Computer organization and architecture

Lesson 16

Pipelined processor

We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages:

1) Fetch
2) Decode
3) Execute
4) Memory
5) Writeback

They are similar to the five steps that the multicycle processor used to perform `LDR`

Five instructions can execute simultaneously, one in each stage.

Fetch

The processor reads the instruction from instruction memory.

Decode

The processor reads the source operands from the register file and decodes the instruction to produce the control signals.

Execute

The processor performs a computation with the ALU.

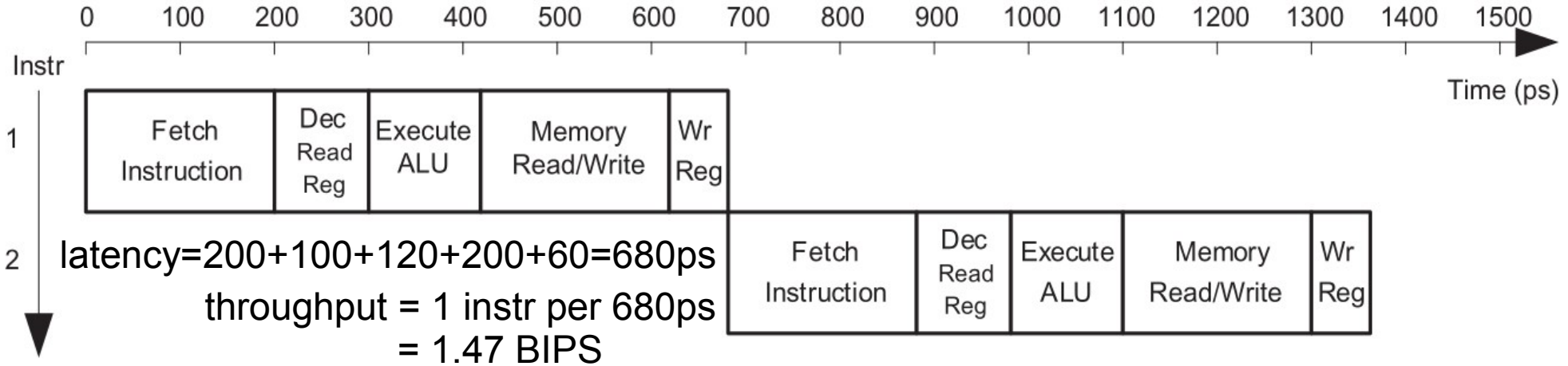
Memory The processor reads or writes data memory.

Writeback

The processor writes the result to the register file, when applicable.

Timing diagrams

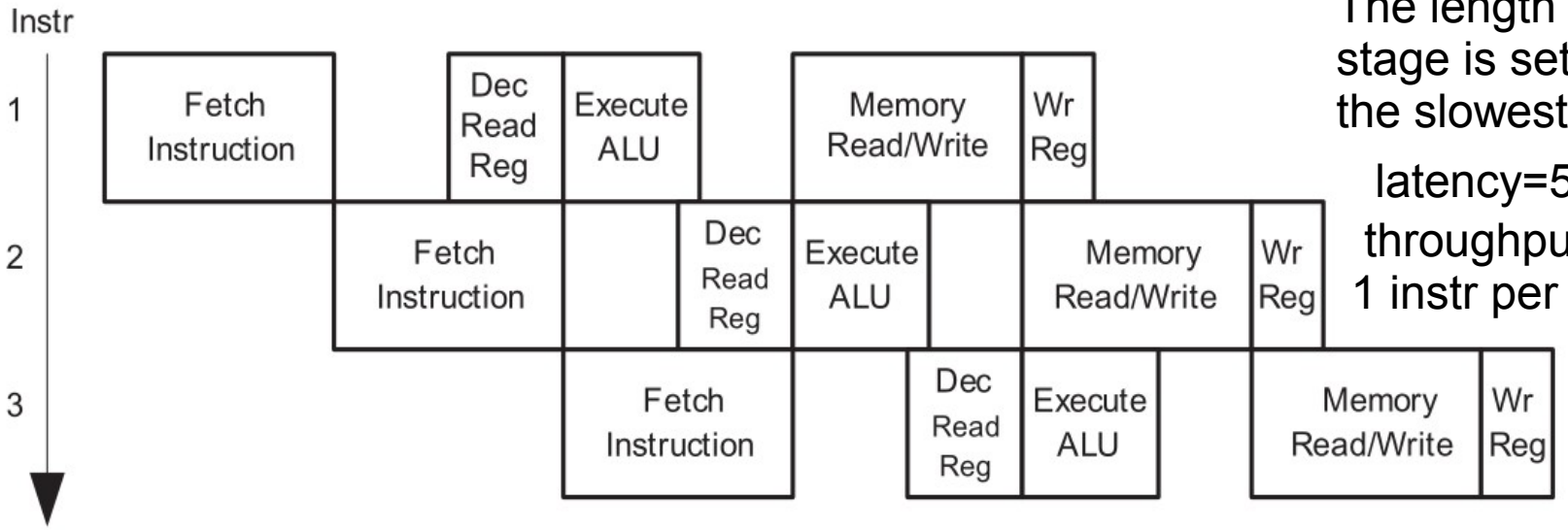
single-cycle processor



latency=200+100+120+200+60=680ps

throughput = 1 instr per 680ps
= 1.47 BIPS

pipelined processor



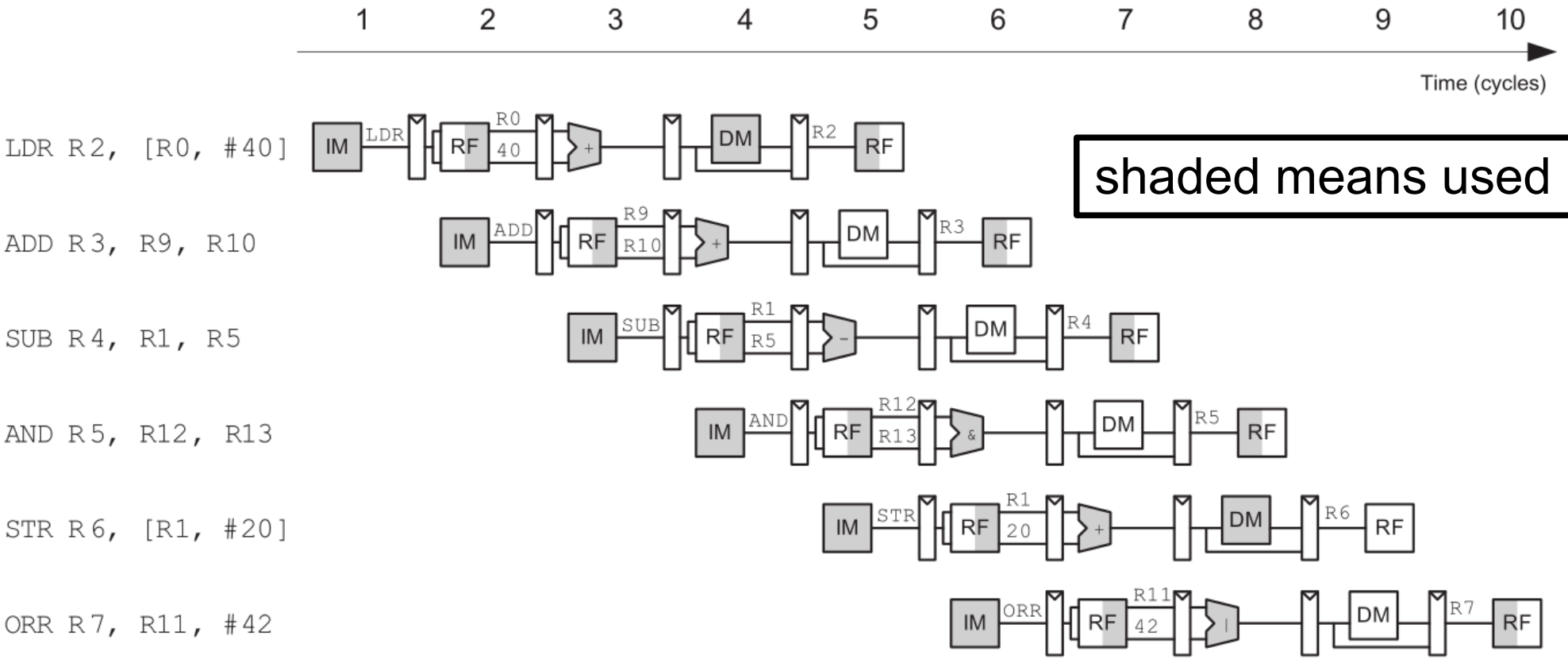
The length of a pipeline stage is set at 200 ps by the slowest stage.

latency=5×200=1000ps

throughput =
1 instr per 200 ps = 5 BIPS

Pipelining increases latency but improves throughput.

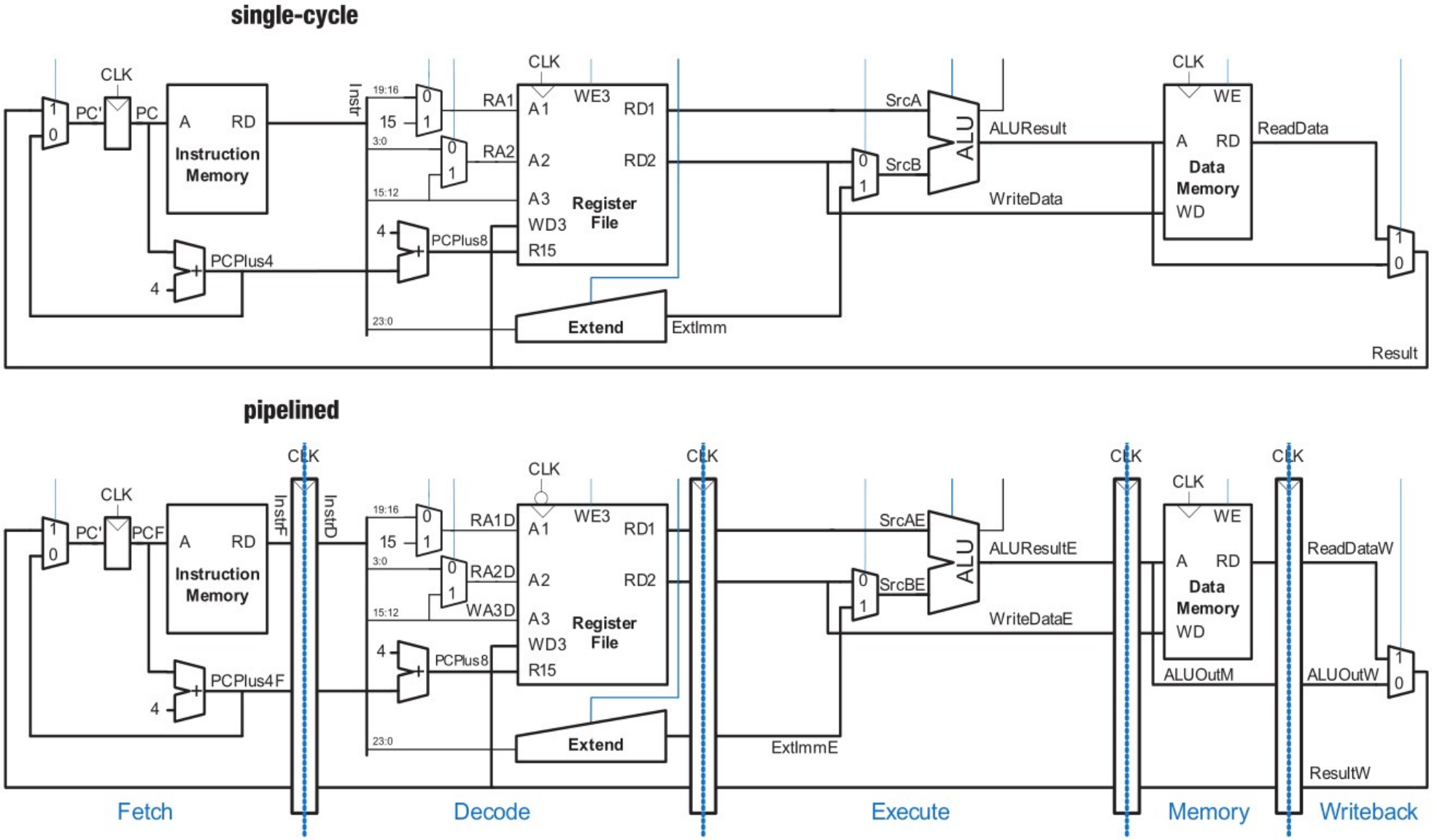
Abstract view of pipeline in operation



In the pipelined processor, the register file is written in the 1st part of a cycle and read in the 2nd part.

This way, data can be written and read back within a single cycle.

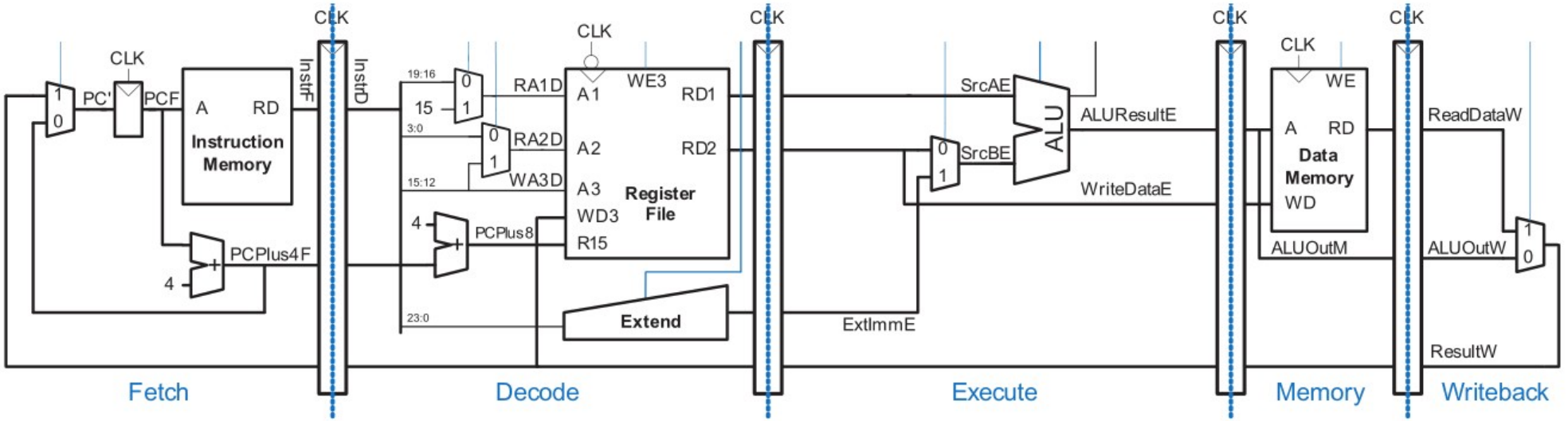
The pipelined datapath: chop the single-cycle datapath into five stages and separate them by pipeline registers.



Signals are given a suffix (F, D, E, M, W) to indicate the stage

The register file is read in the Decode stage and written in the Writeback stage.

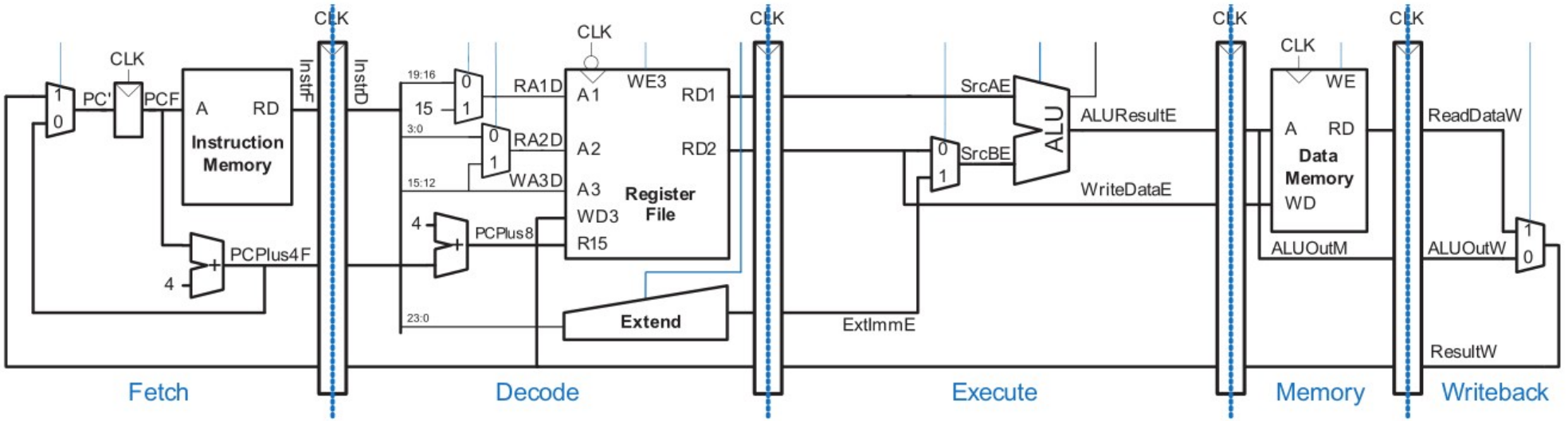
It is drawn in the Decode stage, but the write address and data come from the Writeback stage.



The register file in the pipelined processor writes on the falling edge of CLK so that it can write a result in the first half of a cycle and read that result in the second half of the cycle for use in a subsequent instruction.

All signals associated with a particular instruction must advance through the pipeline in unison.

This datapath has an error related to this issue.

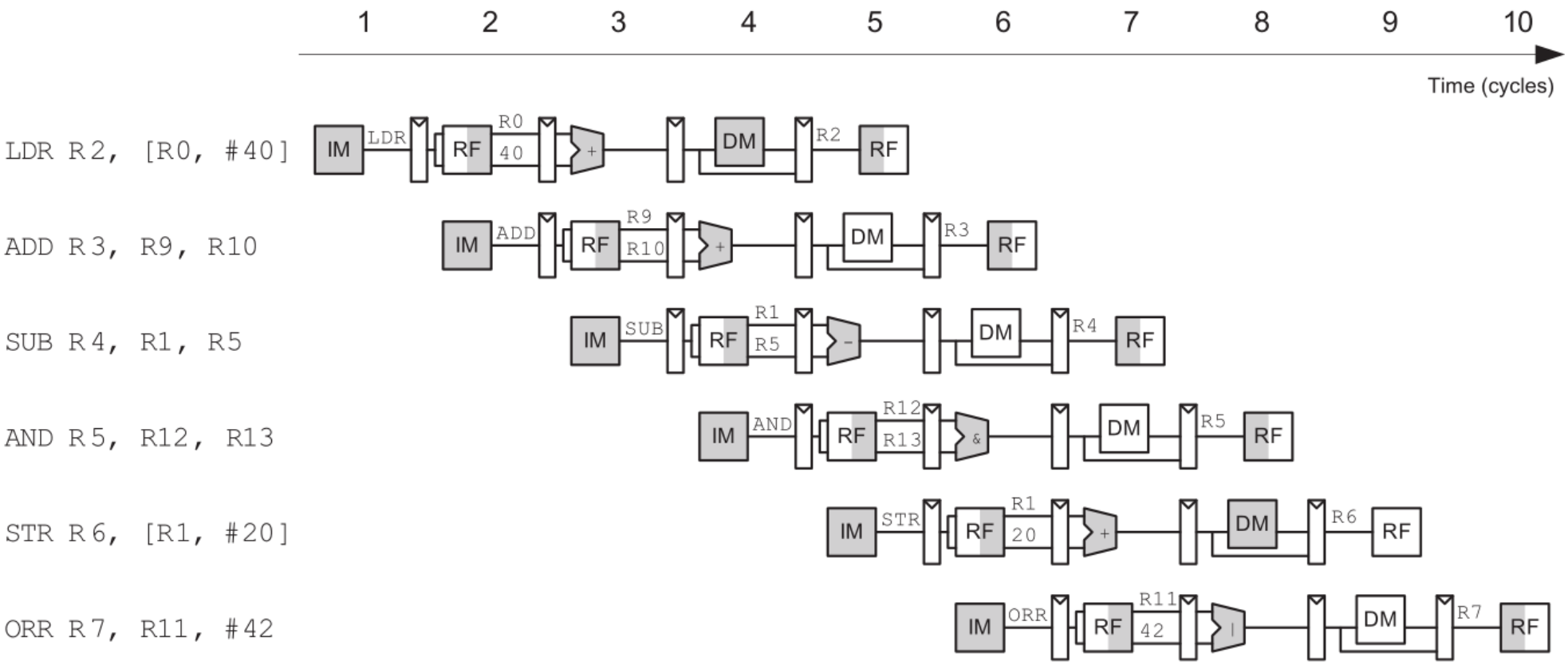


The error is in the register file write logic, which should operate in the Writeback stage.

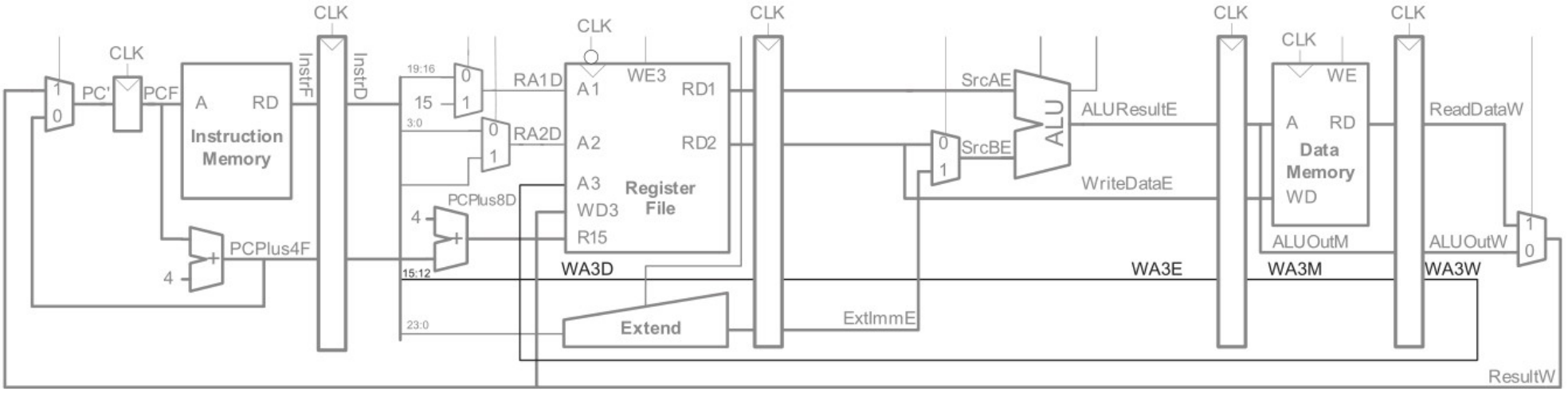
The data value comes from ResultW, a W stage signal.

But the write address comes from $InstrD_{15:12}$ (WA3D), which is a Decode stage signal.

Example: during cycle 5, the result of the LDR instruction would be incorrectly written to R5 rather than R2.

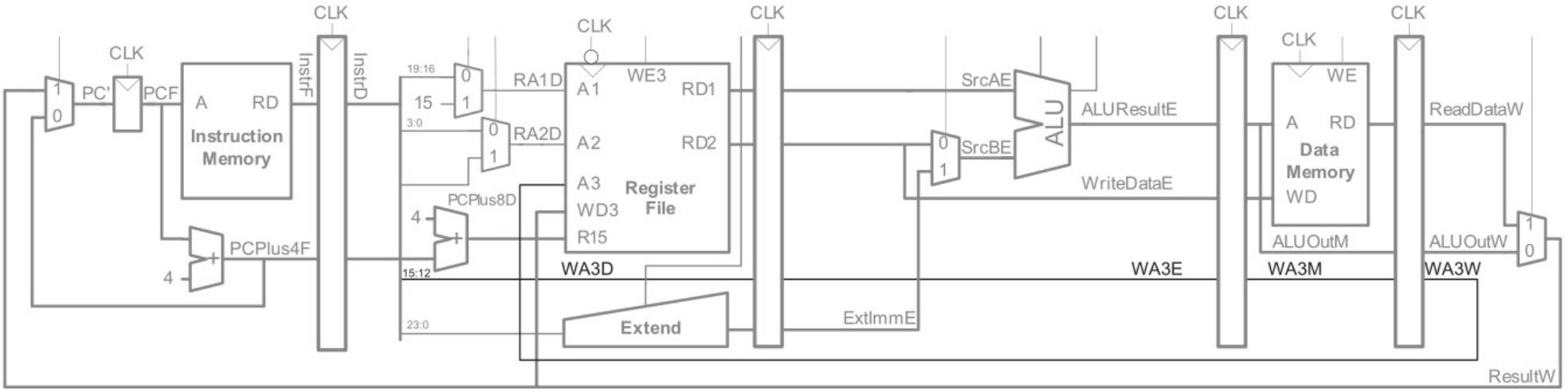


Corrected pipelined datapath



The WA3 signal is now pipelined along through the Execution, Memory, and Writeback stages, so it remains in sync with the rest of the instruction.

WA3W and ResultW are fed back together to the register file in the Writeback stage.



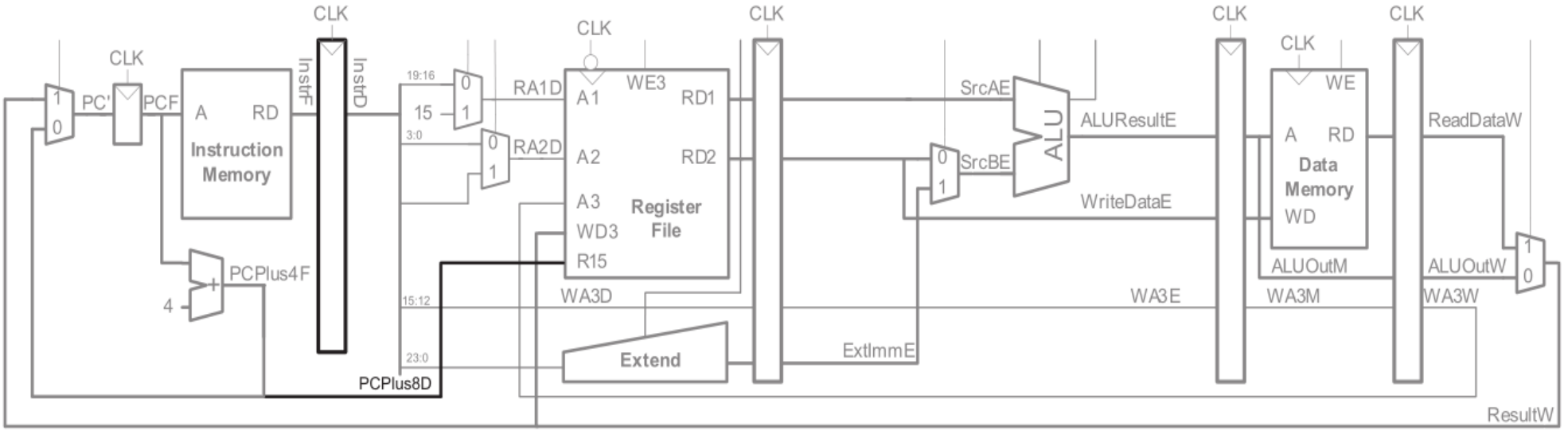
Each time the program counter is incremented, PCPlus4F is simultaneously written to the PC and the pipeline register between the F and D stages.

On the subsequent cycle, the value in both of these registers is incremented by 4 again.

Thus, PCPlus4F for the instruction in the Fetch stage is logically equivalent to PCPlus8D for the instruction in the Decode stage.

Sending this signal ahead saves the pipeline register and second adder.

Optimized PC logic eliminating a register and adder:



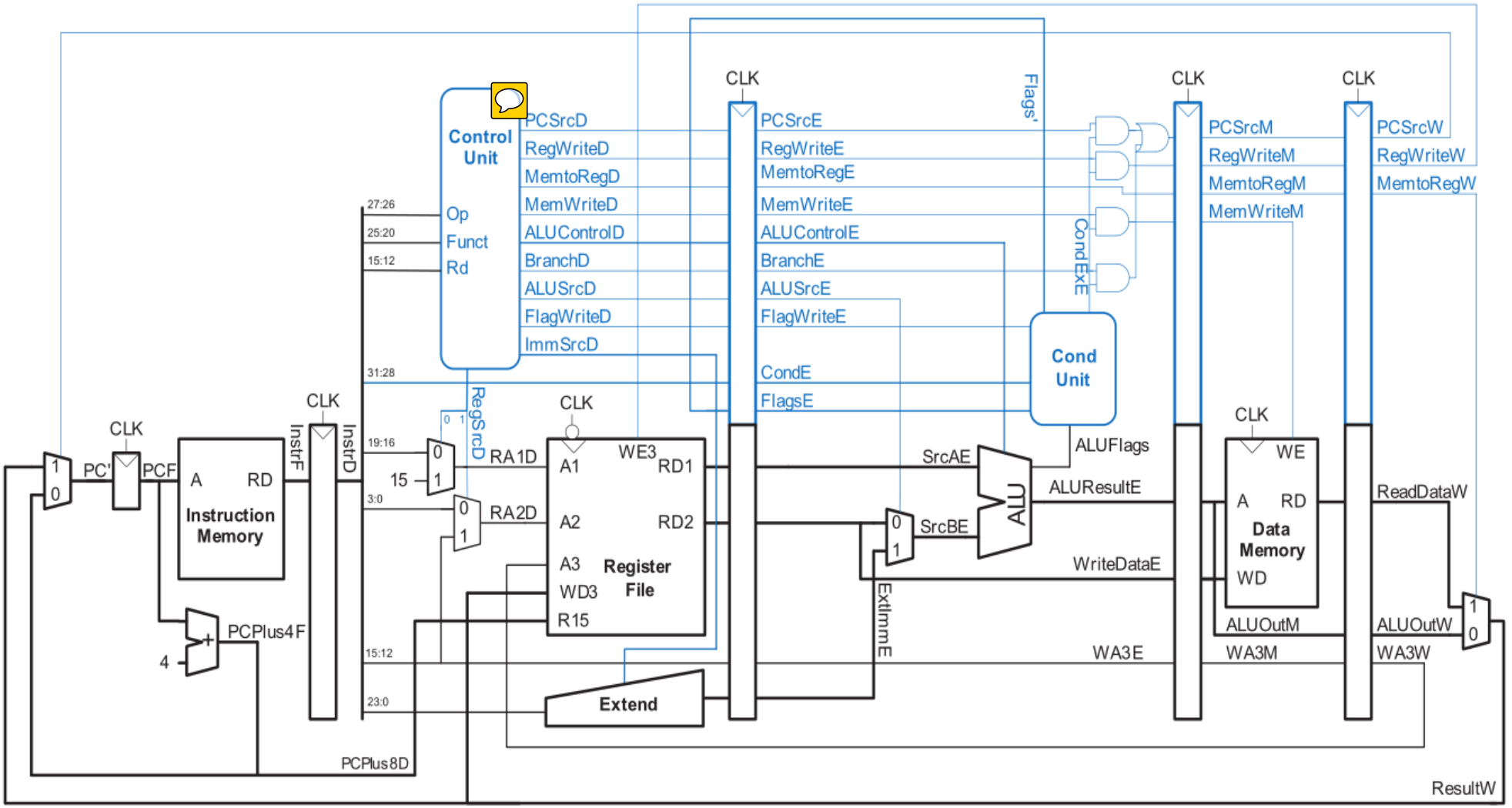
Pipelined control

The pipelined processor takes the same control signals as the single-cycle processor and therefore uses the same control unit.

The control unit examines the Op and Funct fields of the instruction in the Decode stage to produce the control signals.

These control signals must be pipelined along with the data so that they remain synchronized with the instruction. The control unit also examines the Rd field to handle writes to R15 (PC).

Pipelined processor with control

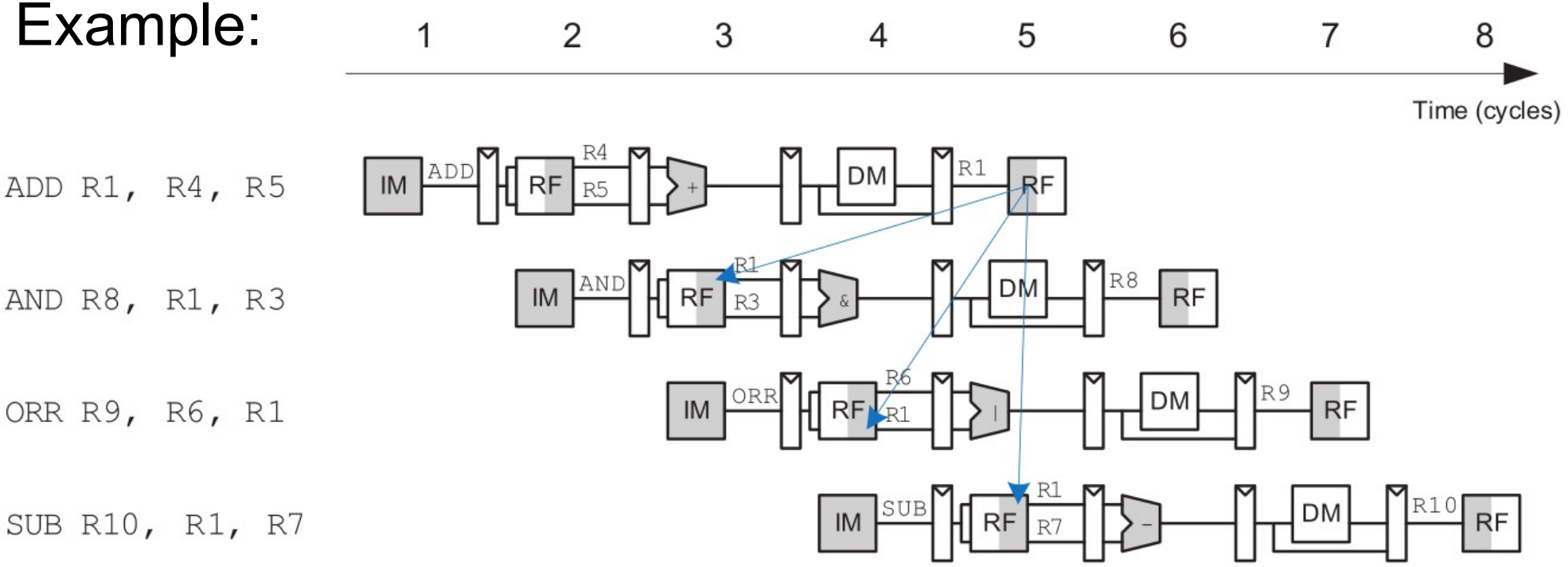


RegWrite must be pipelined into the W stage before it feeds back to the register file, just as WA3 is pipelined.

Hazards

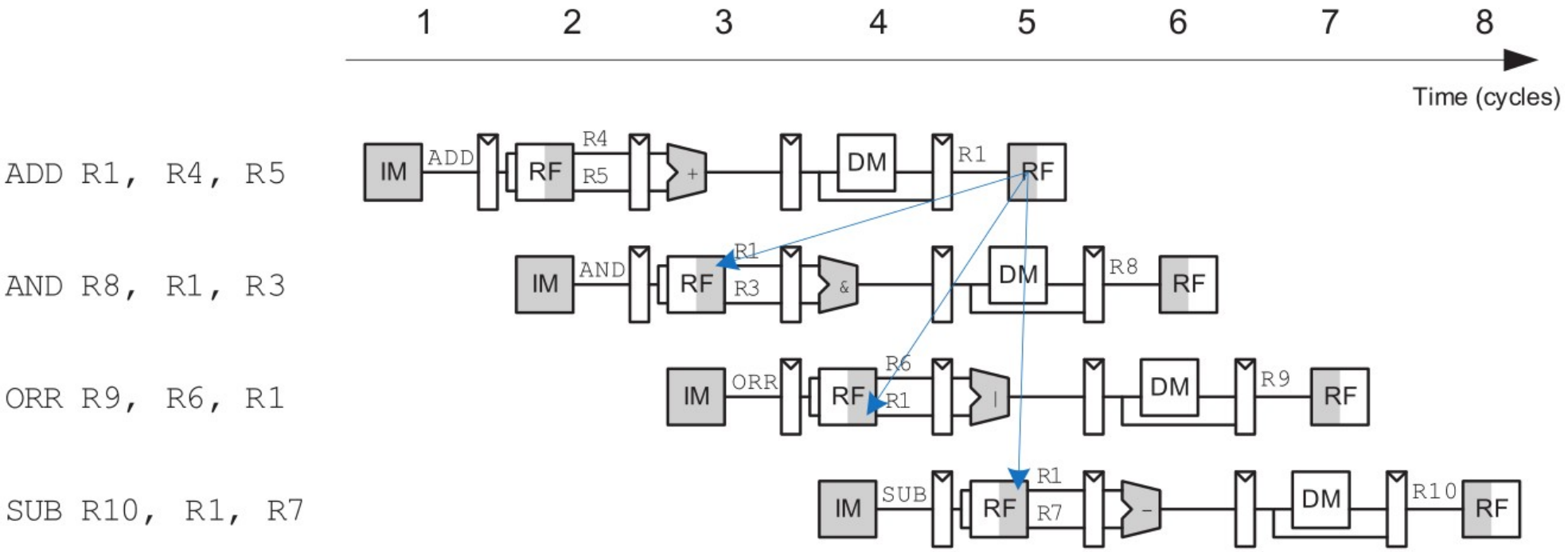
When one instruction is dependent on the results of another that has not yet completed, a **hazard** occurs.

Example:



One instruction writes a register (R1) and subsequent instructions read this register.

This is called a **read after write (RAW) hazard**.



The sum from ADD is computed by the ALU in cycle 3

We can **forward** the result from one instruction to the next to resolve the RAW hazard without waiting for the result to appear in the register file.

In other situations we may have to **stall** the pipeline to give time for a result to be produced before the subsequent instruction uses the result.

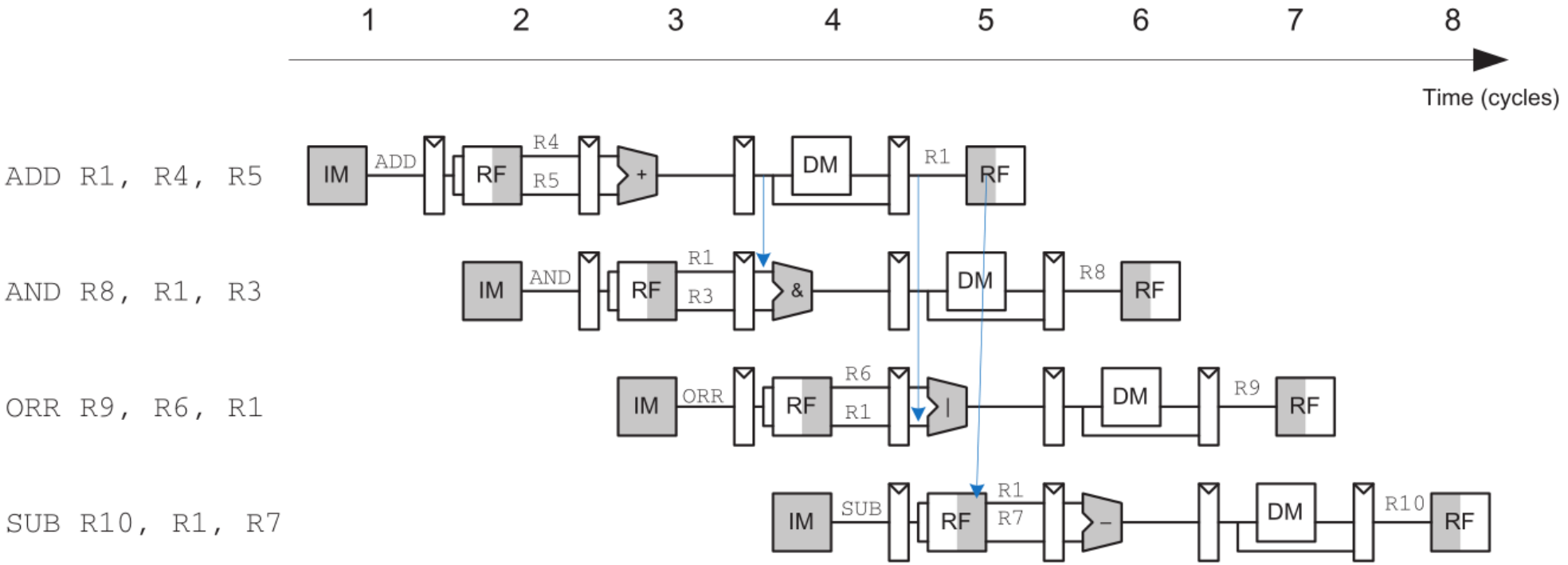
Hazards are classified as data hazards or control hazards.

A **data hazard** occurs when an instruction tries to read a register that has not yet been written back by a previous instruction.

A **control hazard** occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place.

We will enhance the pipelined processor with a **Hazard Unit** that detects hazards and handles them appropriately, so that the processor executes the program correctly.

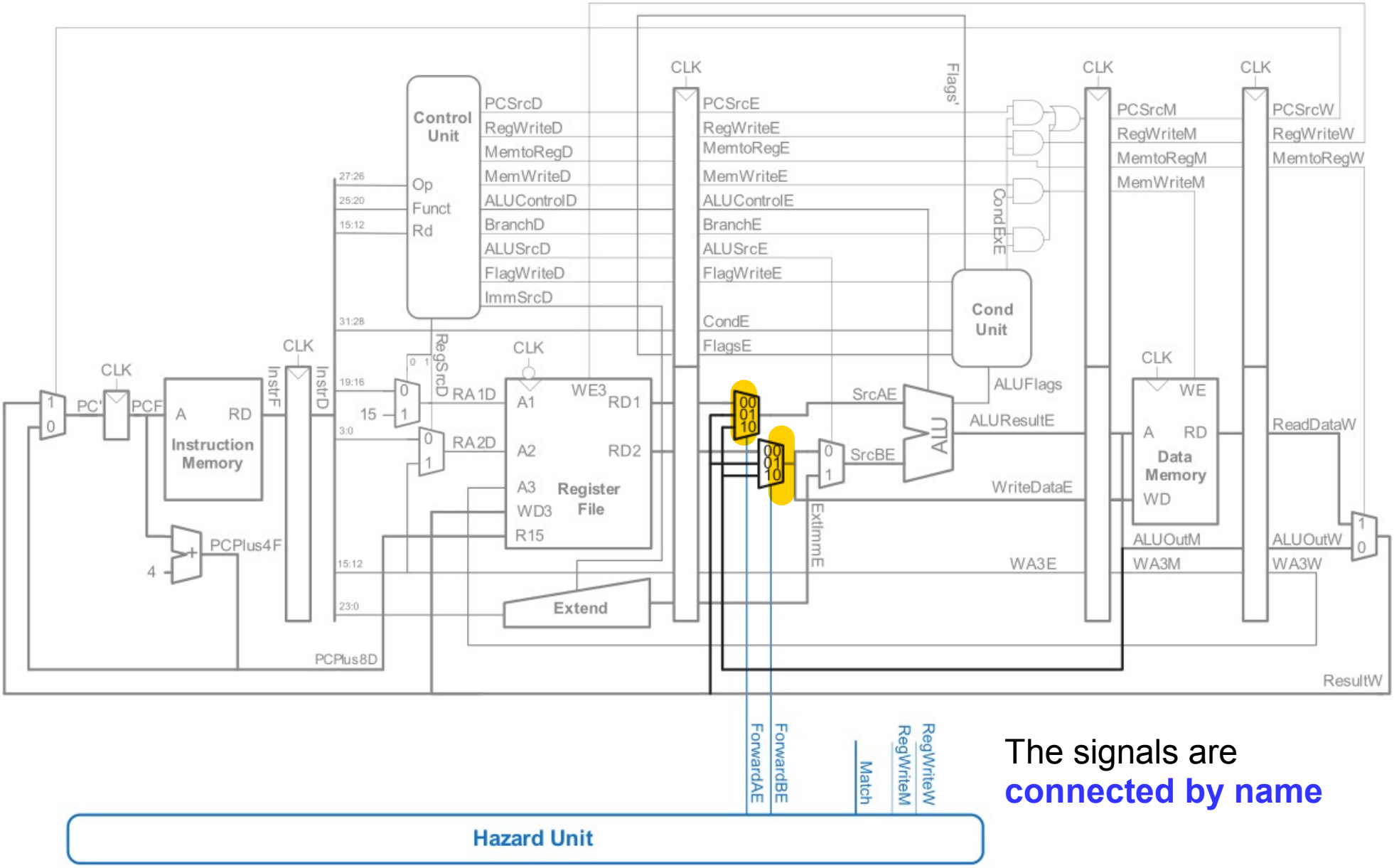
Some data hazards can be solved by **forwarding** (also called **bypassing**)



This requires adding multiplexers in front of the ALU to select the operand from either the register file or the Memory or Writeback stage.

Pipelined processor with forwarding

A **Hazard Unit** and 2 **forwarding multiplexers** are added



The signals are **connected by name**

The Hazard Unit receives 4 match signals from the datapath (abbreviated to Match) that indicate whether the source registers in the Execute stage match the destination registers in the Memory and Execute stages:

```
Match_1E_M = (RA1E == WA3M) | Match_2E_M = (RA2E == WA3M)
Match_1E_W = (RA1E == WA3W) | Match_2E_W = (RA2E == WA3W)
```

The Hazard Unit also receives the RegWrite signals from the Memory and Writeback stages to know whether the destination register will actually be written.

Example: `STR` and `B` do not write results to the register file and, hence, do not need to have their results forwarded.

The Hazard Unit computes control signals for the forwarding multiplexers to choose operands from the register file or from the results in the Memory or Writeback stage (ALUOutM or ResultW).

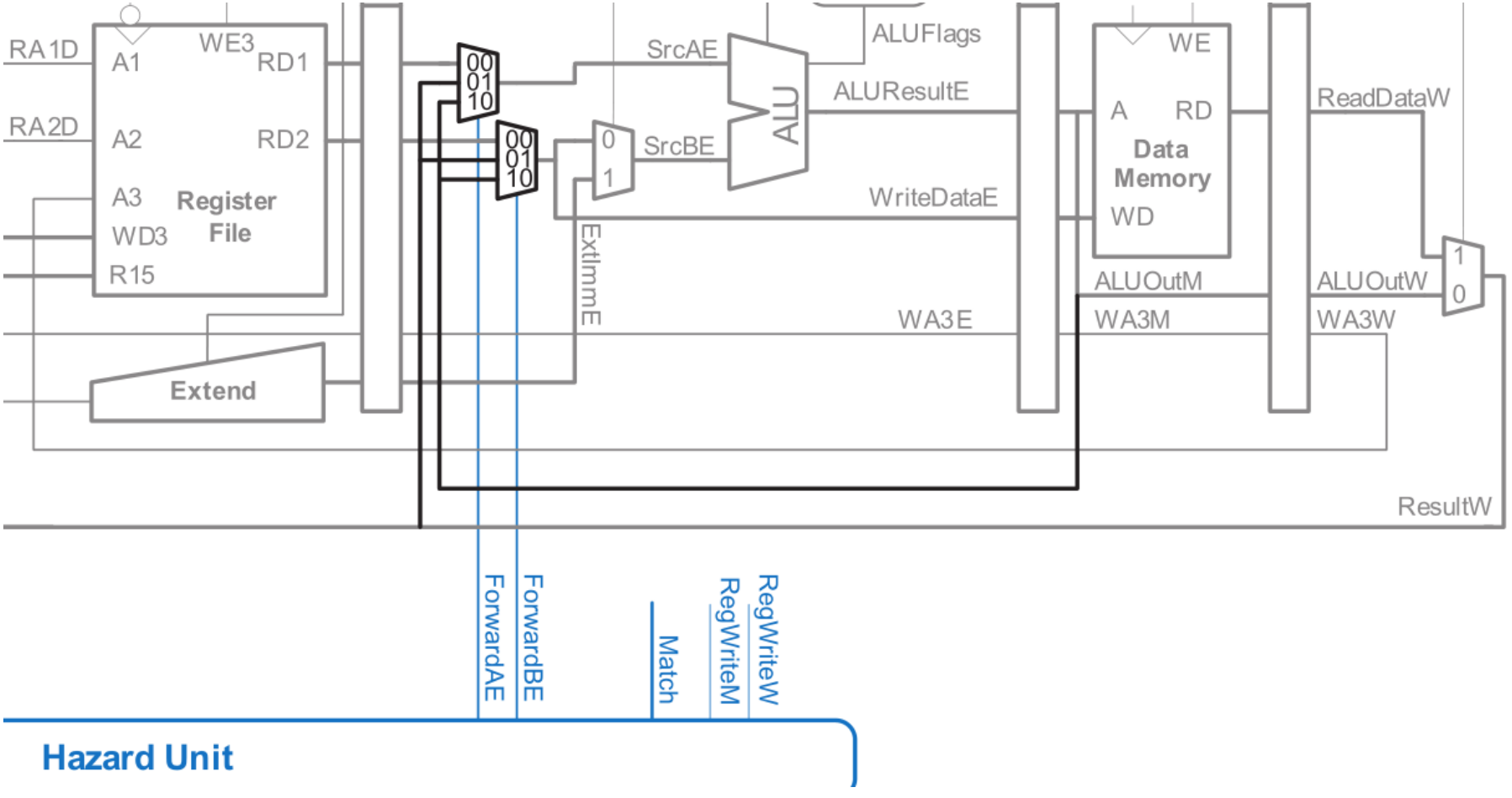
It should forward from a stage if that stage will write a destination register and the destination register matches the source register.

If both the Memory and Writeback stages contain matching destination registers, then the Memory stage should have priority, because it contains the more recently executed instruction.

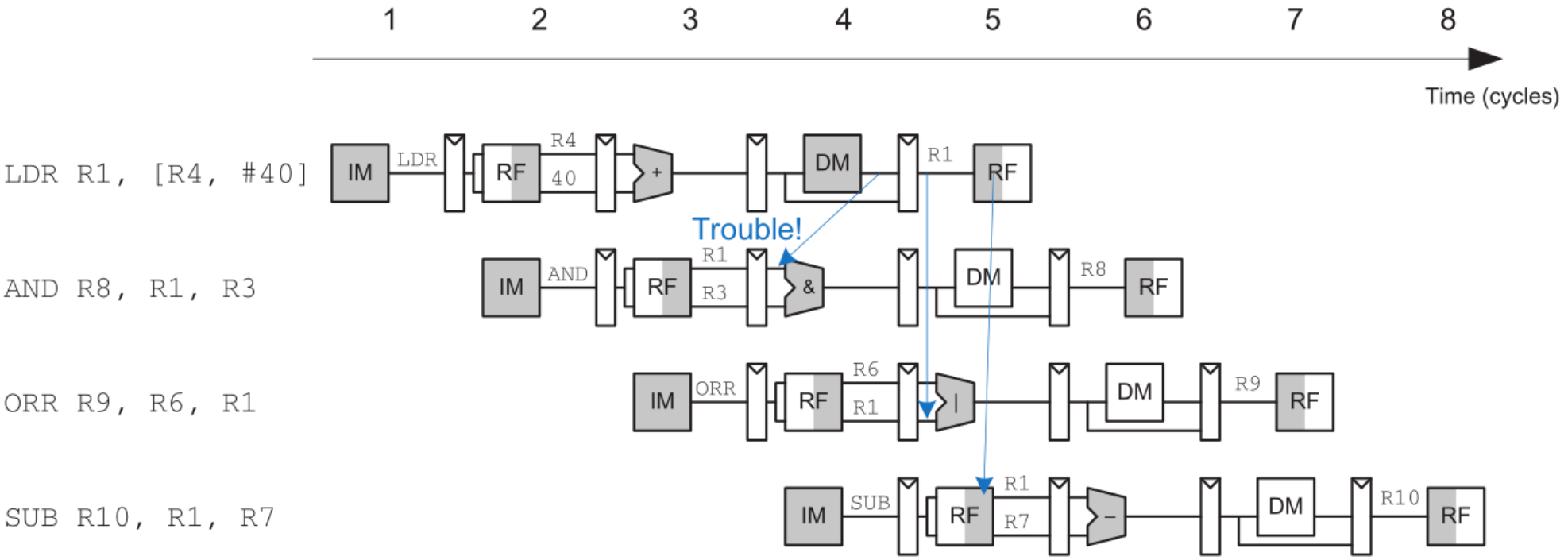
The function of the forwarding logic for SrcAE and SrcBE:

```
if (Match_1E_M • RegWriteM) ForwardAE = 10; // SrcAE = ALUOutM
else if (Match_1E_W • RegWriteW) ForwardAE = 01; //SrcAE = ResultW
else ForwardAE = 00; // SrcAE from regfile

if (Match_2E_M • RegWriteM) ForwardAE = 10; // SrcBE = ALUOutM
else if (Match_2E_W • RegWriteW) ForwardAE = 01; //SrcBE = ResultW
else ForwardAE = 00; // SrcBE from regfile
```



Trouble forwarding from LDR



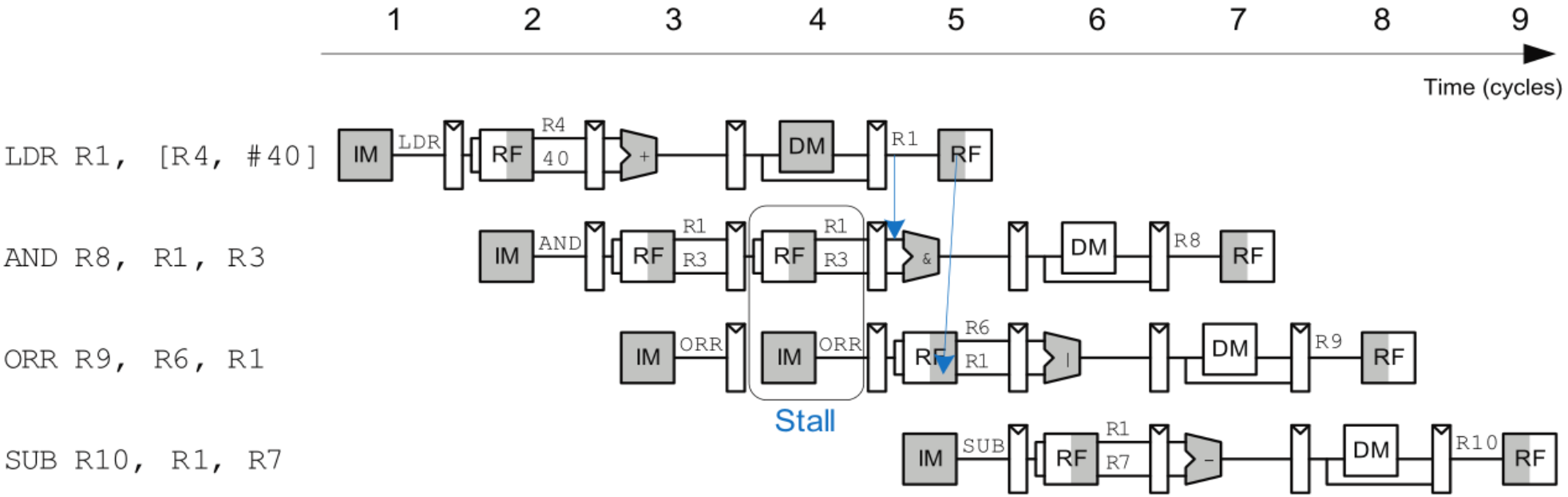
The LDR instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction.

LDR has a **two-cycle latency**, because a dependent instruction cannot use its result until two cycles later.

We cannot solve this hazard with forwarding.

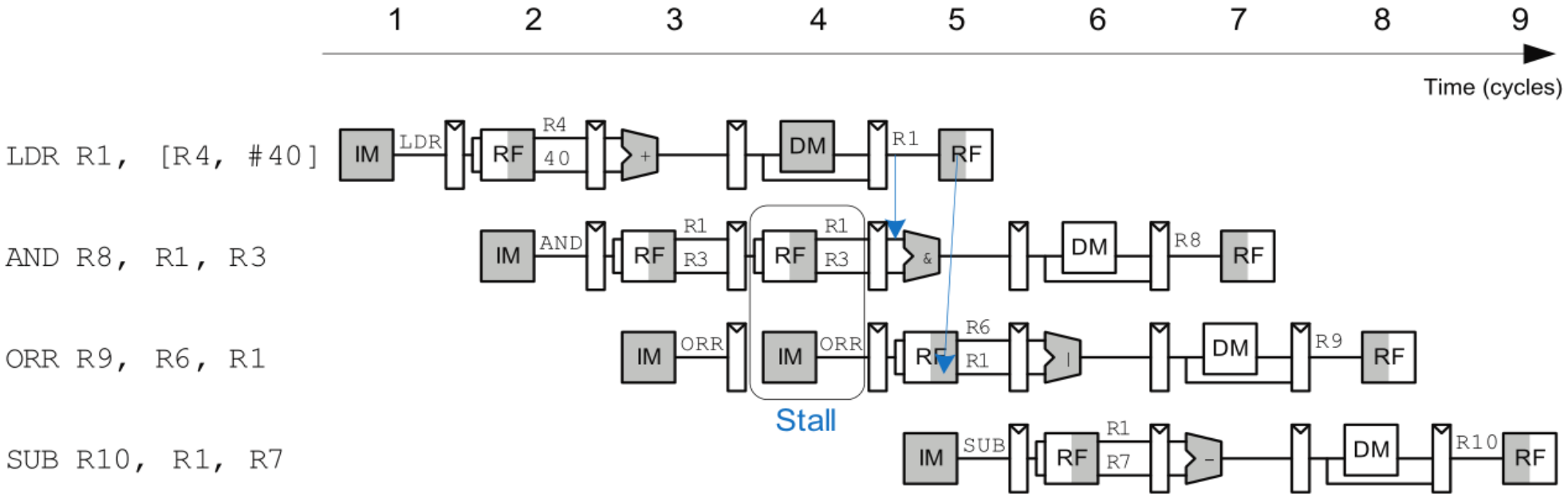
The alternative solution is to **stall** the pipeline.

This means to hold up operation until the data is available.



AND enters the Decode stage in cycle 3 and stalls there through cycle 4.

ORR must remain in the Fetch stage during both cycles as well, because the Decode stage is full.



Note that the Execute stage is unused in cycle 4.

The Memory stage is unused in cycle 5 and

The Writeback stage is unused in cycle 6.

This unused stage propagating through the pipeline is called a **bubble**, and it behaves like a `NOP` instruction.

The bubble is introduced by zeroing out the Execute stage control signals during a Decode stall so that the bubble performs no action and changes no architectural state.

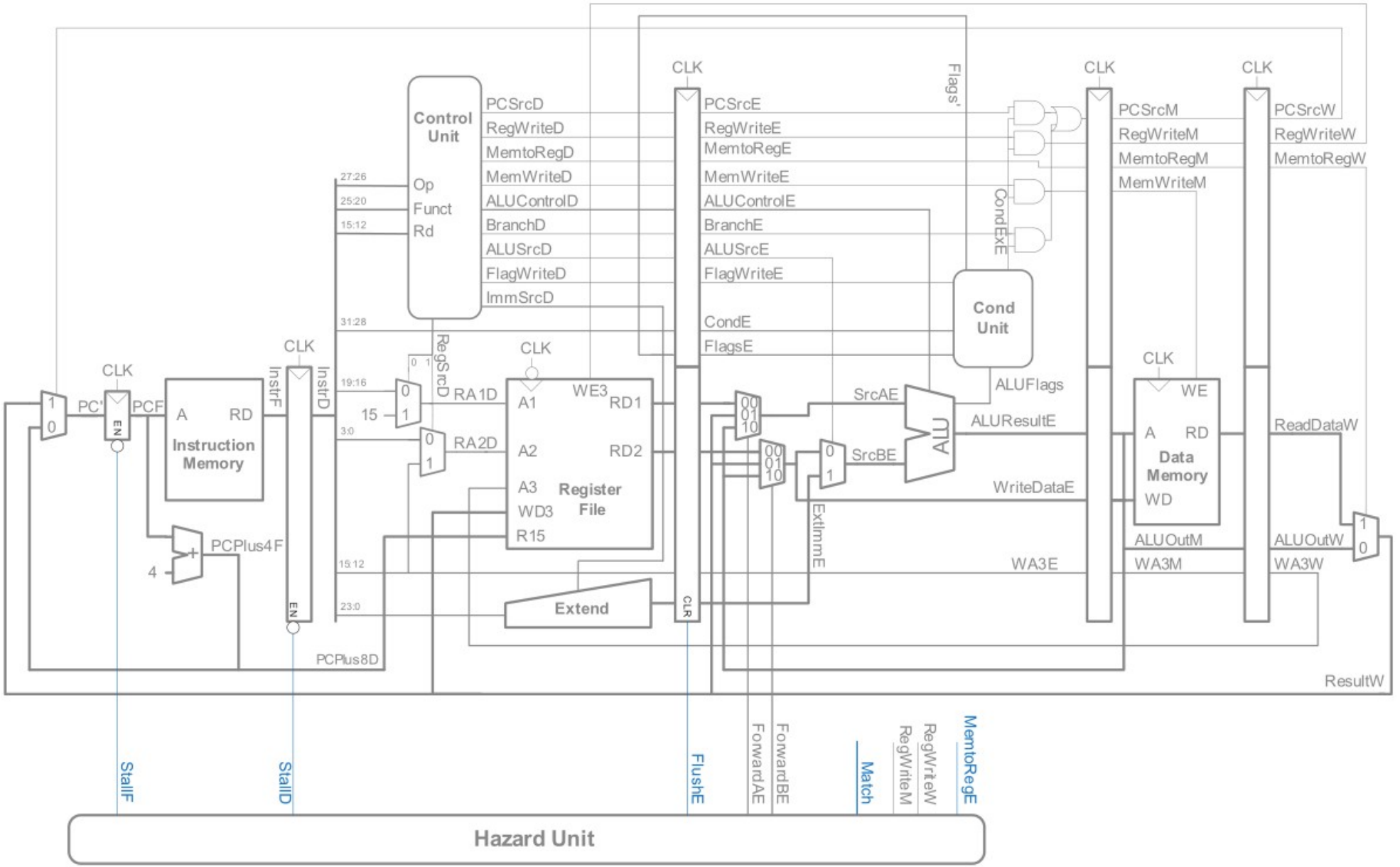
Stalling a stage is performed by disabling the pipeline register, so that the contents do not change.

When a stage is stalled, all previous stages must also be stalled, so that no subsequent instructions are lost.

The pipeline register directly after the stalled stage must be cleared (flushed) to prevent bogus information from propagating forward.

Stalls degrade performance, so they should be used only when necessary.

Pipelined processor with stalls to solve LDR data hazard



The Hazard Unit examines the instruction in the Execute stage.

If it is an `LDR` and its destination register (WA3E) matches either source operand of the instruction in the Decode stage (RA1D or RA2D), then that instruction must be stalled in the Decode stage until the source operand is ready.

Stalls are supported by adding enable inputs (EN) to the Fetch and Decode pipeline registers and a synchronous reset/clear (CLR) input to the Execute pipeline register.

The MemtoReg signal is asserted for the `LDR` instruction.

Hence, the logic to compute the stalls and flushes is

```
Match_12D_E = (RA1D == WA3E) + (RA2D == WA3E)
LDRstall = Match_12D_E • MemtoRegE
StallF = StallD = FlushE = LDRstall
```

Solving control hazards

The B instruction presents a **control hazard**:

The pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched.

Writes to R15 (PC) present a similar control hazard.

One mechanism for dealing with the control hazard is to stall the pipeline until the branch decision is made (i.e., PCSrcW is computed).

Because the decision is made in the Writeback stage, the pipeline would have to be stalled for four cycles at every branch.

This degrades the system performance if it occurs often.

An alternative is to **predict** whether the branch will be taken and begin executing instructions based on the prediction.

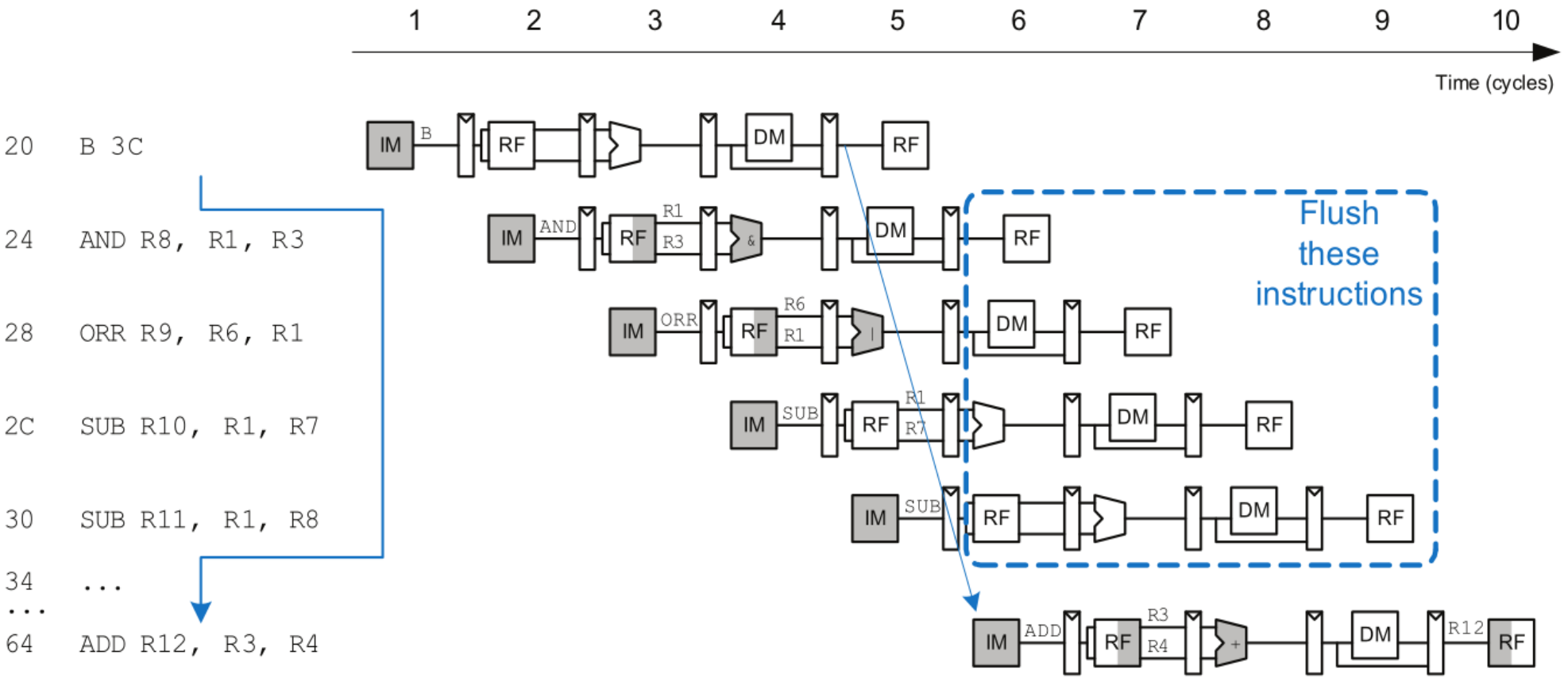
Once the branch decision is available, the processor can throw out the instructions if the prediction was wrong.

In the pipeline presented so far, the processor predicts that branches are not taken and simply continues executing the program in order until PCSrcW is asserted to select the next PC from ResultW instead.

If the branch should have been taken, then the four instructions following the branch must be **flushed** (discarded) by clearing the pipeline registers for those instructions.

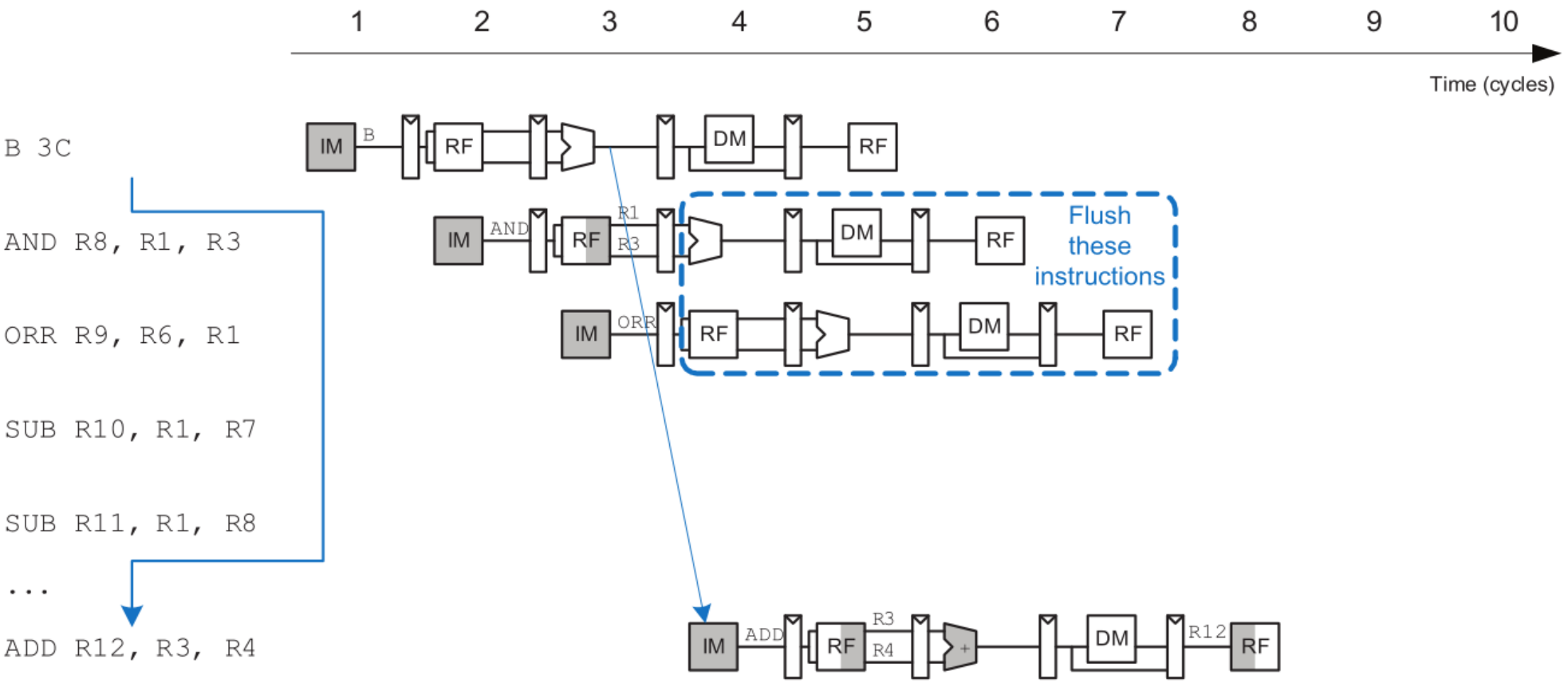
These wasted instruction cycles are called the **branch misprediction penalty**.

Flushing when a branch is taken



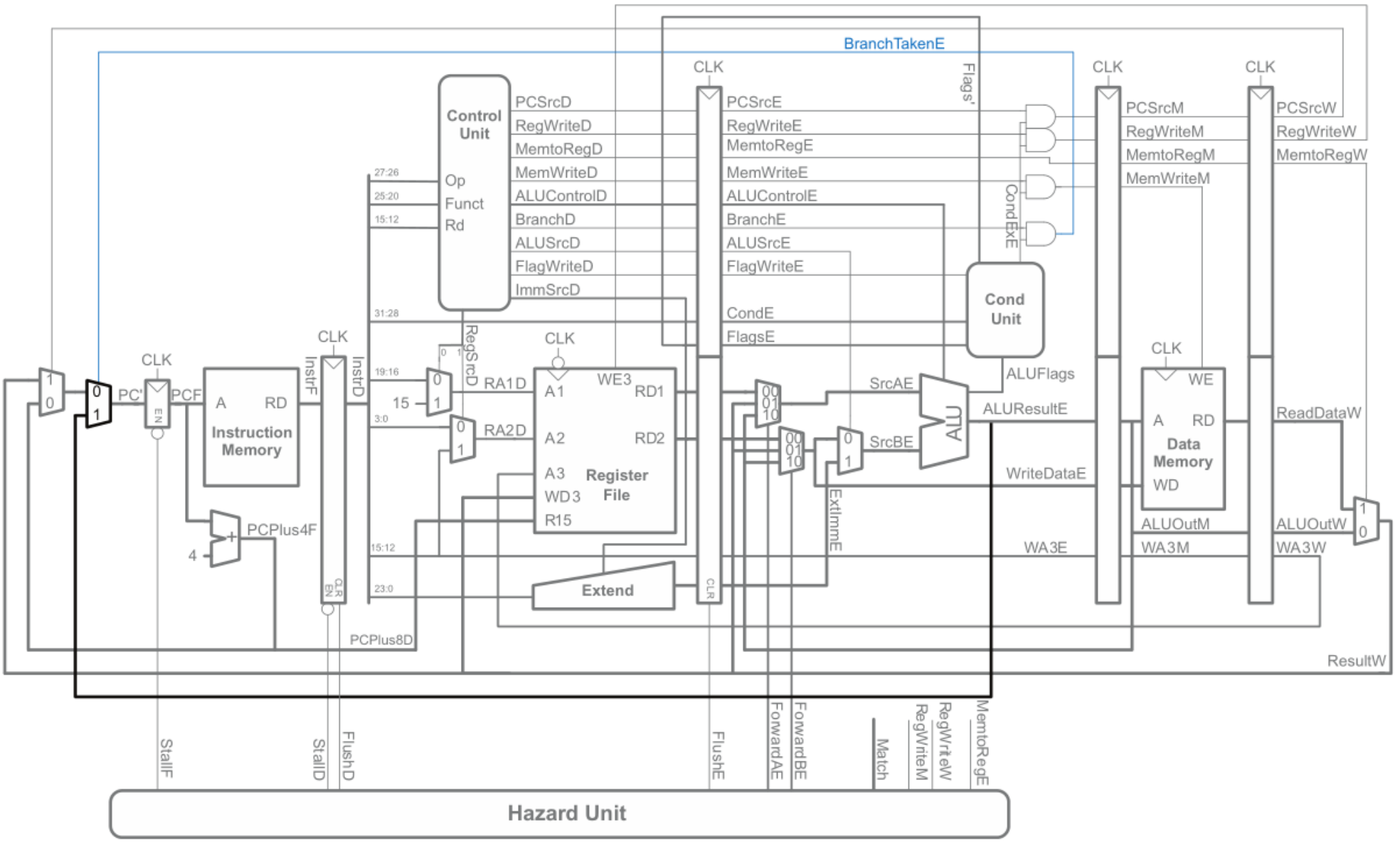
We could reduce the branch misprediction penalty if the branch decision could be made earlier.

The branch decision can be made in the Execute stage when the destination address has been computed and CondEx is known.



Now the branch misprediction penalty is reduced to only two instructions rather than four.

Pipelined processor handling branch control hazard



A branch multiplexer is added before the PC register to select the branch destination from ALUResultE.

The BranchTakenE signal controlling this multiplexer is asserted on branches whose condition is satisfied.

PCSrcW is now only asserted for writes to the PC, which still occur in the Writeback stage.

When a branch is taken, the subsequent two instructions must be flushed from the pipeline registers of the Decode and Execute stages.

When a write to the PC is in the pipeline, the pipeline should be stalled until the write completes.

This is done by stalling the Fetch stage.

Stalling one stage also requires flushing the next to prevent the instruction from being executed repeatedly.

PCWrPending is asserted when a PC write is in progress (in the Decode, Execute, or Memory stage).

During this time, the Fetch stage is stalled and the Decode stage is flushed.

When the PC write reaches the Writeback stage (PCSrcW asserted), StallF is released to allow the write to occur, but FlushD is still asserted so that the undesired instruction in the Fetch stage does not advance.

```
PCWrPendingF = PCSrcD + PCSrcE + PCSrcM;
StallD = LDRstall;
StallF = LDRstall + PCWrPendingF;
FlushE = LDRstall + BranchTakenE;
FlushD = PCWrPendingF + PCSrcW + BranchTakenE;
```

Branches are very common, and even a two-cycle misprediction penalty still impacts performance.

With a bit more work, the penalty could be reduced to one cycle for many branches.

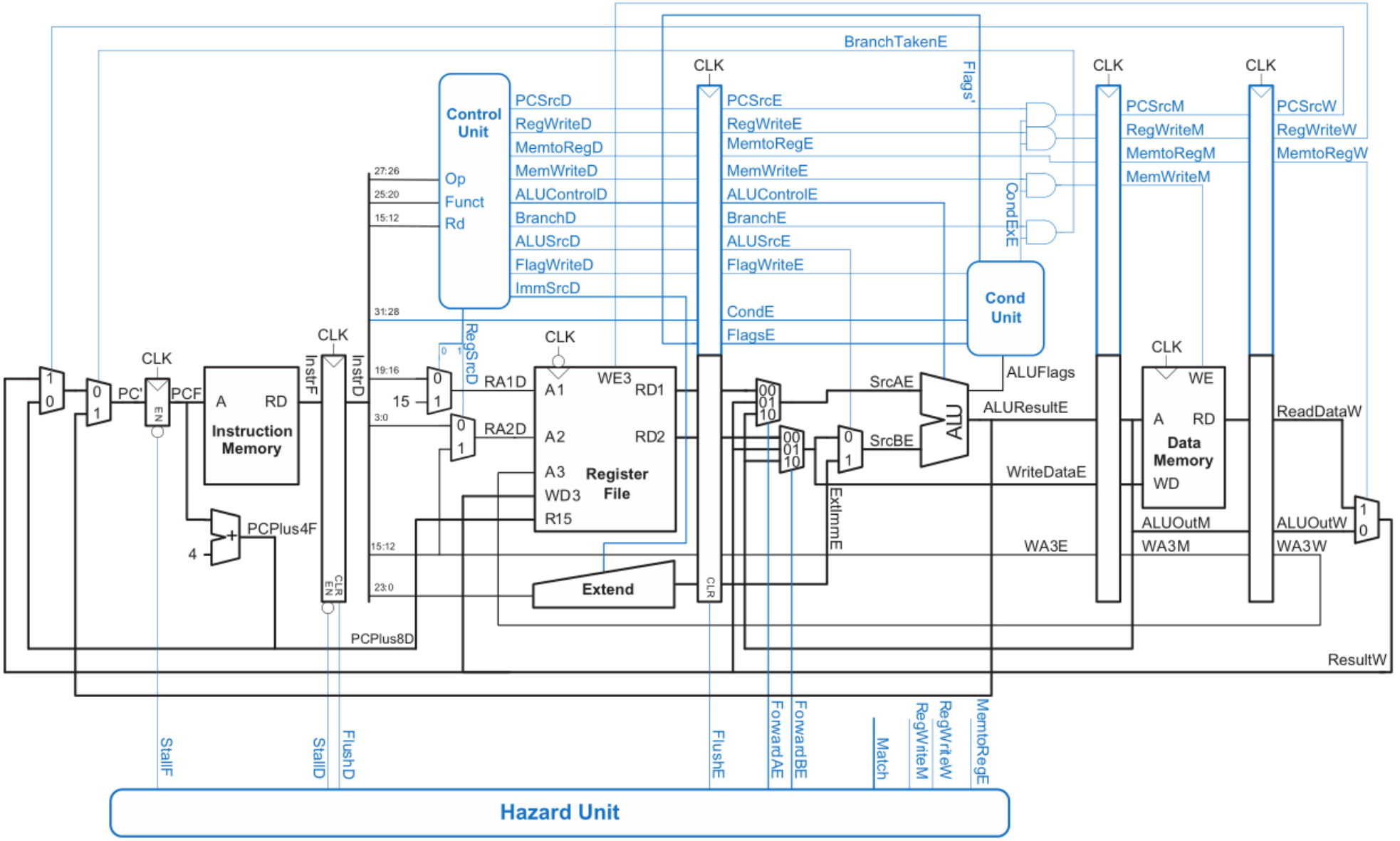
The destination address must be computed in the Decode stage as

$$PCBranchD = PCPlus8D + ExtImmD$$

BranchTakenD must also be computed in the Decode stage based on ALUFlagsE generated by the previous instruction.

This might increase the cycle time of the processor if these flags arrive late.

Pipelined processor with full hazard handling



Performance analysis

$$Execution\ Time = \left(\#instructions\right)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right)$$

The pipelined processor ideally would have a CPI of 1, because a new instruction is issued every cycle.

A stall or a flush wastes a cycle, so the CPI is higher and depends on the specific program being executed.

One of the challenges of designing a pipelined processor is to understand all the possible interactions between instructions and to discover all the hazards that may exist.

The sequencing overhead (clk-to-Q and setup times) of the registers applies to every pipeline stage, not just once to the overall datapath, as it was in the single-cycle processor.

Exercise 16.1

The pipelined processor performance might be better if branches take place during the Decode stage rather than the Execute stage.

Show how to modify the pipelined processor from the last picture to branch in the Decode stage.

How do the stall, flush, and forwarding signals change?