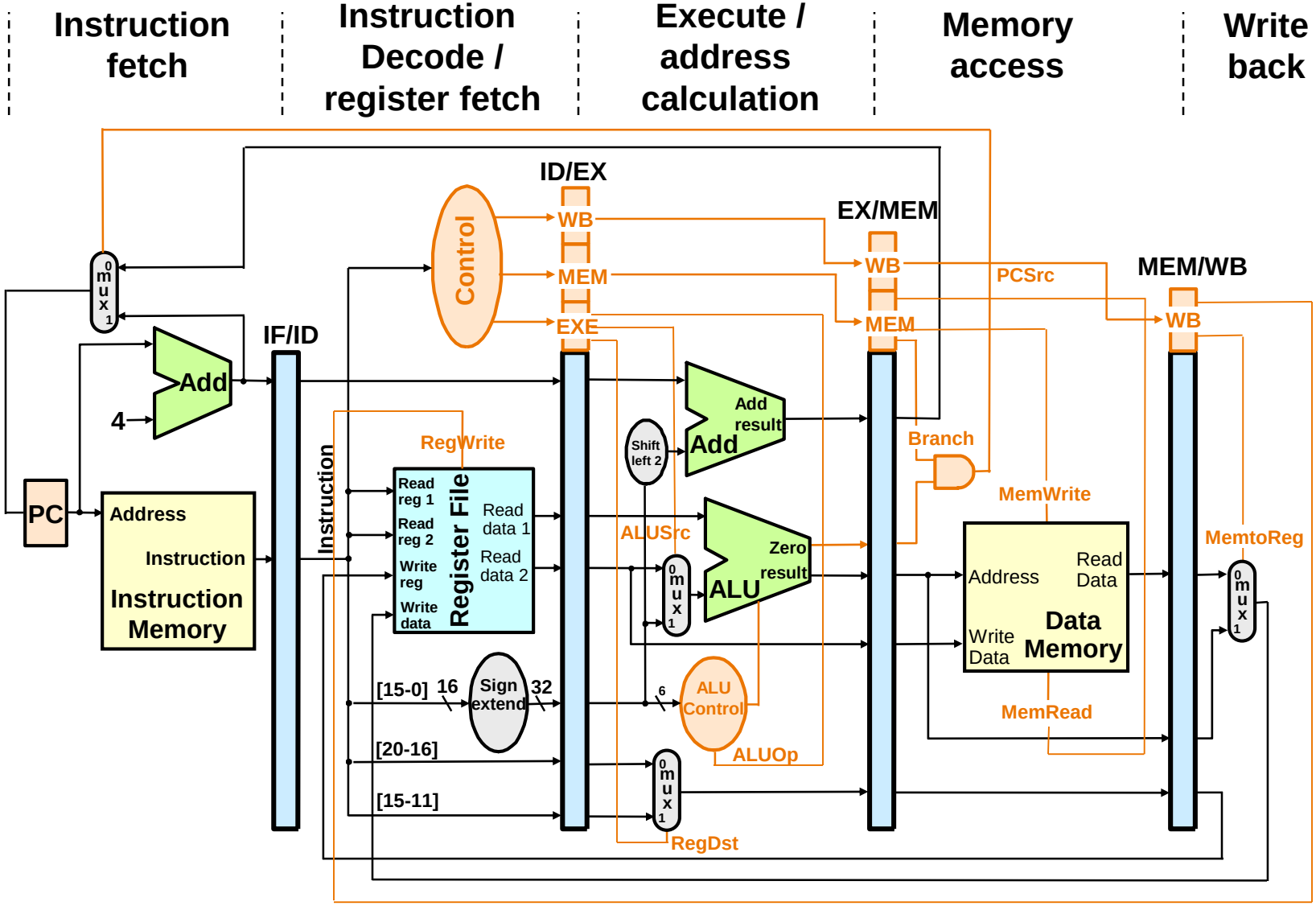
Computer Architecture

Pipeline

By Yoav Etsion & Dan Tsafrir
Presentation based on slides by David Patterson, Avi Mendelson, Randi Katz, and Lihu Rappoport

Pipelined CPU with Control



Pipeline Hazards:

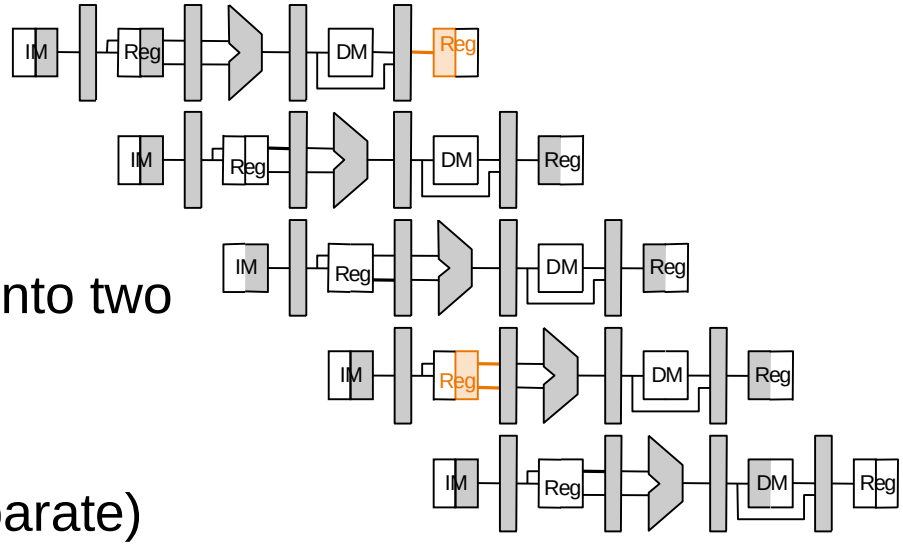
1. Structural Hazards

Structural Hazard

- **Two instructions attempt to use same resource simultaneously**

- **Problem: register-file accessed in 2 stages**
  - Write during stage 5 (WB)
  - Read during stage 2 (ID)

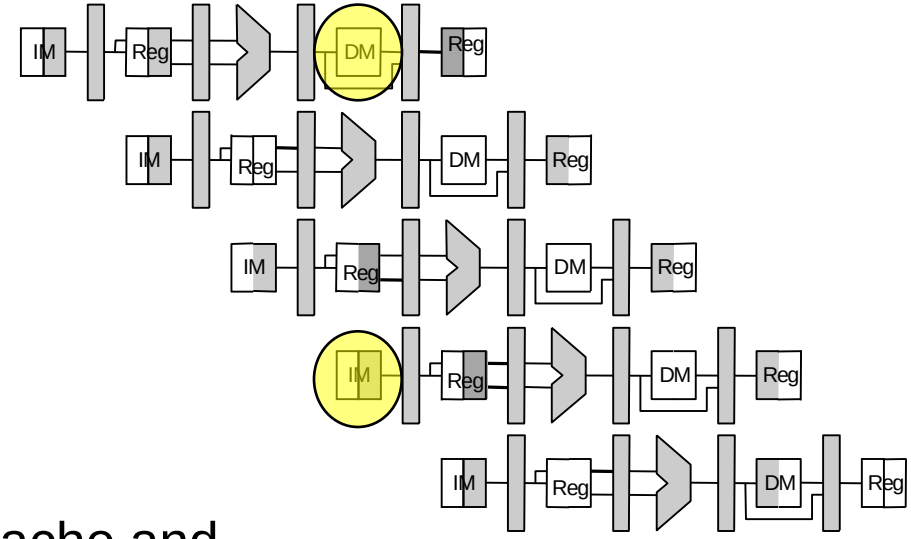
  => Resource (RF) conflict



- **Solution**
  - Clock effectively splits stage into two
  - Reads are combinatorial;
    Writes are sequential
  - 2 read ports, 1 write port (separate)

Structural Hazard

- **Problem: memory accessed in 2 stages**
  - Fetch (stage 1), when reading instructions from memory
  - Memory (stage 4), when data is read/written from/to memory
  - Princeton architecture



- **Solution**
  - Split data/inst. Memories
    - Harvard architecture
  - Today, separate instruction cache and data cache

Pipeline Hazards:

2. Data Hazards

Data Dependencies

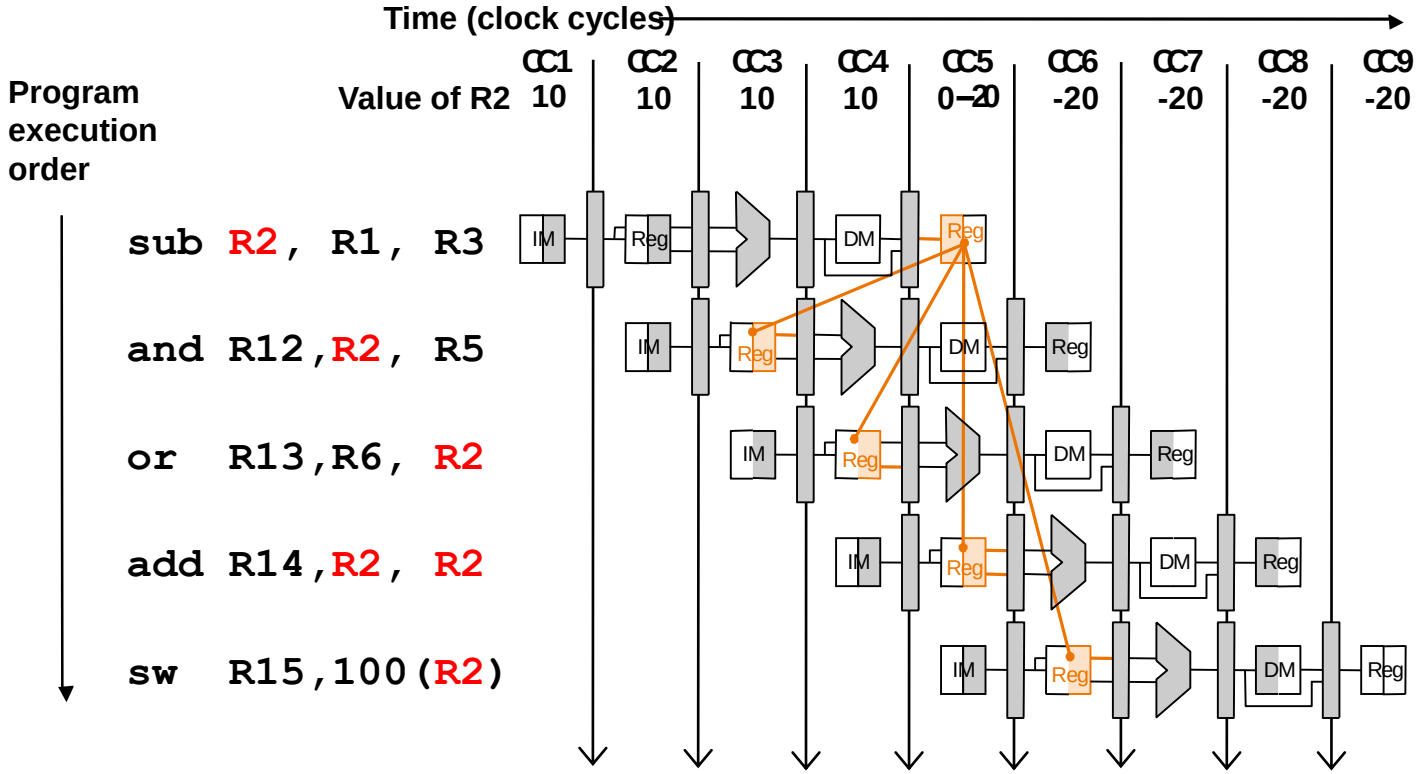
- When two instructions access the same register

- **RAW**: Read-After-Write
  - True dependency
- **WAR**: Write-After-Read
  - Anti-dependency
- **WAW**: Write-After-Write
  - False-dependency

- Key problem with regular ***in-order*** pipelines is RAW
  - We will also learn about *out-of-order* pipelines

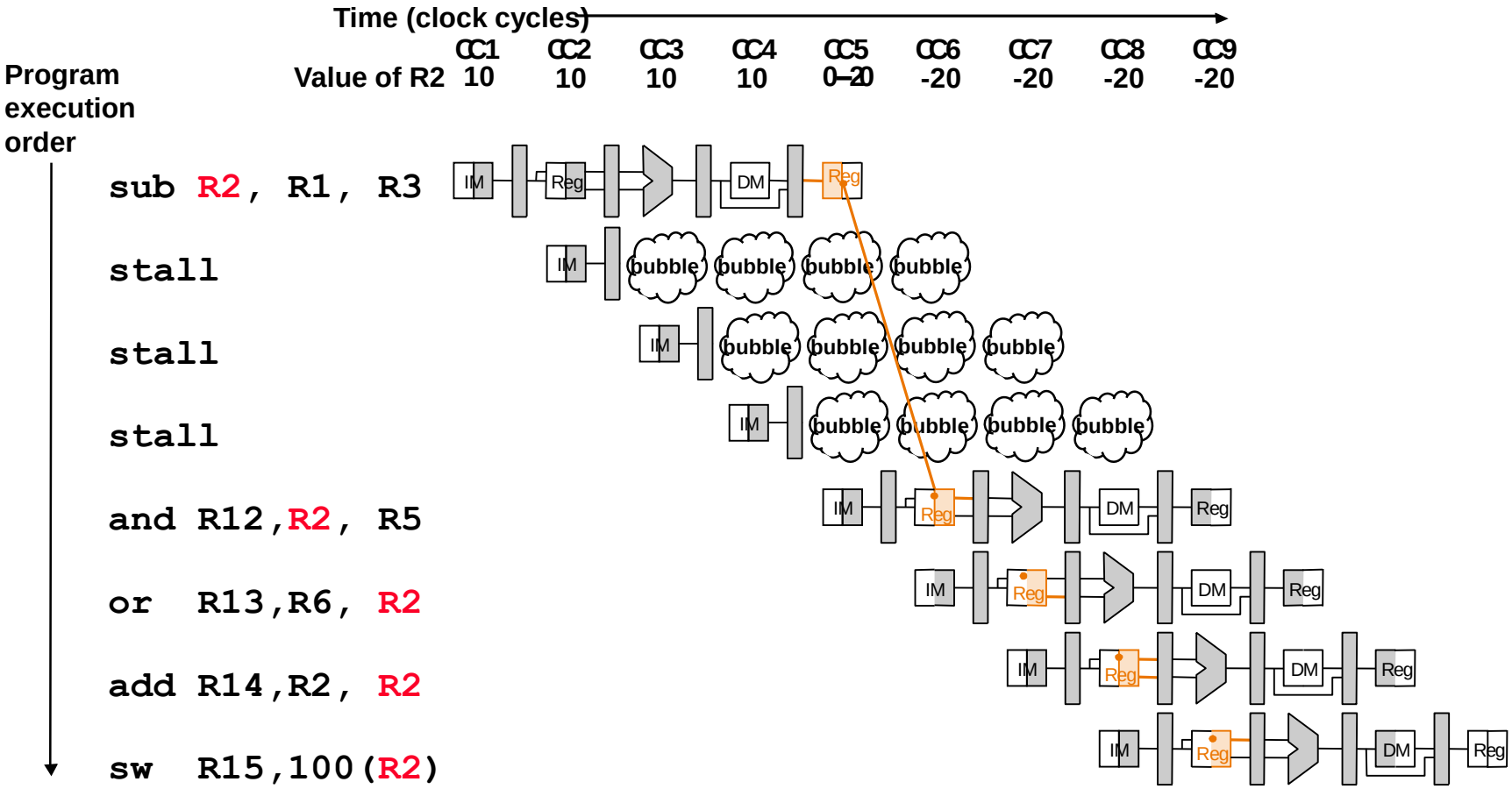
Data Dependencies

- **Problem with starting the next instruction before the first is finished**
  - **dependencies that "go backward in time" are data hazards**



RAW Hazard: HW Solution 1 - Add Stalls

- Let the hardware detect hazard and add stalls if needed



Problem: slow! **Solution**: forwarding whenever possible

RAW Hazard: HW Solution 2 - Forwarding

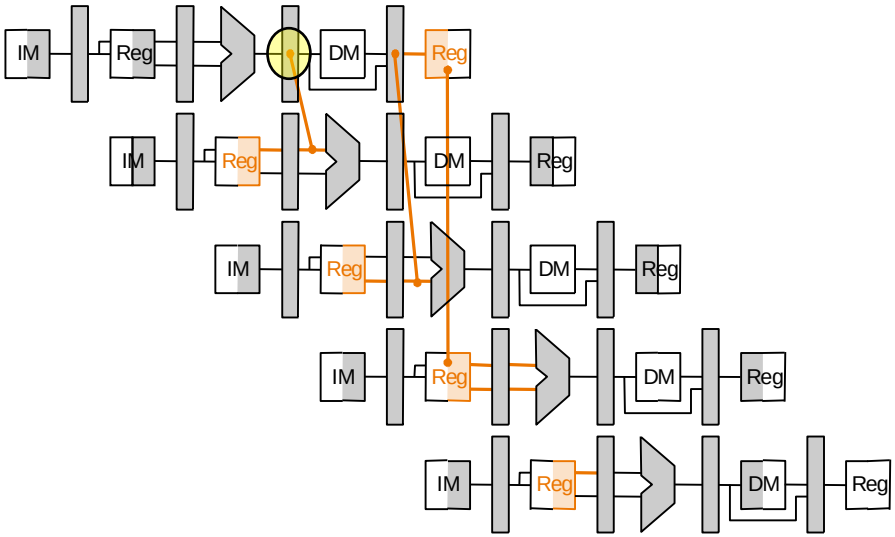
- Use temporary results, don't wait for them to be written to the register file
  - register file forwarding to handle read/write to same register
  - ALU forwarding

Time (clock cycles) →

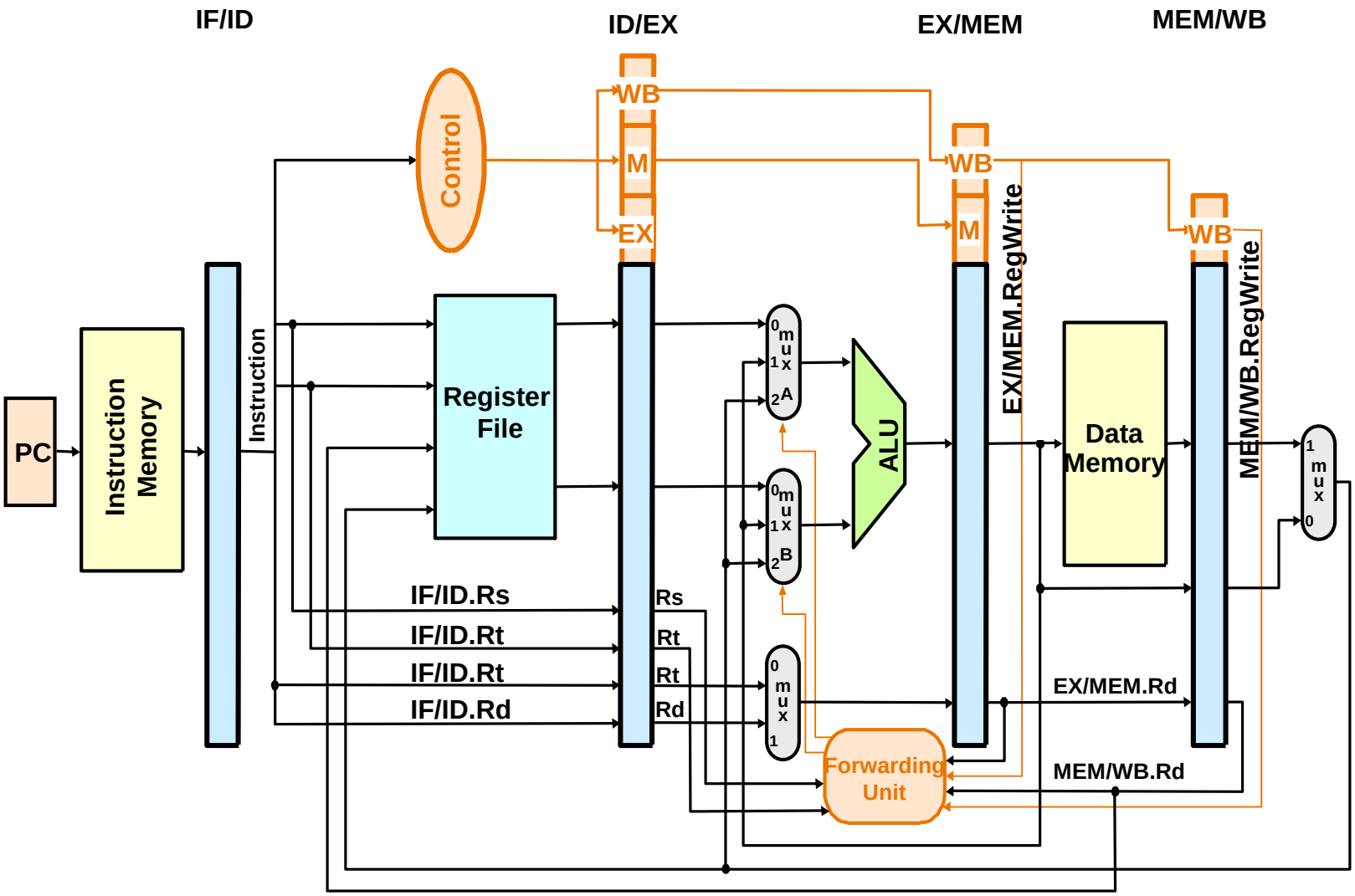
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
Value of R2	10	10	10	10	0–20	-20	-20	-20	-20
Value EX/MEM	X	X	X	–20	X	X	X	X	X
Value MEM/WB	X	X	X	X	–20	X	X	X	X

Program execution order

```
sub R2, R1, R3
and R12,R2, R5
or  R13,R6, R2
add R14,R2, R2
sw  R15,100(R2)
```

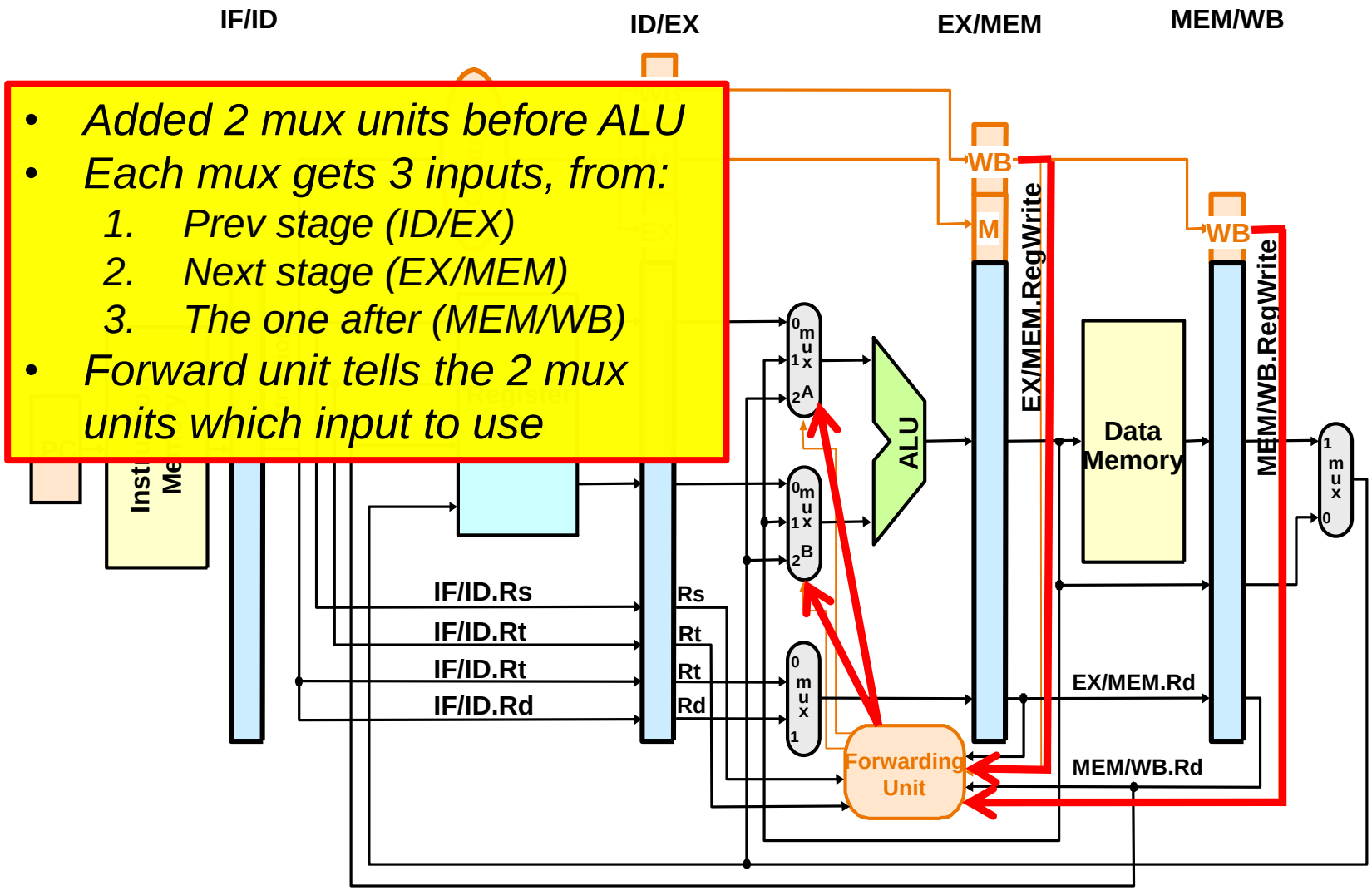


Forwarding Hardware



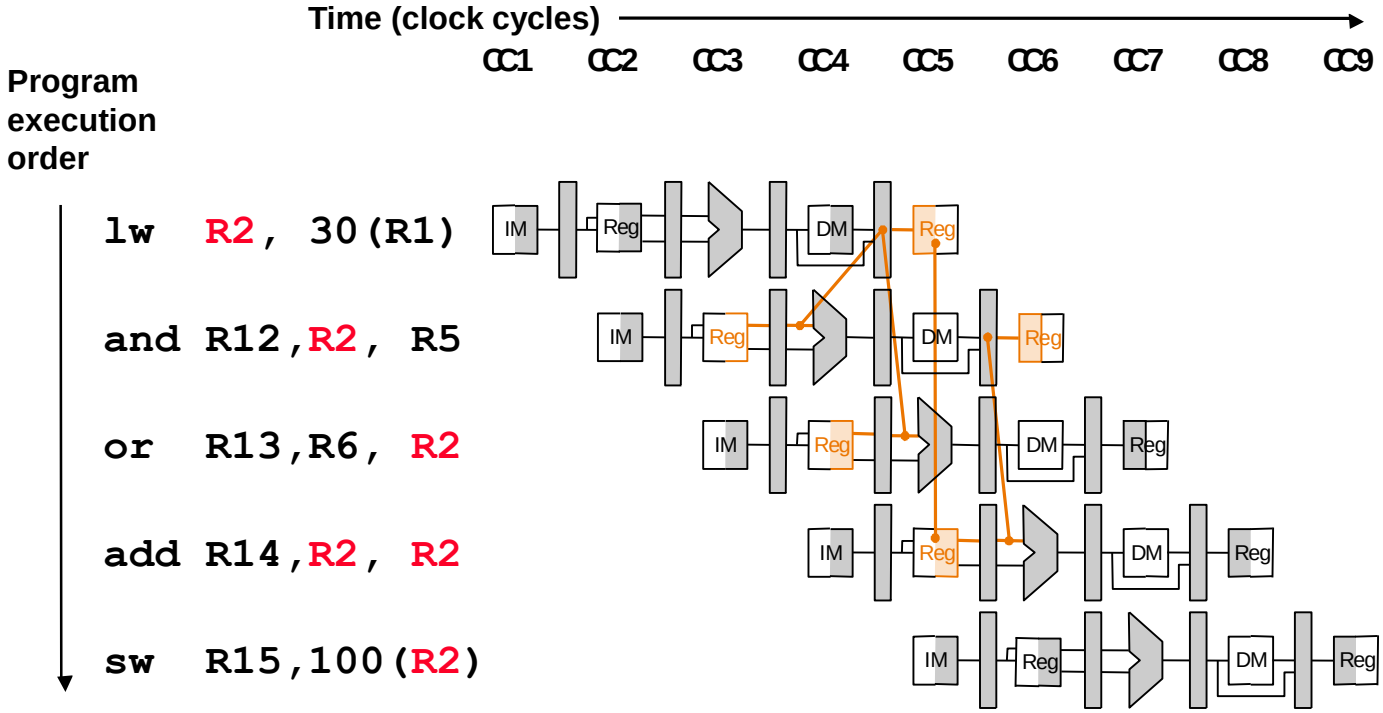
Forwarding Hardware

- *Added 2 mux units before ALU*
- *Each mux gets 3 inputs, from:*
  1. *Prev stage (ID/EX)*
  2. *Next stage (EX/MEM)*
  3. *The one after (MEM/WB)*
- *Forward unit tells the 2 mux units which input to use*



Can't always forward (stall inevitable)

- **“load” op can cause “un-forwardable” hazards**
  - load value to R
  - In the next instruction, use R as input



⇒ **A bigger problem in longer pipelines**

Pipeline Hazards:

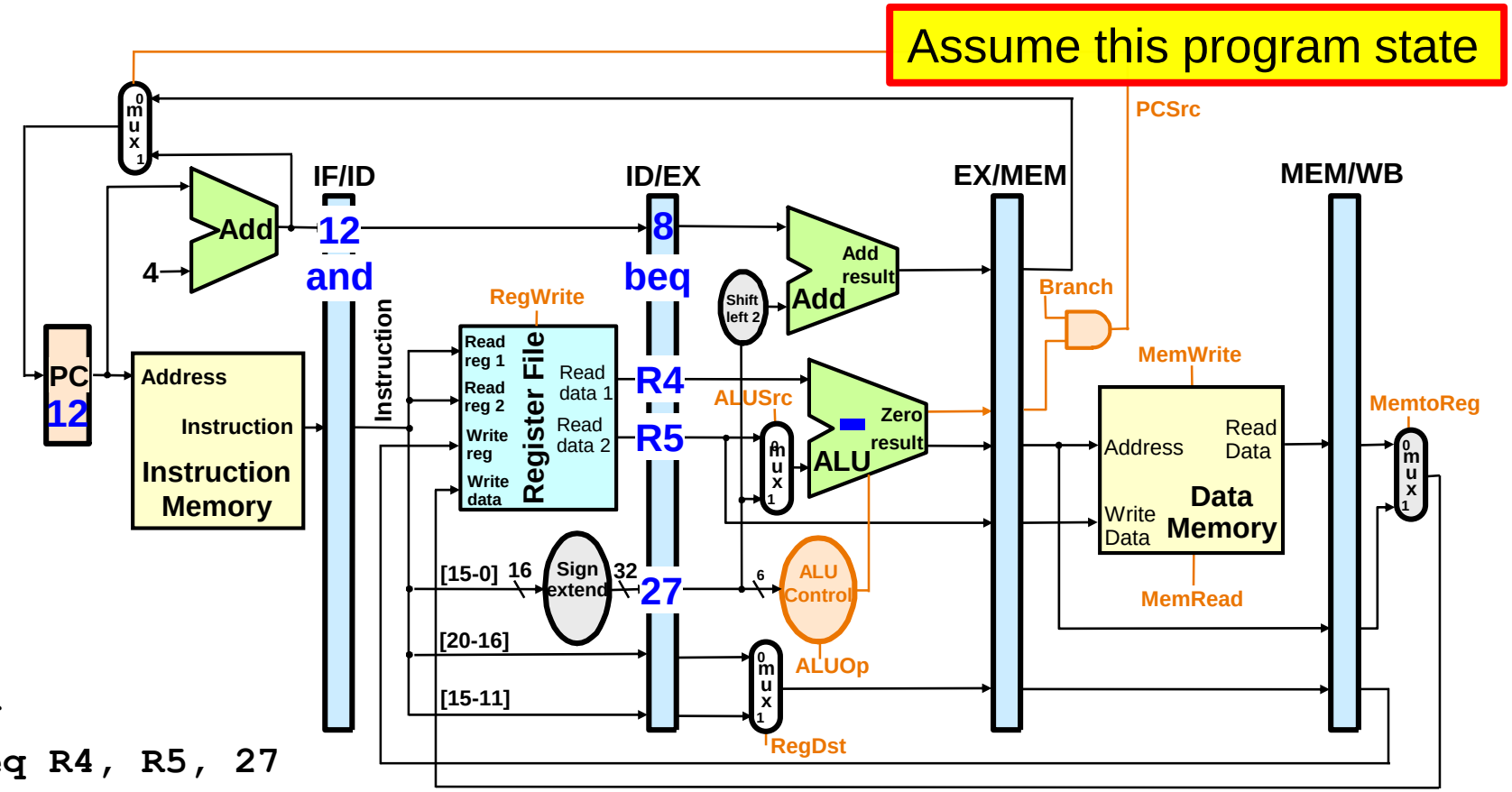
3. Control Hazards

Branch, but where?

- The decision to branch happens deep within the pipeline
- Likewise, the target of the branch becomes known deep within the pipeline
- How does this affect the pipeline logic?
- For example…

Executing a BEQ Instruction (i)

BEQ R4, R5, 27 → if (R4-R5=0) then PC ← PC+4+SignExt(27)*4 ;
else PC ← PC+4



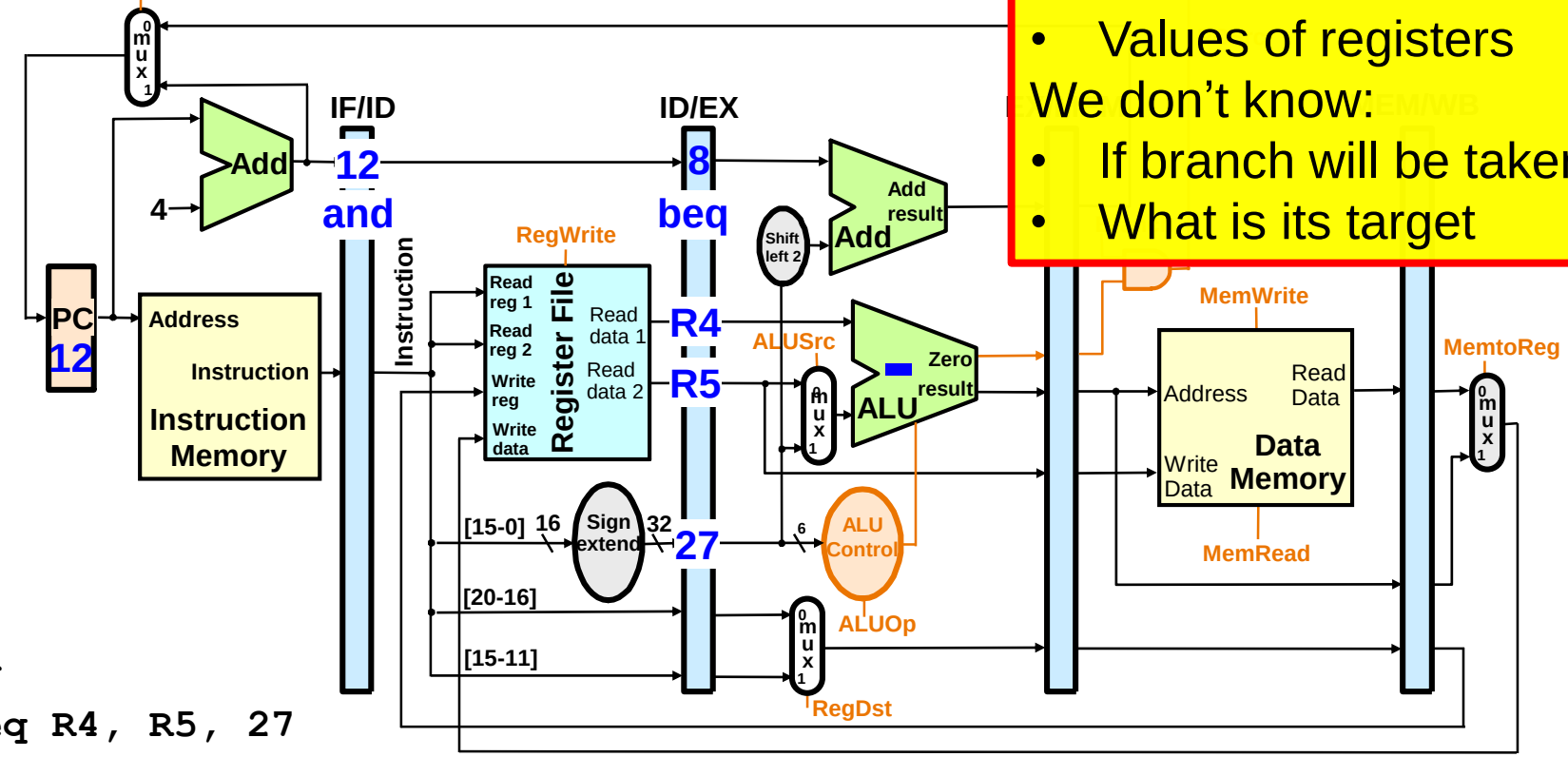

```
 0 or
 4 beq R4, R5, 27
 8 and
12 sw
16 sub
```

Executing a BEQ Instruction (i)

BEQ R4, R5, 27 → if (R4-R5=0) then PC ← PC+4+SignExt(27)*4 ;
else PC ← PC+4

We know:
- Values of registers

We don't know:
- If branch will be taken
- What is its target



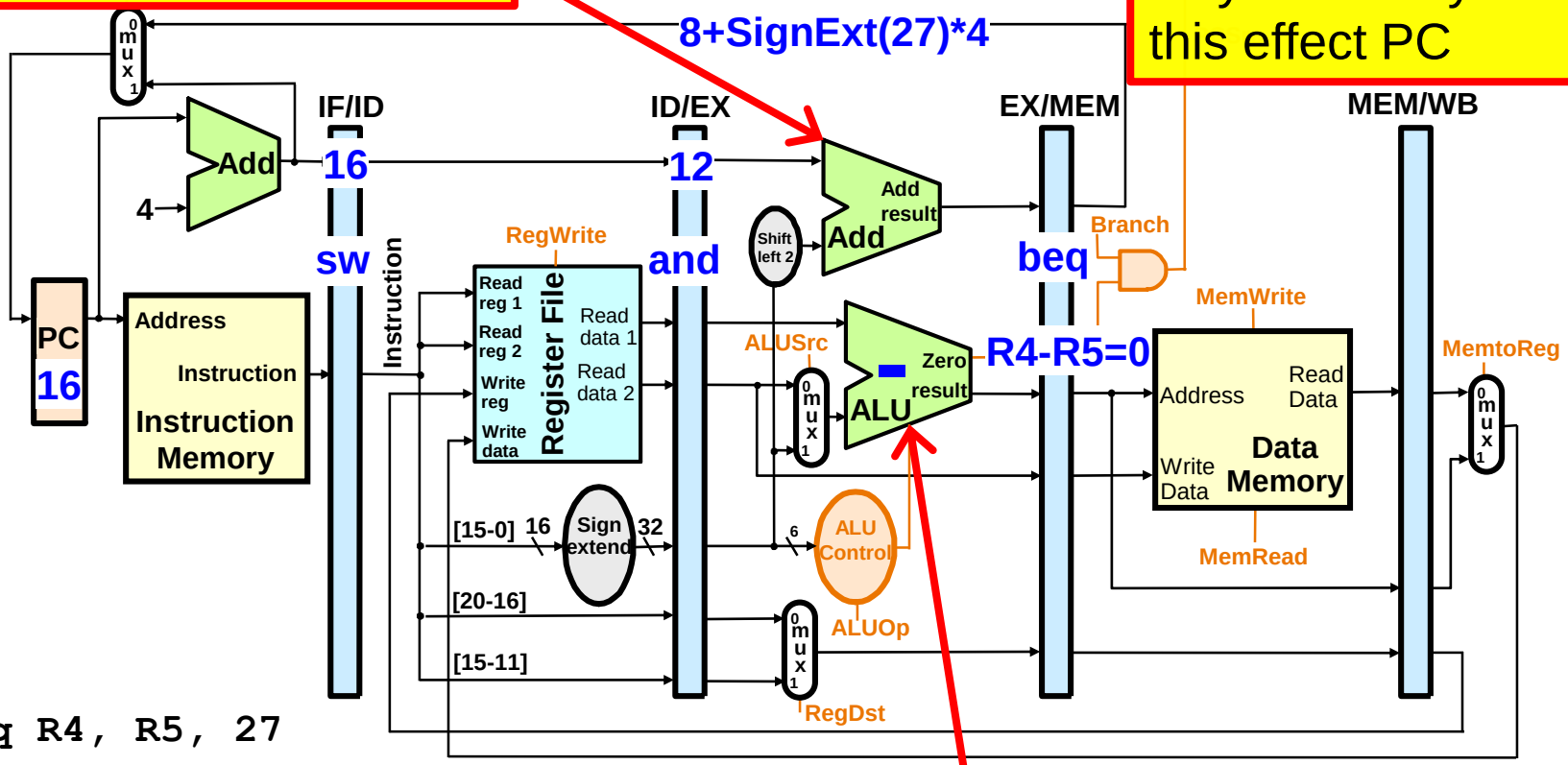

```
 0 or
 4 beq R4, R5, 27
 8 and
12 sw
16 sub
```

Executing a BEQ Instruction (ii)

BEQ R4, R5, 27 → if (R4-R5=0) then PC ← PC+4+SignExt(27)*4 ;
else PC ← PC+4

Calculate branch target

…Now we know, but only in next cycle will this effect PC

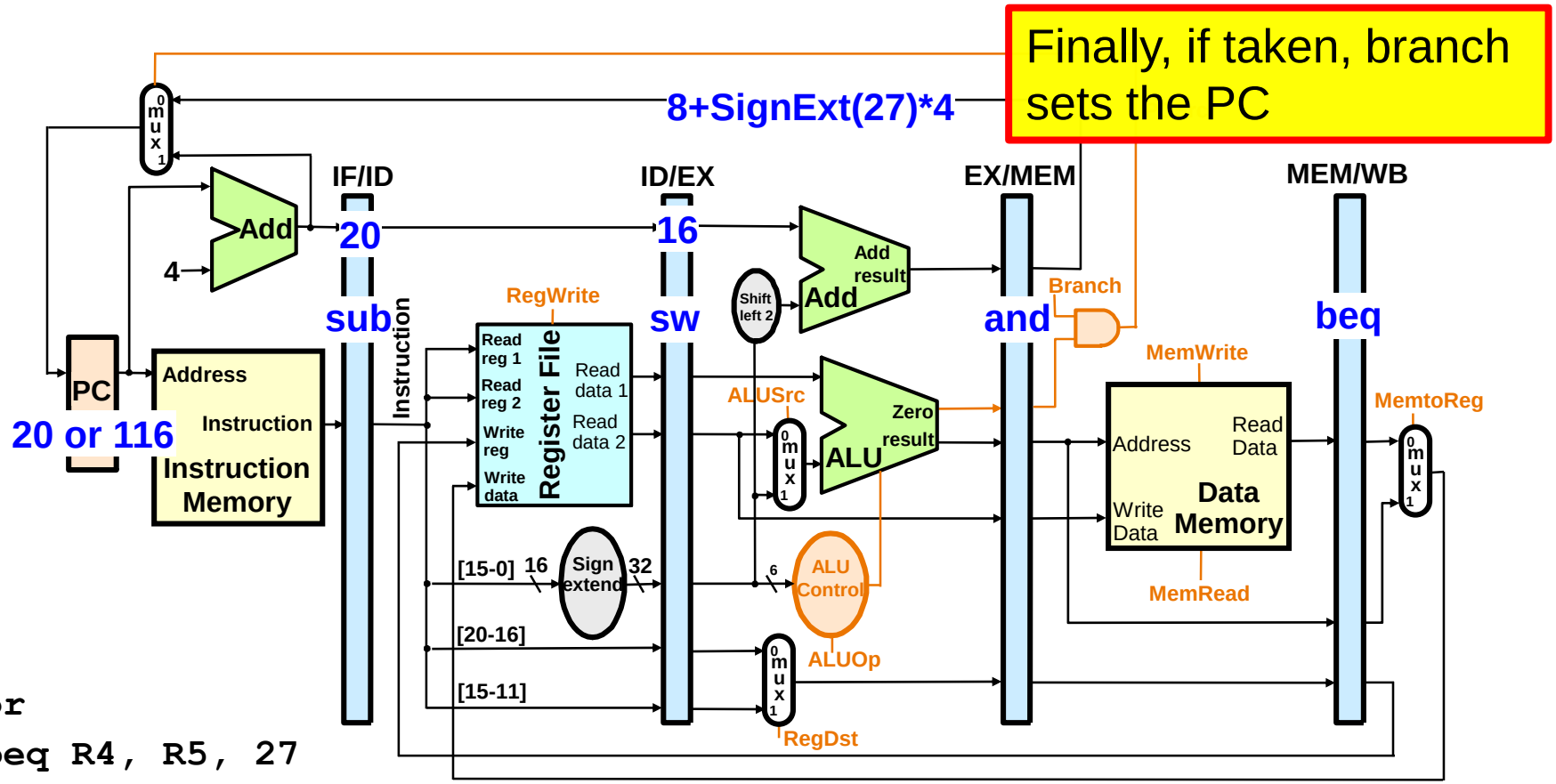



```
 0 or
 4 beq R4, R5, 27
 8 and
12 sw
16 sub
```

Calculate branch condition = compute R4-R5 & compare to 0

Executing a BEQ Instruction (iii)

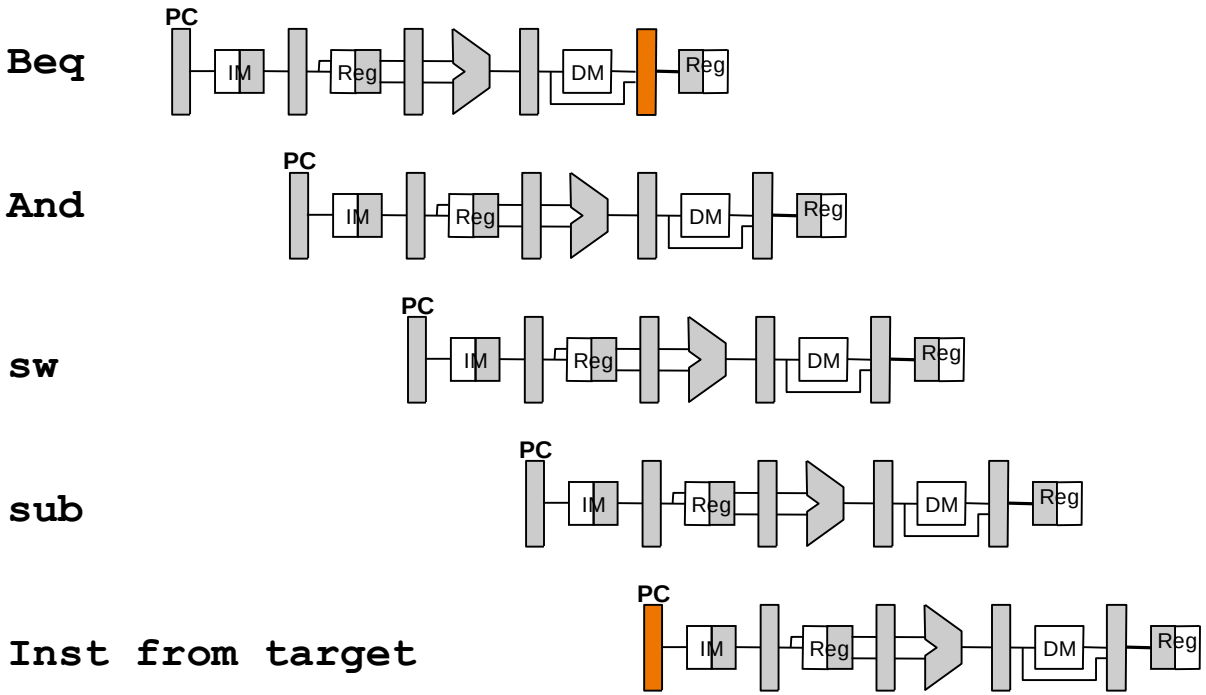
BEQ R4, R5, 27 → if (R4-R5=0) then PC ← PC+4+SignExt(27)*4 ;
else PC ← PC+4




```
 0 or
 4 beq R4, R5, 27
 8 and
12 sw
16 sub
```

Control Hazard on Branches

Outcome:
The 3 instructions following the branch are in the pipeline even if branch is taken!



Stall

- **Easiest solution:**
  - Stall pipe when branch encountered until resolved
- **But there's a prices. Assume:**
  - CPI = 1
  - 20% of instructions are branches (realistic)
  - Stall 3 cycles on every branch (extra 3 cycles for each branch)
- **Then the price is:**
  - $CPI_{new} = 1 + 0.2 \times 3 = 1.6$ // 1 = all instr., including branch
  - [$CPI_{new} = CPI_{Ideal}$ + avg. stall cycles / instr.]
- **Namely:**
  - IPC drops from 1 to 1/1.6
  - We lose ~37% of the performance!

Traps, Exceptions and Interrupts

- **Indication of events that require a higher authority to intervene (i.e. the operating system)**
- **Atomically changes the protection mode and <u>branches</u> to OS**
  - Protection mode determines what the running is allowed to do (access devices, memory, etc).

- **Traps:** *Synchronous*
  - The program asks for OS services (e.g. access a device)
- **Exceptions:** *Synchronous*
  - The program did something bad (divide-by-zero; prot. violation)
- **Interrupts**: *Asynchronous*
  - An external device needs OS attention (finished an operation)
- Can these be handled like regular branches?

Branch Prediction and Speculative Execution

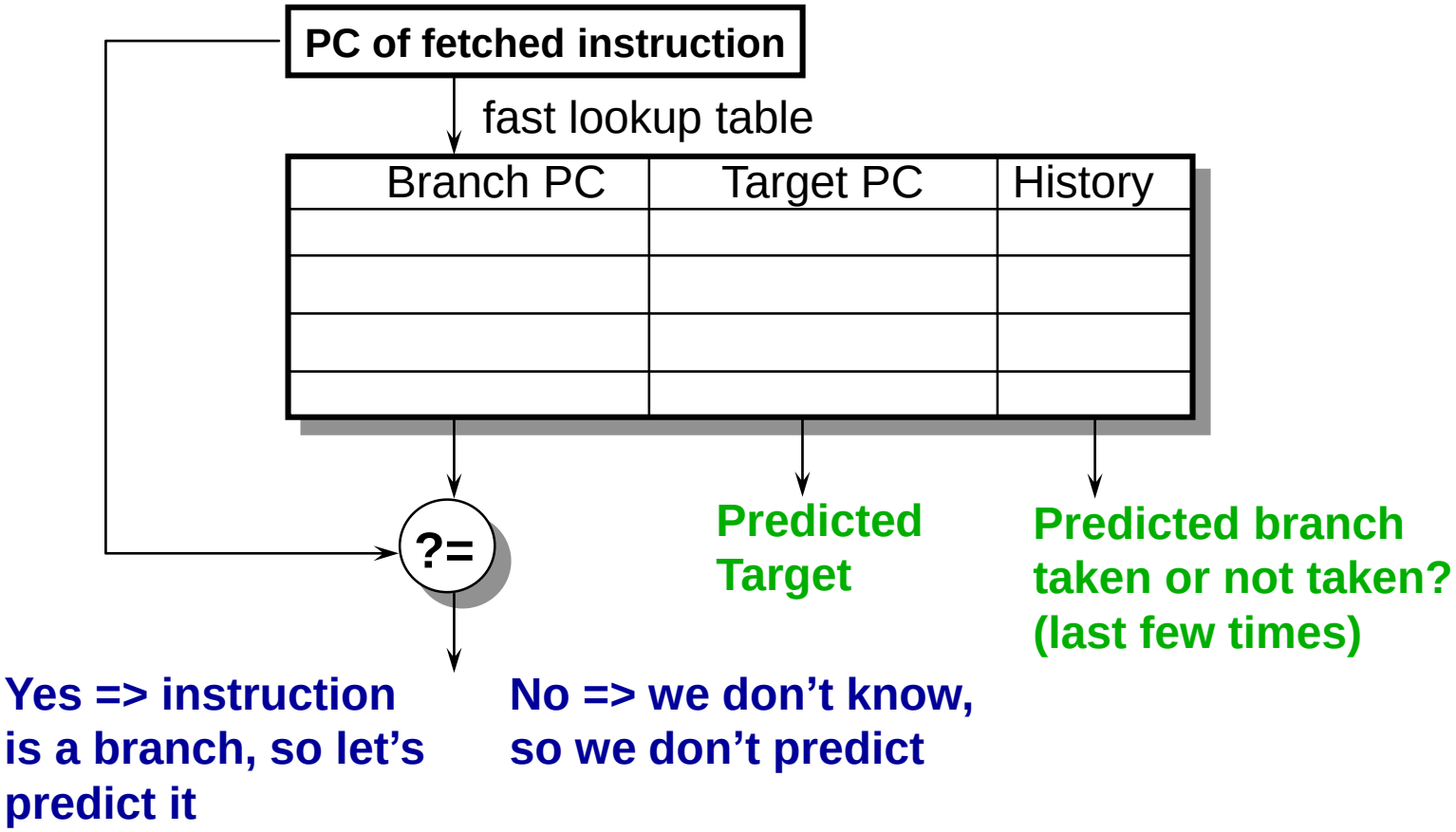
Static prediction: *branch not taken*

- **Execute instructions from the fall-through (not-taken) path**
  - As if there is no branch
  - If the branch is not-taken (~50%), no penalty is paid
- **If branch actually taken**
  - Flush the fall-through path instructions before they change the machine state (memory / registers)
  - Fetch the instructions from the correct (taken) path
- **Assuming ~50% branches not taken on average**
  - CPI new = 1 + (0.2 × 0.5) × 3 = 1.3
  - 23% slowdown instead of 37%
  - What happens in longer pipelines?

Dynamic branch prediction

- **Branch prediction is a key impediment to performance**
  - Modern processors employ complex branch predictors
  - Often achieve < 3% misprediction rate
- **Given an instruction, we need to predict**
  - Is it a branch?
  - Branch taken?
  - Target address?
- **To avoid stalling**
  - Prediction needed at end of ‘fetch’
  - Before we even now what the instruction is…
- **A simple mechanism: Branch Target Buffer (BTB)**

BTB – the idea



(Works in a straightforward manner only for direct branches, otherwise target PC changes)

How it works in a nutshell

- **Until proven otherwise, assume branches are not taken**
  - Fall through instructions (assume branch has no effect)
- **Upon the first time a branch is taken**
  - Pay the price (in terms of stalls), but
  - Save the details of the branch in the BTB
    (= PC, target PC, and whether or not branch was taken)
- **While fetching, HW checks *in parallel***
  - Whether PC is in BTB
- **If found, make a prediction**
  - Taken? Address?
- **Upon misprediction**
  - Flush (throw out) pipeline content & start over from right PC

Prediction steps

1. **Allocate**
   - Insert instruction to BTB once identified as *taken* branch
   - Do we want to insert *not-taken* branches?
     - Option: Implicitly predict they'd continue not to be taken
   - Insert both conditional & unconditional
     - To identify, and to save arithmetic
2. **Predict**
   - BTB lookup done in parallel to PC-lookup, providing:
     - Indication whether PC is a branch (=> BTB "hit")
     - Branch target
     - Branch direction (forward or backward in program)
     - Branch type (conditional or not)
3. **Update (when branch taken & its outcome becomes known)**
   - Branch target, history (taken or not)

Misprediction

- **Occurs when**
  - Predict = not taken, reality = taken
  - Predict = taken, reality = not taken
  - Branch taken as predicted, but wrong target (indirect, as in the jmp register)

- **Must flush pipeline**
  - Reset pipeline registers (similar to turning all into NOPs)
    - Commonly, other flush methods are easier to implement
  - Set the PC to the correct path
  - Start fetching instruction from correct path

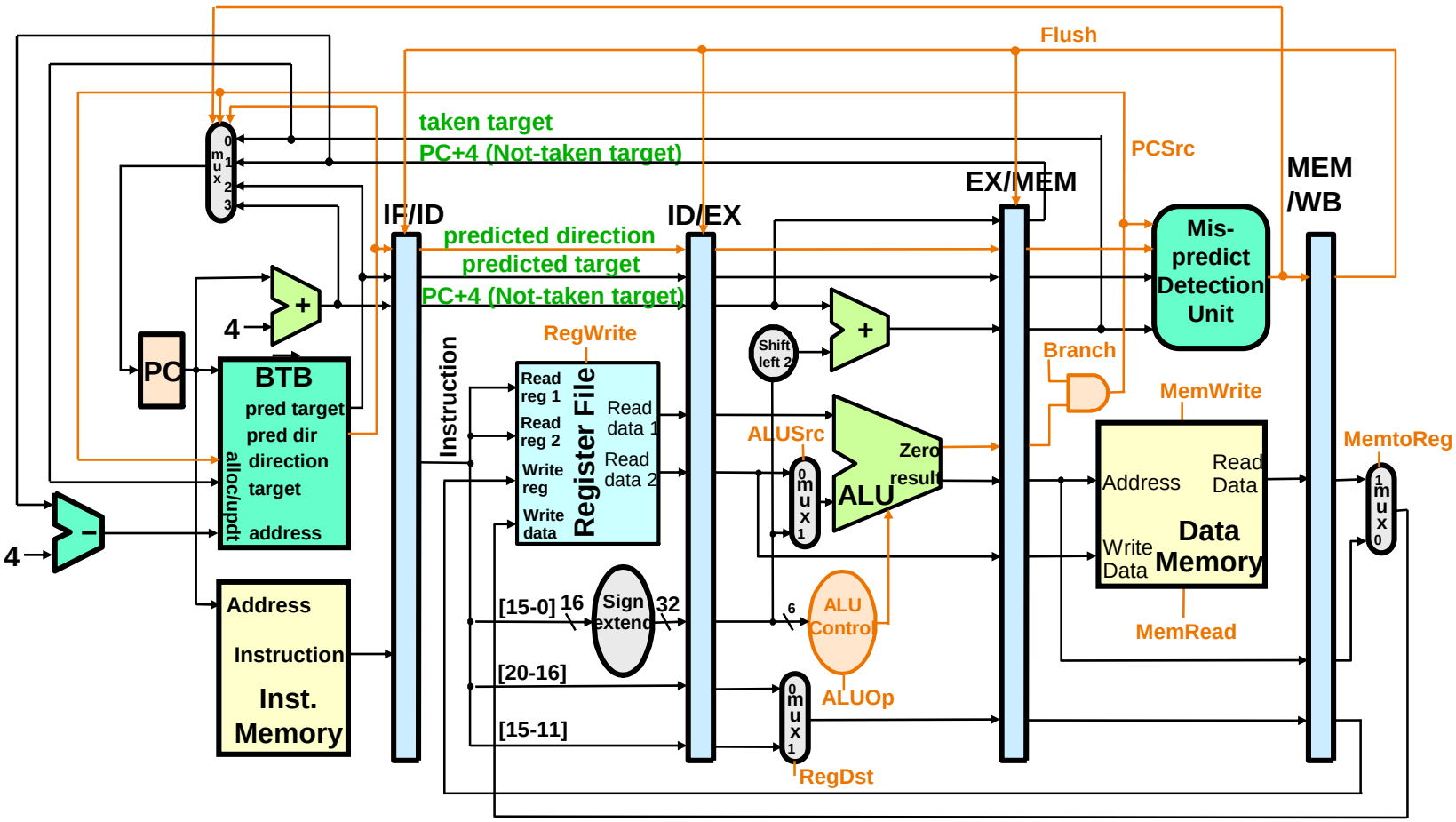
CPI

- **Assuming a fraction of *p* correct predictions**
  - CPI_new = 1 + (0.2 × (1-*p*)) × 3
- **Example, *p*=0.7**
  - CPI_new = 1 + (0.2 × 0.3) × 3 = 1.18
- **Example, *p*=0.98**
  - CPI_new = 1 + (0.2 × 0.02) × 3 = 1.012
  - (But this is a simplistic model; in reality the price can sometimes be much higher.)

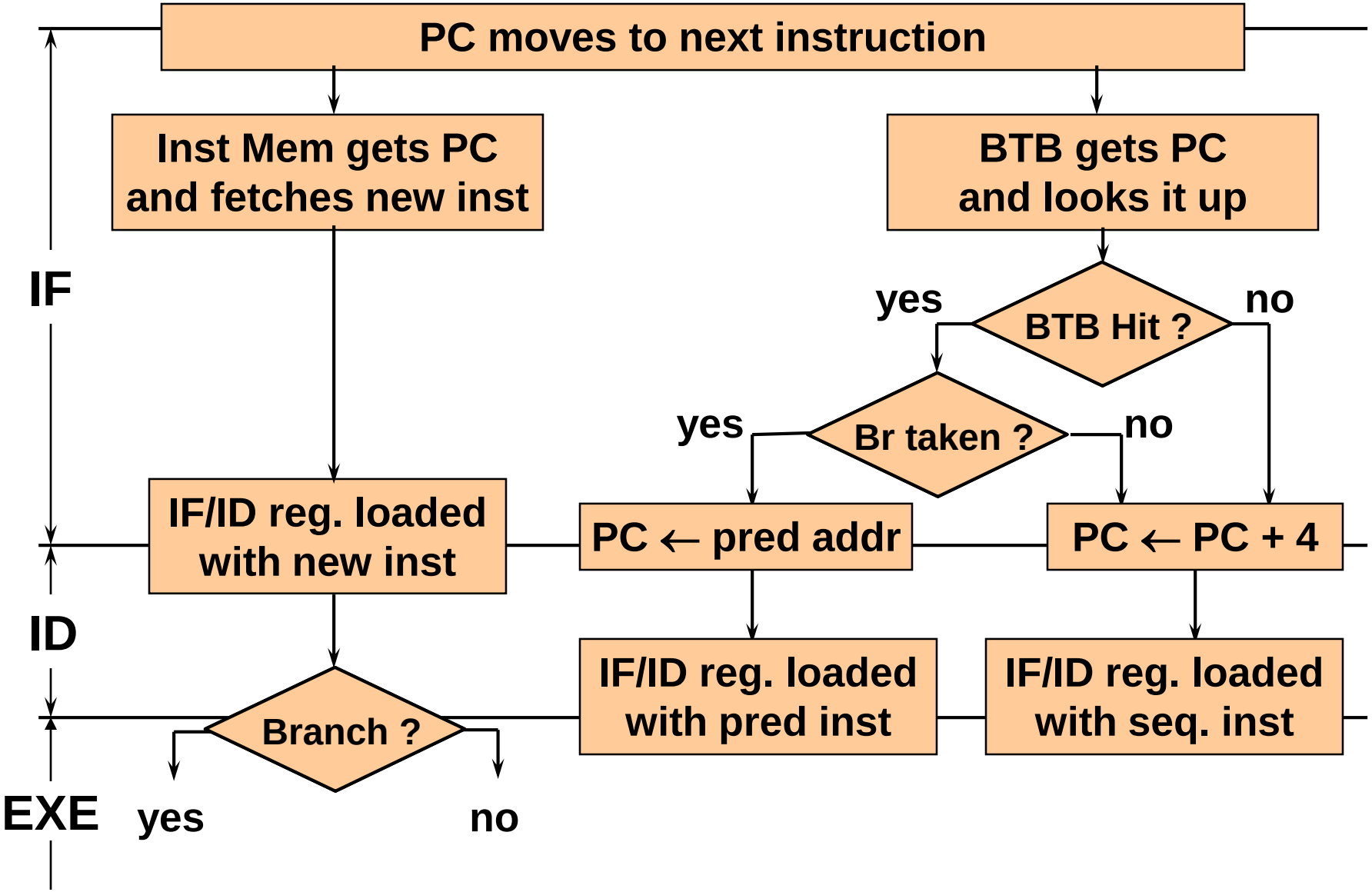
History & prediction algorithm

- **“Always backward” prediction**
  - Works for long loops
- **Some branches exhibit “locality”**
  - Typically behave as the last time they were invoked
  - Typically depend on their previous outcome (& it alone)
- **Can save a history window**
  - What happened last time, and before that, and before…
  - The bigger the window, the greater the complexity
- **Some branches regularly alternate between taken & untaken**
  - Taken, then untaken, then taken, …
  - Need only one history bit to identify this
- **Some branches are correlated with previous branches**
  - Those that lead to them

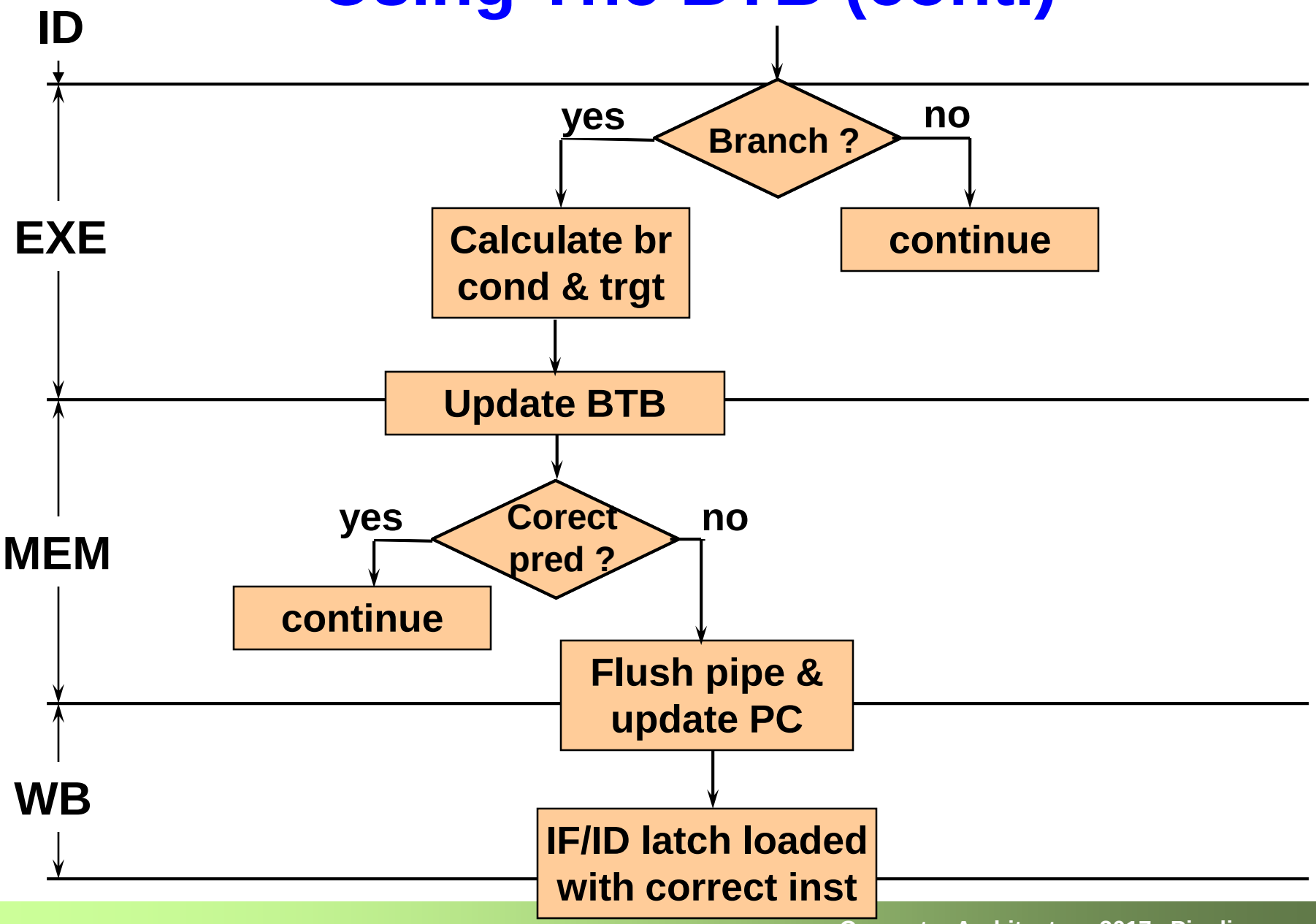
Adding a BTB to the Pipeline



Using The BTB



Using The BTB (cont.)



Prediction algorithm

- **Can do an entire course on this issue**
  - Still actively researched
- **As noted, modern predictors can often achieve misprediction < 3%**
- **Still, it has been shown that these 3% can sometimes significantly worsen performance**
  - A real problem in *out-of-order* pipelines
- **We did not talk about the issue of indirect branches**
  - As in virtual function calls (object oriented)
  - Where the branch target is written in memory, elsewhere