TED ÜNİVERSİTESİ

CMPE361 Computer Organization

Department of Computer Engineering

TED University- Fall 2023

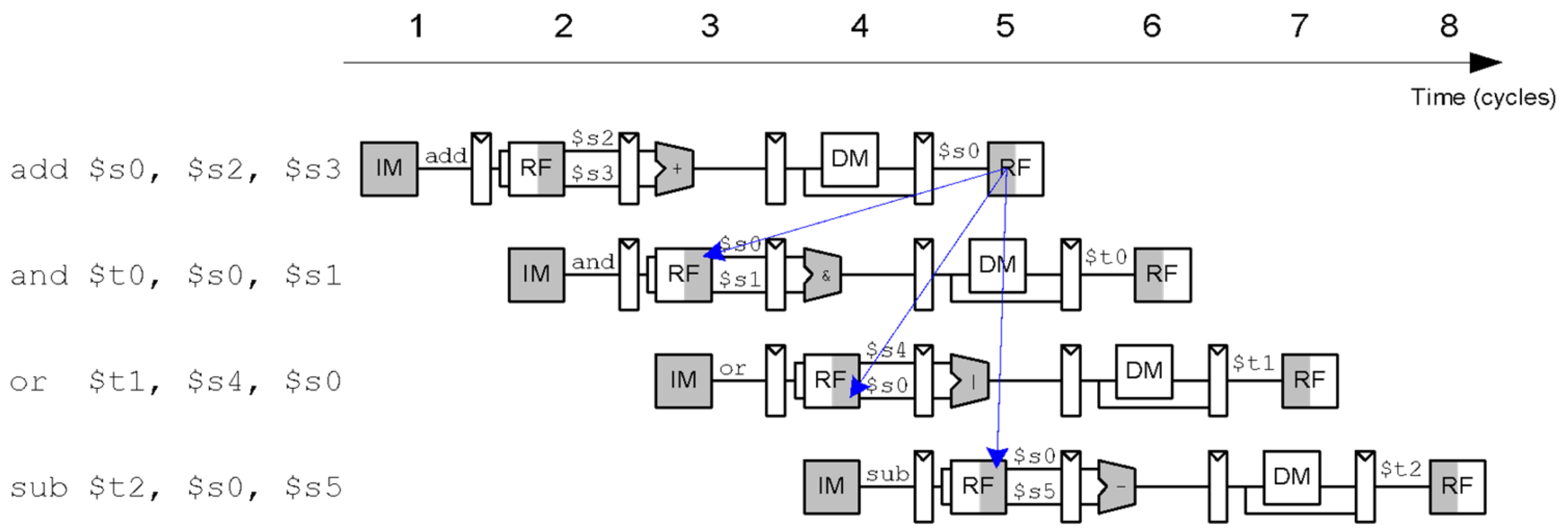
Pipelined MIPS processor: Data Hazards

These Slides are mainly based on slides of the text book (downloadable from the book’s website).

Pipeline Hazards

- When an instruction depends on the result from another instruction that hasn't completed yet, hazards (that can cause wrong results) occur.
- Hazard Types:
  - **Data hazard example:** a register value needed has not yet been written back to register file by the previous instruction
  - **Control hazard:** next instruction to be executed has not been decided yet (caused by branches)

Data Hazard Example



$s0 is destination for add, source for and, or and sub instructions. This is a source of hazard…

Pipeline Hazard Example

- add instruction writes a result into $s0 in the first half of cycle 5. However, the and instruction reads $s0 in cycle 3, obtaining the wrong value.
- Also, or instruction reads $s0 in cycle 4, before add can writes to it in cycle 5, agin obtaining the wrong value.
- The sub instruction reads $s0 in the second half of cycle 5, obtaining the correct value, written in the first half of the cycle 5 by add instruction.
- The subsequent instructions read the correct value of $s0. As they start after stage 5…

Pipeline Hazard Example (cont.)

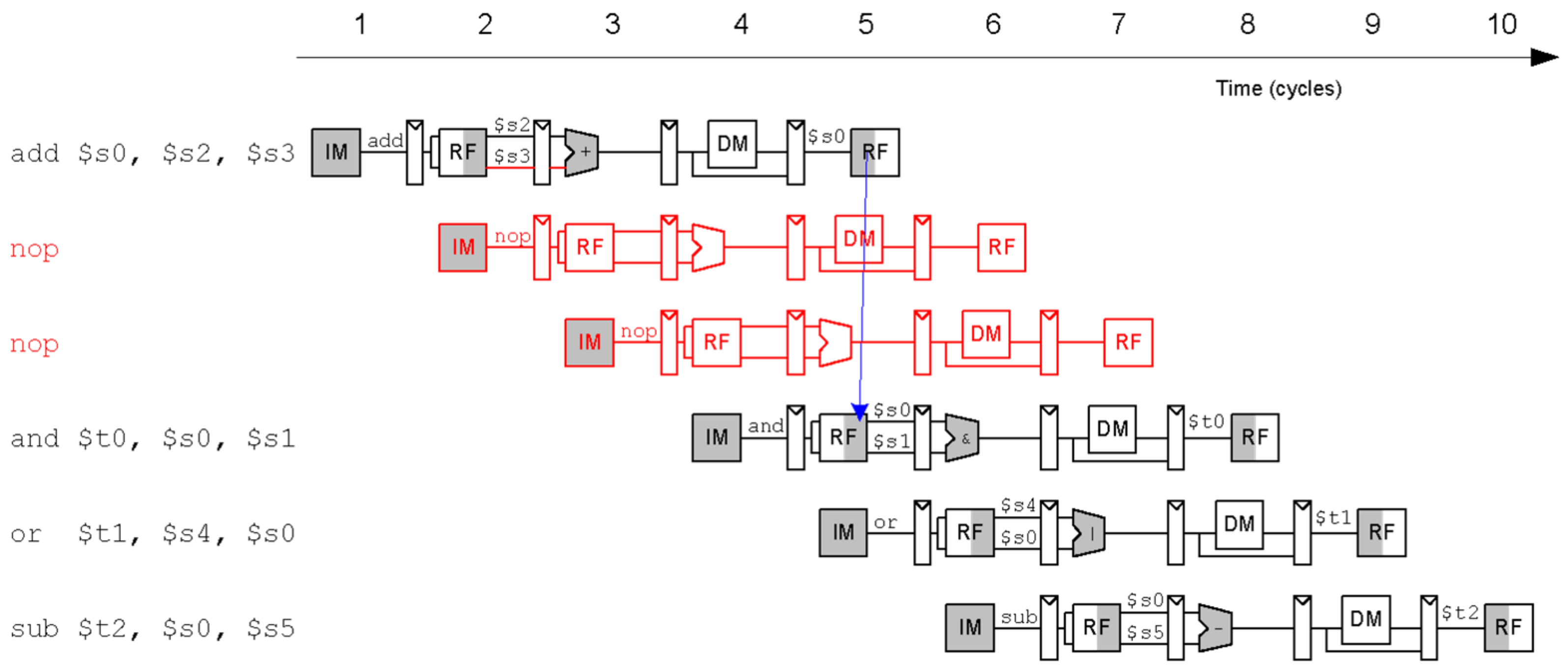
- Notice that the sum from the add instruction is computed by the ALU in cycle 3, also it is not strictly needed by the and instruction until the ALU uses it in cycle 4.
- So, in principle, we should be able to forward the result from cycle 3 of add instruction to the cycle 4 of and instruction, without slowing down the pipeline.

Methads of Handling Data Hazards

- Data hazards can be handled by various approaches, in the design stage. Prepare to:-
  - ✓ Insert `nops` (no operation instruction) instruction code at compile time
  - ✓ Rearrange the program code at compile time
  - ✓ Forward data at run time
  - ✓ Stall the processor at run time

nops for Data Hazard Elimination

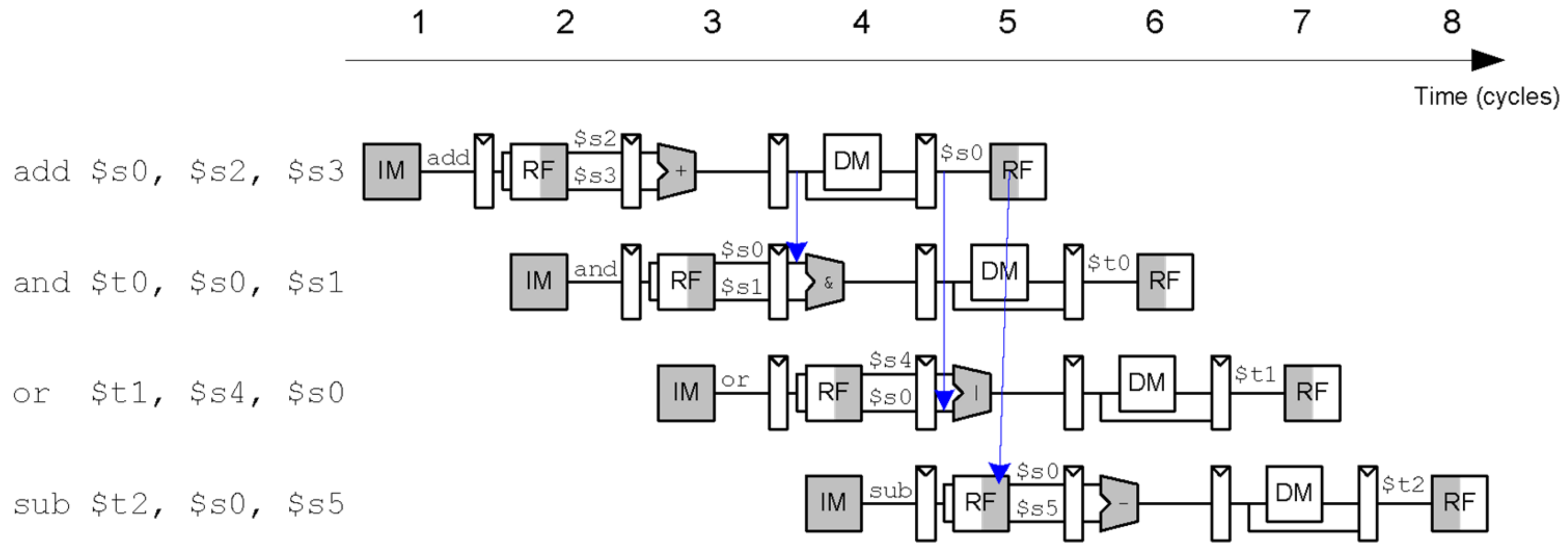
- Insert enough `nop` so that next instruction is delayed till the result is ready. Thus, and instr is delayed until stage 4, moving independent useful instructions forward.



Forwarding to Handle Data Hazards

- Use Forwarding if
  - an instruction in the Execute stage has a source register matching the destination register of an earlier instruction in the Memory or Writeback stage.

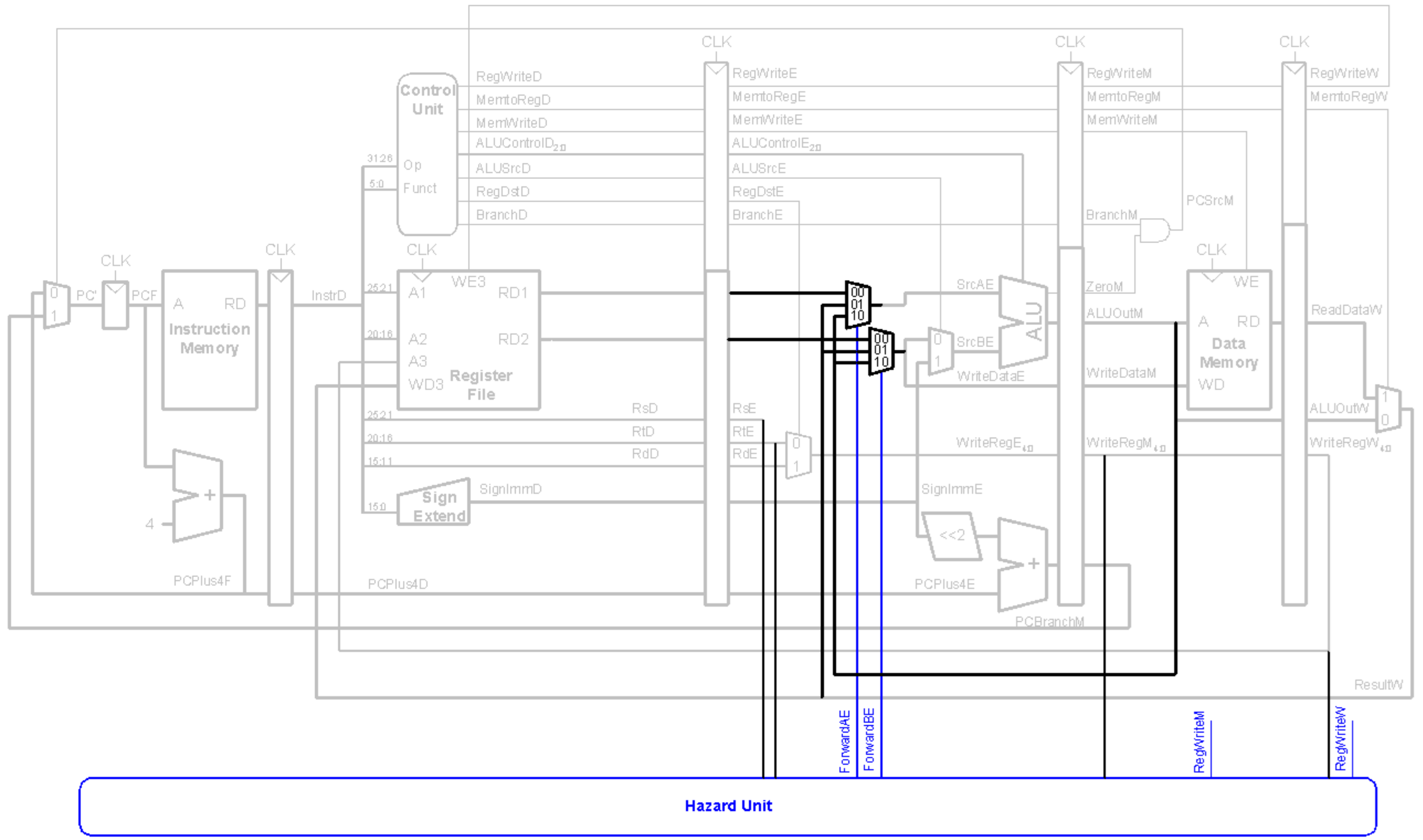
Forwarding to handle data hazards



Enhance the design to manage hazards

- Enhance the design by a hazard control unit to hold hazard signals, delivering them with the related data.
- In this case, 2 multiplexers are added in front of the ALUs to select the operand from one of 3 possible inputs: RF, Memory, or WriteBack stage.
- The hazard detection unit must include proper control signals to control forwarding with the right RegWrite signals.
  - for example, the sw and beq instructions do not write results to the register file and hence do not need to have their results forwarded.

Enhanced Design that includes forwarding



Data Forwarding Logic

- Forward to Execute stage from either Memory stage or Writeback stage using related multiplexers
- Forwarding logic for *ForwardAE signal* to select the correct mux input

```
if    ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
          then ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
          then ForwardAE = 01
else           ForwardAE = 00
```

- **Forwarding logic for *ForwardBE* is the same, only replace *rsE* with *rtE***

Data Forwarding from different stages

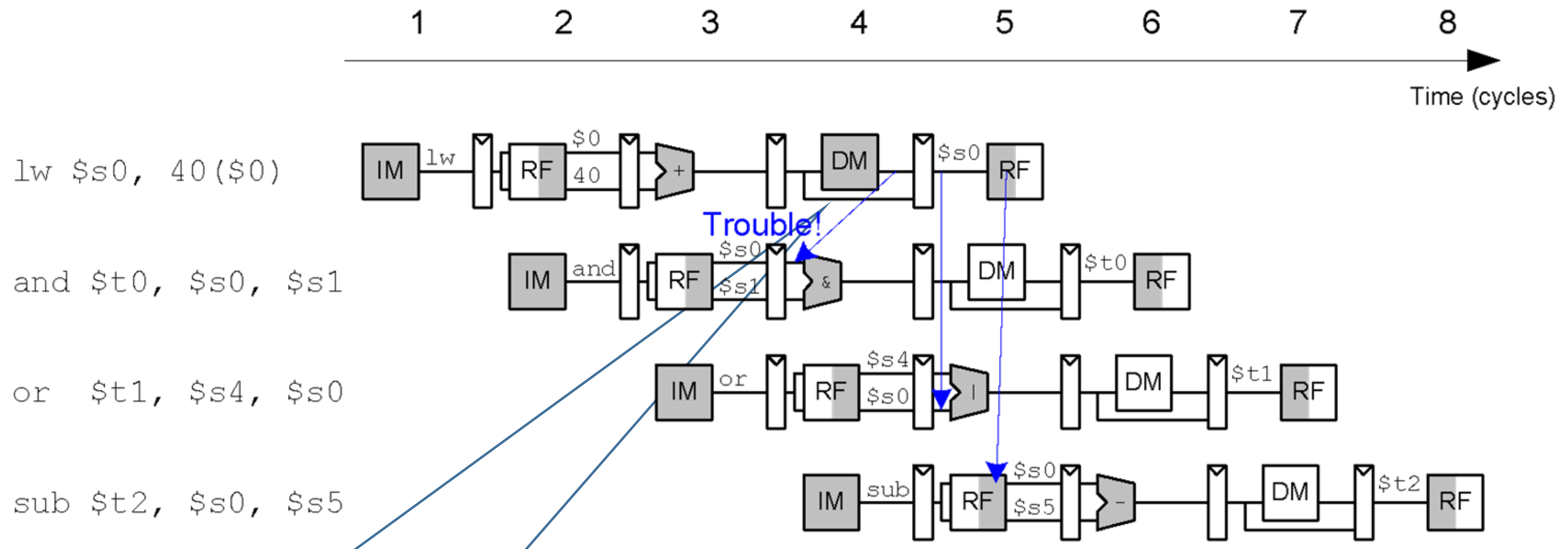
- If both the Memory and Writeback stages contain matching destination registers, the Memory stage has priority:
    - ✓ memory stage belongs to a more recently executed instruction, so it contains more recent data and control.

Forwarding may not solve the problem!

For example,

- the lw 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.
  - This means lw instruction has two-cycle latency,
    - so, a dependent next instruction cannot use result of lw until two cycles later.

Is forwarding always a solution?



- Why is it a trouble to forward data to inputs of ALU (in Execute stage) from the output of DM (in memory stage)?
- Why is it not a problem to forward data to input of ALU from the input of DM?

- ALU is a combinational circuit. If we forward the data from output of DM to input of ALU, an additional clock period is required. Sso forarding cannot by synchronized with execute stae of the next instruction.
- Forarding cannot solve this problem

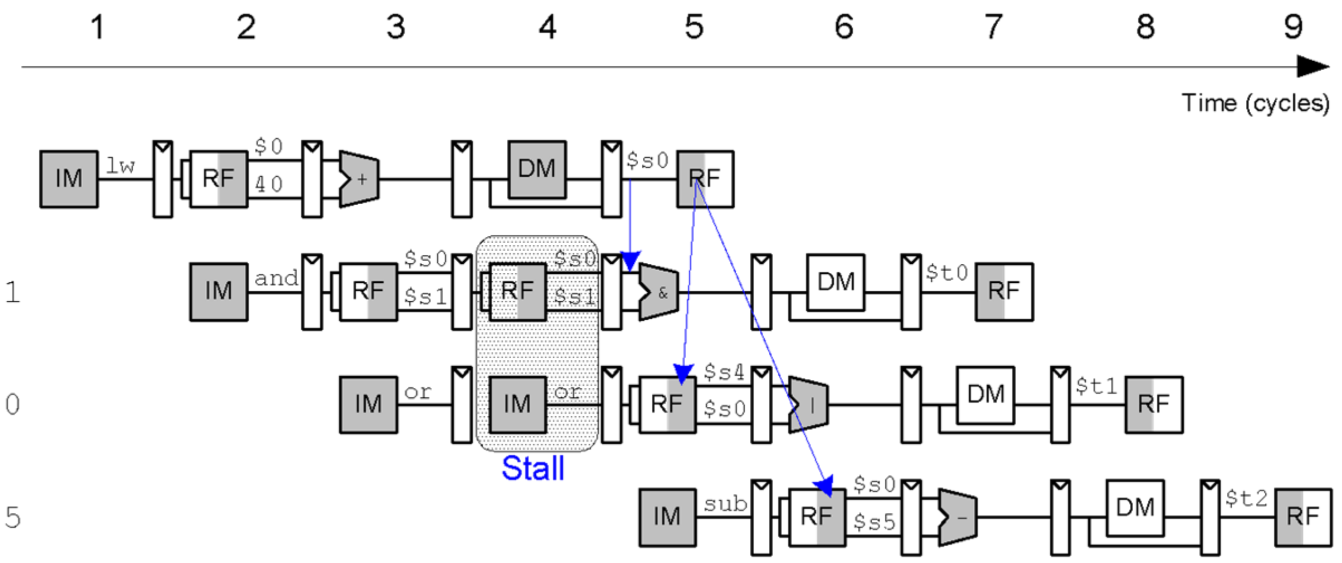
Stall the pipeline for some hazards

Stall: hold up an operation until the data is available:

- In this case, and instruction enters decode at stage 3 and stays(stalls) there through stage 4.
- or instruction also stalls in fetch stage as decode stage is not freed from the previous instruction, and

Stalling

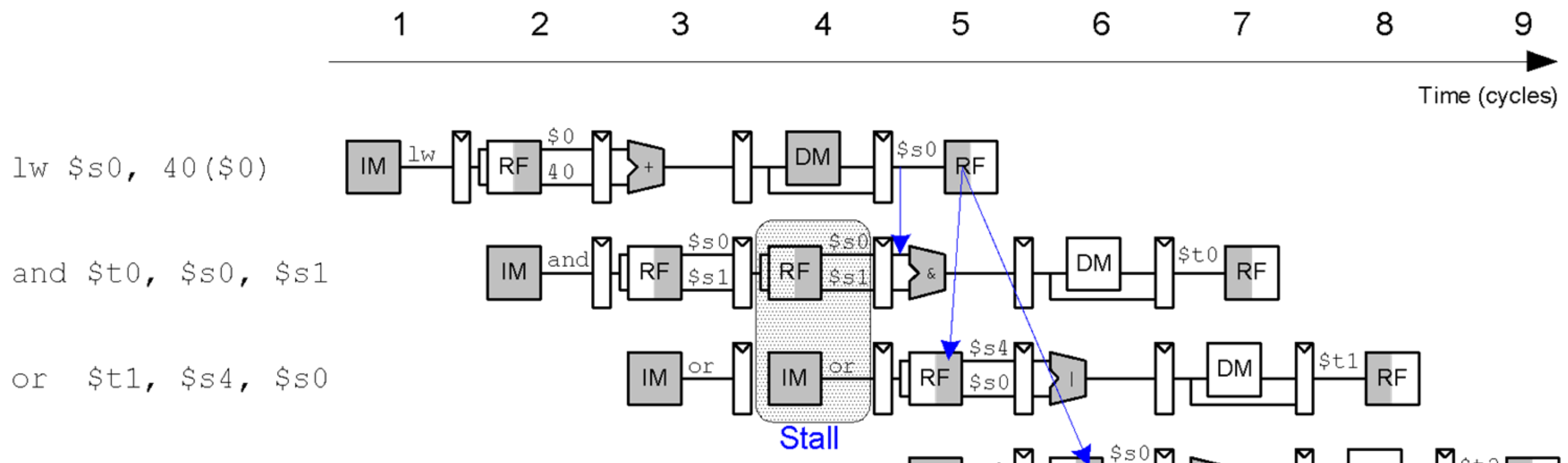
```
lw  $s0, 40($0)
and $t0, $s0, $s1
or  $t1, $s4, $s0
sub $t2, $s0, $s5
```



Stall solution to hazards: Example

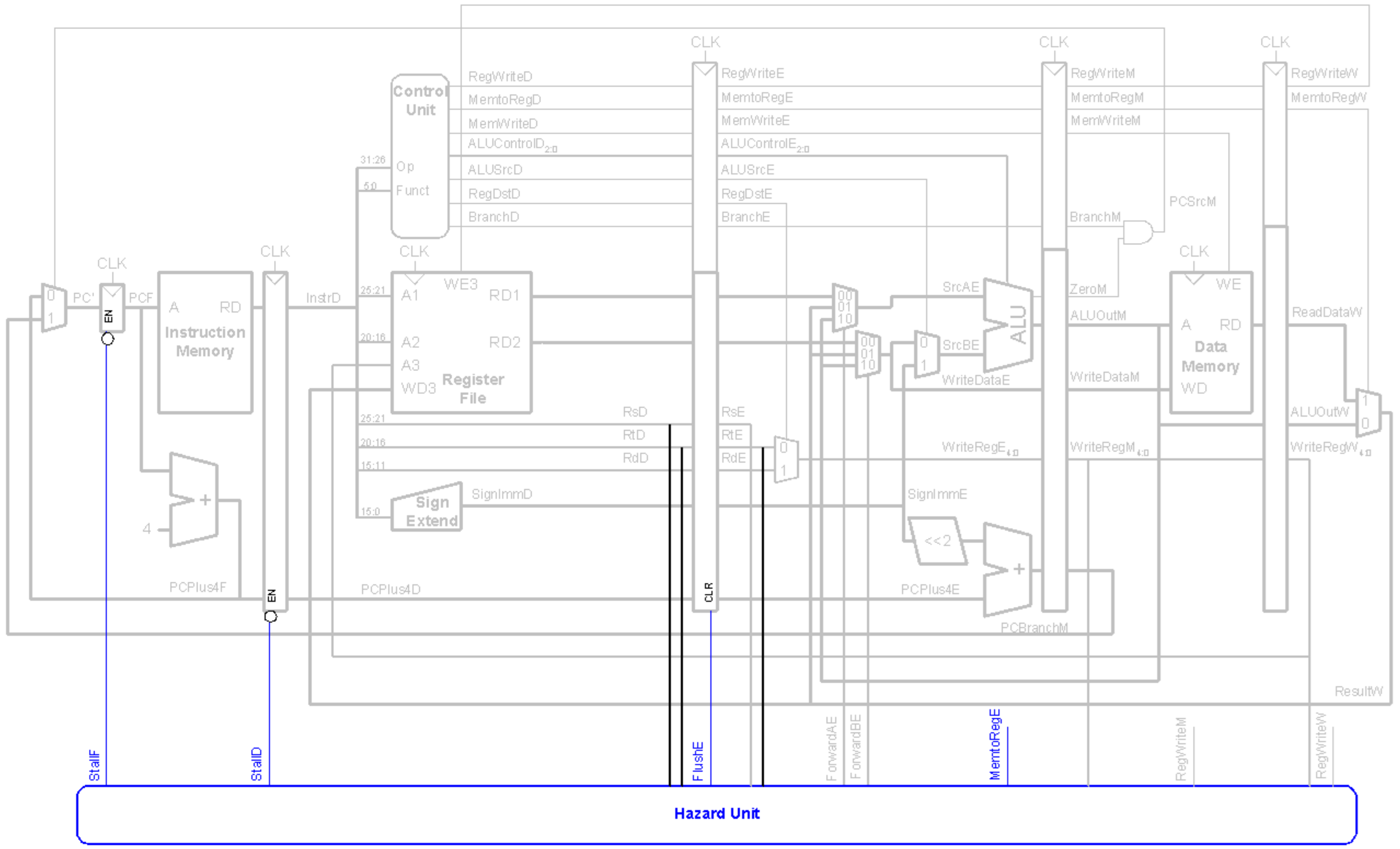
- In cycle 5, the result can be forwarded from the Writeback stage of lw to the Execute stage of and.
- In cycle 5, source $s0 of the or instruction is read directly from the register file, with no need for forwarding.

Implementation of Stalling



1. Stalling requires by disabling the pipeline register, so that the contents do not change.
2. When a stage is stalled, all following stages must also be stalled, so that no subsequent instructions are lost.
3. The pipeline register directly after the stalled stage must be cleared to prevent bogus information from propagating forward.
4. Stalls degrade performance, so they should only be used when necessary.

Stalling Hardware



Stalling example: lw

- When a lw stall occurs, StallD and StallF are asserted to force the Decode and Fetch stage pipeline registers to hold their old values.
- FlushE is also asserted to clear the contents of the Execute stage pipeline register, to avoid any adverse effect.

Stall Logic

- The MemtoReg signal is asserted for the lw instruction.

- The logic to compute the stalls and flushes:

```
lwstall =
  ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE


StallF = StallD = FlushE = lwstall
```

- All signals including lwstall are maintained my the hazard unit