Speculative Execution

Speculative Execution

Instruction stream speculation

- Predict program behavior before execution flow determined
- Execute **PREDICTED** program flow
- Verify program results before commitment to state
- Recover from **MISPREDICTION**

Branch prediction

- Predict target address and action (taken/not-taken)

Instruction / data cache prefetch

- Load instructions and data to L1 cache before cache miss

Trace cache

- Cache frequent instruction flow sequences
- Span multiple control blocks

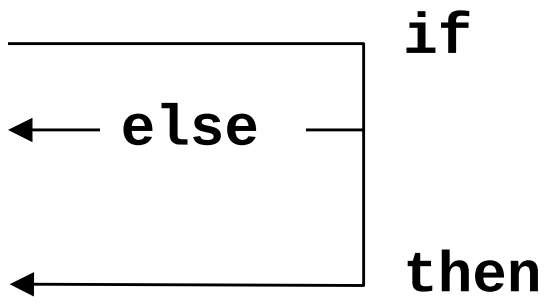
Branch predication

- Execute multiple branch flow alternatives
- Commit only verified instruction flow

Branch Behavior

`if-then-else`

PC	Instruction
PC_{BEQZ}	`BEQZ Reg, offset`
PC_{BEQZ} + 4	`fall-through`
	`...`
PC_{BEQZ} + 4 + offset	`target`
	`continue`



Speculation in dynamic scheduling

At **PC < PC_{BEQZ}**

- Prediction → branch taken
- Issue instructions target, continue, ...
- Cannot retire until **PC > PC_{BEQZ}**

At **PC = PC_{BEQZ}**

- Verify **BEQZ**
- Continue or correct misprediction

Basic Branch Prediction

Branch predictor in fetch + decode stage

Branch History Register (BHR)

Index	Source identifier (hash of instruction PC)
Entry	**Branch outcome history** **Prediction (target address, taken/not-taken) by some algorithm**

Branch Predictor

PC → BHR → prediction + target address

Predict taken ⇒ PC ← target address

Predict not taken or miss in BHR ⇒ PC ← fall-through address

Branch validation (verification)

Complete detailed branch execution

Compare with prediction

Correct prediction ⇒ continue program flow

Misprediction ⇒ flush pipeline → restart with correct target address

Update history table

Branch Prediction for DLX Pipeline

1. Branch predictor in IF stage

Identifies branch instruction

According to source address

Predicts branch from branch history

Taken

Predicts branch target address

Not-taken

Uses fall-through address

2. Validate branch instruction in ID stage

Usual Calculation:

Target address

Condition flag — taken or not-taken

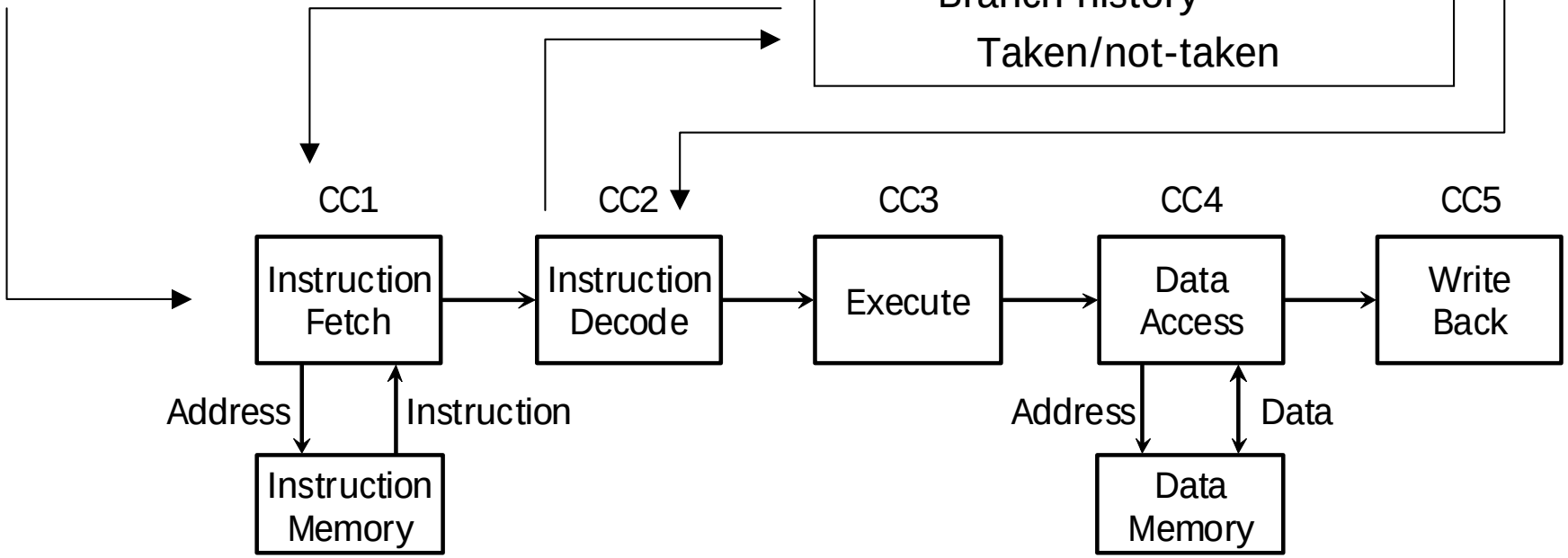
3. After validation

Update branch predictor

Target address

Branch history

Taken/not-taken



Simple Branch Prediction Performance for Loop

Simple static loop

```
        ADDI R1, R0, #N    ; N iterations
L1:     SUBI R1, R1, #1
        ALU Block          ; B lines of code
        BNEZ R1, L1
        I fall-through
```

	ADDI R1, R0, # N	IF	ID	EX	MEM	WB		
L1:	SUBI R1, R1, #1		IF	ID	EX	MEM	WB	
	< B-2 lines of ALU code >							
	BNEZ R1, L1	IF	ID	EX	MEM	WB		
	$I_{fall-through}$		IF	ID	ϕ	ϕ	ϕ	
L1:	SUBI R1, R1, #1			IF	ID	EX	MEM	WB
	< B-2 lines of ALU code >							
	BNEZ R1, L1	IF	ID	EX	MEM	WB		
L1:	SUBI R1, R1, #1		IF	ID	EX	MEM	WB	
	...							
	< B-2 lines of ALU code >							
	BNEZ R1, L1	IF	ID	EX	MEM	WB		
L1:	SUBI R1, R1, #1		IF	ID	ϕ	ϕ	ϕ	
	$I_{fall-through}$			IF	ID	EX	MEM	WB

R1 = N - 1

R1 = N - 2

R1 = 0

$$CPI_{\text{branch}}^{\text{stall}} = \frac{2}{N \times B + 2} \xrightarrow[N \times B \to \text{large}]{} 0$$

Importance of Branch Hazards

Branch instructions ⇒ program control flow

Loop instructions account for most run time

Branch statistics

Branch ≈ 20% — 25% of instructions

≈ 2/3 of conditional branches are taken

$$\frac{1 \text{ branch}}{4 \text{ instructions}} \times \frac{2 \text{ taken branches}}{3 \text{ branch}} = \frac{1 \text{ taken branch}}{6 \text{ instructions}}$$

Branch verification (correct execution) = N clock cycles

N wrong instructions → pipeline before branch verification

N wrong instructions must be flushed

N wasted clock cycles

$$CPI^{stall} = \frac{N \text{ wasted clock cycles}}{\text{taken branches}} \times \frac{1 \text{ taken branch}}{6 \text{ instructions}} = \frac{N \text{ wasted clock cycles}}{6 \text{ instructions}}$$

N ≥ 1 for DLX ⇒ $CPI^{stall} \geq 1/6$

Dynamic Branch Prediction

Past behavior predicts future branch behavior

Assumes repetitive/cyclical behavior in branches

Possible approaches

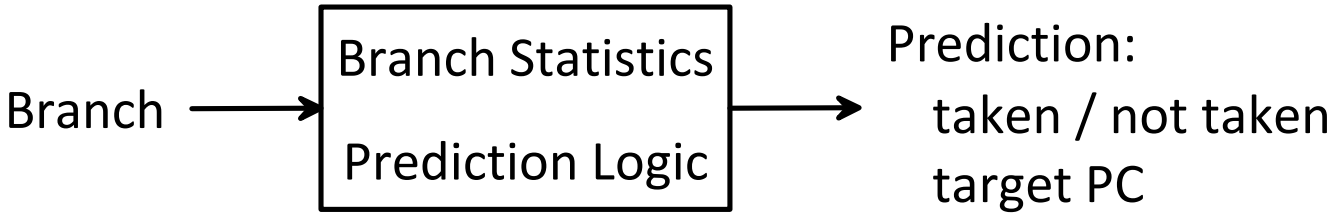
Global prediction

Predict from recent execution pattern of all recent branches

Local prediction

Predict from recent execution pattern of individual branch

Combination of global and local prediction



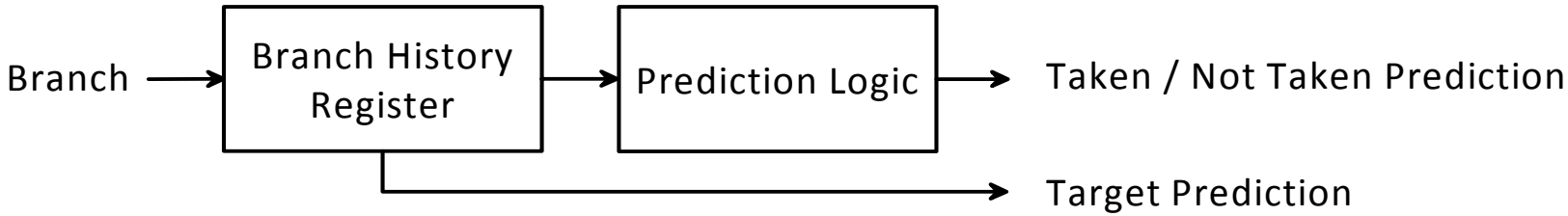
Limitations

Complex hardware

Large branch history register storage

Long prediction time

One-Level Prediction



Branch History Register (BHR)

Index — hash function of branch PC

N bit entry (profile)

N most recent branch outcomes

Target — PC of last taken branch

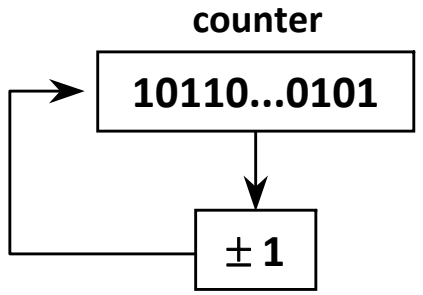
Prediction Logic

n-bit counter ⇒ 2^n levels

Operates on n-bit BHR entry

000…0 = Strongly Not Taken

111…1 = Strongly Taken



Prediction

0xxxx...x = not taken

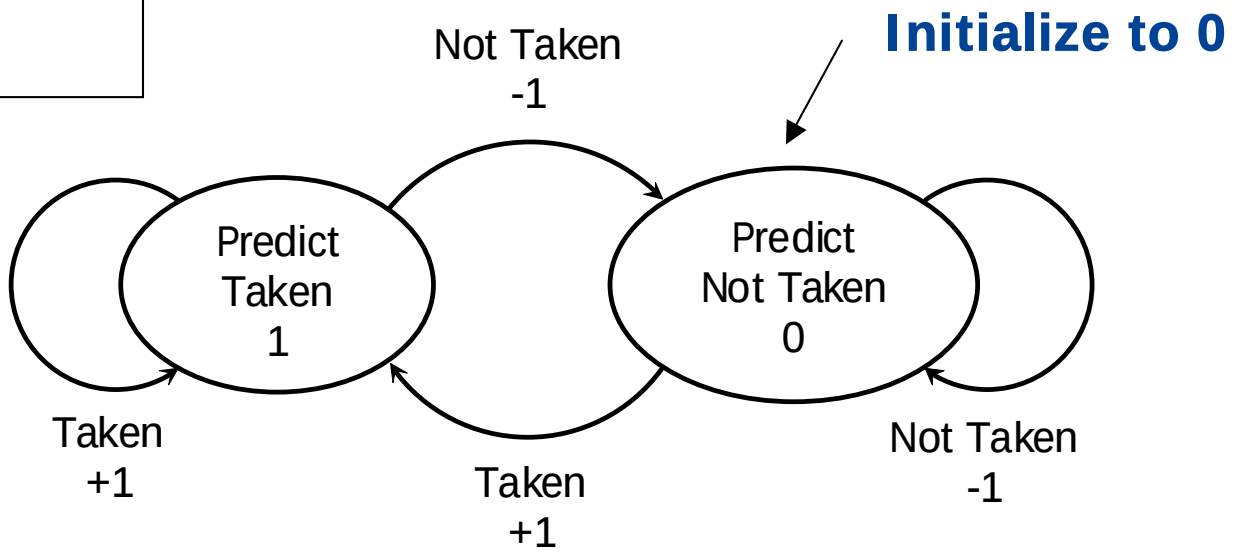
1xxxx...x = taken

Initialize	`counter ← 0`
Taken	`counter++`
Not Taken	`counter--`

1-Bit Saturating Up/Down Counter

1-bit confidence level

Remembers last action only



After validation of branch action

Taken ⇒ **`counter++`**

Not Taken ⇒ **`counter--`**

Prediction based on last action

`Counter = 0` ⇒ not taken

`Counter = 1` ⇒ taken

Predicts single static loops with accuracy of 1 / (loop iterations)

2-Bit Saturating Up/Down Counter

2-bit confidence level

Remembers up to 3 actions

After branch validation

Taken ⇒ **`counter++`**

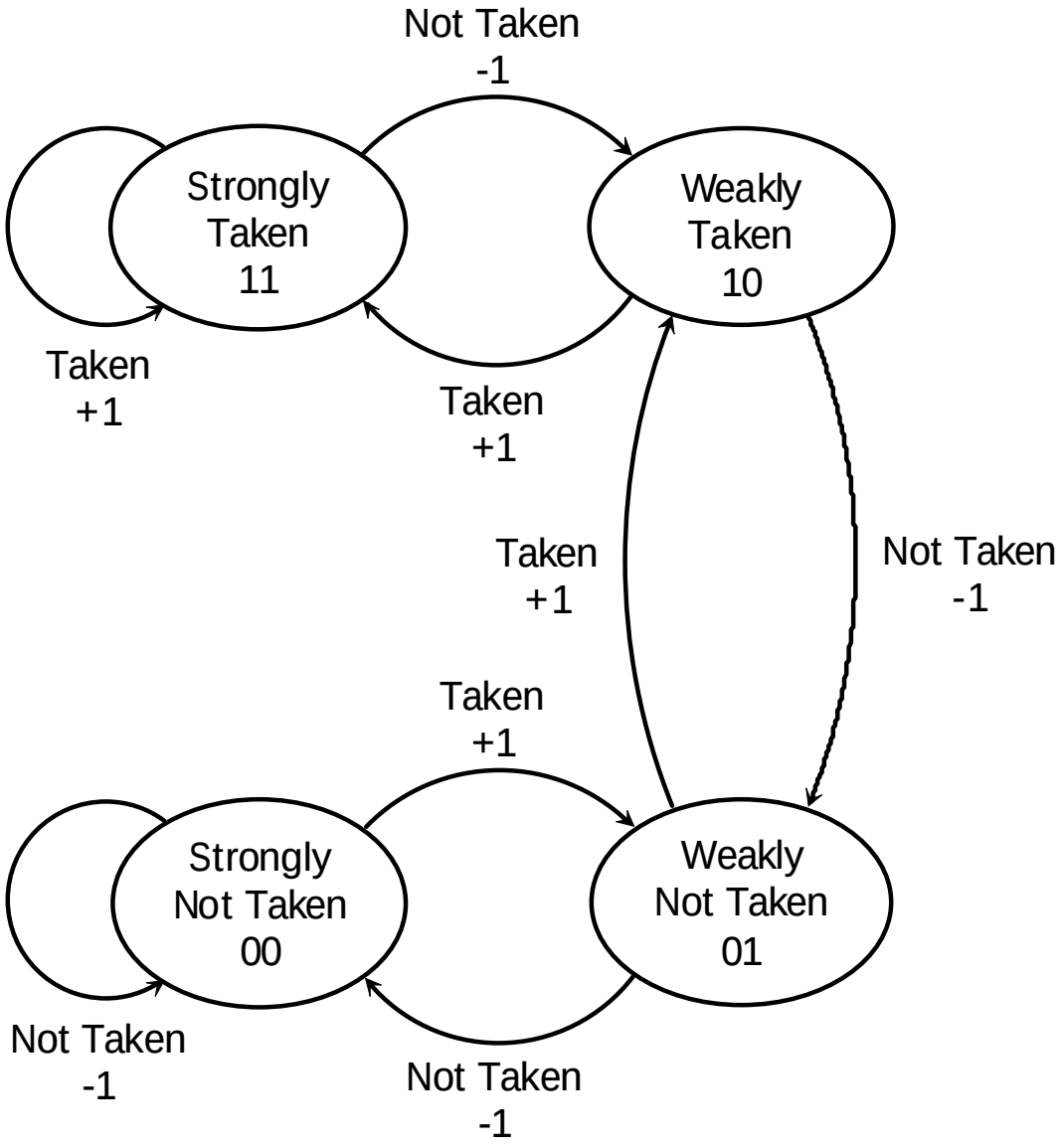
Not Taken ⇒ **`counter--`**

Prediction

`Counter = 0x` ⇒ not taken

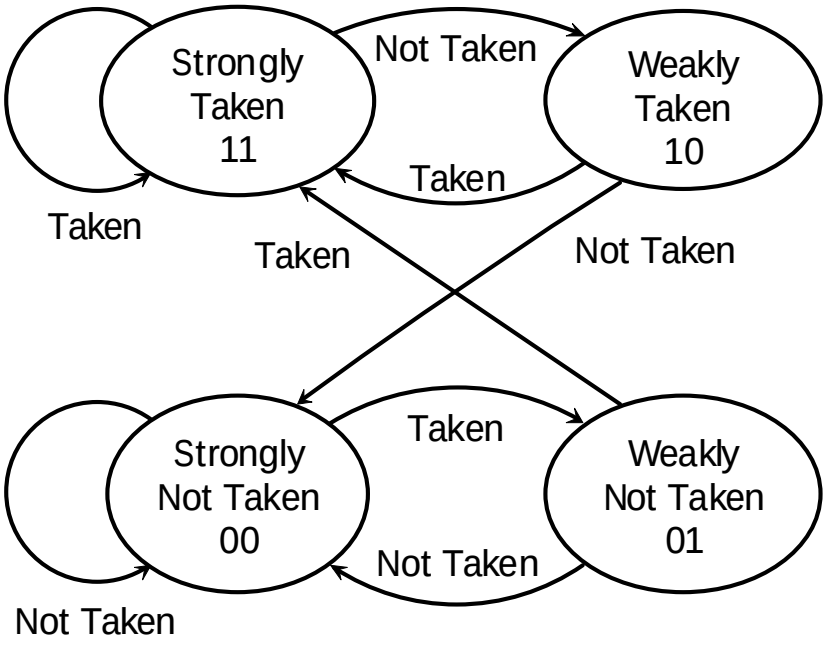
`Counter = 1x` ⇒ taken

Predicts 2 nested loops



Initialize to 00

Alternative 2-Bit Counters

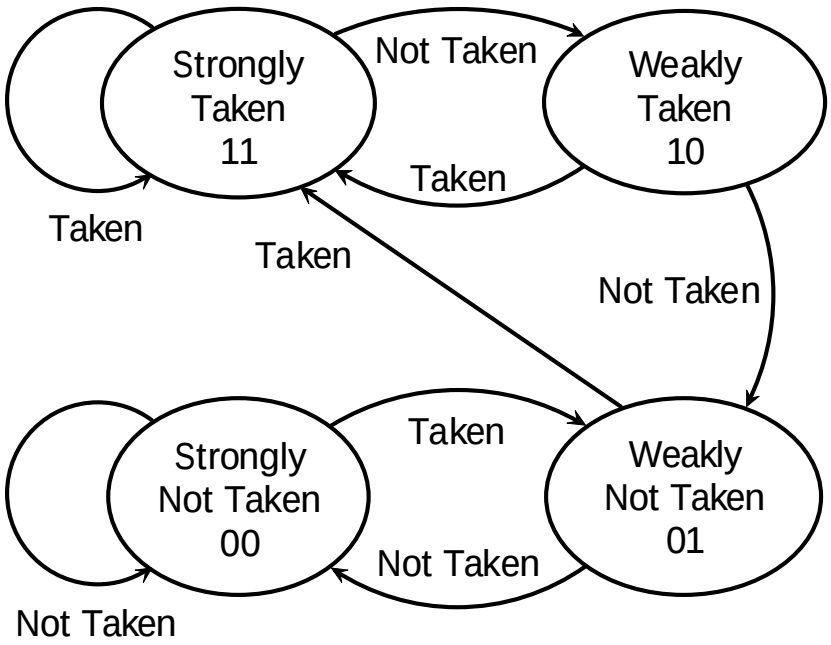


2 consecutive taken branches

Predict strongly taken

2 consecutive not taken branches

Predict strongly not taken



2 consecutive taken branches

Predict strongly taken

Preference to taken branches

Short Nested Loop

```
    ADDI R1, R0, #Len
L1: ADDI R2, R0, #Len
L2: SUBI R2, R2, #1
    BNEZ R2, L2
    SUBI R1, R1, #1
    BNEZ R1, L1
```

Example

Len = 3 global profile

N-bit BHR entries

Branch profile — global branch actions

1 = taken

0 = not taken

11…1011…1011…101…11…1011…100

1 2 3 Len-1 Len

123456789ABC

1101 1101 1100

1 2 3

Branch	1-bit	2-bit	3-bit	4-bit
init	0	00	000	0000
1	1	01	001	0001
2	1	11	011	0011
3	0	10	110	0110
4	1	01	101	1101
5	1	11	011	1011
6	1	11	111	0111

Branch	1-bit	2-bit	3-bit	4-bit
7	0	10	110	1110
8	1	01	101	1101
9	1	11	011	1011
A	1	11	111	0111
B	0	10	101	1110
C	0	00	100	1100

Short Nested Loop

```
     ADDI R1, R0, #L
L1:  ADDI R2, R0, #L
L2:  SUBI R2, R2, #1
     BNEZ R2, L2
     SUBI R1, R1, #1
     BNEZ R1, L1
```

1-bit mispredictions

$1 + (L - 2) \times 2 + 1$

2-bit mispredictions

$2 + (L - 2) \times 1 + 2$

Large L

mispredictions

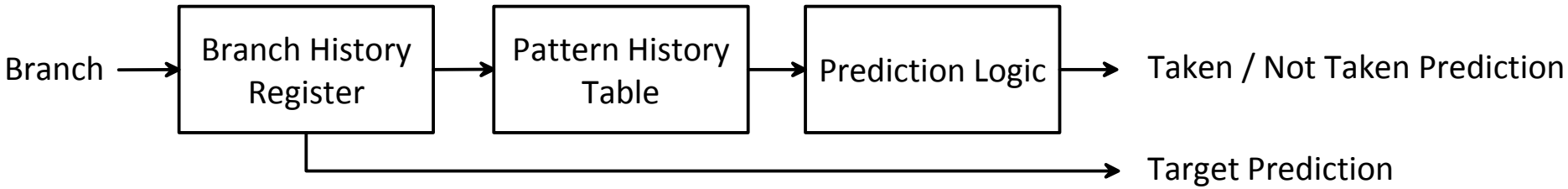
~ L for 2-bit

~ 2L for 1-bit

Counters at predict time
Based on previous Action

R1 (at L2)	R2 (at L2)	Action	1-bit	2-bit
L	L	T	0	00
L	L – 1	T	1	01
L	L – 2 … 2	T	1	10
L	1	NT – T	1 – 0	11 – 10
L – 1	L	T	1	10
L – 1	L – 1	T	1	11
L – 1	L – 2 … 2	T	1	11
L – 1	1	NT – T	1 – 0	11 – 10
L – 2 … 2	L	T	1	10
L – 2 … 2	L – 1	T	1	11
L – 2 … 2	L – 2 … 2	T	1	11
L – 2 … 2	1	NT – T	1 – 0	11 – 10
1	L	T	1	10
1	L – 1	T	1	11
1	L – 2 … 2	T	1	11
1	1	NT – NT	1 – 0	11 – 10

Yeh-Patt Two-Level Adaptive Prediction



Branch History Register (BHR)

Profile — sequence of observed branch actions

Target — PC of last taken branch

Characterizes history of branch behavior

Pattern History Table (PHT)

Pattern — observed next branch action for specific profile

Characterizes pattern of branch behavior given particular history

Prediction

Confidence level = n-bit saturating up/down counter

Yeh and Patt, Alternative Implementations of Two-Level Adaptive Branch Prediction, 1992

Yeh-Patt — BHR

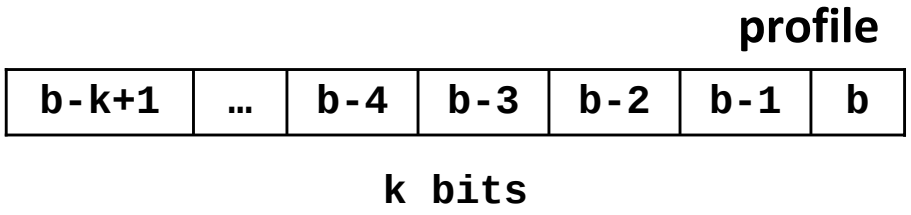
Index

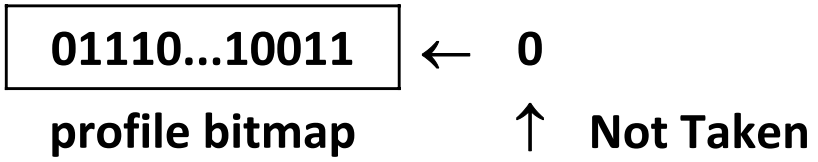
- Global table
  - 1 entry
  - No index
- Local table
  - 1 entry per branch
  - Index = hash function of branch source PC
  - Source address aliasing ⇒ instruction misidentified ⇒ misprediction



Entry

- Profile
  - k-bit bitmap = last k actions of b last branches
  - Taken = 1 / Not Taken = 0
  - Shift in newest action from right
- Target — PC of last taken branch





Yeh-Patt — PHT + Prediction Logic

Index

k-bit profile

Branch action sequence



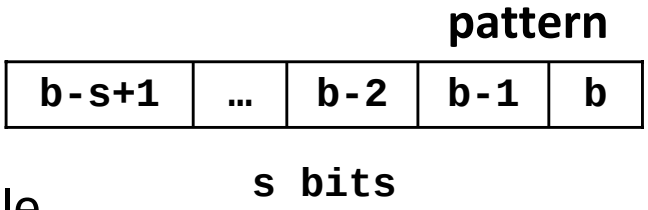
Entry

Pattern

s-bit bitmap = last s (out of b) next actions for profile

Taken = 1 / Not Taken = 0

Shift in newest action from right



Prediction Logic

n-bit saturating up/down counter

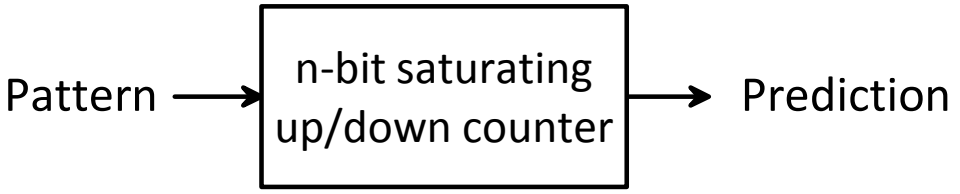
Counts s-bit pattern from PHT (from left to right)

000…0 = Strongly Not Taken —— **111…1** = Strongly Taken

Prediction

0xxxx...x = not taken

1xxxx...x = taken



Yeh-Patt — Simple Example

Profile for short nested loop with Len = 3

k = 2 branch actions (PHR)

s = 1 next action (BHT)

1-bit counter (prediction = last action for pattern)

```
     ADDI R1, R0, #L
L1:  ADDI R2, R0, #L
L2:  SUBI R2, R2, #1
     BNEZ R2, L2
     SUBI R1, R1, #1
     BNEZ R1, L1
```

Actual branch sequence — 110111011100

Branch		Global Profile m		Observed Profiles				Actual Next Action				Predicted Next Action (last action)			
start	—	0	0	00				1				—			
BNEZ R2	1	0	1	00	01			1	1			1	—		
BNEZ R2	1	1	1	00	01	11		1	1	0		1	1	—	
BNEZ R2	0	1	0	00	01	11	10	1	1	0	1	1	1	0	—
BNEZ R1	1	0	1	00	01	11	10	1	1	0	1	1	1	0	1
BNEZ R2	1	1	1	00	01	11	10	1	1	1	1	1	1	0	1
BNEZ R2	1	1	1	00	01	11	10	1	1	0	1	1	1	1	1
BNEZ R2	0	1	0	00	01	11	10	1	1	0	0	1	1	1	1
BNEZ R1	1	0	0												

First observation Repeat = misprediction

Yeh-Patt — Building BHR and PHT

Branch actions for some branch instruction

1	0	1	0	1	0	1	0	1	0	1	0

5-bit BHR (starting from 5th branch)

1	0	1	0	1	0	1	0	1	0	1	0	BHR Entry	next
1	0	1	0	1								10101	0
	0	1	0	1	0							01010	1
		1	0	1	0	1						10101	0
			0	1	0	1	0					01010	1
				1	0	1	0	1				10101	0
					0	1	0	1	0			01010	1

3-bit patterns (PHT)

10101	0	0	0
01010	1	1	1

PC → BHR profile = last 5 branches → PHT entry = 3 Next Actions after profile

PHT entry → Counter → Prediction

Example — Alternating Branches in One-Level Prediction

```
     ADD R1, R0, R0
L1:  ADDI R1, R1, #L
     ANDI R2, R1, #L
     BNEZ R2, L1
     BEQZ R2, L1
```

Misprediction rate for **BNEZ**

1-bit counter — 100%

2-bit counter — 50%

Counting 2-bit BHR entries

Entry 01: counter 00 —> 00 —> 01

Entry 10: counter 00 —> 01 —> 00

Counter **never** counts up to 10

Counter **always** predicts 0

Separate Counters for BNEZ and BEQZ

State	Action		counters at prediction time			
			1-bit BHR (last action)		2-bit counter 2-bit BHR	
	BNEZ	BEQZ	BNEZ	BEQZ	BNEZ	BEQZ
1	1		0		00	
2	0	1	1	0	01	00
3	1		0		00	
4	0	1	1	1	01	01
5	1		0		00	
6	0	1	1	1	01	10
7	1		0		00	
8	0	1	1	1	01	11
9	1		0		00	
10	0	1	1	1	01	11
11	1		0		00	
12	0	1	1	1	01	11

For **BNEZ** 0 is correct half the time

Prediction

1x = taken

0x = not taken

Example — Alternating Branches with Y-P

```
        ADD R1, R0, R0
L1:     ADDI R1, R1, #L
        ANDI R2, R1, #L
        BNEZ R2, L1
        BEQZ R2, L1
```

4-bit BHR

2-bit PHT for each BHR pattern

	1	0	1	0	1	0	1	0	1	0	1	0	BHR	PHT
Learning For BNEZ	init												0000	00
	1												0001	00
	1	0											0010	00
	1	0	1										0101	00
	1	0	1	0									1010	00
		0	1	0	1								0101	00
			1	0	1	0							1010	01
Steady State For BNEZ				0	1	0	1						0101	00
					1	0	1	0					1010	11
						0	1	0	1				0101	00
							1	0	1	0			1010	11
								0	1	0	1		0101	00

Example — Alternating Branches with Y-P

```
        ADD R1, R0, R0
L1:     ADDI R1, R1, #L
        ANDI R2, R1, #L
        BNEZ R2, L1
        BEQZ R2, L1
```

k = 4-bit BHR

s = 2-bit PHT for each BHR pattern

Prediction function

Output of 2-bit counter on PHT bits

00 ⇒ 00 → 00 → 00

01 ⇒ 00 → 00 → 01

10 ⇒ 00 → 01 → 00

11 ⇒ 00 → 01 → 10

0x = Not Taken

1x = Taken

Correctly predicts every branch from 8th branch run

Steady state sequence for **BNEZ**

BHR:PHT Alternate 0101:00 / 1010:11

Data at Prediction Time

State	Action		BHR	PHT	Y-P
	BNEZ	BEQZ			BNEZ
1	1		0000	00	–
2	0	1	0001	00	–
3	1		0010	00	–
4	0	1	0101	00	–
5	1		1010	00	-
6	0	1	0101	00	0
7	1		1010	01	0
8	0	1	0101	00	0
9	1		1010	11	1
10	0	1	0101	00	0
11	1		1010	11	1
12	0	1	0101	00	0

Steady state Y-P misprediction rate for **BNEZ** — 0%

Another Example

Branch actions for some branch instruction

1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	1

k = 5-bit BHR (from 5th branch run)

1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	1	1
1	0	1	0	1												
	0	1	0	1	0											
		1	0	1	0	1										
			0	1	0	1	1									
				1	0	1	1	1								
					0	1	1	1	0							
						1	1	1	0	1						
							1	1	0	1	0					
								1	0	1	0	1				
									0	1	0	1	0			
										1	0	1	0	1		
											0	1	0	1	1	
												1	0	1	1	1

s = 3-bit PHT

10101	1	0	1
01010	1	1	1
01011	1	1	1
10111	0	0	0
01110	1	1	1
11101	0	0	0
11010	1	1	1

Misprediction rate

(Repeating profile 10101011)

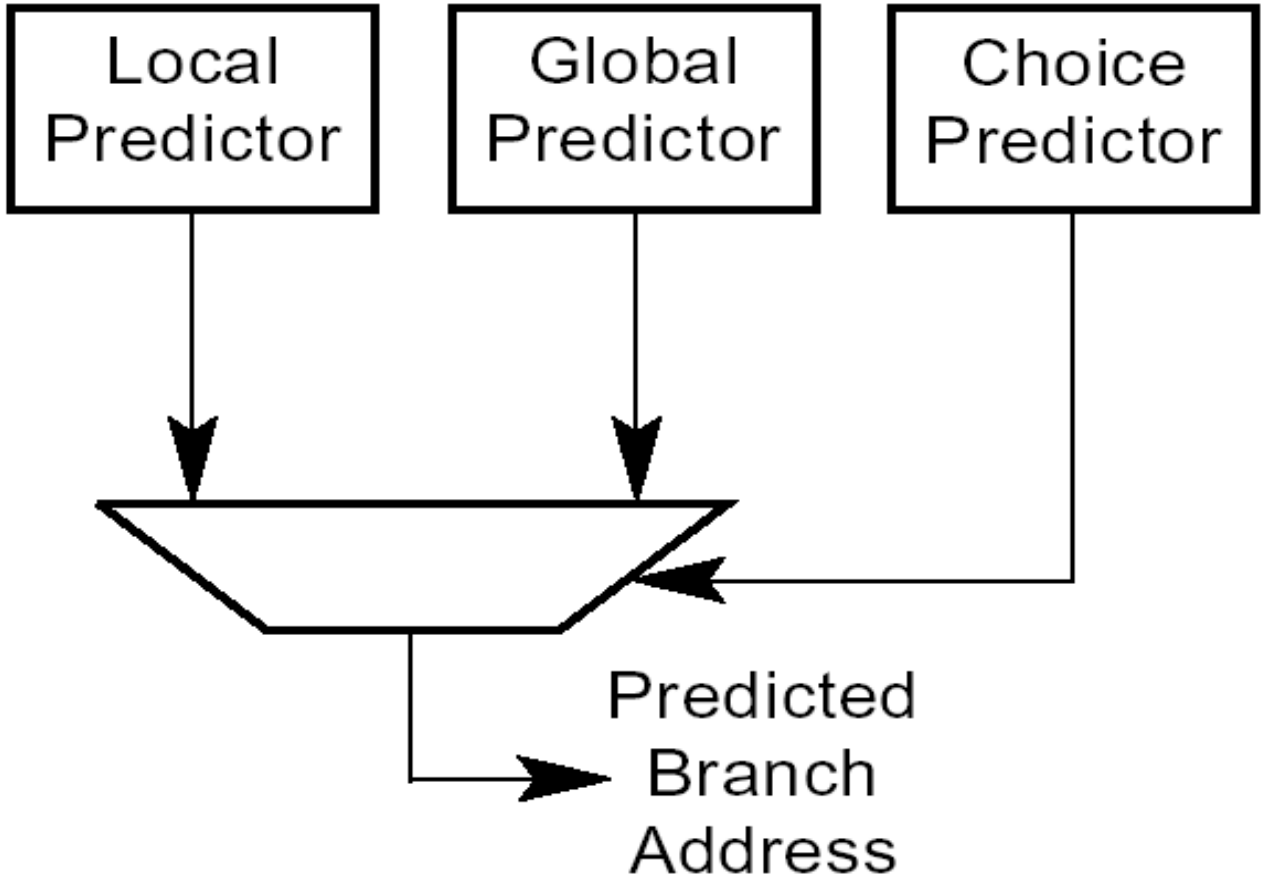
10101 50% (always predicts 0)

other 0%

PHT entry for 10101 is either 101 or 010

Counter (101) = 00—>01—>00—>01 Counter (010) = 00—>00—>01—>00

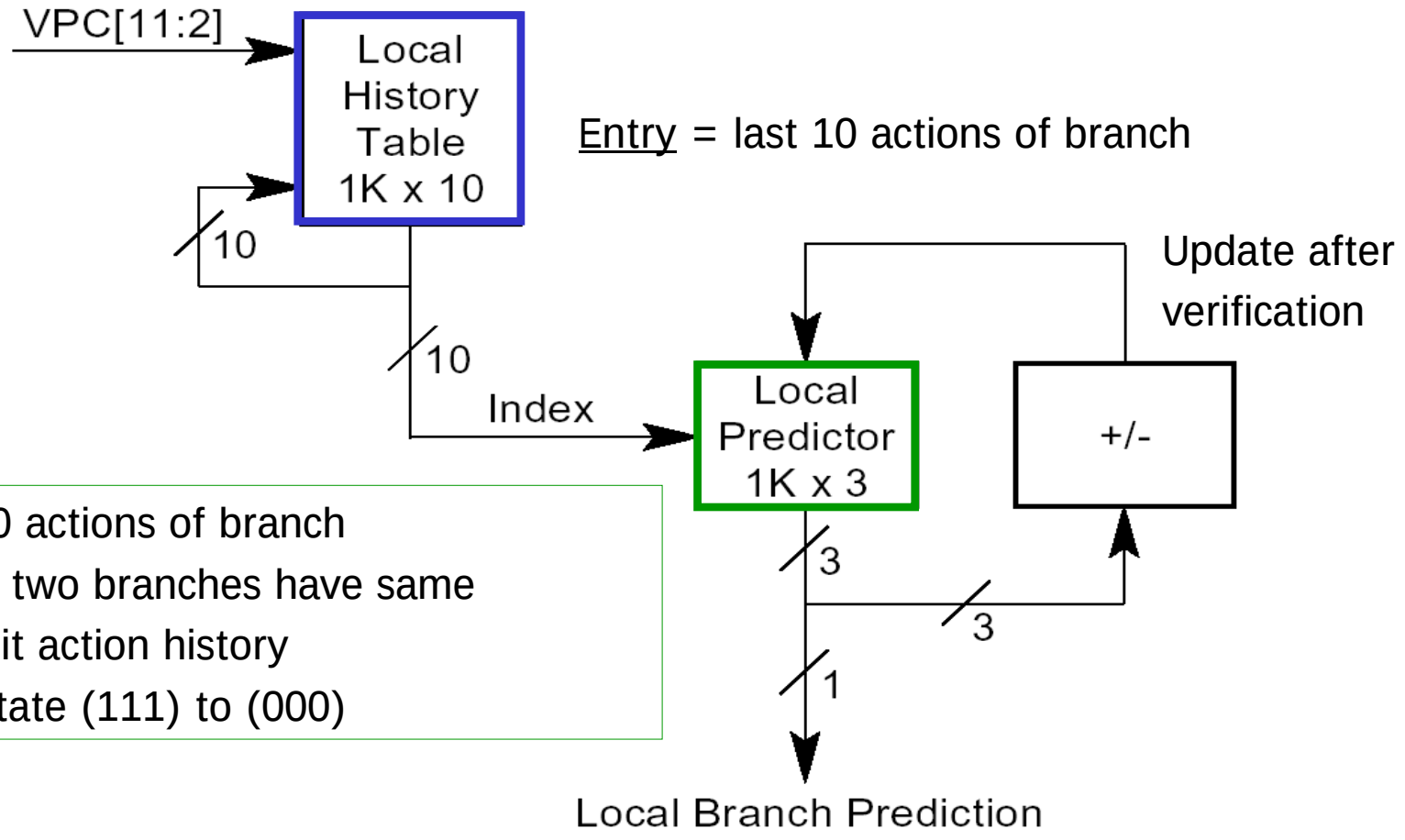
DEC Alpha Branch Predictor — 1



Ref: 21264/EV68A Microprocessor Hardware Reference Manual, 1992

Two-Level Local Predictor — 2

Index: Branch source address bits 2 to 11 (10 bits ⇒ 2^{10} = 1024 entries)
Instruction addresses aligned on 4-byte (2-bit) boundaries



Entry = last 10 actions of branch

Index: Last 10 actions of branch
Alias if two branches have same 10-bit action history

Entry: 3-bit state (111) to (000)

Predictor = (0xx) ⇒ not-taken
Predictor = (1xx) ⇒ taken

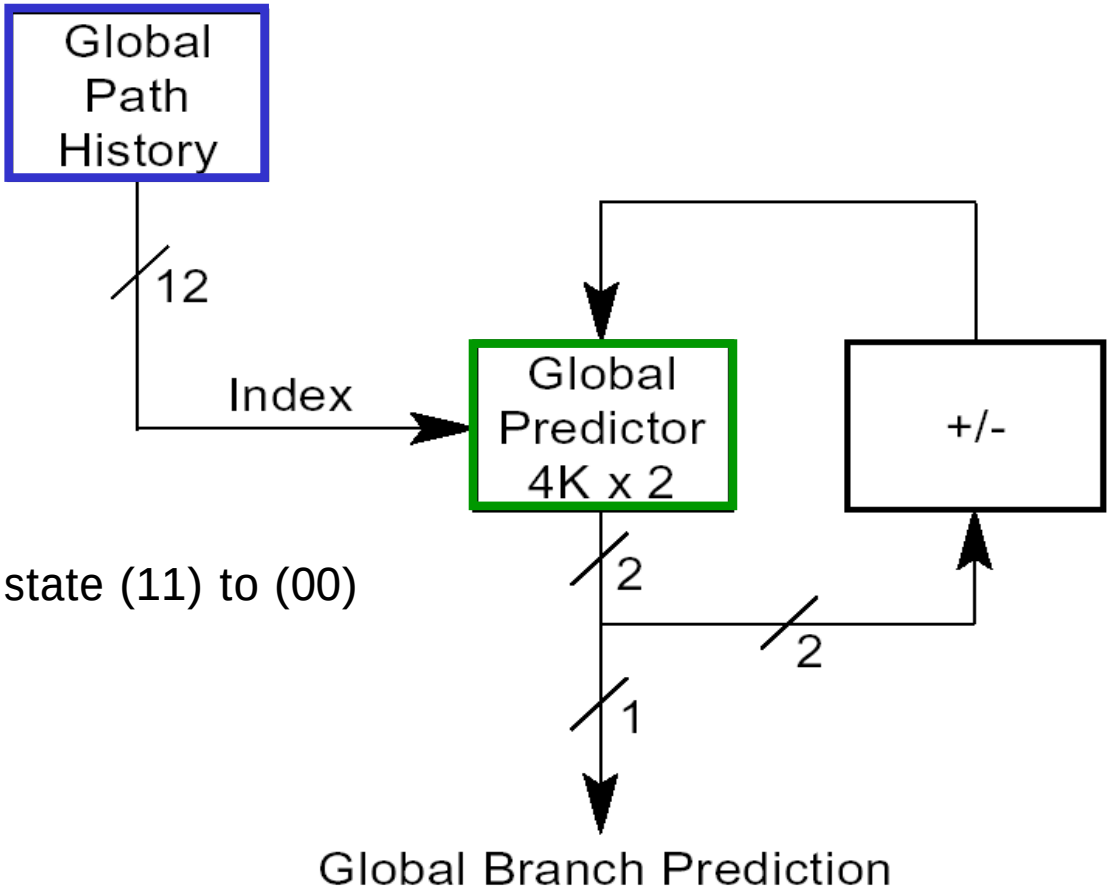
Global Predictor — 3

Last 12 actions of all branches

2^{12} = 4096 possible action histories

Index: Last 12 actions of all branches

Entry: 2-bit state (11) to (00)



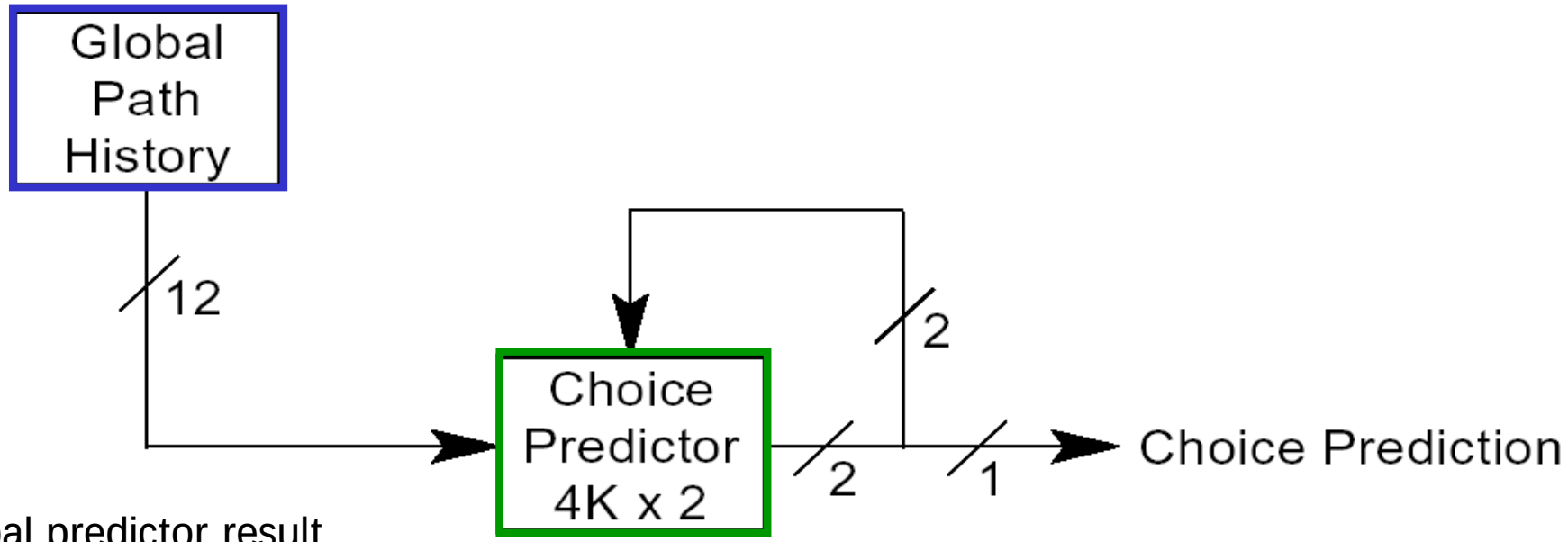
Prediction: Predictor = (0x) ⇒ not-taken

Predictor = (1x) ⇒ taken

Choice Predictor — 4

Last 12 actions of all branches

2^{12} = 4096 possible action histories



Global predictor result

1 = correct

0 = misprediction

Prediction: (0x) chooses Local Predictor

(1x) chooses Global Predictor

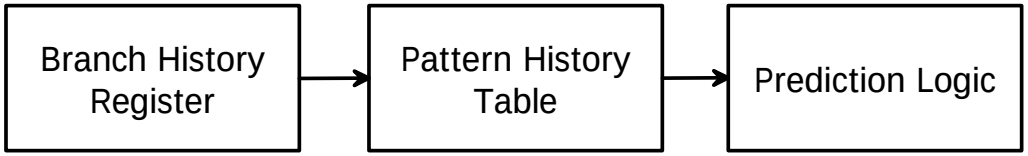
Yeh-Patt Versus Alpha

Yeh-Patt model

Stage 1 — store k-bit profile (taken / not taken history) in BHR

Stage 2 — store s-bit pattern (next action history) in PHT

Stage 3 — count over stored pattern



Alpha model

Local

Stage 1 — store k = 10 bit profile

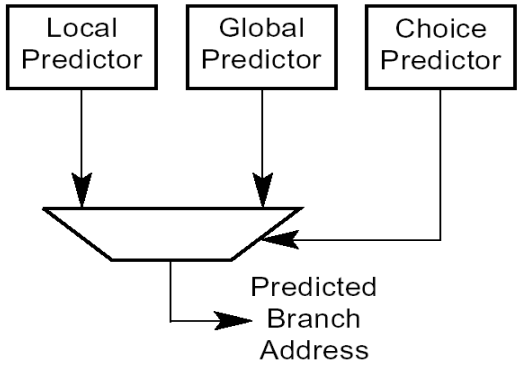
Stage 2 — count over all next actions for profile

Global

Stage 1 — store k = 12 bit profile

Stage 2 — count over all next actions for profile

Choice predictor — local / global



Event Horizon in Counter

Event horizon

Prediction on s-bit pattern from PHT ≠ counting all next actions

Example

Branch history = **11 11110 11110 11110 11110...**

Next actions for profile **11**

1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0
1	1	1														
	1	1	1													
		1	1	1												
			1	1	1											
				1	1	0										
							1	1	1							
								1	1	1						
									1	1	0					
												1	1	1		
													1	1	1	
														1	1	0

Compare

2-bit counter on all next actions

2-bit counter on 2-bit pattern

nexts		1	1	1	1	0	1	1	0	1	1	0	1	1
count	00	01	10	11	11	10	11	11	10	11	11	10	11	11
patterns	00	01	11	11	11	10	01	11	10	01	11	10	01	11
count	00	01	10	10	10	01	01	10	01	01	10	01	01	10
outcome		1	1	1	0	1	1	0	1	1	0	1	1	0

Address Prediction

Branch Target Buffer (BTB)

- Organized as cache with LRU eviction
- Integrated with action predictor or separate table

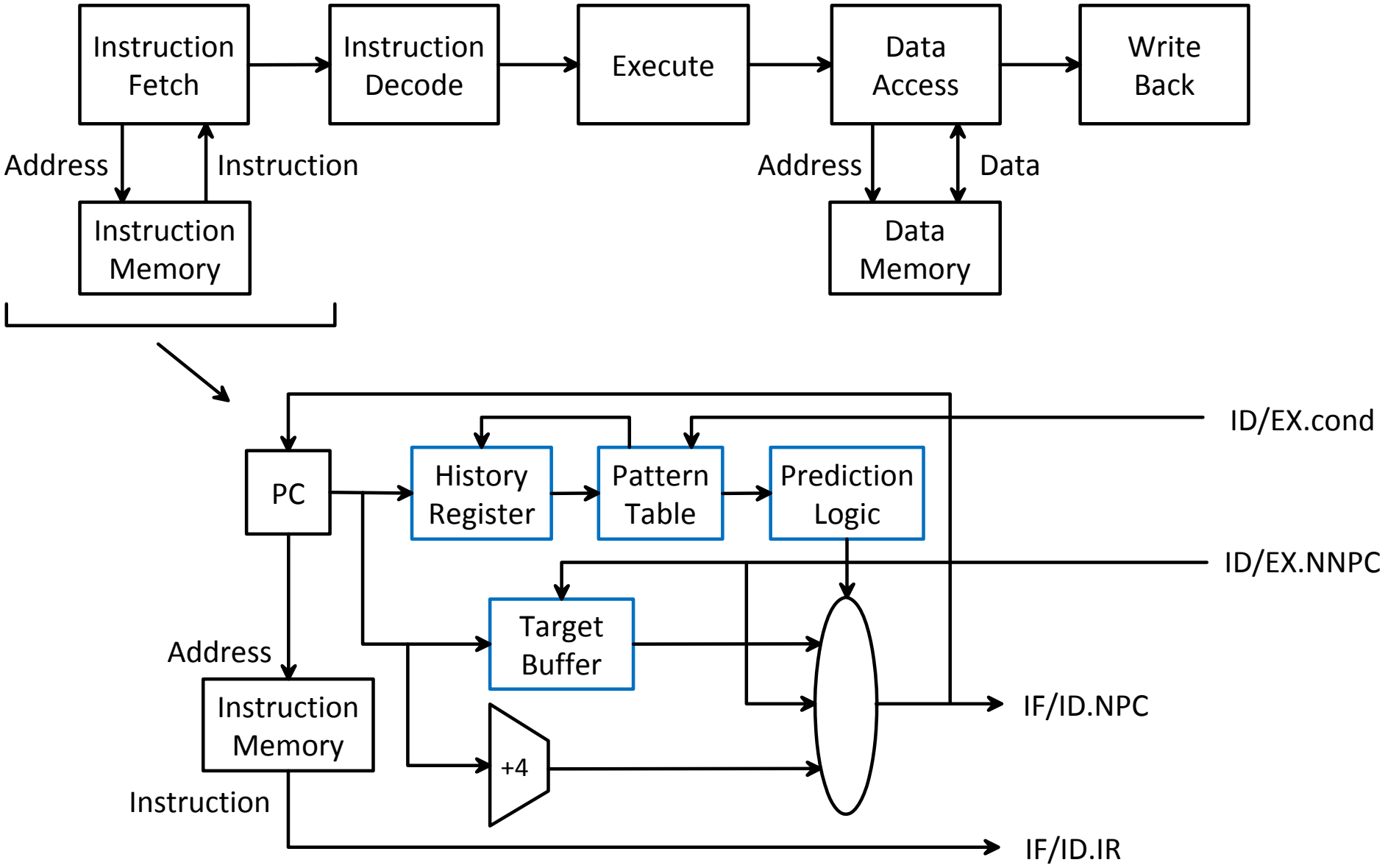
Index

- Branch PC

Entry

- Last verified target address
- Valid
  - Initialized to 0
  - Changed when entry is written
- Special
  - Indicates low confidence in predicted address

DLX Pipeline with Y-P Branch Prediction



Low Confidence Address Predictions

Indirect branches

x86	**JMP [ESI]**	**EIP ← Mem[Regs[ESI]]**
DLX	**JR R1**	**PC ← Regs[R1]**

Pentium M caches target for indirect branches

Target prediction made from cache of register instead of last jump

Return from subroutine

Function call, trap, interrupt

Lowest confidence branch addresses

Enter subroutine from multiple departure points

Each subroutine call has private return point

x86	**CALL [ESI]**	**PUSH EIP** **EIP ← Mem[Regs[ESI]]**
	RET	**POP EIP**
DLX	**JALR R2, offset**	**Regs[R2] ← PC**
	JR R2	**PC ← Regs[R2]**

Predicting Subroutine Return Addresses

x86 Call/Return instructions

CALL pushes return address onto stack

First run of **RETURN** misses in branch history cache

Store entry after verification with Special Field

Subsequent runs of **RETURN** pre-load stack entry for target address

RISC JALR/JR instructions

Recognize subroutine call

Encounter instruction **JR R2**

R2 points to instruction immediately below **JALR R2**, **`offset`**

Mark **JALR** instructions as Probable Call in BTB entry

Create BTB entry for **JALR** target (from subroutine)

Mark entry as Subroutine Entry Point (SEP)

Target address = pointer to **JR** return instruction

On **JALR** hit at PC

Target field of **JR** pointed to by SEP ← PC + 4

Returning **JR** has correct target address

Exotic Branch Mispredictions

Exotic branch conditions

Simple branch prediction has accuracy ≈ 90%

Typically

- 80% of branches ~ 100% accuracy
- 20% of branches ~ 50% accuracy

Least predictable branches — on-time test of single operand

x86 — **`CMP EAX, immediate`** followed by **`JZ offset`**

DLX — **`BEQZ R1, offset`**

Special treatment for tested operands (proposed)

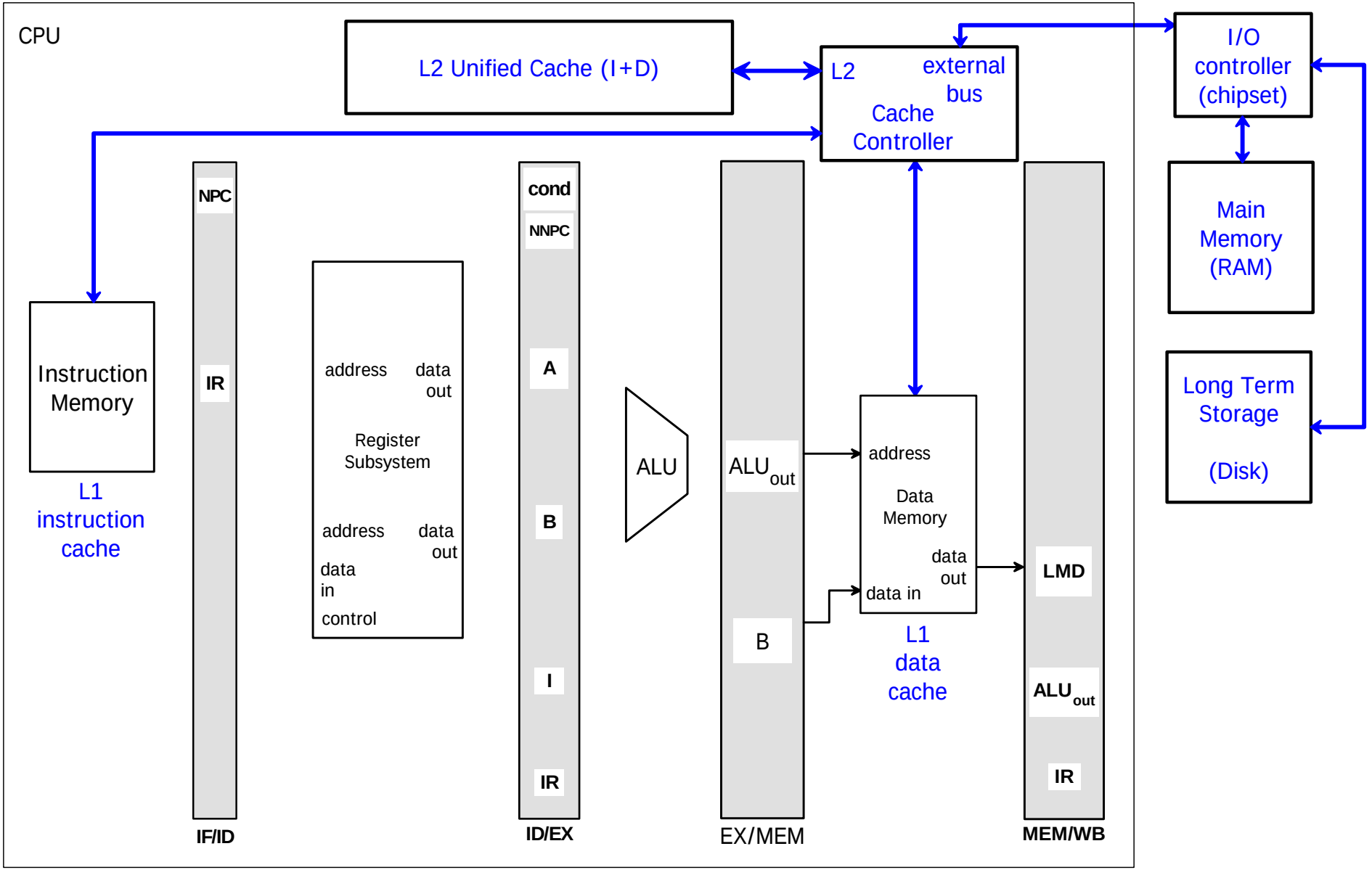
Maintain cache of tested operands and test

- Entry for **`BEQZ R1`** caches value of **`R1`** and output of **`(R1 == 0)`**

On update to operand that hits in table

- Update table entry
- Update action prediction for associated branch instruction

DLX Memory Hierarchy



CPU and Memory Hierarchy

CPU controller accesses L1 cache

if (address in L1 cache) {access performed in 1 clock cycle}

else {

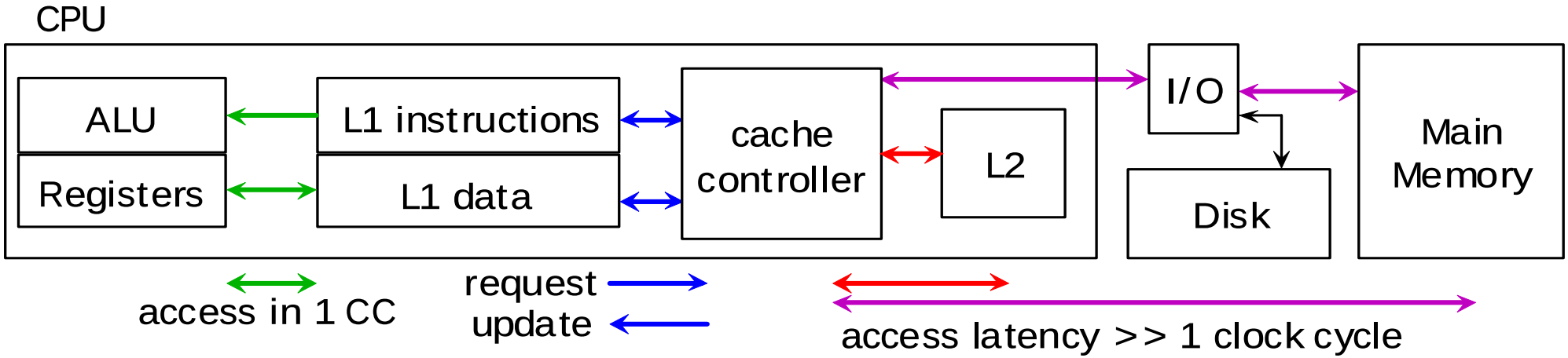
L1 cache accesses cache controller

cache controller initiates access to L2 and main memory

if (address in L2 cache) {controller copies contents to L1 from L2}

else {controller copies location to L1 from main memory}

}



Cache miss penalty

Address not in L1 ⇒ delay in memory access

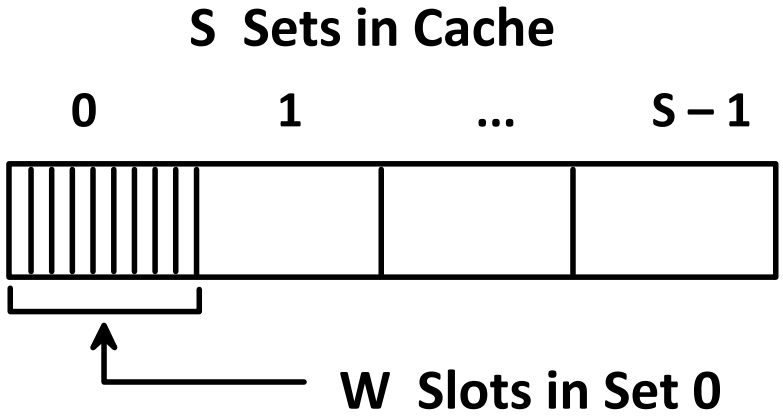
Cache Organization — 1

Copy memory data to cache

How much data to copy?

Which data is in cache?

How to find data in cache?



Copy DATA LINE (block) to cache

Line = **B** bytes

Page size (virtual memory) = integer × block size

Sets and Slots

Cache = **S** sets

Set = **W** slots

Slot = **1** data line

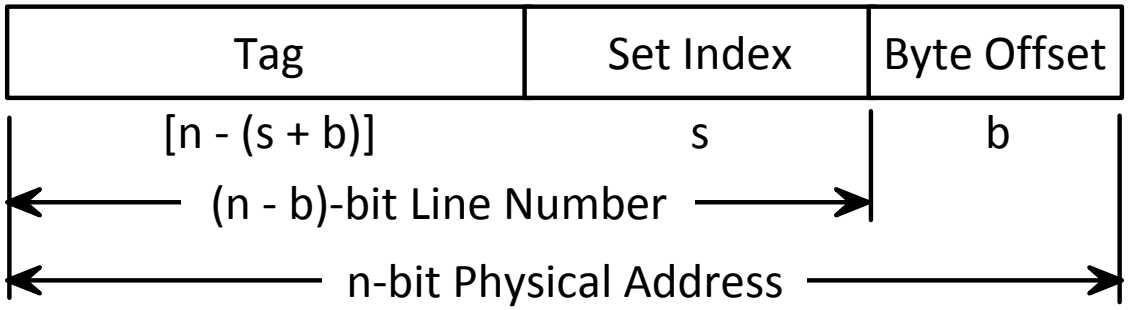
Copy memory line to:

- Specific Set
- Any Slot in Set

Cache Organization — 2

Physical address space

N = 2^n bytes



Lines

B = 2^b bytes / line

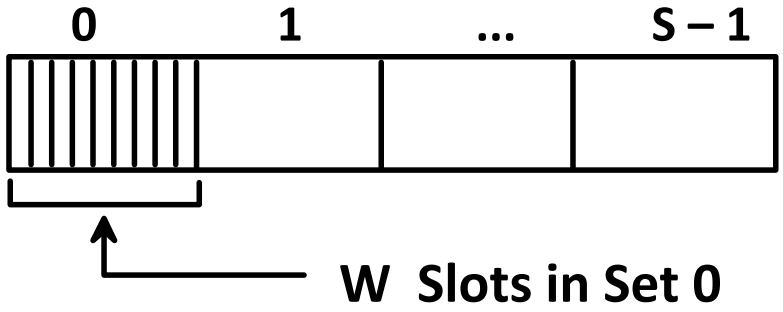
Line number = int (Address / B)

Sets

S = 2^s sets

Set index = s bits

Line → line number % S



W-way associativity

Line → any slot in set

Identify line by Tag field in cache directory

Total Cache Size

$$\text{total cache size} = S \times W \times B = \frac{\text{sets}}{\text{cache}} \times \frac{\text{blocks}}{\text{set}} \times \frac{\text{bytes}}{\text{block}} = \frac{\text{bytes}}{\text{cache}}$$

Cache Definitions and Policies

Cache hit

CPU accesses line in cache

Cache miss

CPU accesses line not in cache

Read miss

Load line to cache

Write miss

Write allocate — load line to cache

No write allocate — write to main memory (no load to cache)

Swapping out cache line

Need new line in full set

Remove least recently used (LRU)

WRITE BACK — update RAM on line swap-out

WRITE THROUGH — update RAM on every write to cache

Cache Performance Issues

L1 Cache hit

CPU memory access = 1 clock cycle

L1 Cache Miss

CPU stalls — L1 loads missing line

Miss Rate

Cache misses per cache accesses

Instruction fetch, load, store

Depends on cache size and organization

Miss Penalty

Cache update latency

Stall cycles while cache ← missing line

Performance of Two-Level Unified Cache

Parameters

M_{L1}^{I} = Instruction Miss Rate at L1

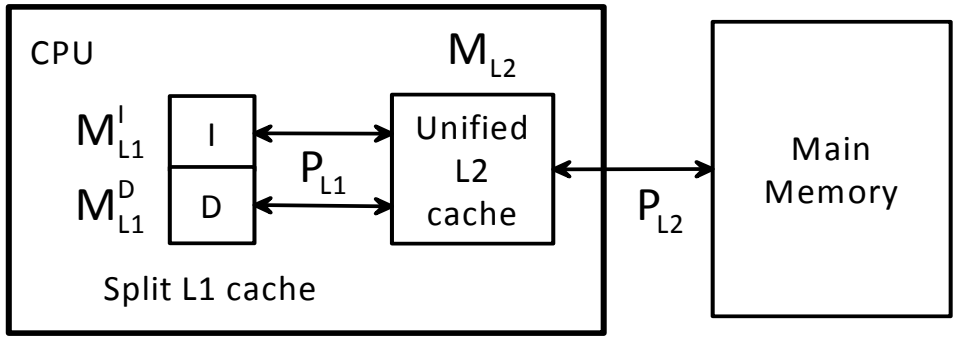
$1 - M_{L1}^{I}$ = Instruction Hit Rate at L1

M_{L1}^{D} = Data Miss Rate at L1

$1 - M_{L1}^{D}$ = Data Hit Rate at L1

M_{L2} = Miss Rate at L2

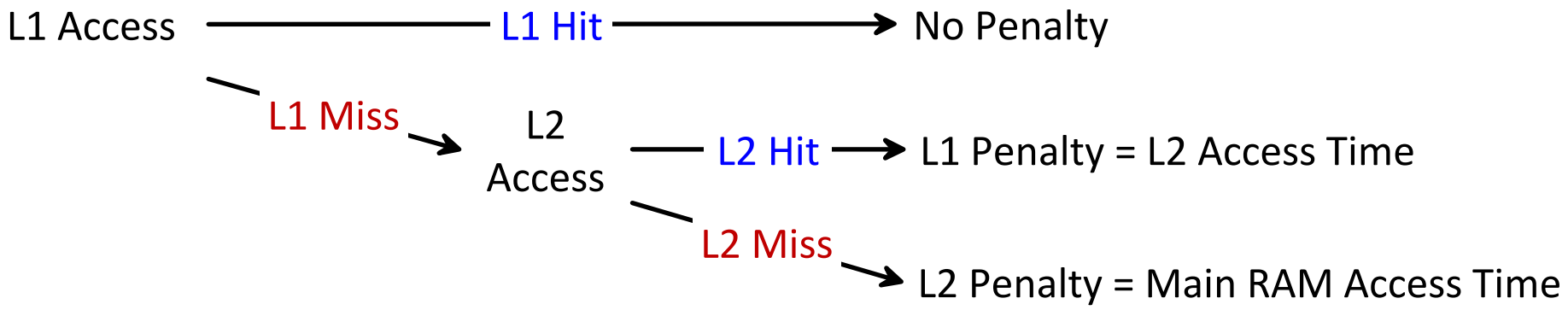
$1\text{-}M_{L2}$ = Hit Rate at L2



P_{L1} = Miss Penalty at L1 $\qquad$ P_{L2} = Miss Penalty at L2

L1 Instruction Access per Instruction = 1

$$\text{L1 Data Access per Instruction} = \frac{IC^{\text{data access}}}{IC}$$



$$CPI_{\text{cache}}^{\text{stall}} = \frac{\text{cache miss stall cycles}}{\text{instruction}} = \left[M_{L1}^{I} + M_{L1}^{D} \times \frac{IC^{\text{data access}}}{IC} \right] \times \left(P_{L1} + P_{L2} \times M_{L2} \right)$$

1-Level Cache versus 2-Level Cache

1 level cache

$$\mathrm{CPI}^{\text{stall}}_{\text{1-level}} = \left[M^{I}_{L1} + M^{D}_{L1} \times \frac{IC^{\text{data access}}}{IC} \right] \times \underbrace{\text{Penalty}^{\text{L1-to-RAM}}}_{\text{50 clock cycles}}$$

2 level cache

$$\mathrm{CPI}^{\text{stall}}_{\text{2-level}} = \left[M^{I}_{L1} + M^{D}_{L1} \times \frac{IC^{\text{data access}}}{IC} \right] \times \underbrace{\left(\underbrace{\text{Penalty}^{\text{L1-to-L2}}}_{\text{5 clock cycles}} + \underbrace{M_{L2}}_{<0.01} \times \underbrace{\text{Penalty}^{\text{L2-to-RAM}}}_{\text{50 clock cycles}} \right)}_{<5.5 \text{ clock cycles}}$$

Issues Affecting Miss Rate

Compulsory miss

- Demand access update
  - Cache ← line on first access
  - First access to line ⇒ cache miss
- Not affected by cache size or associativity

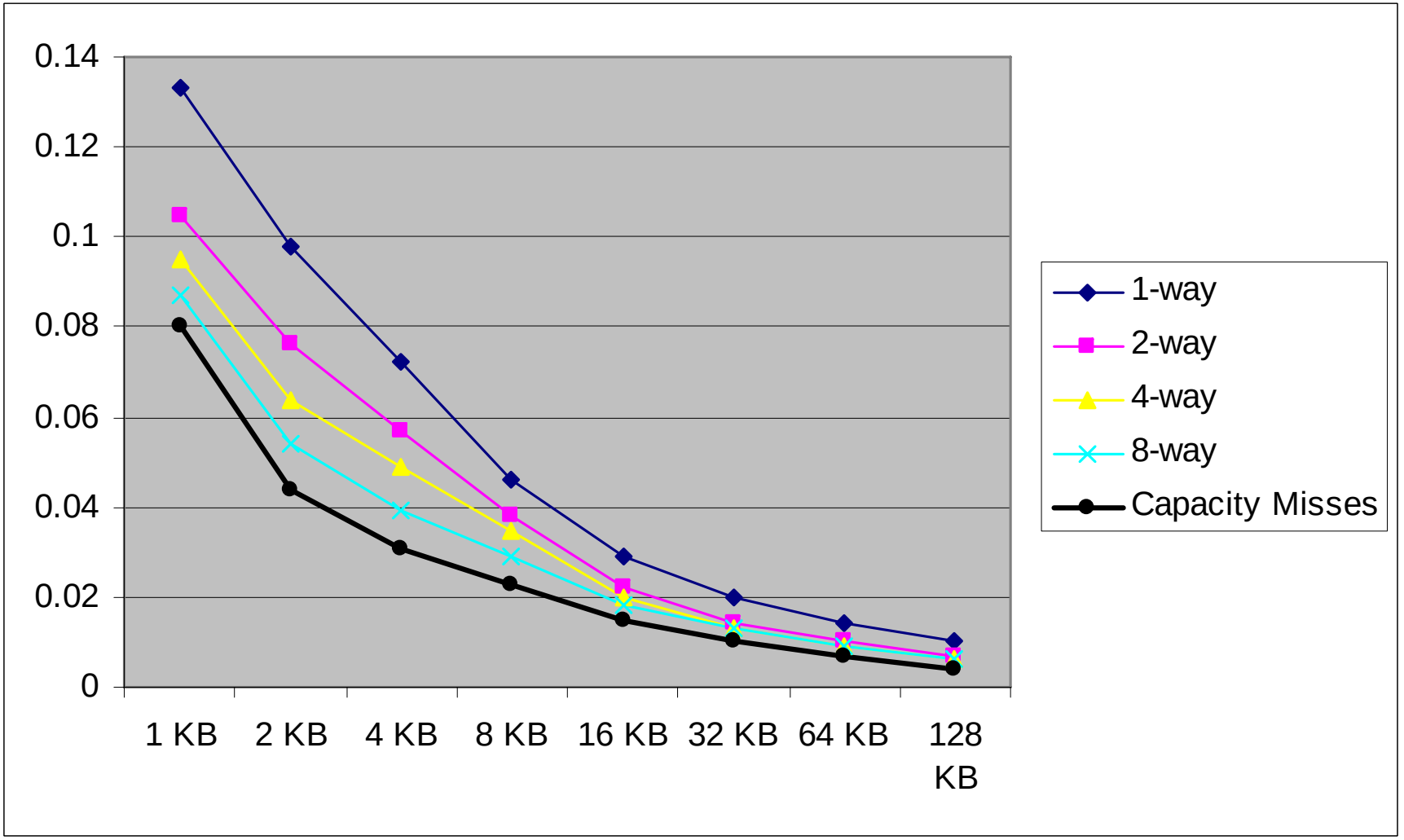
Capacity miss

- Swap out lines for required load ⇒ miss
- Cache < main memory
- Larger cache ⇒ lower capacity miss rate

Conflict miss

- Swap out lines for required load ⇒ miss
- Line ← specific set
  - Spare capacity in other sets
  - Example — address aliasing
- Higher associativity ⇒ lower conflict miss rate

Miss Rate



Total miss rate drops as capacity or associativity increases

Associativity Trade-Off

Any slot in set ← line

Find line ⇒ search all tags

Larger associativity ⇒ more blocks per set ⇒ longer search time

For fixed cache capacity = S × W × B

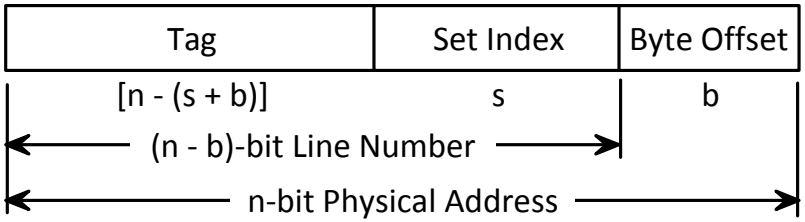
Larger associativity W

⇒ fewer sets S

⇒ smaller set index s

⇒ larger tag size n – (s + b)

⇒ longer tag search

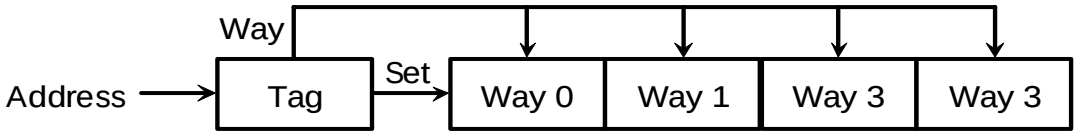


Small advantage beyond 4-way associativity

	s	n – (s + b)	Tag Search Length
Direct mapped	n – b	0	$2^0 = 1$
Fully associative	0	n – b	2^{n-b}

Tag Search Alternatives

Serial access



Address → tag controller

Hit ⇒ memory ← set number / way number

Prepare tagged line for access

Longer latency at lower power

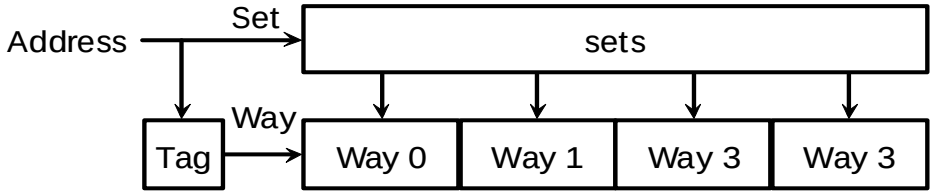
Parallel access

Address → tag controller + set controller

Prepare set for access

Hit ⇒ memory ← way number

Shorter latency at higher power



Way prediction

Address fed to tag controller and way predictor

Hit ⇒ way predictor feeds set number and way number to memory

Misprediction ⇒ tag controller feeds correct way number on next CC

Cache Prefetching

Prefetching "hides memory latency"

- Predict cache update before demand miss
- Initiate cache update request
- Bring memory block to cache before needed
- Memory latency does not cause delay

Data cache prefetching

- Useful for sequential accesses to large data structures
- Software prefetch
  - Assembly language prefetch instructions for cache update
  - Inserted by programmer or compiler
- Hardware — stride buffer predicts sequential accesses

Instruction cache prefetching

- Hardware-based prefetch mechanisms
- Predict instruction cache updates based on program structure

Data Cache Latency Example

Loop over large data structure

Accesses 1024 × 1024 SSE (16-byte vector) operands = 16 MB

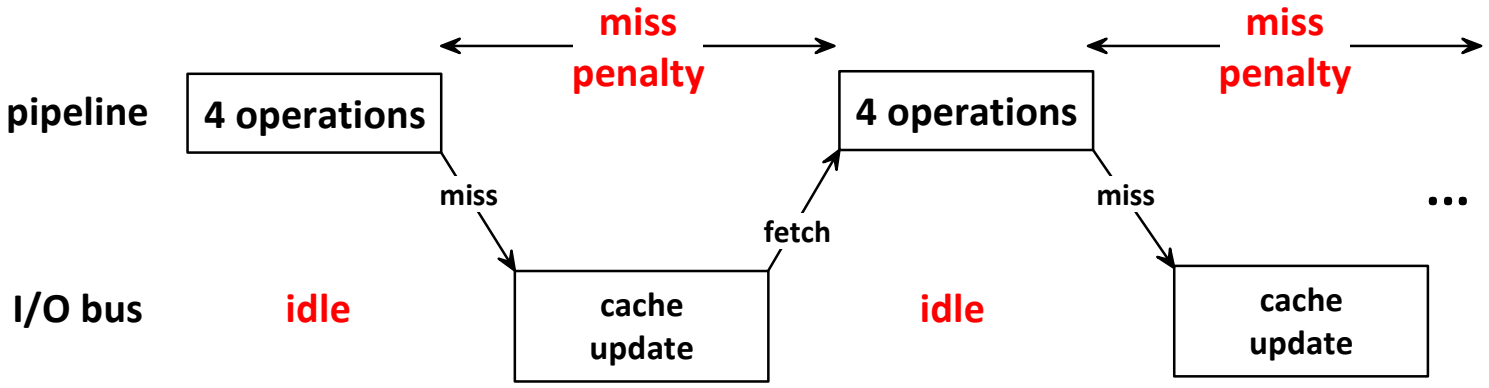
Sequential vector reads — no repeated access to same data

```
for (i = 0; i < 1024; i++) {
     for (j = 0; j < 1024; j++) {
          SSE_vector_operation a[i][j];
     }
}
```

Pentium 4 with 16 KB L1 cache and 64-byte cache line

Data miss every 64 bytes = 4 SSE operands

Data miss in L1 1024 × 1024 / 4 = 256K compulsory misses



Data Cache Prefetch Instruction

Software prefetch loads 128 bytes (2 cache lines)

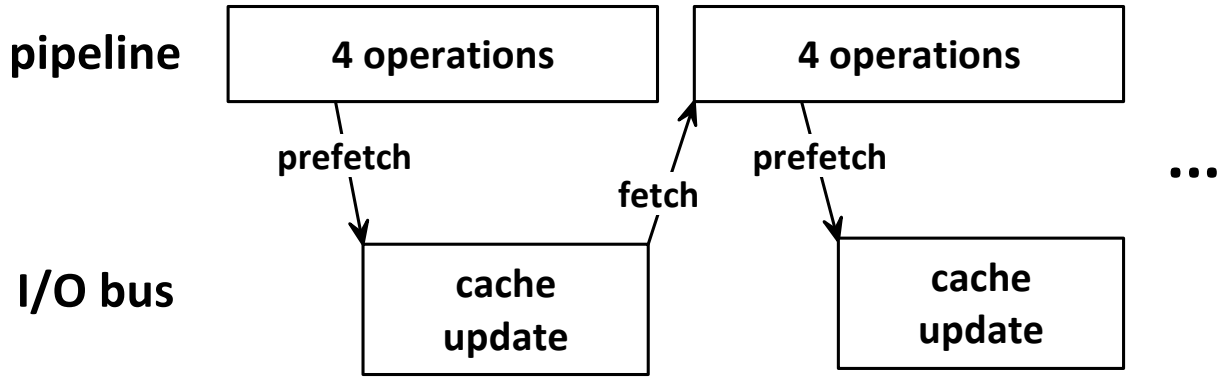
128 bytes = (128/16) = 8 SSE operands

Prefetch 8 operands forward

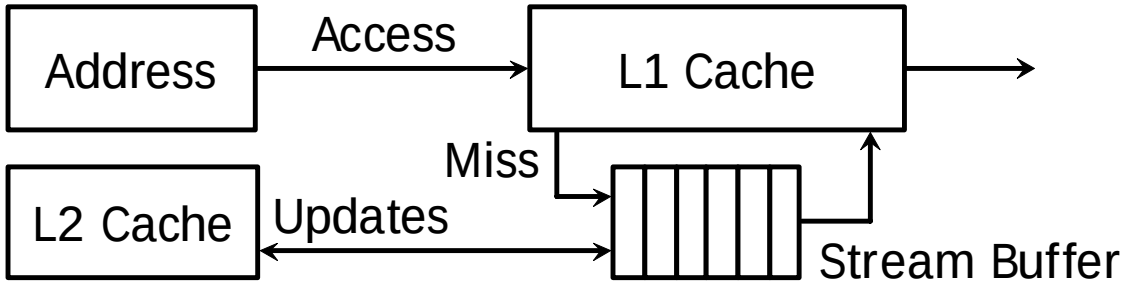
```
for (i = 0; i < 1024; i++) {
     for (j = 0; j < 1024; j++) {
          prefetch a[i][j+8];
          SSE_vector_operation a[i][j];
     }
}
```

Prefetch filtering

NOP on prefetch of cache hit



General Stream Buffer



Stream

Successive accesses to sequential cache lines

Stream buffer

FIFO buffer

Generates + buffers update requests for successive cache lines

Request buffer implementation

Requested line → cache from higher memory level

Line buffer implementation

Requested line → stream buffer until demand miss

Request filtering

Stream buffer skips lines already in cache to minimize prefetches

Stream Buffer Operation

Cache miss to data line L on CC k

CC	Request Buffer	Line Buffer
k	Stream buffer allocated to missed line **L** Buffer generates, buffers, and issues cache request for line L Pipeline suffers miss penalty for line **L**	
k + 1, 2, 3, …	Stream buffer generates, FIFO-buffers, and issues cache request for lines **L + 1, 2, 3, …**	
k + miss penalty	Line **L** brought to cache	Line **L** brought to cache
k + miss penalty + 1	Line **L + 1** brought to cache	Line **L + 1** brought to stream buffer Copied to cache on demand
k + miss penalty + 2	Line **L + 2** brought to cache	Line **L + 2** brought to stream buffer Copied to cache on demand
No further miss penalties Stream buffer issues sequential requests until full or re-allocated to new miss		

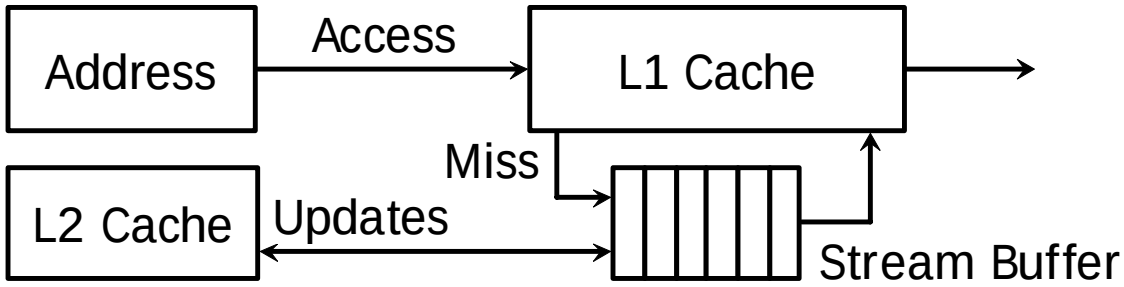
Stream Buffer Example

Lines in cache

0	1	2	3	256	257	6	15	8	...

CC	Line Accessed	Cache	Stream Buffer Slots					
			0	1	2	3	4	5
0	2	Hit						
1	3	Hit						
2	4	Miss	4					
3	4	Miss	4	5				
4	4	Miss	4	5	7		Skip 6	
5	4	Miss	4	5	7	9	Skip 8	
6	4	Miss	4	5	7	9	10	
...								
10	4	Hit	4	5	7	9	10	11
11	5	Hit	4	5	7	9	10	11

Stride Buffer



Stride

Sequential accesses to lines L, L+s, L+2s, L+3s, ...

Access stride = s

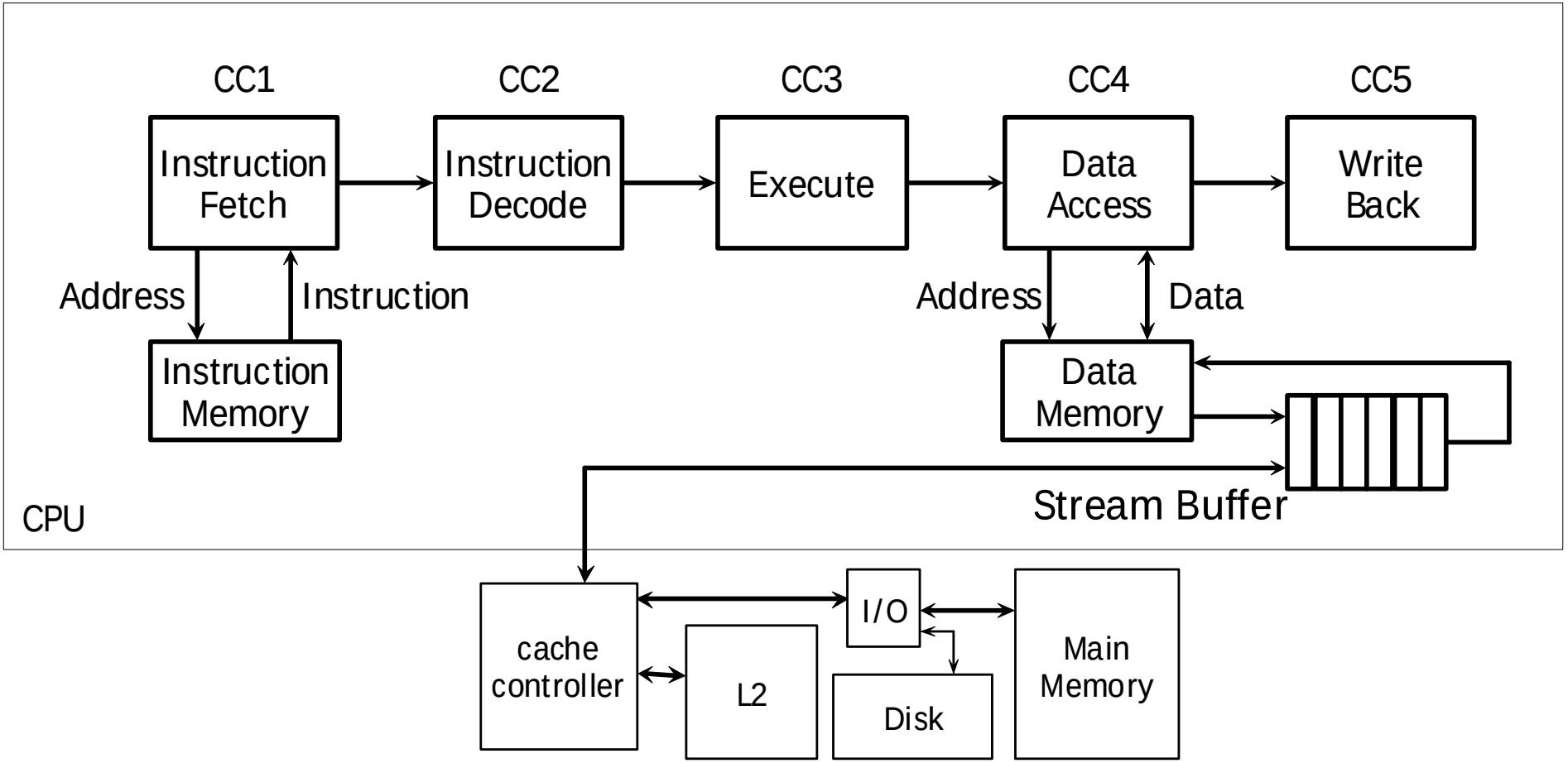
Stride buffer

Variation of stream buffer

Prefetches lines according to stride s

Stream buffer allocated following 2 sequential cache misses

Stream Buffer for DLX Data Cache

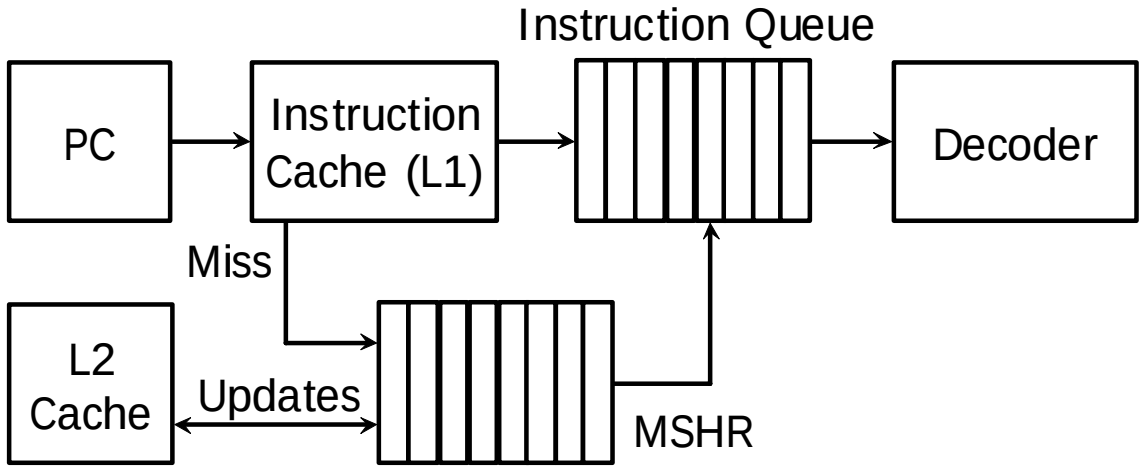


Out-of-Order Fetch

Miss Information Status History Register (MSHR)

Variation of stream buffer for instruction cache

Additional random access buffer for instruction blocks



On cache miss

Allocate MSHR slot to missed address

- Requests cache update for missed address
- Writes pointer for missed addresses in Instruction Queue

PC advances

- L1 hits → Instruction Queue
- L1 misses → Instruction Queue on update

MSHR Example

Blocks in cache

0	1	2	3	256	257	6	15	8	...

CC	Block In PC	Cache	MSHR Slots			Instruction Queue						
			0	1	2	0	1	2	3	4	5	6
0	2	Hit				2						
1	3	Hit				3						
2	4	Miss	4			P4						
3	5	Miss	4	5		P4	P5					
4	6	Hit	4	5		P4	P5	6				
5	7	Miss	4	5	7	P4	P5	6	P7			
6	8	Hit	4	5	7	P4	P5	6	P7	8	...	...
...												
50			4	5	7	4	P5	6	P7	8	...	...
51			4	5	7	5	6	P7	8	...	...	...

Fetch Directed Instruction Prefetching

Branch prediction → cache prefetch

FIFO between branch predictor and L1 instruction cache

Branch detected at address PC

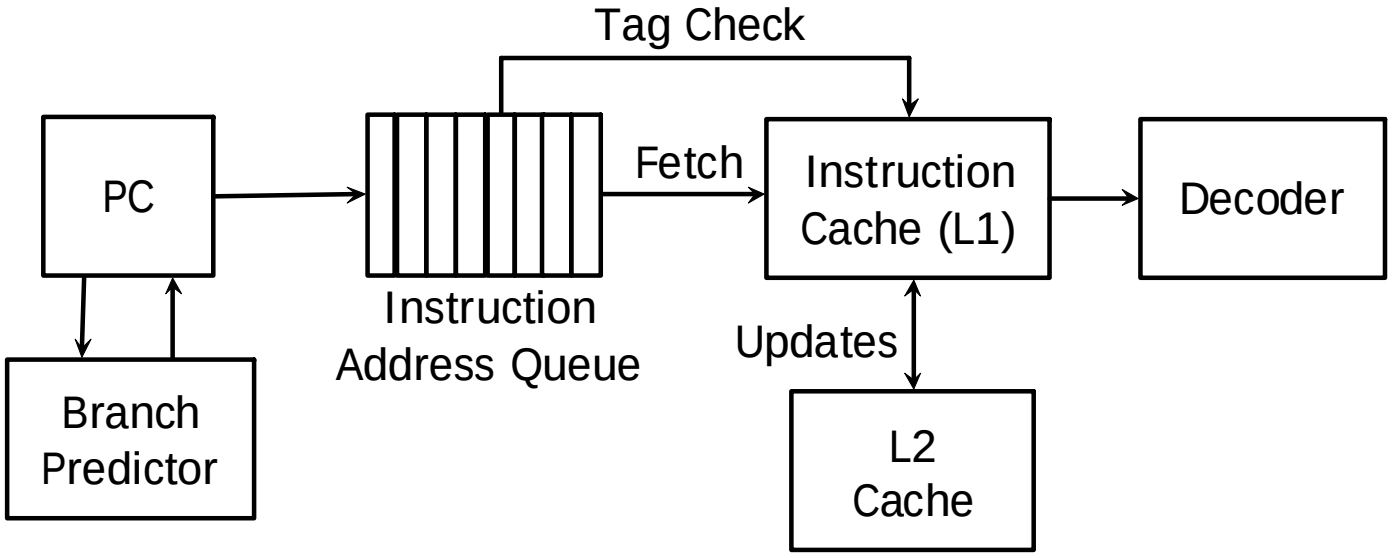
Branch prediction

Predicted next PC → address queue

Address queue initiates tag check

Cache update on miss

L1 prefetches missed line before access to fetch address



Fetch Directed Prefetching for DLX

