5 : The Processor (Pipelined)

Rajesh Panicker,
NUS

CG3207

Acknowledgement :

- Text by Harris and Harris and companion materials
- Text by Patterson & Hennessy and companion materials by Mary Jane Irwin, PSU

Instruction Critical Paths

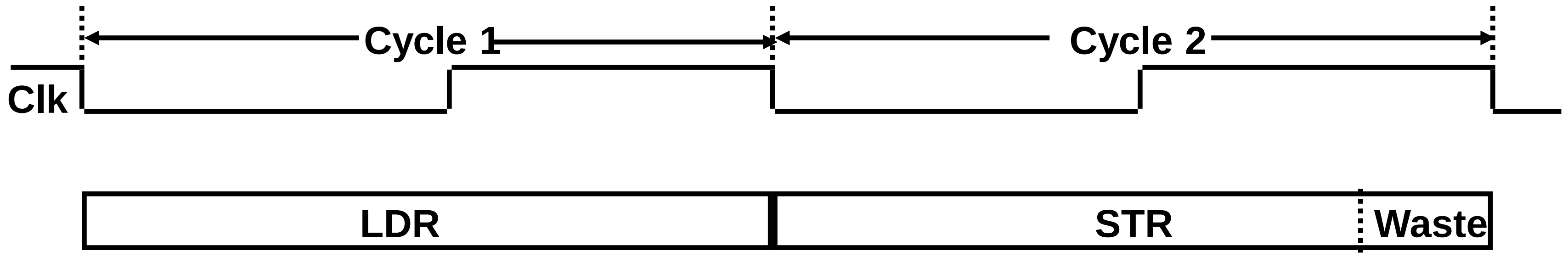
- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC read/write, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU (120 ps)
  - Adders (75 ps)
  - Register File access (reads – 100 ps, writes – 60 ps)

Instr.	**I Mem**	**Reg Rd**	**ALU Op**	**D Mem**	**Reg Wr**	**Total**
DP	200	100	120		60	480
`LDR`	200	100	120	200	60	680
`STR`	200	100	120	200		620
`B`	200	100	120			420

Note : Addition for PC+8 is done in parallel with Imem access and takes even less time. So adders are not in the critical path

Single Cycle Disadvantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

How Can We Make It Faster?

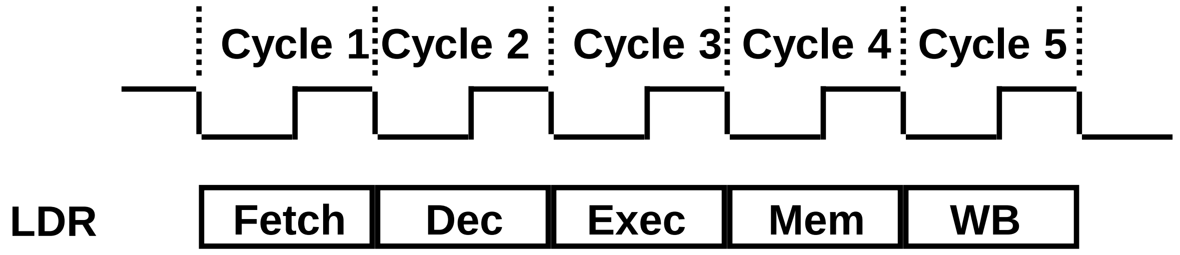
- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – (all?) modern processors are pipelined for performance

- Remember *the* performance equation

  CPU time = Instruction Count x CPI x Clock Cycle Time

- Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
  - A five stage pipeline is nearly five times faster because the CCT is nearly five times faster

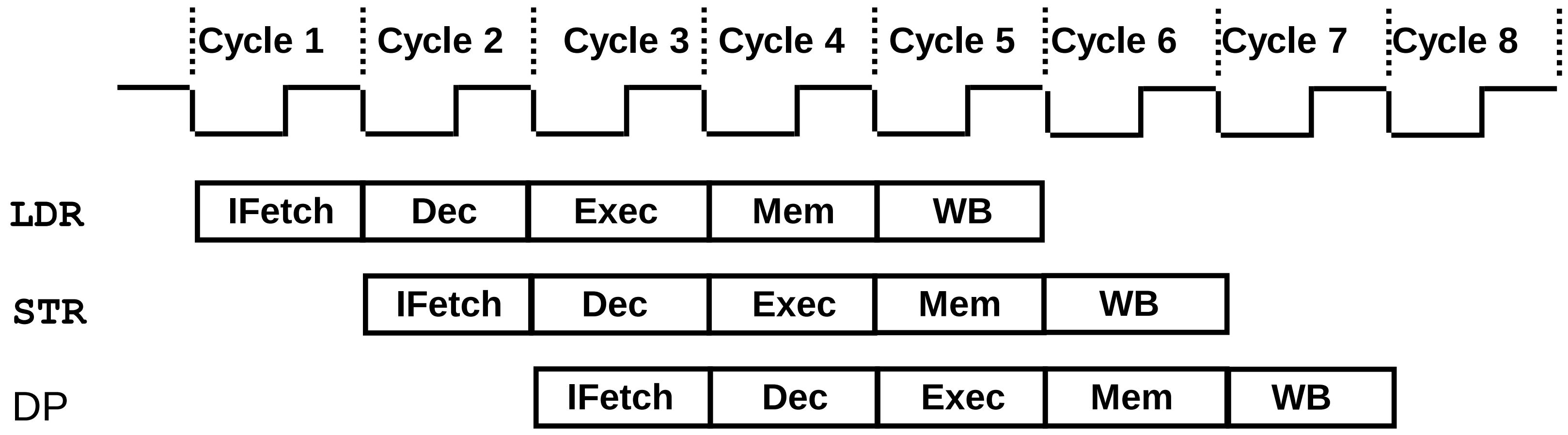
The Five Stages of Load Instruction



- Fetch: Instruction Fetch and Update PC
- Dec: Registers Fetch and Instruction Decode
- Exec: Execute DP-type; calculate memory address
- Mem: Read/write the data from/to the Data Memory
- WB: Write the result data into the register file

A Pipelined ARM Processor

- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced

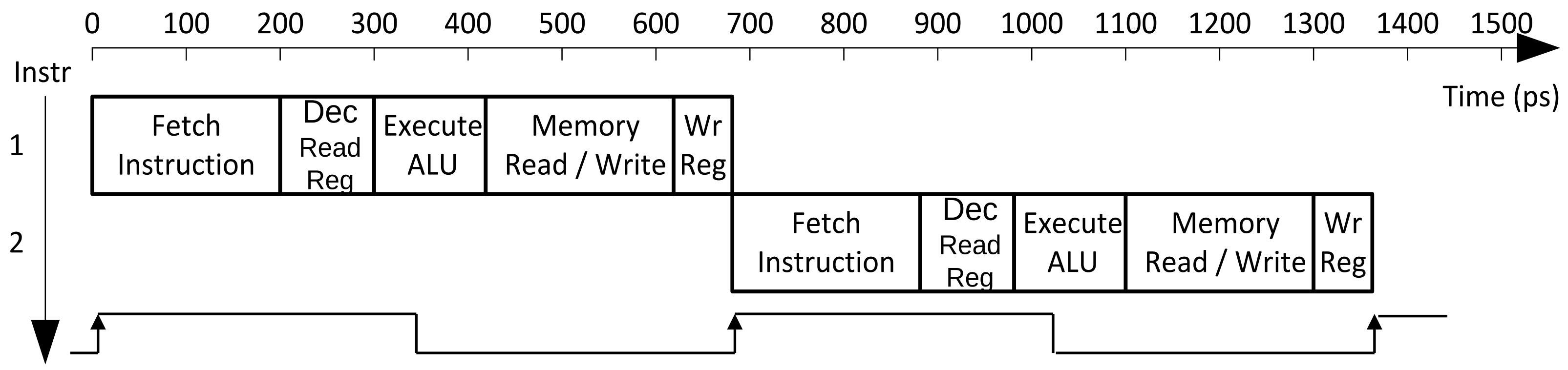


- clock cycle (pipeline stage time) is limited by the **slowest** stage
  - for some stages don't need the whole clock cycle (e.g., WB)
  - for some instructions, some stages are wasted cycles (i.e., nothing is done during that cycle for that instruction)

Single Cycle versus Pipeline

Single-Cycle

#instructions/sec = 1/680ps = 1.47 Billion

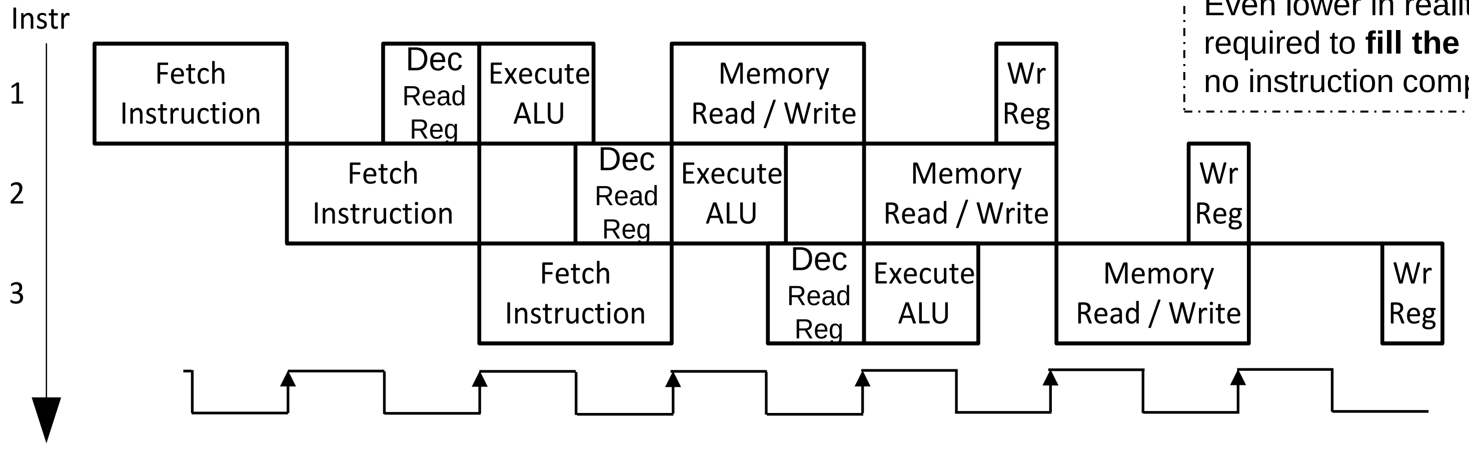


Speedup = 5/1.47 = 3.4
Pipeline stages not balanced =>
Speedup less than ideal value of 5.

Even lower in reality - some cycles required to **fill the pipeline** during which no instruction completes execution

Pipelined

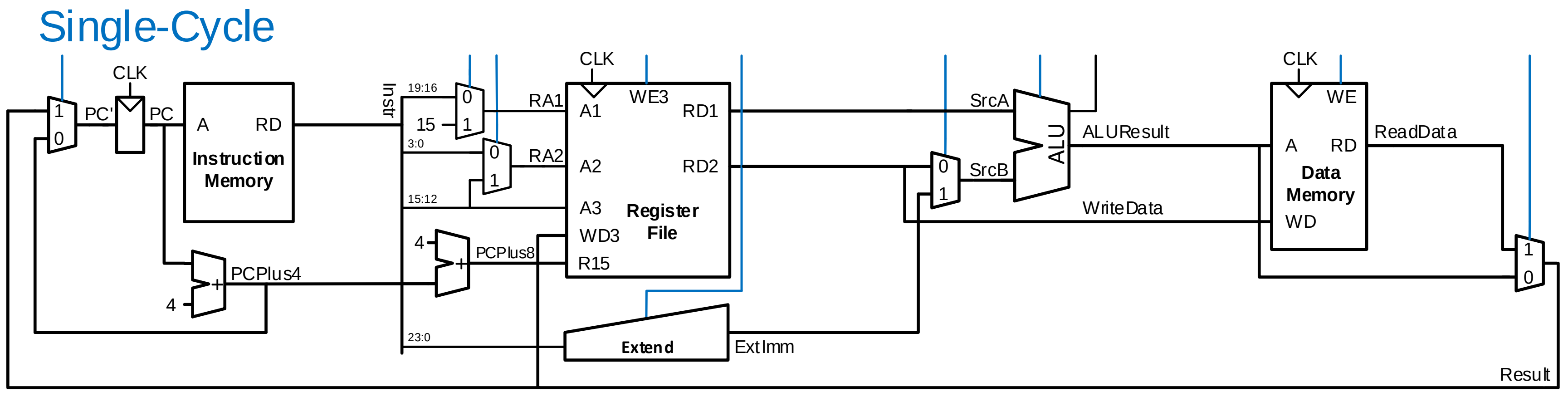
#instructions/sec = 1/200ps = 5 Billion

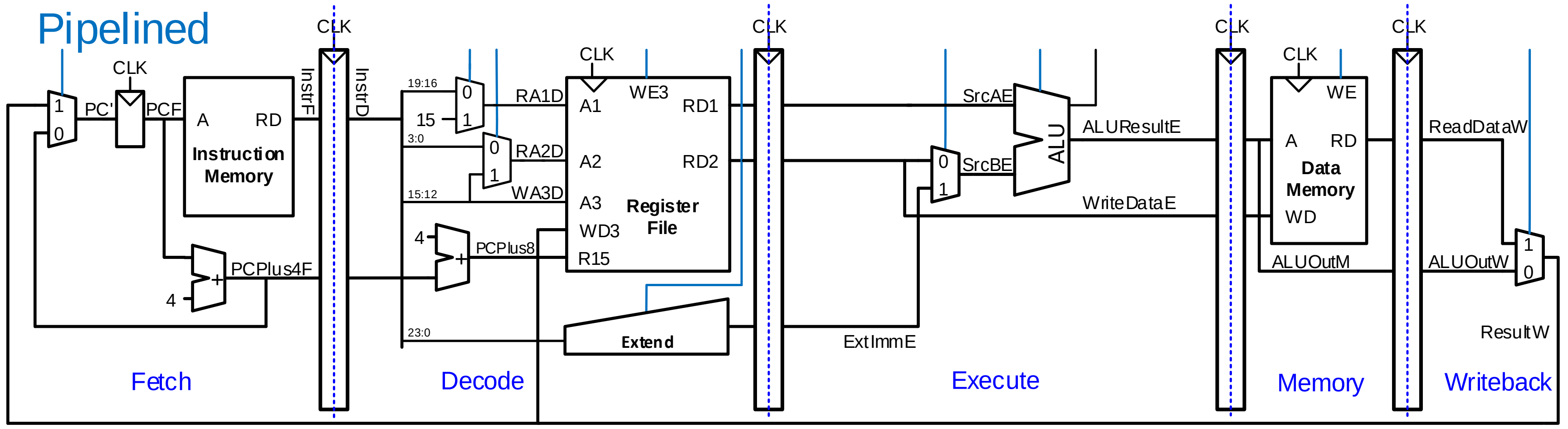


(b)

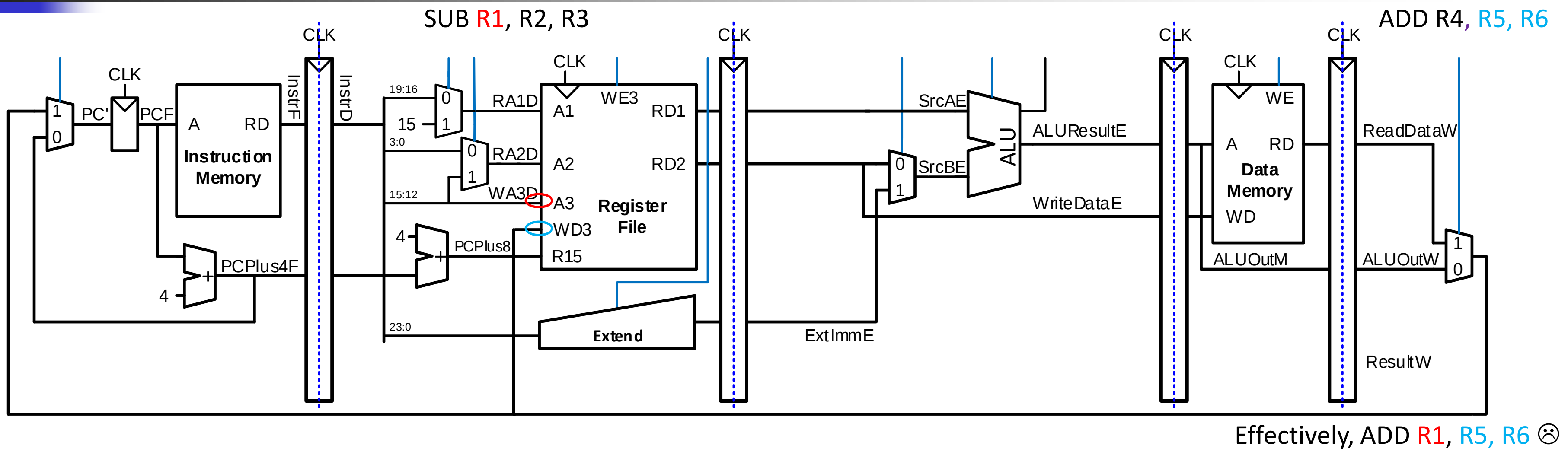
ARM Pipeline Datapath Additions/Mods

- State registers between each pipeline stage to isolate them

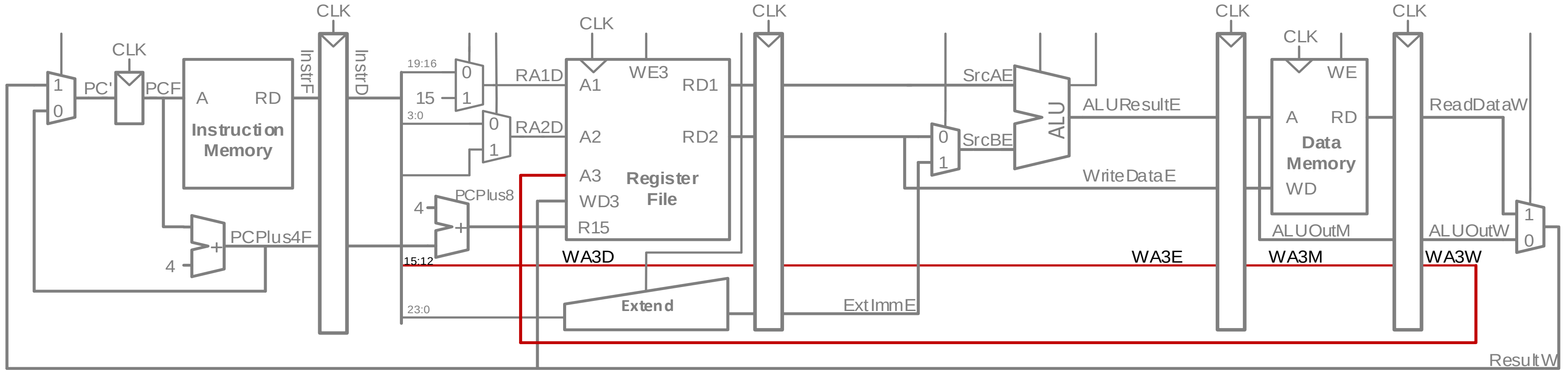




Corrected Pipelined Datapath

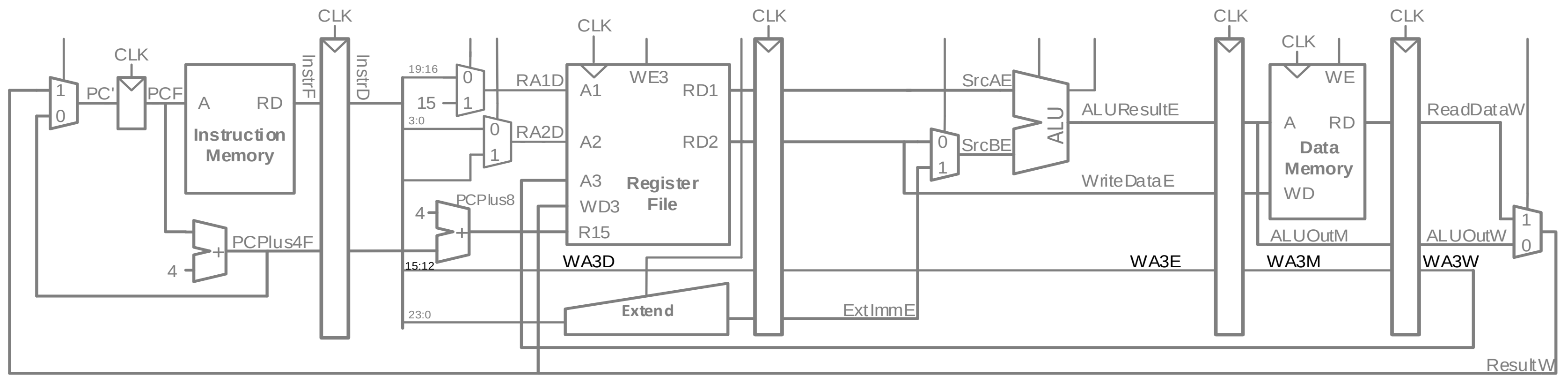


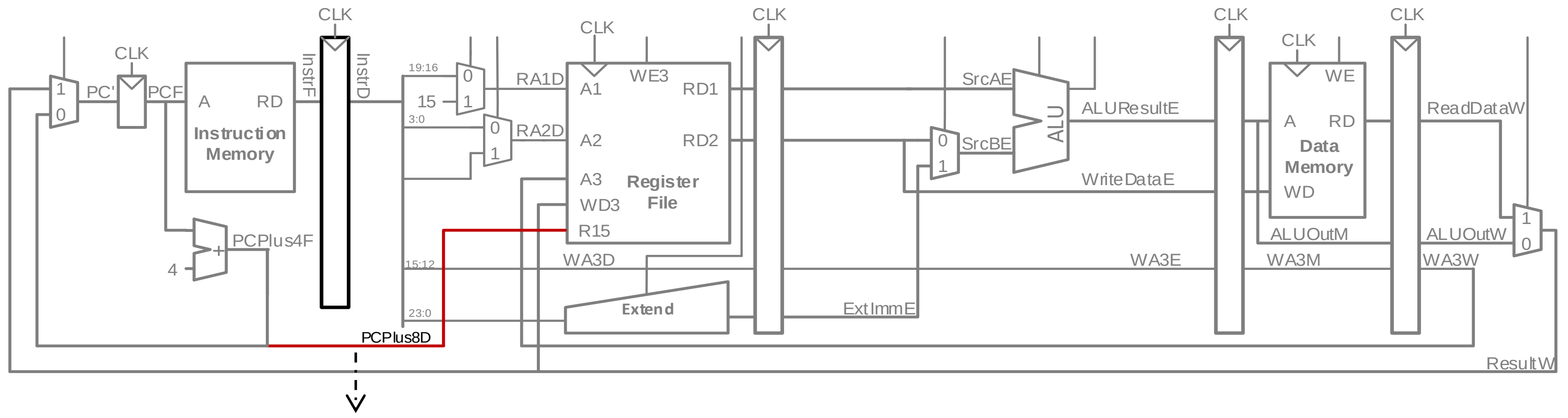
- *WA3* must arrive at same time as *Result* (all the information related to an instruction should move together, i.e., should be delayed equally)



Optimized Pipelined Datapath

- Remove adder by using *PCPlus4F* after *PC* has been updated to *PC*+4

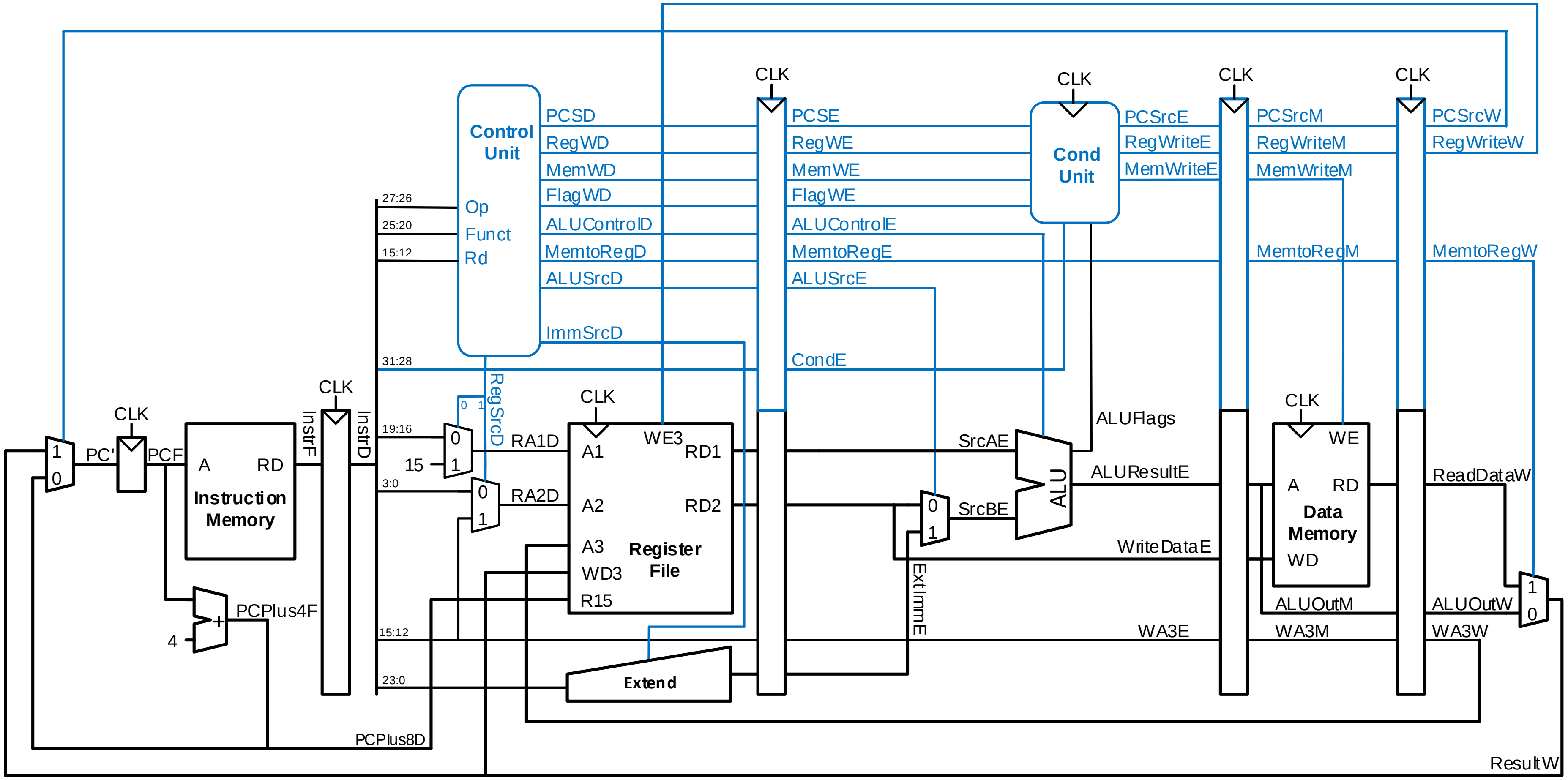




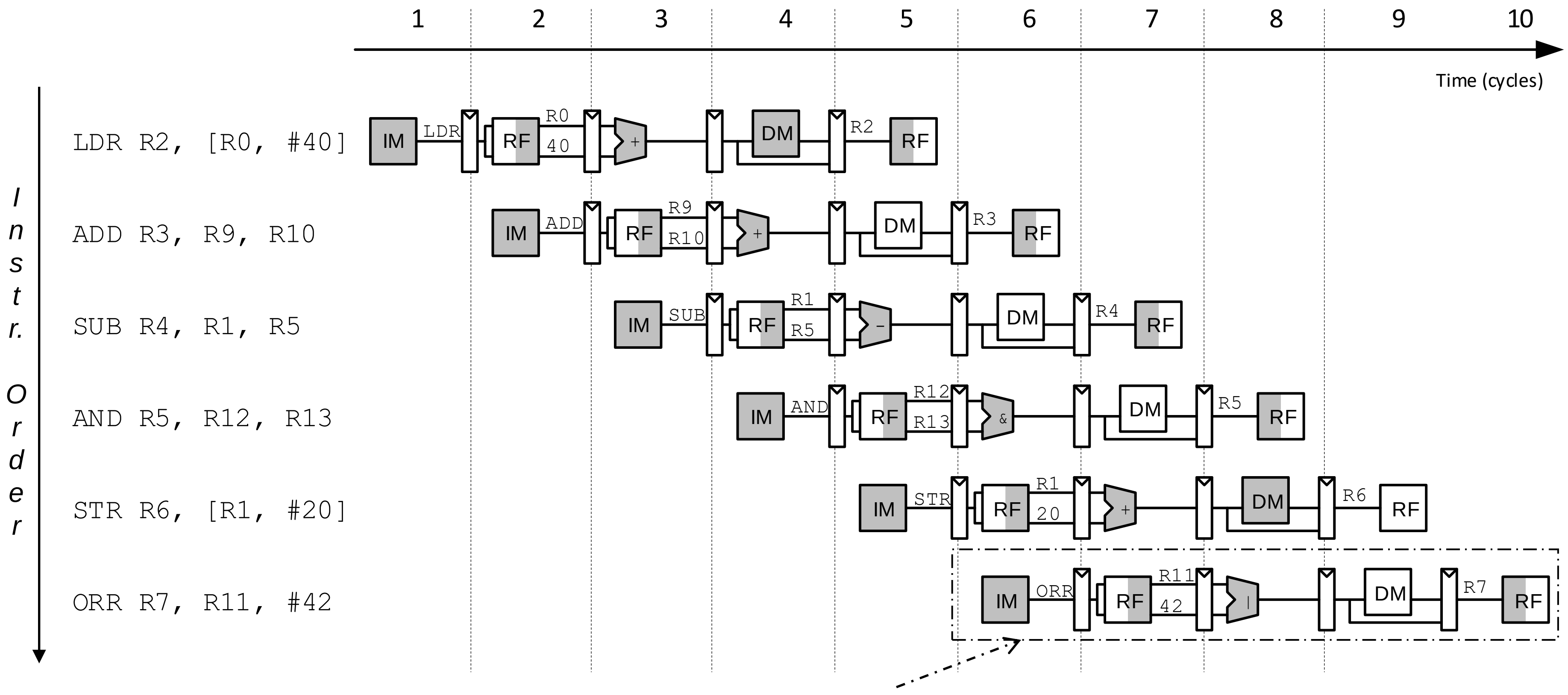
Verify that this works! (Hint : $PC_{Fetch} = PC_{Decode}+4$)

Pipelined Processor Control

- Same control unit as single-cycle processor
- Control delayed to proper pipeline stage



Why Pipeline? For Performance!



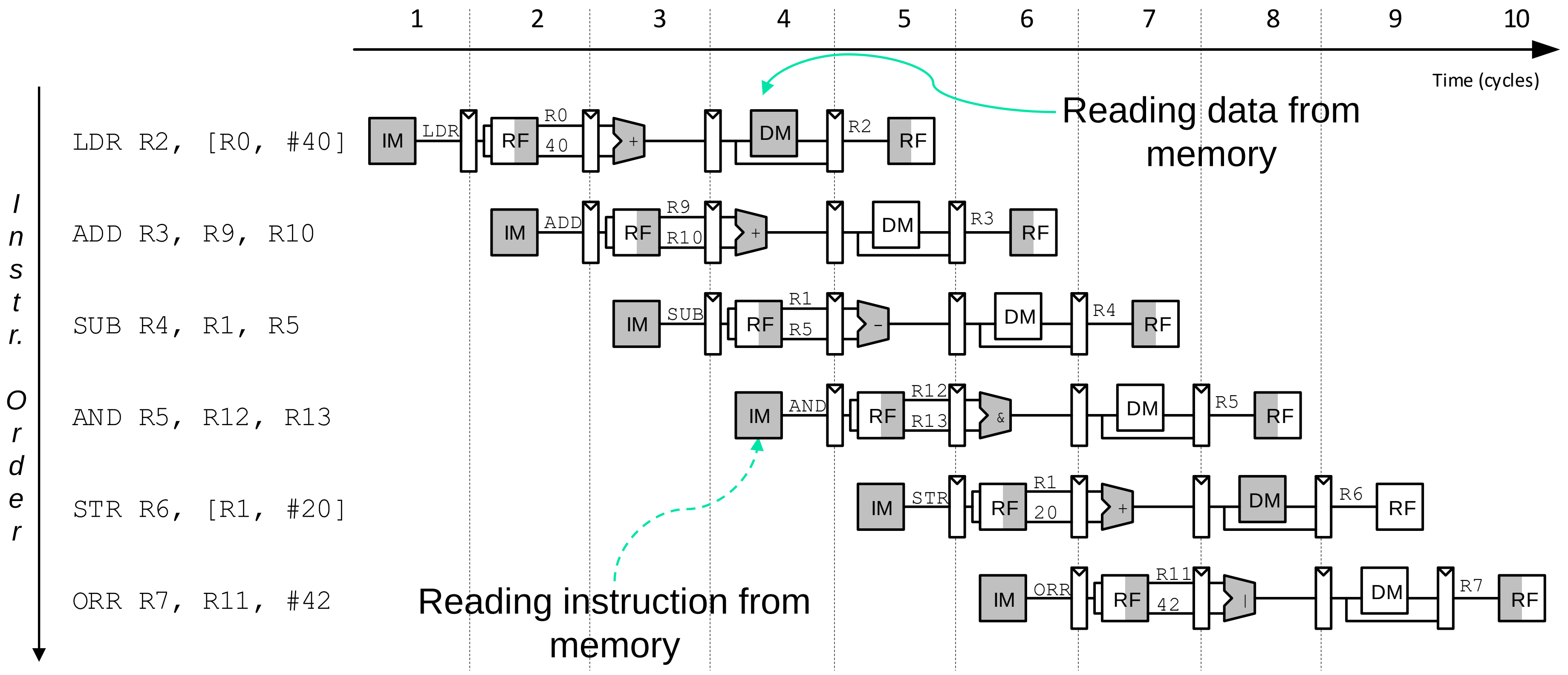
Abstract pipeline diagram

- Once the pipeline is full, (ideally) one instruction is completed every cycle, so CPI = 1

Can Pipelining Get Us Into Trouble?

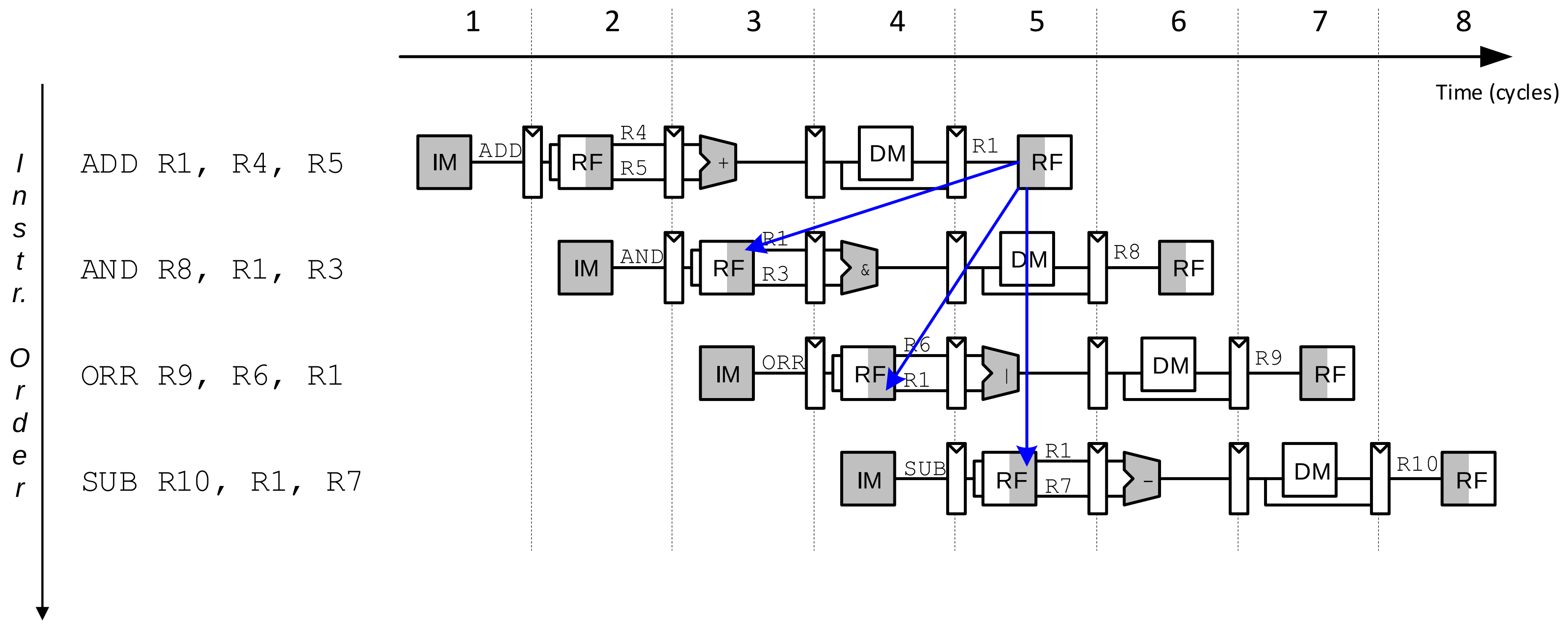
- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - Branch instructions, writes to R15, exceptions
- Can usually resolve hazards by waiting
  - Compiler (software) or pipeline control (hardware) must detect the hazard
  - and take action to resolve hazards

A Single Memory Would Be a Structural Hazard



- Fix with separate instr and data memories (IM and DM)
  - Or at least separate caches

Data Hazard

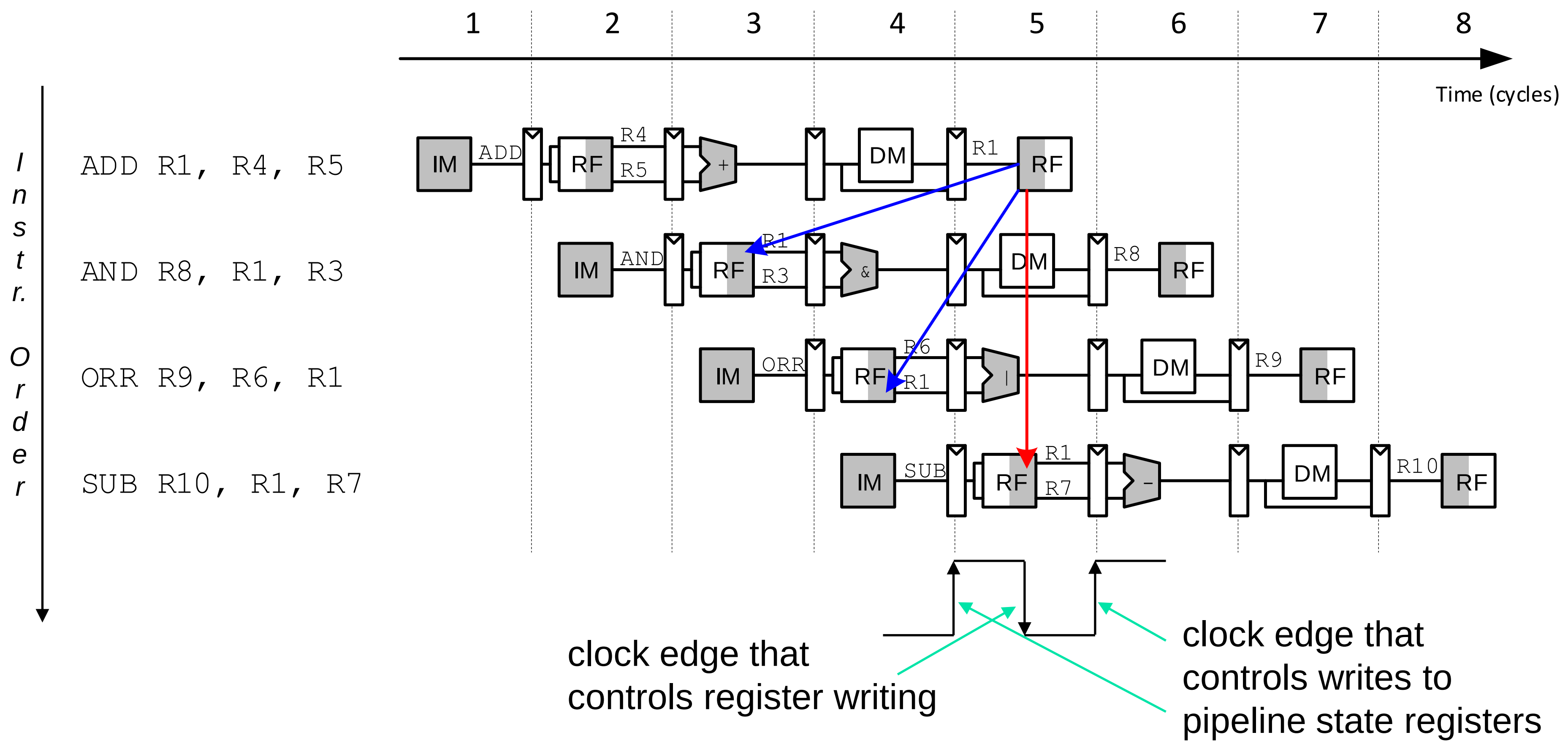


- Also known as RAW (read after write) hazard or true data dependency
- Occurs very frequently in practice -> represents the flow of information in the program

Handling Data Hazards

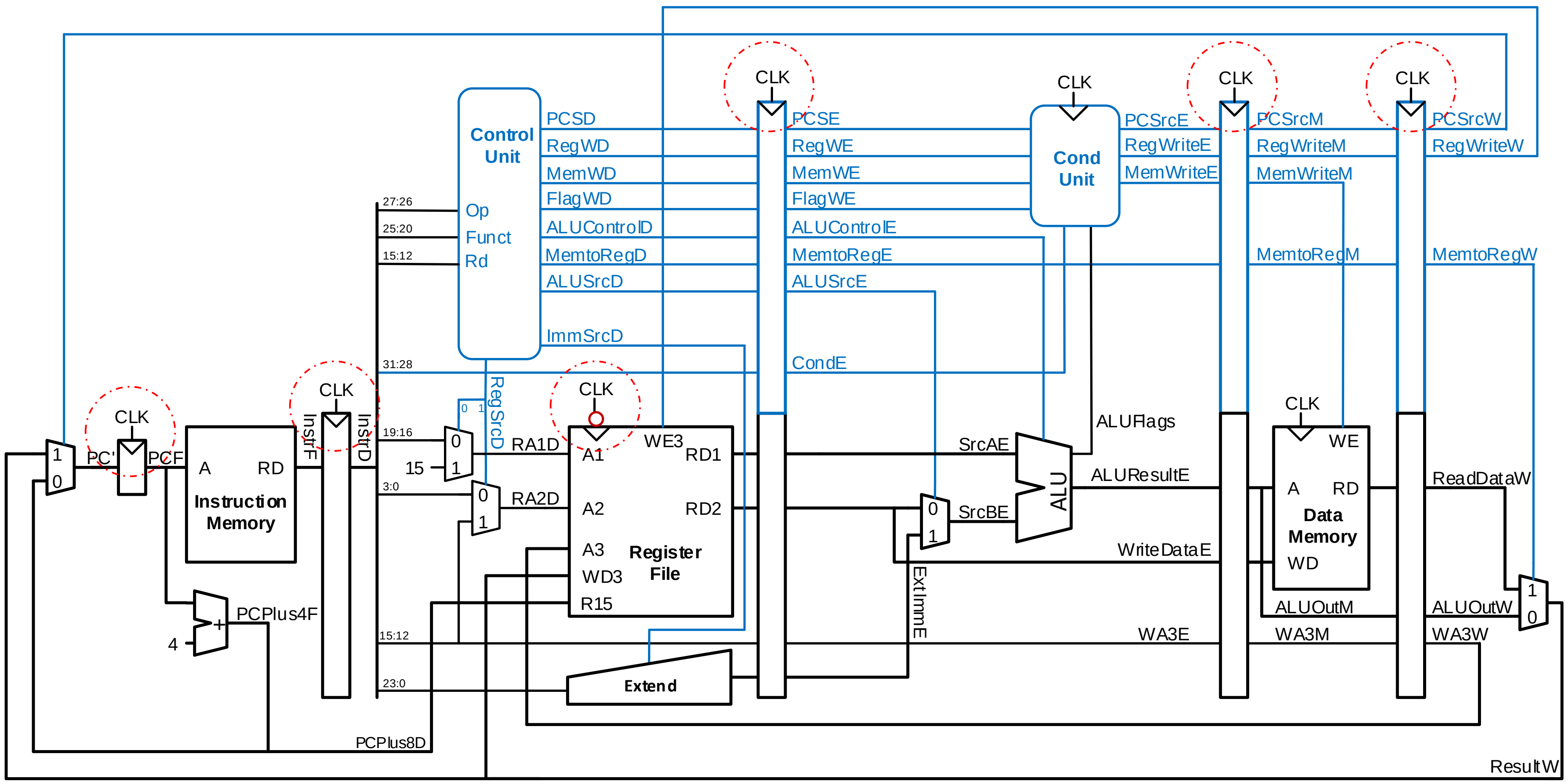
- Write register file and pipeline registers at different edges of the clock
- Insert `NOP`s in code at compile time
  - The instruction 0x00000000, which stands for `ANDEQ R0, R0, R0` which has no effect
  - The assembly instruction `NOP` will most likely expand to `MOV R0, R0` which is encoded 0xE1A00000
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Data Hazard : Using Different Clock Edges



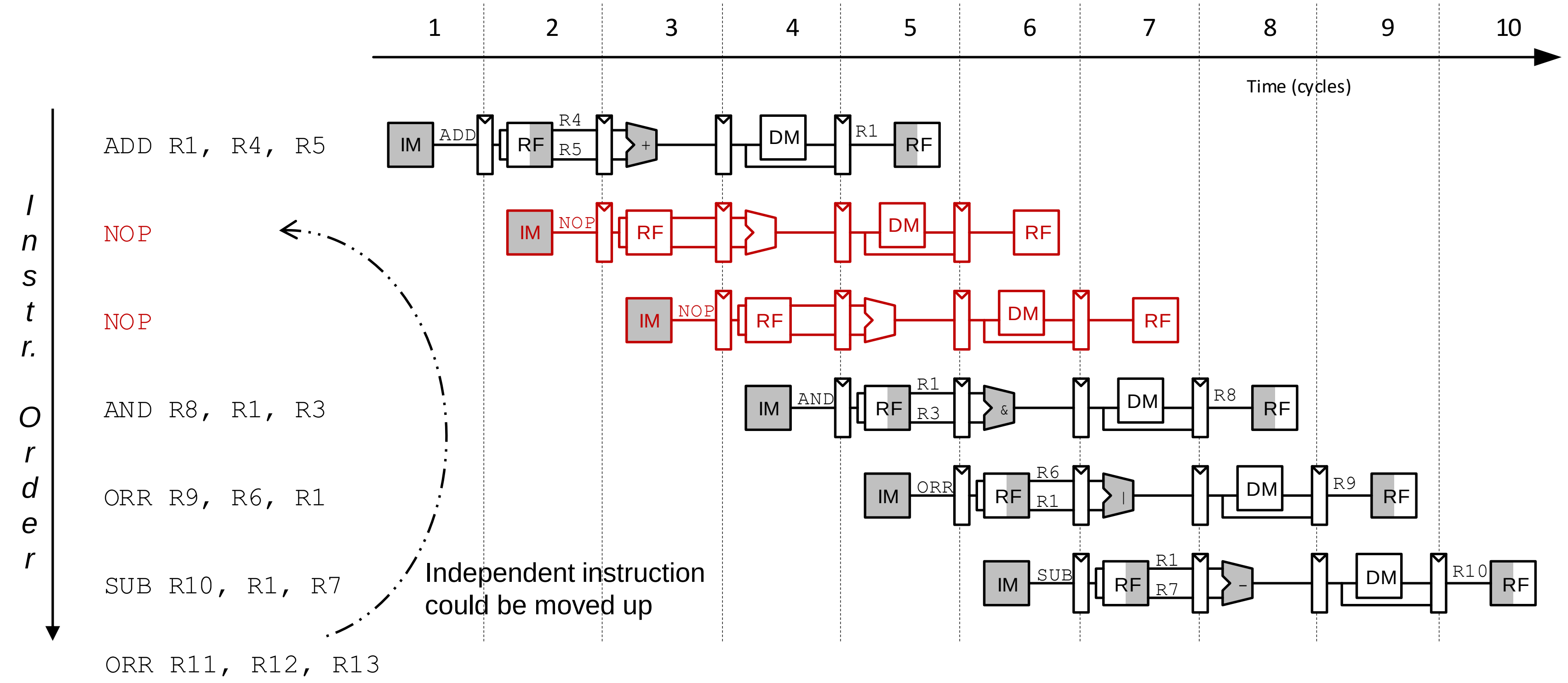
- Reading in the second half of the cycle and writing in the first half helps for instructions which are spaced by at least 2 instructions

Data Hazard : Using Different Clock Edges

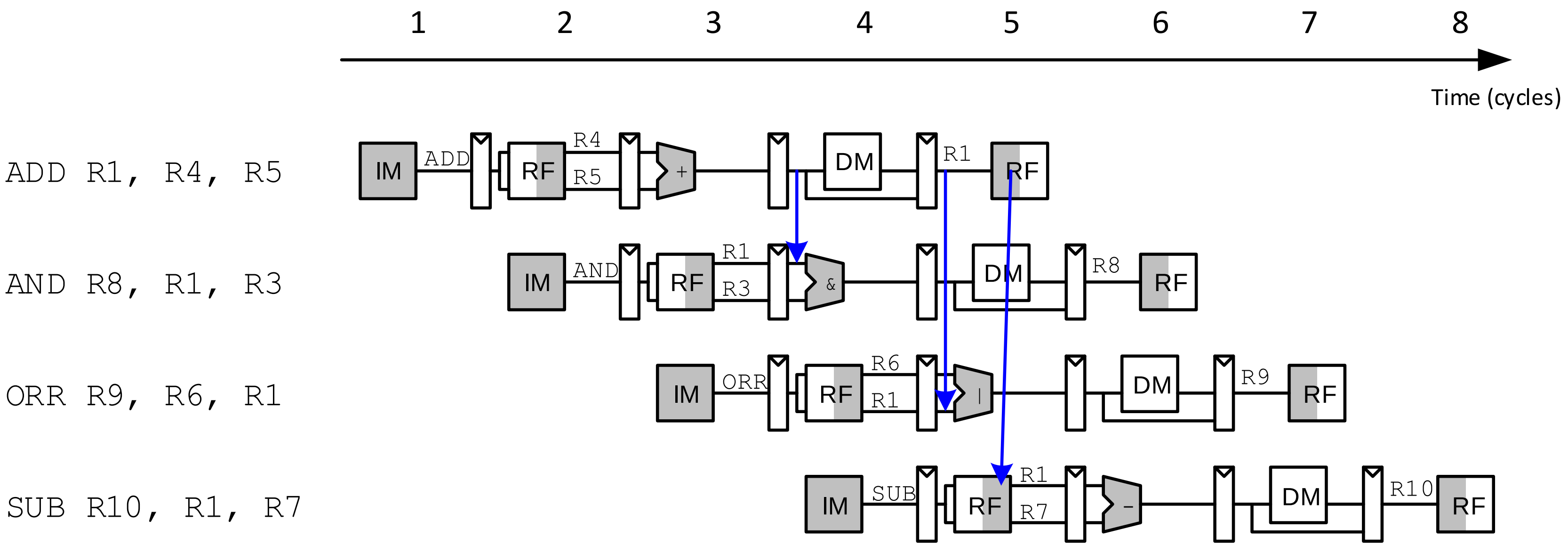


Data Hazard : Inserting NOPs

- Insert enough NOPs for result to be ready
  - NOPs waste time => CPI is greater than the ideal value of 1
  - NOPs waste code memory => makes the code bulkier / bloated
  - Compiler needs to know the microarchitecture => code is not very portable
- Or move independent useful instructions forward
  - Might not be possible all the time



Data Hazard : Data Forwarding

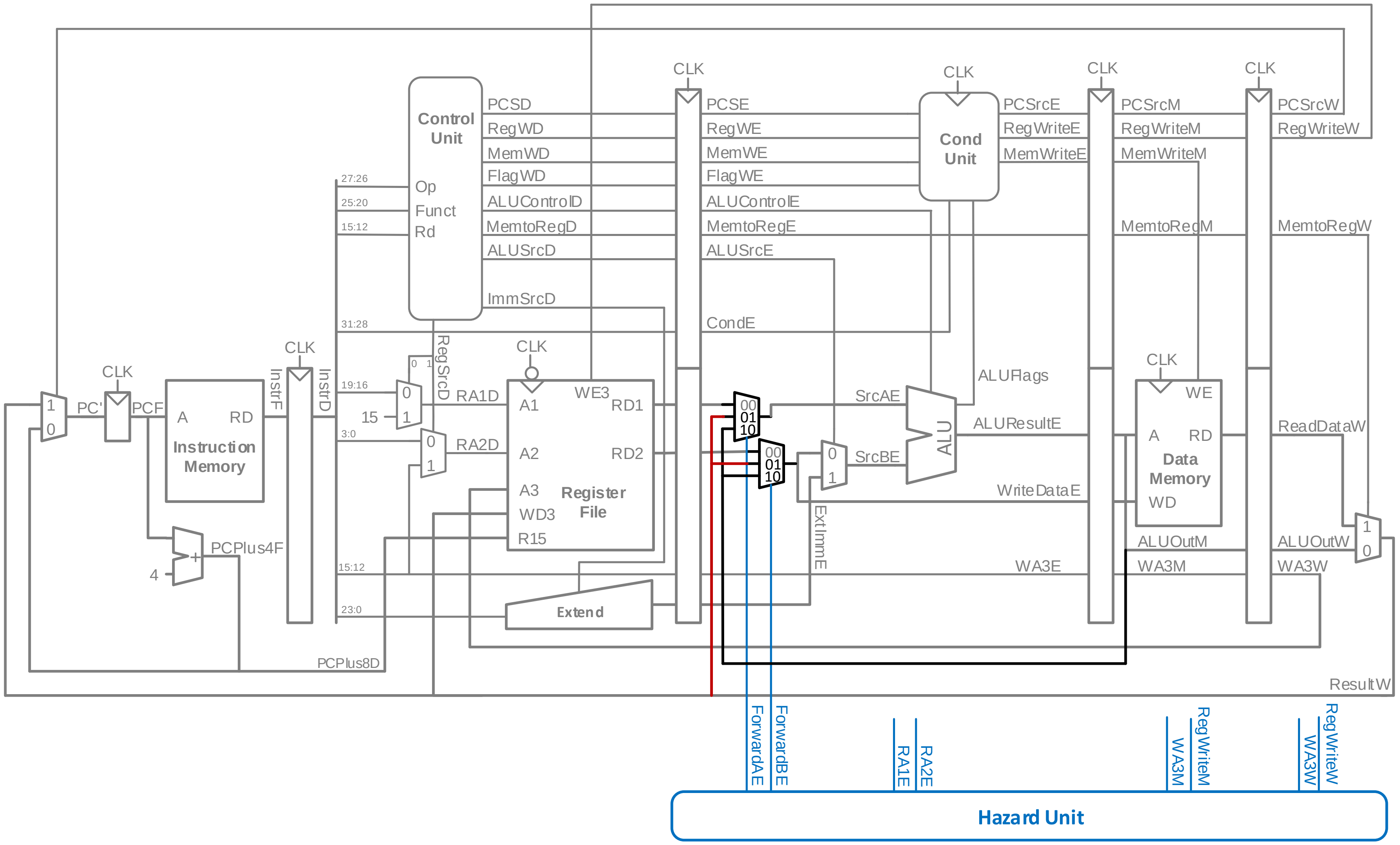


- Check if register read by the instruction which is currently in Execute stage matches register written by the instruction which is currently in Memory or Writeback stage
- If so, forward result

Data Forwarding …

- Take the result from the earliest stage / point (latest in the code) that it exists in any of the later (earlier in the code) pipeline state registers and forward it to the functional units (e.g., ALU) that need it that cycle
  - Take it from where it is present, pass it to where it is needed
- For ALU functional unit: the inputs can come from any pipeline register rather than just from E by
  - adding multiplexers to the inputs of the ALU
  - connecting ALUOutM and ResultW to SrcAE and SrcBE ALU mux inputs
  - adding the proper control hardware to control the new muxes
- Other functional units may need similar forwarding logic (e.g., the Data Memory)
- Can achieve a CPI of 1 even in the presence of data dependencies
- Data forwarding is also called Bypassing
  - The usual register file write and then read process for data access is bypassed
  - Register is still written, for future reads

Data Forwarding : Circuitry



Data Forwarding : Conditions

Source register for instruction in **Execute** stage matches destination register for instruction in **Memory** stage?

Match_1E_M = (RA1E == WA3M)
Match_2E_M = (RA2E == WA3M)

Each comparison requires a 4-bit comparator. Total 4 x 4-bit comparators

Source register for instruction in **Execute** stage matches destination register for instruction in **Writeback** stage?

Match_1E_W = (RA1E == WA3W)
Match_2E_W = (RA2E == WA3W)

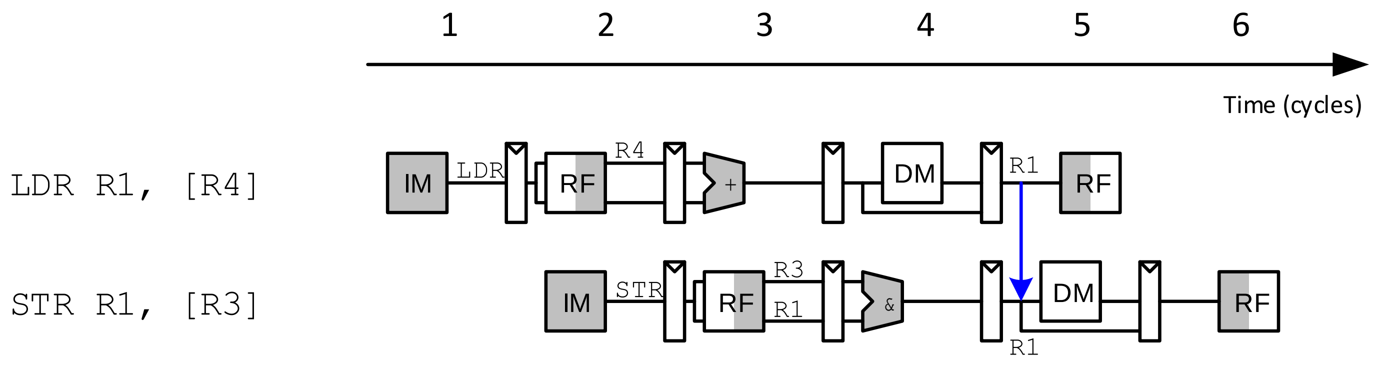
Forward only if the instruction in M / W really writes to a register. `LDR` / DP with condition not true, `B`, `STR` etc. do not write to any register

If it matches, forward result:

```
if          (Match_1E_M &  RegWriteM)    ForwardAE = 10;
else if     (Match_1E_W & RegWriteW)     ForwardAE = 01;
else                                     ForwardAE = 00;
```

ForwardBE same but with Match2E

Data Forwarding (Memory-Memory copy)



- Check if the register used in Memory stage (i.e., *Rd* of `STR`) by the `STR` instruction matches register written by `LDR` in Writeback stage (i.e., *Rd* of `LDR`)
- If so, forward result

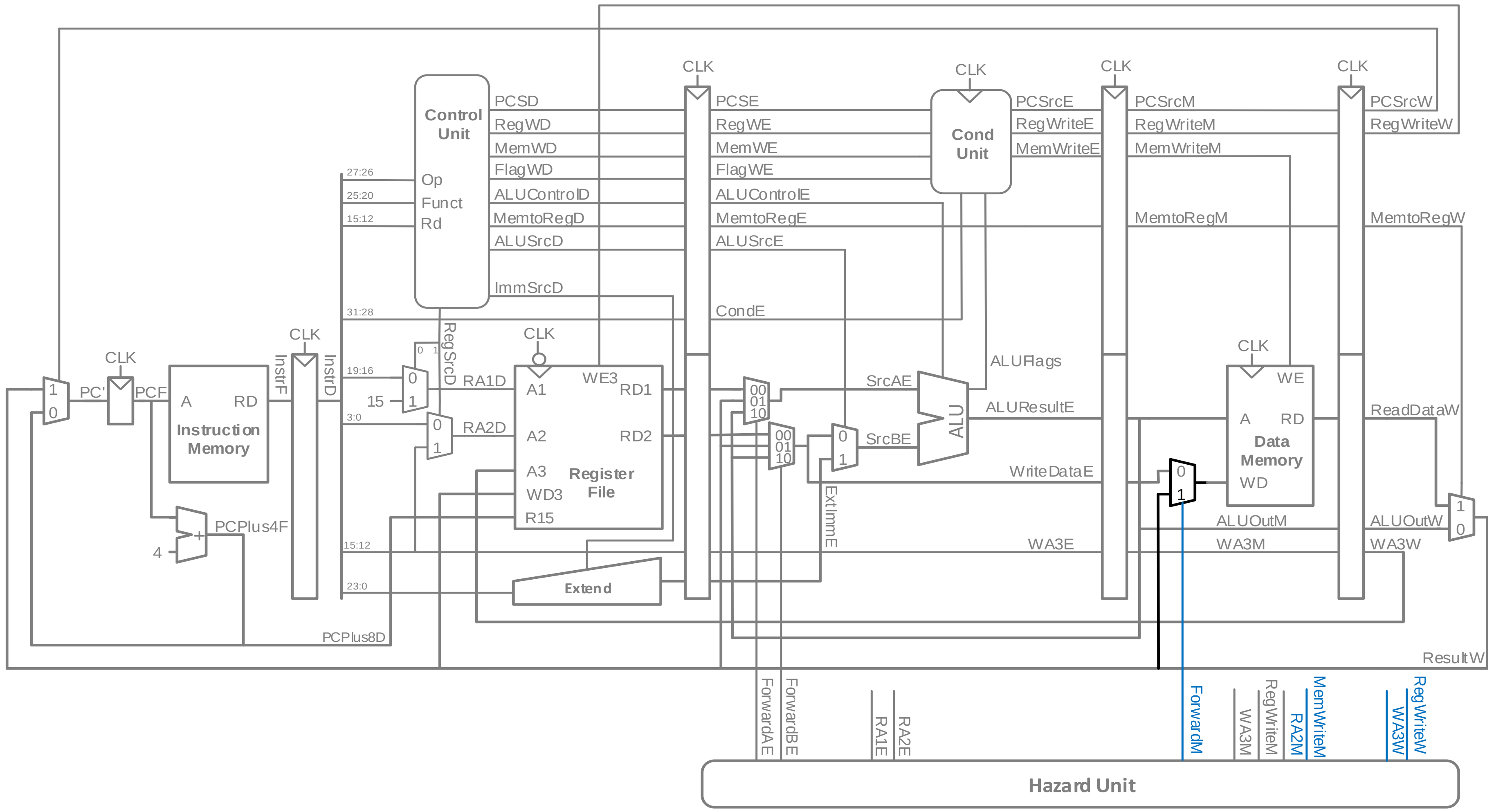
`STR` writes [RA2] into the memory

Check if M has `STR` and W has `LDR`

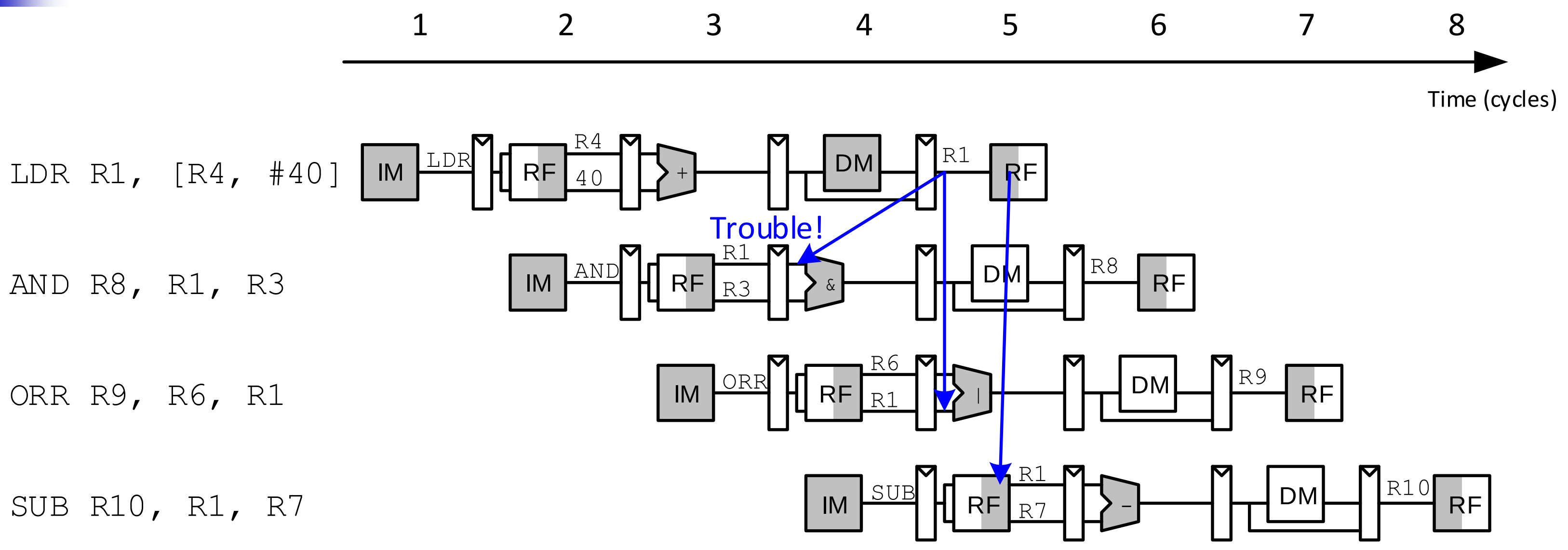
ForwardM = (RA2M == WA3W) & MemWriteM & MemtoRegW & RegWriteW

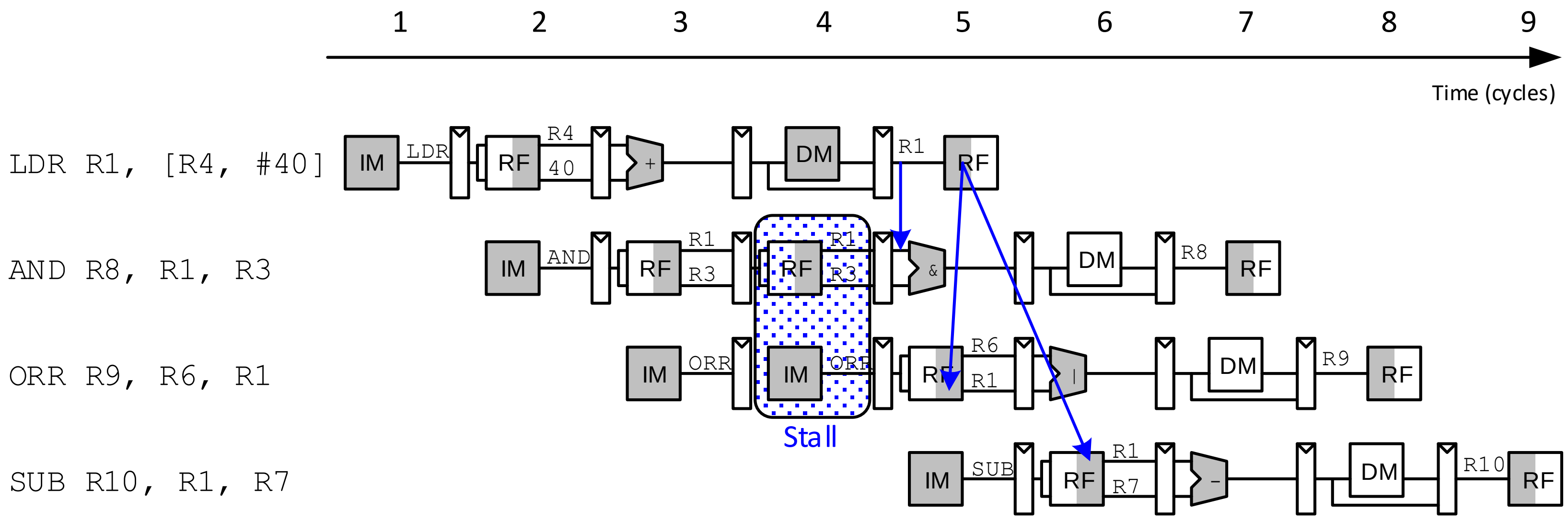
Ensure that the `LDR` is indeed executed

Data Forwarding : Circuitry (Mem-Mem copy)

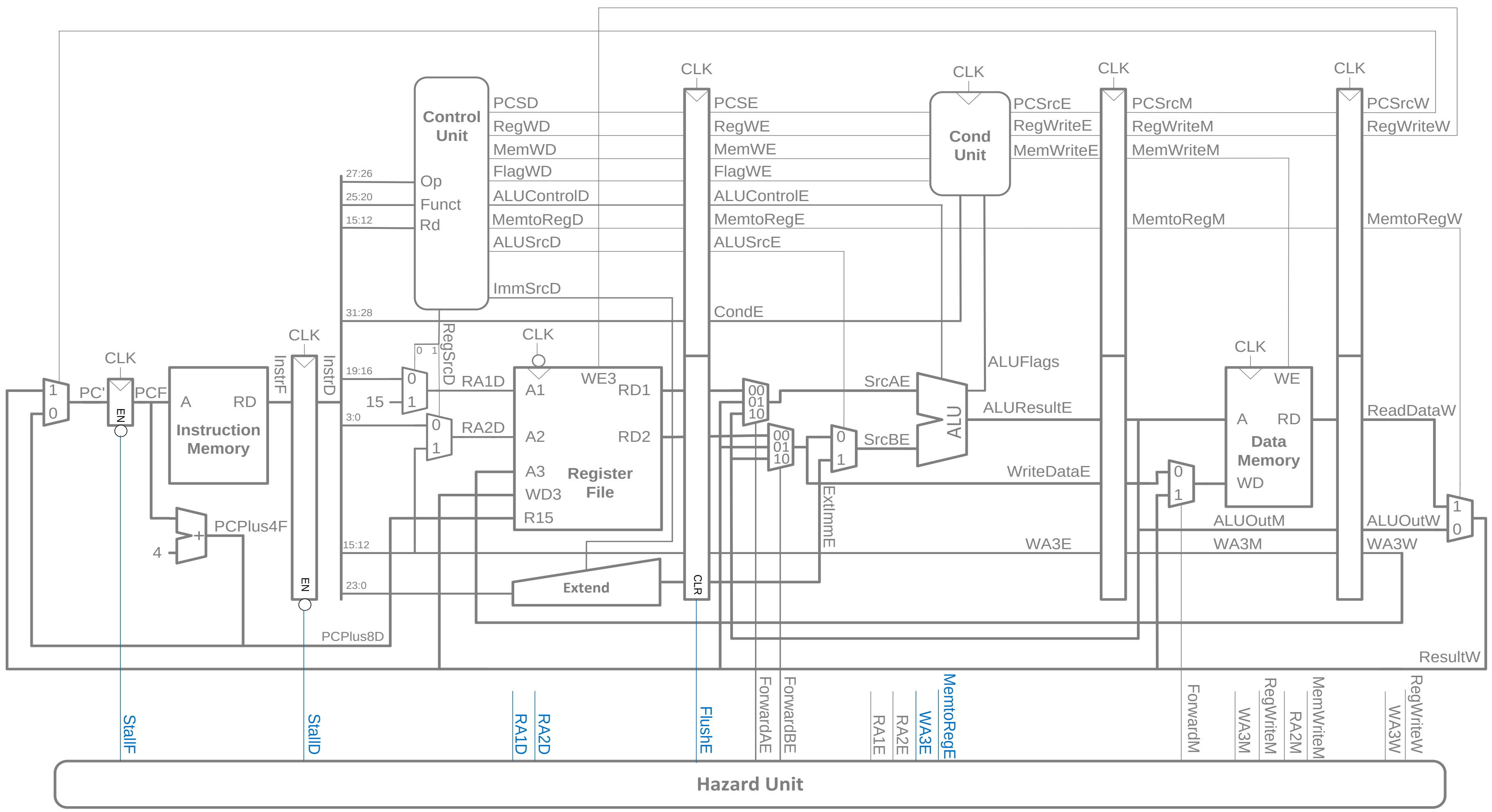


Load and Use Hazard : Stalling





Load and Use Hazard : Stalling Circuitry



Load and Use Hazard : Condition

- Is either source register of instruction in in the Decode stage the same as the destination of the instruction which is currently in the Execute stage?

  Match_12D_E = (RA1D == WA3E) || (RA2D == WA3E)

- Is a `LDR` in the Execute stage AND Match_12D_E?

  ldrstall = Match_12D_E & MemtoRegE & RegWriteE

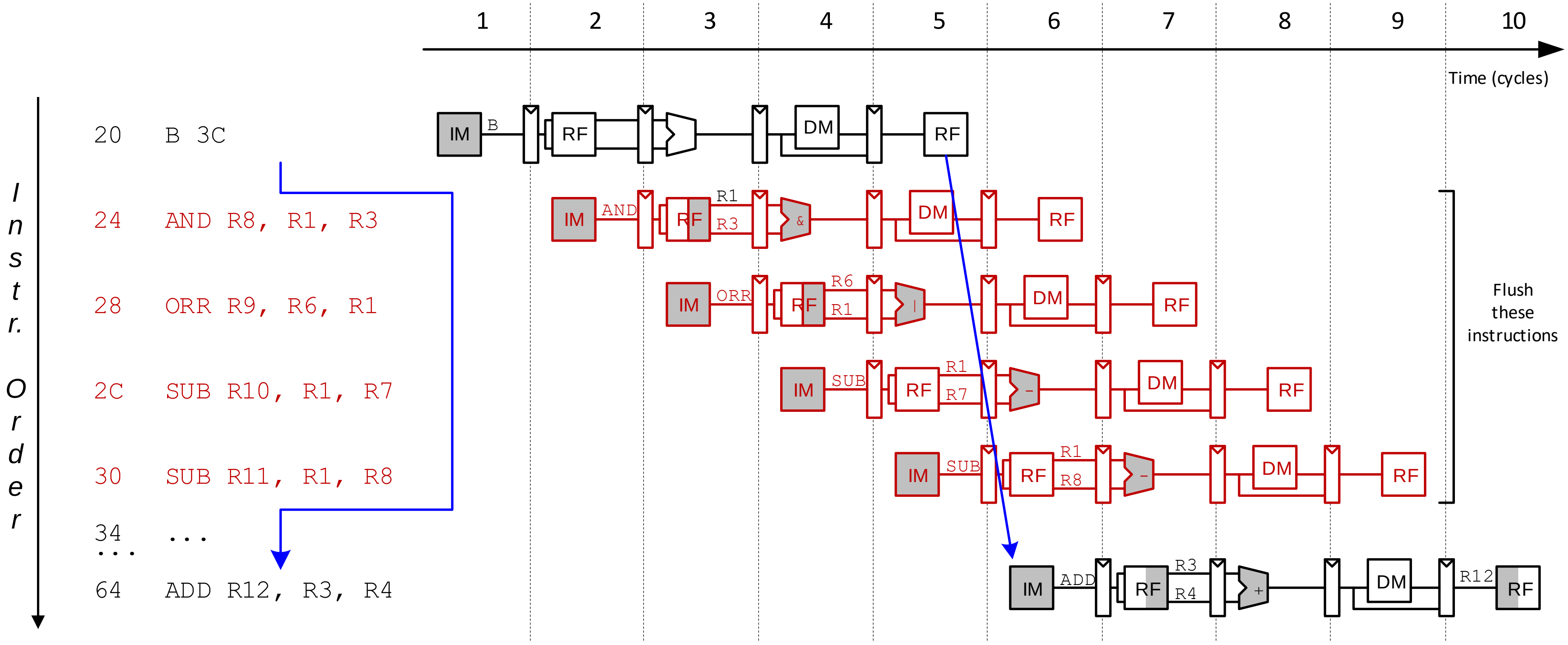
  StallF = StallD = FlushE = ldrstall

- After the 1 cycle stall, the forwarding circuitry (W to E) will take care of delivering the correct data

- `MOV` does not use RA1. `B`, `LDR`, `DP Imm` doesn't use RA2. Mem-mem copy forwarding takes care of `STR` following `LDR`. Stalls can be avoided in these cases by adding appropriate exception conditions (Omitted here for simplicity)

Control Hazards

- Branch instructions, writes to R15 can cause control hazards



```
20   B 3C
24   AND R8, R1, R3
28   ORR R9, R6, R1
2C   SUB R10, R1, R7
30   SUB R11, R1, R8
34   ...
...
64   ADD R12, R3, R4
```

- Compile time NOPs or rearranging the code* can help, but
  - NOPs reduce performance, even when the branch is not taken
  - Increases the code size and makes the code microarchitecture dependent

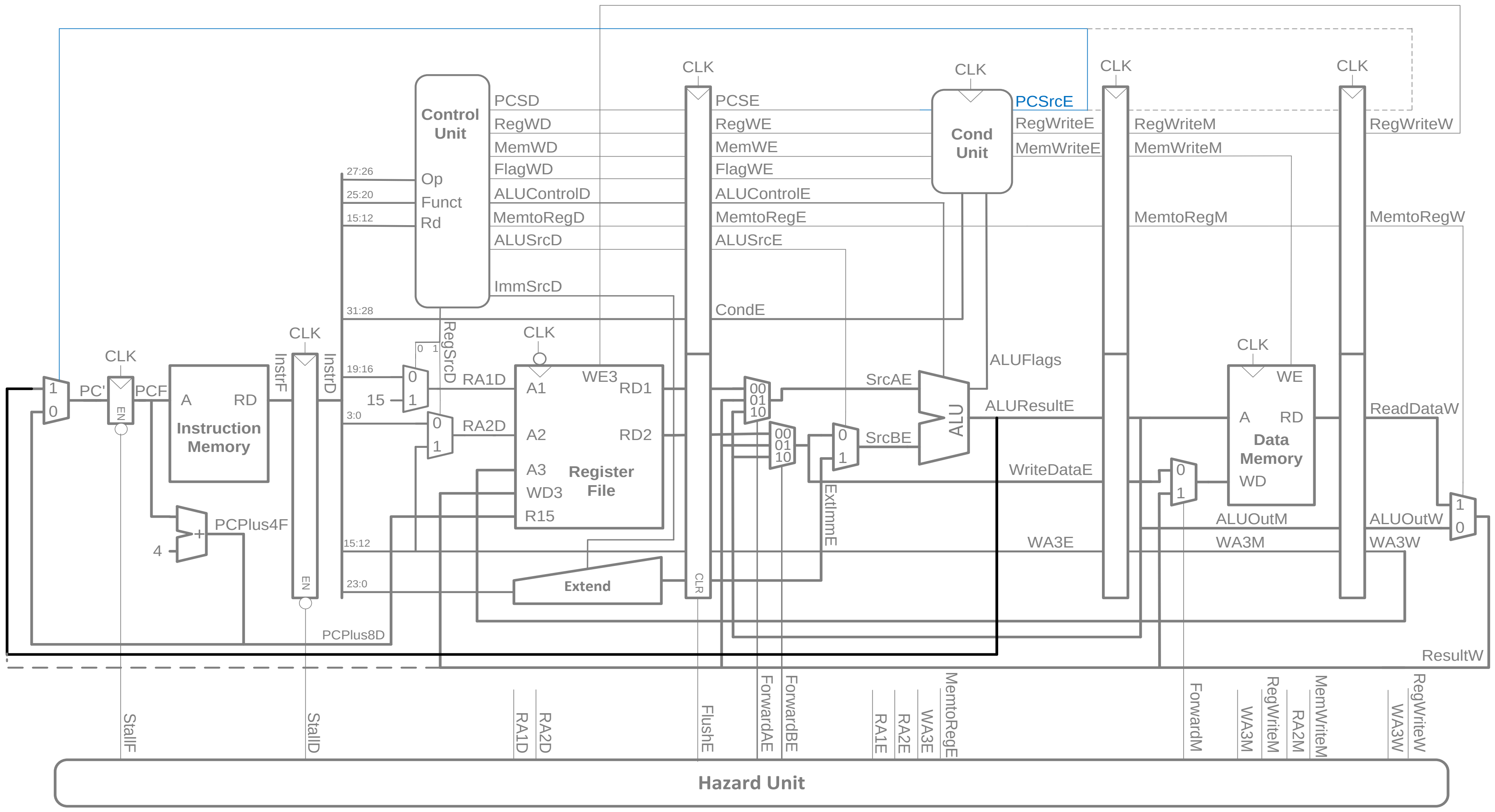
* The instruction(s) immediately following the branch should be "safe instruction(s)", which are executed irrespective of whether the branch is taken or not (provided it is specified so in the architecture and supported by the compiler). These instructions should hopefully be useful; and shouldn't affect the correctness even if they are not. Google for **delay slot**

Control Hazards

BTA = Branch Target Address

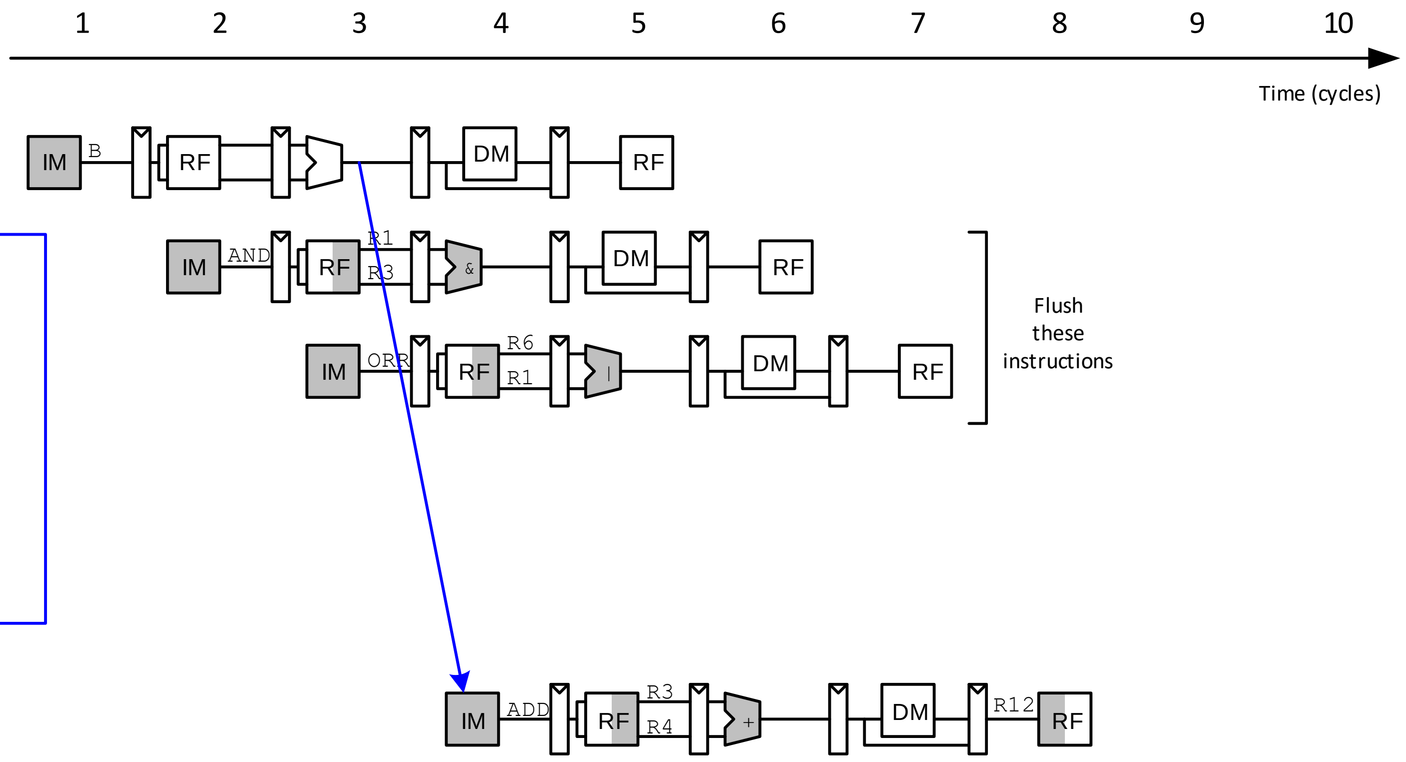
- Possible approaches
  - Stall until branch decision and BTA are available (impacts CPI)
  - Delay decision (delay slot - requires compiler support)
  - Early BTA - move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Branch prediction - predict and hope for the best !
  - Fetch and execute from both branches (if BTA is known)
  - Predicated/conditional execution, fine grained multithreading,…
- Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards
- Early BTA
  - Determine BTA in Execute stage
  - For now, ignoring the case of `LDR PC`, … (`LDR` with `PC` as destination)
  - Branch misprediction penalty = 2 cycles
  - Could increase the critical path delay ☹

Pipelined Processor with Early BTA



Control Hazards with Early BTA

```
20   B 3C

24   AND R8, R1, R3

28   ORR R9, R6, R1

2C   SUB R10, R1, R7

30   SUB R11, R1, R8

34   ...
...
64   ADD R12, R3, R4
```

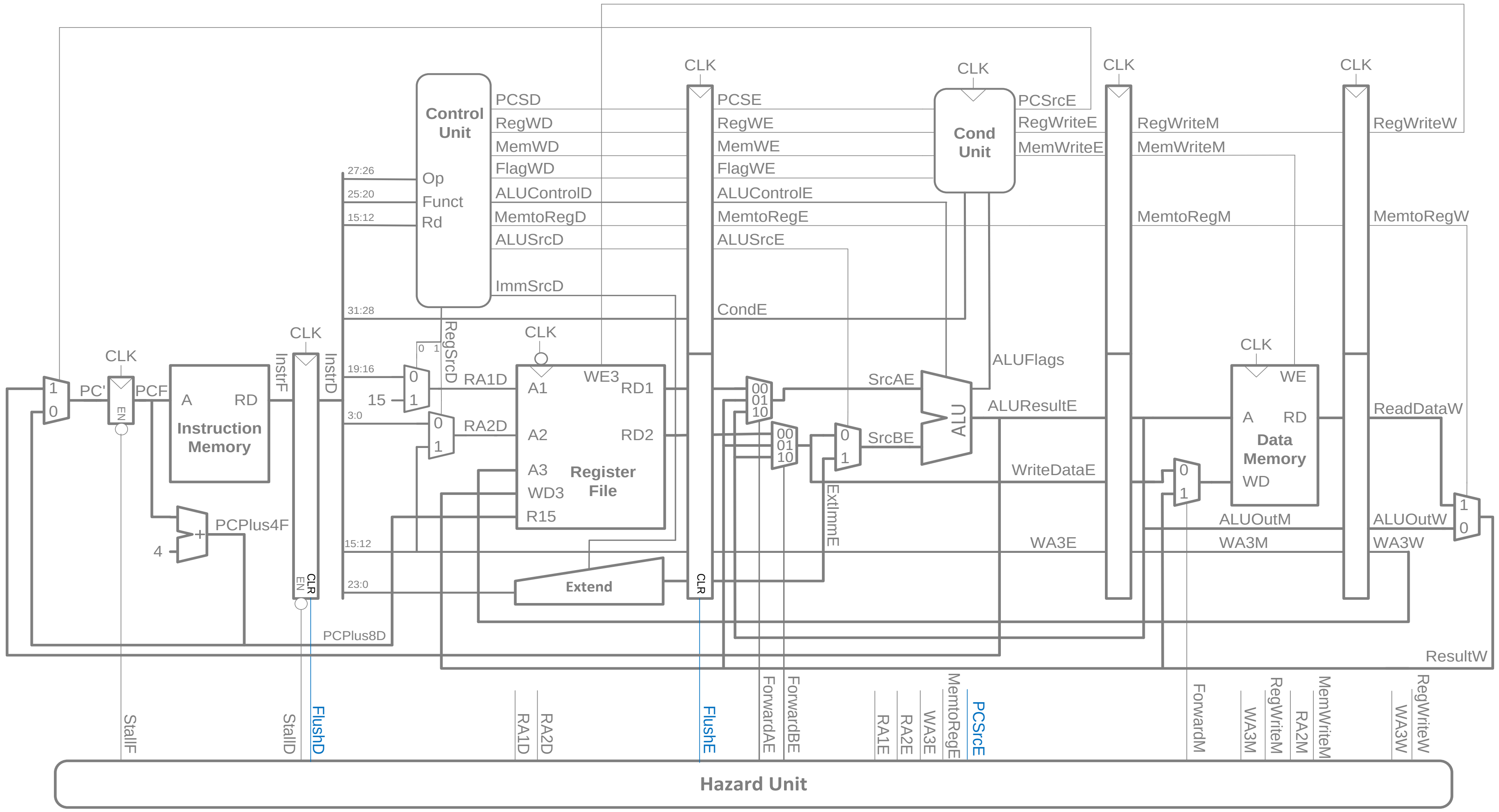


- If PCSrcE is true, flush the two earlier stages to quash the two following instructions

FlushD = FlushE = PCSrcE

FlushE (effective) = ldrstall (load and use) || PCSrcE (control)

Control Hazards : Flushing Circuitry



Stall vs Flush

- Stall involves deasserting the write control of the pipeline register
- Flush involves asserting the 'clear' of the pipeline register
  - In practice, clear only those bits which can cause a change in state (PCSE, MemWE, RegWE, FlagWE is sufficient for E. InstrD for D)
- If any stage is stalled, (i) the previous stages should also be stalled, and (ii) the following stage should be flushed (think load and use)
  - If (ii) is not done, the instruction in the latest stage being stalled will be executed 2 times – first using incorrect data and later using correct data (this is not ok if the instruction writes to flags or cause exceptions etc.)
  - Effectively, an NOP is inserted between the instruction in the latest stage being stalled and the instruction in the *following* pipeline stage (*earlier* in the code)
  - Instructions in the stages being stalled are bounced in place, while the instructions in the later stages of the pipeline (earlier in the code) are allowed to proceed to completion normally
- If any stage is flushed without stalling the previous stage, the instruction which is supposed to go into the flushed stage is replaced with an NOP (think branch)

Summary

- Almost all modern day processors use pipelining
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Potential speedup: a CPI of 1 and a fast CCT
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to "fill" pipeline and time to "drain" (flush) it can impact speedup for deep pipelines and short code runs
- Must detect and resolve hazards at compile time (by the compiler) or runtime (by the hardware)
  - Compile time NOPs reduce performance, increases code size and limits the portability of the code
  - Run time hazard detection requires additional hardware