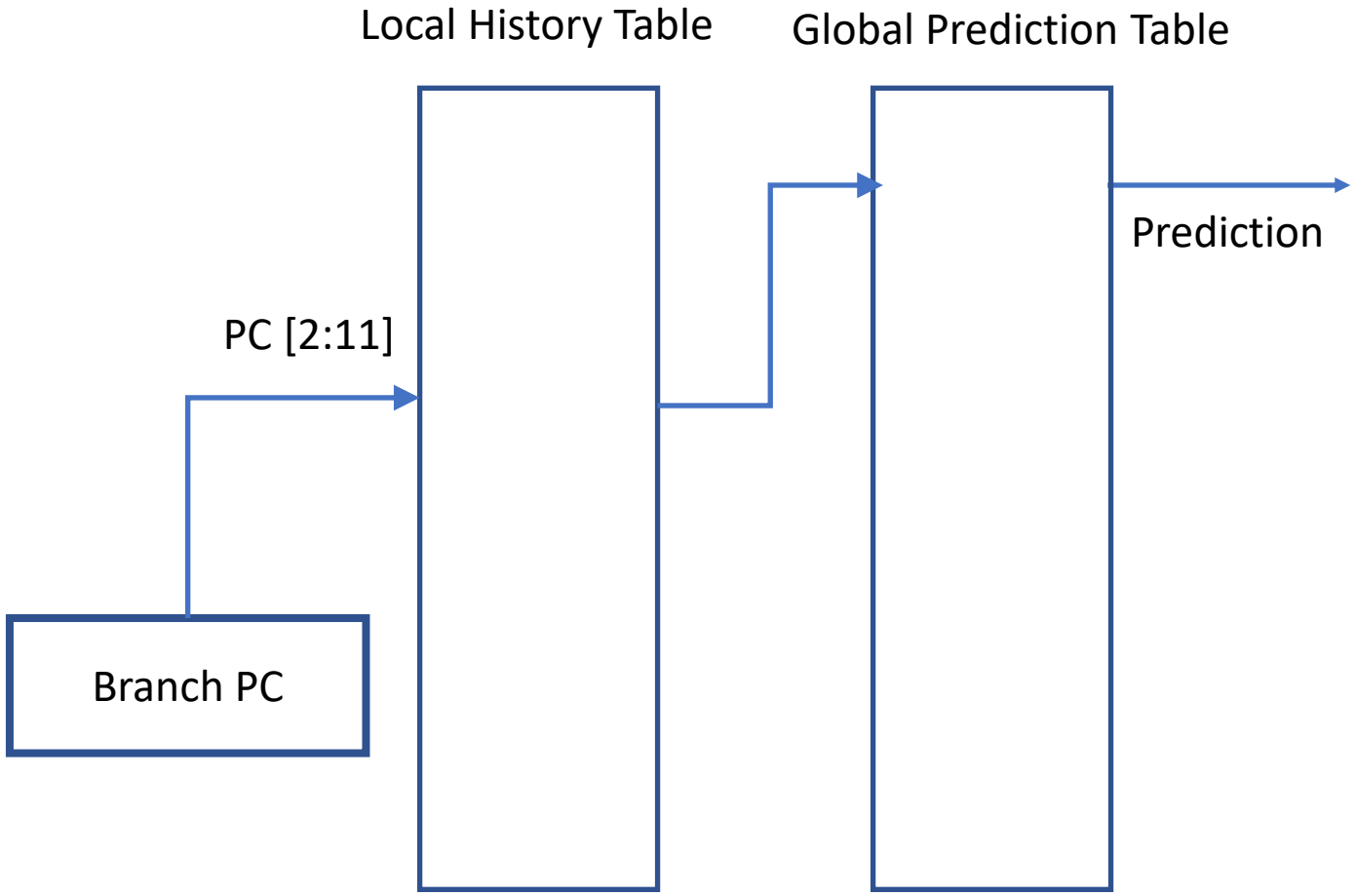
Global History Predictor

- **Can be used to capture correlating branches from different branch addresses**
- **Prediction is based on global branch history, i.e., outcome (taken/not taken) of previous branches from all PCs**
  - History stored in a global history register (GHR), shifted for each new branch
- **Key parameter: length of global history (size of GHR)**
  - Long history reduces aliasing, but requires large branch prediction table
  - Short history is more space-efficient, but less precise due to aliasing

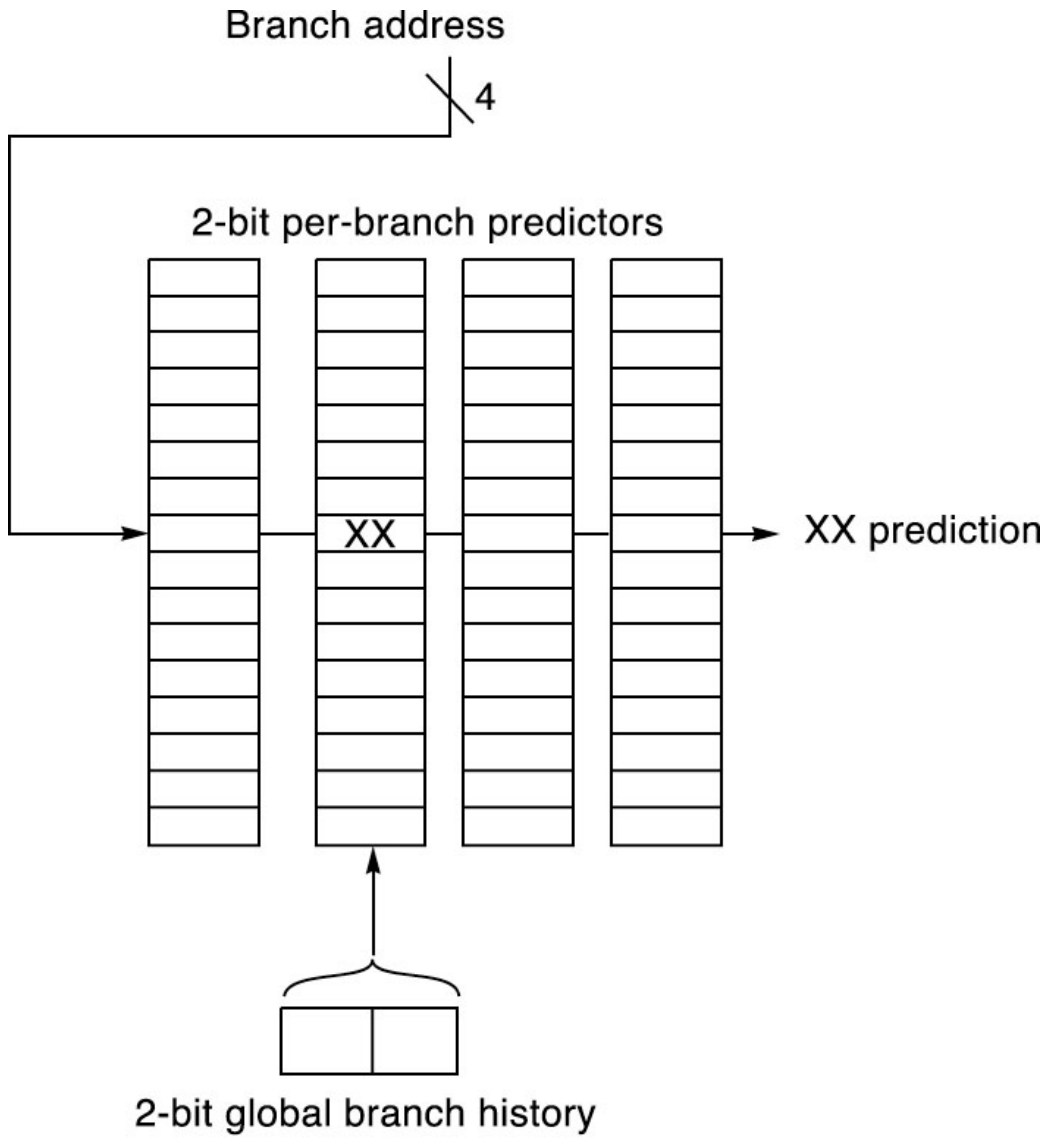


Local History Predictor

- **Uses history of the current branch PC to predict branch outcome**
  - Local History Table stores outcome of the last N instances of each branch PC (subject to aliasing)
  - Local history used to index into global prediction table which stores counts corresponding to that history
- **Key parameters:**
  - Number of history bits per Branch PC (determines size of Global Prediction Table)
  - Number of bits used for Branch PC (determines size of Local History Table)

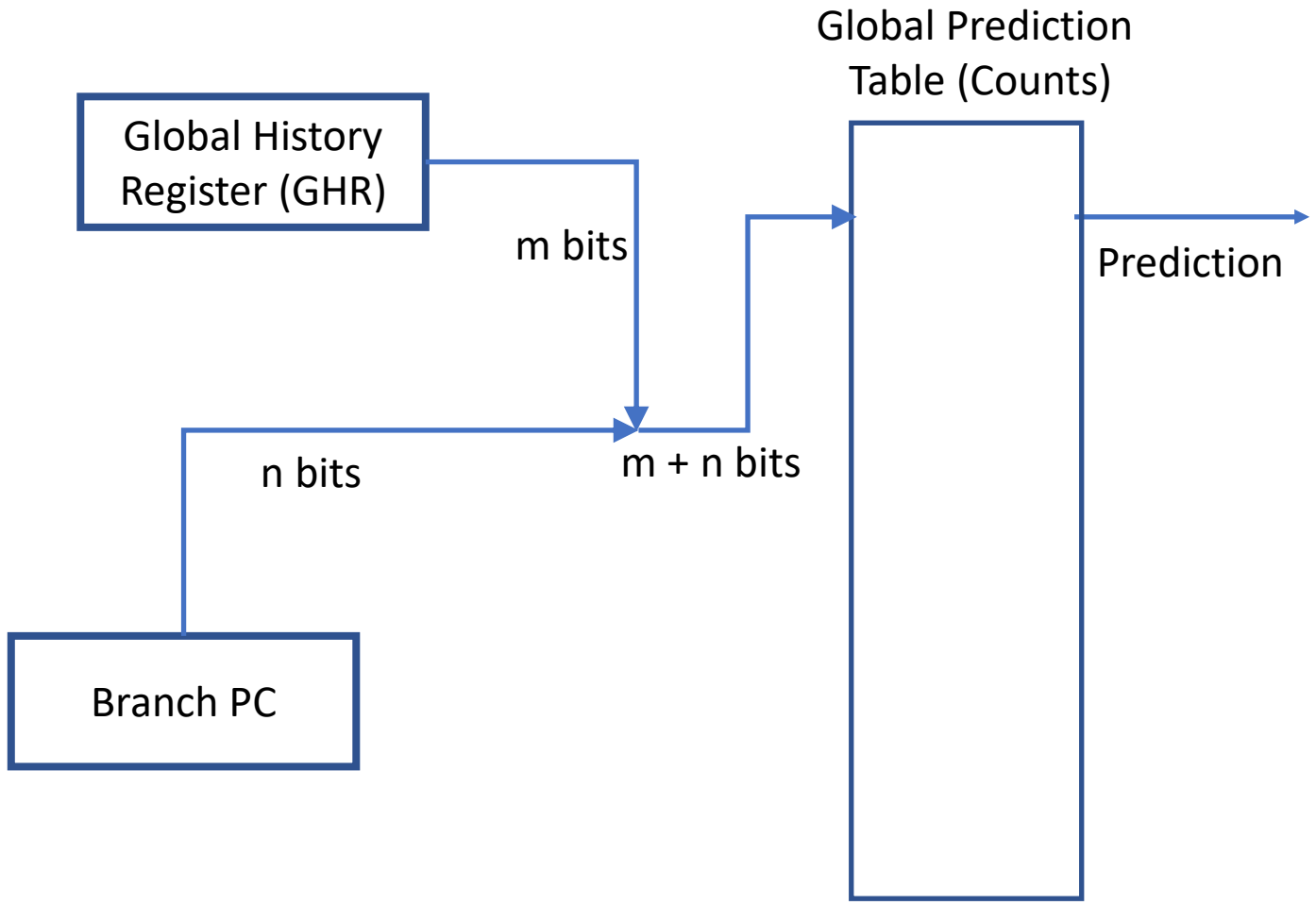


Combining Local and Global Histories: Correlating Branch Predictor with 2-bit Global History Register



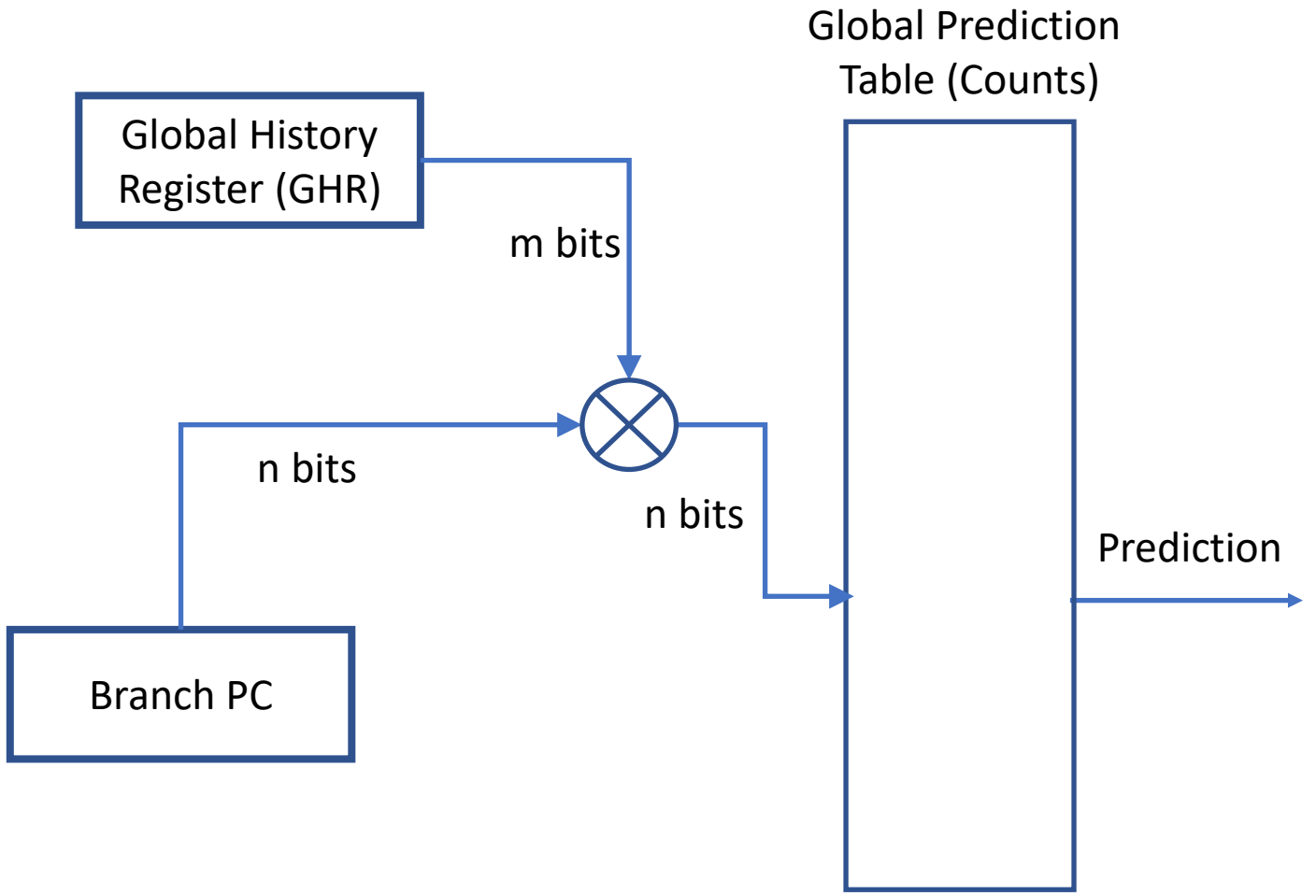
Combining Global and Local History: Global History Predictor with Index Selection

- **We can combine a few bits from (local) branch PC and a few bits from the Global History Register to index into prediction table**
  - ➢ Size of table depends on GHR size and #bits selected from branch PC
  - \+ Less interference between global and local histories
  - \- Requires large prediction table to be accurate
- **Key parameters:**
  - ➢ GHR size
  - ➢ Number of bits used from Branch PC



Combining Global and Local History: Global History Predictor with Index Sharing (GShare)

- **We can XOR a few bits from (local) branch PC and a few bits from the Global History Register to index into prediction table**
  - ➢ Size of table depends on # bits from Branch PC
  - \+ Space-efficient: Good accuracy with smaller table sizes.
  - \- More aliasing: Different PC & GHR combinations map to same entry



Tournament Predictor: Adaptively Combining Branch Predictors

- Some branches are predicted more accurately with *global* predictors
- Other branches are predicted better with *local* predictors
- It is possible to combine both types of predictors, and dynamically select the right predictor for the right branch
- The selector is yet another predictor with state machine per entry

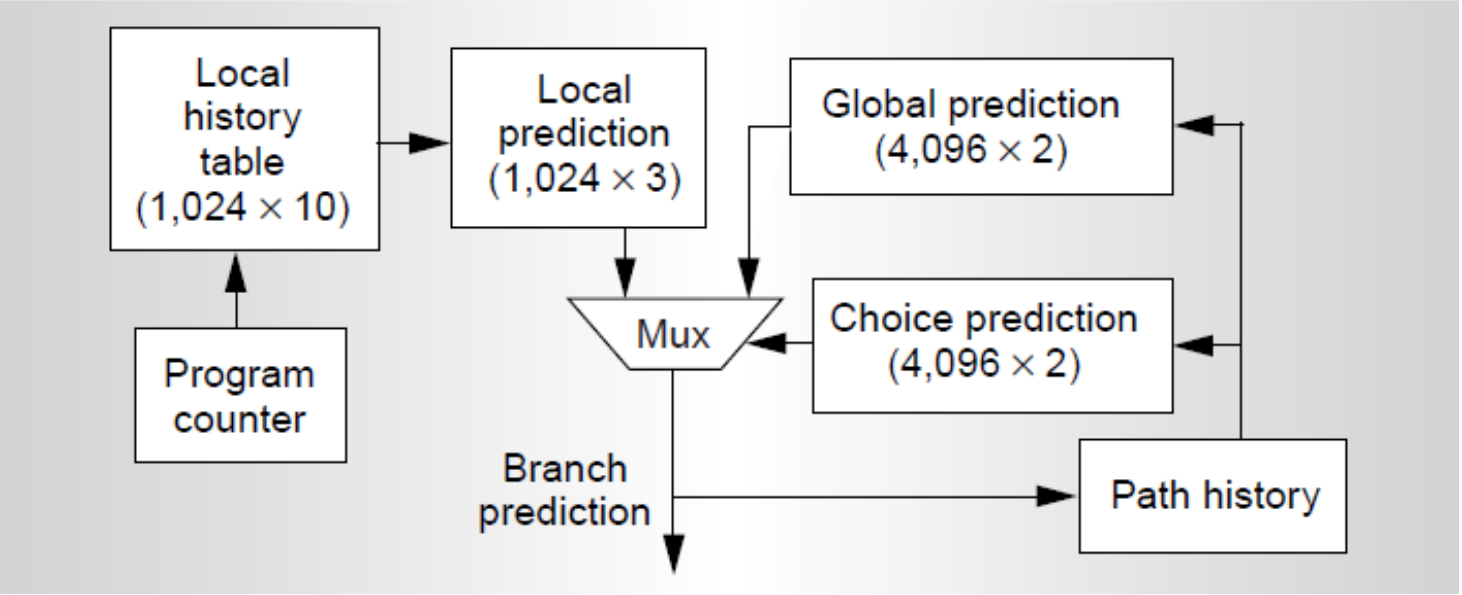


Figure from R.E. Kessler, "The Alpha 21264 Processor," IEEE Micro, 1999

TAGE Branch Predictor

- **TAgged GEometric history length branch prediction (Seznec & Michaud,JILP 2006)**
- **More features published in 2007, 2011, 2014, 2016)**
  - Variations of TAGE won the last four branch prediction championships (CBP)
- **History is tagged**
  - Avoids aliasing (branch predicted using another branch's history)
- **History length is geometric**
  - Hard-to-predict branches use longer history than more predictable branches
  - Uses a base predictor and four other predictors with geometrically increasing history lengths

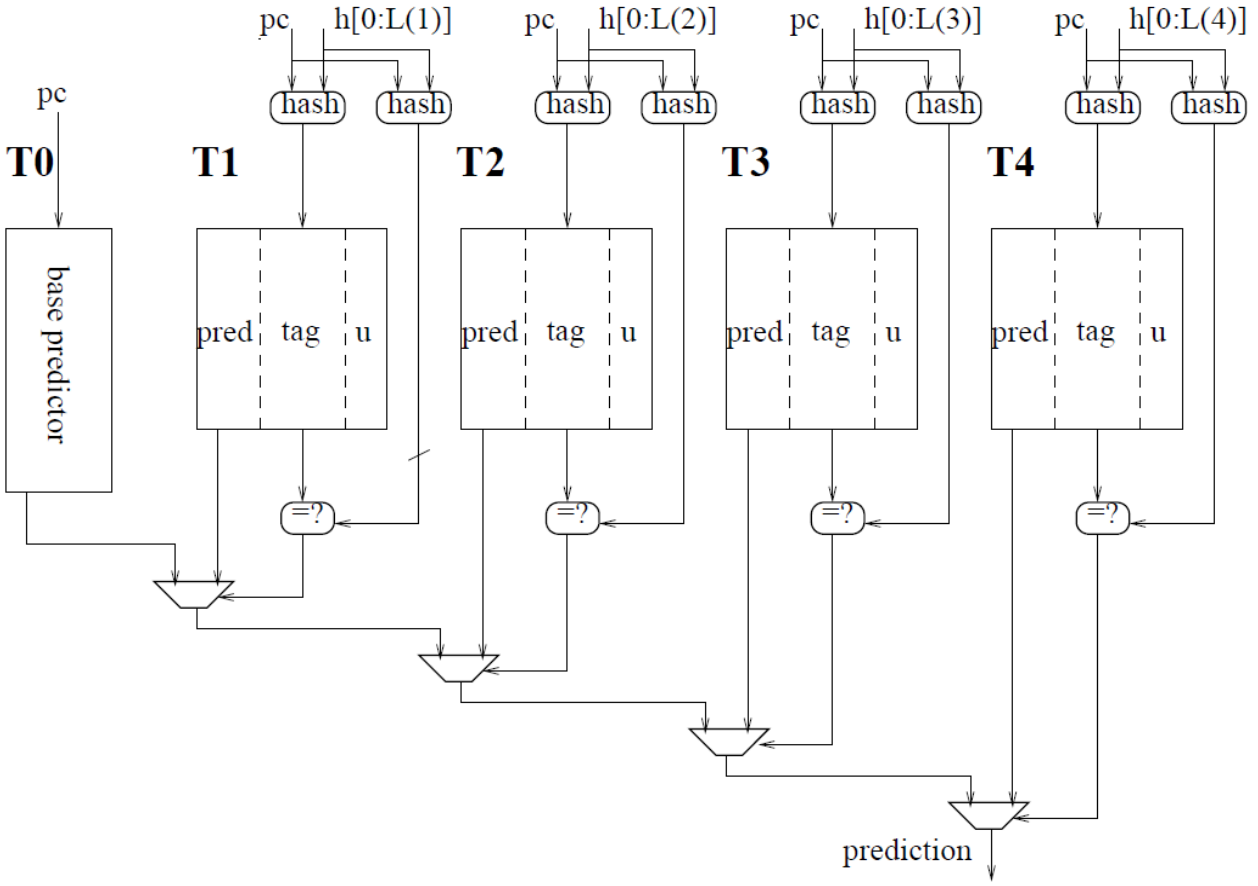
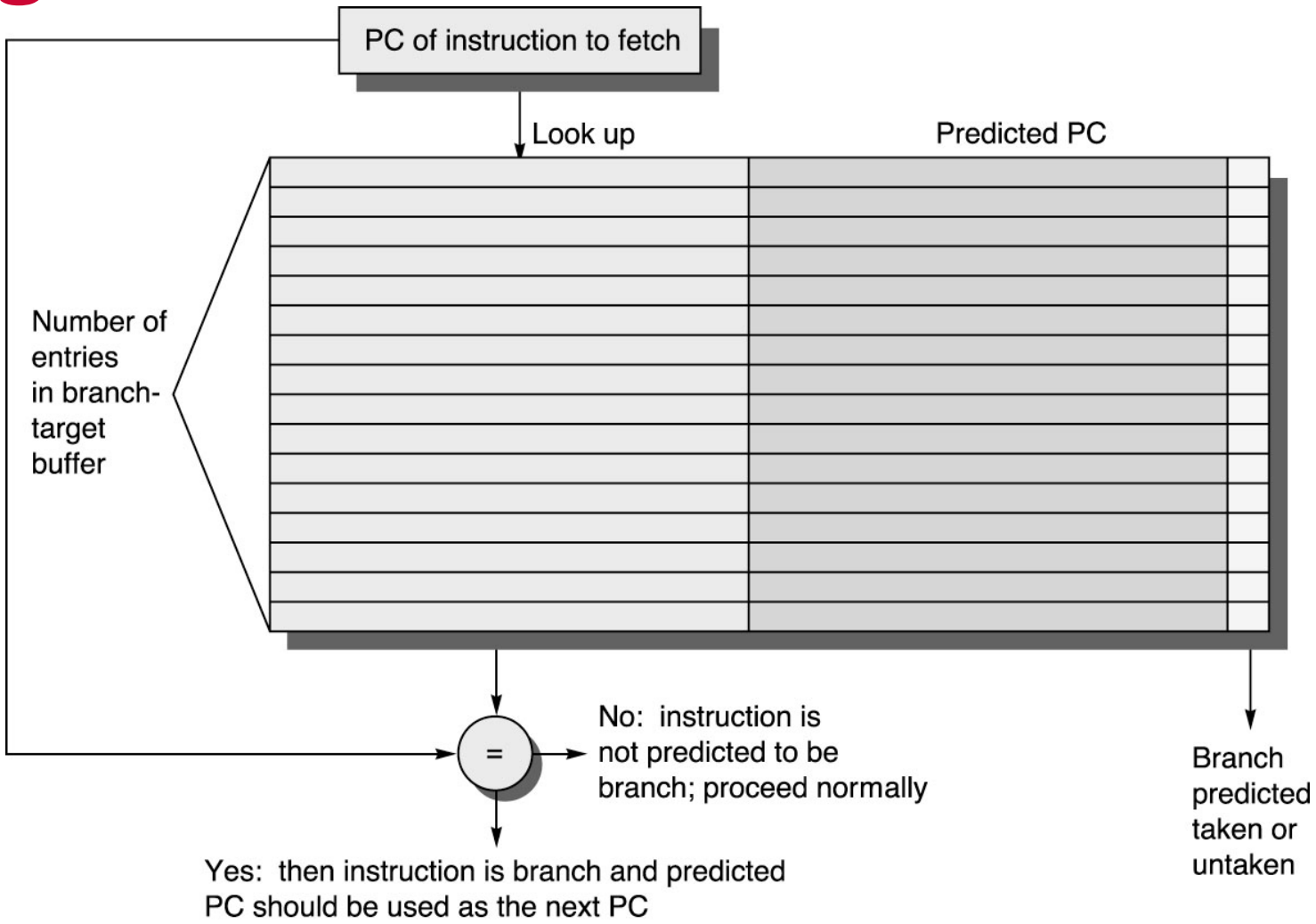


Figure from A. Seznec, "The L-TAGE Branch Predictor," JILP, 2007.

Other Components of Branch Prediction: Branch Target Buffer (BTB)

- **A cache that stores branch targets**
- **Accessed by the PC of the instruction currently being fetched**
- **Allows branch target to be read in the IF stage**
  - ➢When a branch is predicted taken (and also for unconditional branches), the fetch of the instruction at the branch target address can proceed immediately in the next cycle
  - ➢Saves stall cycles that would have been needed to wait for the decoding of the branch and the computation of the target

Branch Target Buffer



Predicting Indirect Branches

- **Indirect branches have multiple potential targets**
  - ➢Target address comes from a register, which can have many possible values
- **Branch target buffers could be used for indirect branch target prediction**
  - ➢However, many mispredictions can happen because the BTB can store only one target per branch
- **Most indirect branches come from return instructions**

Predicting Return Addresses: Return Address Stack

- Return Address Stack (RAS) is a small address buffer organized as a stack
- When a Call is encountered, the Return address (which is Call address + 4) is pushed onto the RAS
- When a Return instruction is encountered, the address from the top of the RAS is popped and used as the target

Dynamic Instruction Scheduling

Dynamic Scheduling in Out-of-Order Processors

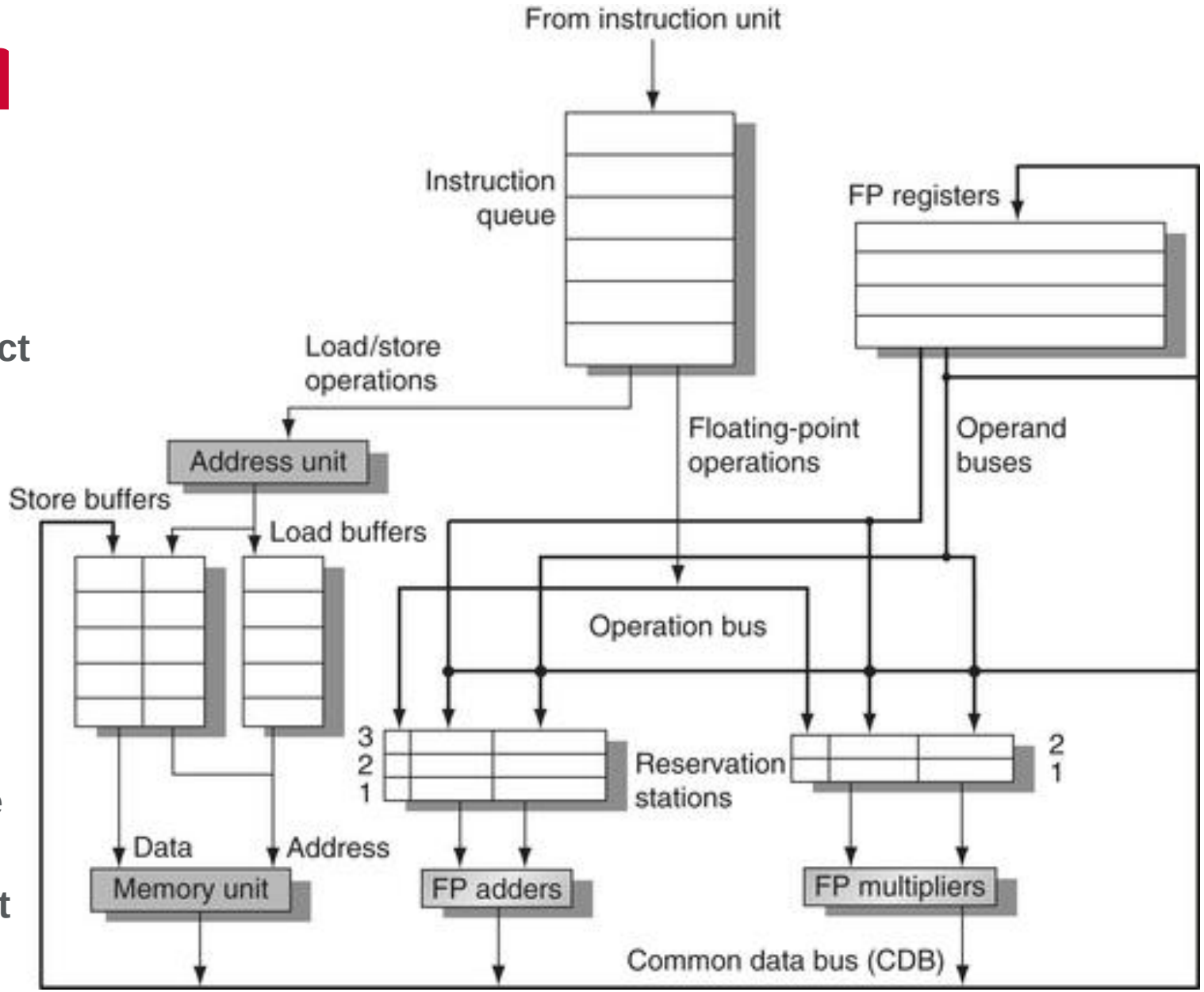
- **To overcome data hazards and enable OoO execution, processors dynamically schedule instructions that are ready to execute**
  - Instructions are scheduled even if older instructions haven't completed
  - Instructions are scheduled if their dependences are satisfied, and there is a functional unit available
- **Key ideas for dynamic scheduling**
  - Tracking data dependences (to ensure data hazards are avoided)
  - Checking for structural hazards
  - Register renaming (to avoid WAR and WAW, i.e., name dependences) – discussed last week
- **The Instruction Decode (ID) stage in the simple 5-stage pipeline can be split into 3 stages:**
  - **Decode**: Identify instruction type, source and destination registers
  - **Issue**: Check for structural hazards and data hazards then send instruction to functional unit
  - **Read Operands**: Read input operands from register file

Dynamic Scheduling using Tomasulo's Algorithm

- **Invented by Robert Tomasulo. First used in the IBM 360/91 floating point unit**
  - IBM 360/91 had long memory access latencies and long floating point delays
  - Tomasulo's algorithm aims at speeding up runtime by allowing out-of-order execution
- **Tomasulo's algorithm features:**
  - Distributed hazard detection logic (across reservation stations and Common Data Bus). Tracks when input operands of instructions are available (to avoid RAW hazards)
  - Implements register renaming in hardware (to avoid WAR and WAW hazards)
- **Reservation stations**
  - Buffer operands of instructions waiting to issue
  - Fetch and buffer operands as soon as they are available
  - Pending instructions designate the reservation station that will provide their input
  - When instructions issue, register specifiers for input operands are renamed to the designated reservation station (effectively implementing register renaming)

Tomasulo's Algorithm

- Instructions sent from the instruction unit into the instruction queue (issued in FIFO order).
- The reservation stations include the operation, operands, and info used to detect and resolve hazards
- Load buffers: (1) hold components of effective address; (2) track outstanding loads waiting for memory, and (3) hold the results of completed loads waiting for the CDB.
- Store buffers: (1) hold components of the effective address, (2) hold destination memory addresses of outstanding stores and (3) hold the address and value to store until memory is available.
- All results from FP units & load unit are put on the CDB, which goes to the FP register file, reservation stations and store buffers.



ARCH Figure 3.6

Tomasulo's Algorithm: Instruction Execution Steps

1. Issue (Dispatch)

- Get the next instruction from the head of the instruction queue (FIFO order)
- If there is a matching and empty reservation station, issue the instruction to the station with the operand values (if they are available in registers). If the operands are not in registers, keep track of the functional units that will produce the operands
- If there are no empty reservation stations, then the instruction stalls until one is available
- Effectively, the issue step renames registers (eliminating name dependences)

2. Execute

- If one or more of the operands is not yet available, monitor the common data bus.
- When an operand becomes available, it is placed into any reservation station awaiting it.
- When all operands are available, instruction can be executed at the functional unit.

3. Write Result

- When the result is available, write it on the CDB and then to registers and any reservation stations or store buffers waiting for it
- Stores are buffered in the store buffer until both the value to be stored and the store address are available, then written to memory when available

Tomasulo's Algorithm: RS & RF

- **Each reservation station (RS) has the following fields:**
  - Op: Operation (e.g., add, multiply,…) to perform on source operands S1 and S2
  - Q1, Q2: Source reservation stations that will produce S1 and S2, respectively
    - If operand value is available then corresponding Q value is zero
  - V1, V2: Value of source operands S1 and S2, respectively
    - For each operand, only one of Q or V is valid
  - A: Info needed for memory address calculation for loads and stores
  - Busy: True if reservation station is occupied, false if it's free
- **Each register in the register file (RF) has a field "Qi" indicating the number of the reservation station that will produce the value to be written into this register**
- **Examples in textbook Section 3.5 illustrate in detail how Tomasulo's algorithm works**

Tomasulo's Algorithm: Example

```
1.    L.D     F6,32(R2)
2.    L.D     F2,44(R3)
3.    MUL.D   F0,F2,F4
4.    SUB.D   F8,F2,F6
5.    DIV.D   F10,F0,F6
6.    ADD.D   F6,F8,F2
```

- What is the state of hardware structures after first load writes its result?
- Note: Instruction status is not a hardware structure

		Instruction status		
Instruction		Issue	Execute	Write result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

Register status

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

ARCH Figure 3.7

Tomasulo's Algorithm: Example

```
1.    L.D     F6,32(R2)
2.    L.D     F2,44(R3)
3.    MUL.D   F0,F2,F4
4.    SUB.D   F8,F2,F6
5.    DIV.D   F10,F0,F6
6.    ADD.D   F6,F8,F2
```

- What is the state of hardware structures when MUL is ready to write its result?
- Note: Instruction status is not a hardware structure

		Instruction status		
Instruction		Issue	Execute	Write result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	
SUB.D	F8,F2,F6	√	√	√
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√	√	√

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	Mem[44 + Regs[R3]]	Regs[F4]			
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

Register status

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1					Mult2			

ARCH Figure 3.8

Precise Interrupts in Superscalar Processors

Precise Interrupts

- **Process state consists of program counter, registers and memory**
- **An interrupt or exception is precise if the saved process state is consistent with the sequential architectural model**
    1. All instructions preceding interrupted instruction have been executed and modified state correctly
    2. All instructions following interrupted instruction are unexecuted and haven't modified state
    3. If interrupt is caused by exception due to an instruction in the program, the saved PC points to the interrupted instruction
- **Providing precise state can be difficult on pipelined processors that allow out of order execution**

Precise Interrupts: Example

```
I1:    LOAD [ADDR]→R1
I2:    ADD R2, R3 → R4
```

```
I1:    FDIV F2, F3 → F1
I2:    ADD R2, R3 → R4
```

- `I1` results in a page fault (top) or floating point exception (bottom)
- `I1` takes longer to run than `I2`
- By the time exception happens, `I2` has already completed execution, and R4 has already been modified
  - ➢ Architectural state at time of exception is imprecise

Why is Precise State Needed?

- **With speculative execution, precise state must be maintained to recover from mispredictions and exceptions**
  - Exceptions, e.g., page faults, occur without warning, but infrequently
  - Branch mispredictions occur in predictable locations, but happen frequently
- **Precise state is necessary and/or desirable**
  - I/O and timer interrupts: makes restarting possible
  - Allows restart after page fault (virtual memory systems)
  - Software debugging: isolate instruction causing bug
  - Graceful recovery from arithmetic exceptions by software
  - Implementing unimplemented op-codes in software
  - Precise interrupts from privileged instructions are necessary to implement virtual machines

Solution: In-Order Execution

- We could avoid the problem completely by disallowing out of order completion at the expense of performance
- Result shift register (RSR) can be used for instructions to reserve write back cycles to the register file
- Instructions that conflict with earlier instructions marked in the RSR stall at the issue stage

STAGE	FUNCTIONAL UNIT SOURCE	DESTN. REGISTER	VALID	PROGRAM COUNTER
1			0	
2	INTEGER ADD	0	1	7
3			0	
4			0	
5	FLT PT ADD	4	1	6
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
N			0	

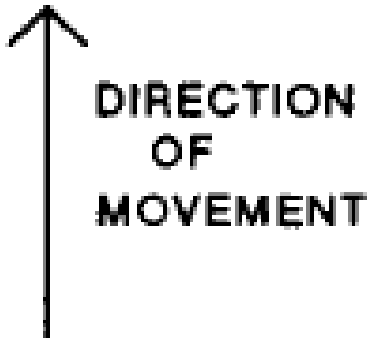


Figure from J. Smith and A. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans. Computers, 1988

Precise Interrupts in Out-of-Order Processors

- **The general approach:**
  - Maintain copies of the speculative state and the precise state
  - Important to implement efficiently (to quickly recover from branch mispredictions and handle exceptions)
- **Register buffering methods**
  - Checkpoint repair
  - Reorder buffer
  - History Buffer
  - Future file

Checkpoint Repair

- **Multiple copies of the register file provide multiple logical spaces, organized as a stack**
- **A single logical space is active at any given time**
- **Periodically, the active logical space's architecture state is pushed onto the stack as a backup or checkpoint**
- **An exception or mispredicted branch causes the following:**
  - The backup copy is made safe by allowing all instructions before the branch or exception to complete
  - The state is recovered from the backup copy
  - The backup copy used depends on: (1) the location of the exception or branch; and (2) the time backups were taken
- **Disadvantage**
  - Requires a lot of storage and overhead to create and store backups
  - This is especially bad for branch recovery, since a backup at every branch is needed

Reorder Buffer (ROB)

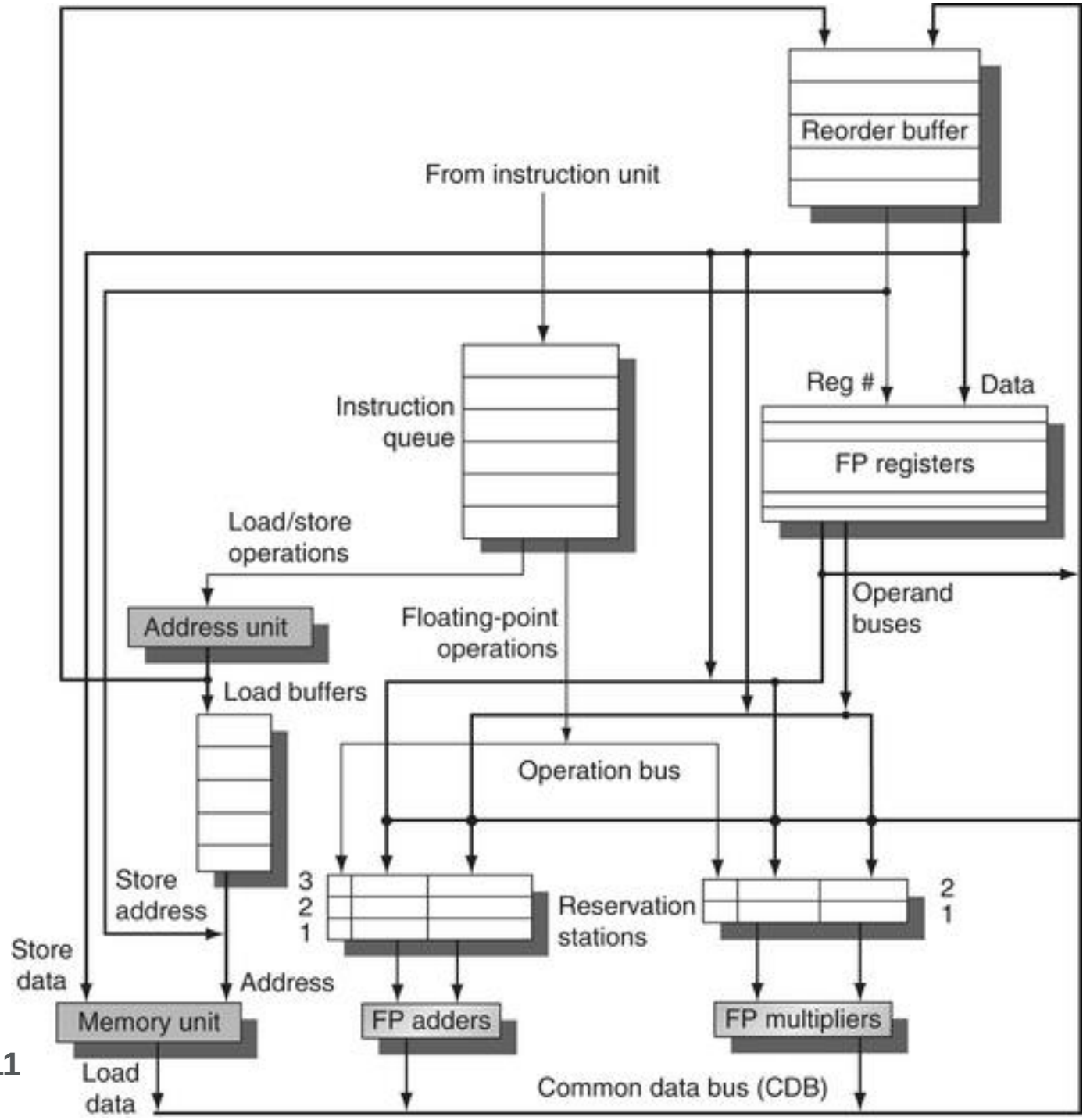
- **The reorder buffer FIFO is used to hold the speculative state while the register file holds the in-order (precise) state**
  - The architectural state is obtained by taking the most recent entry for a register from either the register file or the reorder buffer
- **When an instruction is decoded, an entry is allocated on the top of the reorder buffer to hold the result of the instruction**
- **ROB includes the following fields:**
  - **Instruction Type:** Whether the instruction is a branch (no destination), store (memory address destination) or a register operation (any other ALU instruction with register destination)
  - **Destination:** Location where results should be written. Either register number (for ALU operations or loads) or memory address (for stores)
  - **Value:** Holds the value of the instruction result until the instruction commits and writes result to destination
  - **Ready:** True if the instruction has completed execution, i.e. Value is valid

Reorder Buffer Operation

- **When a value reaches the head of the ROB**
  - If the associated instruction is not complete, the slot remains there until it completes
    - Instructions can continue to be decoded until the reorder buffer is full
    - ROB size can limit performance since it can throttle instruction decode
  - If there is a fault associated with the value, the ROB is discarded, and the in-order state of the register file is used
  - If there is no fault, the value is written to the register file and the entry removed from the ROB
- **The in-order (precise) state is always available in the register file, thus it is restored immediately**
- **ROB is the single site that completely records the program order of all instructions**
  - Note that register writes and memory accesses need to complete in program order for correctness (not just for precise interrupts)

Reorder Buffer

- ROB is added for all instructions
- ROB takes over the function of store buffer for store instructions
- For a wide superscalar, CDB is wider to allow multiple instructions to complete every cycle
- For fast execution, need to bypass RF and read data directly from ROB
  - ➢ High overhead to enable reads from multiple ROB entries in parallel



ARCH Figure 3.11

Instruction Execution Steps with Reorder Buffer

1. Issue (Dispatch)

- Get the next instruction from the head of the instruction queue (FIFO order)
- If there is an empty RS and an available ROB entry, issue the instruction to the station with the operand values, if they are available in registers or ROB. The number of ROB entry allocated to instruction is sent to RS to tag the result when written to the CDB
- If there is no empty RS or ROB is full, then the instruction stalls

2. Execute

- If one or more of the operands is not yet available, monitor the CDB till they are written there
- When all operands are available in the RS, instruction can be executed at the functional unit

3. Write Result

- When the result is available, write it on the CDB (with ROB tag) and then to ROB and any reservation stations waiting for it
- For stores: If value is available then it's written into the ROB entry value field; otherwise CDB is monitored until value is available

4. Commit (Retire): In order, only from head of the ROB

- ALU (store) instruction reaches head of the ROB: If value is available then value is written to register (memory) and ROB entry is removed
- Branch instruction reaches head of the ROB: If branch prediction is correct, ROB entry is removed. If branch is incorrectly predicted, ROB is flushed and execution restarts at the correct branch target

ROB Example

```
L.D     F6,32(R2)
L.D     F2,44(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F2,F6
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

- **Latencies:**
  - ADD: 2 cycles
  - MUL: 6 cycles
  - DIV: 12 cycles
- **What is the state of different structures when MUL is ready to commit?**

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	No	L.D	F6,32(R2)	Commit	F6	Mem[32 + Regs[R2]]
2	No	L.D	F2,44(R3)	Commit	F2	Mem[44 + Regs[R3]]
3	Yes	MUL.D	F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	Yes	SUB.D	F8,F2,F6	Write result	F8	#2 – #1
5	Yes	DIV.D	F10,F0,F6	Execute	F10	
6	Yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	MUL.D	Mem[44 + Regs[R3]]	Regs[F4]			#3	
Mult2	Yes	DIV.D		Mem[32 + Regs[R2]]	#3		#5	

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	. . .	Yes	Yes

Announcements

- **Reading Assignments**
  - ARCH “Hennessy & Patterson”
    - Appendix C.2 starting from Branch Prediction (Read)
    - Chapter 3.3, 3.4, 3.5, 3.6 (Read)
    - Chapter 3.7, 3.8 (Skim)
  - (750 Students) A. Seznec, “The L-TAGE Branch Predictor,” Journal of Instruction Level Parallelism, 2007. https://jilp.org/vol9/v9paper6.pdf (Read)
- **You need to complete gem5 tutorial (deadline: Sep 15). Not graded**
- **Assignment 1 due Sept 27**

Exam Logistics

- **Exam 1 is on Tuesday Oct 1 during class time (1:30-2:20 PM)**
- **Open book, notes, calculator**
- **Exam will be available on the course canvas page. Link active during class time**
- **You need to join the zoom link and turn your camera on**
  - Zoom link will be sent on Piazza the day of the exam
- **Attendance will be taken on exam days. You need to be on zoom for your exam to count.**