

Course Description

Lecture 6: Control Hazards and Branch Prediction

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Credit: Brandon Lucia

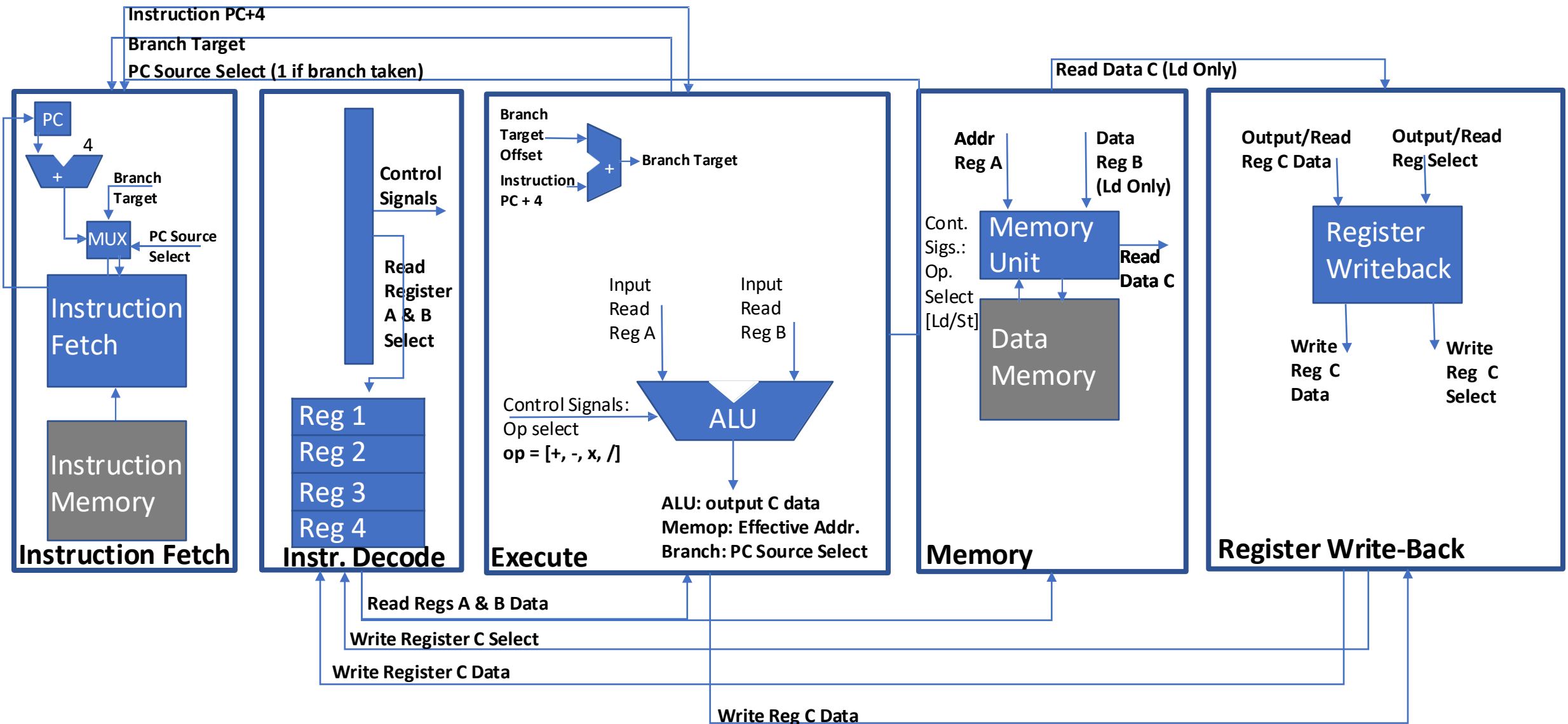
Some details...

- Prior feedback considered: Class has now moved to Piazza instead of Slack
- TAs have dedicated OH
- Instructions on labs will be covered more deeply in the recitations
- Lab hand-outs will be more clear --- but, please attend recitation and OH
- Lab 1 released today – due Sep 25th, 11:59:59 pm ET
- HW 2 released today – due Sep 23rd 11:59:59 pm ET

Recap: Pipelined Datapath Microarchitecture

- Understanding pipelining as a general microarchitectural optimization
- Data hazards and their effect on the pipelined microarchitectural datapath
- Forwarding as a mitigation for data hazards in a pipelined architecture

A Simple Pipelined Processor Datapath



Types of Data Hazards

```
sub  x6  x5  x4  
lw   x16 0xabc  
add  x12 x6  x14
```

Read-After-Write (RAW)

```
sub  x8  x16 x4  
add  x16 x6  x14  
lw   x16 0xabc
```

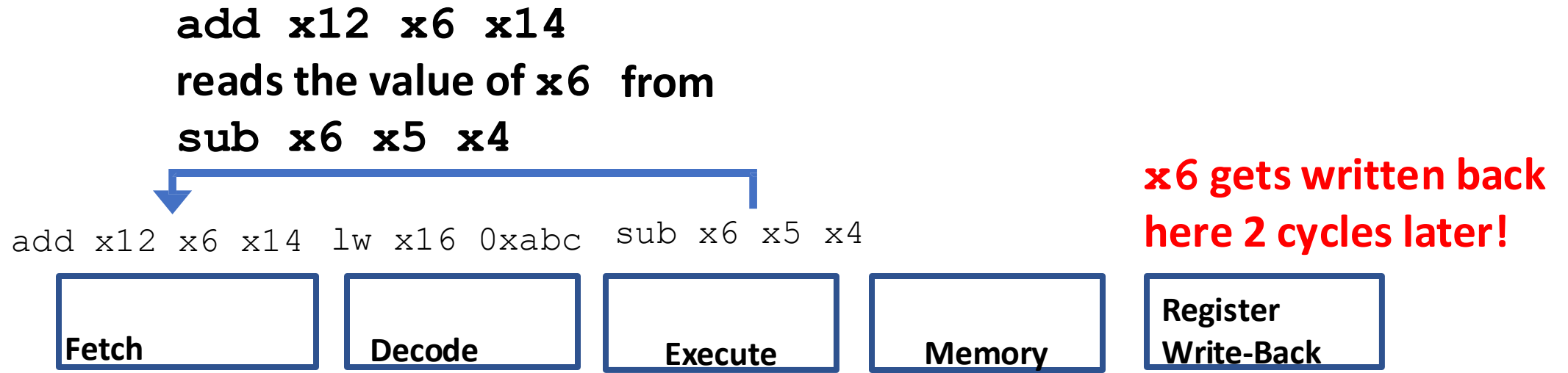
Write-After-Read (WAR)

```
lw   x6  0xabc  
sub  x6  x5  x4  
add  x12 x6  x14
```

Write-After-Write (WAW)

Only Read-After-Write (RAW) hazards are possible in our simple pipeline

Example: Pipelined Execution w/ RAW Hazard



Read-After-Write (RAW) Hazard:

Input register does not contain updated data during register read cycle due to yet-to-be-completed register writeback from older instruction

Example: Pipelined Execution w/ RAW Hazard

add x12 x6 x14 sub x6 x5 x4

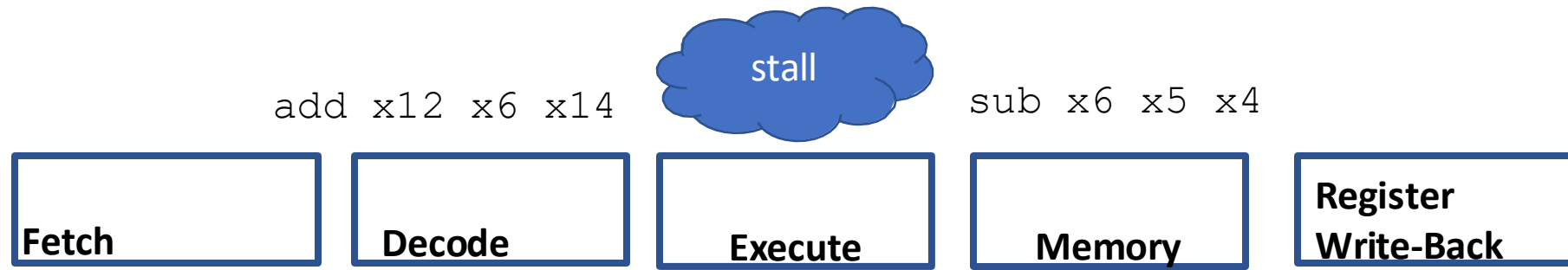


Example: Pipelined Execution w/ RAW Hazard

add x12 x6 x14 sub x6 x5 x4



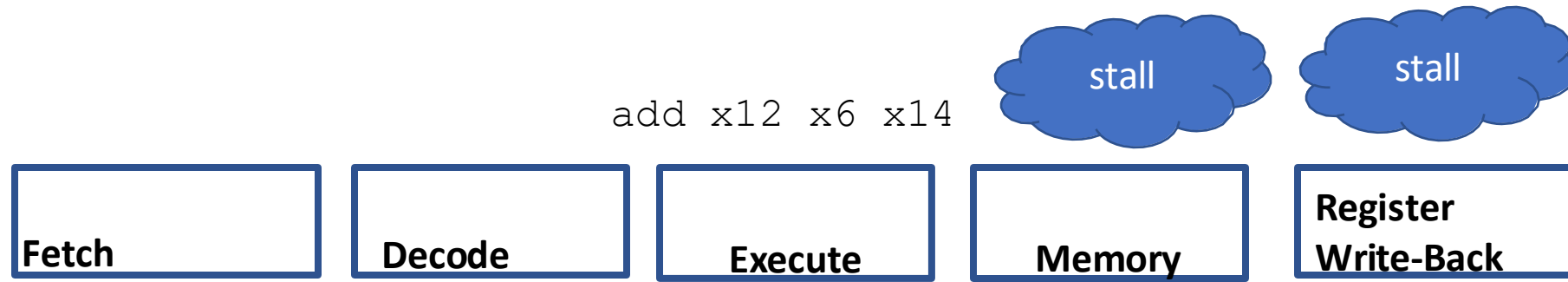
Example: Pipelined Execution w/ RAW Hazard



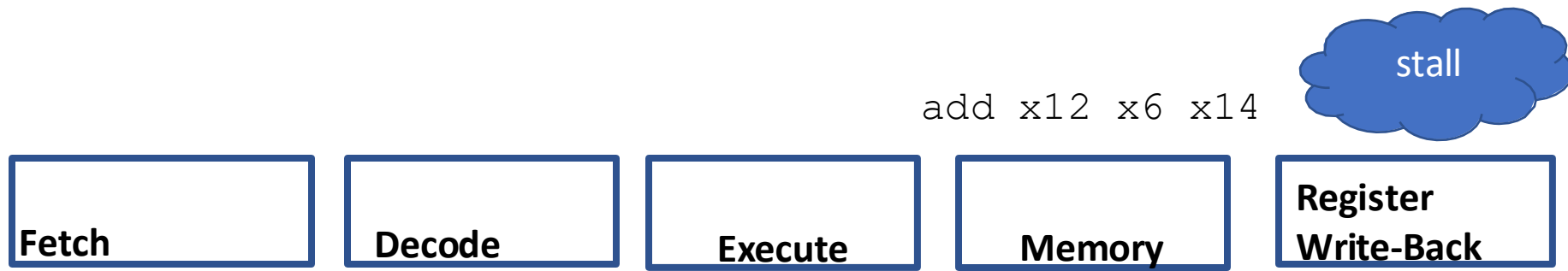
Example: Pipelined Execution w/ RAW Hazard



Example: Pipelined Execution w/ RAW Hazard



Example: Pipelined Execution w/ RAW Hazard



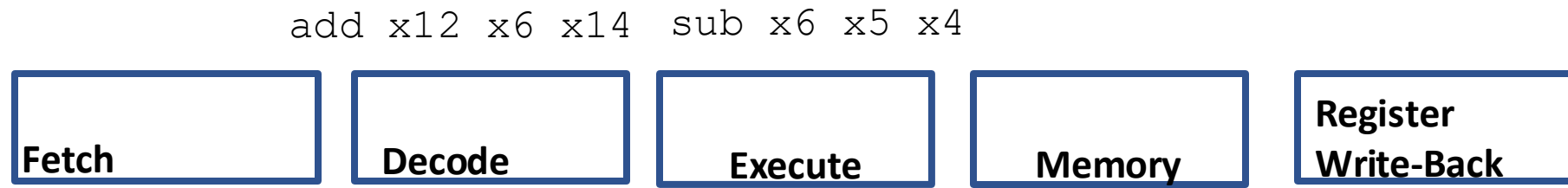
Example: Pipelined Execution w/ RAW Hazard



How do we avoid the stall cycles?



Example: Pipelined Execution w/ RAW Hazard

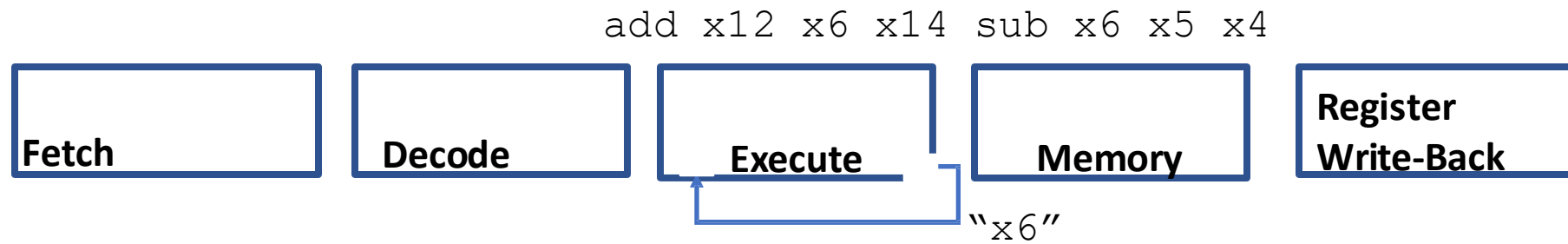


Value of x6 is available after `sub` Executes

We can forward the value to the add!

Forwarding to avoid a pipeline RAW Hazard

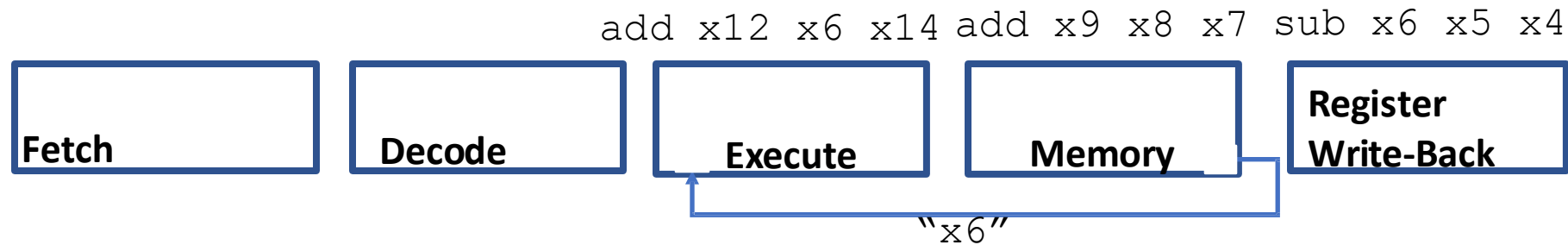
Value of x6 is available from EX/MEM pipeline register!



We can *forward* the value in the EX/MEM pipeline register from the sub back to Execute to act as the input operand for the add

Forwarding to avoid a pipeline RAW Hazard

Can also forward if there are intervening instructions

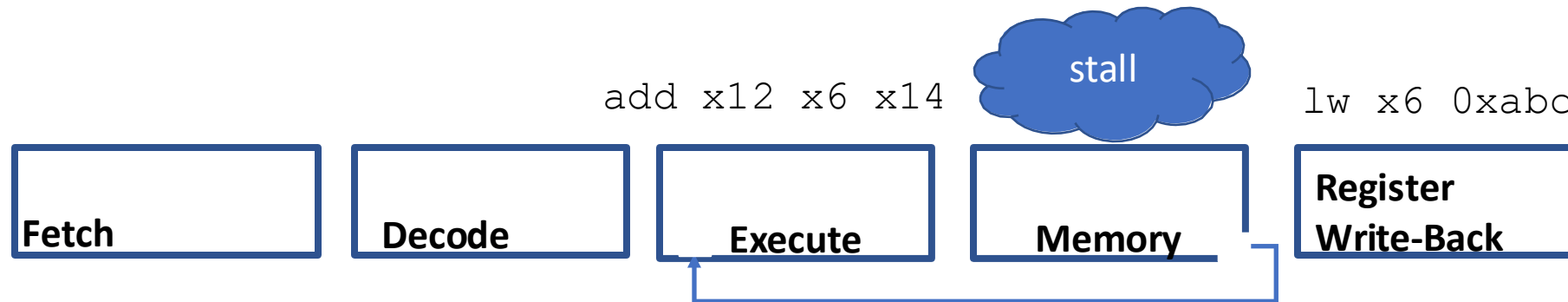


Can forward the value in the MEM/WB pipeline register from the sub back to Execute for the add (going around the unrelated operation now in the memory stage)

Immediately preceding & dependent on load = stall

```
lw x6 0xabc  
add x12 x6 x14
```

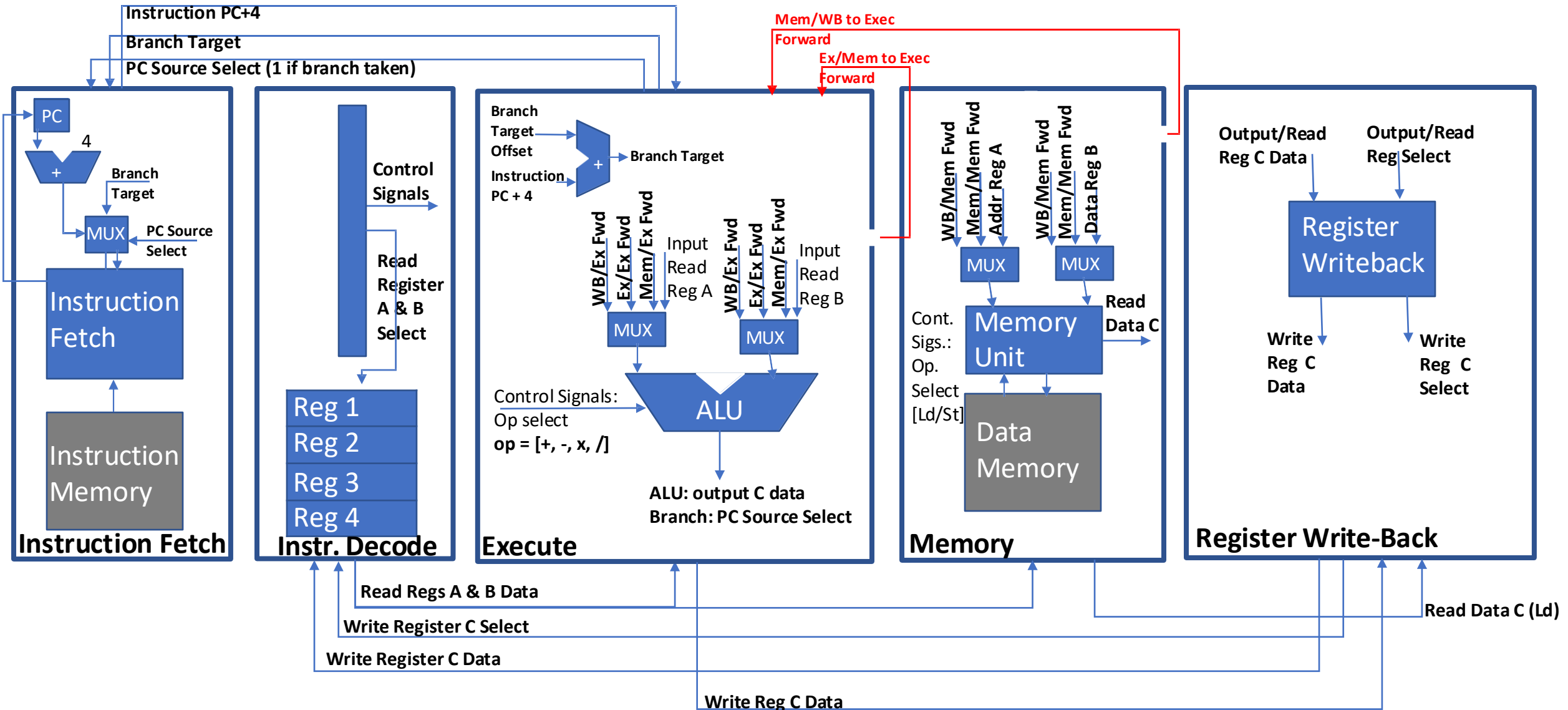
Value of x6 is available from Memory!



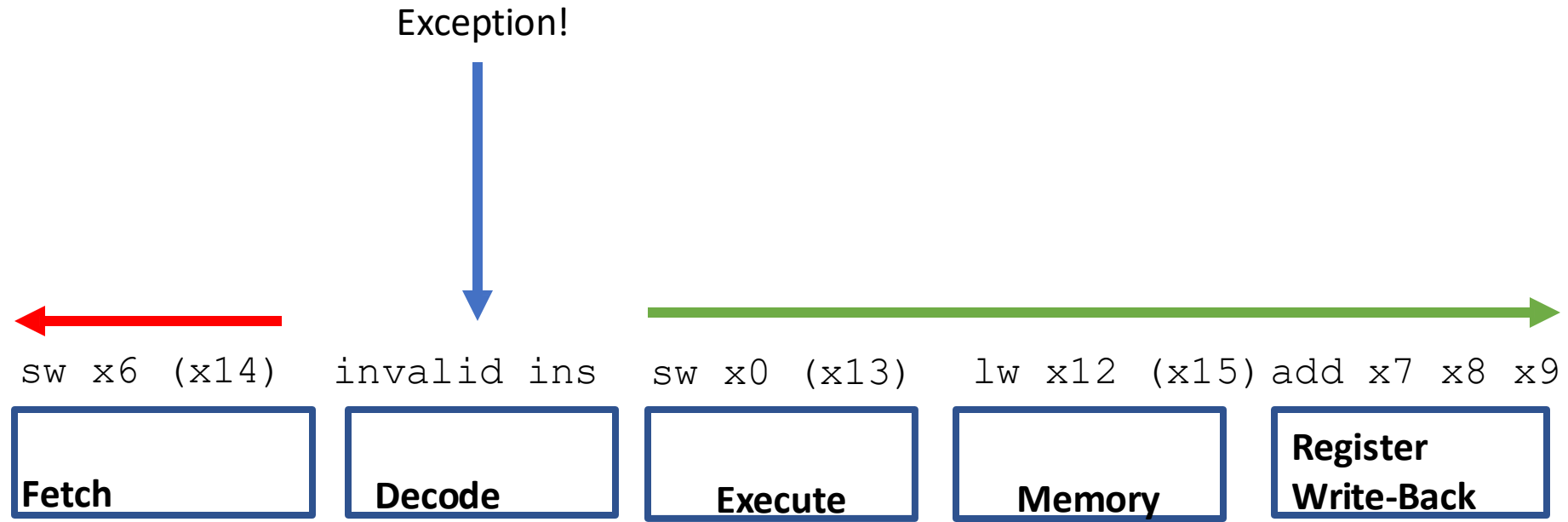
We can *forward* the value in Memory's pipeline register from the lw back to Execute's input for the add

(Still requires stalling...)

Adding Forwarding Support



Exception Handling



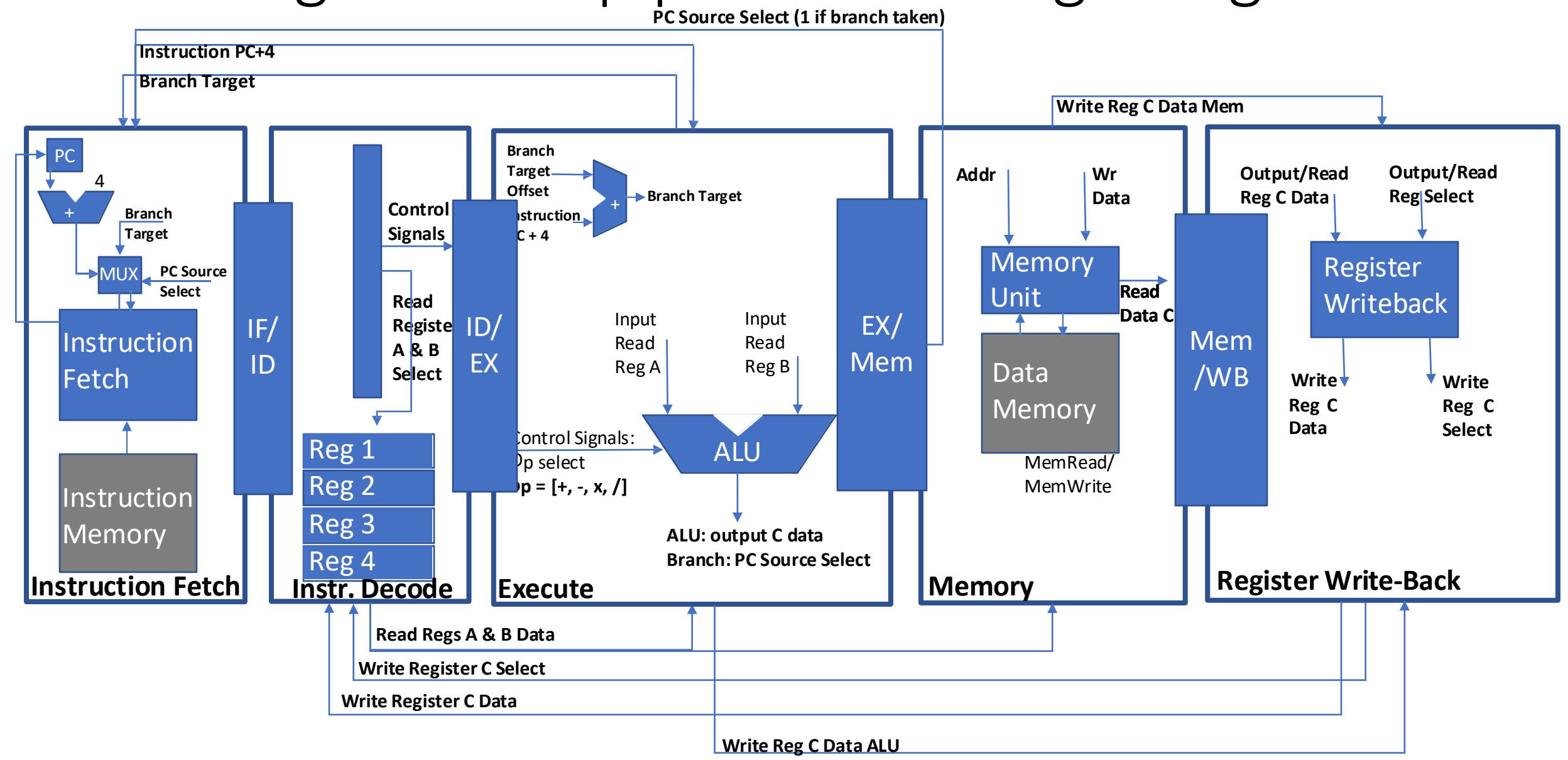
Basic Exception Idea: Nuke everything that started after the current instruction, finish everything that started before the current instruction, jump to exception handler

Today: More Pipelined Microarchitecture

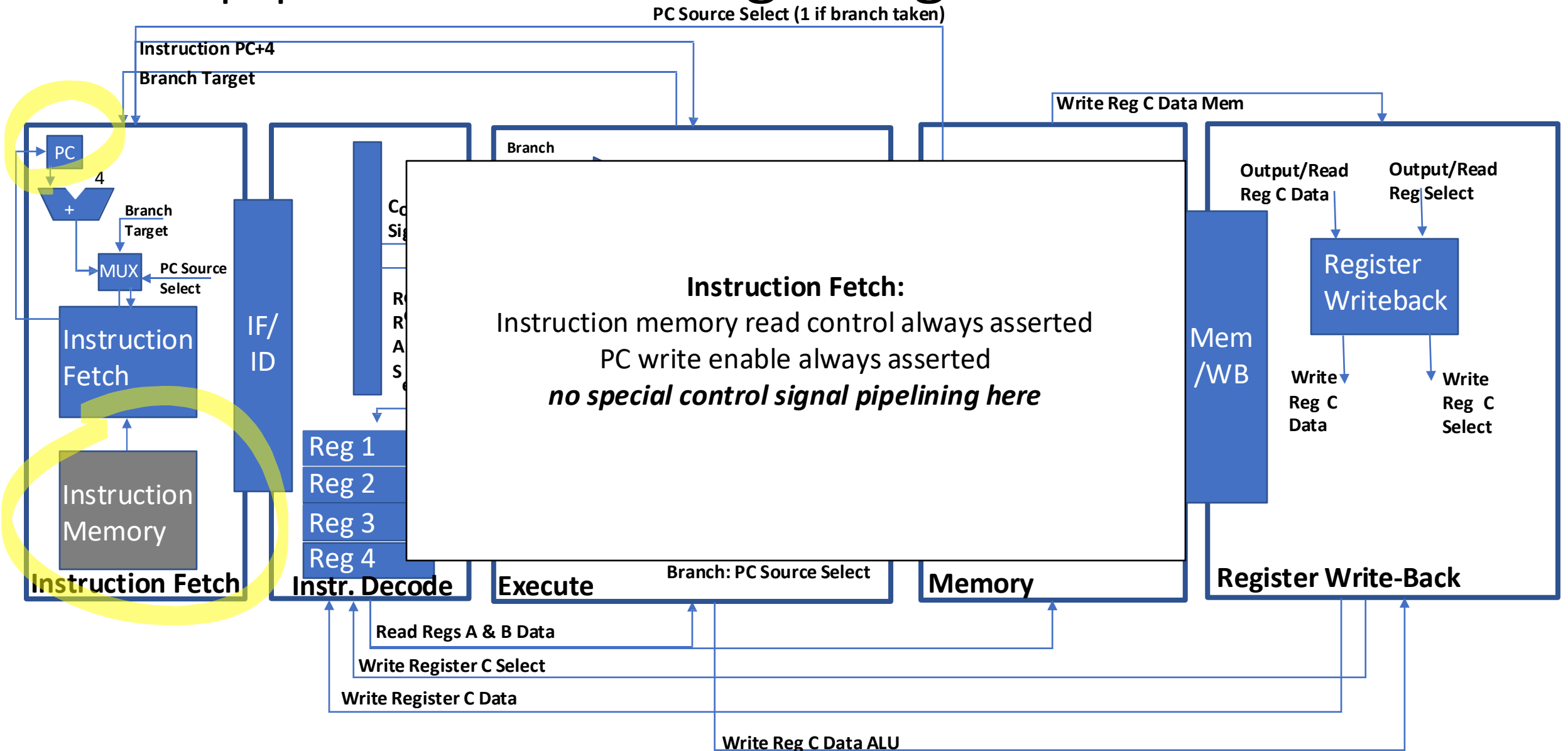
- Quick recap of pipeline mechanics
- Introduction to structural hazards
- Introduction to control hazards on branches
- Simple approaches to handling control hazards
- Branch prediction for handling control hazards

Pipeline Control Signals

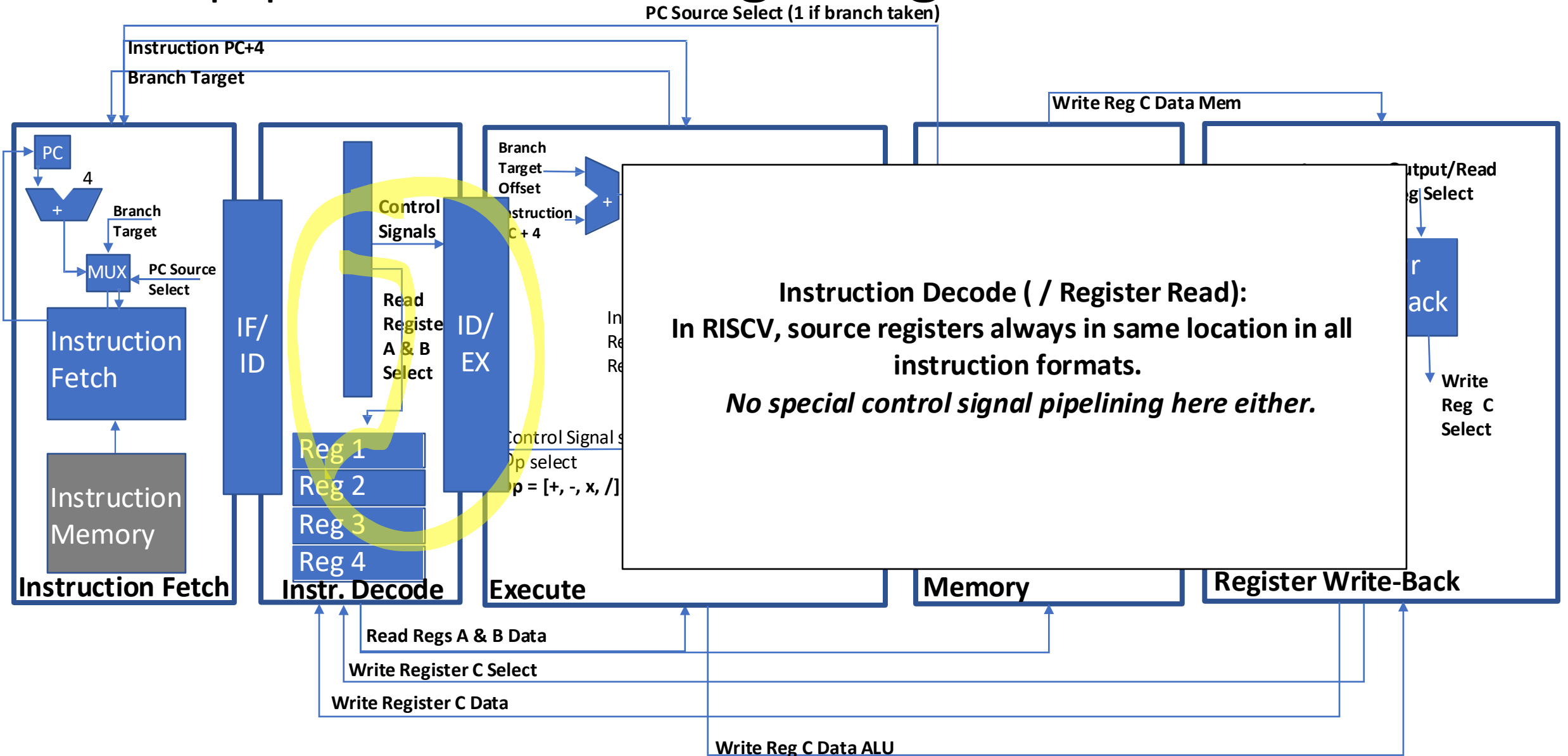
Control signals also pipelined through stages



Which pipeline control signals get set where?



Which pipeline control signals get set where?



Recall: R-type Arithmetic Operations

RV32I encoding

<https://metalcode.eu/2019-12-06-rv32i.html>

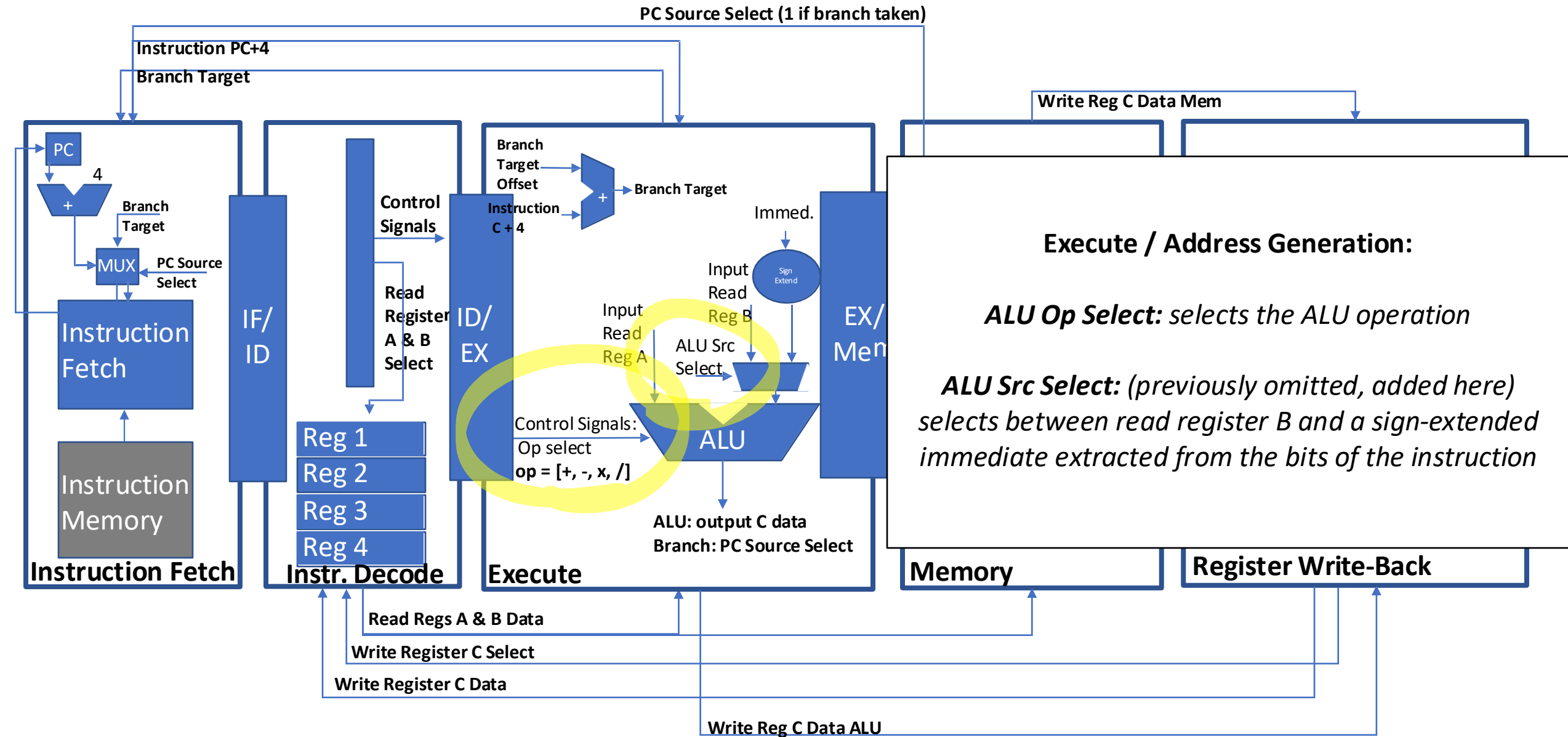
[31:25] 7	[24:20] 5	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
function 7	source 2	source 1	function 3	destination	opcode
0000000	xR	xL	000 : ADD	xD	0110011 : OP
0100000	xR	xL	000 : SUB	xD	0110011 : OP
0000000	xR	xL	001 : SLL	xD	0110011 : OP
0000000	xR	xL	010 : SLT	xD	0110011 : OP
0000000	xR	xL	011 : SLTU	xD	0110011 : OP
0000000	xR	xL	100 : XOR	xD	0110011 : OP
0000000	xR	xL	101 : SRL	xD	0110011 : OP
0100000	xR	xL	101 : SRA	xD	0110011 : OP
0000000	xR	xL	110 : OR	xD	0110011 : OP
0000000	xR	xL	111 : AND	xD	0110011 : OP

Func 7 = 32 reg x5 reg x6 SUB reg x7 OP

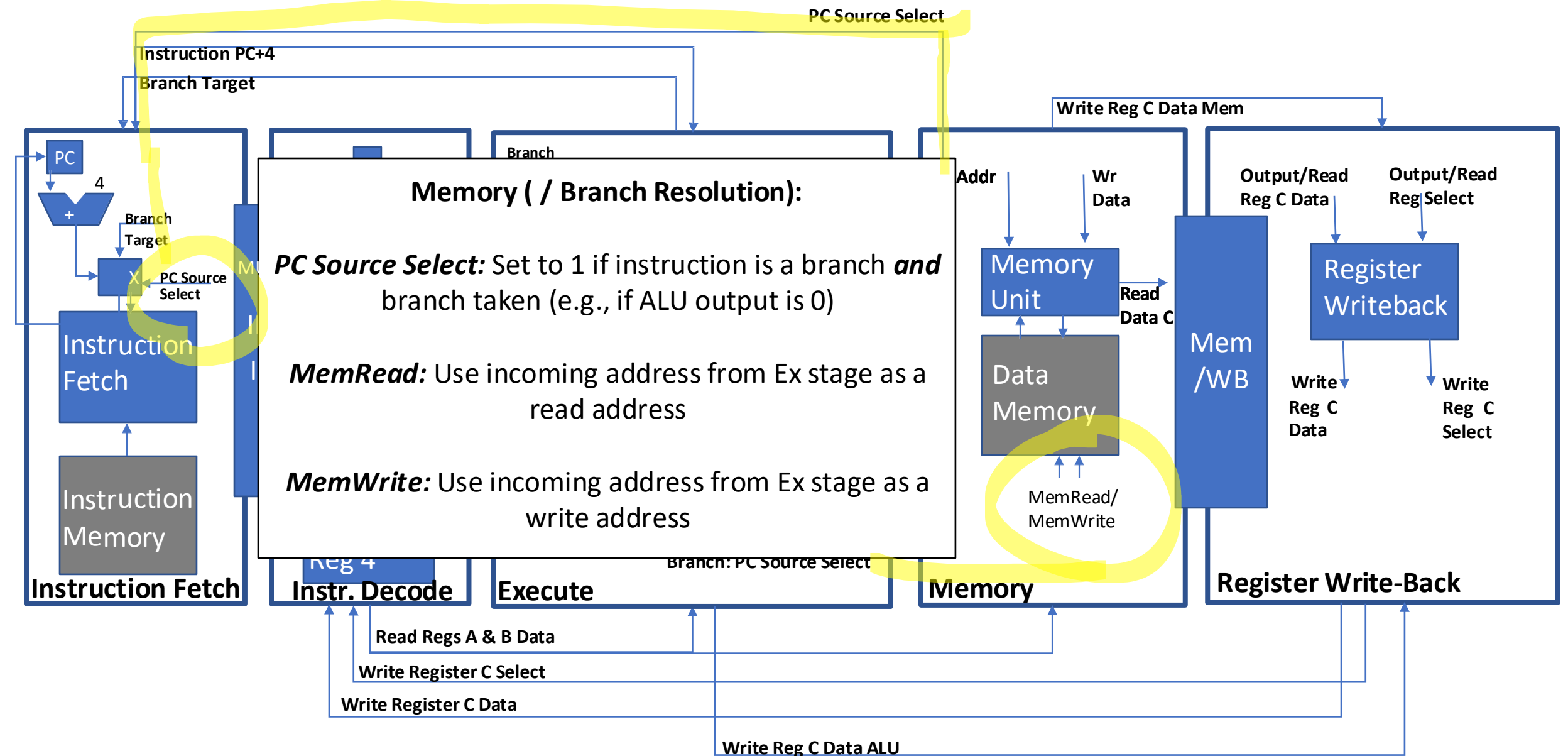
$$x7 = x5 - x6$$

0100000	00101	00110	000	00111	0110011
---------	-------	-------	-----	-------	---------

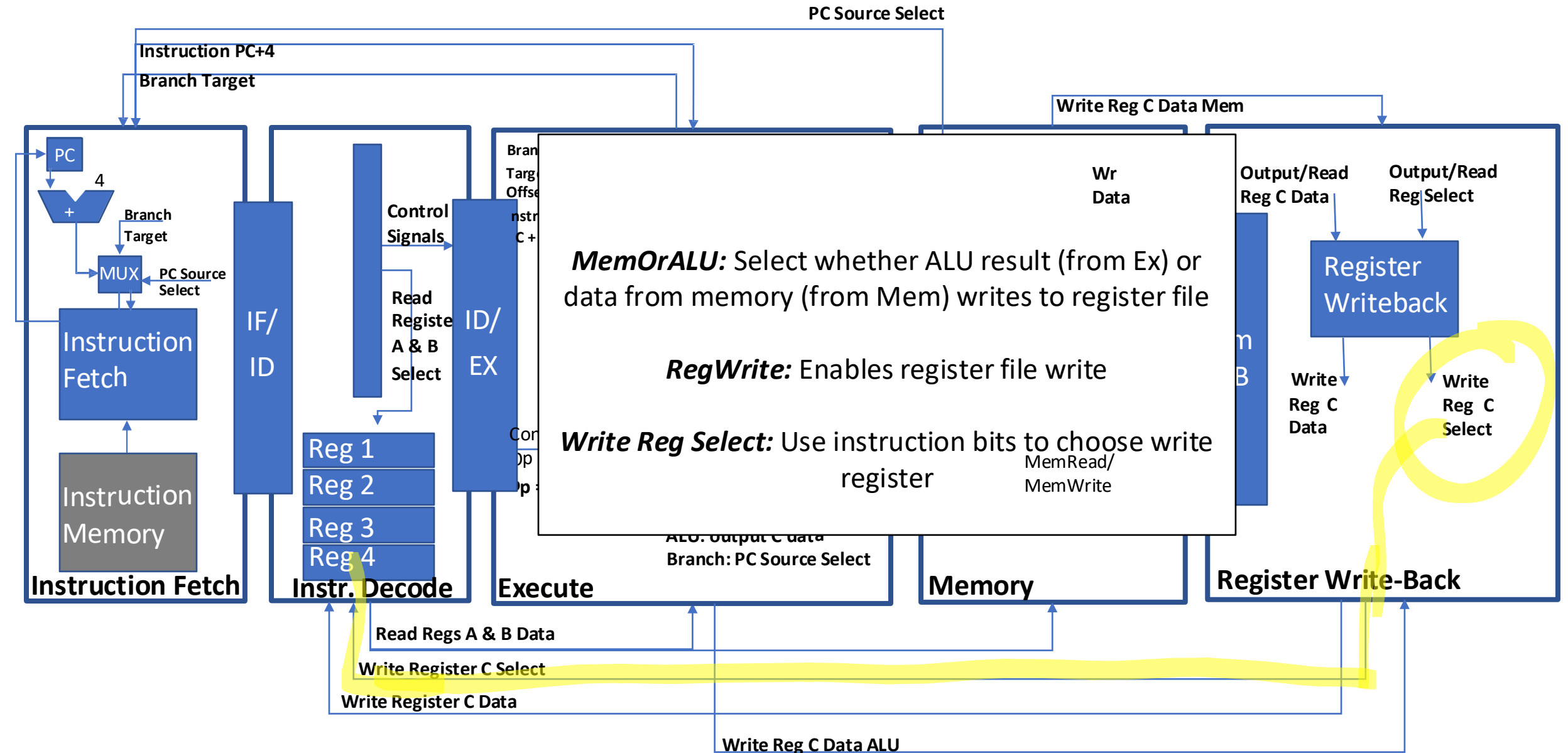
Which pipeline control signals get set where?



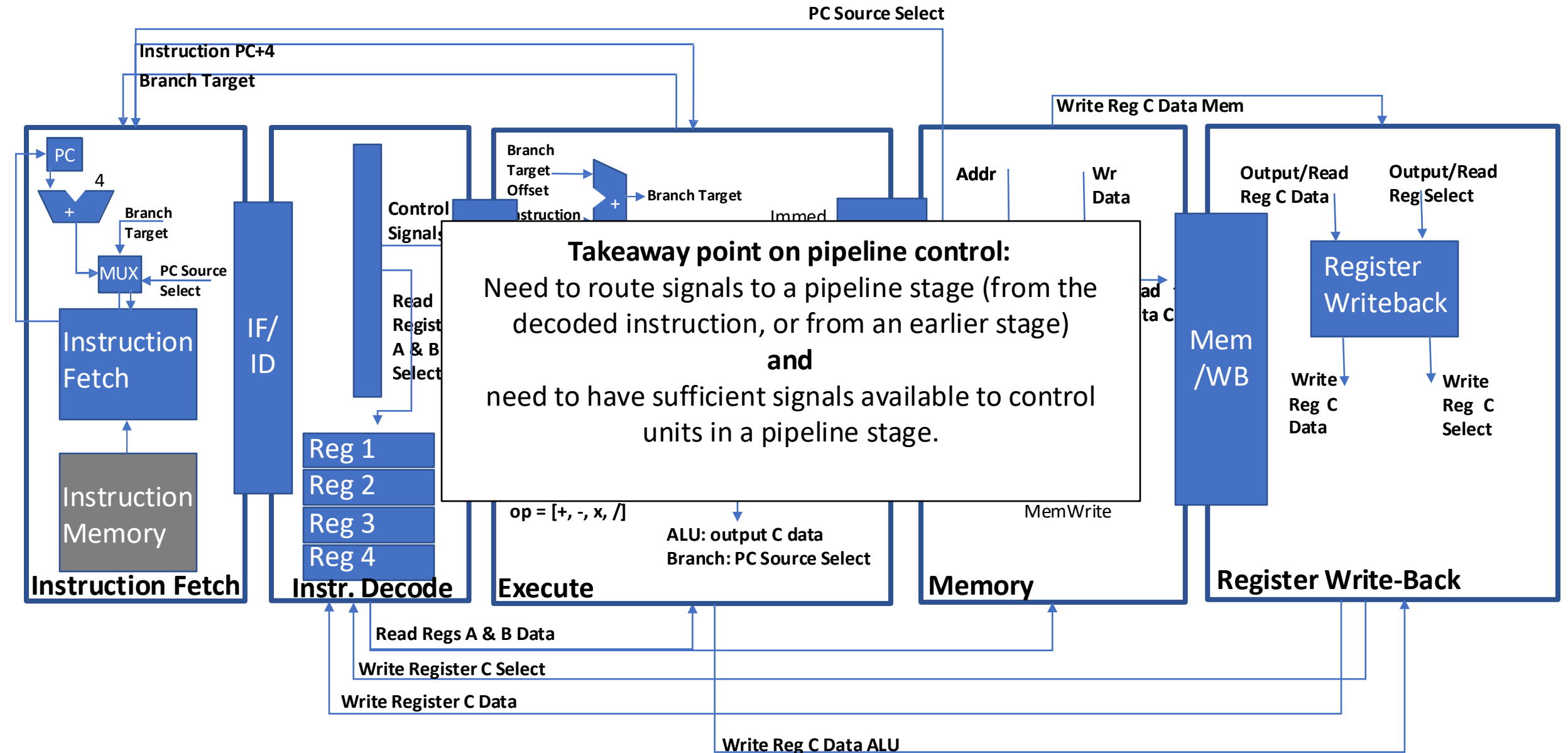
Which pipeline control signals get set where?



Which pipeline control signals get set where?



Which pipeline control signals get set where?

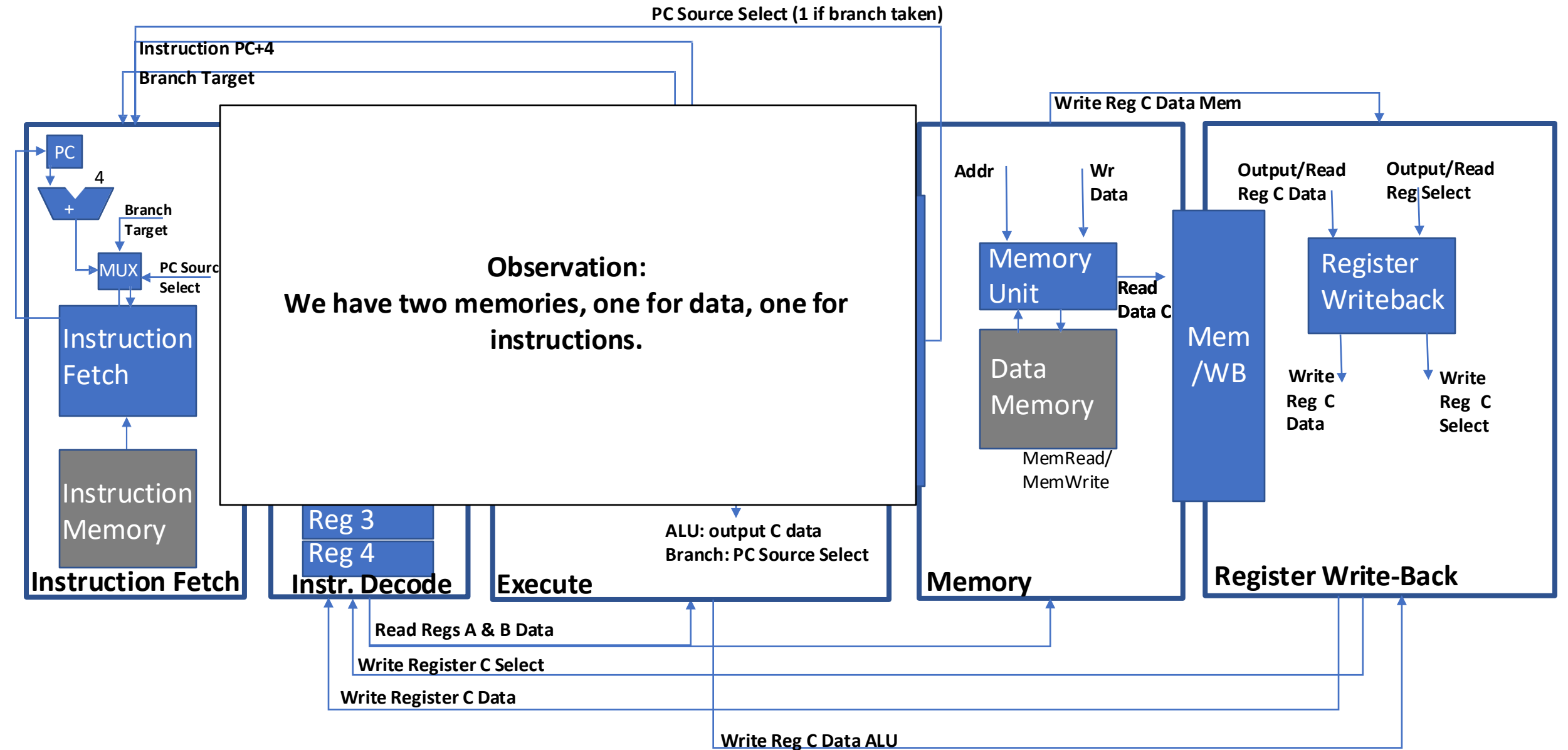


More on Pipeline Hazards

Structural Hazards

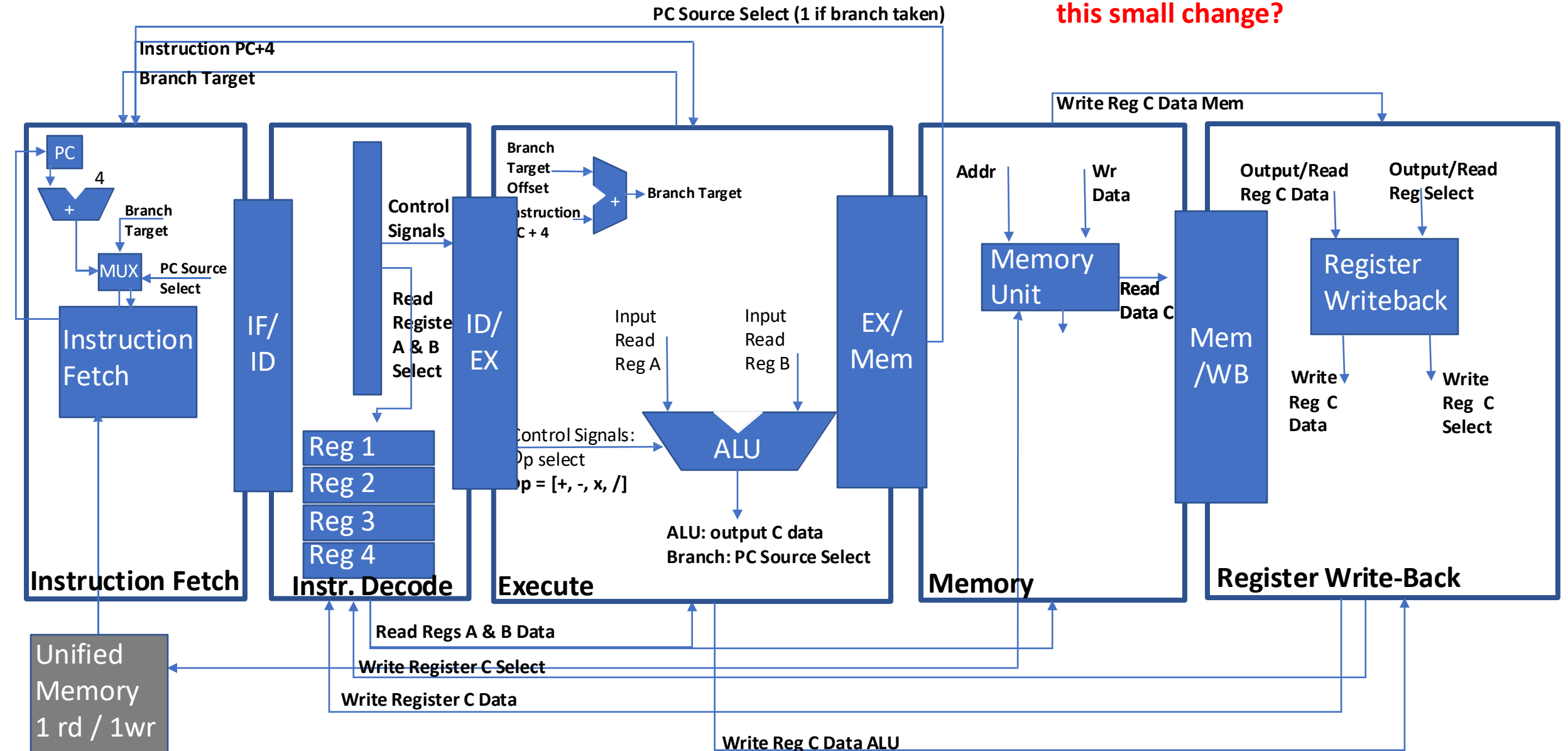
Control Hazards

Structural Hazards



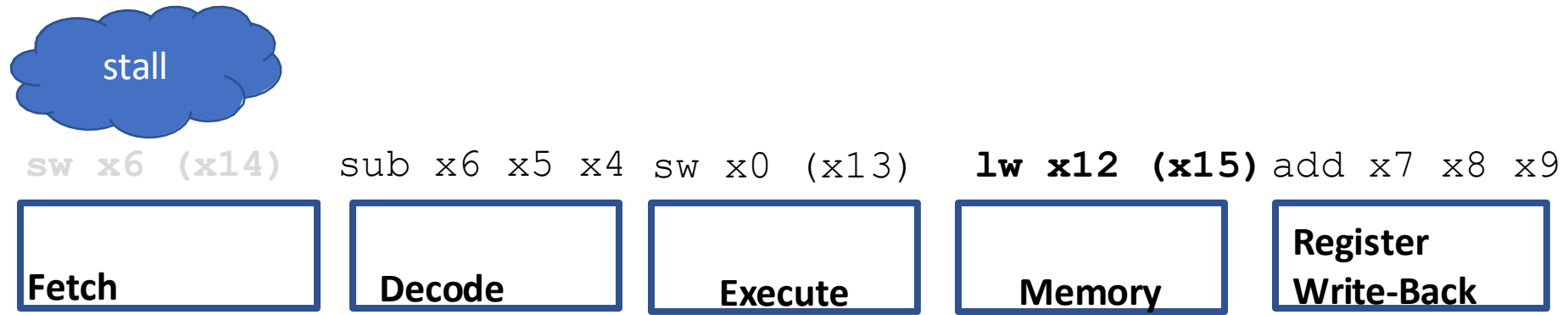
Structural Hazards

What happens when we make this small change?



Structural Hazards

No software or clever architectural mitigation. Need two memories or two memory ports.



Fetch is blocked with
no access to read port

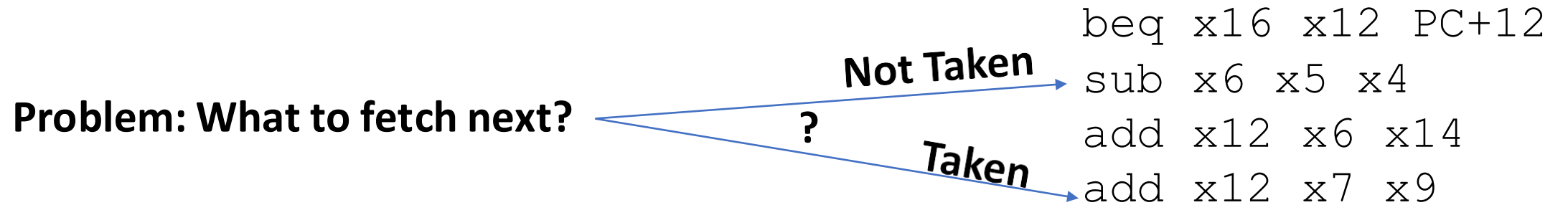
Load occupies unified
memory read port

Control Hazards

```
beq  x16  x12  PC+12
sub  x6   x5   x4
add  x12  x6   x14
add  x12  x7   x9
```

Branch-induced Control Hazard

Example: Pipelined Execution w/ Branch



beq x16 x12 PC+12

Fetch

Decode

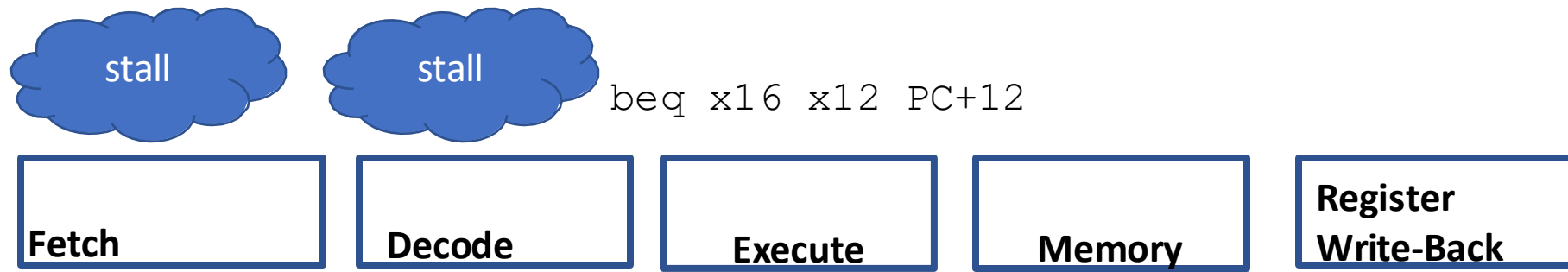
Execute

Memory

**Register
Write-Back**

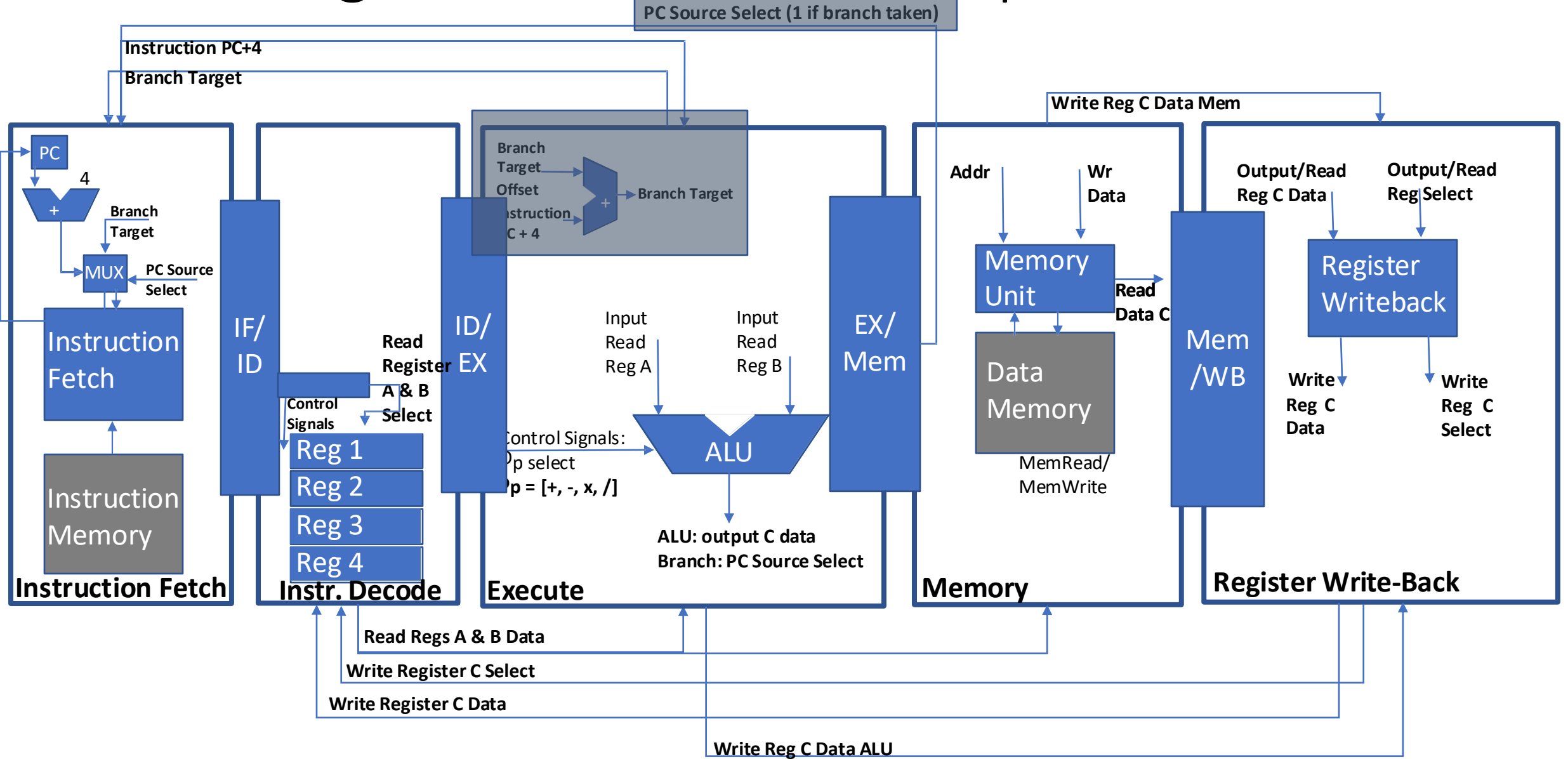
Example: Pipelined Execution w/ Branch

Option #1: Stall on Branch



**Proposal: We know the next PC only after beq finishes Ex
(What signals do we need to determine next PC?)**

Determining the Next PC in the Pipeline



Example: Pipelined Execution w/ Branch

Option #1: Stall on Branch



2 pipeline bubbles per branch!

Example: Pipelined Execution w/ Branch

Option #1: Stall on Branch

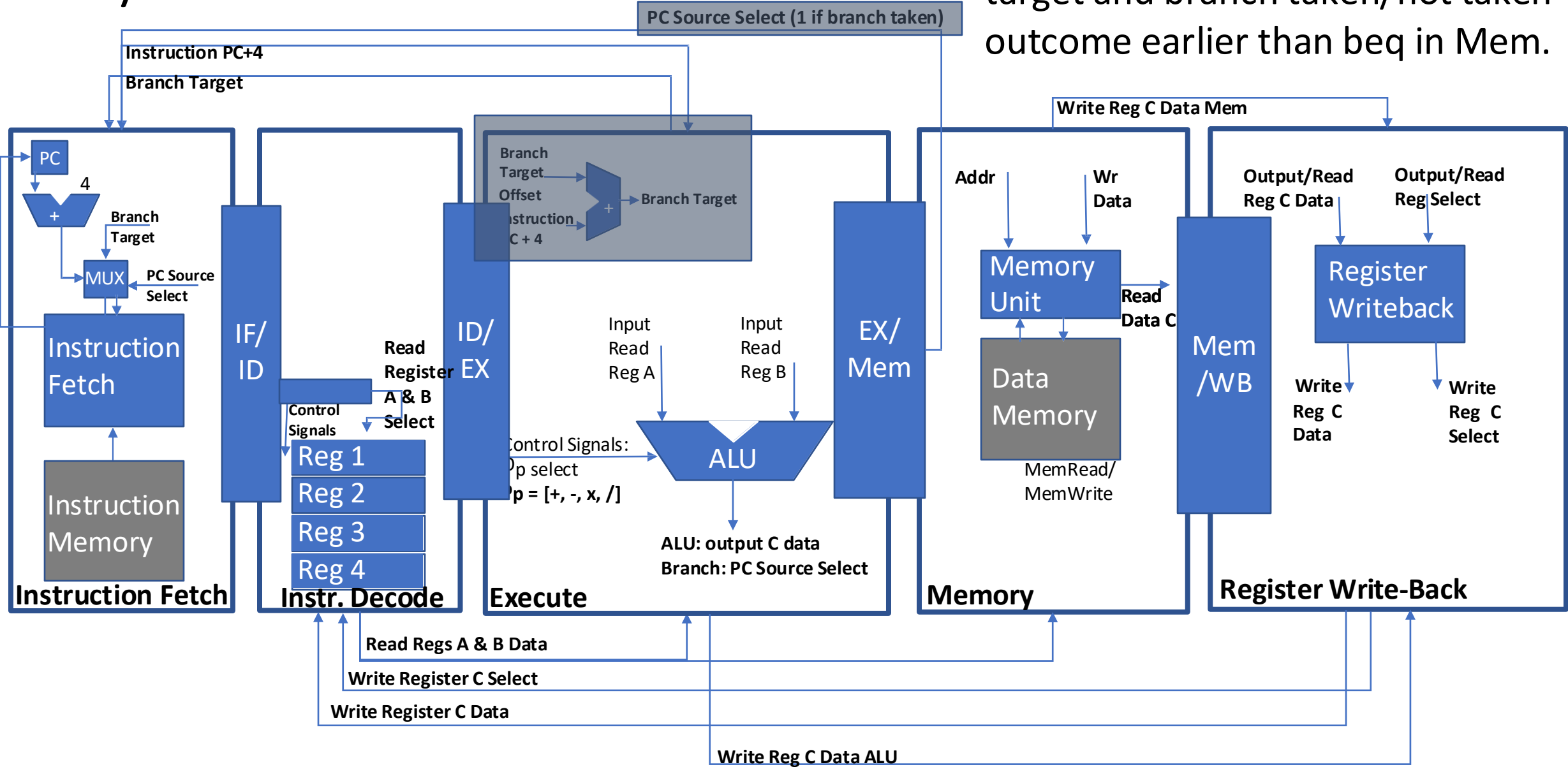


2 pipeline bubbles per branch!

Can we do better than 2 bubbles?

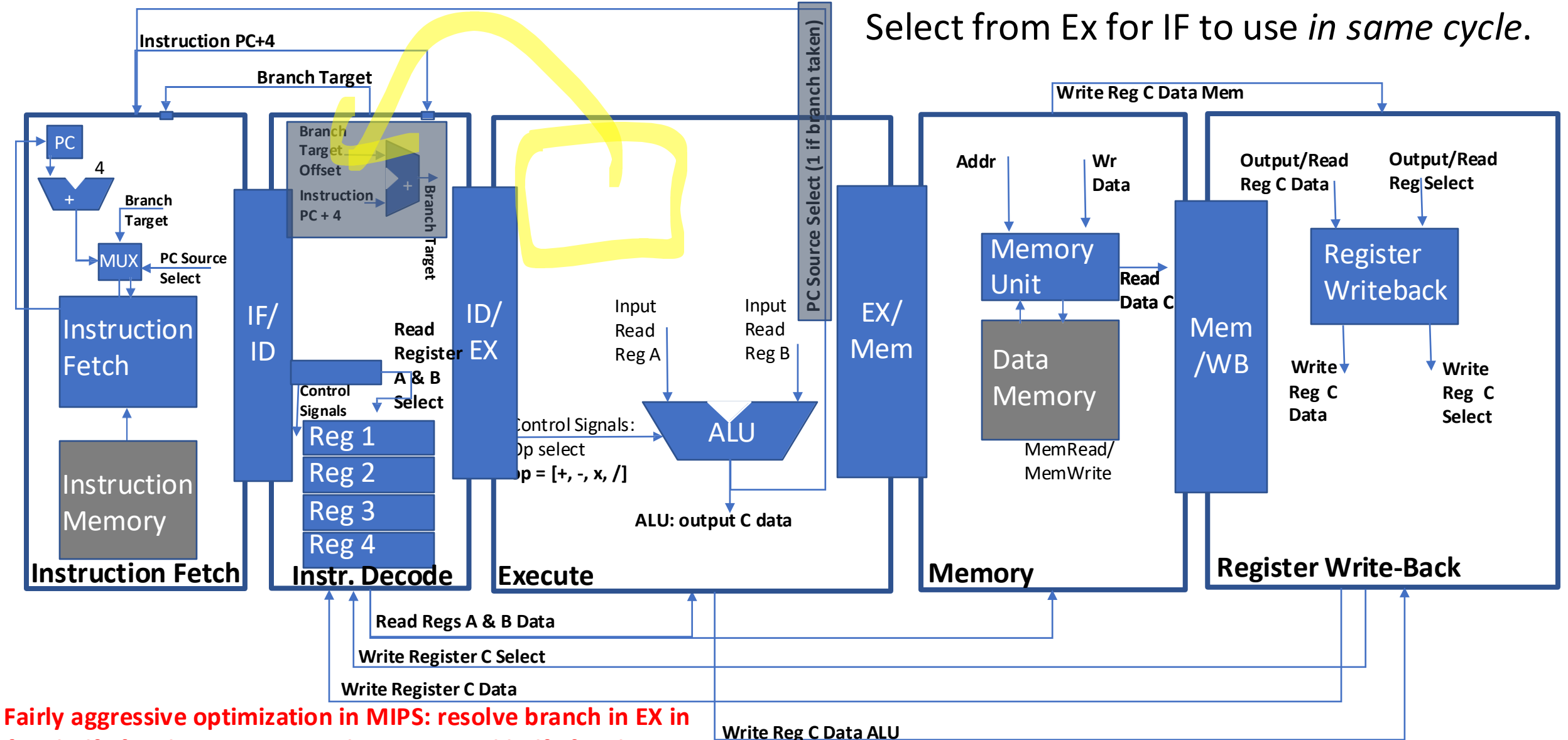
Early Branch Resolution

Proposal: Resolve the branch target and branch taken/not taken outcome earlier than beq in Mem.



Early Branch Resolution

Key Idea: Move Branch Target logic to ID and add forwarding path for PC Source Select from Ex for IF to use *in same cycle*.



Fairly aggressive optimization in MIPS: resolve branch in EX in first half of cycle, use PC Src Select in second half of cycle in IF to fetch the correct instruction. Could limit clock frequency...

Example: Pipelined Execution w/ Early Resolution

Problem: What to fetch next?

Not Taken

?

Taken

beq x16 x12 PC+12

sub x6 x5 x4

add x12 x6 x14

add x12 x7 x9

beq x16 x12 PC+12

Fetch

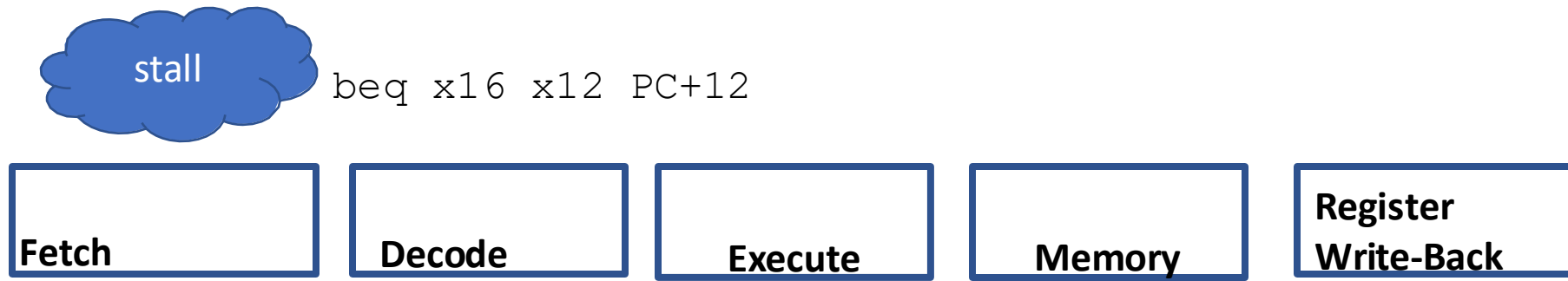
Decode

Execute

Memory

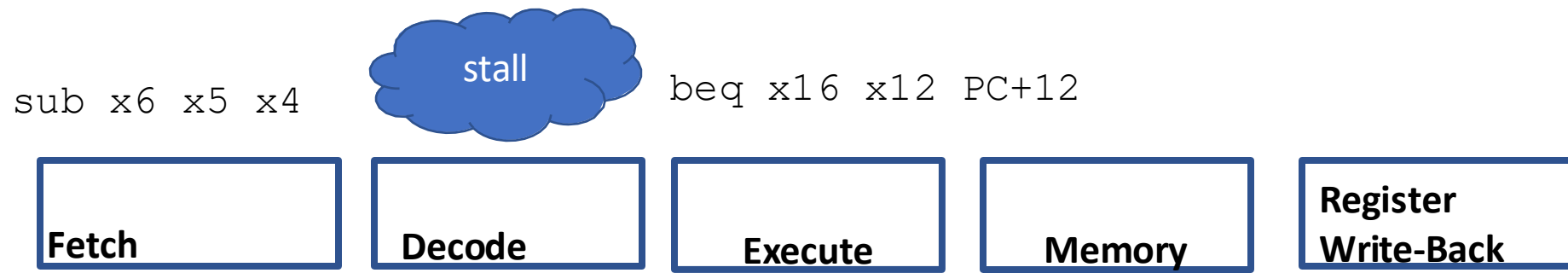
**Register
Write-Back**

Example: Pipelined Execution w/ Early Resolution



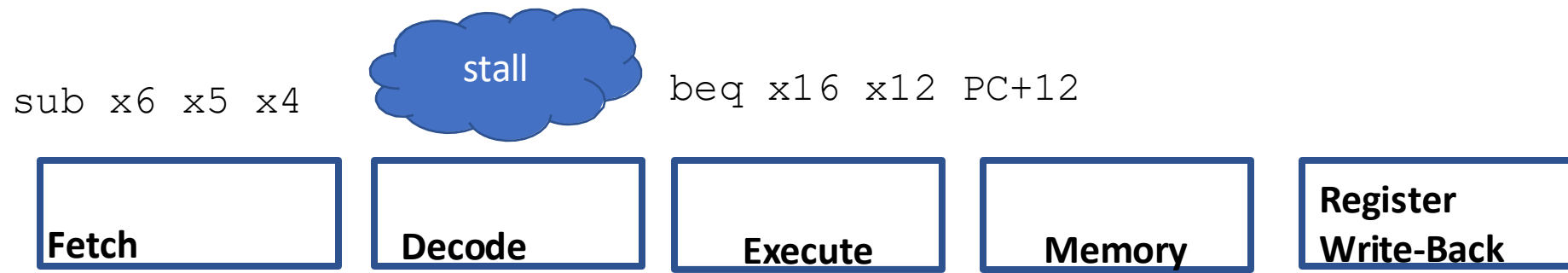
We know the next PC when beq is in Ex

Example: Pipelined Execution w/ Early Resolution



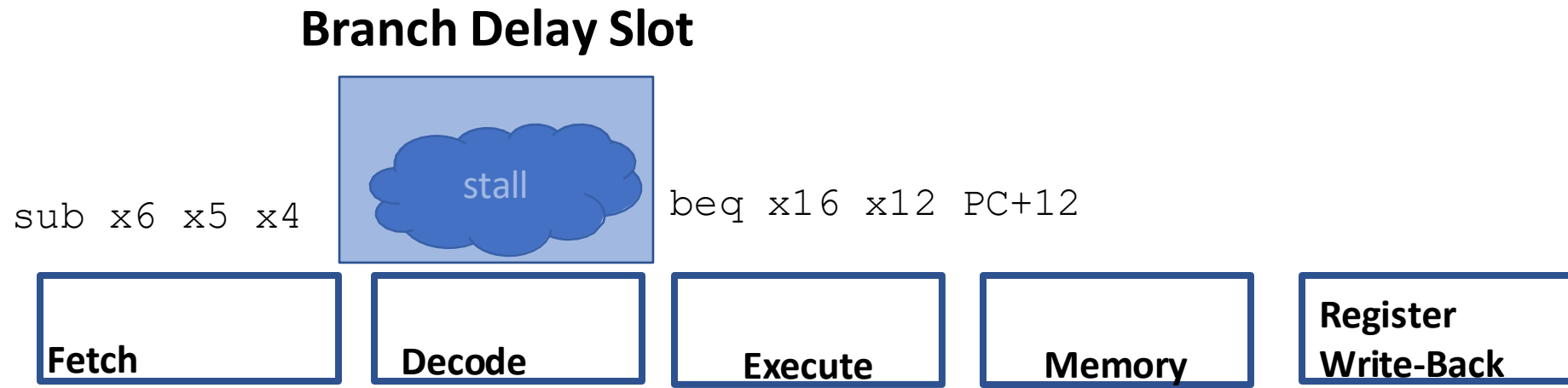
1 pipeline bubble per branch!

Example: Pipelined Execution w/ Early Resolution



Can we do even better than 1 stall per branch?

MIPS-style Delayed Branch Execution



Branch takes an extra cycle to resolve, so how about just fetching *some* instruction in the “delay slot”?

Filling Branch Delay Slots

Branch Delay Slot
filled w/ not-taken
instruction

```
beq x16 x12 PC+12
sub x6 x5 x4
add x12 x6 x14
add x12 x7 x9
```

```
sub x6 x5 x4    beq x16 x12 PC+12
```



Only useful if branch ends up resolving not-taken

Otherwise, need to erase the effects of the sub when branch finally resolves

Filling Branch Delay Slots

Branch Delay Slot
filled w/ taken
instruction

```
beq x16 x12 PC+12
sub x6 x5 x4
add x12 x6 x14
add x12 x7 x9
```

```
add x12 x7 x9  beq x16 x12 PC+12
```



Only useful if branch ends up resolving taken

Otherwise, need to erase the effects of the add when branch finally resolves

Filling Branch Delay Slots

Branch Delay Slot
filled w/ other
independent insn

```
mul x8 x15 x1      beq x16 x12 PC+12
beq x16 x12 PC+12  mul x8 x15 x1
sub x6 x5 x4       sub x6 x5 x4
add x12 x6 x14     add x12 x6 x14
add x12 x7 x9      add x12 x7 x9
```

Compiler reorders code to fill slot

```
mul x8 x15 x1      beq x16 x12 PC+12
```



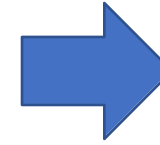
Why is it OK for the compiler to reorder the mul and the beq in this instance?

Filling Branch Delay Slots

Branch Delay Slot
filled w/ other
independent insn

```
mul x8 x15 x1
beq x16 x12 PC+12
sub x6 x5 x4
add x12 x6 x14
add x12 x7 x9
```

Compiler reorders code to fill slot



```
beq x16 x12 PC+12
mul x8 x15 x1
sub x6 x5 x4
add x12 x6 x14
add x12 x7 x9
```

```
mul x8 x15 x1  beq x16 x12 PC+12
```



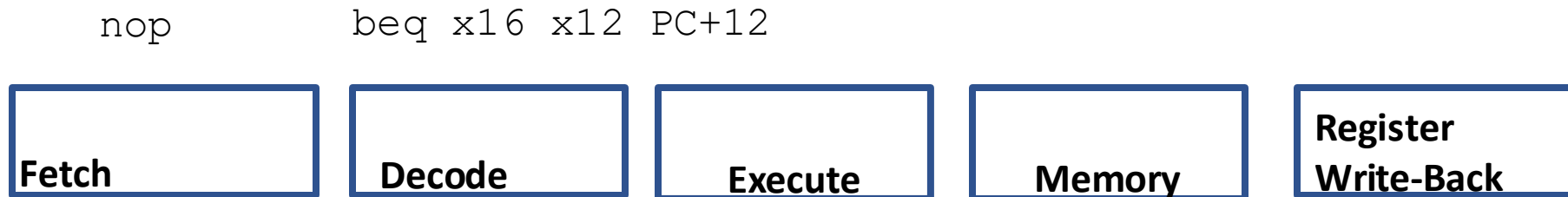
Why is it OK for the compiler to reorder the mul and the beq in this instance?

- Data-independent and control-equivalent
- We will come back to compiler reordering in a few lectures
- Compiler knows about branch delay slot (**it is architectural**)

Filling Branch Delay Slots

Branch Delay Slot
filled w/ nop

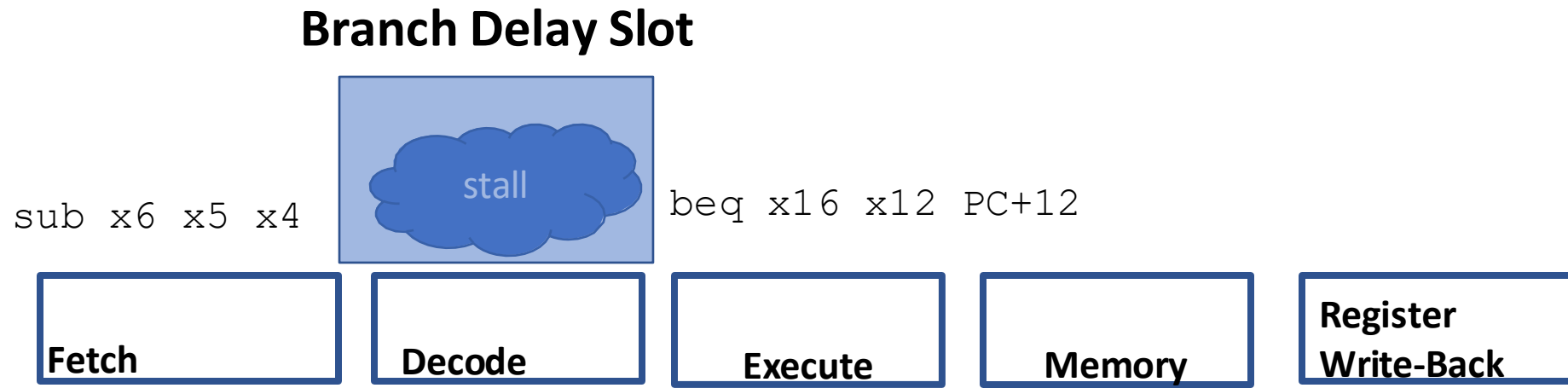
```
beq x16 x12 PC+12
nop
sub x6 x5 x4
add x12 x6 x14
add x12 x7 x9
```



Not really sure what to put in the delay slot. Cannot (or do not want to) slot in taken/not-taken conditional next instructions...

No data-independent, control-equivalent ops to put in slot

RISCV does not have/allow/require delay slots



From the RISC-V RV32I Spec: *“Control transfer instructions in RV32I do not have architecturally visible delay slots.”*

- *What is “architecturally visible”?*
- *Why do they ban delayed branches at the ISA level?*

Why do they ban delayed branches at the ISA level?

“For their first microprocessor with a 5-stage pipeline, this indecision could have caused a one clock-cycle stall of the pipeline. MIPS-32 solved this problem by redefining branch to occur in the instruction after the next one. Thus, the following instruction is always executed. The job of the programmer or compiler writer was to put something useful into the delay slot. Alas, this “solution” didn’t help later MIPS-32 processors with many more pipeline stages (hence many more instructions fetched before the branch outcome is computed), but it made life harder for MIPS-32 programmers, compiler writers, and processor designers ever after, since incremental ISAs demand backwards compatibility (see Section 1.2). In addition, it makes the MIPS-32 code much harder to understand (see Figure 2.10 on page 29). While architects shouldn’t put features that help just one implementation at a point in time, they also shouldn’t put in features that hinder some implementations.”

Branch Prediction to avoid control hazards

Fetch the best guess if we know which way is most likely

20% Not Taken

? 80% Taken

beq x16 x12 PC+12
sub x6 x5 x4
add x12 x6 x14
add x12 x7 x9

beq x16 x12 PC+12

Fetch

Decode

Execute

Memory

**Register
Write-Back**

Branch Prediction to avoid control hazards

Fetch the best guess if we know which way is most likely

20% Not Taken

? 80% Taken

```
beq x16 x12 PC+12
sub x6 x5 x4
add x12 x6 x14
add x12 x7 x9
```

```
beq x16 x12 PC+12
```



How to guess about which way a branch is most likely to resolve?

There is “typical” branch behavior


```
beq x16 x12 PC+12  
sub x6 x5 x4  
add x12 x6 x14  
add x12 x7 x9
```

```
sub x6 x5 x4  
add x12 x6 x14  
add x12 x7 x9  
beq x16 x12 PC-12
```

What will these programs tend to do?


There is “typical” branch behavior

Most branch instructions jump forward



```
beq x16 x12 PC+12
sub x6 x5 x4
add x12 x6 x14
add x12 x7 x9
```

Backward branch: 80% taken



```
sub x6 x5 x4
add x12 x6 x14
add x12 x7 x9
beq x16 x12 PC-12
```

Forward and backward branches have different characteristics
2/3 of all branches are taken in general

Statically defined hints about branches?

```
beq.t90 x16 x12 PC+12  
sub x6 x5 x4  
add x12 x6 x14  
add x12 x7 x9
```

```
sub x6 x5 x4  
add x12 x6 x14  
add x12 x7 x9  
beq.nt90 x16 x12 PC-12
```

From the RISC-V RV32I Spec:

"We considered but did not include static branch hints in the instruction encoding. These can reduce the pressure on dynamic predictors, but require more instruction encoding space and software profiling for best results, and can result in poor performance if production runs do not match profiling runs."

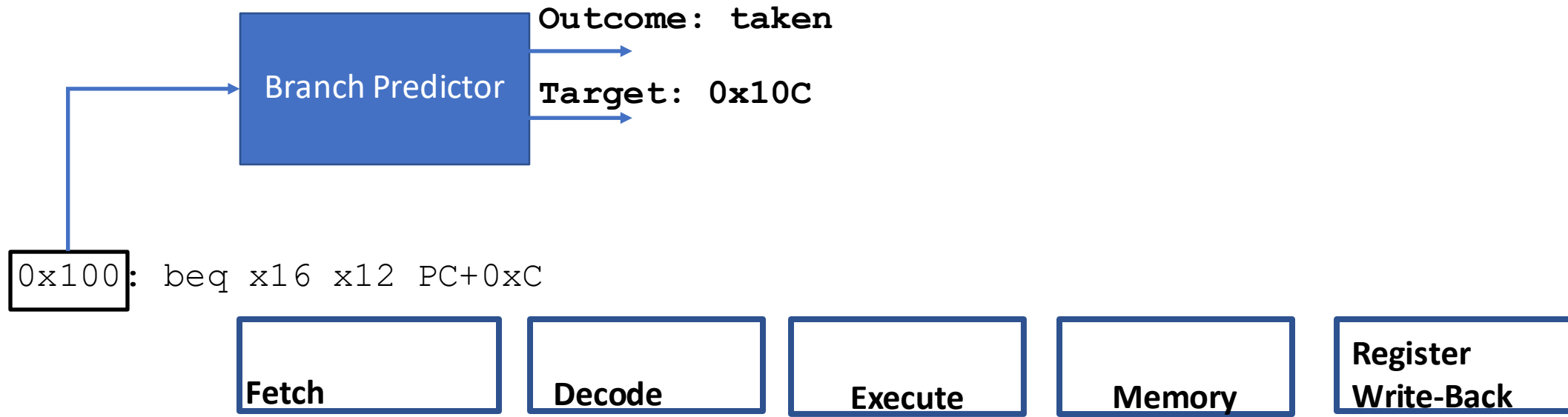
From the Intel's manual:

"Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch. Use these prefixes only with conditional branch instructions (Jcc)."

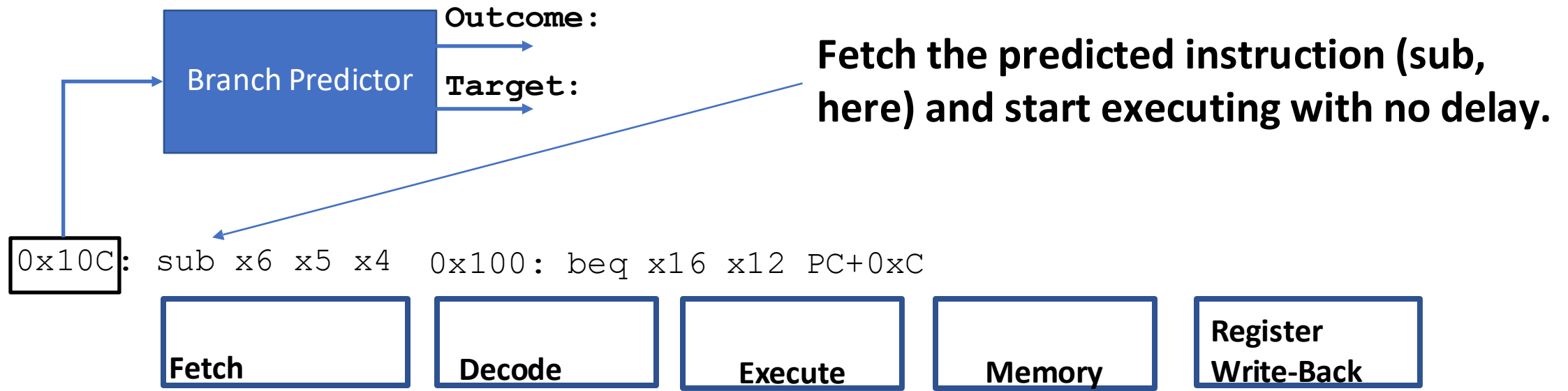
From the internet:

"gcc is right to not generate the prefix, as they have no effect for all processors since the Pentium 4."

Dynamically predicting branch behavior



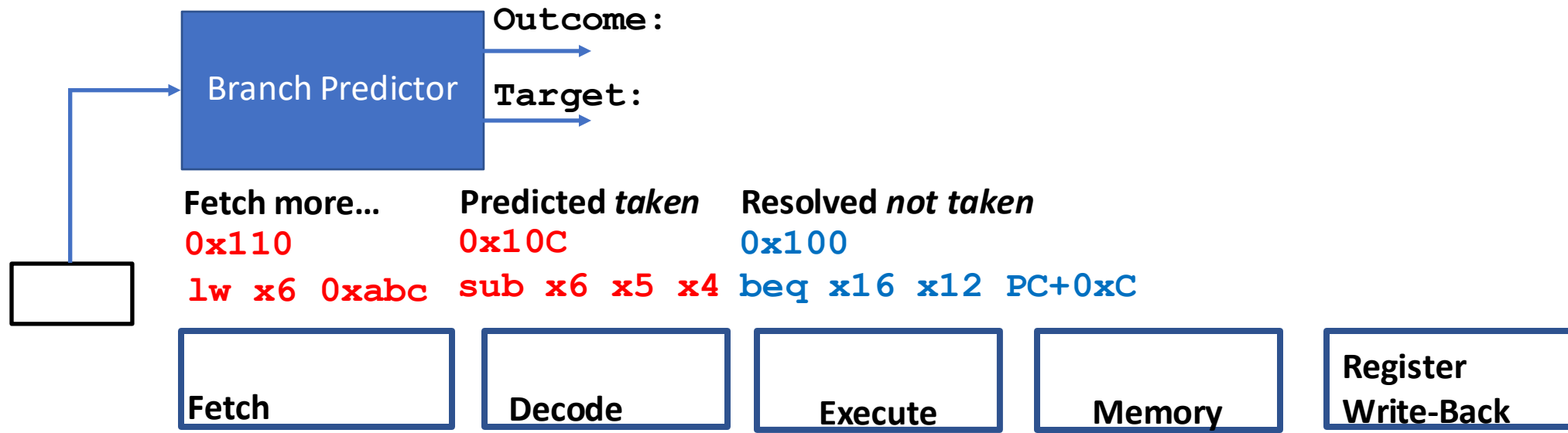
Dynamically predicting branch behavior



A dynamic branch predictor is a circuit that learns the likely outcome of a branch and keeps a record of its past target computations

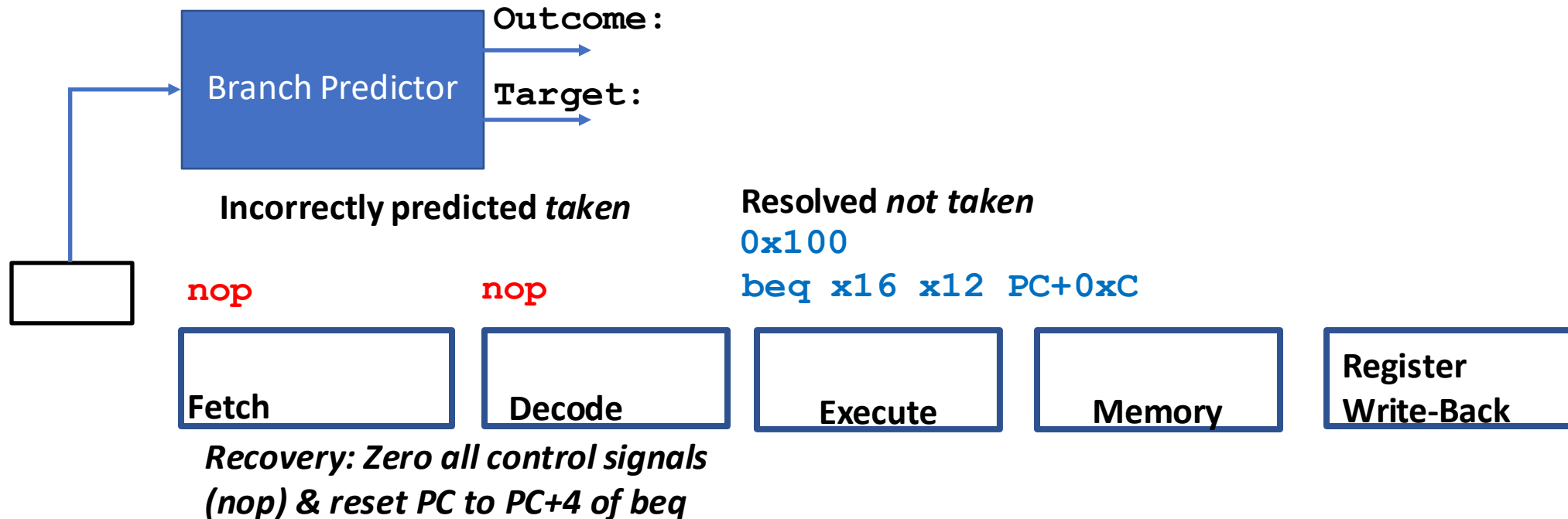
Guess what happens after the current instruction and run *speculatively*.

Speculation entails guessing about what's next



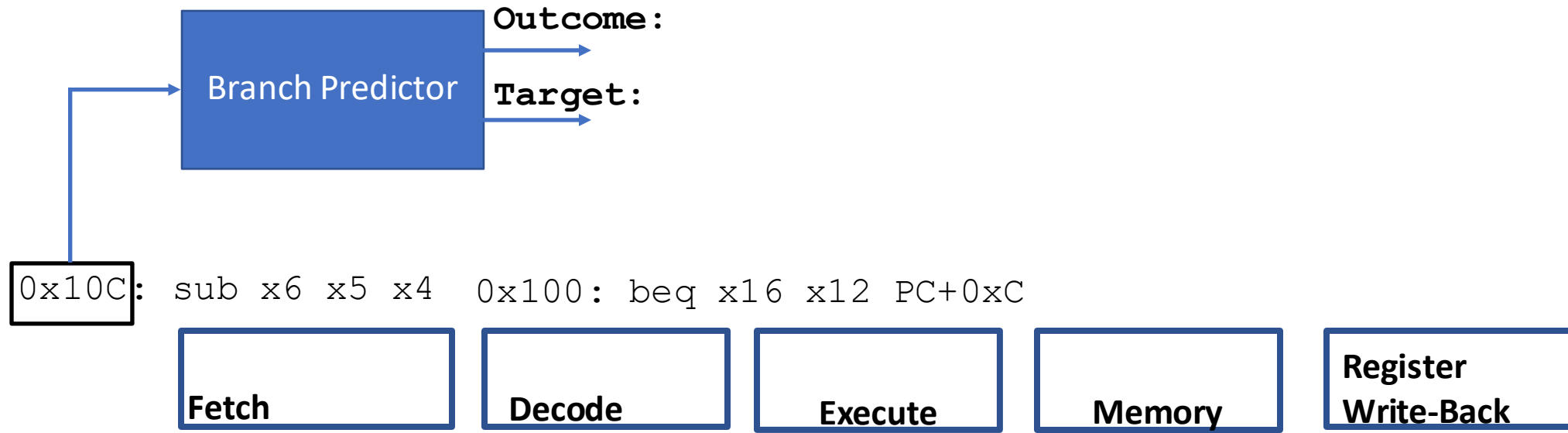
Question: What do we do on a **misprediction** (also known as a *branch mis-speculation*)?

Recovering from branch misprediction



Recovery from *mis-speculation* entails wasted work in the pipeline. What about other state? Register file? Anything else might end up corrupted by speculative execution? Main cost? How cost scales?

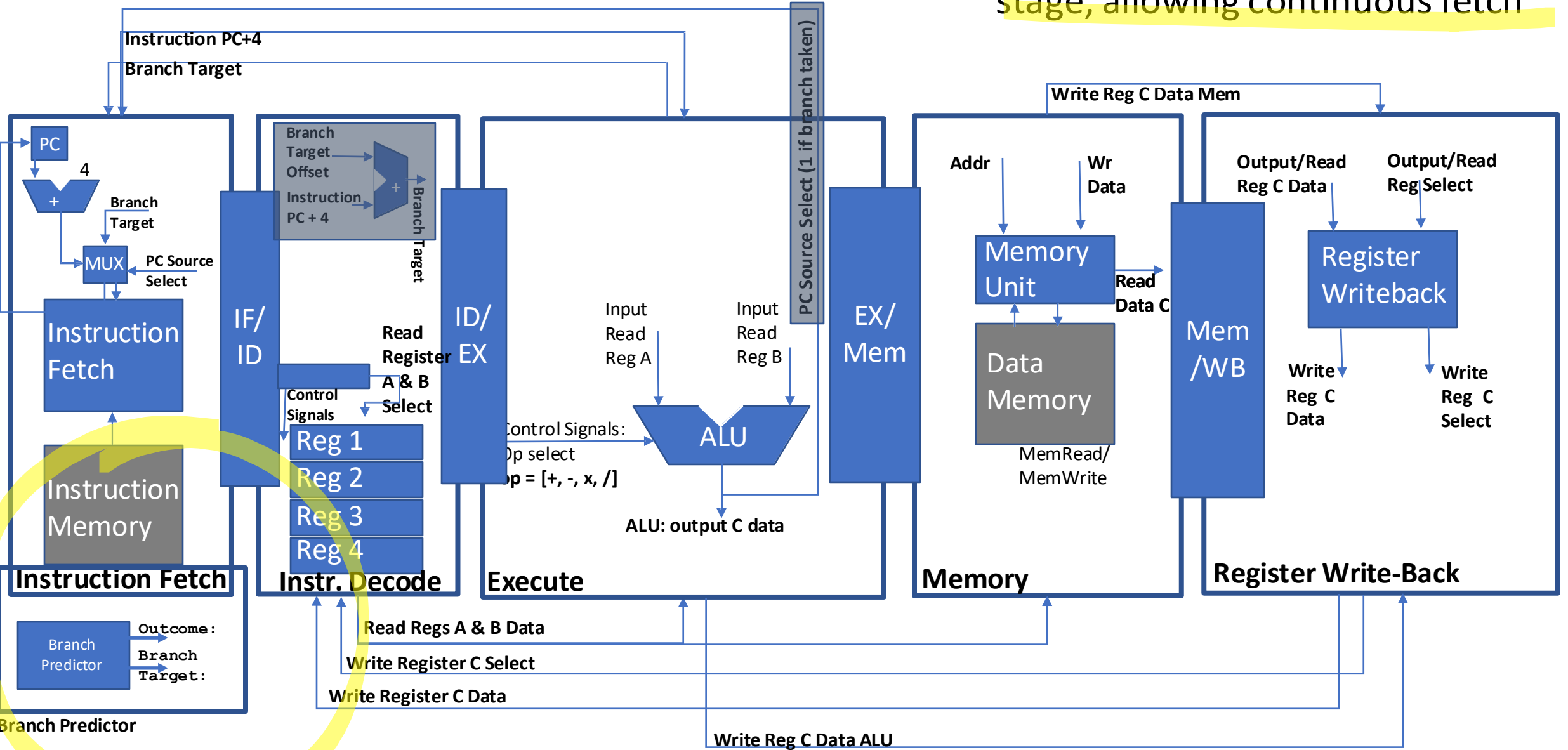
Dynamically predicting branch behavior



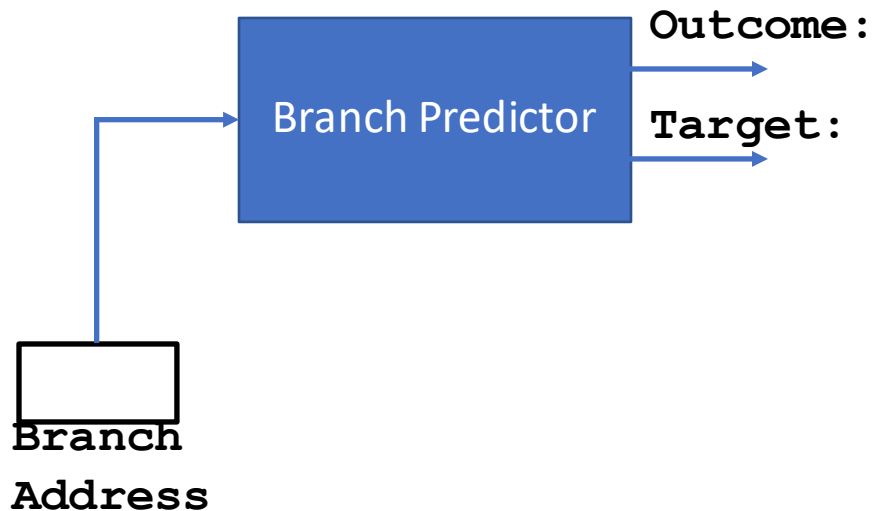
Question: where does the branch predictor live in our pipeline?

Branch Predictor in the pipeline

Key Idea: Add predictor to fetch stage, allowing continuous fetch



Dynamically predicting branch behavior



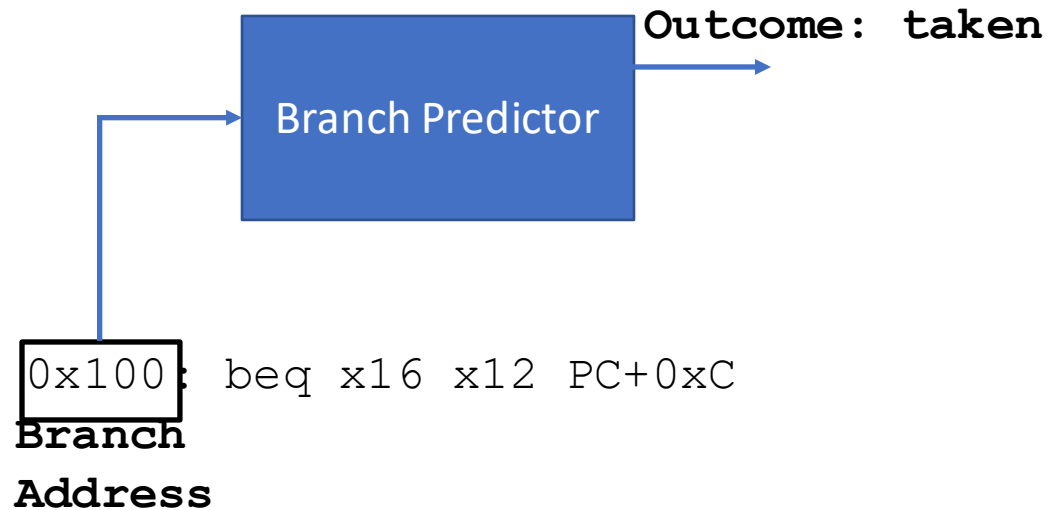
Predictors learn from past behavior

- Need to predict branch outcome: 0/1
- Need to “predict” branch target: PC
- Need to validate prediction
- Need to update predictor

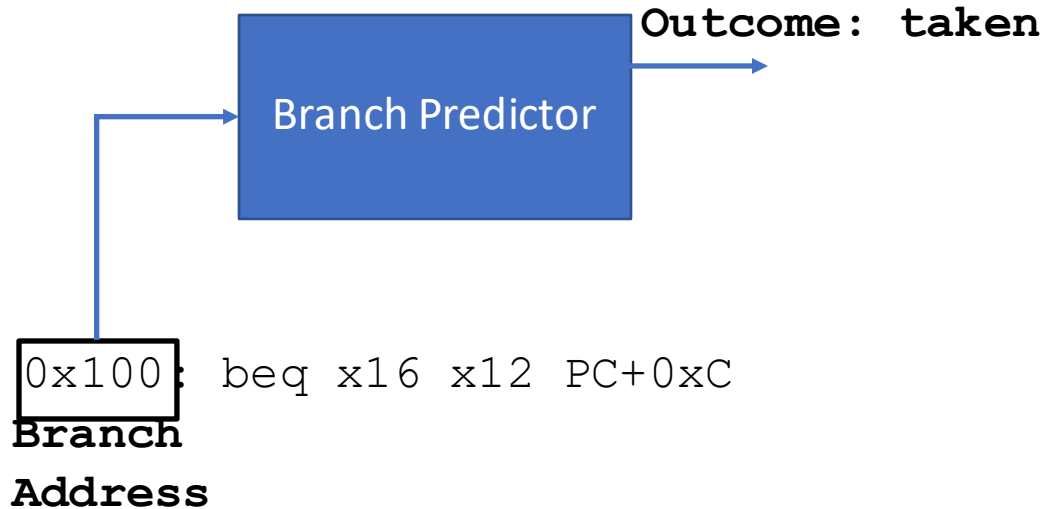
- Many different types of predictor

Predicting Branch Outcomes

What info do we have?



Predicting Branch Outcomes

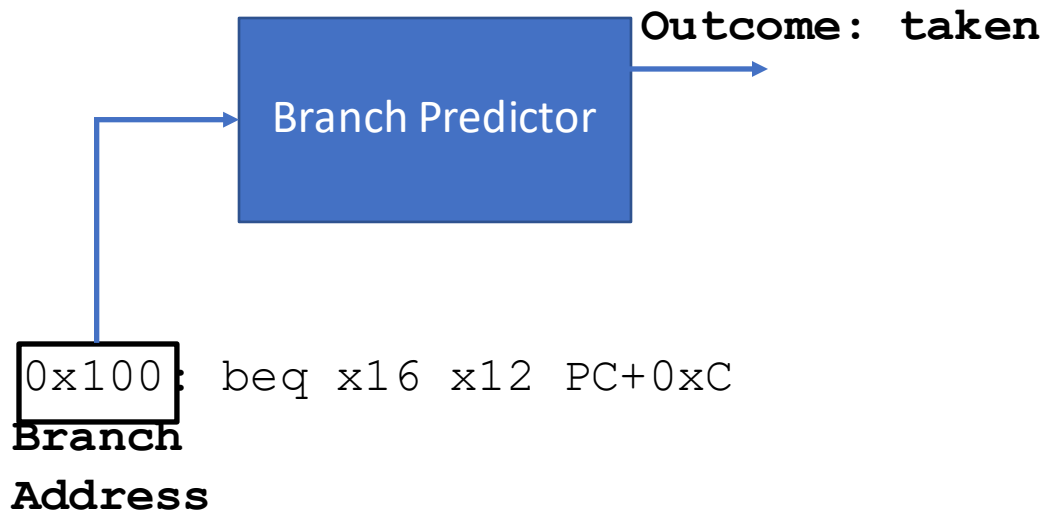


What info do we have?

- History of prior branch outcomes

0x100: 111111111011111111101111111110...

Predicting Branch Outcomes



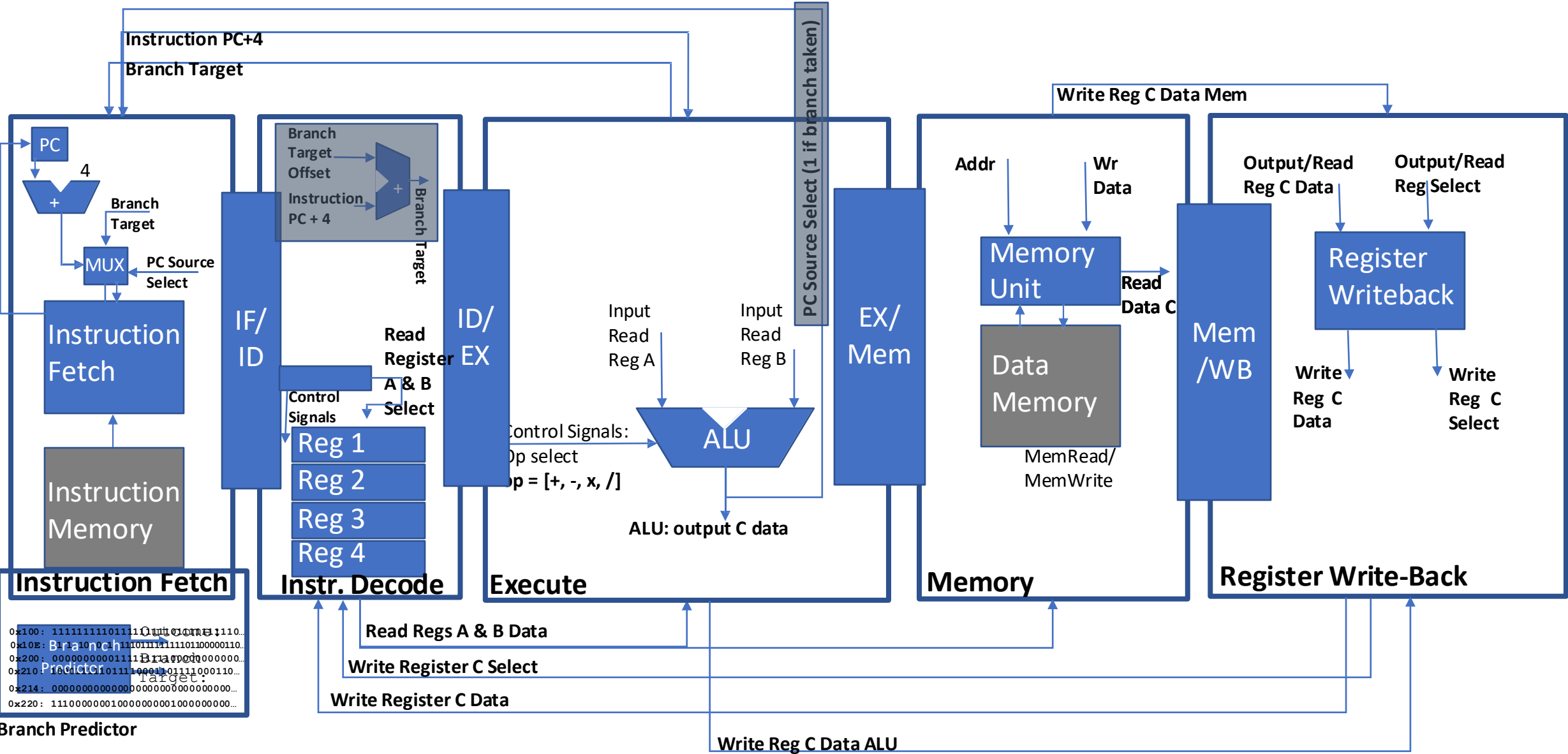
What info do we have?

- History of prior branch outcomes
- For every branch in the program
- **Hardware idea:** keep history in table and choose most likely outcome

```
0x100: 111111111011111111101111111110...
0x10E: 111001111011111111101100000110...
0x200: 000000000011111111100000000000...
0x210: 100011111011110001101111000110...
0x214: 000000000000000000000000000000...
0x220: 111000000010000000001000000000...
```

Branch Predictor in the pipeline

Proposal: Store predictor state in fetch stage with prediction logic



Problems with this design?

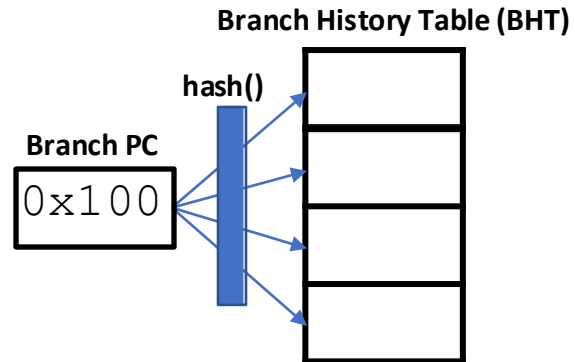
Predicting Branch Outcomes

Making branch history implementable

- Idea 1: Hash table from PC to entry
- Eliminates table size = #branch insns.



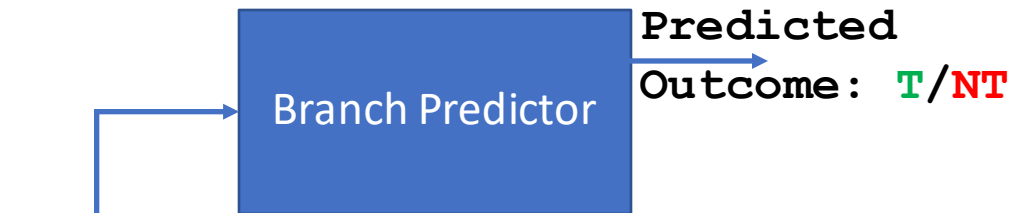
Branch
Address



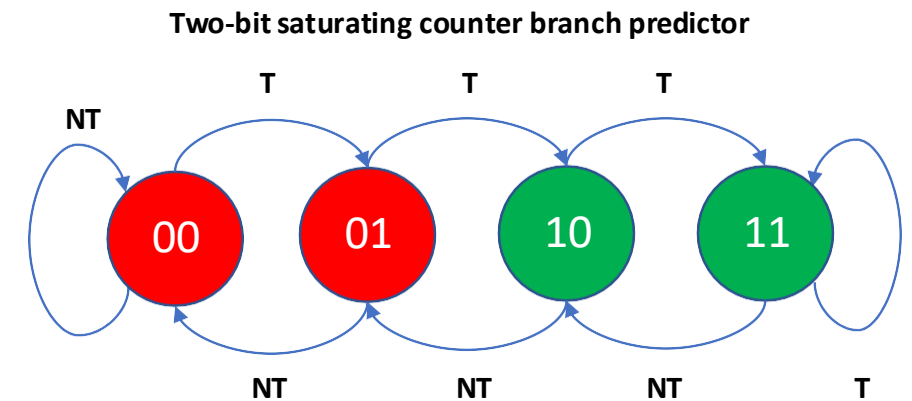
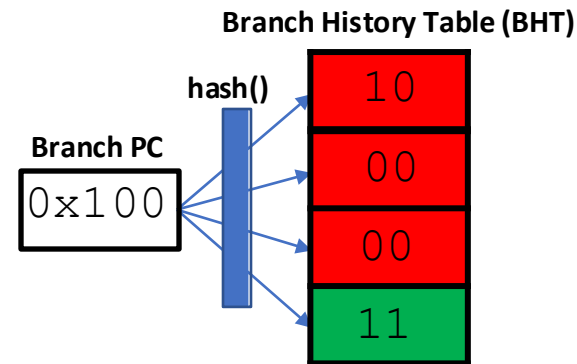
Predicting Branch Outcomes

Making branch history implementable

- Idea 2: Concise history of outcomes
- Eliminates entry size = #branch executions



Branch Address



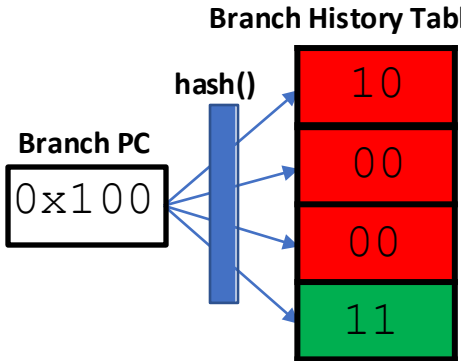
Bimodal BHT Branch Predictor (Lab 1)

Making branch history implementable

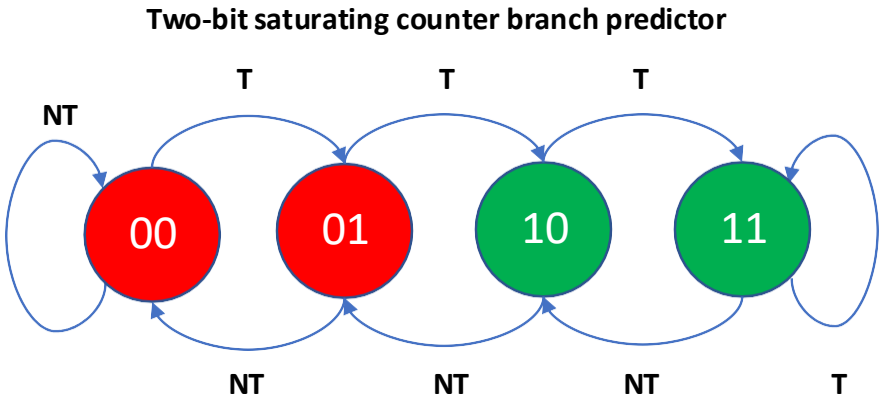
- Idea 2: Concise history of outcomes
- Eliminates entry size = #branch executions



Branch Address



Branch History Table (BHT)



Two-bit saturating counter branch predictor

Example history - 0x100: 1010110110110110...

Bimodal BHT Branch Predictor

["Combining Branch Predictors", McFarling '93]

benchmark	description
doduc	Monte Carlo simulation
eqntott	conversion from equation to truth table
espress	minimization of boolean functions
fp PPP	quantum chemistry calculations
gcc	GNU C compiler
li	lisp interpreter
mat300	matrix multiplication
nasa7	NASA Ames FORTRAN Kernels
spice	circuit simulation
tomcatv	vectorized mesh generation

Figure 2: SPEC Benchmarks Used for Evaluation

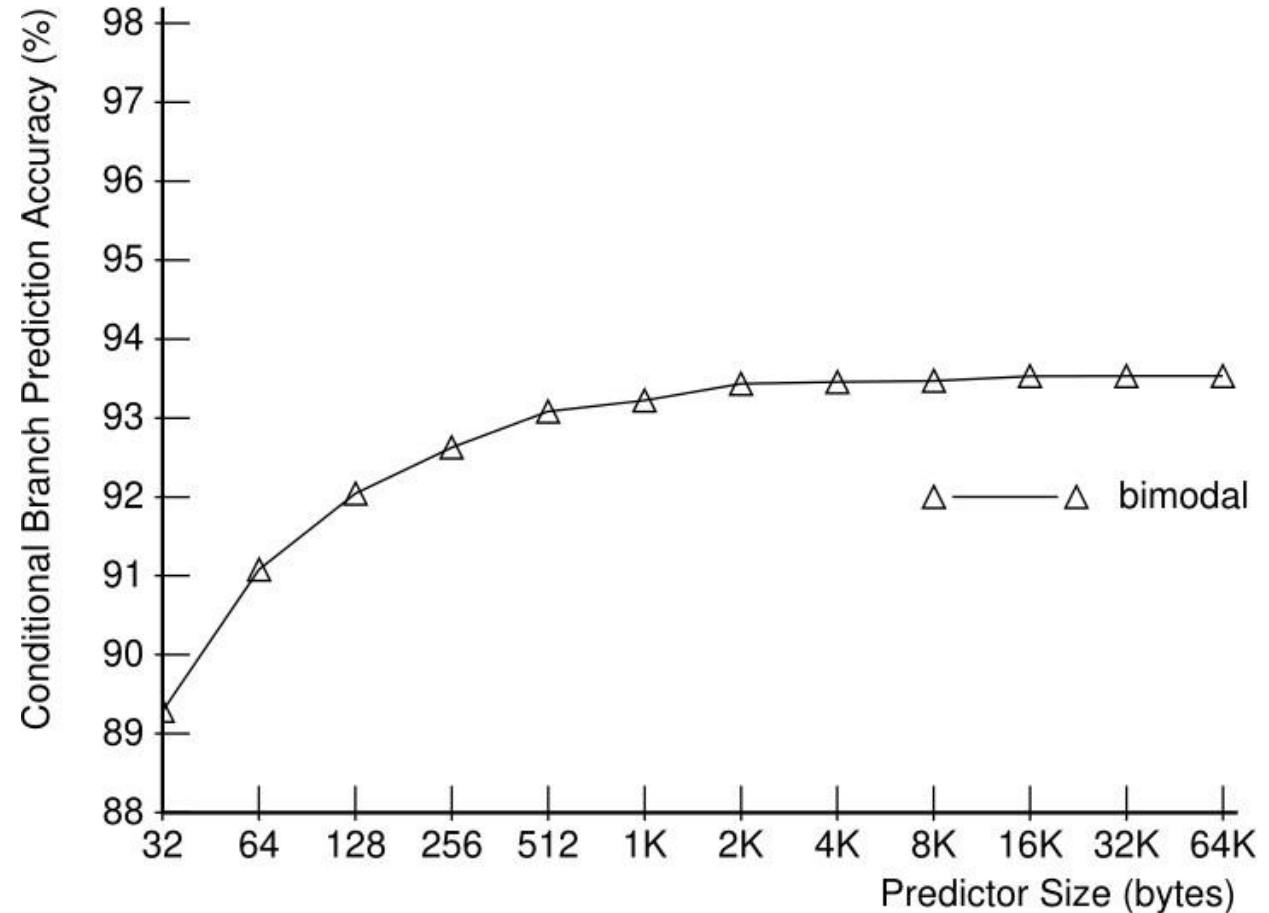
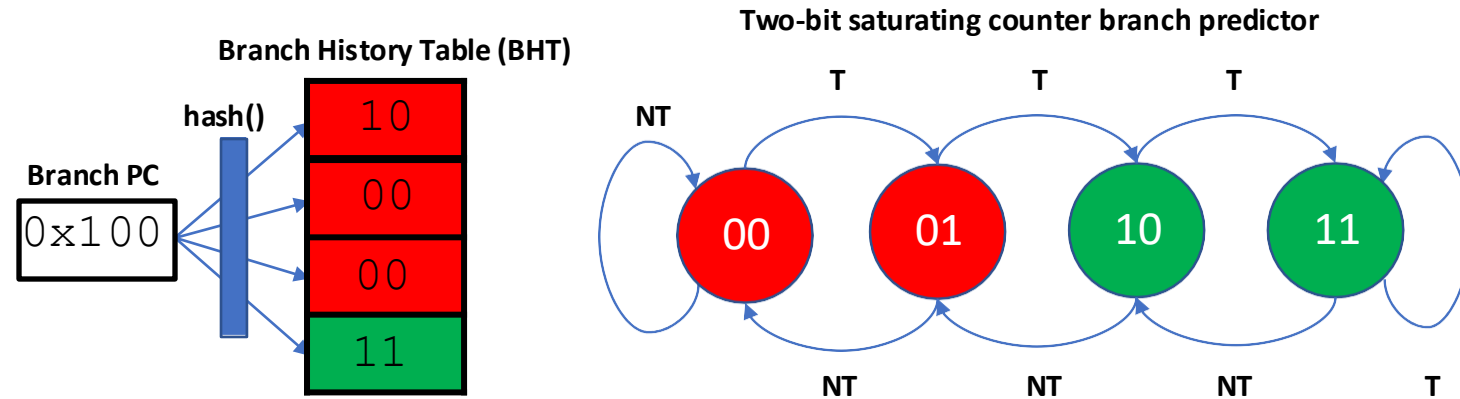


Figure 3: Bimodal Predictor Performance

Predicting Branch Outcomes



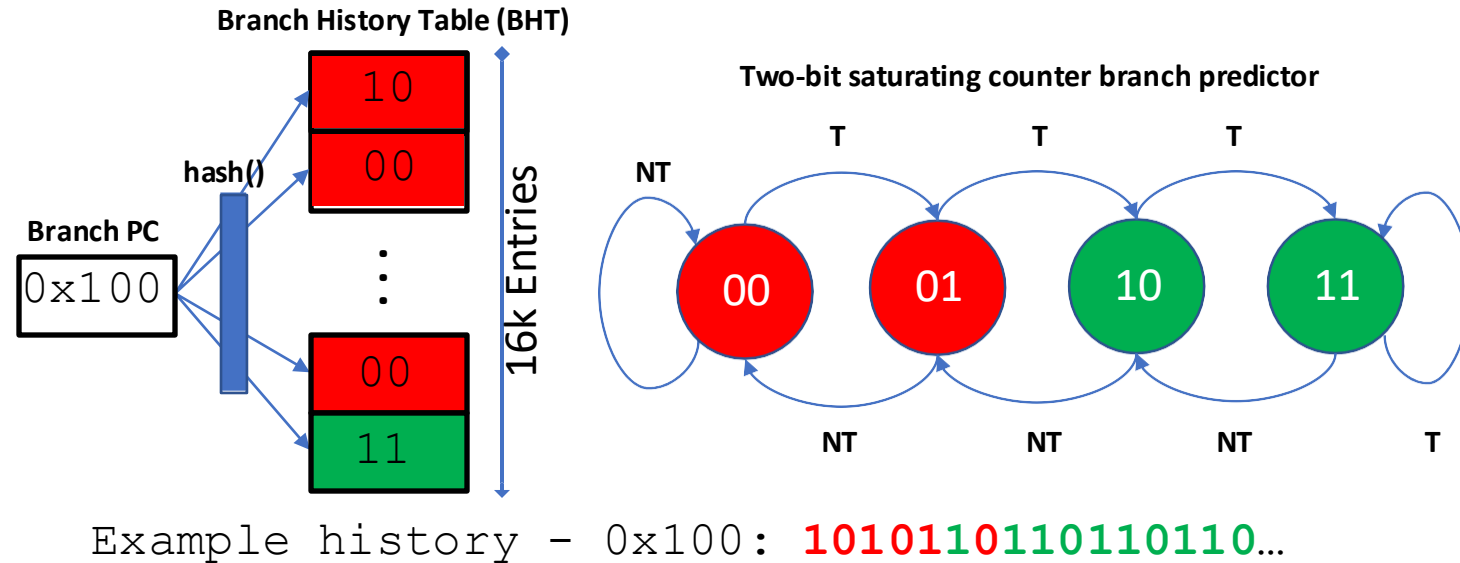
Example history - 0x100: **1010110110110110...**

Limitations of 2-bit BHT branch prediction

- Limitation 1: branch interference due to **hash table collisions**
- Limitation 2: single-branch decision making **misses correlation**

How to handle each of these problems?

Avoiding collisions

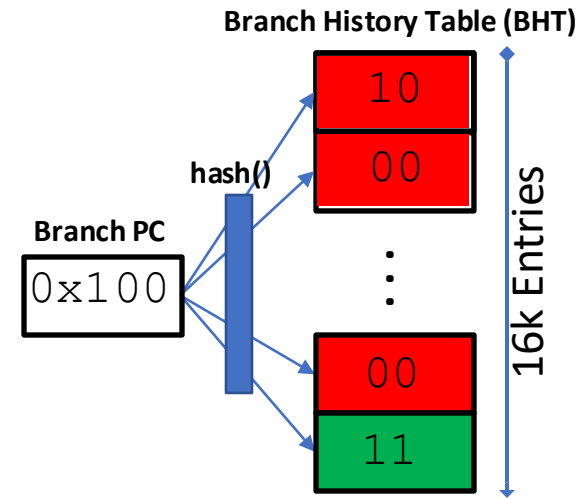


Large table size (e.g., 16k entries) avoids collisions

- Each entry is small, making total cost tolerable (e.g., 32kb)
- Large enough table and collisions do not limit prediction accuracy

Catching correlated branches

```
if (a == 1) { a=0 }  
if (b == 1) { b=0 }  
if (a != b) { ... }
```

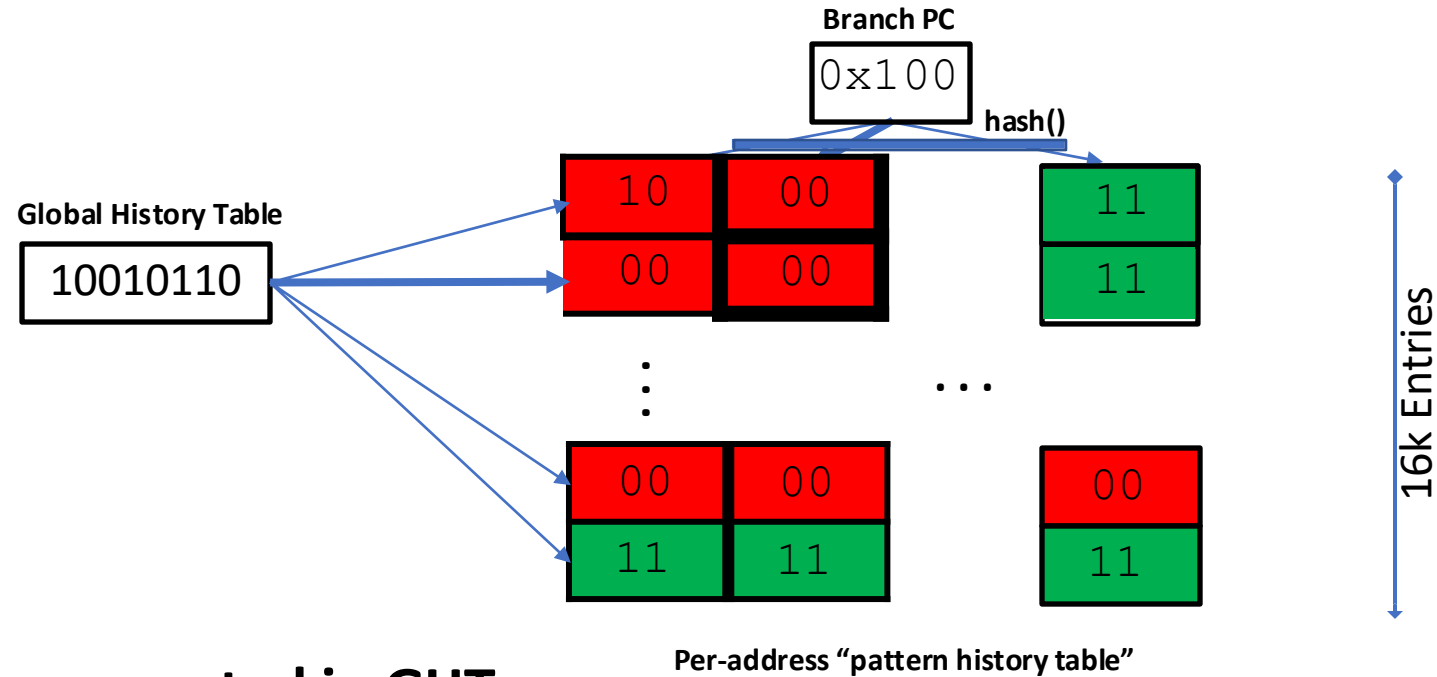


There are correlation of the outcomes of consecutive branches

- The outcome of the third branch is **correlated** with the first two
- Our per-branch predictor cannot capture this common pattern

Two-Level Branch Predictor (Option for Lab 1): GAp (Global Adaptive w/ per-address table)

```
if (a == 1) { a=0 }  
if (b == 1) { b=0 }  
if (a != b) { ... }
```

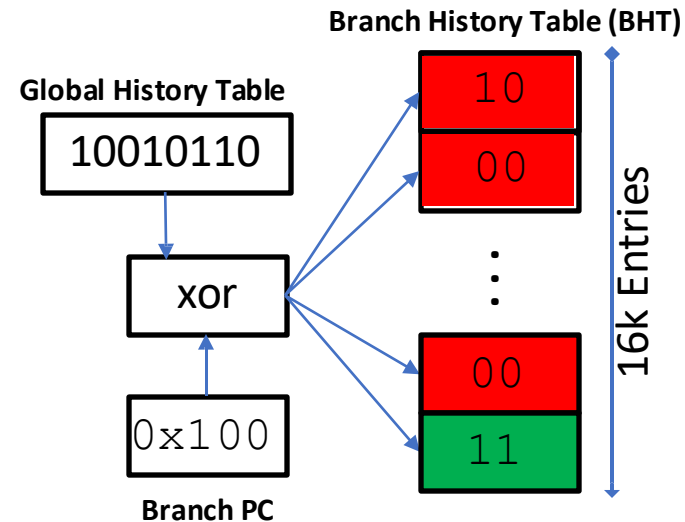


Track history of outcomes of all branches executed in GHT

- Use PC to select which PHT to use
- Use global pattern history to index into PHT
- Use PHT entry's 2-bit counter to predict outcome
- **After each branch resolves, updated predictor in per-address pattern history table & shift its outcome (T=1, NT=0) into GHT**

Global Index Sharing Predictor

```
if (a == 1) { a=0 }  
if (b == 1) { b=0 }  
if (a != b) { ... }
```

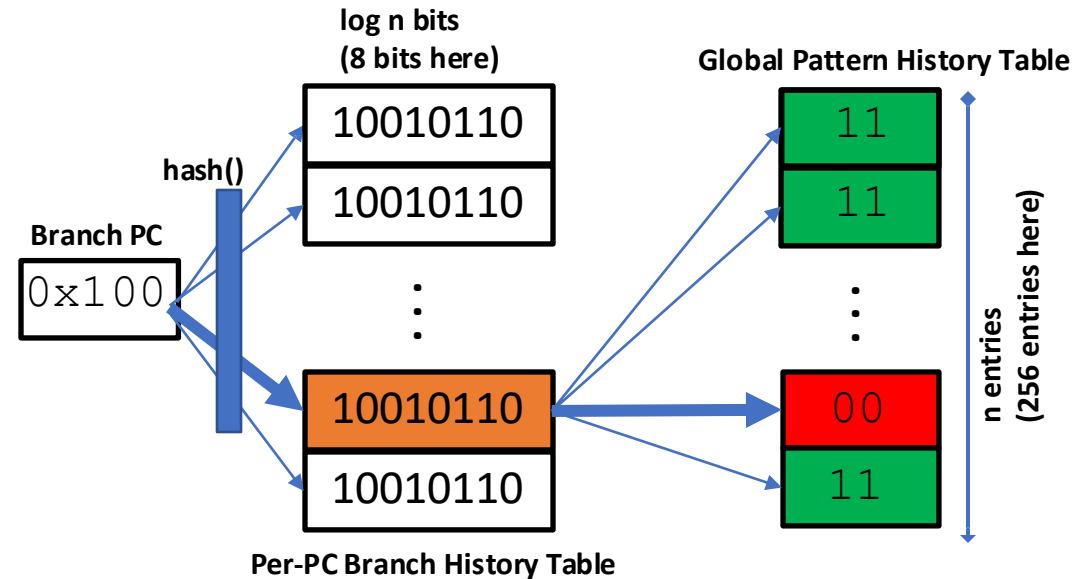


Index sharing predictor tracks local history in global context *concisely*

- XOR GHT with branch PC to select BHT
- Use 2-bit counter in BHT to make prediction for branch in GHT context
- XOR maps branches & contexts that matter to different BHTs
- **Gshare combining addr bits with history bits often *better***

Local/Global Correlating Predictor (Optional for Lab 1): PAg (Per-Address Adaptive global history table)

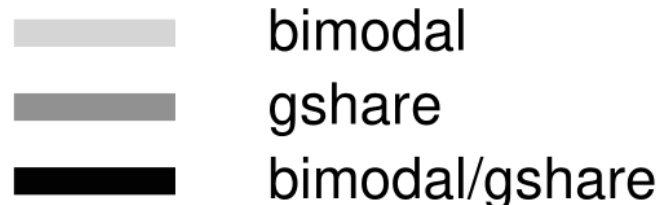
```
if (a == 1) { a=0 }  
if (b == 1) { b=0 }  
if (a != b) { ... }
```



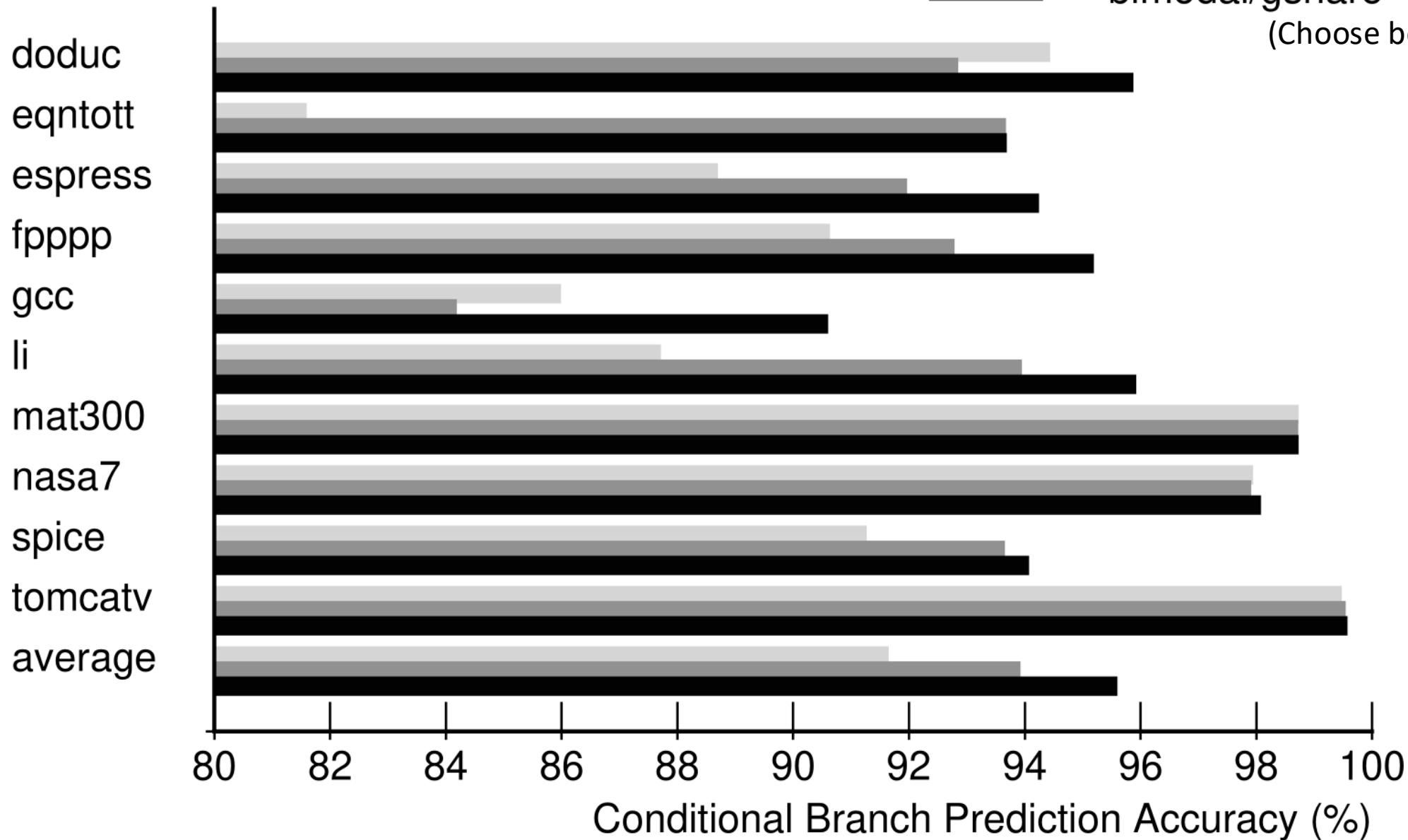
Use per-branch history to index into a global, shared table of predictors. Per-PC branch history table stores history for that branch only, not global history.

- Use PC to select which **BHT** to use
- Use branch history to index into global **PHT**
- Use **PHT** entry's 2-bit counter to predict outcome

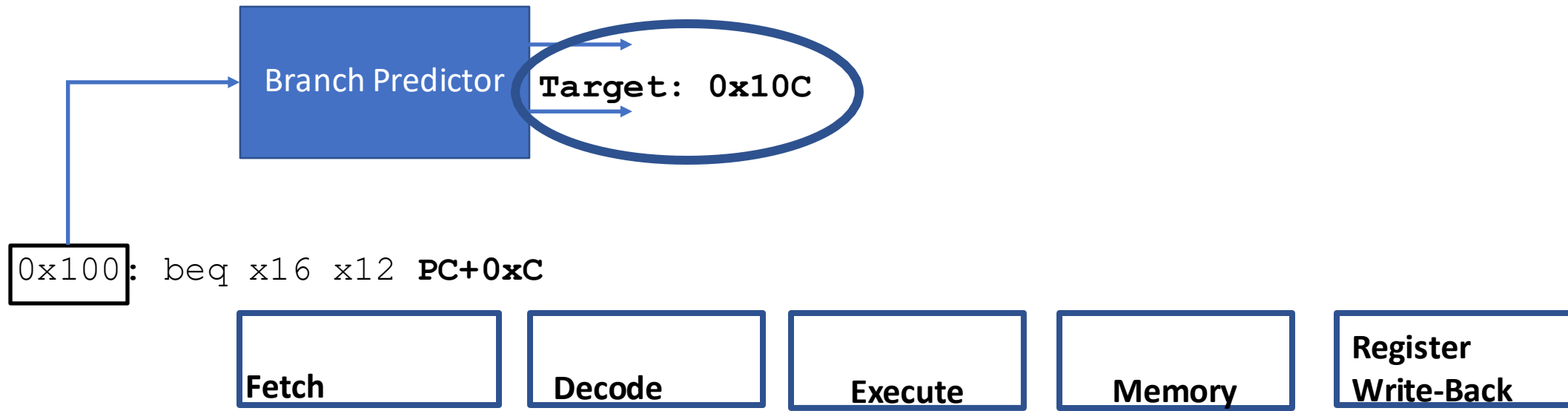
Quantifying Predictor Accuracy



(Choose best option)



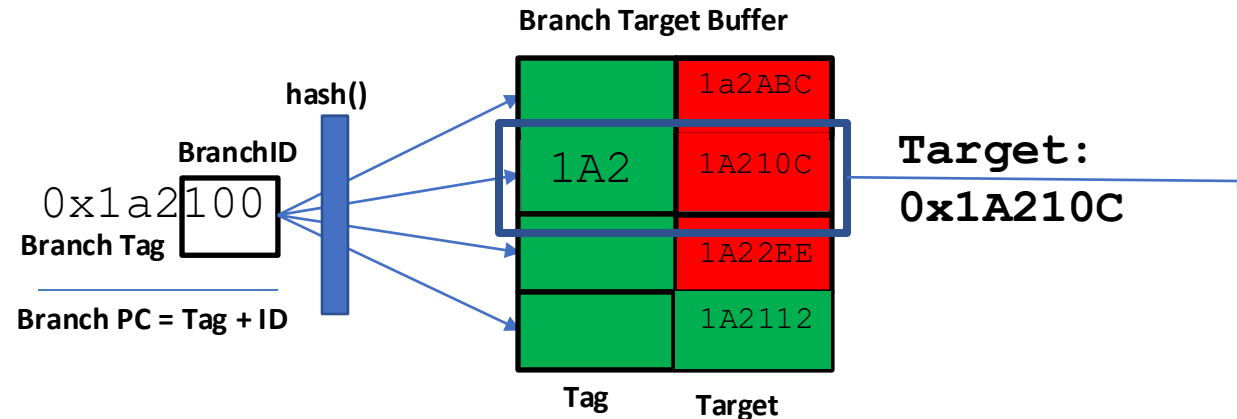
Dynamically predicting branch behavior



Need to predict branch target

- Target gets resolved only in Decode, which leads to 1-cycle stall
- Predict outcome and target both in Fetch & avoid all stalls

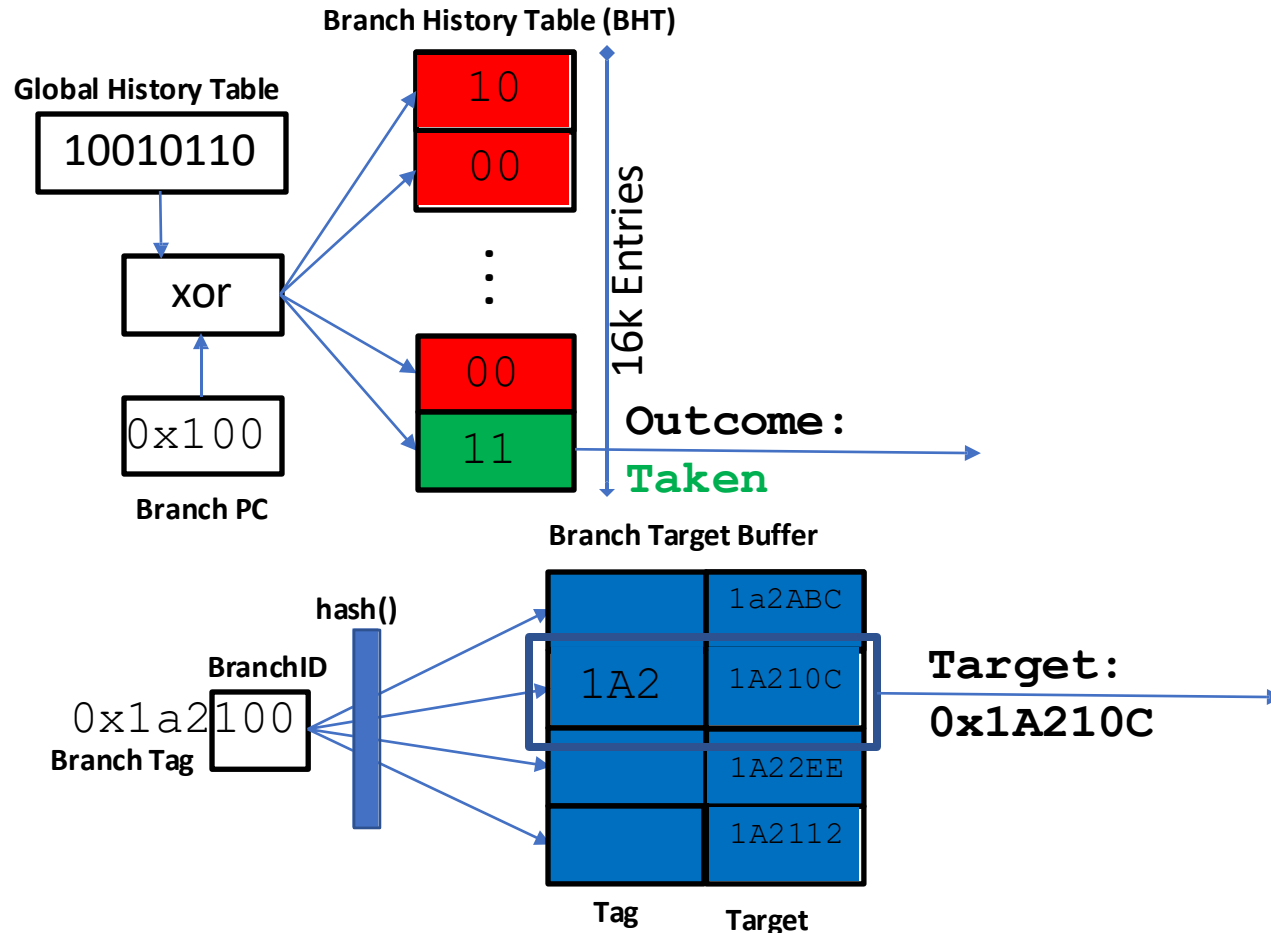
Branch Target Buffer Implementation



Branch Target Buffer (BTB) logs branch target

- BTB is associative memory table indexed by branch PC low order bits
- Need tag because *some PCs do not point to branches*
- Associative memory can be set-, fully-associative or direct-mapped

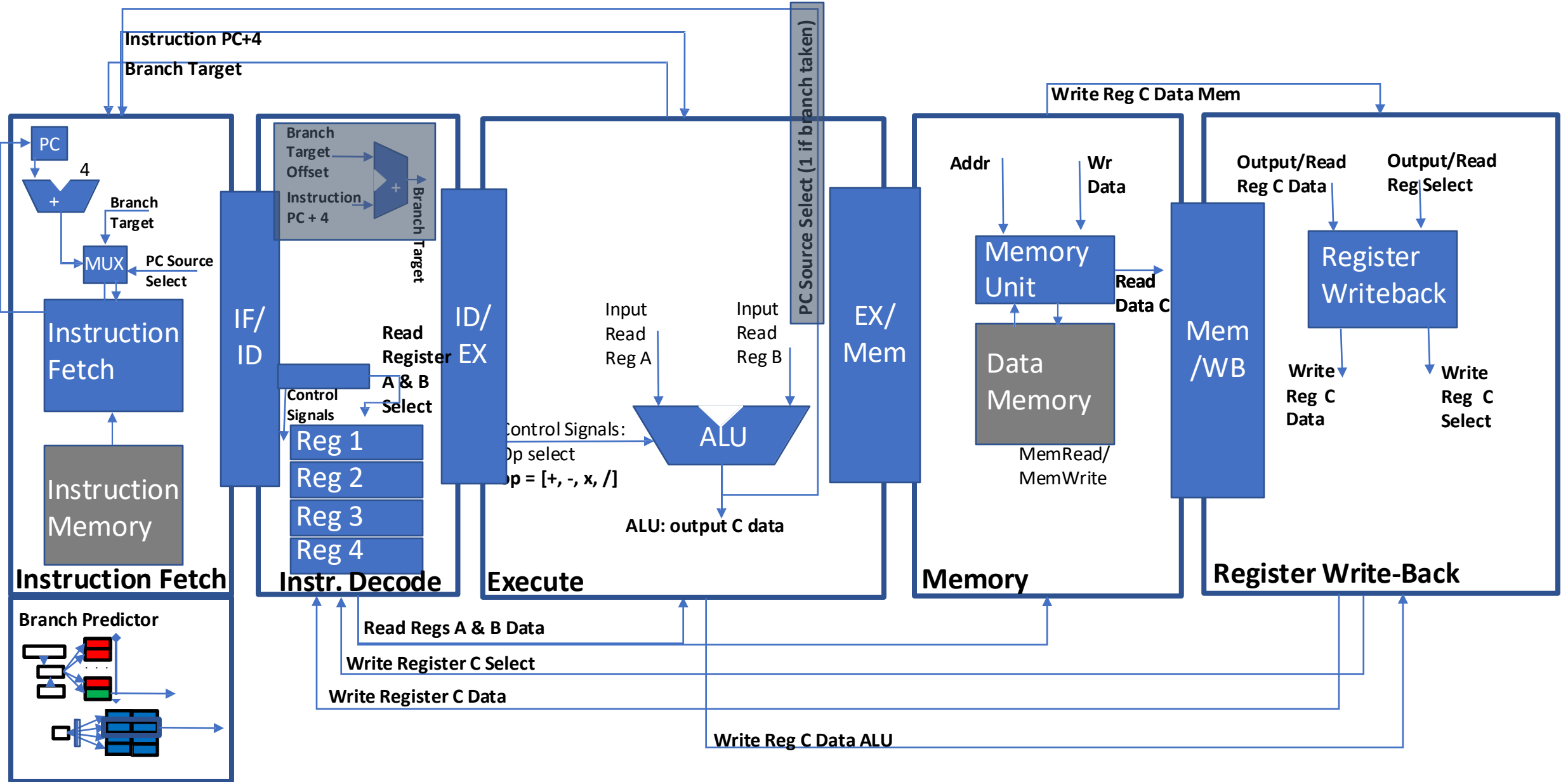
Putting it all together: A Gshare branch predictor + BTB



Branch predictors resolve branches in the fetch stage avoiding stalls

- Need misprediction detection logic added to decode stage
- Need logic to flush instructions on predicted path after misprediction
- Flushed instructions are effectively stalls in the pipeline, but worse: wasted work.

Branch Predictor in the pipeline



What did we just learn?

- Control hazards introduce stalls because on a branch the pipeline doesn't know what to fetch next
- Single stall cycle with early branch resolution (in Decode)
- Branch delay slots and static prediction do OK, but still need stalls and nops often
- Dynamic Branch Prediction uses per-branch information and global branch outcome history information to predict outcomes and targets
- Branch predictor accuracies are in the 90+% in a lot of cases (you will see these figures in Lab 1)

What to think about next?

- Caches as a microarchitectural optimization (next time)
 - Implementation of cache hierarchies
 - Cache design tradeoffs
- Performance Evaluation (next next time)
 - Design spaces, Pareto Frontiers, and design space exploration