CSCI 6461 Computer Systems Architecture

Pipelined processor

Processing Stages

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

- Fetch
- Decode
- Execute
- Memory
- Writeback

Simultaneous Execution

- They are similar to the discrete stages in the single-cycyle processor used to perform `ldr`
- Five instructions can execute simultaneously,
  - one in each stage.
- As we will see, this introduces significant performance efficiencies
  - but introduces the idea of a **hazard**

The 5 Stages

- **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.

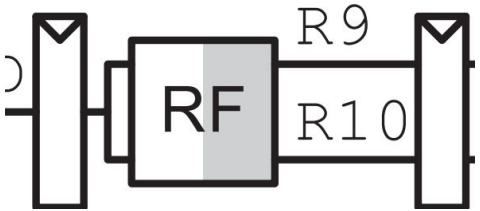
Simple Program

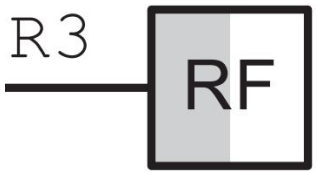
Our test case for pipeline analysis:

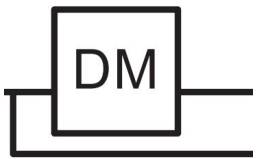
```
ldr r2, [ r0, #40 ]
add r3, r9, r10
sub r4, r1, r5
and r5, r12, r13
str r6, [ r1, #20 ]
orr r7, r11, #42
```

A note on shading

- Some resources, such as the register file (RF), can support read & write in the same phase.
  - shading of the RF on the right side means it is **read** in the phase
  - and shading on the left side means it is **written** in the phase
- No shading means idle

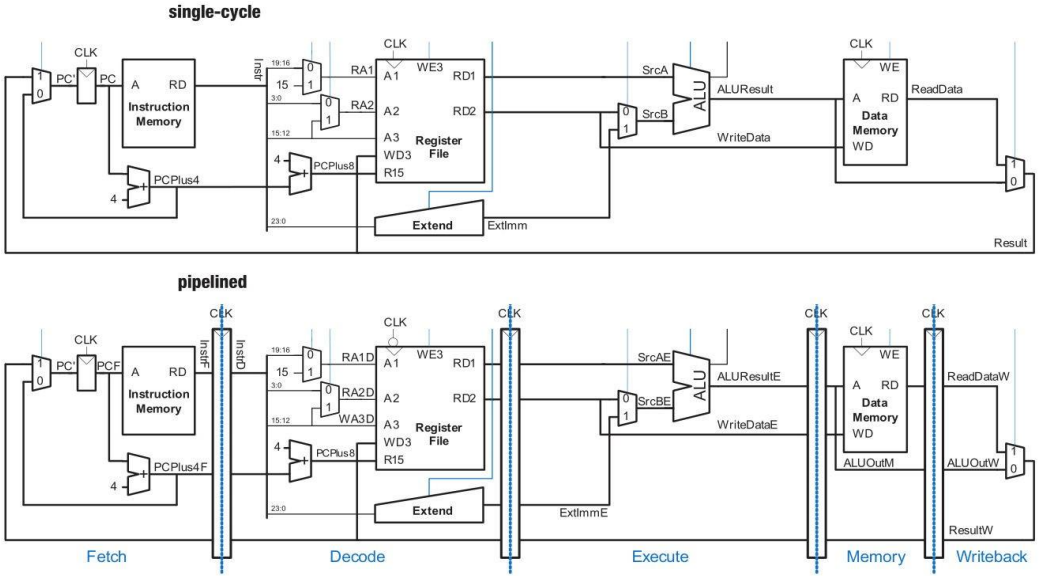






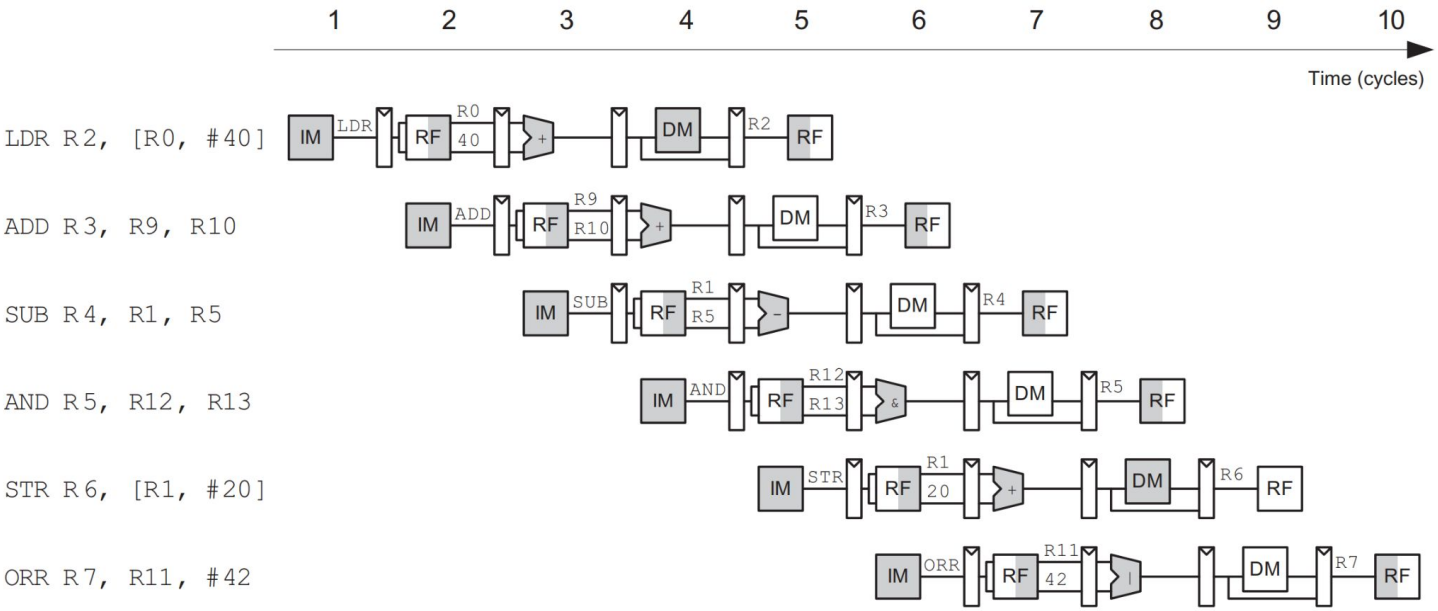
Chop the single-cycle into 5 stages

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

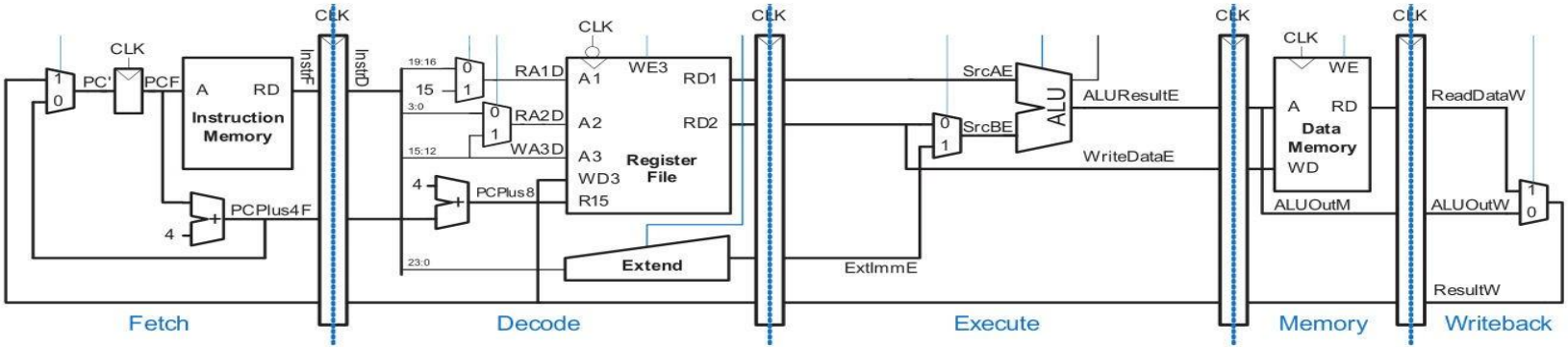
Theoretical Instruction Pipeline



- 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.

Register File Write Stage

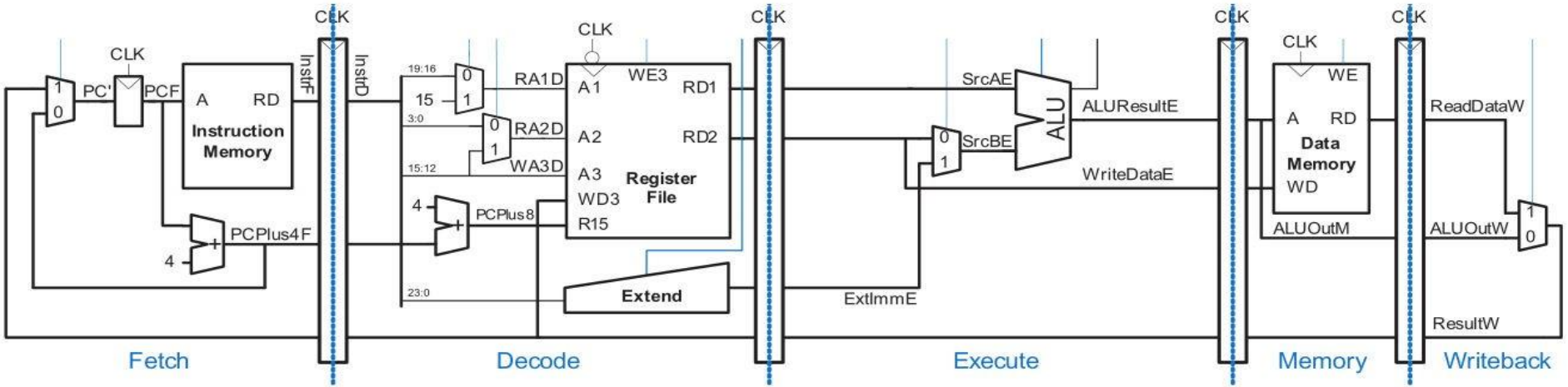
- The register file is read in the **Decode** stage and written in the **Writeback** stage.
- It is placed in the **Decode** stage, but the write data comes from the **Writeback** stage.



CLK cycle write and reads

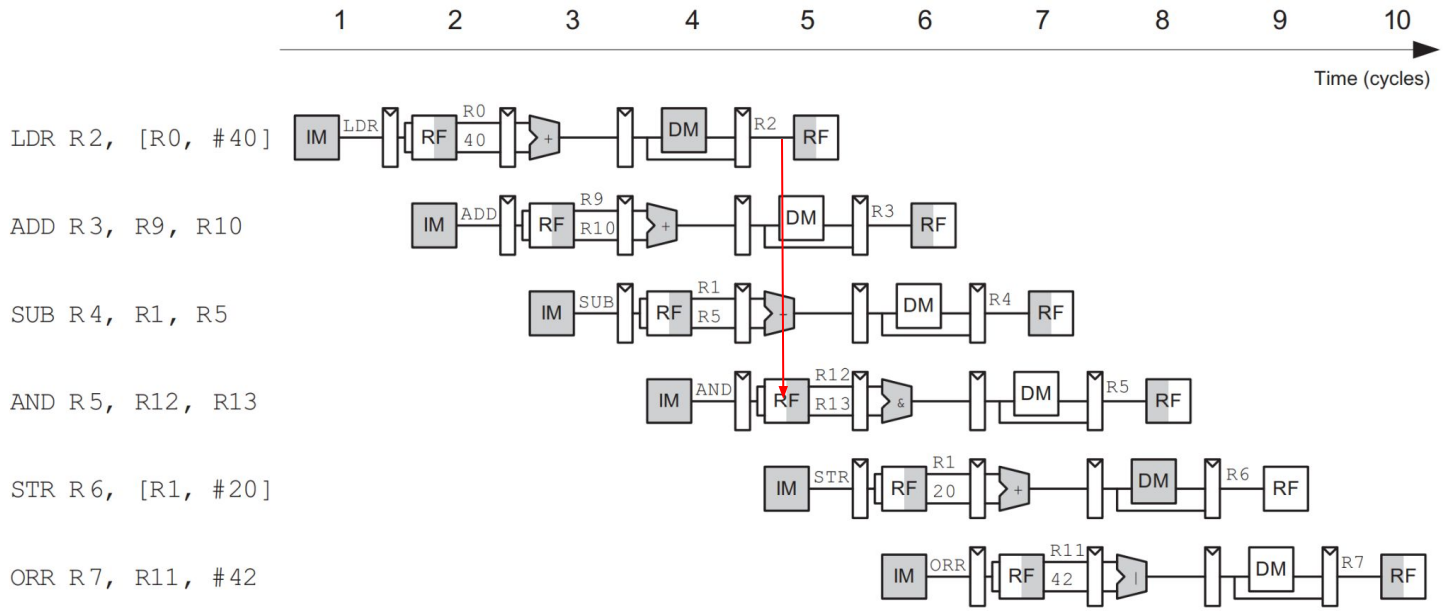
- The register file in the pipelined processor writes on the falling edge of CLK
  - so that it can write a result in the first half of a cycle and read that result in the second half of the cycle
  - for use in a subsequent instruction.
- A critical issue in pipelining is that all signals associated with a particular instruction must advance through the pipeline in unison

Datapath Stage Errors



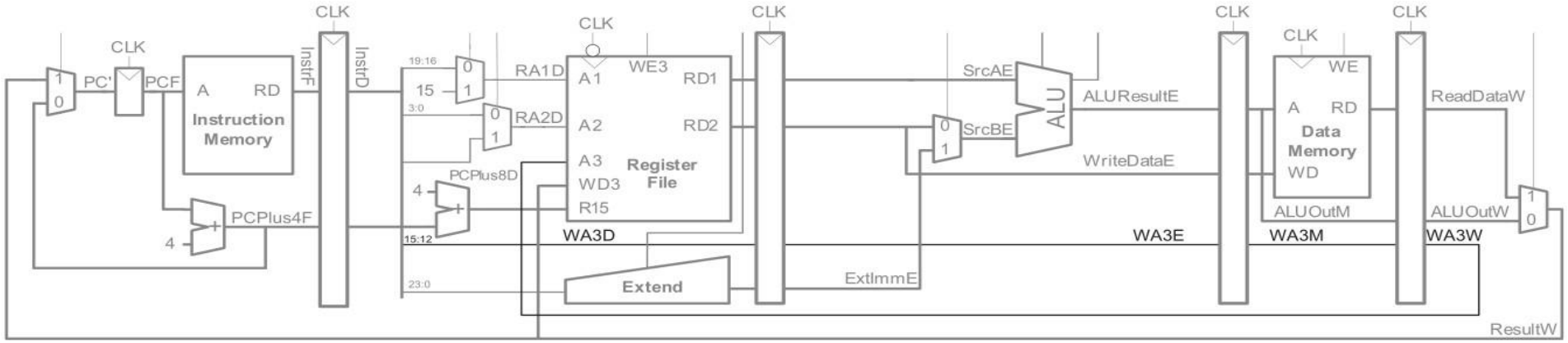
- Register file write logic operate in the **Writeback** stage.
- The data value comes from **ResultW** , a **Writeback** stage signal.
- But the write address comes from $InstrD_{15:12}$ (WA3D), which is a **Decode** stage signal.

Datapath Stage Errors



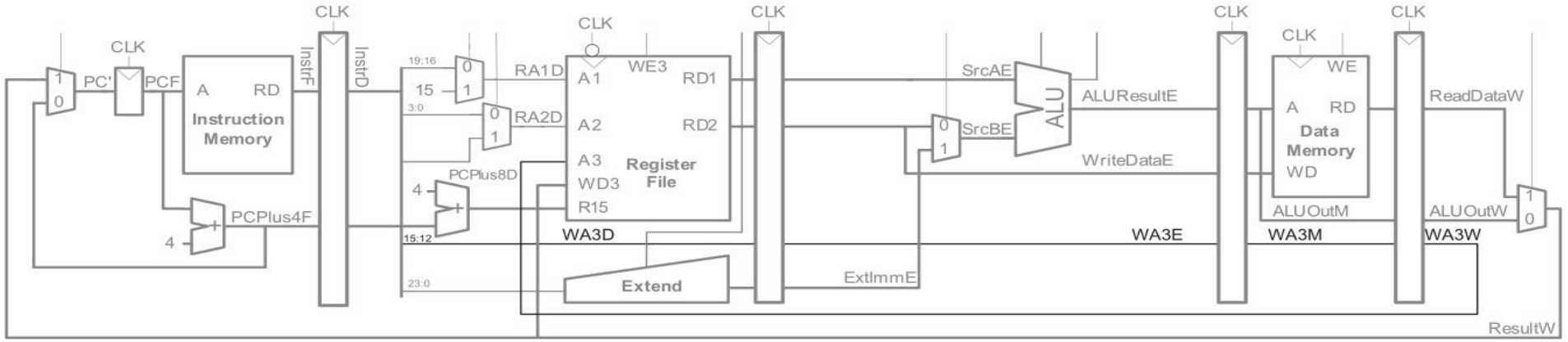
During cycle 5, the result of the `ldr` instruction would be incorrectly written to `r5` (`and r5, r12, r13`) rather than `r2`.

Corrected **WA3D** pipelined datapath



- The **WA3D** signal is now pipelined along through the Execution, Memory, and Writeback stages (**WA3E**, **WA3M**, **WA3W**), 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.

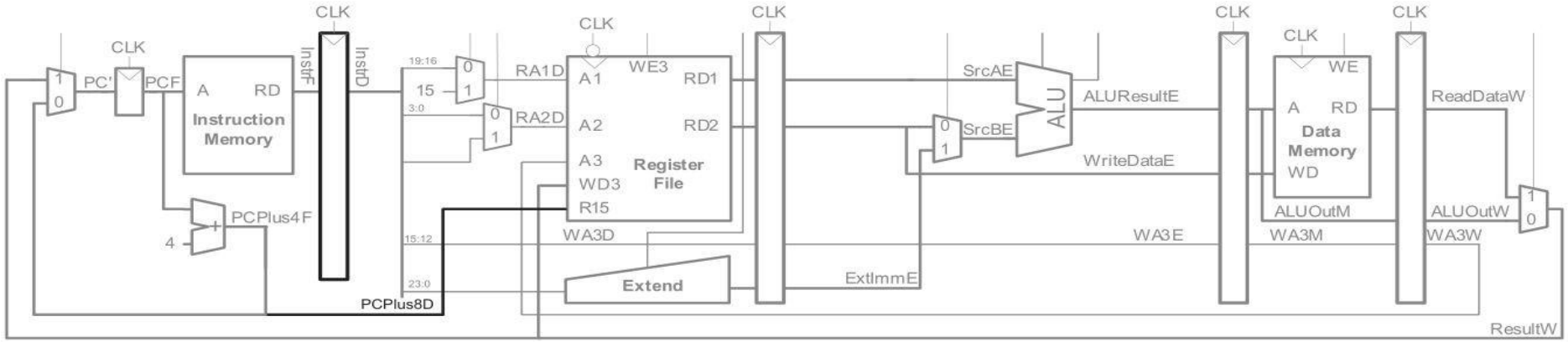
Incrementing the PC



- 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.

Optimizing PC incrementation

- Sending this signal ahead saves the pipeline register and second adder.
- Optimized PC logic eliminating a register and adder:



PC control hazard

PC' logic is also problematic:

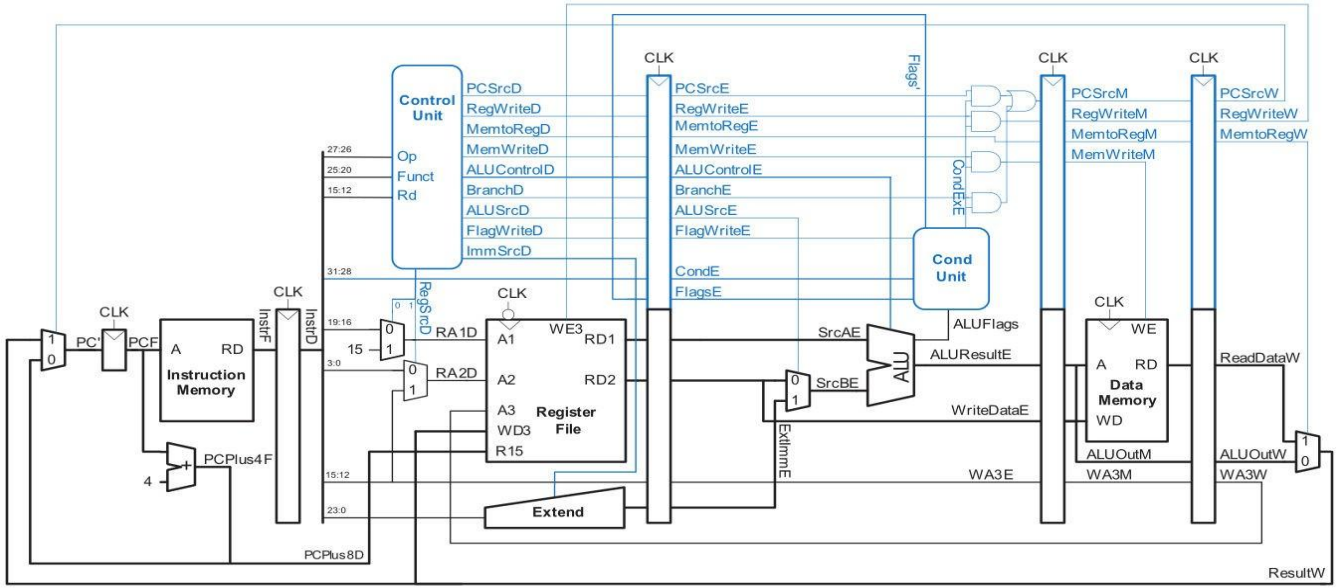
- it might be updated with a Fetch
- or a Writeback stage signal
  - **PCPlus4F** (eg, sequence/branch)
  - **ResultW** (eg, `pop {pc}`)
- This control hazard will be fixed in the next section

Pipelined Control

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** for 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

Consequences of Pipelining in 5 Stages

Hazard Types

- 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.
- Let's look at a data hazard first

Data Hazards

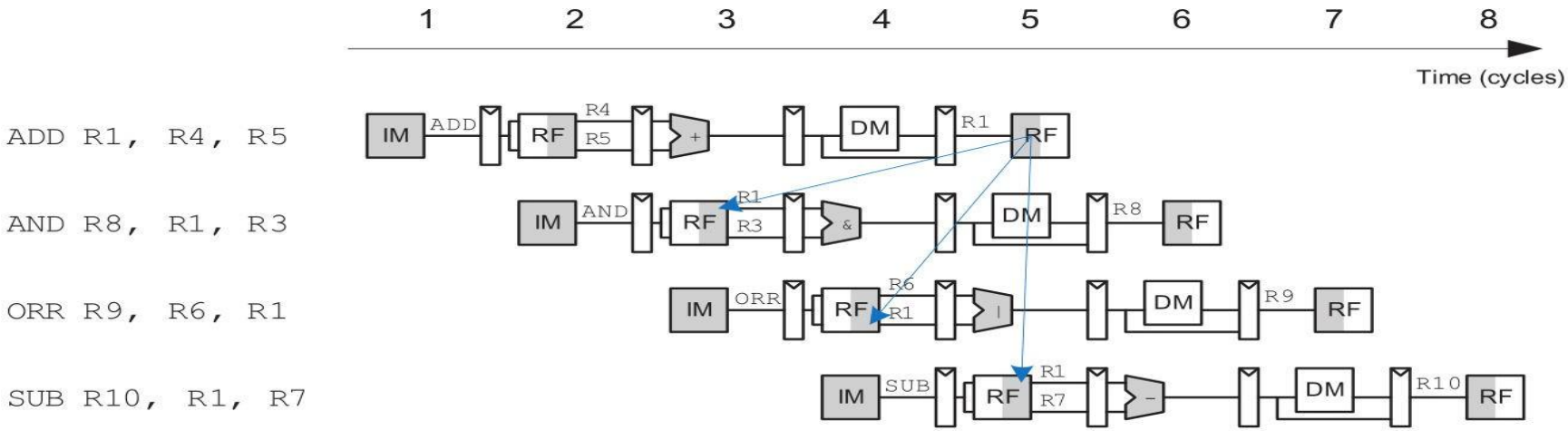
RAW and Stalls

Data Hazards

There are several kinds of data hazards to be aware of:

- **Read after Write** (RAW) occurs when the value produced by an instruction is required by a subsequent instruction
- **Write after Write** (WAW) occur when the output register of an instruction is used for write after written by previous instruction

RAW Hazard



- The `add` instruction writes a result into R1 in the first half of cycle 5.
- The `and` instruction reads R1 on cycle 3, obtaining the wrong value.
- The `orr` instruction reads R1 in the second half of cycle 4, again, obtaining the wrong result
- What about the `sub` instruction?

Result Forwarding

- 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.

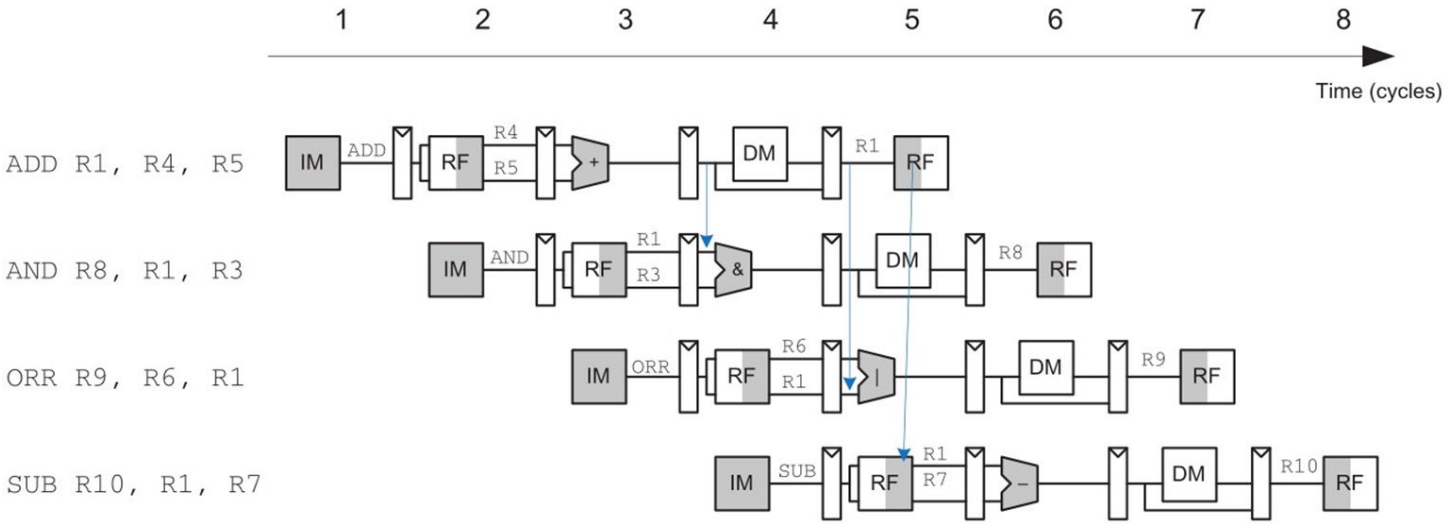
Hazard Unit

Stage forwarding & multiplexers

Hazard Units

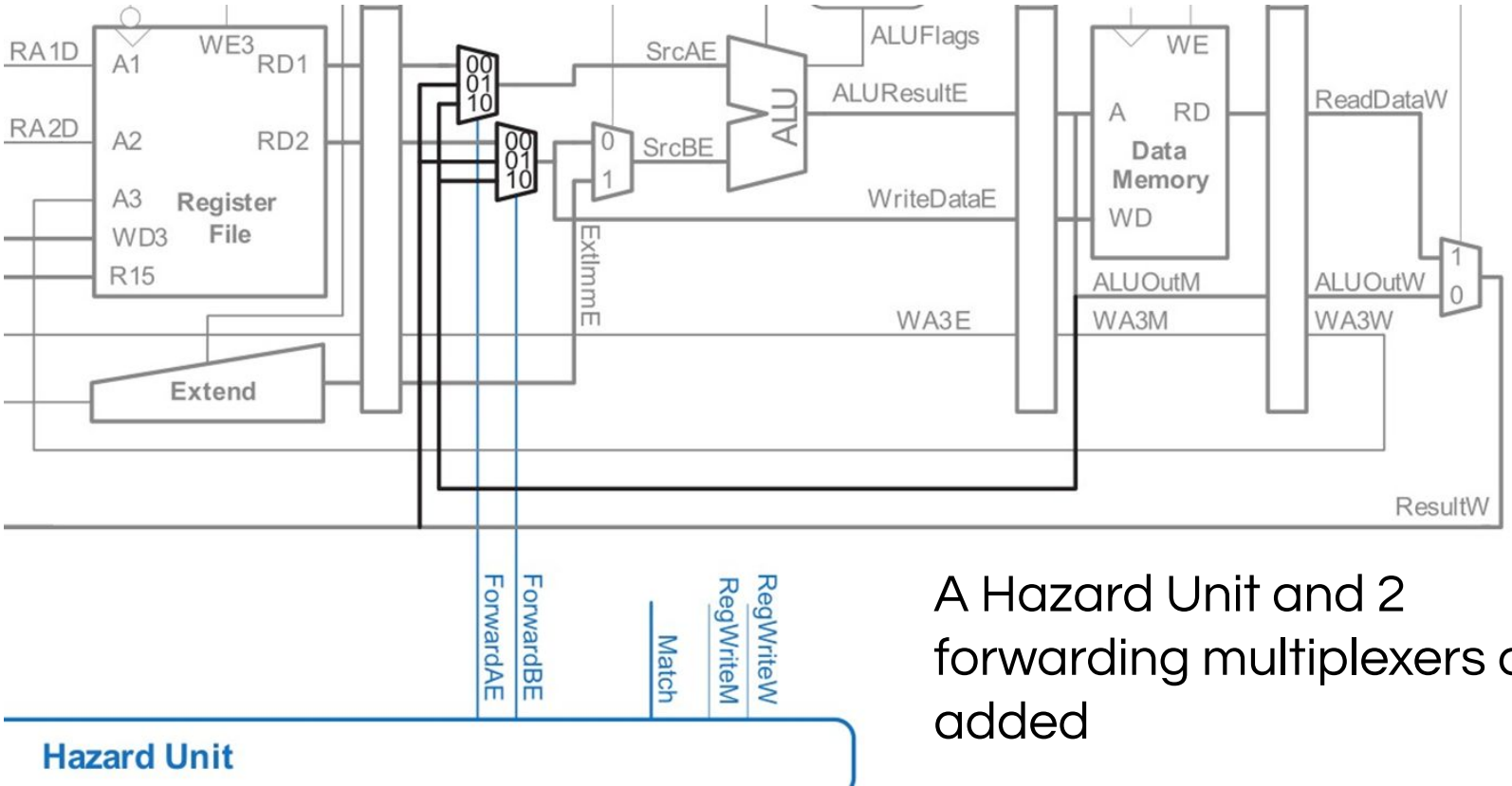
- We will enhance the pipelined processor with a **Hazard Unit**
- The HU detects hazards and handles them appropriately
  - so that the processor executes the program correctly
- The HU must correctly compute the signals to stage forward results

Add stage forwarding



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

Architecture with Hazard Unit



A Hazard Unit and 2 forwarding multiplexers are added

Hazard Unit

- 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:
  - For example: Match_1E_M = (RA1E == WA3M) means a read of A1 in the execute phase matches a write of A3 in the memory phase instruction ahead

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

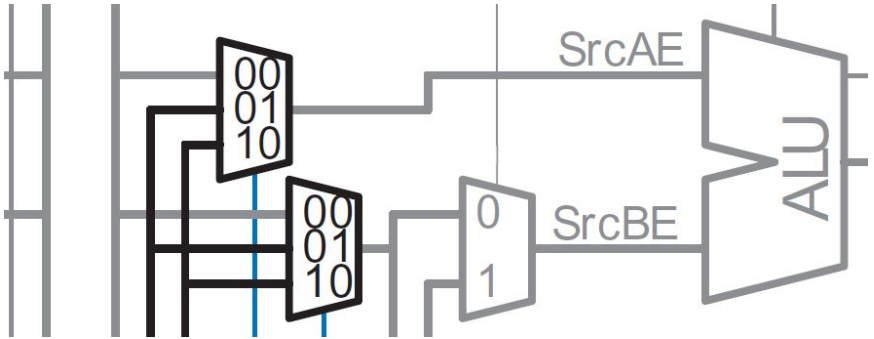
Hazard Unit

- 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.

Forwarding Logic

```
if (Match_1E_M && RegWriteM) // (RA1E == WA3M)
   ForwardAE = 10; // SrcAE = ALUOutM
else if (Match_1E_W && RegWriteW)
   ForwardAE = 01; // SrcAE = ResultW
else ForwardAE = 00; // SrcAE from regfile
```

The forwarding logic for **SrcBE** (**ForwardBE**) is identical except that it checks **Match_2E** .



`Match_1E_W = (RA1E == WA3W)`

- Consider: `Match_1E_W = (RA1E == WA3W)`
- This hazard arises when we have a **RA1** (**19:16**) in execute that matches **WA3** in the write phase.
  - `add r5, r4, r2 // WA3W`
  - `sub r7, r8, r9 // Other instr`
  - `add r6, r5, r1 // RA2E`

`Match_2E_M = (RA2E == WA3M)`

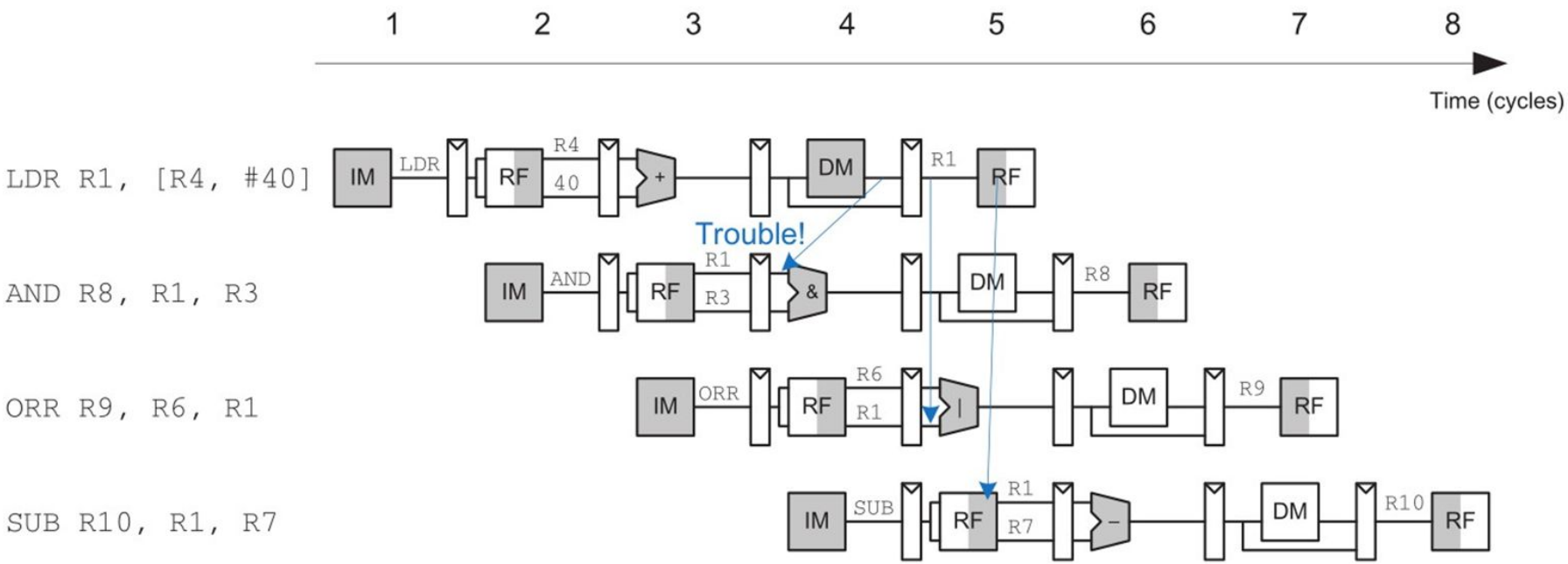
- Consider: `Match_2E_M = (RA2E == WA3M)`.
- This hazard arises when we have a **RA2 (3:0)** in execute that matches **WA3** in the memory phase.
  - `add r5, r4, r2 // WA3M`
  - `add r6, r4, r5 // RA2E`

This is the tightest RAW dependency the ARM pipeline can still resolve without stalling : "one-instruction-apart" case

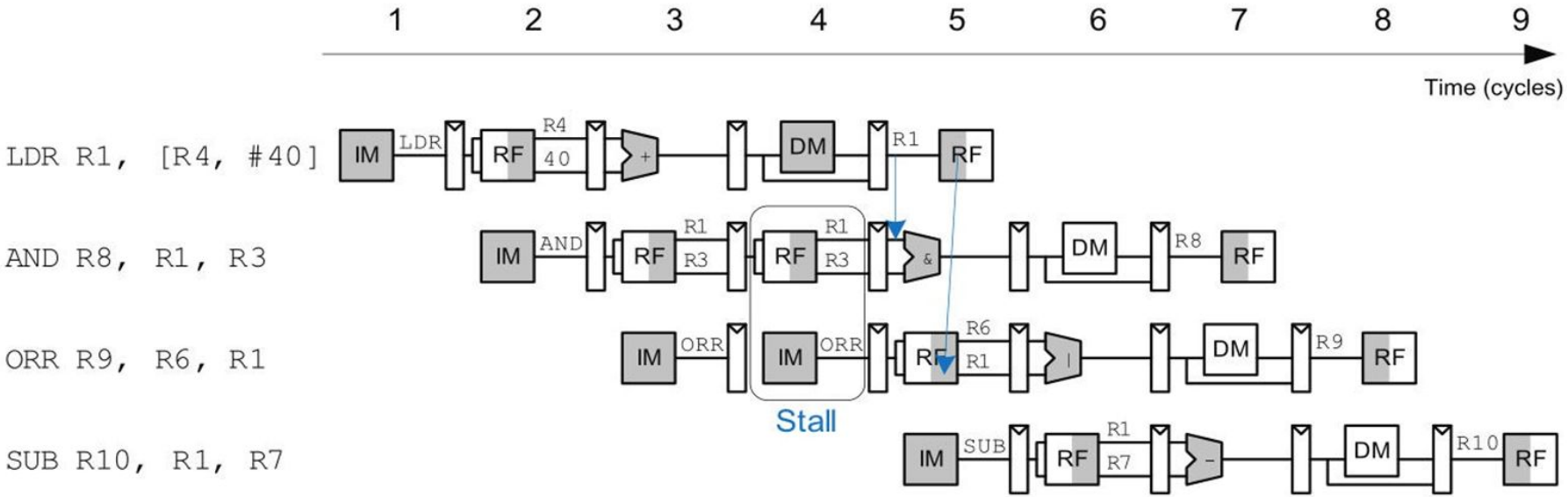
Trouble forwarding from `ldr`

- 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.

For example

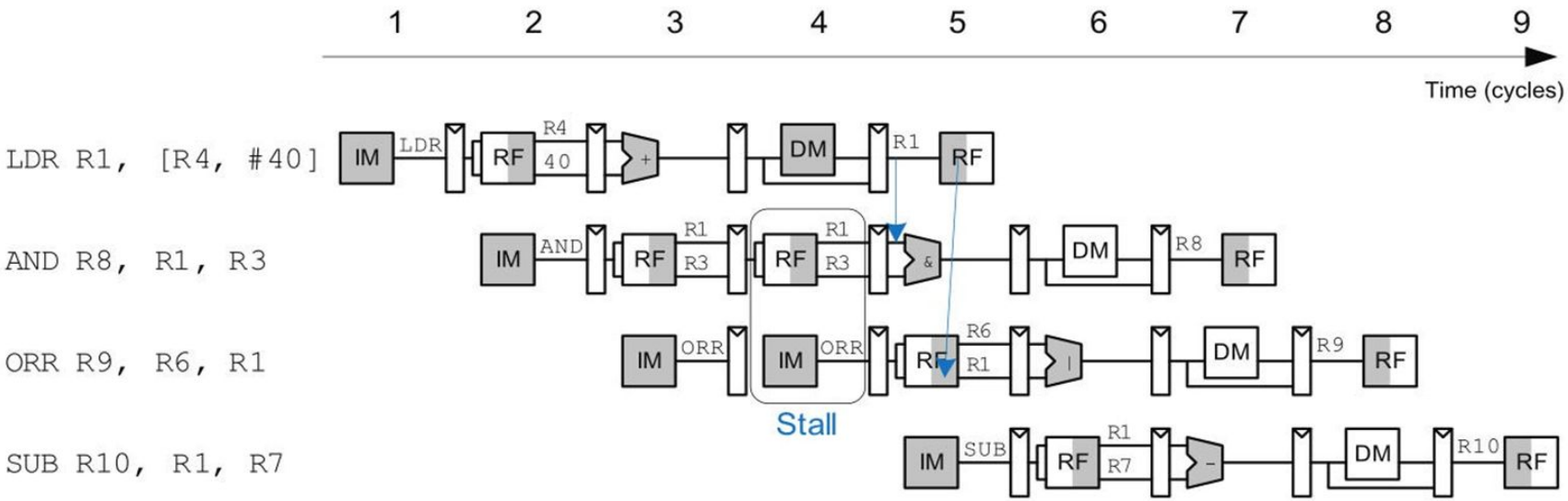

1
2
3
4
5
6
7
8
Time (cycles)
LDR R1, [R4, #40]
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
IM
LDR
RF
R4
40
+
DM
R1
RF
Trouble!
IM
AND
RF
R1
R3
&
DM
R8
RF
IM
ORR
RF
R6
R1
|
DM
R9
RF
IM
SUB
RF
R1
R7
−
DM
R10
RF

Pipeline Stalls



- `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.

Unused Stages



Execute stage is unused in cycle 4, the Memory stage is unused in cycle 5 and the Writeback stage is unused in cycle 6.

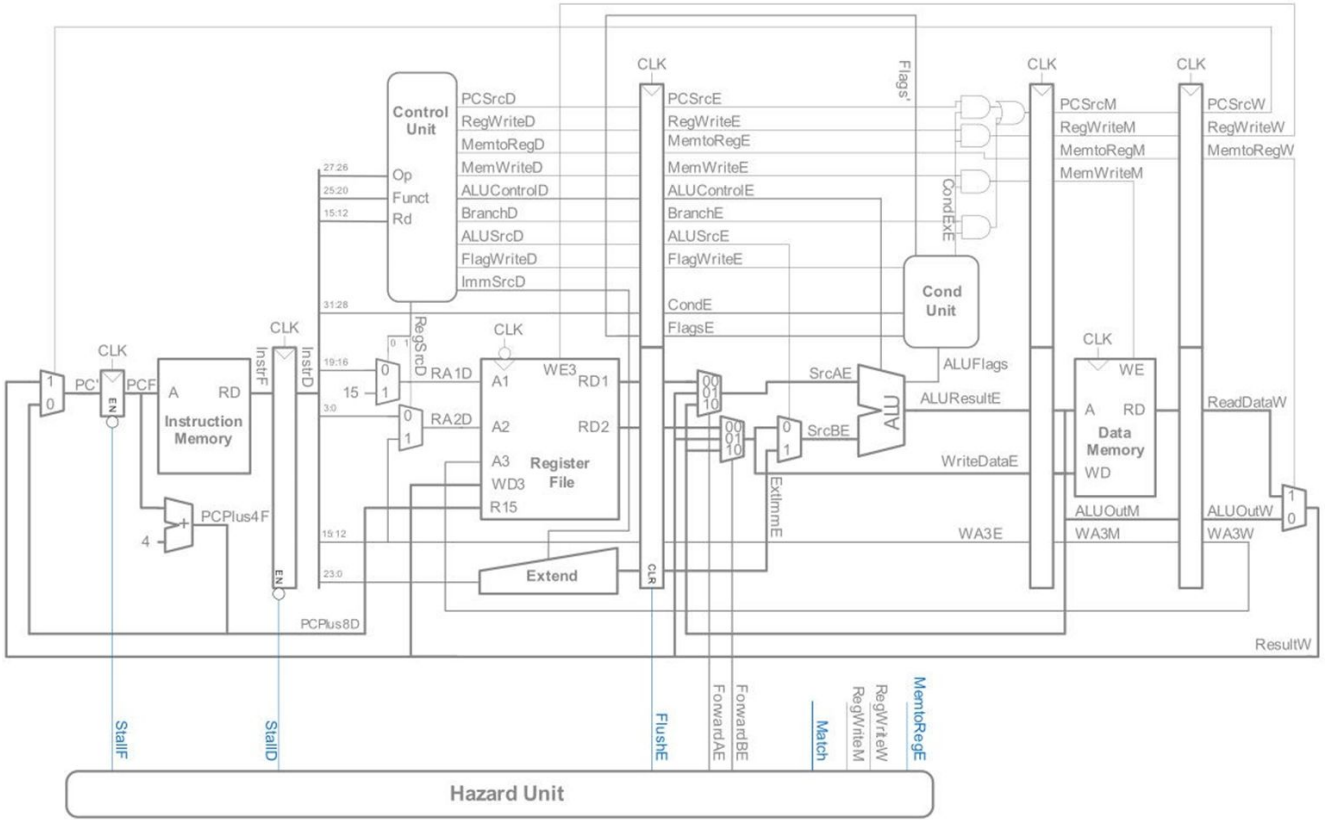
Bubbles

- This unused stage propagating through the pipeline is called a **bubble** .
- 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.

Stalls degrade performance

- 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.

Pipeline with stalls for `ldr`



`ldr` and Hazard Unit

- 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

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.

Stall Computation Logic

The logic to compute the stall and flush:

```
Match_12D_E = (RA1D == WA3E) || (RA2D == WA3E)
LDRstall = Match_12D_E && MemtoRegE
StallF = StallD = FlushE = LDRstall
```

Control Hazards

The consequences of **B**

Brancing instructions

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. (eg, `mov pc, lr` or `pop {pc}`)

B instruction

- 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.

PC changes in different phases

The program counter can be changed in different phases

For example:

- `bx lr` `//what phase?`
- `ldr pc, [sp, #-4]!` `//what phase?`

As you can see, these PC updates become apparent in different phases of the pipeline

Branch Predicition

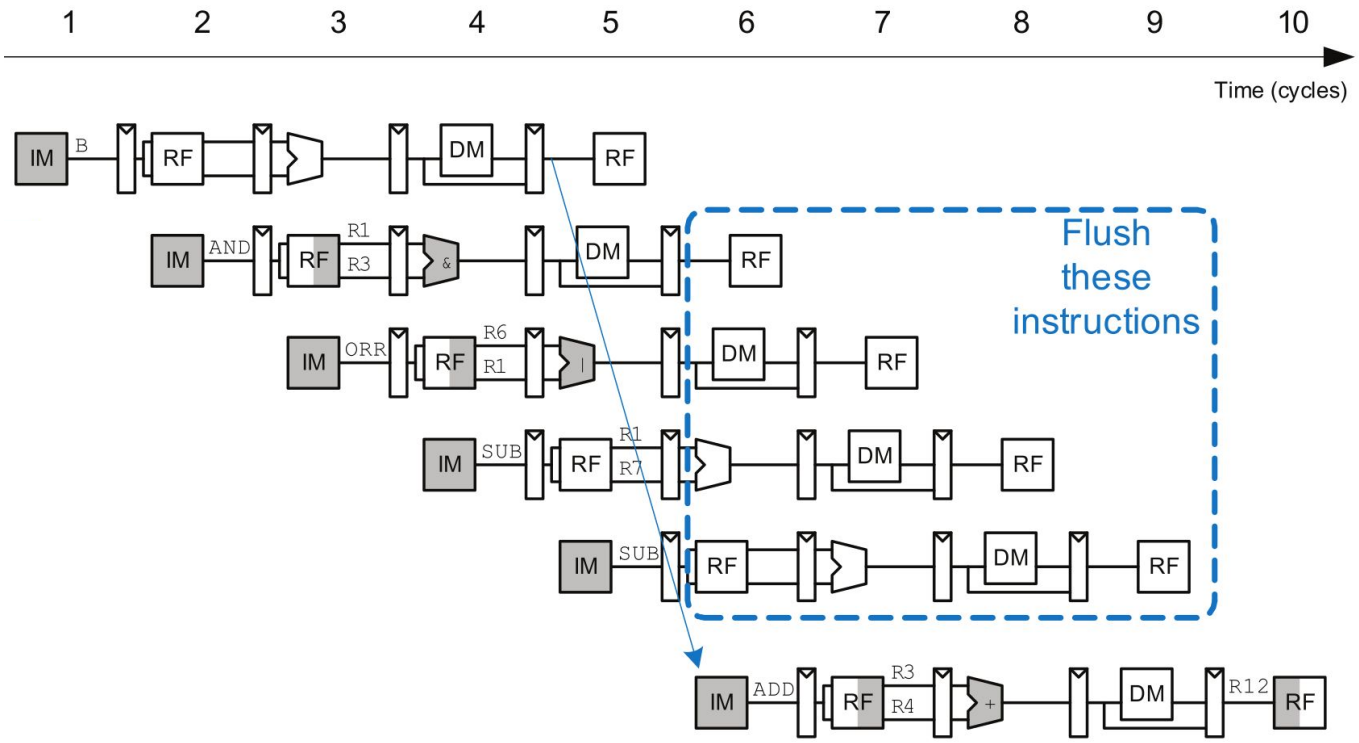
- An alternative is to predict whether the branch will be taken
  - and begin executing instructions based on the prediction.
- Once the branch decision is available, the processor can throw out the instructions
  - if the prediction was wrong.

Misprediction

- 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.

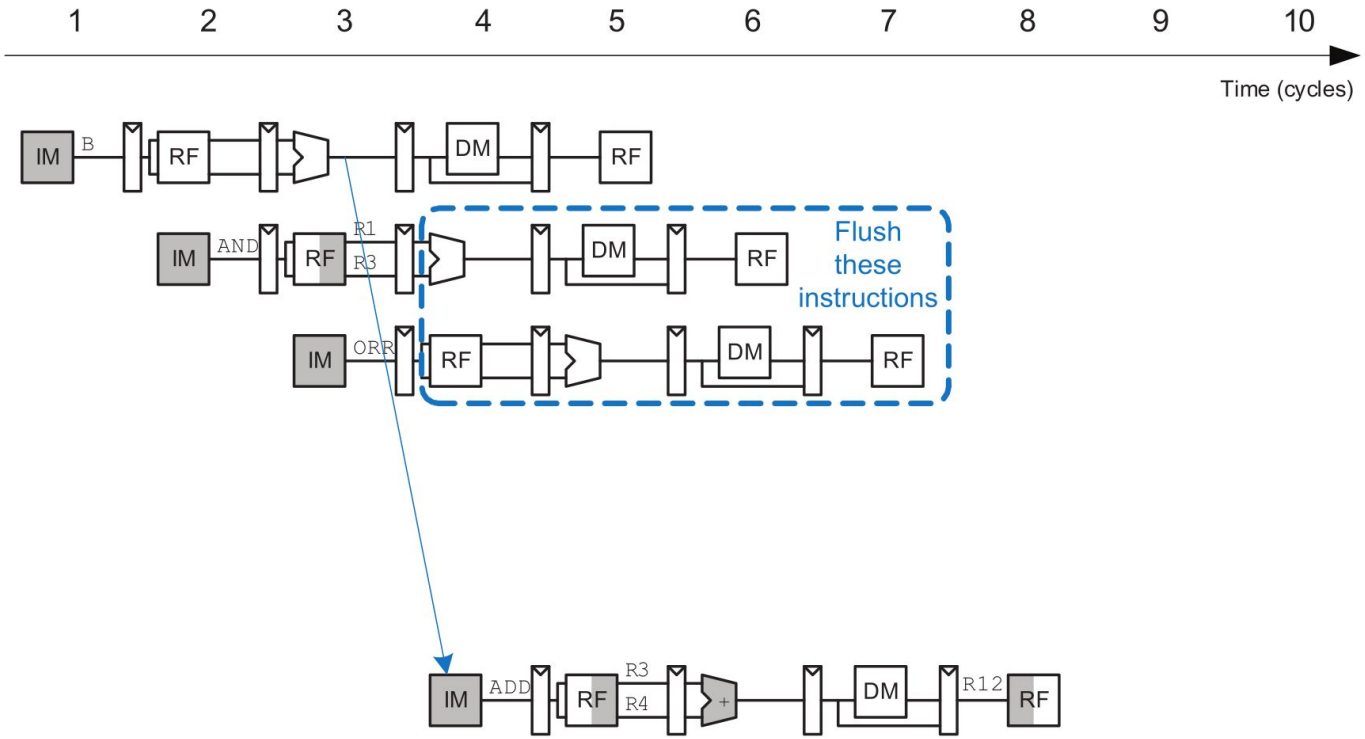
Flushing on a b

```
b end
and r8, r1, r3
orr r9, r6, r1
sub r10, r1, r7
sub r11, r1, r8
...
end:
  add r12, r3, r4
```

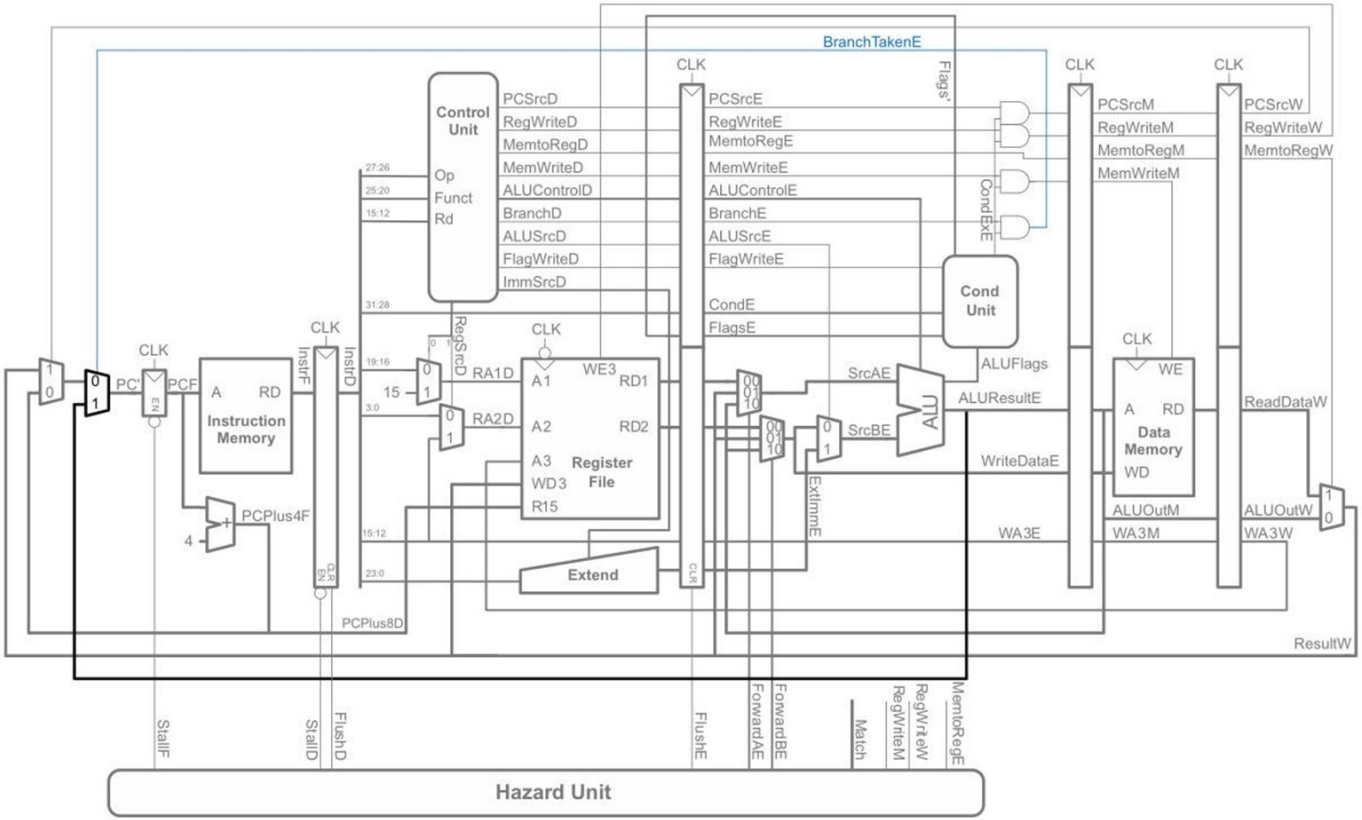


Execute Stage Decision

```
b end
and r8, r1, r3
orr r9, r6, r1
sub r10, r1, r7
sub r11, r1, r8
...
end:
  add r12, r3, r4
```



Branch Control Hazard added



Branch Multiplexer

- 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.

Branch Multiplexer

- 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.

Fetch stalled / Decode 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;
```

Reducing the Penalty

- Branches are very common,
- 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`

Reducing the Penalty

- **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.

Pipeline with Full Hazard