

18-447

Computer Architecture
Lecture 8: Data and Control
Dependence Handling

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2014, 1/29/2014

Readings for Next Few Lectures

- P&H Chapter 4.9-4.11
- Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proceedings of the IEEE, 1995
 - More advanced pipelining
 - Interrupt and exception handling
 - Out-of-order and superscalar execution concepts
- McFarling, “Combining Branch Predictors,” DEC WRL Technical Report, 1993.
- Kessler, “The Alpha 21264 Microprocessor,” IEEE Micro 1999.


Data Dependence Handling:

More Depth & Implementation

Remember: Data Dependence Types

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR)

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$



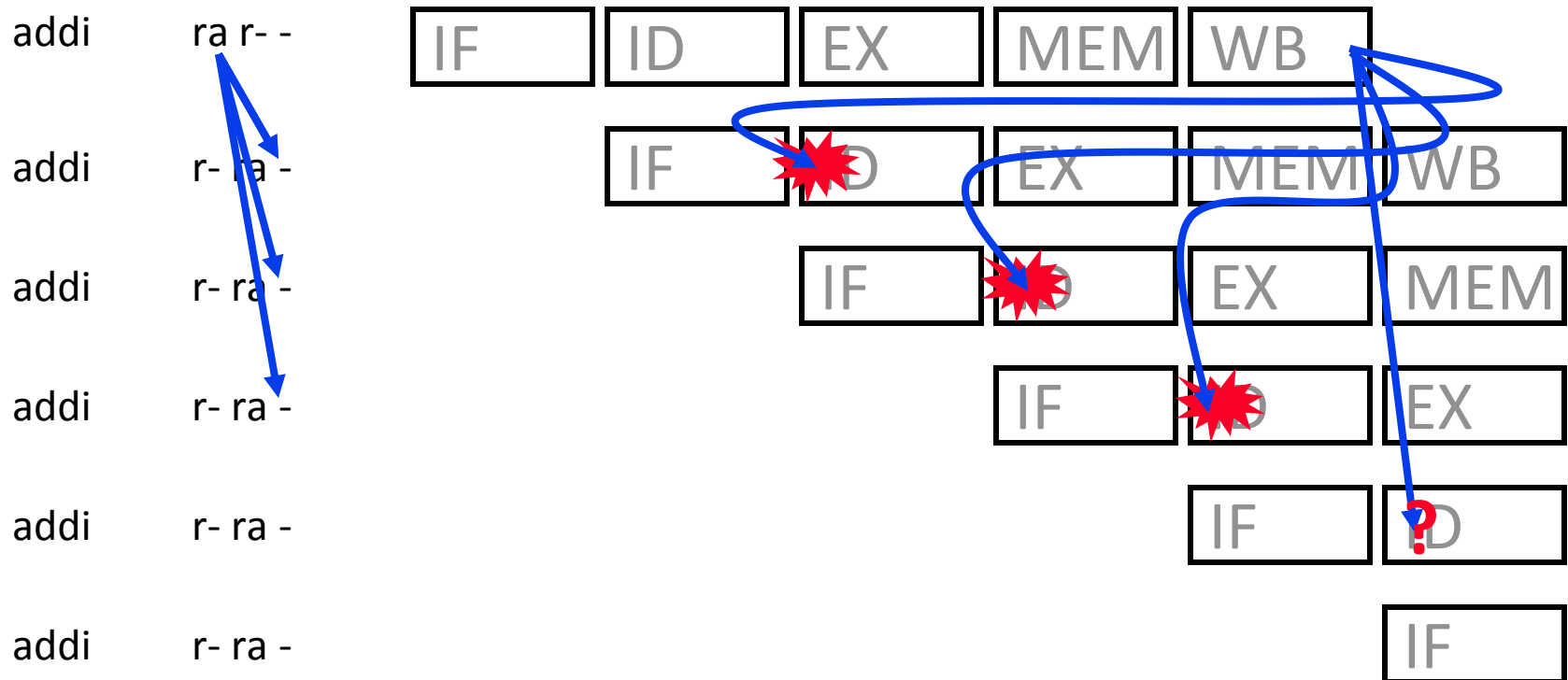
Write-after-Write
(WAW)

How to Handle Data Dependences

- Anti and output dependences are easier to handle
 - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
 - **Detect and wait** until value is available in register file
 - **Detect and forward/bypass** data to dependent instruction
 - **Detect and eliminate** the dependence at the software level
 - No need for the hardware to detect dependence
 - **Predict** the needed value(s), execute “speculatively”, **and verify**
 - **Do something else** (fine-grained multithreading)
 - No need to detect

RAW Dependence Handling

- Following flow dependences lead to conflicts in the 5-stage pipeline

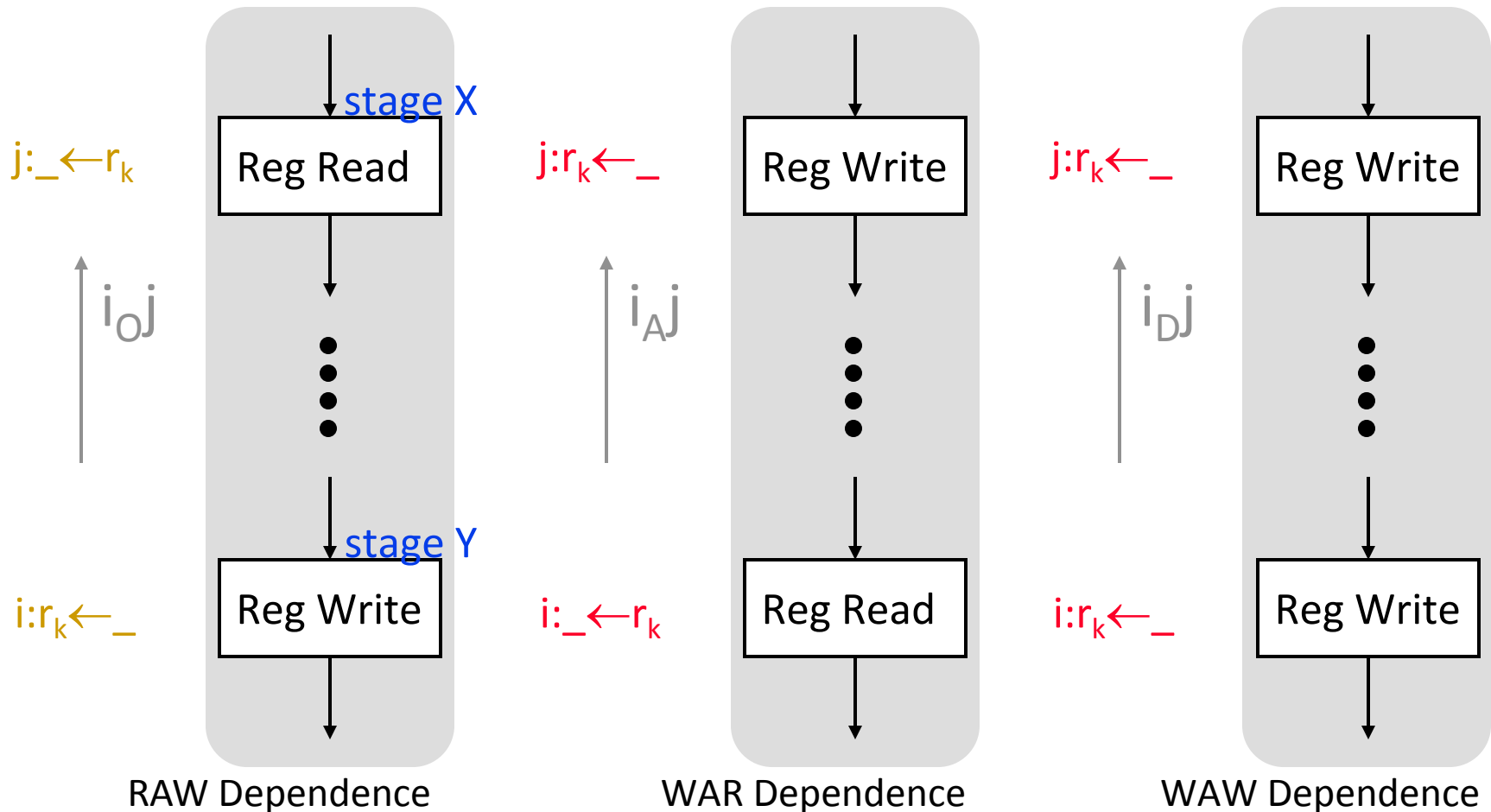


Register Data Dependence Analysis

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

- For a given pipeline, when is there a potential conflict between 2 data dependent instructions?
 - ❑ dependence type: RAW, WAR, WAW?
 - ❑ instruction types involved?
 - ❑ distance between the two instructions?

Safe and Unsafe Movement of Pipeline



$\text{dist}(i,j) \leq \text{dist}(X,Y) \Rightarrow$ Unsafe to keep j moving

$\text{dist}(i,j) > \text{dist}(X,Y) \Rightarrow$ Safe

RAW Dependence Analysis Example

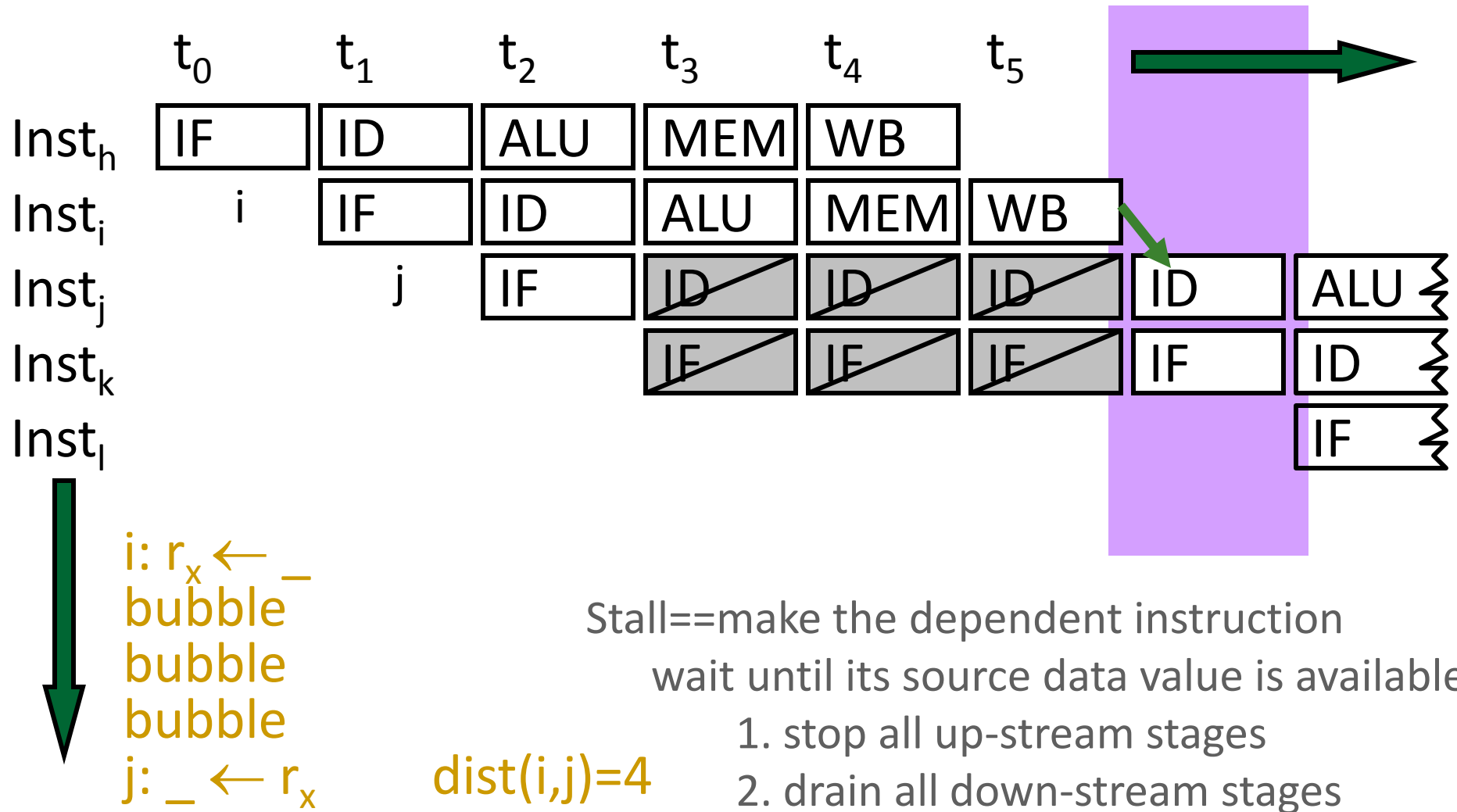
	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

- Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
 - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
 - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$

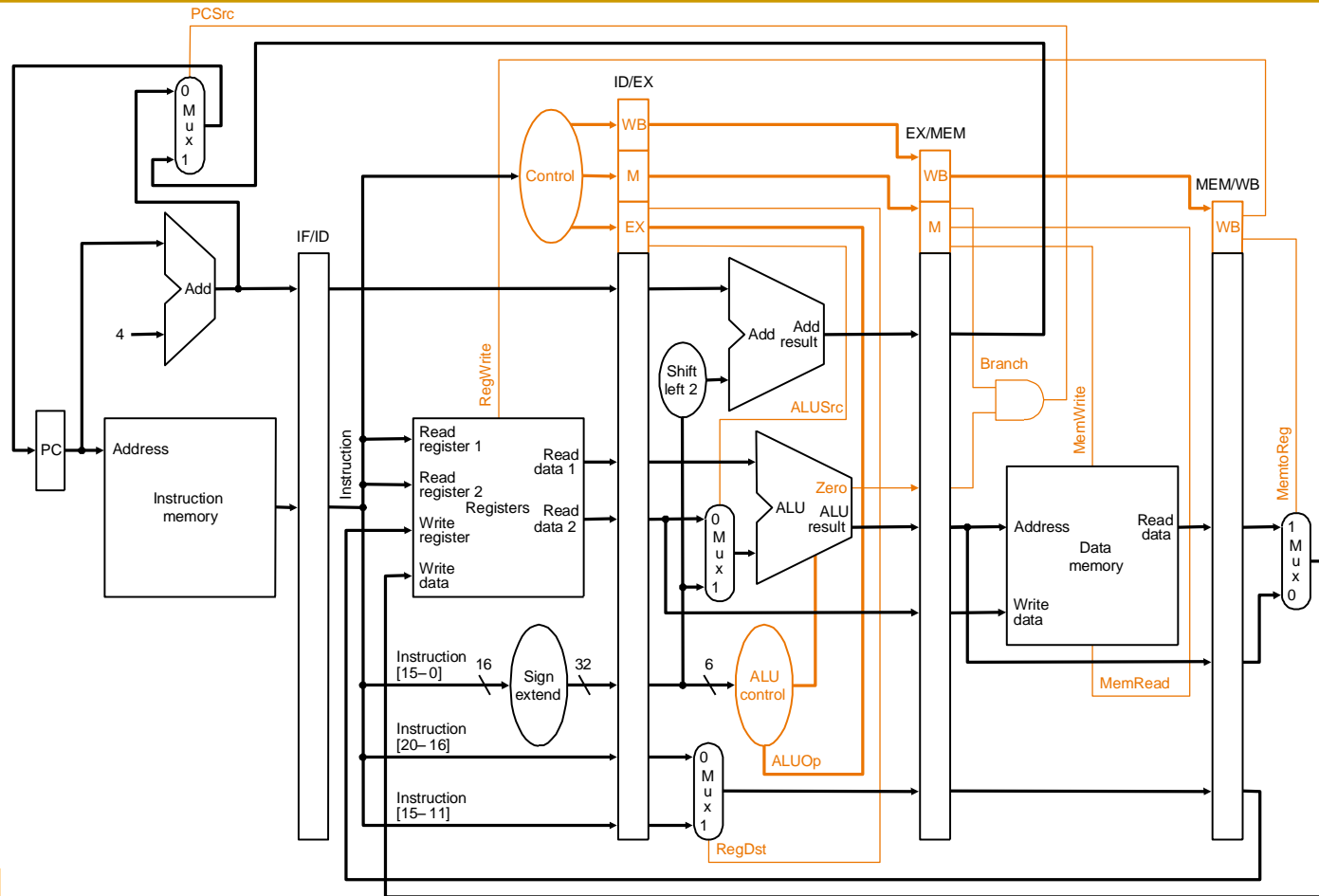
What about WAW and WAR dependence?

What about memory data dependence?

Pipeline Stall: Resolving Data Dependence



How to Implement Stalling



■ Stall

- ❑ disable **PC** and **IR** latching; ensure stalled instruction stays in its stage
- ❑ Insert “invalid” instructions/nops into the stage following the stalled one

Stall Conditions

- Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
 - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
 - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$
- In other words, must stall when I_B in ID stage wants to read a register to be written by I_A in EX, MEM or WB stage

Stall Conditions

■ Helper functions

- ❑ $rs(I)$ returns the rs field of I
- ❑ $use_rs(I)$ returns true if I requires $RF[rs]$ and $rs \neq r0$

■ Stall when

- ❑ $(rs(IR_{ID}) == dest_{EX}) \ \&\& \ use_rs(IR_{ID}) \ \&\& \ RegWrite_{EX}$ or
- ❑ $(rs(IR_{ID}) == dest_{MEM}) \ \&\& \ use_rs(IR_{ID}) \ \&\& \ RegWrite_{MEM}$ or
- ❑ $(rs(IR_{ID}) == dest_{WB}) \ \&\& \ use_rs(IR_{ID}) \ \&\& \ RegWrite_{WB}$ or
- ❑ $(rt(IR_{ID}) == dest_{EX}) \ \&\& \ use_rt(IR_{ID}) \ \&\& \ RegWrite_{EX}$ or
- ❑ $(rt(IR_{ID}) == dest_{MEM}) \ \&\& \ use_rt(IR_{ID}) \ \&\& \ RegWrite_{MEM}$ or
- ❑ $(rt(IR_{ID}) == dest_{WB}) \ \&\& \ use_rt(IR_{ID}) \ \&\& \ RegWrite_{WB}$

- It is crucial that the EX, MEM and WB stages continue to advance normally during stall cycles

Impact of Stall on Performance

- Each stall cycle corresponds to 1 lost ALU cycle
- For a program with N instructions and S stall cycles,
Average $CPI = (N + S) / N$
- S depends on
 - frequency of RAW dependences
 - exact distance between the dependent instructions
 - distance between dependences

suppose i_1, i_2 and i_3 all depend on i_0 , once i_1 's dependence is resolved, i_2 and i_3 must be okay too

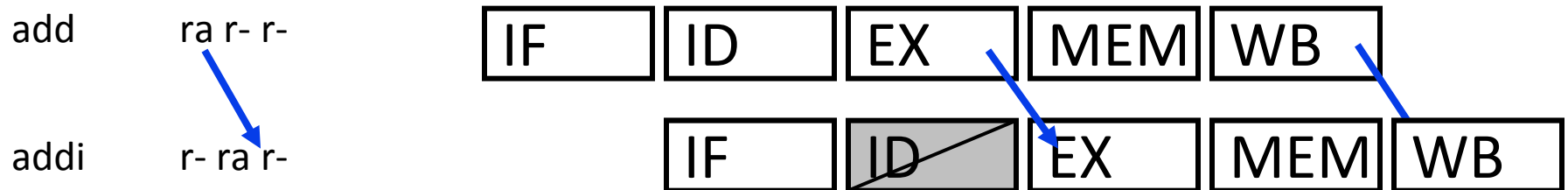
Sample Assembly (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { }

```
for2tst:  addi    $s1, $s0, -1           3 stalls
          slti    $t0, $s1, 0         3 stalls
          bne     $t0, $zero, exit2
          sll     $t1, $s1, 2         3 stalls
          add     $t2, $a0, $t1       3 stalls
          lw      $t3, 0($t2)
          lw      $t4, 4($t2)         3 stalls
          slt     $t0, $t4, $t3       3 stalls
          beq     $t0, $zero, exit2
          .....
          addi    $s1, $s1, -1
          j       for2tst
exit2:
```

Data Forwarding (or Data Bypassing)

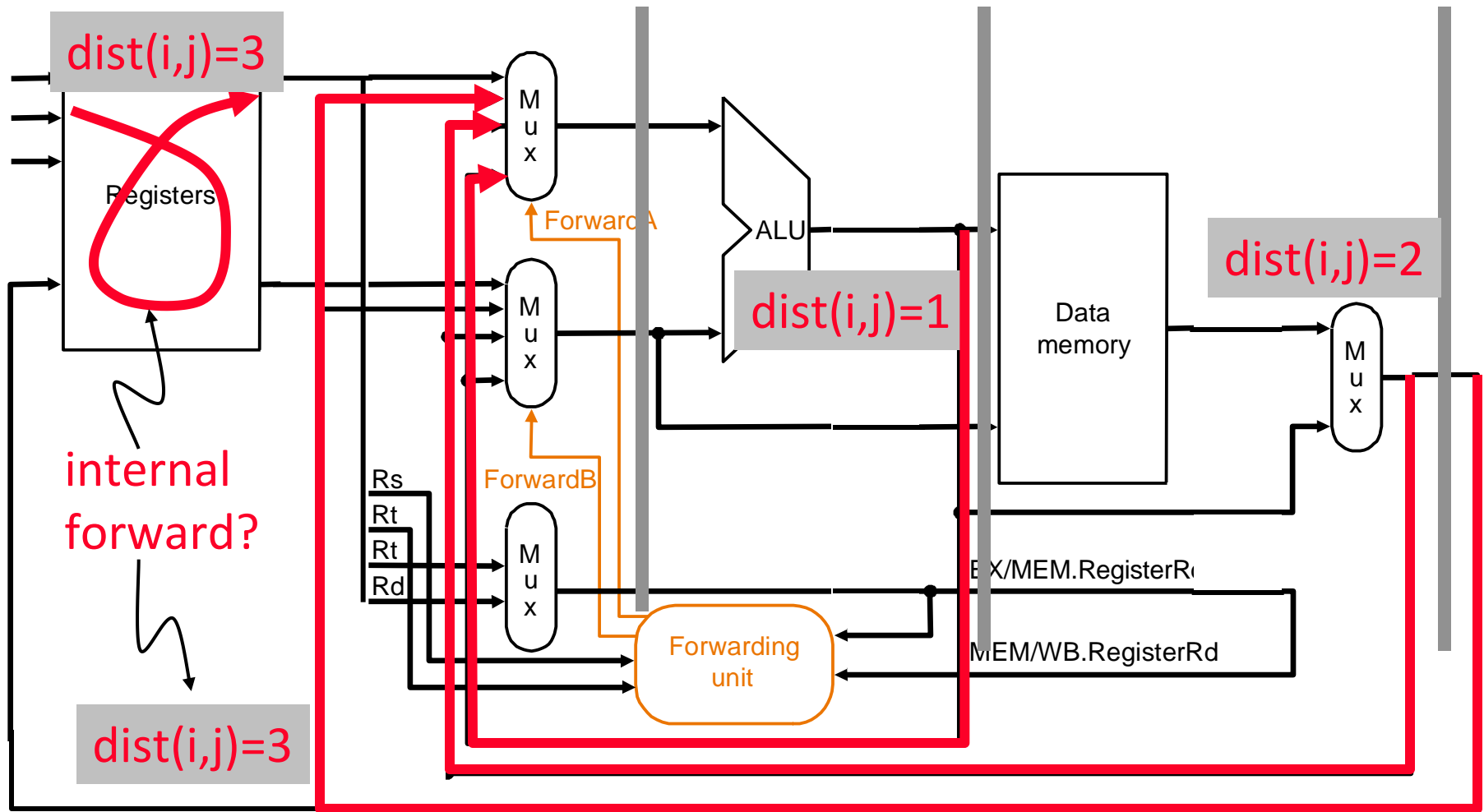
- It is intuitive to think of RF as state
 - “add rx ry rz” literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]
- But, RF is just a part of a computing abstraction
 - “add rx ry rz” means 1. get the results of the last instructions to define the values of RF[ry] and RF[rz], respectively, and 2. until another instruction redefines RF[rx], younger instructions that refers to RF[rx] should use this instruction’s result
- What matters is to maintain the correct “dataflow” between operations, thus



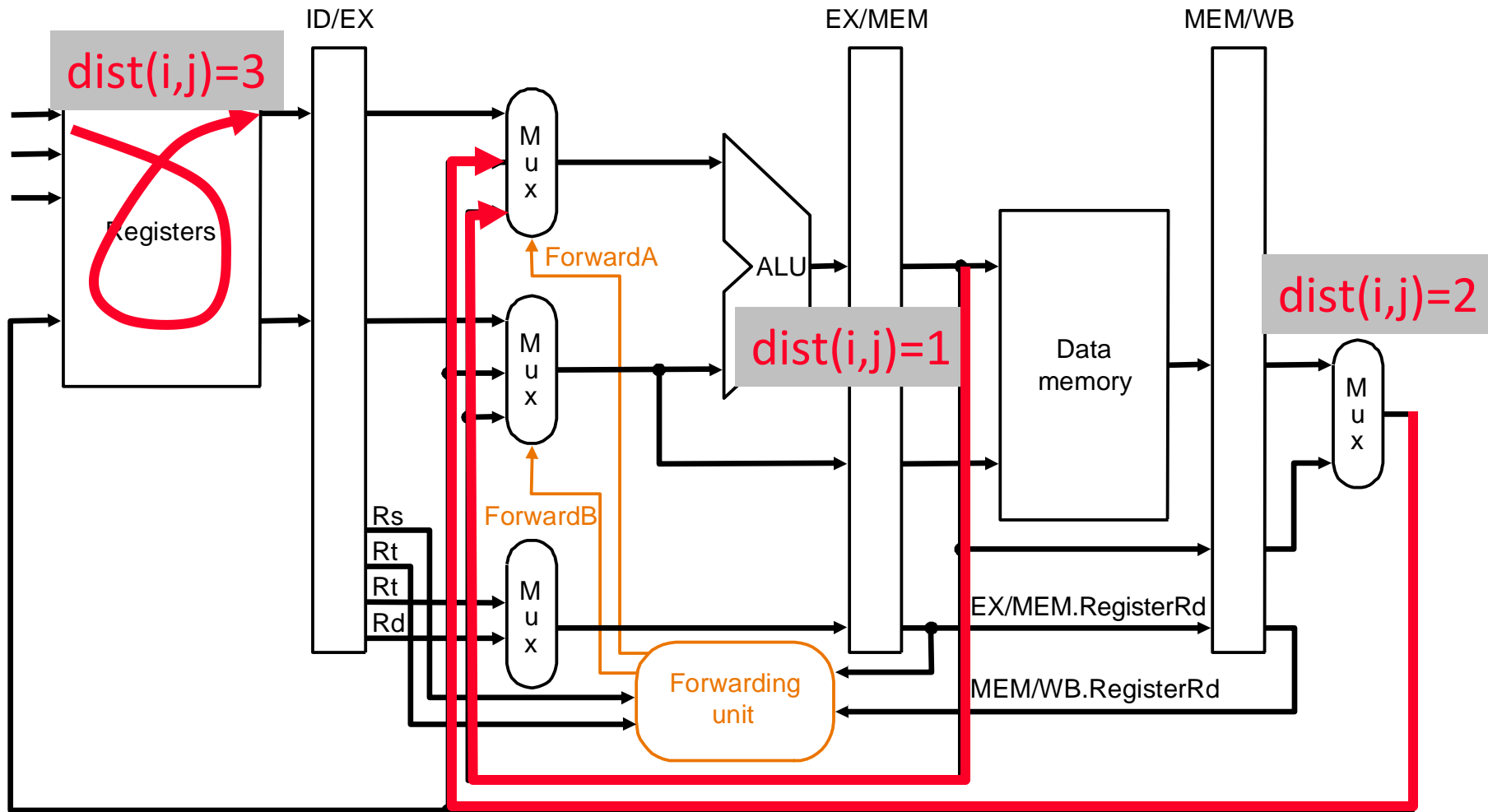
Resolving RAW Dependence with Forwarding

- Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
 - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
 - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$
- In other words, if I_B in ID stage reads a register written by I_A in EX, MEM or WB stage, then the operand required by I_B is not yet in RF
 - ⇒ retrieve operand from datapath instead of the RF
 - ⇒ retrieve operand from the youngest definition if multiple definitions are outstanding

Data Forwarding Paths (v1)



Data Forwarding Paths (v2)



b. With forwarding

Assumes RF forwards internally

Data Forwarding Logic (for v2)

```
if (rsEX!=0) && (rsEX==destMEM) && RegWriteMEM then
    forward operand from MEM stage          // dist=1
else if (rsEX!=0) && (rsEX==destWB) && RegWriteWB then
    forward operand from WB stage  // dist=2
else
    use AEX (operand from register file)      // dist >= 3
```

Ordering matters!! Must check youngest match first

Why doesn't use_{rs}() appear in the forwarding logic?

What does the above not take into account?

Data Forwarding (Dependence Analysis)

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID						use
EX	use produce	use	use	use		
MEM		produce	(use)			
WB						

- Even with data-forwarding, RAW dependence on an immediately preceding LW instruction requires a stall

Sample Assembly, Revisited (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { }

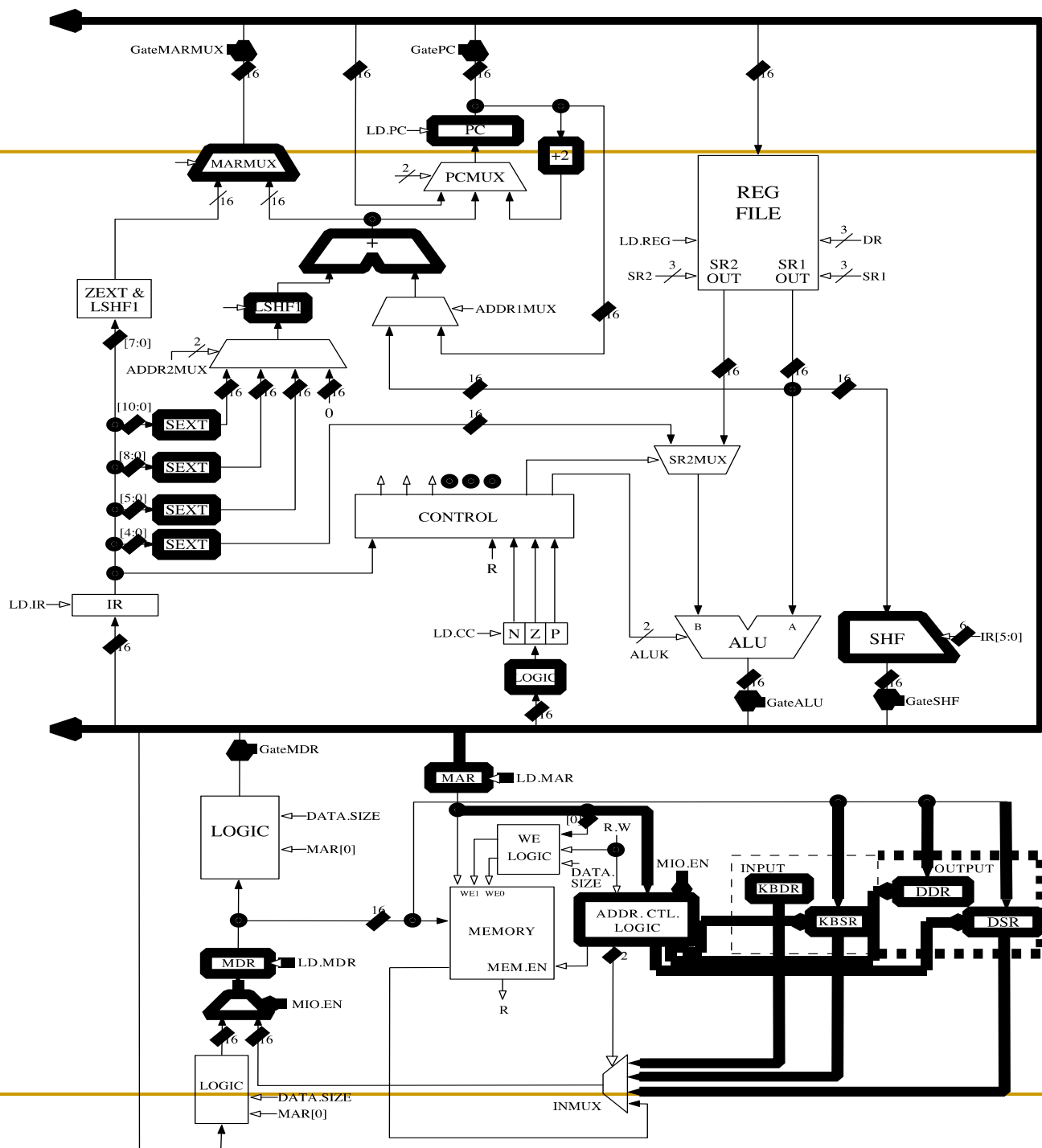
```
                addi    $s1, $s0, -1
for2tst:        slti    $t0, $s1, 0
                bne     $t0, $zero, exit2
                sll     $t1, $s1, 2
                add     $t2, $a0, $t1
                lw      $t3, 0($t2)
                lw      $t4, 4($t2)
                nop
                slt     $t0, $t4, $t3
                beq     $t0, $zero, exit2
                .....
                addi    $s1, $s1, -1
                j       for2tst
```

exit2:

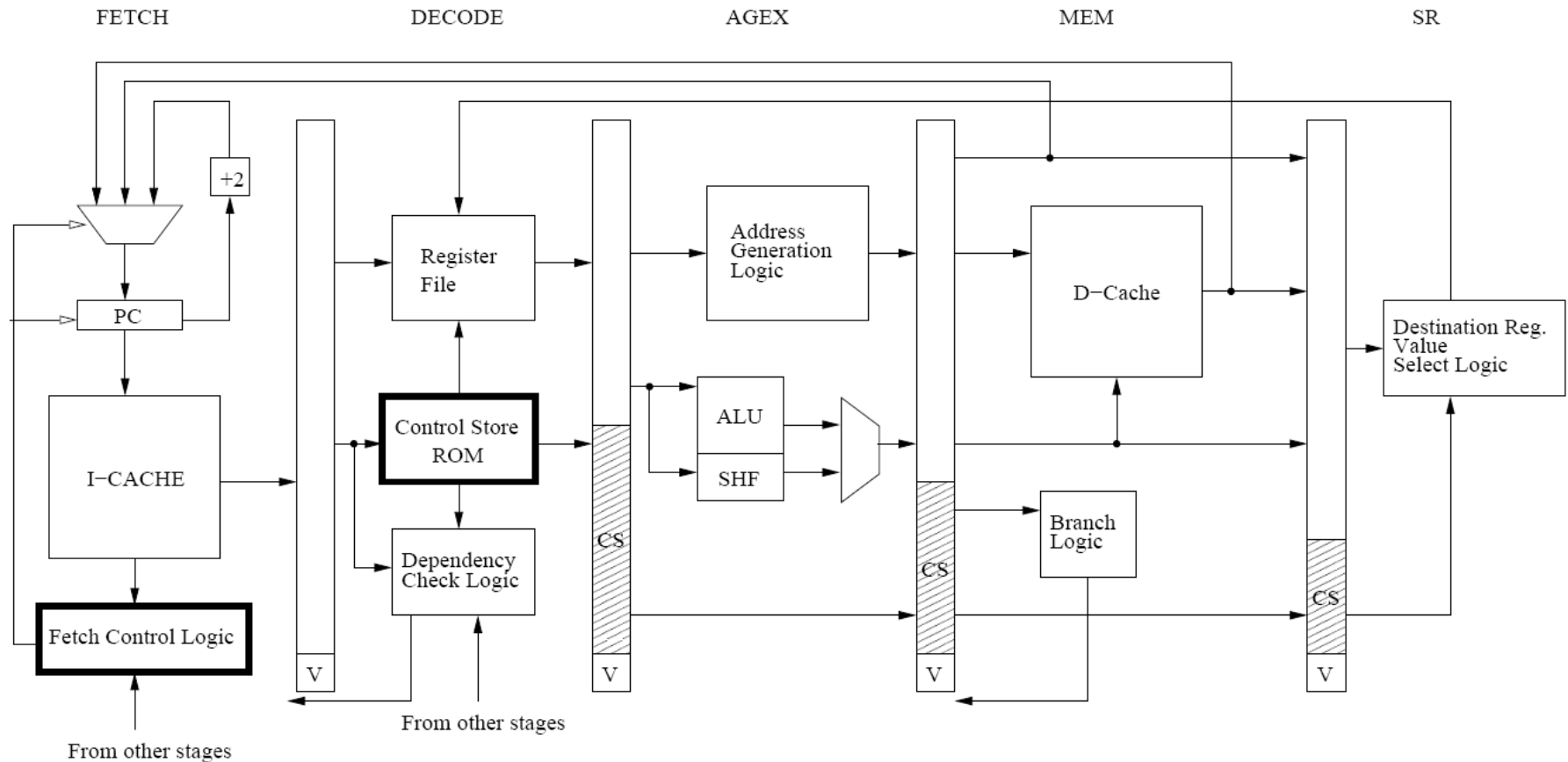
Pipelining the LC-3b

Pipelining the LC-3b

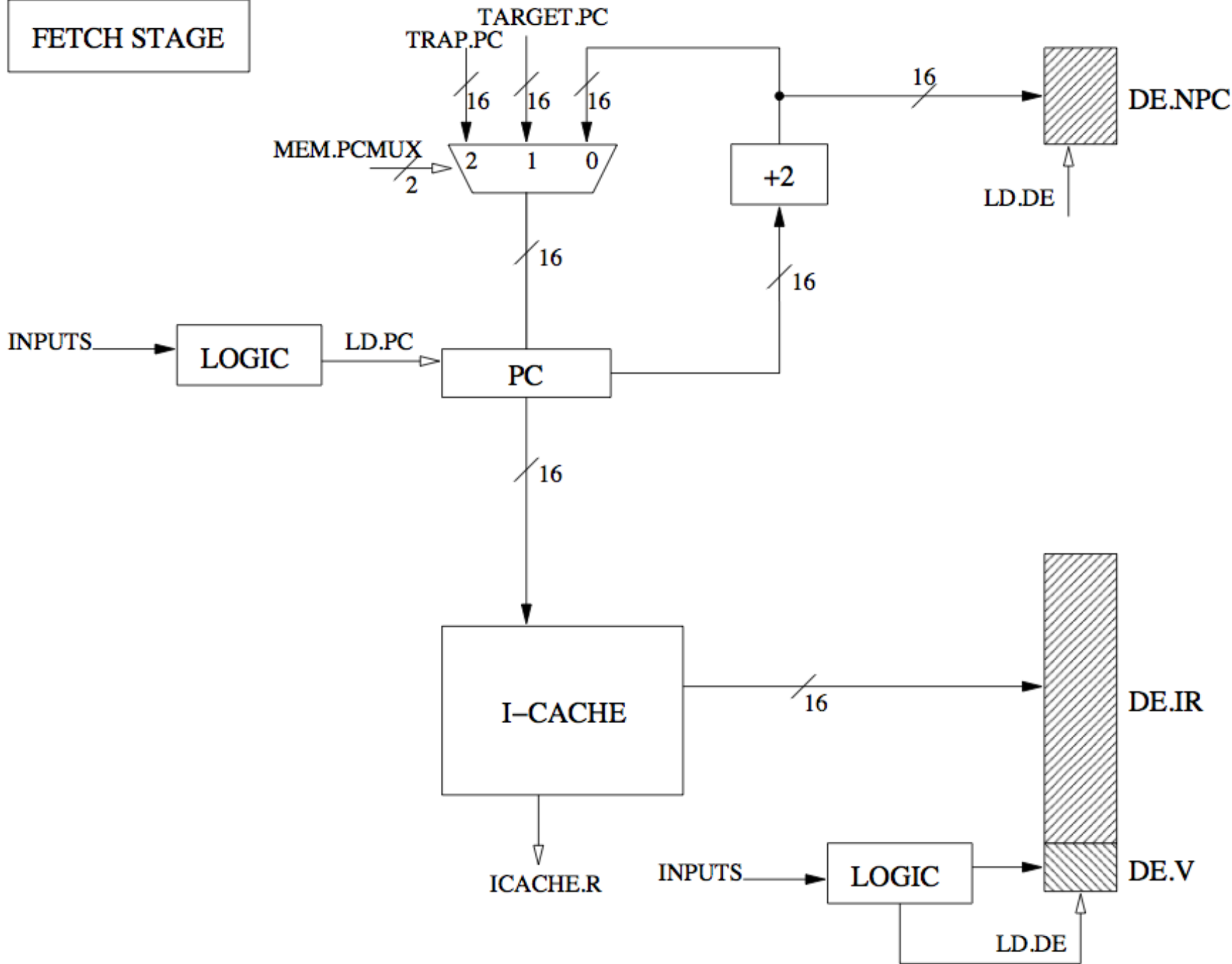
- Let's remember the single-bus datapath
- We'll divide it into 5 stages
 - Fetch
 - Decode/RF Access
 - Address Generation/Execute
 - Memory
 - Store Result
- Conservative handling of data and control dependences
 - Stall on branch
 - Stall on flow dependence

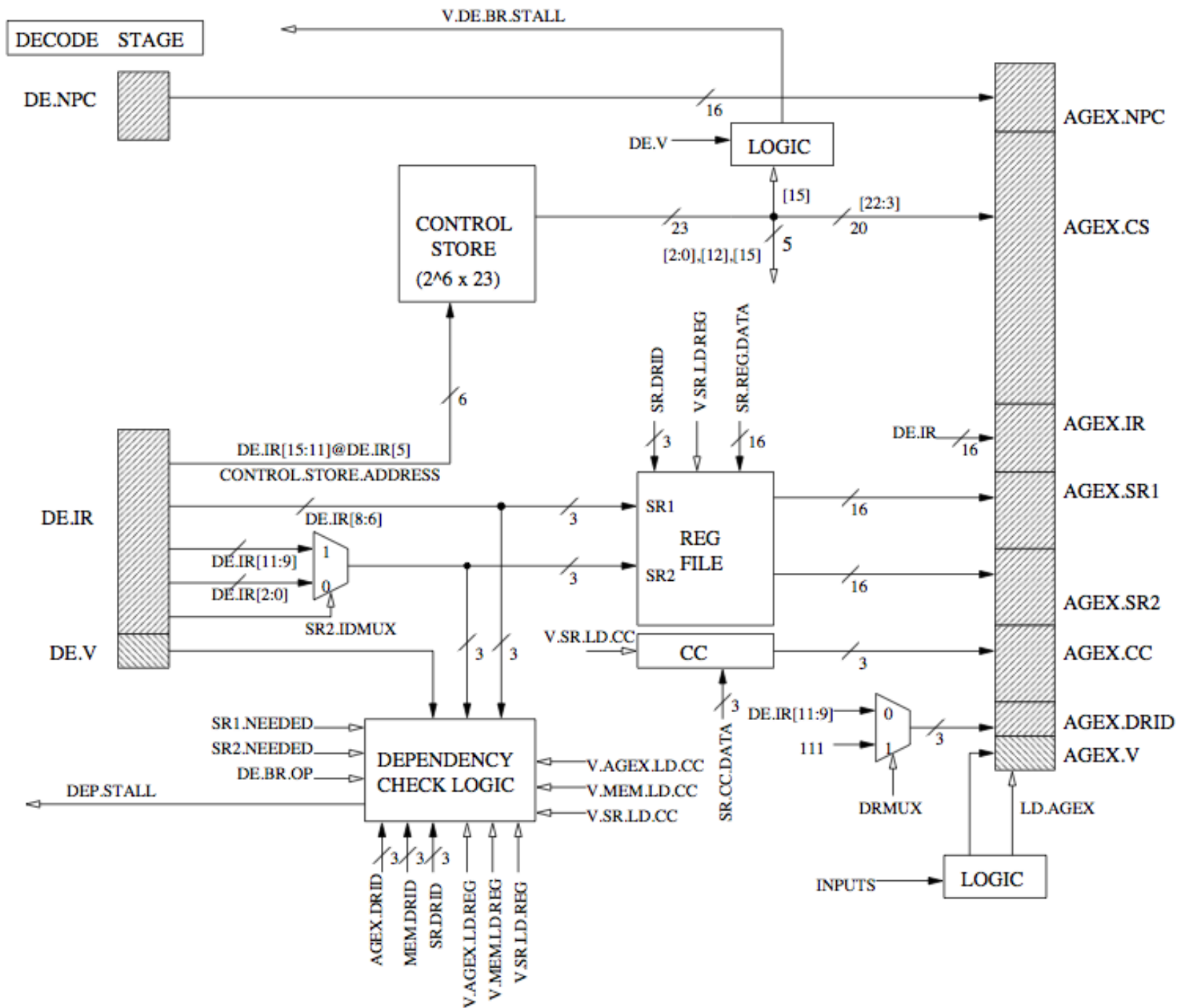


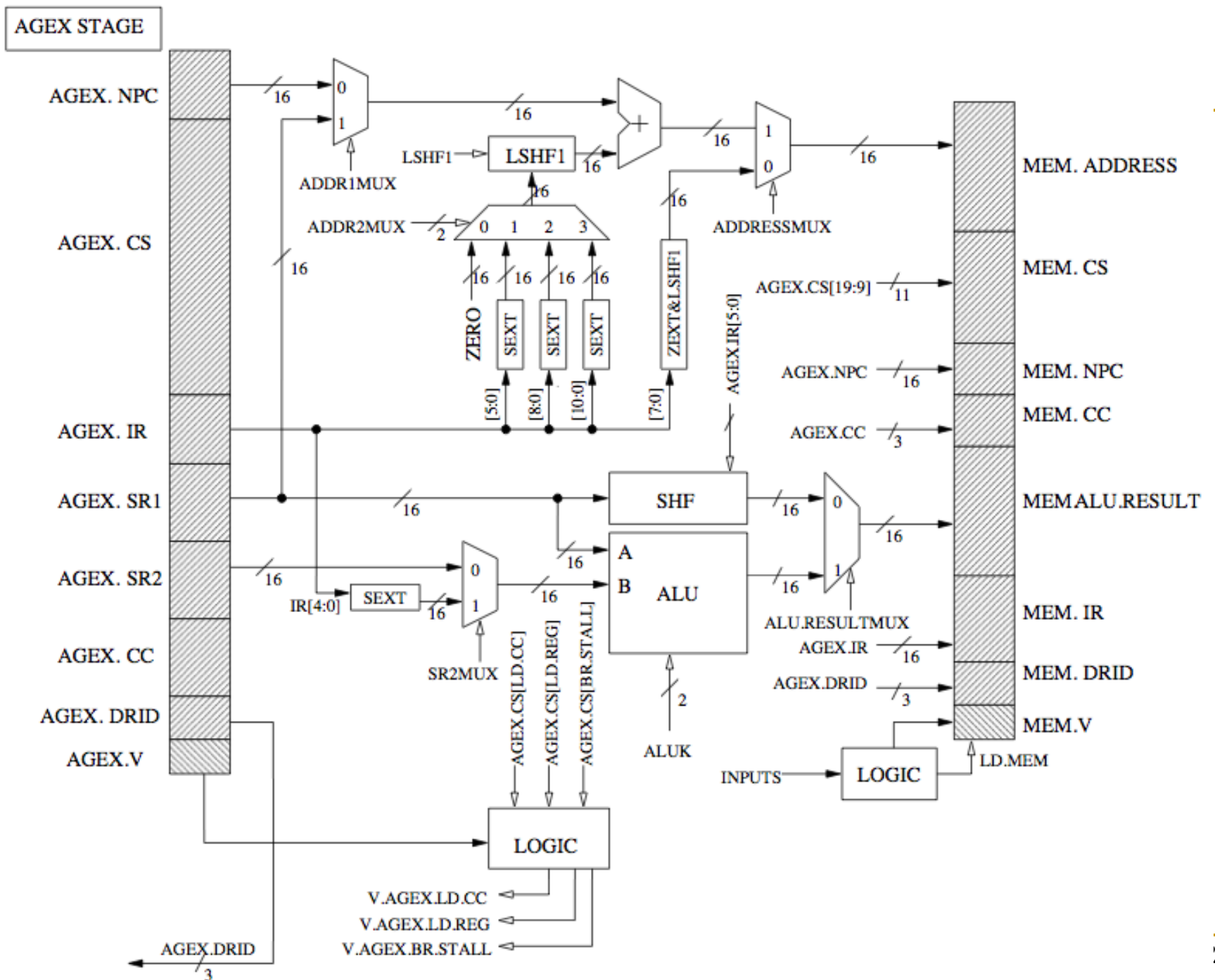
An Example LC-3b Pipeline

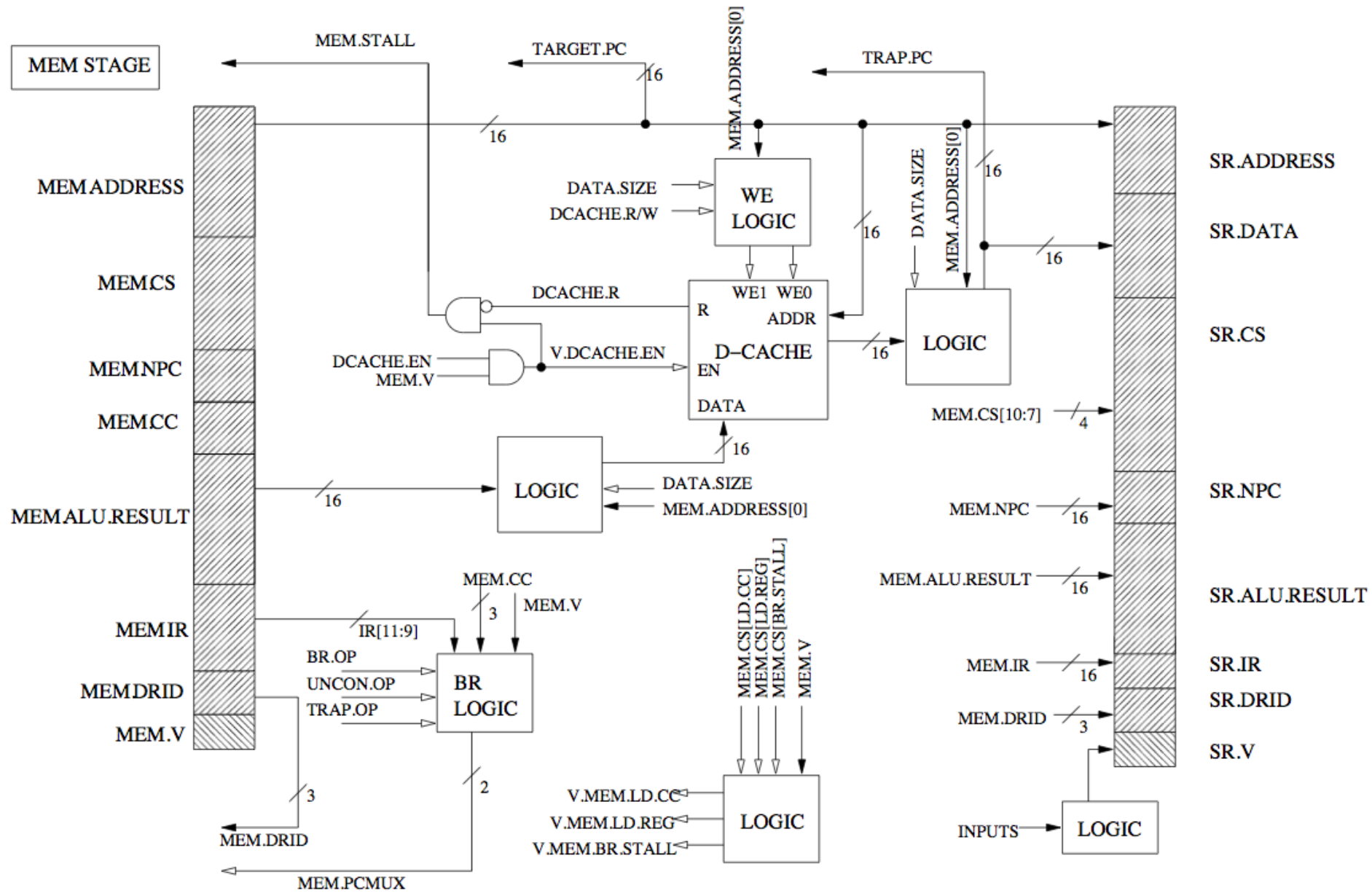


FETCH STAGE

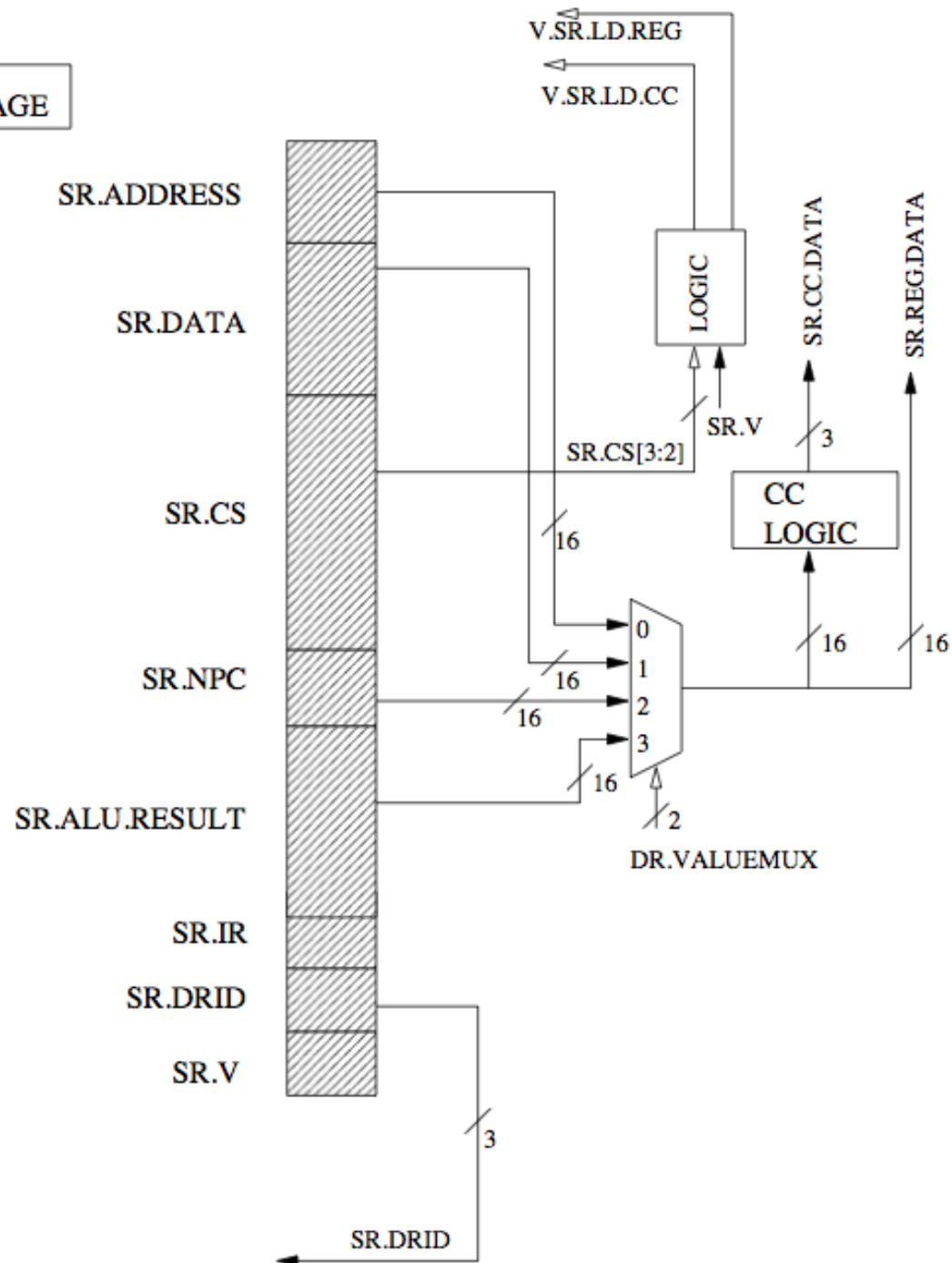








SR STAGE



Control of the LC-3b Pipeline

- Three types of control signals
- Datapath Control Signals
 - Control signals that control the operation of the datapath
- Control Store Signals
 - Control signals (microinstructions) stored in control store to be used in pipelined datapath (can be propagated to stages later than decode)
- Stall Signals
 - Ensure the pipeline operates correctly in the presence of dependencies

Stage	Signal Name	Signal Values	
FETCH	MEM.PCMUX/2:††	PC+2	;select pc+2
		TARGET.PC	;select MEM.TARGET.PC (branch target)
		TRAP.PC	;select MEM.TRAP.PC
		NO(0), LOAD(1)	
DECODE	LD.PC/1:†	NO(0), LOAD(1)	
		NO(0), LOAD(1)	
	LD.DE/1:†	NO(0), LOAD(1)	
		NO(0), LOAD(1)	
	DRMUX/1:	11.9	;destination IR[11:9]
		R7	;destination R7
	SR1.NEEDED/1:	NO(0), YES(1)	;asserted if instruction needs SR1
	SR2.NEEDED/1:	NO(0), YES(1)	;asserted if instruction needs SR2
	DE.BR.OP/1:	NO(0), BR(1)	;BR Opcode
	SR2.IDMUX/1:†	2.0	;source IR[2:0]
		11.9	;source IR[11:9]
	LD.AGEX/1:†	NO(0), LOAD(1)	
	V.AGEX.LD.CC/1:††	NO(0), LOAD(1)	
	V.MEM.LD.CC/1:††	NO(0), LOAD(1)	
	V.SR.LD.CC/1:††	NO(0), LOAD(1)	
	V.AGEX.LD.REG/1:††	NO(0), LOAD(1)	
	V.MEM.LD.REG/1:††	NO(0), LOAD(1)	
	V.SR.LD.REG/1:††	NO(0), LOAD(1)	
AGEX	ADDR1MUX/1:	NPC	;select value from AGEX.NPC
		BaseR	;select value from AGEX.SR1(BaseR)
	ADDR2MUX/2:	ZERO	;select the value zero
		offset6	;select SEXT[IR[5:0]]
		PCoffset9	;select SEXT[IR[8:0]]
		PCoffset11	;select SEXT[IR[10:0]]
	LSHF1/1:	NO(0), 1bit Left shift(1)	
	ADDRESSMUX/1:	7.0	;select LSHF(ZEXT[IR[7:0]],1)
		ADDER	;select output of address adder
	SR2MUX/1:	SR2	;select from AGEX.SR2
		4.0	;IR[4:0]
	ALUK/2:	ADD(00), AND(01)	
ALU.RESULTMUX/1:		XOR(10), PASSB(11)	
		SHIFTER	;select output of the shifter
		ALU	;select tput out the ALU
	LD.MEM/1:†	NO(0), LOAD(1)	
		NO(0), YES(1)	
		RD(0), WR(1)	
MEM	DCACHE.EN/1:	NO(0), YES(1)	;asserted if the instruction accesses memory
	DCACHE.RW/1:	RD(0), WR(1)	
	DATA.SIZE/1:	BYTE(0), WORD(1)	
	BR.OP/1:	NO(0), BR(1)	;BR
	UNCON.OP/1:	NO(0), Uncond.BR(1)	;JMP,RET,JSR,JSRR
SR	TRAP.OP/1:	NO(0), Trap(1)	;TRAP
	DR.VALUEMUX/2:	ADDRESS	;select value from SR.ADDRESS
		DATA	;select value from SR.DATA
		NPC	;select value from SR.NPC
		ALU	;select value from SR.ALU.RESULT
	LD.REG/1:	NO(0), LOAD(1)	
	LD.CC/1:	NO(0), LOAD(1)	

Table 1: Data Path Control Signals
†: The control signal is generated by logic in that stage
††: The control signal is generated by logic in another stage

Control Store in a Pipelined Machine

Number	Signal Name	Stages
0	SR1.NEEDED	DECODE
1	SR2.NEEDED	DECODE
2	DRMUX	DECODE
3	ADDR1MUX	AGEX
4	ADDR2MUX1	AGEX
5	ADDR2MUX0	AGEX
6	LSHF1	AGEX
7	ADDRESSMUX	AGEX
8	SR2MUX	AGEX
9	ALUK1	AGEX
10	ALUK0	AGEX
11	ALU.RESULTMUX	AGEX
12	BR.OP	DECODE, MEM
13	UNCON.OP	MEM
14	TRAP.OP	MEM
15	BR.STALL	DECODE, AGEX, MEM
16	DCACHE.EN	MEM
17	DCACHE.RW	MEM
18	DATA.SIZE	MEM
19	DR.VALUEMUX1	SR
20	DR.VALUEMUX0	SR
21	LD.REG	AGEX, MEM, SR
22	LD.CC	AGEX, MEM, SR

Table 2: Control Store ROM Signals

Stall Signals

- Pipeline stall: Pipeline does not move because an operation in a stage cannot complete
- Stall Signals: Ensure the pipeline operates correctly in the presence of such an operation
- Why could an operation in a stage not complete?

Signal Name	Generated in	
ICACHE.R/1:	FETCH	NO, READY
DEP.STALL/1:	DEC	NO, STALL
V.DE.BR.STALL/1:	DEC	NO, STALL
V.AGEX.BR.STALL/1:	AGEX	NO, STALL
MEM.STALL/1:	MEM	NO, STALL
V.MEM.BR.STALL/1:	MEM	NO, STALL

Table 3: STALL Signals

Pipelined LC-3b

- <http://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=18447-lc3b-pipelining.pdf>

End of Pipelining the LC-3b

Questions to Ponder

- What is the role of the hardware vs. the software in data dependence handling?
 - ❑ Software based interlocking
 - ❑ Hardware based interlocking
 - ❑ Who inserts/manages the pipeline bubbles?
 - ❑ Who finds the independent instructions to fill “empty” pipeline slots?
 - ❑ What are the advantages/disadvantages of each?

Questions to Ponder

- What is the role of the hardware vs. the software in the order in which instructions are executed in the pipeline?
 - Software based instruction scheduling → static scheduling
 - Hardware based instruction scheduling → dynamic scheduling

More on Software vs. Hardware

- Software based scheduling of instructions → static scheduling
 - ❑ Compiler orders the instructions, hardware executes them in that order
 - ❑ Contrast this with dynamic scheduling (in which hardware will execute instructions out of the compiler-specified order)
 - ❑ How does the compiler know the latency of each instruction?
- What information does the compiler not know that makes static scheduling difficult?
 - ❑ Answer: Anything that is determined at run time
 - Variable-length operation latency, memory addr, branch direction
- How can the compiler alleviate this (i.e., estimate the unknown)?
 - ❑ Answer: Profiling

Control Dependence Handling

Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
 - All instructions are control dependent on previous ones. Why?
- If the fetched instruction is a non-control-flow instruction:
 - Next Fetch PC is the address of the next-sequential instruction
 - Easy to determine if we know the size of the fetched instruction
- If the instruction that is fetched is a control-flow instruction:
 - How do we determine the next Fetch PC?
- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?

Branch Types

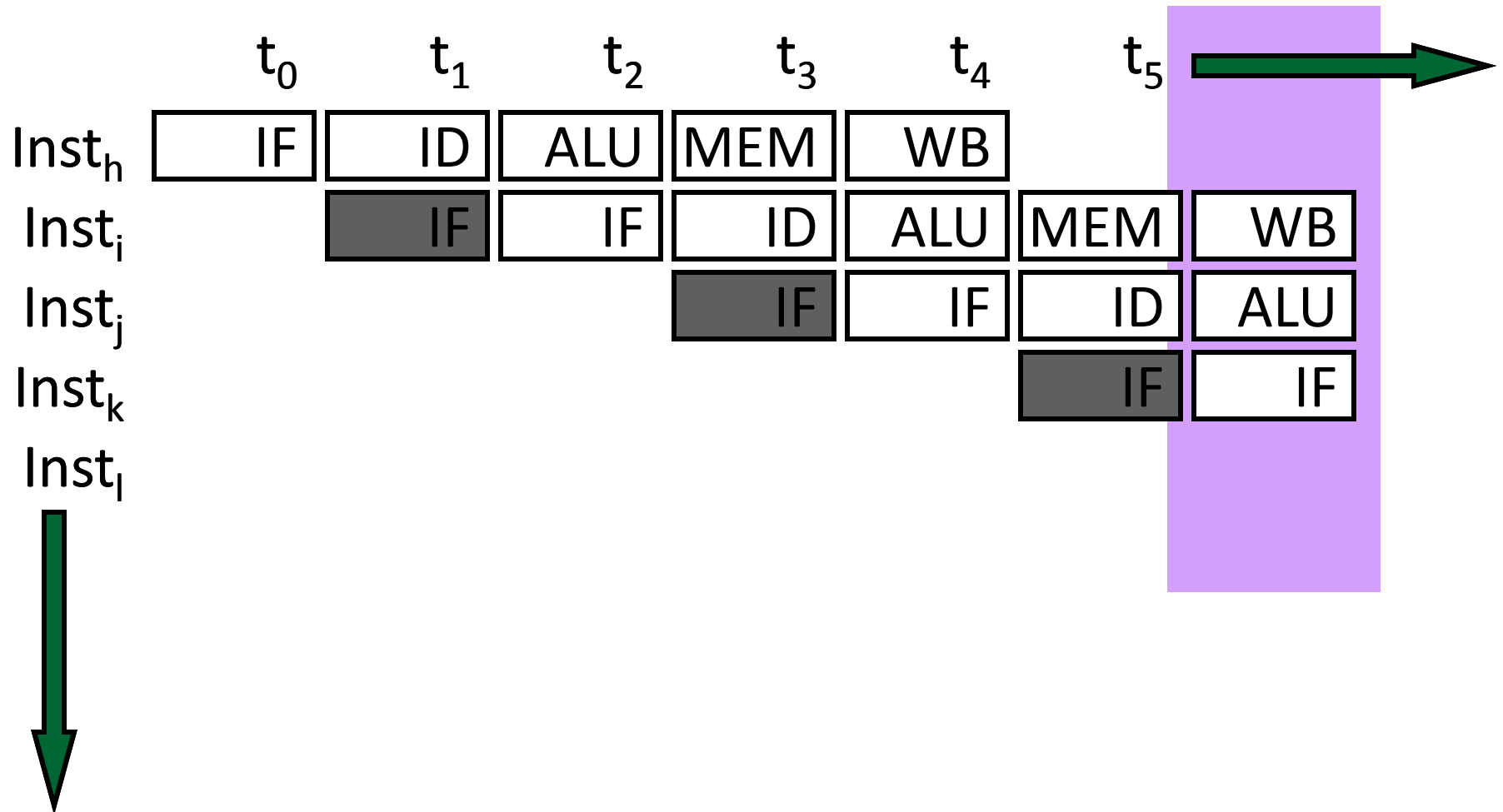
Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:
 - **Stall** the pipeline until we know the next fetch address
 - Guess the next fetch address (**branch prediction**)
 - Employ delayed branching (**branch delay slot**)
 - Do something else (**fine-grained multithreading**)
 - Eliminate control-flow instructions (**predicated execution**)
 - Fetch from both possible paths (if you know the addresses of both possible paths) (**multipath execution**)

Stall Fetch Until Next PC is Available: Good Idea?



This is the case with non-control-flow and unconditional br instructions!

Doing Better than Stalling Fetch ...

- Rather than waiting for true-dependence on PC to resolve, just guess $\text{nextPC} = \text{PC} + 4$ to keep fetching every cycle

Is this a good guess?

What do you lose if you guessed incorrectly?

- ~20% of the instruction mix is control flow
 - ~50 % of “forward” control flow (i.e., if-then-else) is taken
 - ~90% of “backward” control flow (i.e., loop back) is taken

Overall, typically ~70% taken and ~30% not taken

[Lee and Smith, 1984]

- Expect “ $\text{nextPC} = \text{PC} + 4$ ” ~86% of the time, but what about the remaining 14%?

Guessing $\text{NextPC} = \text{PC} + 4$

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of next fetch address prediction and branch prediction
- How can you make this more effective?
- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
 - Software: Lay out the control flow graph such that the “likely next instruction” is on the not-taken path of a branch
 - Hardware: ??? (how can you do this in hardware...)

Guessing $\text{NextPC} = \text{PC} + 4$

- How else can you make this more effective?
- Idea: Get rid of control flow instructions (or minimize their occurrence)
- How?
 1. Get rid of unnecessary control flow instructions → combine predicates (predicate combining)
 2. Convert control dependences into data dependences → predicated execution

Predicate Combining (*not* Predicated Execution)

- Complex predicates are converted into multiple branches
 - `if ((a == b) && (c < d) && (a > 5000)) { ... }`
 - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction instead of having one branch for each
 - Predicates stored and operated on using condition registers
 - A single branch checks the value of the combined predicate
- + Fewer branches in code → fewer mipredictions/stalls
- Possibly unnecessary work
 - If the first predicate is false, no need to compute other predicates
- Condition registers exist in IBM RS6000 and the POWER architecture

Predicated Execution

- Idea: Convert control dependence to data dependence
- Suppose we had a Conditional Move instruction...
 - CMOV condition, $R1 \leftarrow R2$
 - $R1 = (\text{condition} == \text{true}) ? R2 : R1$
 - Employed in most modern ISAs (x86, Alpha)
- Code example with branches vs. CMOVs
if (a == 5) {b = 4;} else {b = 3;}

CMPEQ condition, a, 5;

CMOV condition, b \leftarrow 4;

CMOV !condition, b \leftarrow 3;

Predicated Execution

- Eliminates branches → enables straight line code (i.e., larger basic blocks in code)
- Advantages
 - Always-not-taken prediction works better (no branches)
 - Compiler has more freedom to optimize code (no branches)
 - control flow does not hinder inst. reordering optimizations
 - code optimizations hindered only by data dependencies
- Disadvantages
 - Useless work: some instructions fetched/executed but discarded (especially bad for easy-to-predict branches)
 - Requires additional ISA support
- Can we eliminate all branches this way?

Predicated Execution

- We will get back to this...
- Some readings (optional):
 - Allen et al., “Conversion of control dependence to data dependence,” POPL 1983.
 - Kim et al., “Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution,” MICRO 2005.

How to Handle Control Dependences

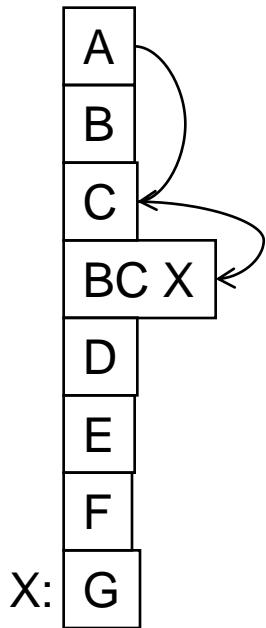
- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:
 - Stall the pipeline until we know the next fetch address
 - Guess the next fetch address (branch prediction)
 - Employ delayed branching (branch delay slot)
 - Do something else (fine-grained multithreading)
 - Eliminate control-flow instructions (predicated execution)
 - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Delayed Branching (I)

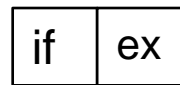
- Change the semantics of a branch instruction
 - Branch after N instructions
 - Branch after N cycles
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.
- Problem: How do you find instructions to fill the delay slots?
 - Branch must be independent of delay slot instructions
- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot

Delayed Branching (II)

Normal code:



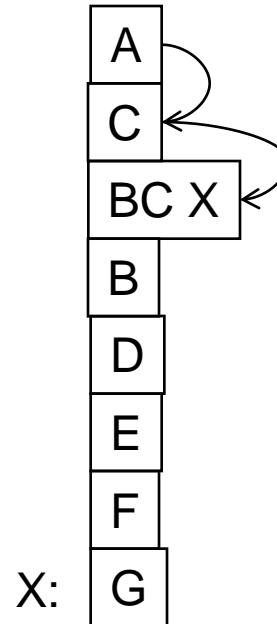
Timeline:



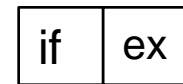
A
B A
C B
BC C
-- BC
G --

6 cycles

Delayed branch code:



Timeline:



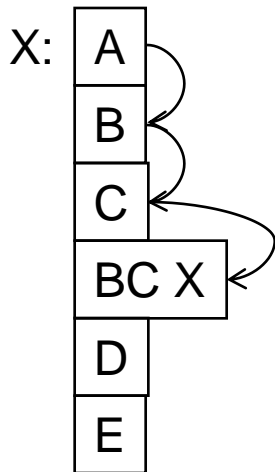
A
C A
BC C
B BC
G B

5 cycles

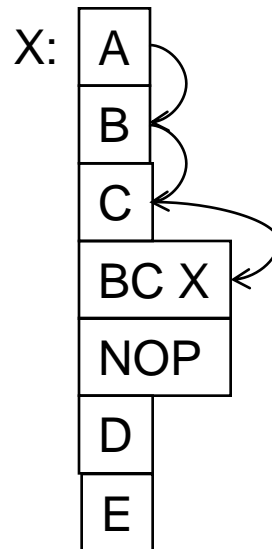
Fancy Delayed Branching (III)

- Delayed branch with squashing
 - In SPARC
 - If the branch falls through (not taken), the delay slot instruction is not executed
 - Why could this help?

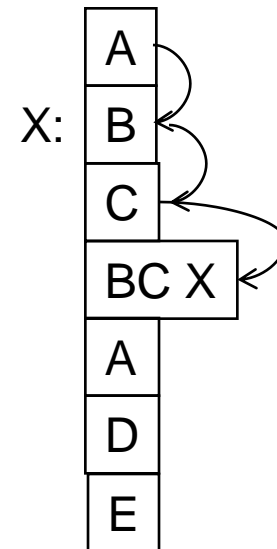
Normal code:



Delayed branch code:



Delayed branch w/ squashing:



Delayed Branching (IV)

■ Advantages:

+ Keeps the pipeline full with useful instructions in a simple way assuming

1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves

2. All delay slots can be filled with useful instructions

■ Disadvantages:

-- Not easy to fill the delay slots (even with a 2-stage pipeline)

1. Number of delay slots increases with pipeline depth, superscalar execution width

2. Number of delay slots should be variable with variable latency operations. Why?

-- Ties ISA semantics to hardware implementation

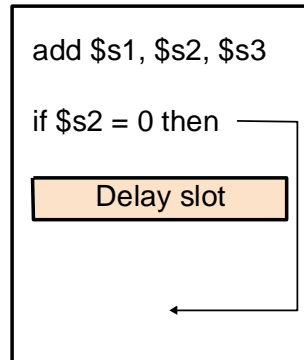
-- SPARC, MIPS, HP-PA: 1 delay slot

-- What if pipeline implementation changes with the next design?

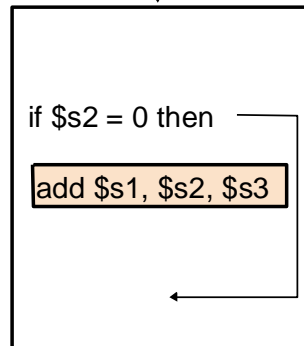
An Aside: Filling the Delay Slot

reordering data
independent
(RAW, WAW,
WAR)
instructions
does not change
program semantics

a. From before

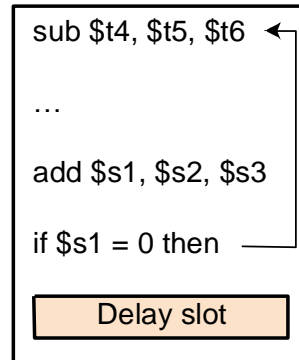


Becomes

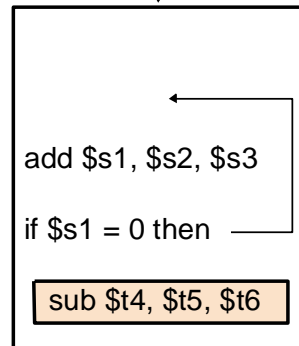


within same
basic block

b. From target

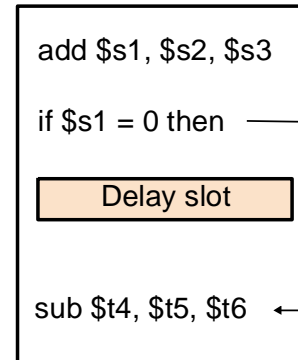


Becomes

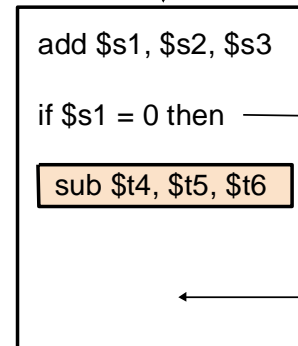


a new
instruction
added to not-
taken path??

c. From fall through



Becomes



a new
instruction
added to
taken??

Safe?

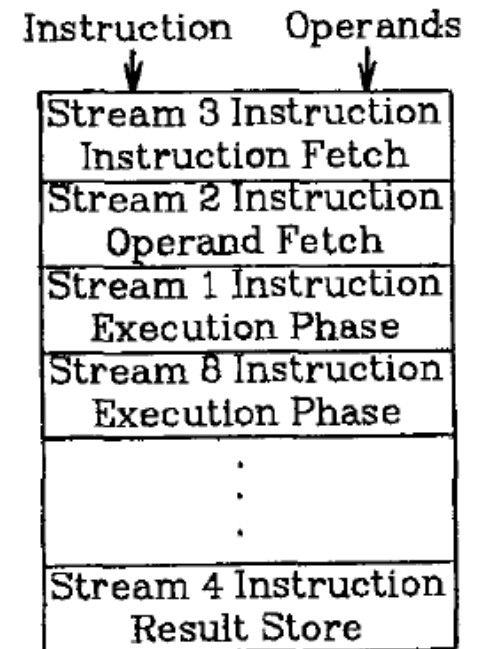
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:
 - Stall the pipeline until we know the next fetch address
 - Guess the next fetch address (branch prediction)
 - Employ delayed branching (branch delay slot)
 - Do something else (fine-grained multithreading)
 - Eliminate control-flow instructions (predicated execution)
 - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
 - By the time the fetched branch/instruction resolves, there is no need to fetch another instruction from the same thread
 - Branch/instruction resolution latency overlapped with execution of other threads' instructions

- + No logic needed for handling control and data dependences within a thread
- Single thread performance suffers
- Extra logic for keeping thread contexts
- Does not overlap latency if not enough threads to cover the whole pipeline



Fine-grained Multithreading

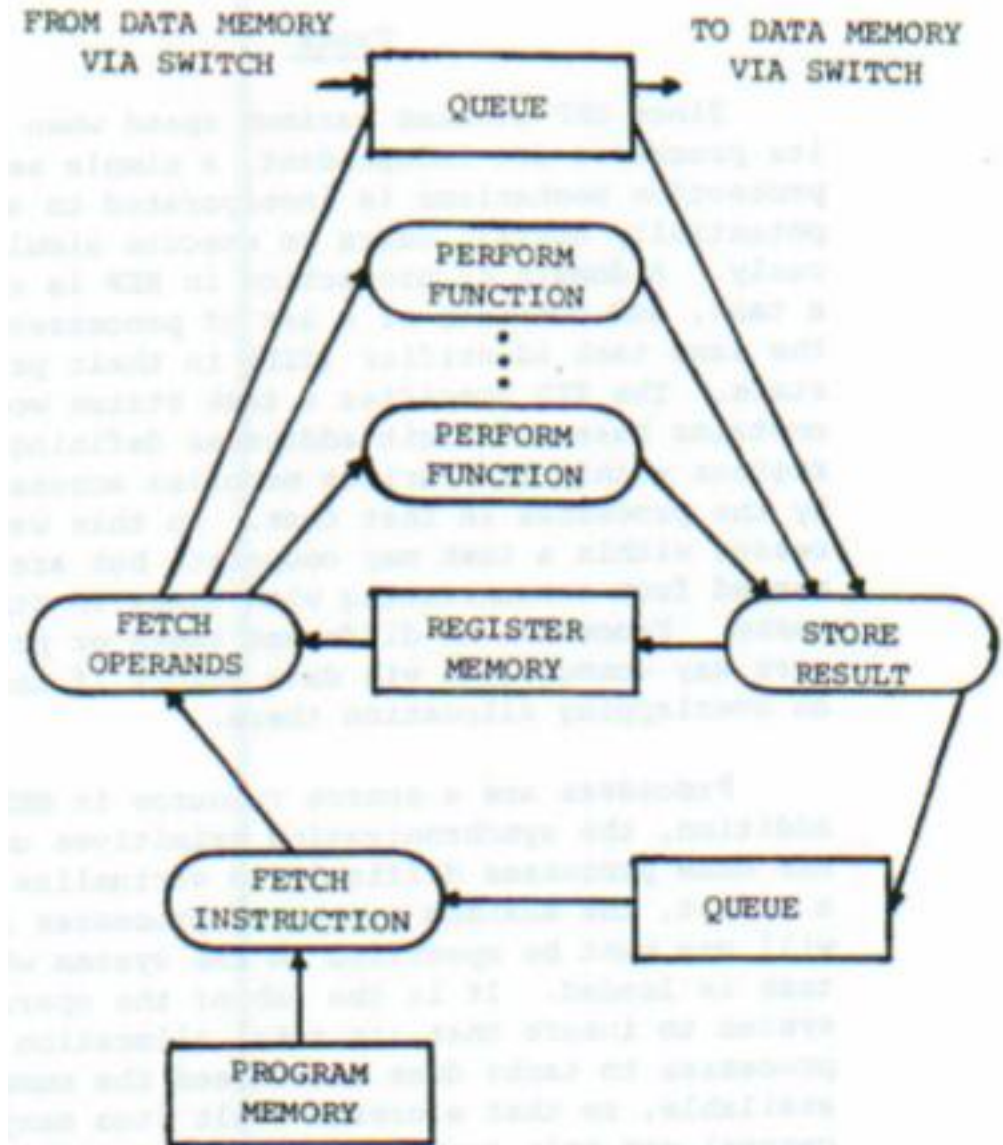
- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently
- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads
- Thornton, “Parallel Operation in the Control Data 6600,” AFIPS 1964.
- Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.

Fine-grained Multithreading: History

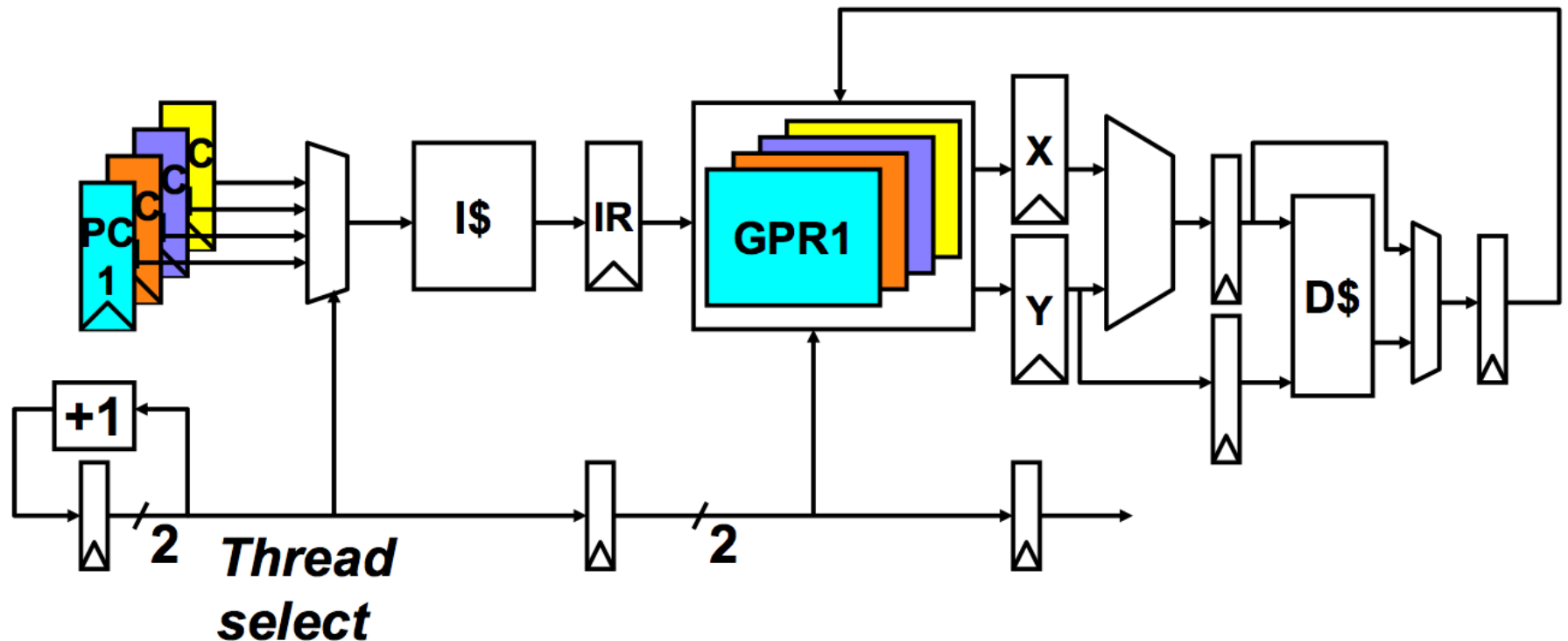
- CDC 6600's peripheral processing unit is fine-grained multithreaded
 - Thornton, “[Parallel Operation in the Control Data 6600](#),” AFIPS 1964.
 - Processor executes a different I/O thread every cycle
 - An operation from the same thread is executed every 10 cycles
- Denelcor HEP (Heterogeneous Element Processor)
 - Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978.
 - 120 threads/processor
 - available queue vs. unavailable (waiting) queue for threads
 - each thread can only have 1 instruction in the processor pipeline; each thread independent
 - to each thread, processor looks like a non-pipelined machine
 - system throughput vs. single thread performance tradeoff

Fine-grained Multithreading in HEP

- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
 - assuming no memory access

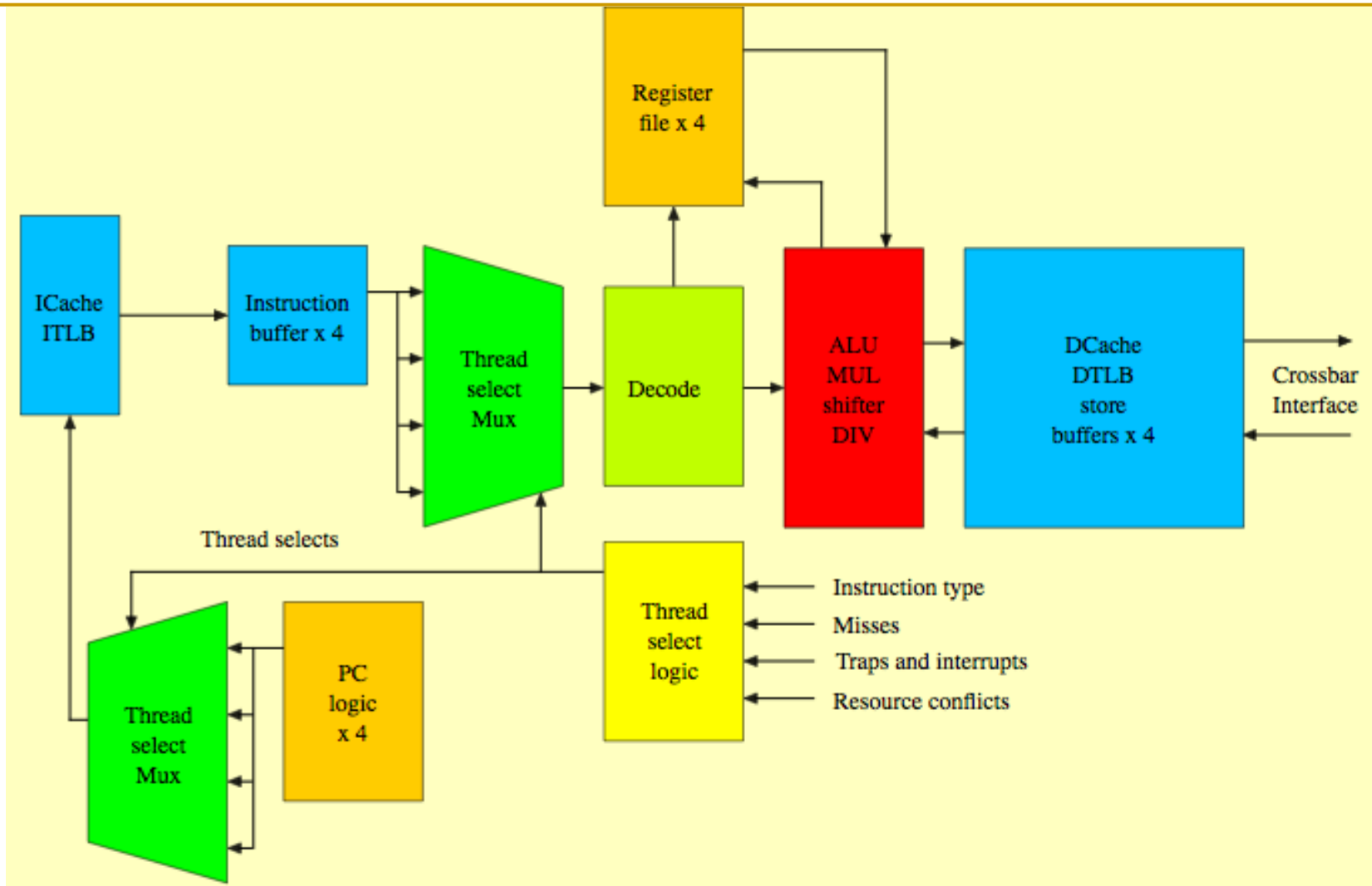


Multithreaded Pipeline Example



- Slide from Joel Emer

Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

Fine-grained Multithreading

■ Advantages

- + No need for dependency checking between instructions
(only one instruction in pipeline from a single thread)
- + No need for branch prediction logic
- + Otherwise-bubble cycles used for executing useful instructions from different threads
- + Improved system throughput, latency tolerance, utilization

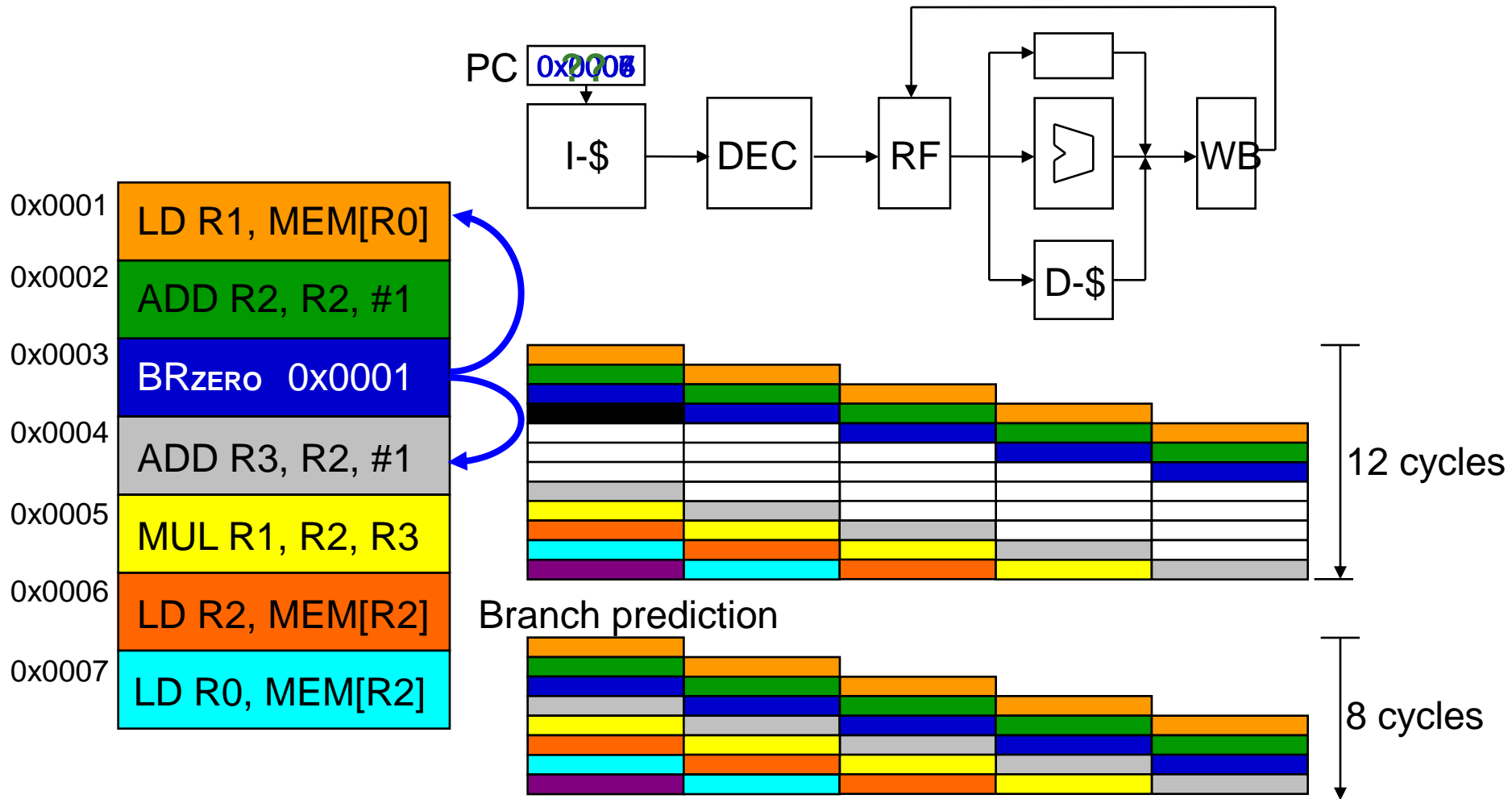
■ Disadvantages

- Extra hardware complexity: multiple hardware contexts, thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles)
- Resource contention between threads in caches and memory
- Some dependency checking logic between threads remains (load/store)

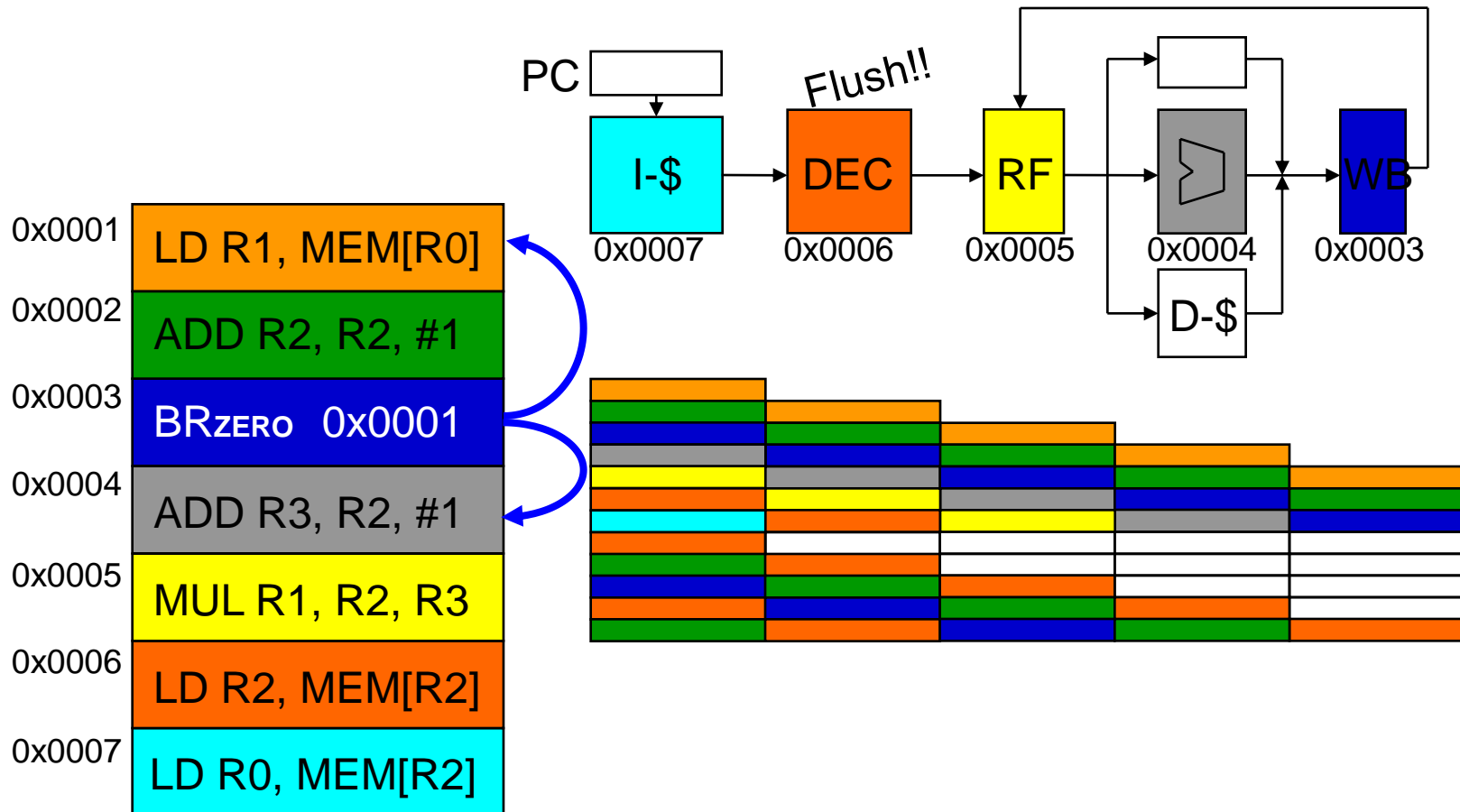
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:
 - Stall the pipeline until we know the next fetch address
 - Guess the next fetch address (branch prediction)
 - Employ delayed branching (branch delay slot)
 - Do something else (fine-grained multithreading)
 - Eliminate control-flow instructions (predicated execution)
 - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Branch Prediction: Guess the Next Instruction to Fetch

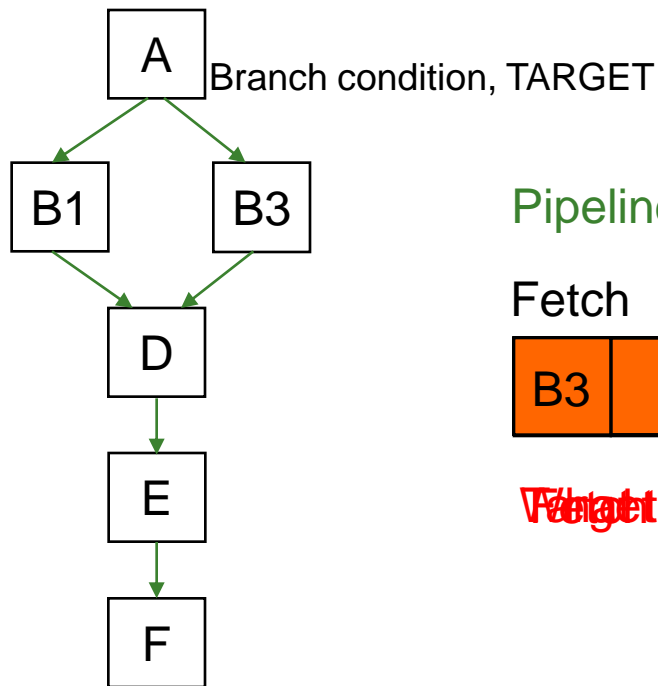


Misprediction Penalty



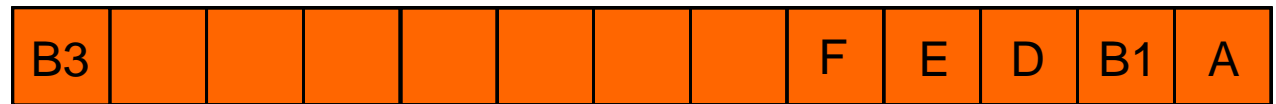
Branch Prediction

- Processors are pipelined to increase concurrency
- How do we **keep the pipeline full** in the presence of branches?
 - ❑ **Guess the next instruction** when a branch is fetched
 - ❑ Requires guessing the direction and target of a branch



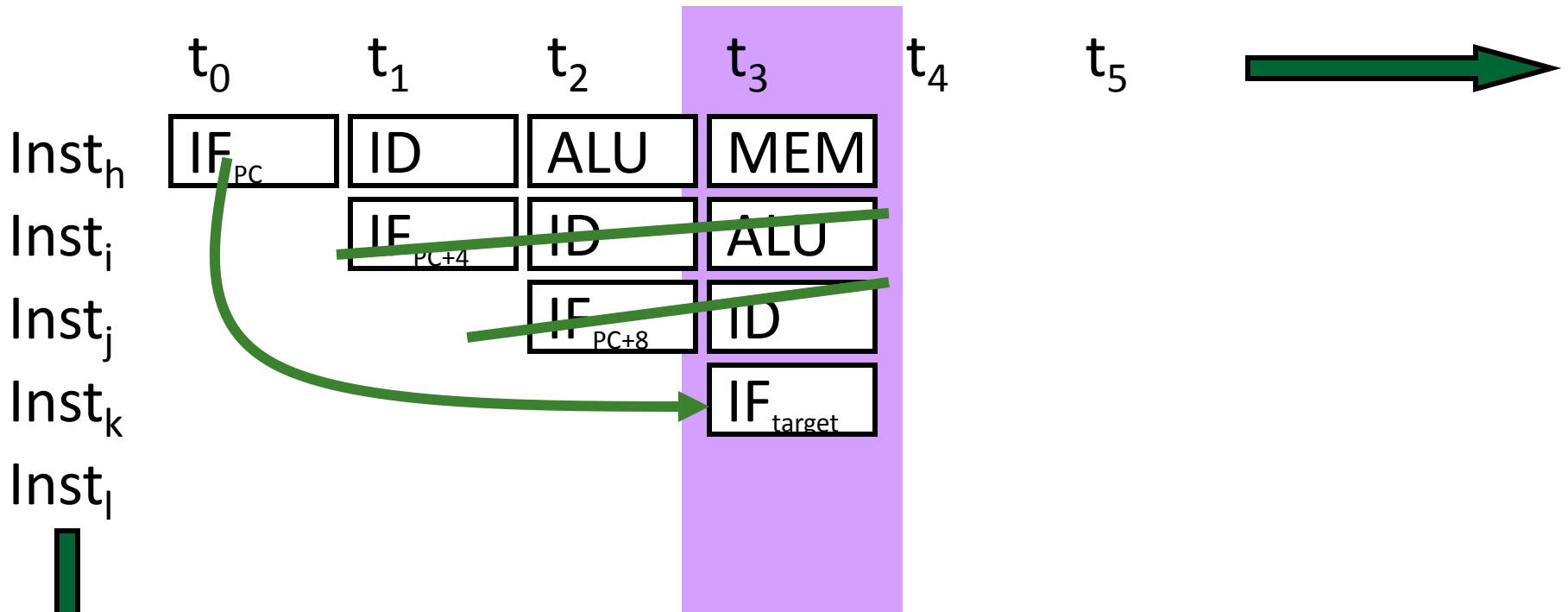
Pipeline

Fetch Decode Rename Schedule RegisterRead Execute



Target Misprediction Detected! Flush the pipeline

Branch Prediction: Always PC+4

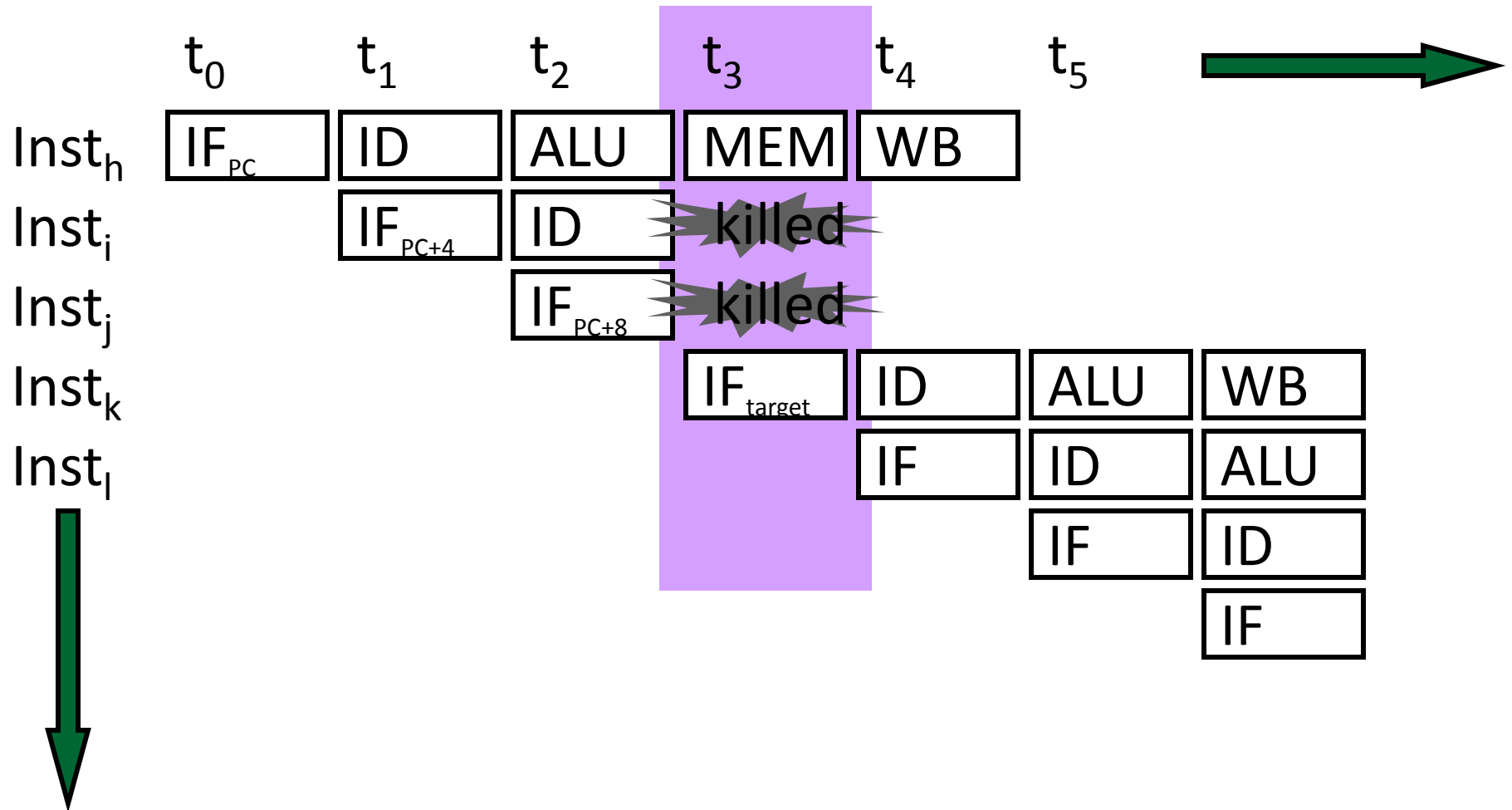


$Inst_h$ is a branch

When a branch resolves

- branch target ($Inst_k$) is fetched
- all instructions fetched since $inst_h$ (so called “wrong-path” instructions) must be flushed

Pipeline Flush on a Misprediction



$Inst_h$ is a branch

Performance Analysis

■ correct guess \Rightarrow no penalty ~86% of the time

■ incorrect guess \Rightarrow 2 bubbles

■ Assume

□ no data hazards

□ 20% control flow instructions

□ 70% of control flow instructions are taken

□ $\text{CPI} = [1 + (0.20 * 0.7) * 2] =$

$$= [1 + 0.14 * 2] = 1.28$$

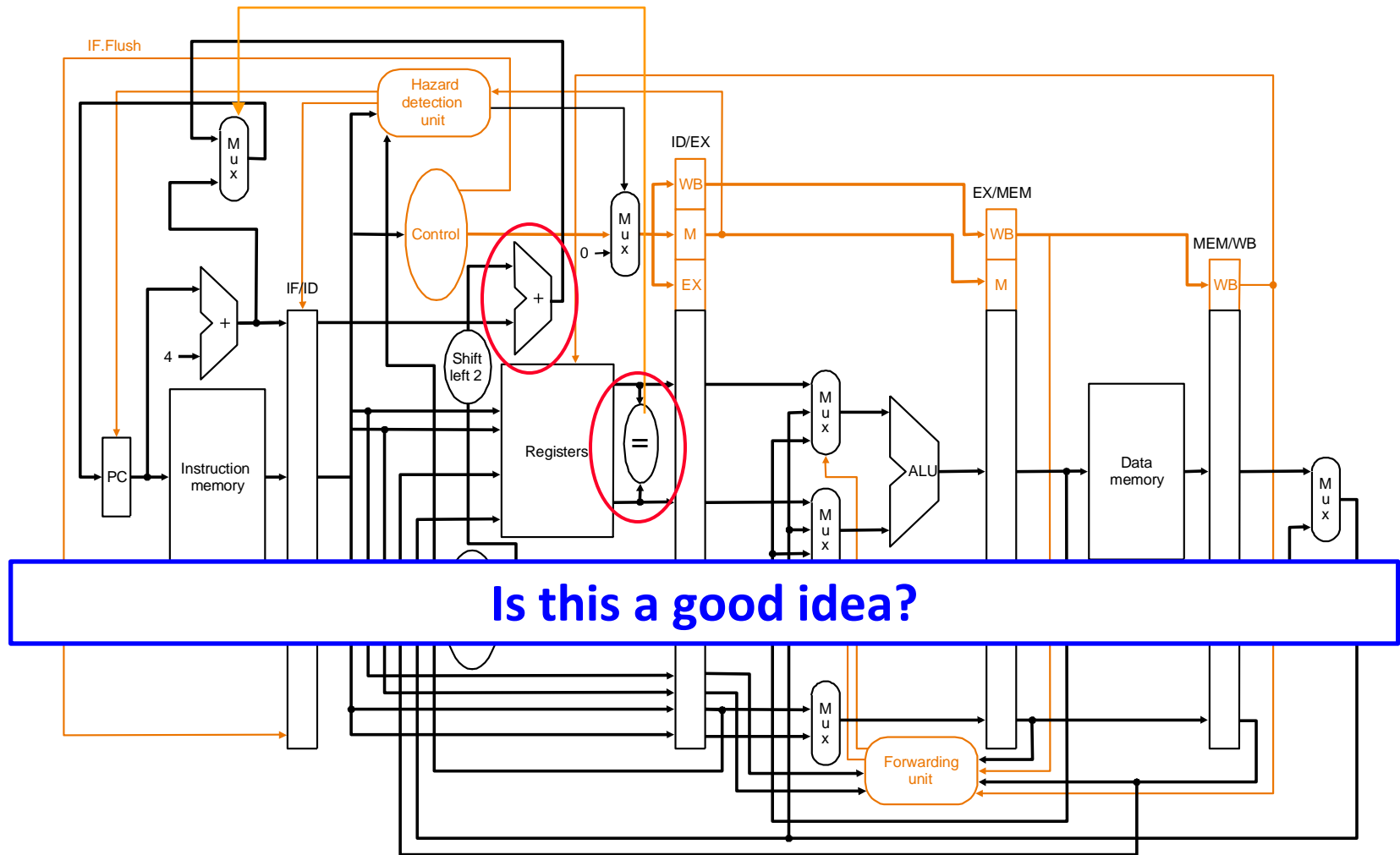
probability of
a wrong guess

penalty for
a wrong guess

Can we reduce either of the two penalty terms?

Reducing Branch Misprediction Penalty

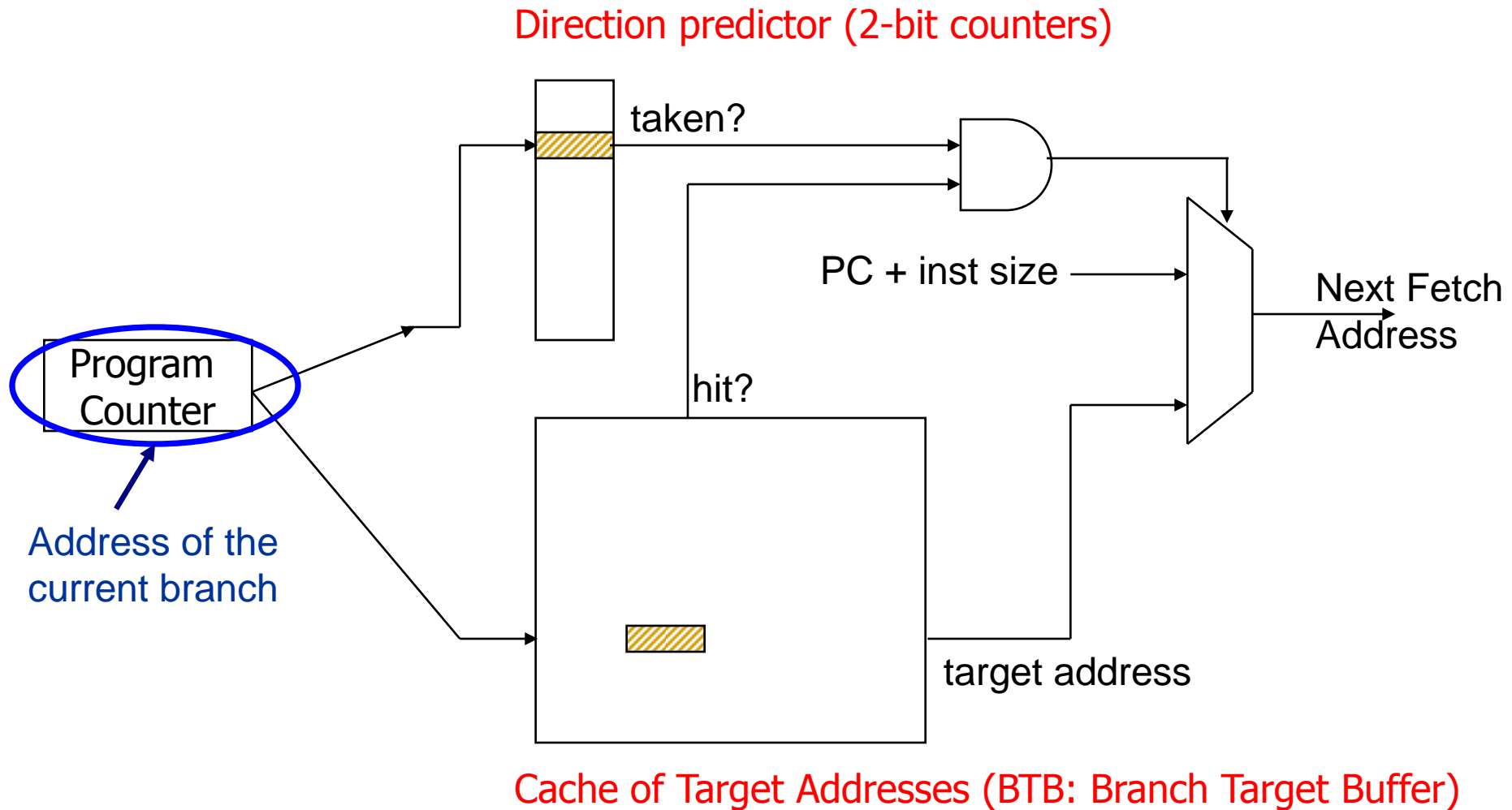
- Resolve branch condition and target address early



Branch Prediction (Enhanced)

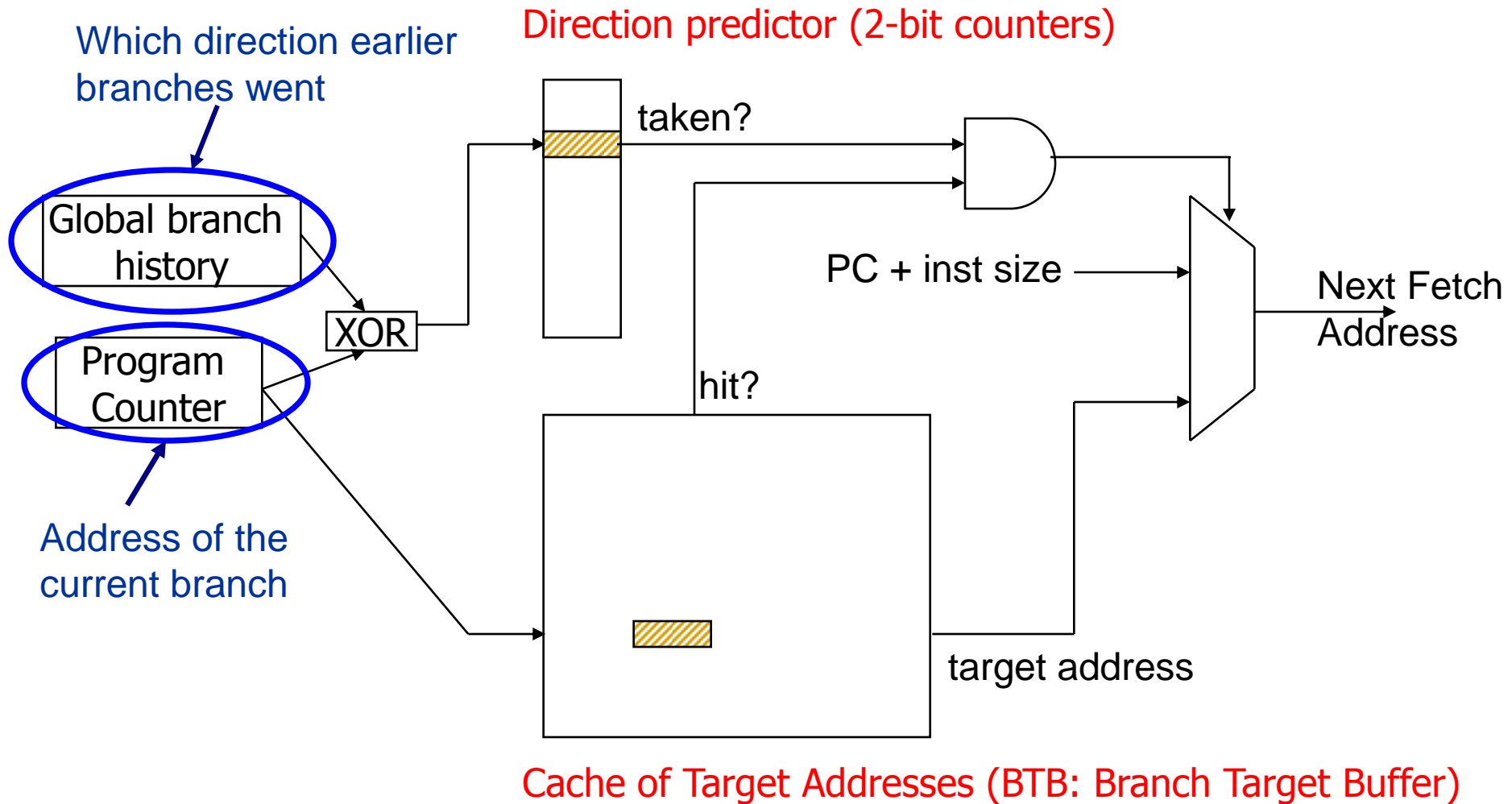
- Idea: Predict the next fetch address (to be used in the next cycle)
- Requires three things to be predicted at fetch stage:
 - Whether the fetched instruction is a branch
 - (Conditional) branch direction
 - Branch target address (if taken)
- Observation: Target address remains the same for a conditional direct branch across dynamic instances
 - Idea: Store the target address from previous instance and access it with the PC
 - Called Branch Target Buffer (BTB) or Branch Target Address Cache

Fetch Stage with BTB and Direction Prediction



Always taken CPI = $[1 + (0.20 * \underline{0.3}) * 2] = 1.12$ (70% of branches taken)

More Sophisticated Branch Direction Prediction



Simple Branch Direction Prediction Schemes

- Compile time (static)
 - Always not taken
 - Always taken
 - BTFN (Backward taken, forward not taken)
 - Profile based (likely direction)
- Run time (dynamic)
 - Last time prediction (single-bit)

More Sophisticated Direction Prediction

- Compile time (static)
 - ❑ Always not taken
 - ❑ Always taken
 - ❑ BTFN (Backward taken, forward not taken)
 - ❑ Profile based (likely direction)
 - ❑ Program analysis based (likely direction)

- Run time (dynamic)
 - ❑ Last time prediction (single-bit)
 - ❑ Two-bit counter based prediction
 - ❑ Two-level prediction (global vs. local)
 - ❑ Hybrid

Static Branch Prediction (I)

■ Always not-taken

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

■ Always taken

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

■ Backward taken, forward not taken (BTFN)

- ❑ Predict backward (loop) branches as taken, others not-taken

Static Branch Prediction (II)

■ Profile-based

- Idea: Compiler determines likely direction for each branch using profile run. Encodes that direction as a hint bit in the branch instruction format.

- + Per branch prediction (more accurate than schemes in previous slide) → accurate if profile is representative!
- Requires hint bits in the branch instruction format
- Accuracy depends on dynamic branch behavior:
 - TTTTTTTTTTTTNNNNNNNNNNNN → 50% accuracy
 - TNTNTNTNTNTNTNTNTNTNTN → 50% accuracy
- Accuracy depends on the representativeness of profile input set

Static Branch Prediction (III)

- **Program-based (or, program analysis based)**
 - Idea: Use heuristics based on program analysis to determine statically-predicted direction
 - Opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
 - Loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)
 - Pointer and FP comparisons: Predict not equal
- + Does not require profiling
- Heuristics might be not representative or good
- Requires compiler analysis and ISA support
- Ball and Larus, "Branch prediction for free," PLDI 1993.
 - 20% misprediction rate

Static Branch Prediction (III)

■ Programmer-based

- Idea: Programmer provides the statically-predicted direction
- Via pragmas in the programming language that qualify a branch as likely-taken versus likely-not-taken

- + Does not require profiling or program analysis
- + Programmer may know some branches and their program better than other analysis techniques
- Requires programming language, compiler, ISA support
- Burdens the programmer?

Aside: Pragmas

- Idea: Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy
- `if (likely(x)) { ... }`
- `if (unlikely(error)) { ... }`
- Many other hints and optimizations can be enabled with pragmas
 - E.g., whether a loop can be parallelized
 - **#pragma omp parallel**
 - **Description**
 - The `omp parallel` directive explicitly instructs the compiler to parallelize the chosen segment of code.

Static Branch Prediction

- All previous techniques can be combined
 - Profile based
 - Program based
 - Programmer based
- How would you do that?
- What are common disadvantages of all three techniques?
 - Cannot adapt to dynamic changes in branch behavior
 - This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its overheads...)

Dynamic Branch Prediction

- Idea: Predict branches based on dynamic information (collected at run-time)
- Advantages
 - + Prediction based on history of the execution of branches
 - + It can adapt to dynamic changes in branch behavior
 - + No need for static profiling: input set representativeness problem goes away
- Disadvantages
 - More complex (requires additional hardware)

Last Time Predictor

- Last time predictor

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

- Always mispredicts the last iteration and the first iteration of a loop branch

- Accuracy for a loop with N iterations = $(N-2)/N$

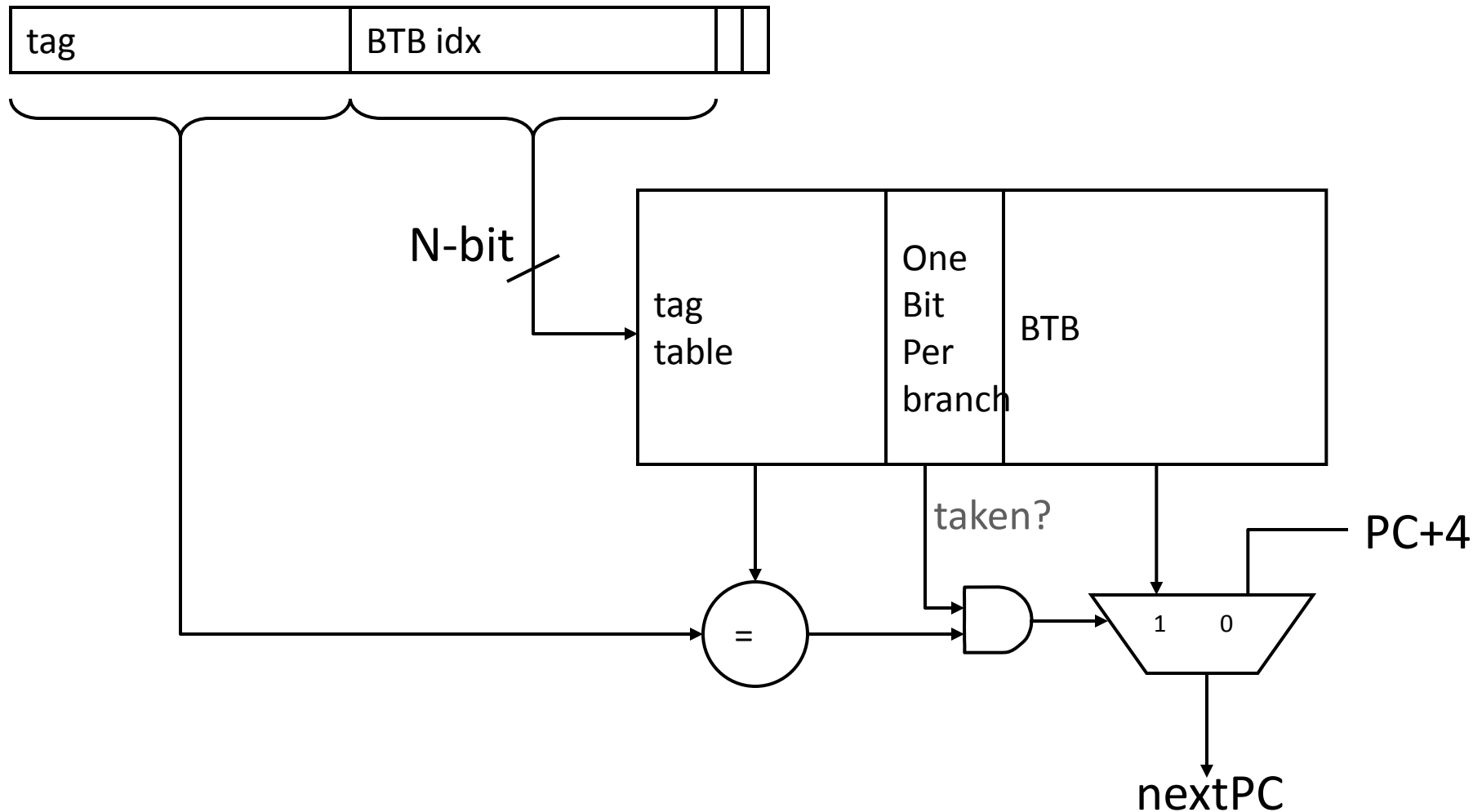
+ Loop branches for loops with large number of iterations

-- Loop branches for loops will small number of iterations

TNTNTNTNTNTNTNTNTN → 0% accuracy

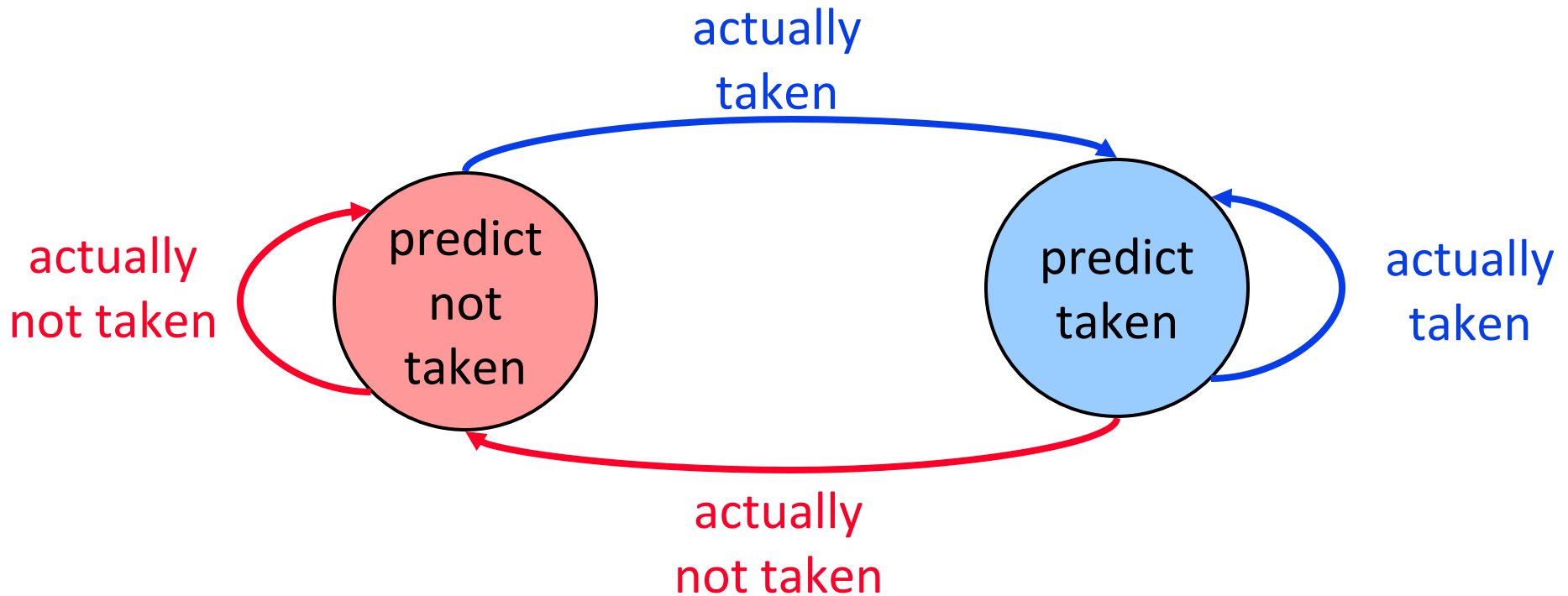
Last-time predictor CPI = $[1 + (0.20 * 0.15) * 2] = 1.06$ (Assuming 85% accuracy)

Implementing the Last-Time Predictor



The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch

State Machine for Last-Time Prediction



Improving the Last Time Predictor

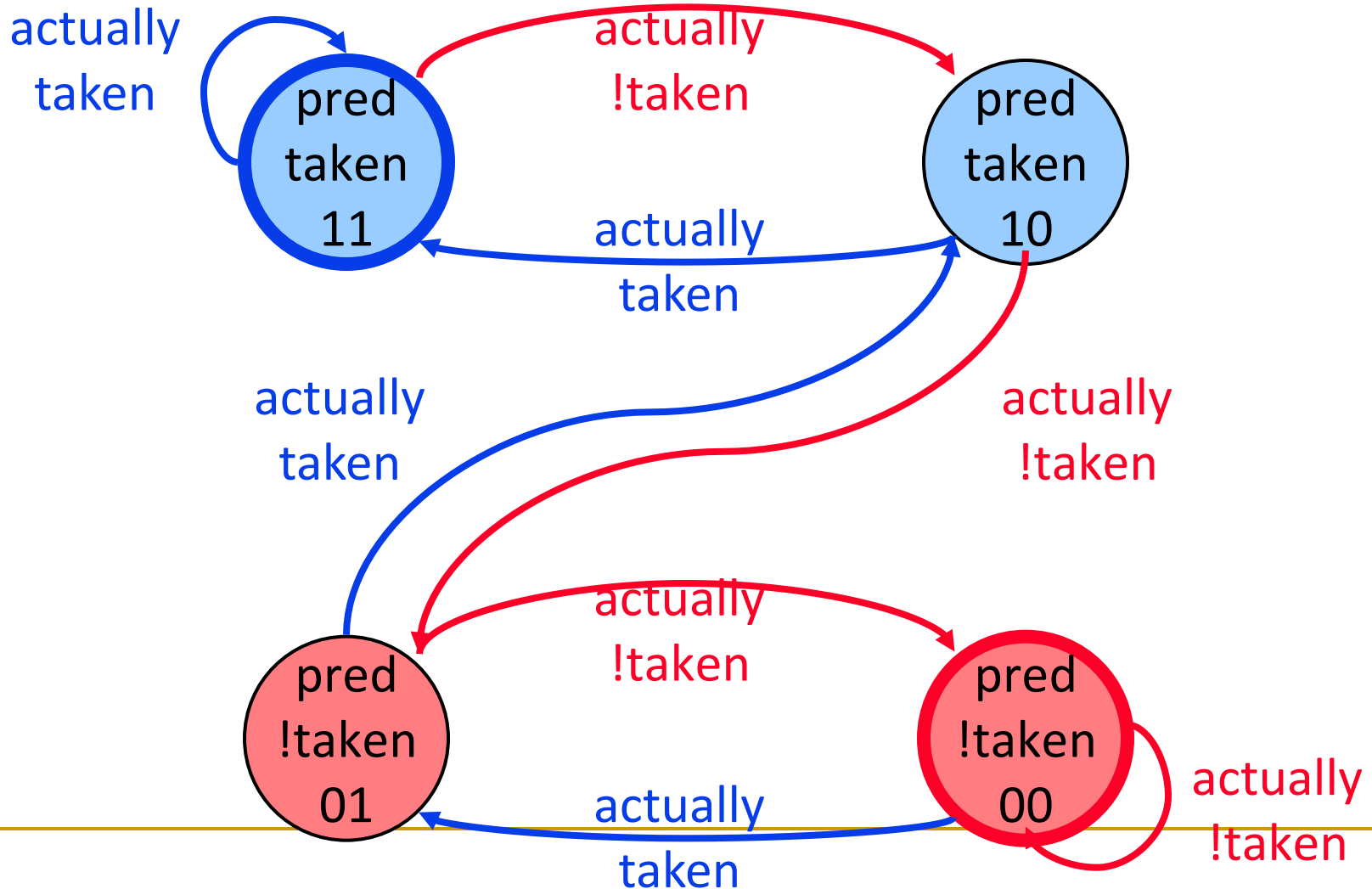
- Problem: A last-time predictor changes its prediction from $T \rightarrow NT$ or $NT \rightarrow T$ too quickly
 - even though the branch may be mostly taken or mostly not taken
- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
 - Use two bits to track the history of predictions for a branch instead of a single bit
 - Can have 2 states for T or NT instead of 1 state for each
- Smith, "A Study of Branch Prediction Strategies," ISCA 1981.

Two-Bit Counter Based Prediction

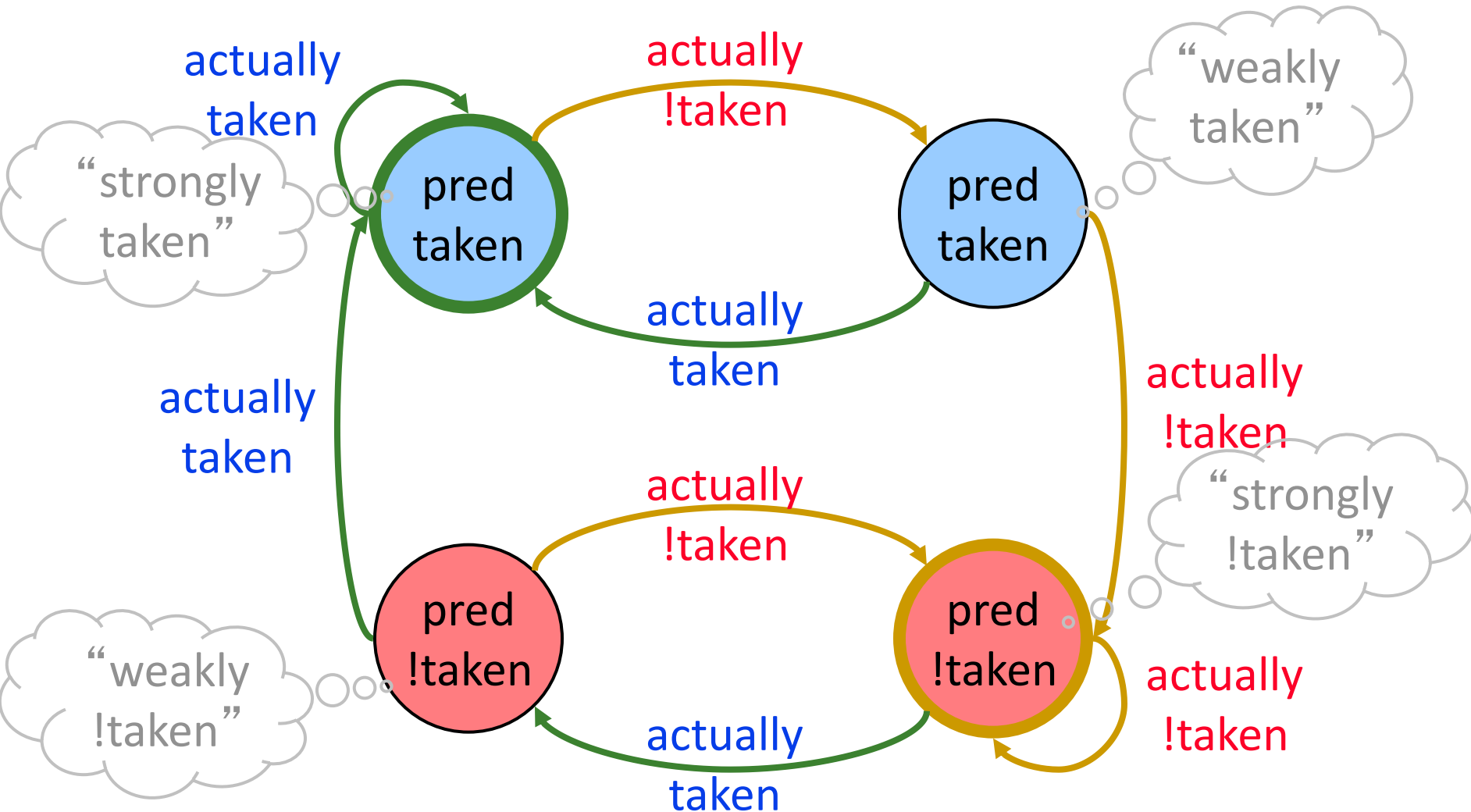
- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome
- Accuracy for a loop with N iterations = $(N-1)/N$
TNTNTNTNTNTNTNTNTN → 50% accuracy
(assuming init to weakly taken)
- + Better prediction accuracy
2BC predictor CPI = $[1 + (0.20 * \underline{0.10}) * 2] = 1.04$ (90% accuracy)
- More hardware cost (but counter can be part of a BTB entry)

State Machine for 2-bit Saturating Counter

- Counter using saturating arithmetic
 - There is a symbol for maximum and minimum values



Hysteresis Using a 2-bit Counter



Change prediction after 2 consecutive mistakes

Is This Enough?

- ~85-90% accuracy for many programs with 2-bit counter based prediction (also called bimodal prediction)
- Is this good enough?
- How big is the branch problem?

Rethinking the The Branch Problem

- Control flow instructions (branches) are frequent
 - 15-25% of all instructions
- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
 - N cycles: (minimum) branch resolution latency
 - Stalling on a branch wastes instruction processing bandwidth (i.e. reduces IPC)
 - $N \times \text{IW}$ instruction slots are wasted (IW: issue width)
- How do we keep the pipeline full after a branch?
- Problem: Need to determine the **next fetch address** when the branch is fetched (to avoid a pipeline bubble)

Importance of The Branch Problem

- Assume a 5-wide *superscalar* pipeline with 20-cycle branch resolution latency
- How long does it take to fetch 500 instructions?
 - Assume no fetch breaks and 1 out of 5 instructions is a branch
 - 100% accuracy
 - 100 cycles (all instructions fetched on the correct path)
 - No wasted work
 - 99% accuracy
 - $100 \text{ (correct path)} + 20 \text{ (wrong path)} = 120 \text{ cycles}$
 - 20% extra instructions fetched
 - 98% accuracy
 - $100 \text{ (correct path)} + 20 * 2 \text{ (wrong path)} = 140 \text{ cycles}$
 - 40% extra instructions fetched
 - 95% accuracy
 - $100 \text{ (correct path)} + 20 * 5 \text{ (wrong path)} = 200 \text{ cycles}$
 - 100% extra instructions fetched

Can We Do Better?

- Last-time and 2BC predictors exploit “last-time” predictability
- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
 - Global branch correlation
- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
 - Local branch correlation

Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch

```
if (cond1)
...
if (cond1 AND cond2)
```

- If first branch not taken, second also not taken

```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

- If first branch taken, second definitely not taken

Global Branch Correlation (II)

branch Y: if (cond1)

...

branch Z: if (cond2)

...

branch X: if (cond1 AND cond2)

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken

Global Branch Correlation (III)

■ Eqntott, SPEC 1992

```
if (aa==2)                ;; B1
    aa=0;
if (bb==2)                ;; B2
    bb=0;
if (aa!=bb) {             ;; B3
    ....
}
```

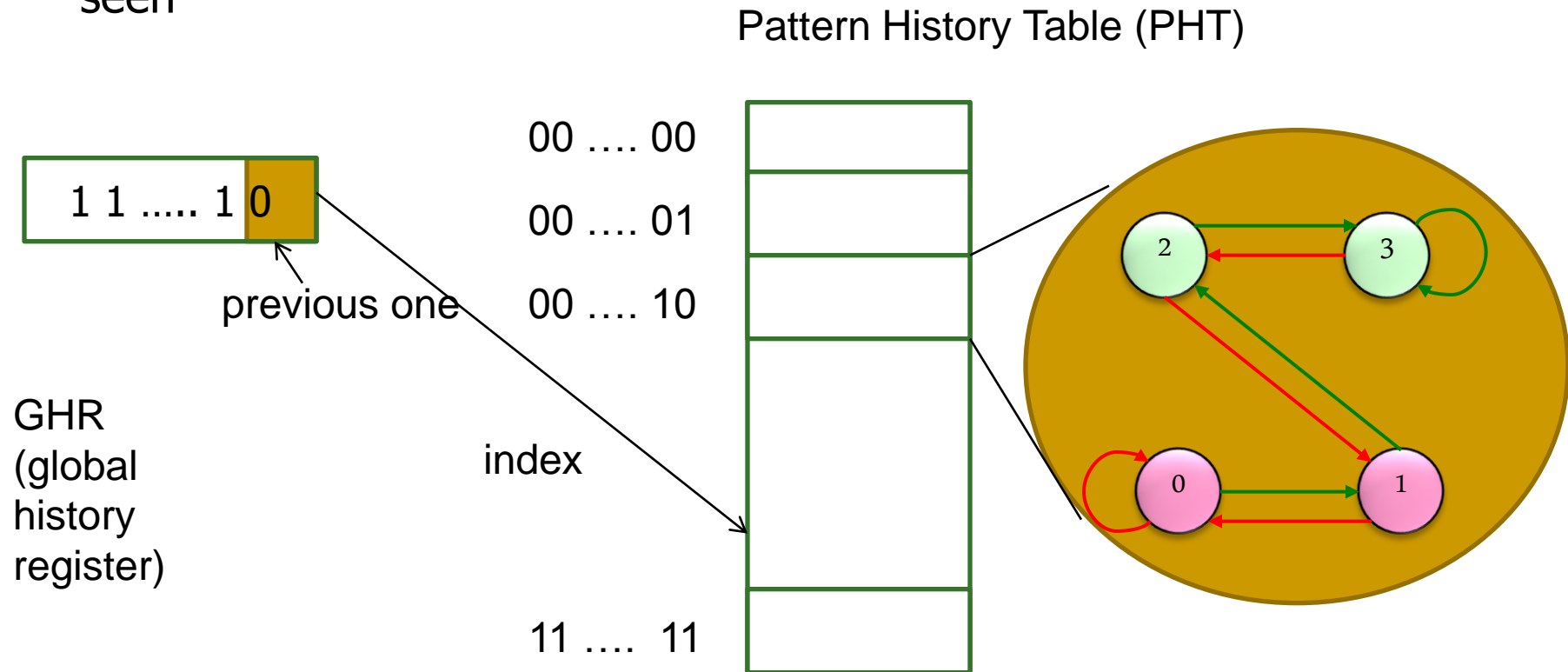
If **B1** is not taken (i.e. `aa==0@B3`) and **B2** is not taken (i.e. `bb=0@B3`)
then **B3** is certainly taken

Capturing Global Branch Correlation

- Idea: Associate branch outcomes with “global T/NT history” of all branches
- Make a prediction based on the outcome of the branch the last time the same global branch history was encountered
- Implementation:
 - Keep track of the “global T/NT history” of all branches in a register → Global History Register (GHR)
 - Use GHR to index into a table of that recorded the outcome that was seen for that GHR value in the recent past → Pattern History Table (table of 2-bit counters)
- Global history/branch predictor
- Uses two levels of history (GHR + history at that GHR)

Two Level Global Branch Prediction

- First level: **Global branch history register** (N bits)
 - The direction of last N branches
- Second level: **Table of saturating counters for each history entry**
 - The direction the branch took the last time the same history was seen



How Does the Global Predictor Work?

```
for (i=0; i<100; i++)  
    for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

test	value	GR	result
j<3	j=1	1101	taken
j<3	j=2	1011	taken
j<3	j=3	0111	not taken
i<100		1110	usually taken

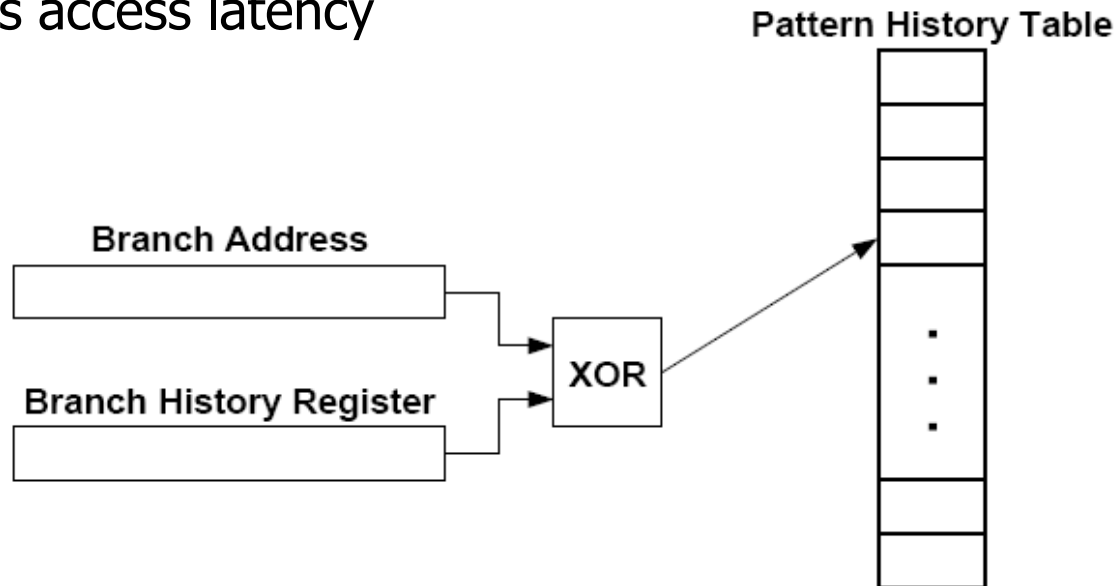
- McFarling, “Combining Branch Predictors,” DEC WRL TR 1993.

Intel Pentium Pro Branch Predictor

- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
 - Which pattern history table to use is determined by lower order bits of the branch address

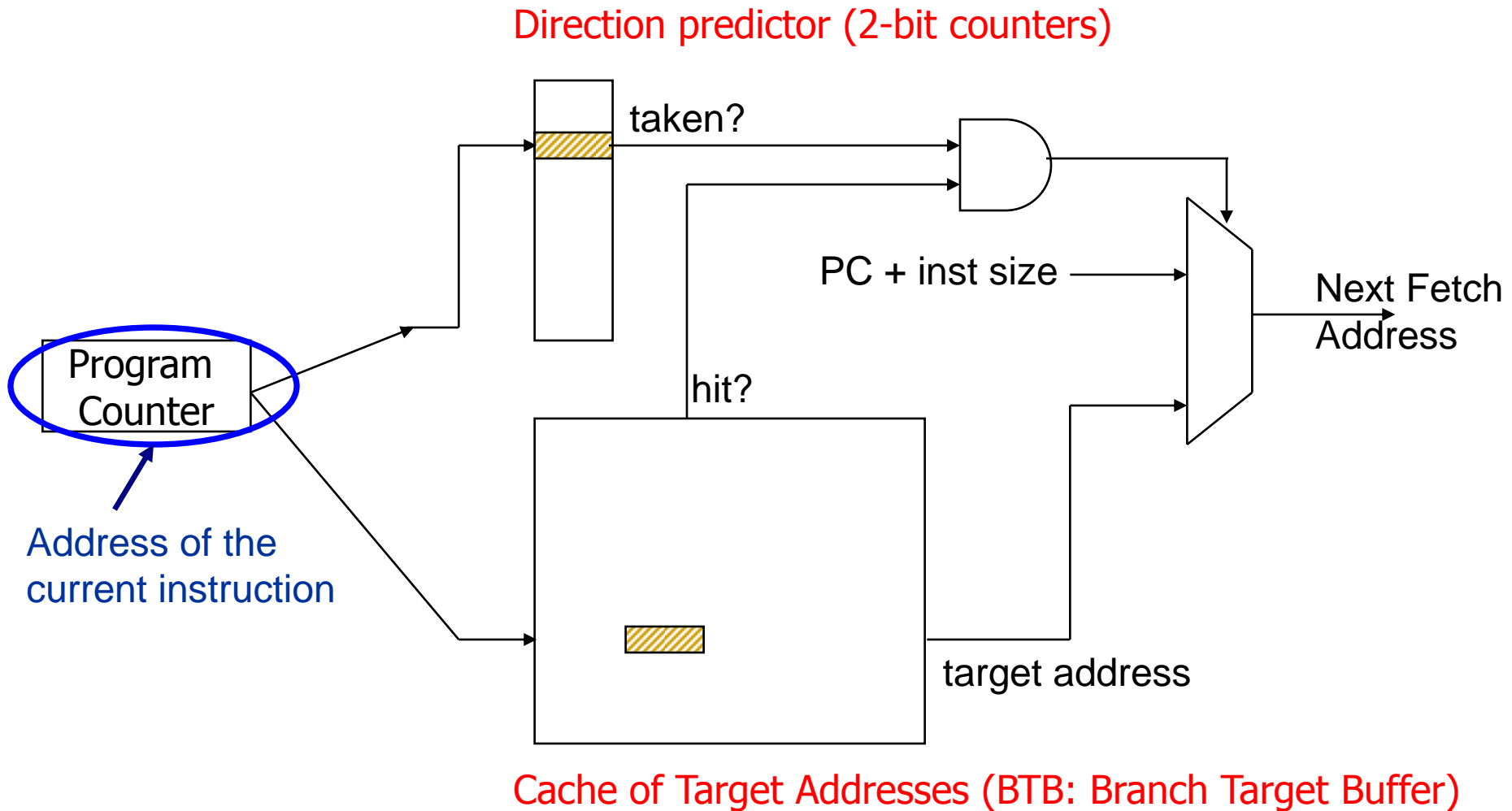
Improving Global Predictor Accuracy

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
 - **Gshare predictor**: GHR hashed with the Branch PC
 - + More context information
 - + Better utilization of PHT
 - Increases access latency

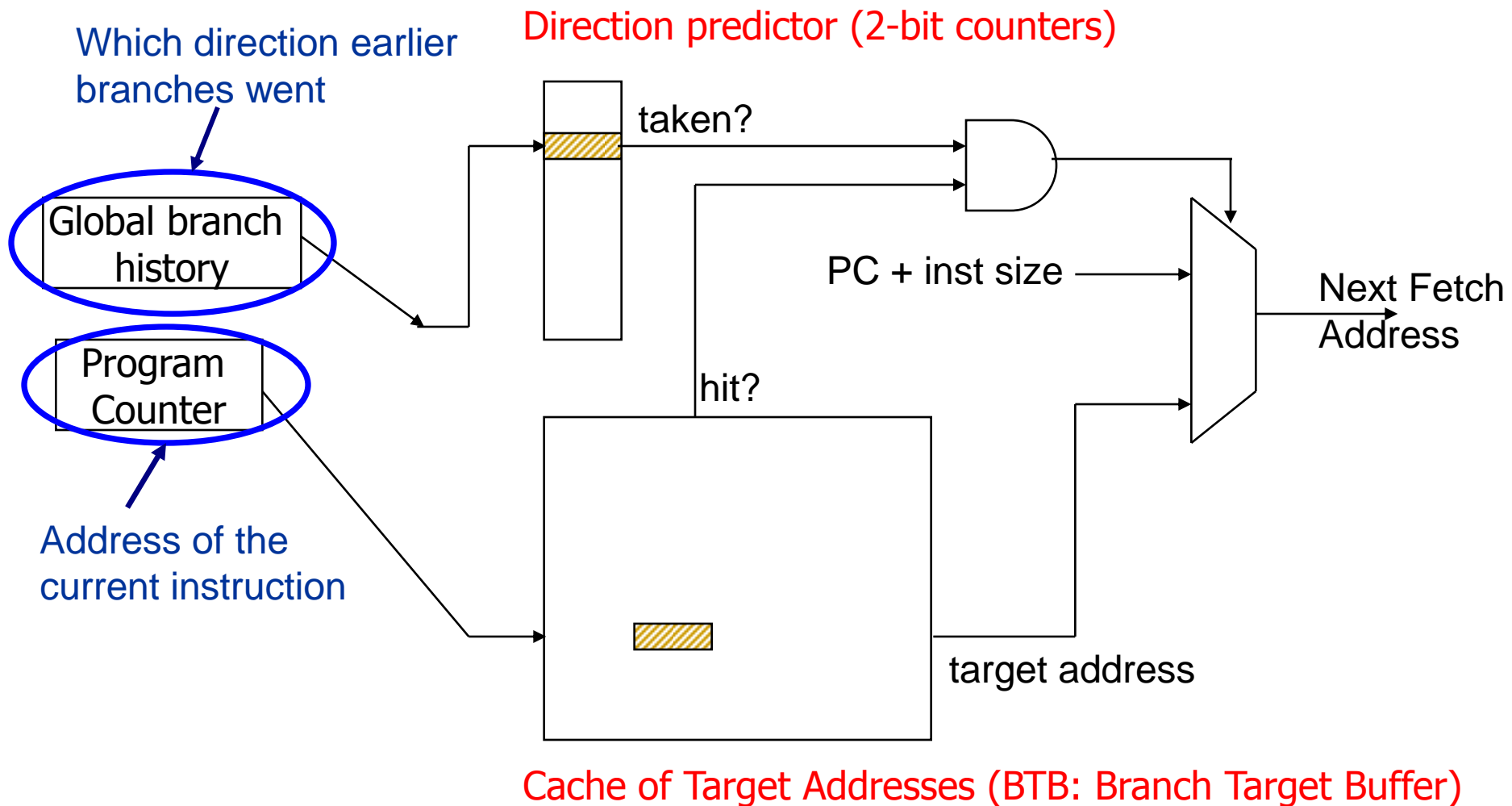


- McFarling, “**Combining Branch Predictors**,” DEC WRL Tech Report, 1993.

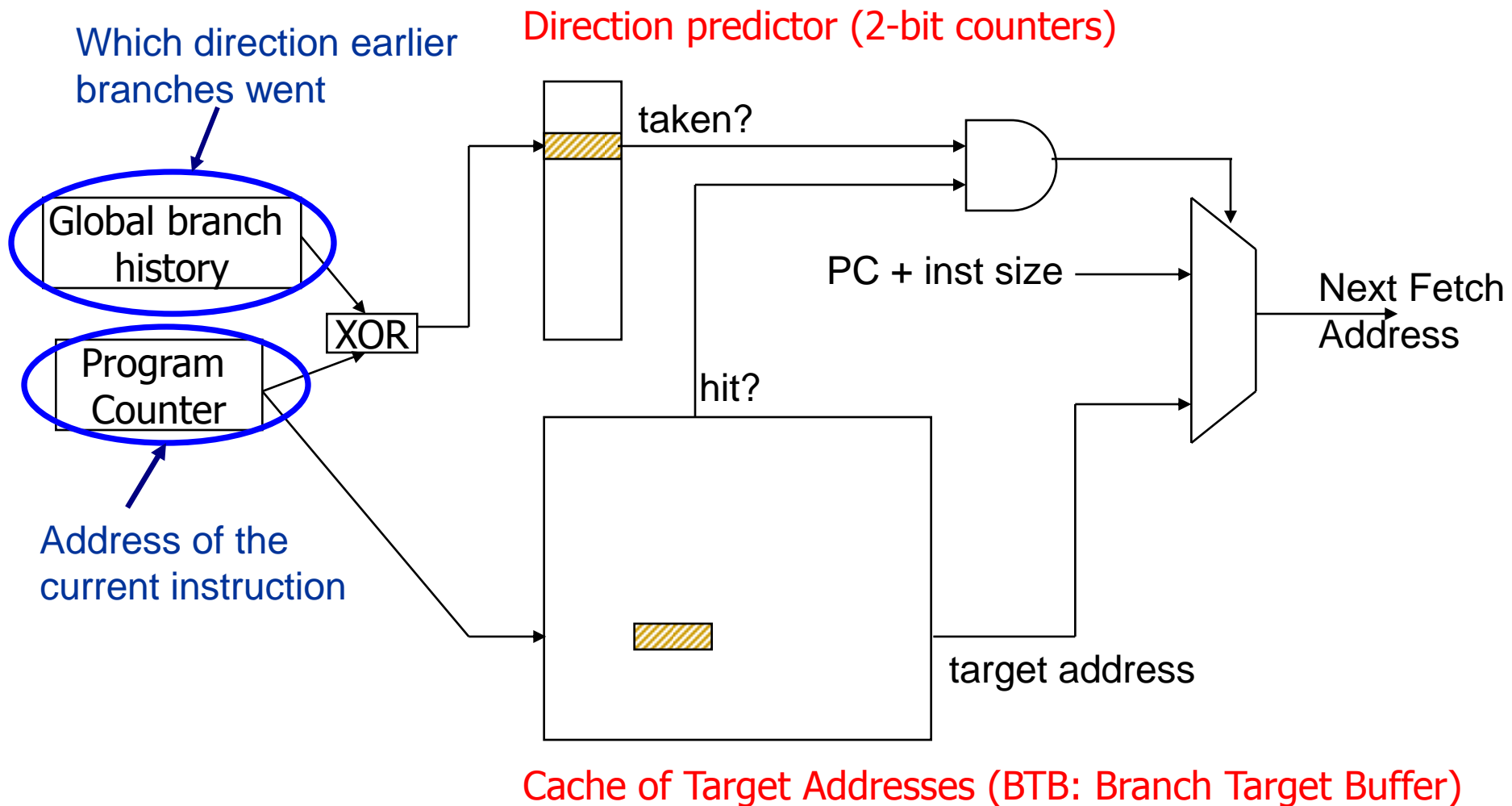
One-Level Branch Predictor



Two-Level Global History Predictor



Two-Level Gshare Predictor



Can We Do Better?

- Last-time and 2BC predictors exploit “last-time” predictability
- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
 - Global branch correlation
- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
 - Local branch correlation

Local Branch Correlation

```
for (i=1; i<=4; i++) { }
```

If the loop test is done at the end of the body, the corresponding branch will execute the pattern $(1110)^n$, where 1 and 0 represent taken and not taken respectively, and n is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

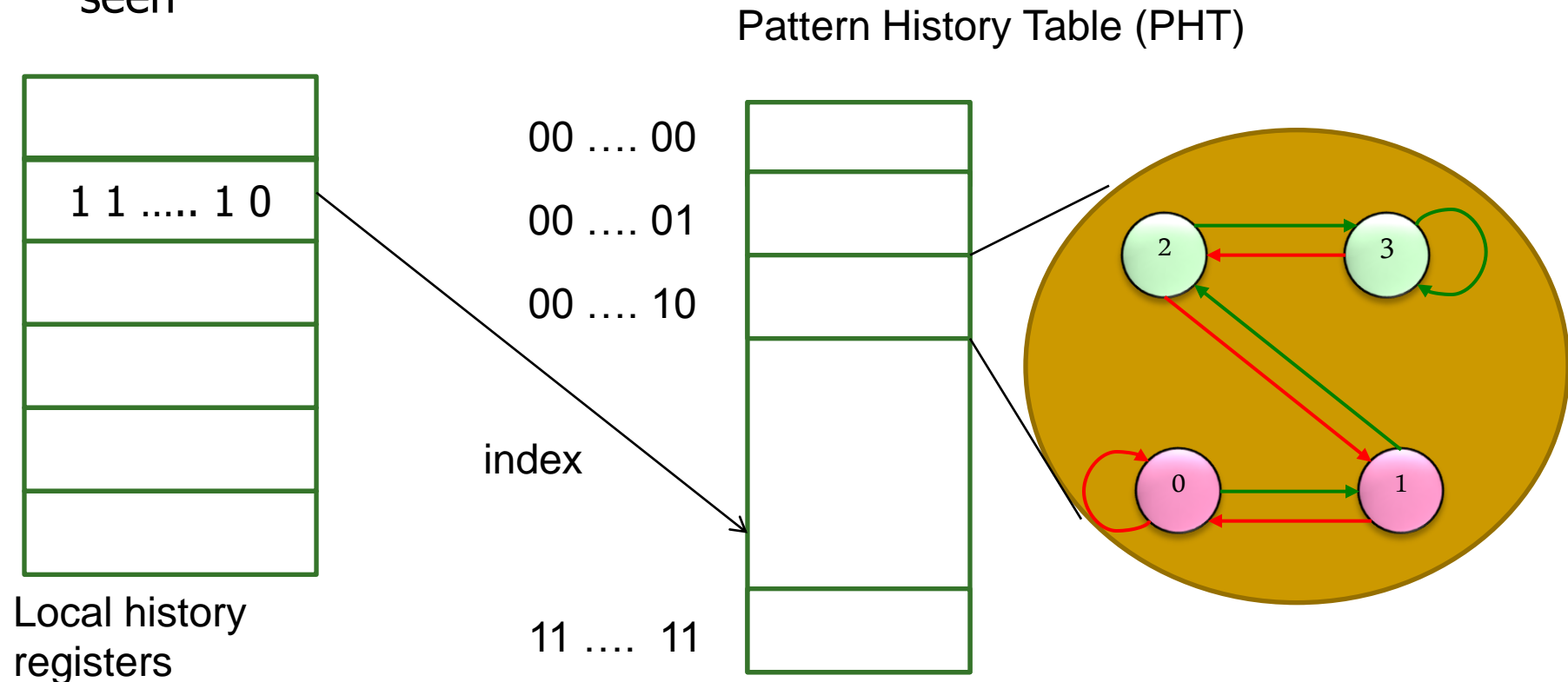
- McFarling, “Combining Branch Predictors,” DEC WRL TR 1993.

Capturing Local Branch Correlation

- Idea: Have a per-branch history register
 - Associate the predicted outcome of a branch with “T/NT history” of the same branch
- Make a prediction is based on the outcome of the branch the last time the same local branch history was encountered
- Called the local history/branch predictor
- Uses two levels of history (Per-branch history register + history at that history register value)

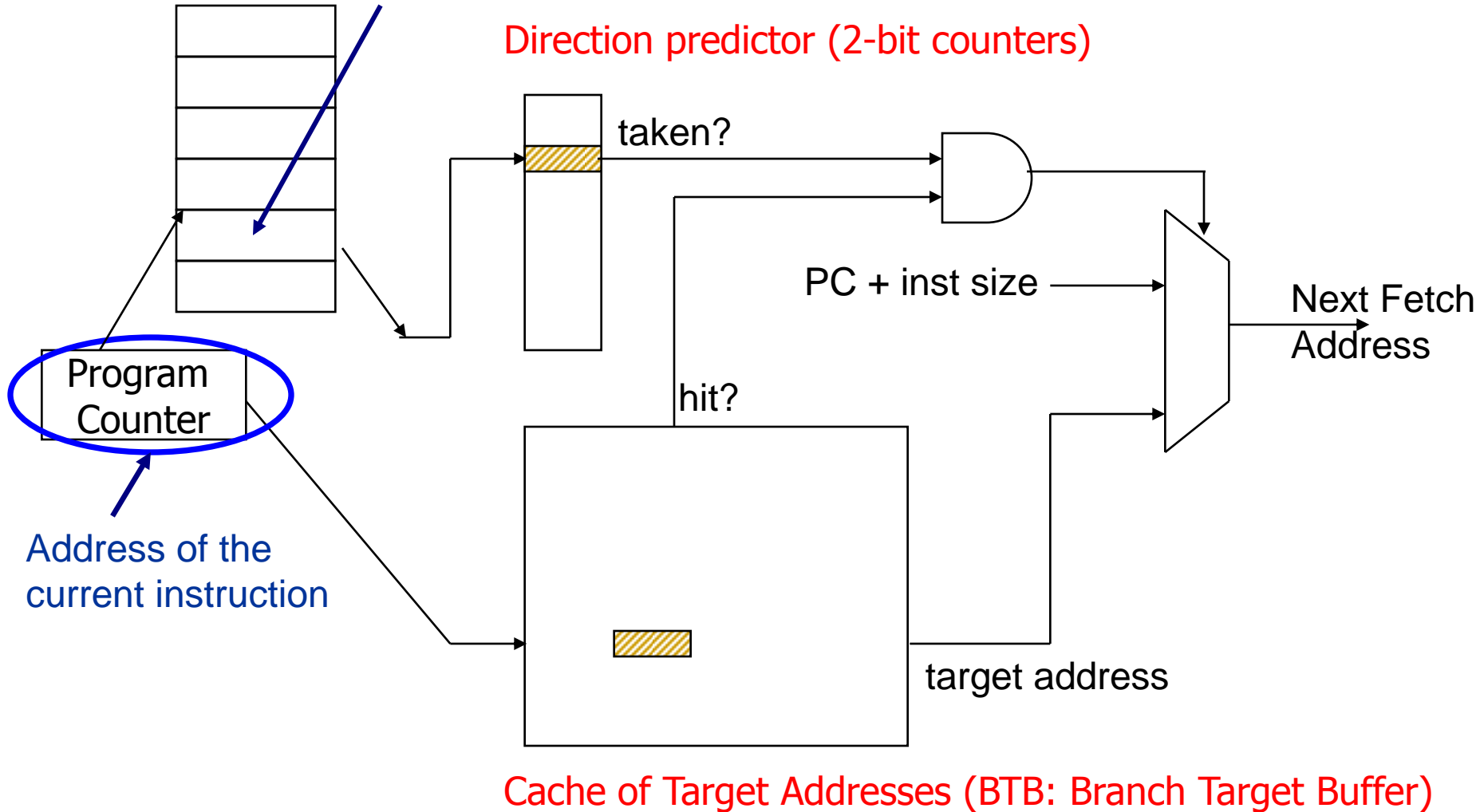
Two Level Local Branch Prediction

- First level: **A set of local history registers** (N bits each)
 - Select the history register based on the PC of the branch
- Second level: **Table of saturating counters for each history entry**
 - The direction the branch took the last time the same history was seen



Two-Level Local History Predictor

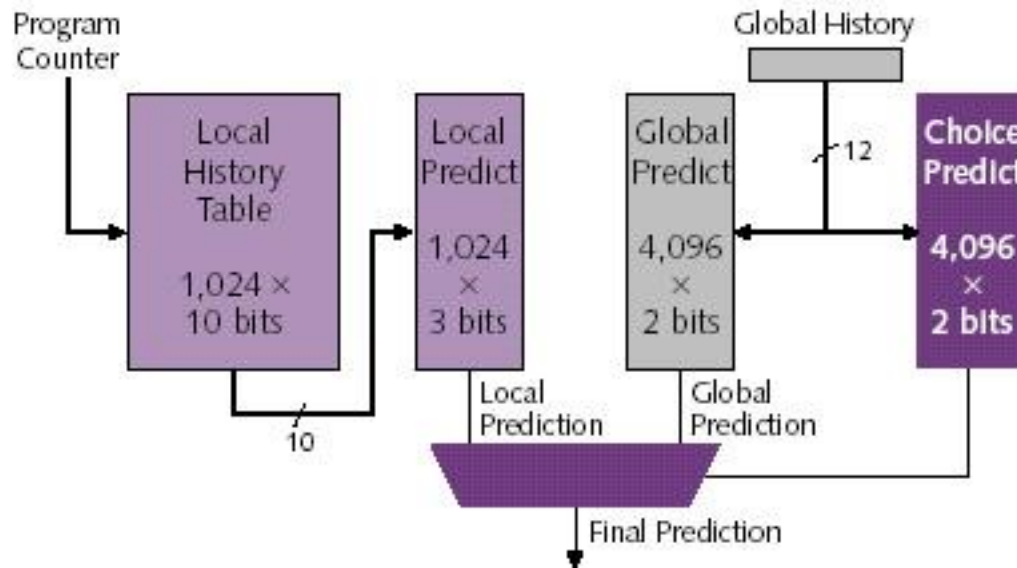
Which directions earlier instances of *this branch* went



Hybrid Branch Predictors

- Idea: Use more than one type of predictor (i.e., multiple algorithms) and select the “best” prediction
 - E.g., hybrid of 2-bit counters and global predictor
- Advantages:
 - + Better accuracy: different predictors are better for different branches
 - + Reduced **warmup** time (faster-warmup predictor used until the slower-warmup predictor warms up)
- Disadvantages:
 - Need “meta-predictor” or “selector”
 - Longer access latency
- McFarling, “**Combining Branch Predictors**,” DEC WRL Tech Report, 1993.

Alpha 21264 Tournament Predictor



- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch
- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Branch Prediction Accuracy (Example)

- Bimodal: table of 2bc indexed by branch address

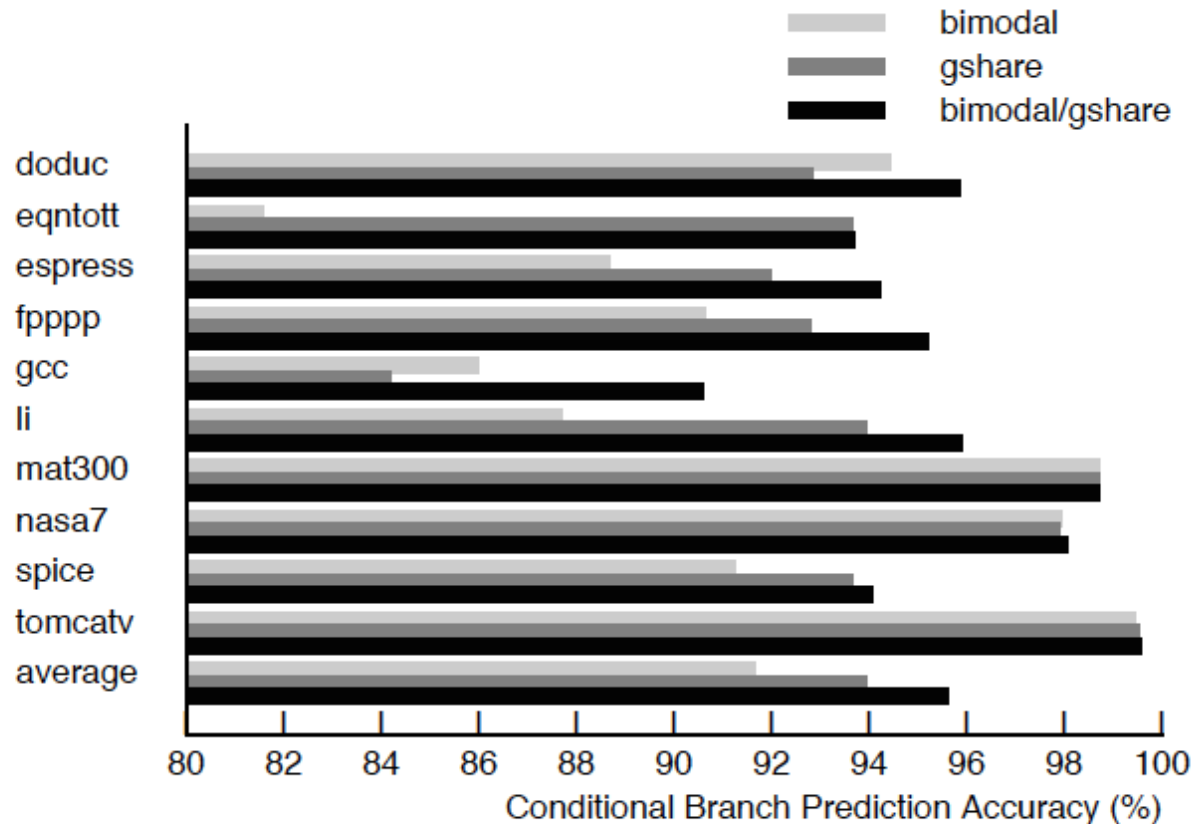


Figure 13: Combined Predictor Performance by Benchmark

Biased Branches

- Observation: Many branches are biased in one direction (e.g., 99% taken)
- Problem: These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing “interference” in branch prediction tables and history registers
- Solution: Detect such biased branches, and predict them with a simpler predictor
- Chang et al., “Branch classification: a new mechanism for improving branch predictor performance,” MICRO 1994.