Unit 13: Pipelined Processors

CSE 220: System Fundamentals I

Stony Brook University

Joydeep Mitra

Parallelism

- Parallelism is a method to increase *throughput* and decrease *latency*
  - If we conceptualize a system as processing tokens (a group of inputs) to produce a group of outputs, then **latency is the time required to process one token**, and
  - ***throughput is no. of such tokens that can be processed per unit time***.
- Processing several tokens at the same time will increase throughput. In fact, this is parallelism! and there are two kinds:
  - **Spatial**: Multiple copies of the hardware are available to process multiple tokens at the same time
  - **Temporal**: A task is broken into stages so while each task must pass through all stages, a different task will be in each stage at any given time.

Spatial vs Temporal

- Consider an example:
  - We need to bake hundreds of cookies for a party!
  - We have two options:
    - invite a friend to help who gets their own oven and tray (spatial), or
    - use two cookie trays. If one is baking, start rolling the other one (temporal).
  - Assuming a latency of 20 mins,
    - without parallelism, we will bake 3 trays/hr.
    - with spatial parallelism, we will bake 6 trays/hr.
    - with temporal parallelism, we will put a tray every 15 mins. to get a throughput of 4 trays/hr.
- Key insight: *even with minimal extra hardware, we can increase throughput*.

Pipelined Processor

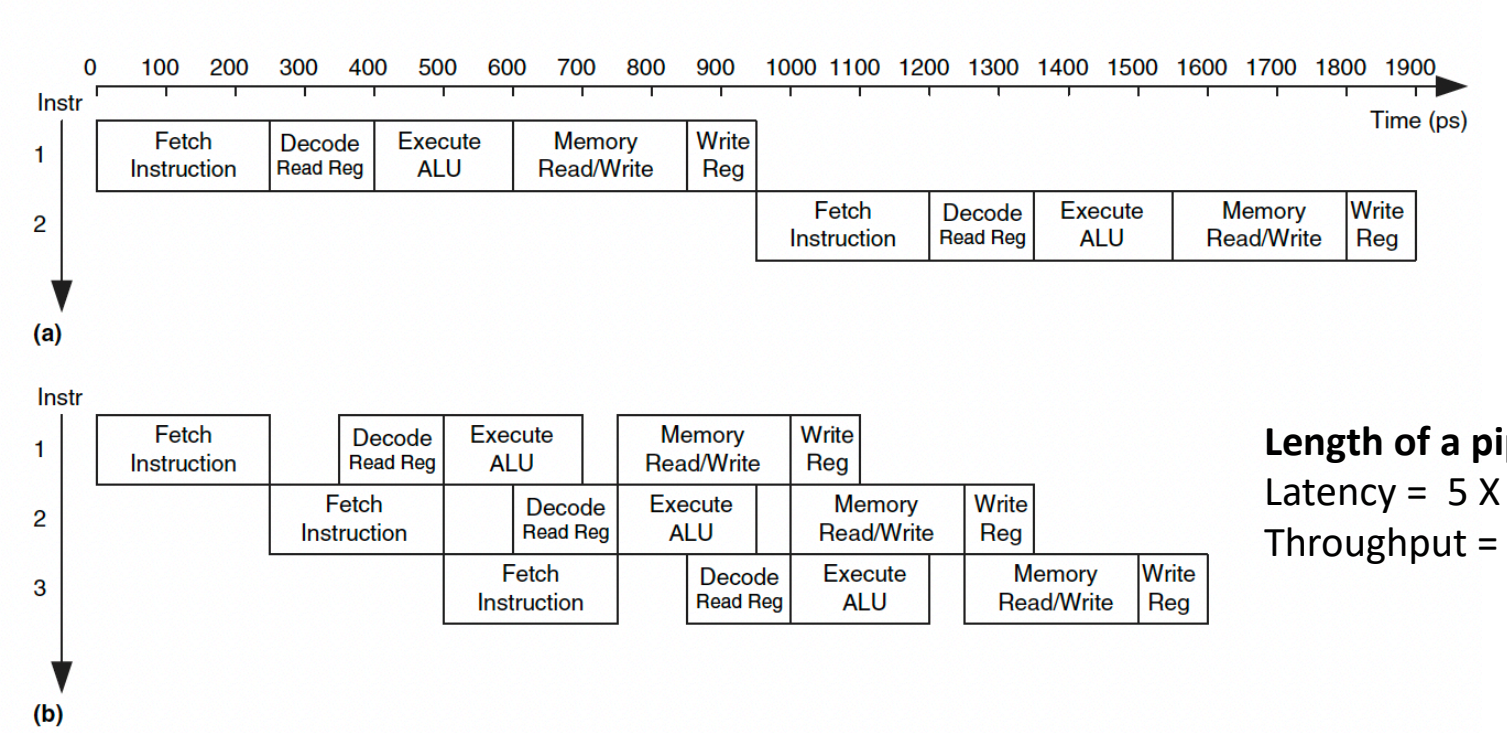
- We design a pipelined processor by sub-dividing the single cycle processor into 5 pipelined stages
- Thus, 5 instructions will execute simultaneously
- Consequently, the clock frequency will be 5 times faster (ideally)
- Latency is almost the same for each instruction
- Pipelining introduces some overhead, but a microprocessor has billions of instructions. So, overall, throughput we will increase

Pipelined Processor

- The biggest delays are incurred by reading/writing memory, register file, and the ALU.
- Hence, we choose five pipeline stages so that each stage involves a slow step.
  - **Fetch**: Processor reads instruction from instruction memory.
  - **Decode**: Processor reads the source operands from the register file and decodes the instruction to produce control signals.
  - **Execute**: Processor performs ALU operations.
  - **Memory**: Processor reads/writes data memory.
  - **Writeback**: Processor writes result back to register file, when applicable.

Timing Diagrams

- Single-Cycle (a) vs Pipelining (b)



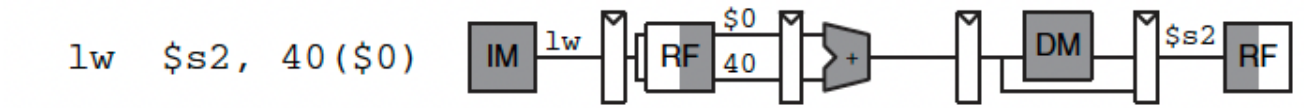
Latency = 950 ps
Throughput = 1 instruction/950 ps

Length of a pipeline stage set as 250 ps by the slowest stage
Latency = 5 X 250 = 1250 ps
Throughput = 1 instruction/250 ps

Pipelined processor has slightly higher latency but significantly higher throughput than single-cycle

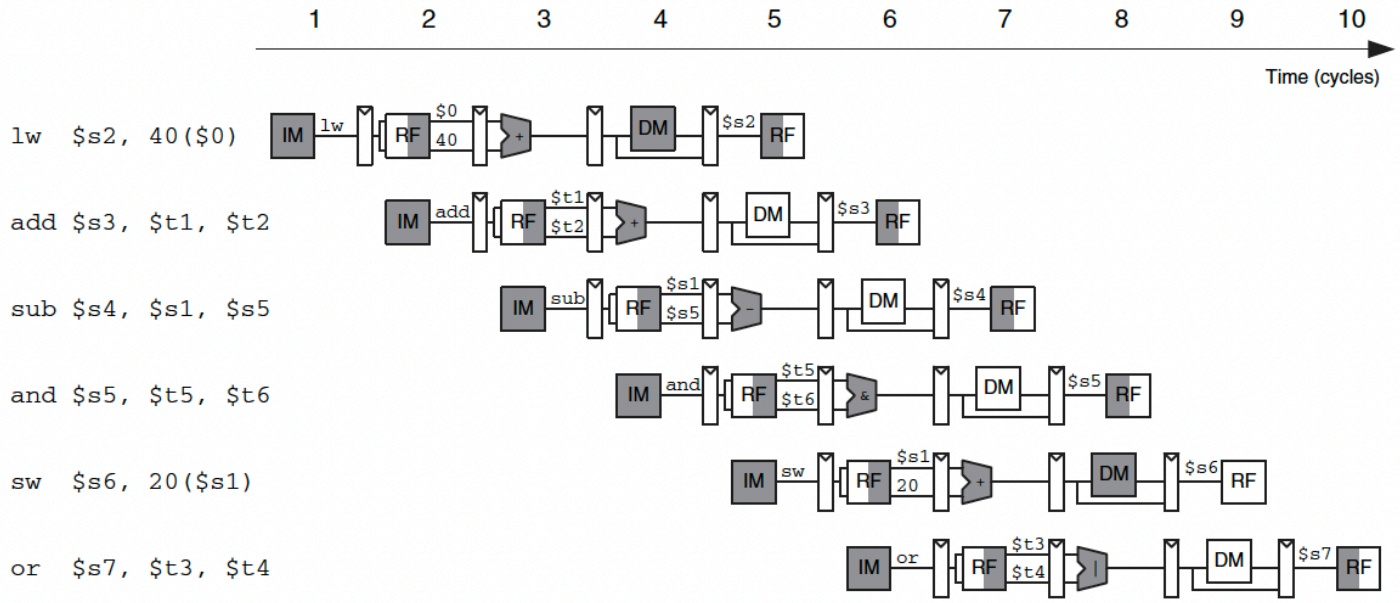
Abstract View of The Pipeline

- It is easier to reason if we have an abstraction of the pipelined processor
- So, we represent each pipeline stage with its major component
  - IM (instruction memory)
  - RF (register file read)
  - ALU execution
  - DM (data memory)
  - RF (write back to register file)
- We will shade a stage when it is actively being used in an instruction
- Shading for RF is different:
  - Data to RF is written in the 1st half of a clock cycle and read from RF in the 2nd half of the clock cycle.
  - This is shown by shading the 1st half of RF for writeback and 2nd half of RF for reads.



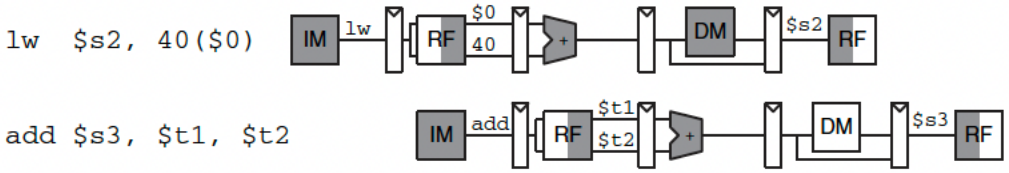
Abstract View of The Pipeline

- Reading across a row shows the clock cycles in which an instruction is in each stage. E.g.,
  - sub instruction is fetched in cycle 3 and executed in cycle 5
- Reading down a column shows what the pipeline stages are doing. E.g., in cycle 6
  - the or instruction is being fetched from IM,
  - $s1 is being read from RF
  - ALU is computing $t5 AND $t6
  - Data memory is idle
  - RF is writing to $s3



Hazard In Pipelined Processor

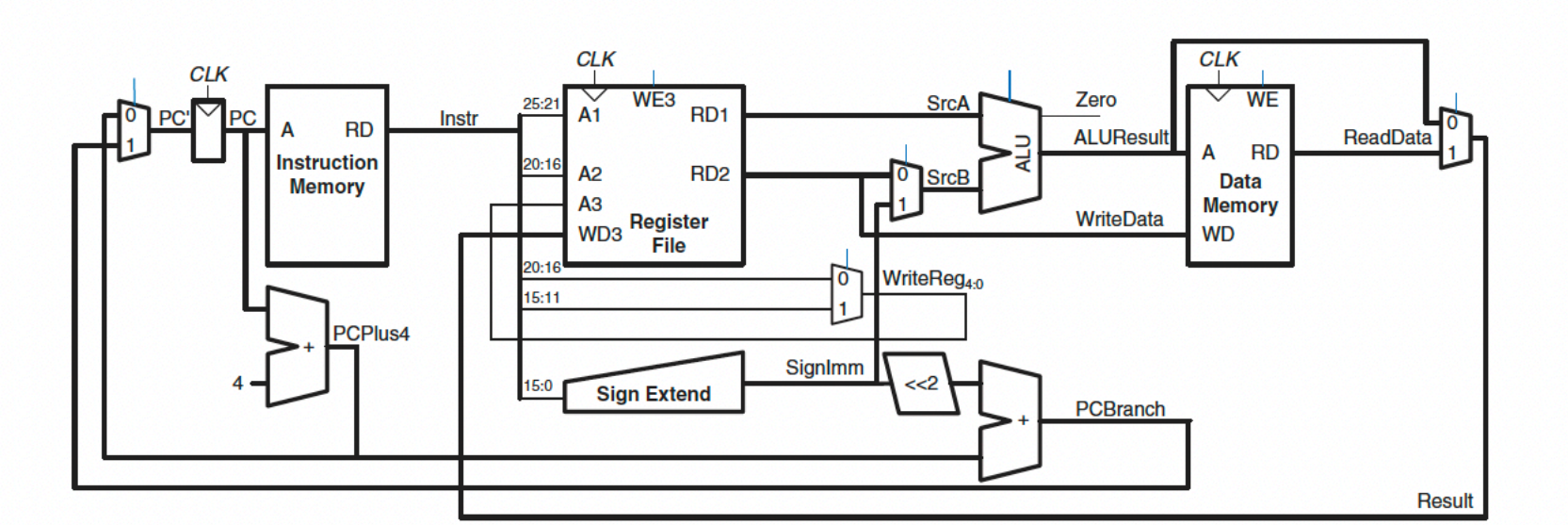
- A central challenge of pipelined processors is deal with **hazards**
- Hazards occur when the result of an instruction I_1 is required by the instruction of I_2 but I_1 hasn't completed
- What will happen if the add instruction used **$s2** instead of **$t2**?



- A hazard would occur! Why?
  - Because $s2 is not written by `lw` by the time it is read by `add`
- Hazards need to be resolved. More on this later.

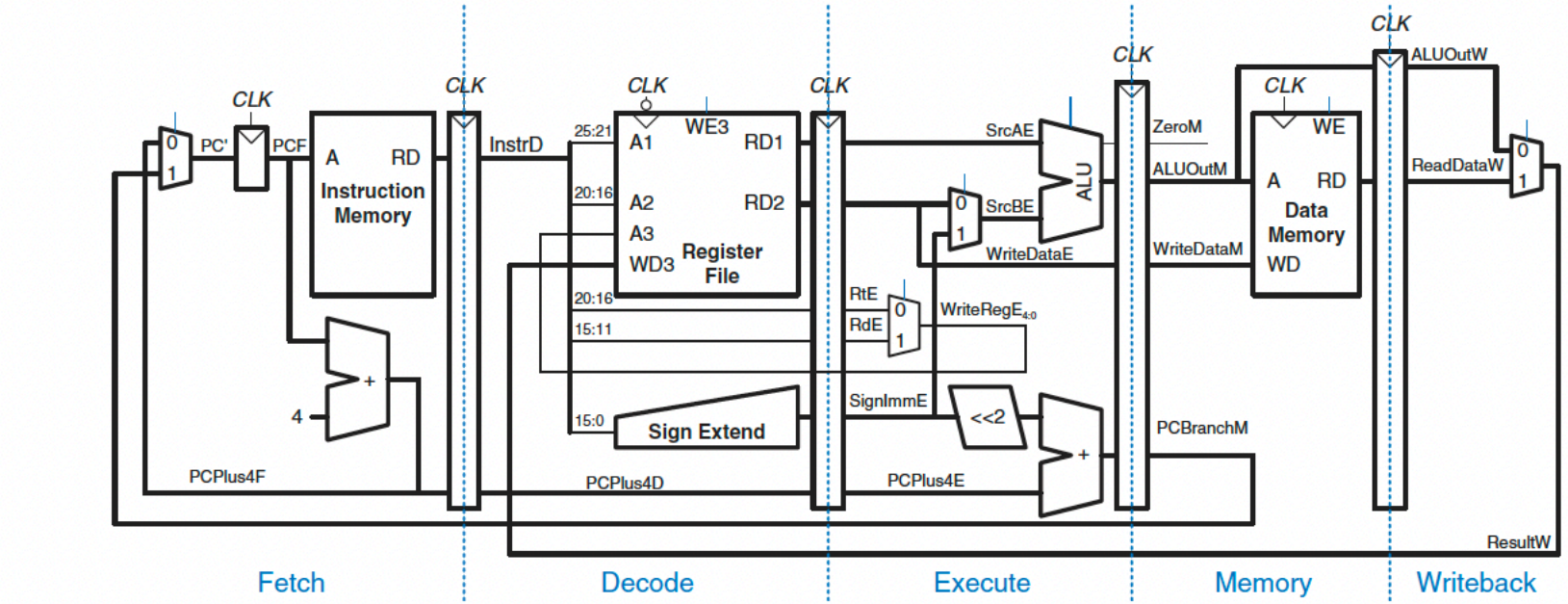
The Pipelined Datapath

- Recall the single-cycle processor's datapath



The Pipelined Datapath

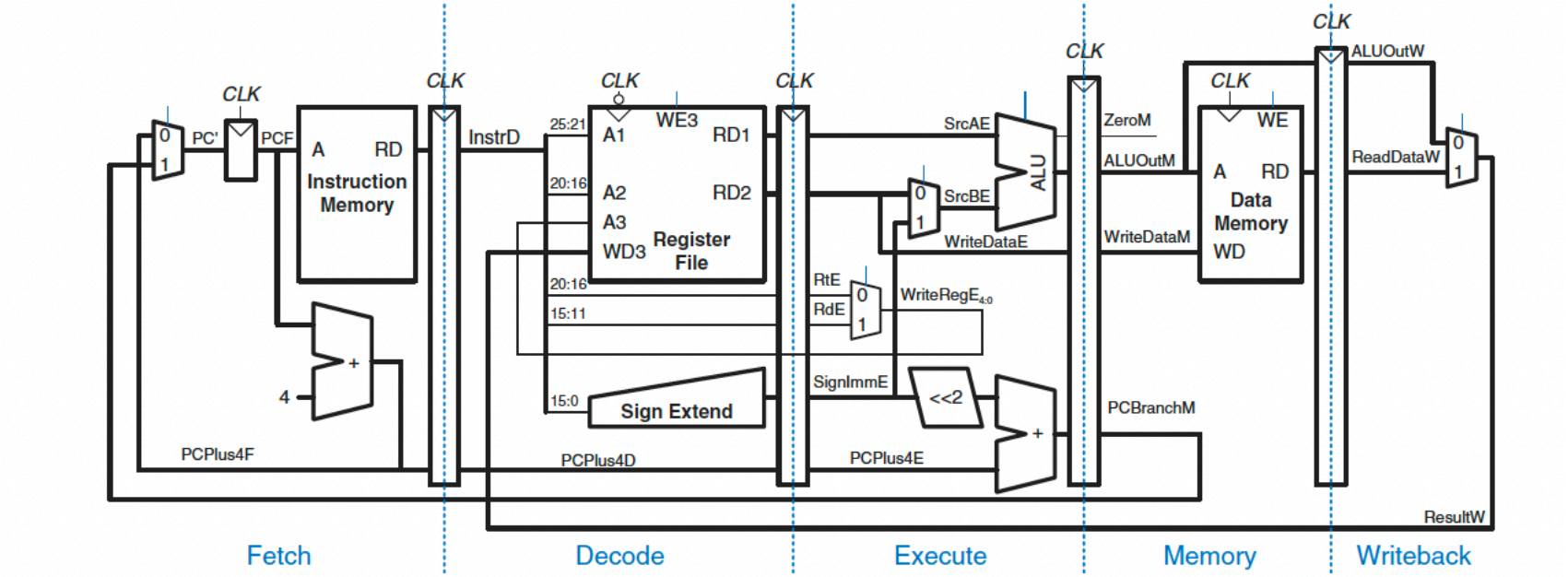
- A pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by **pipeline registers**



- Signals are suffix-ed with F, D, E, M, or W to indicate the stage in which they reside

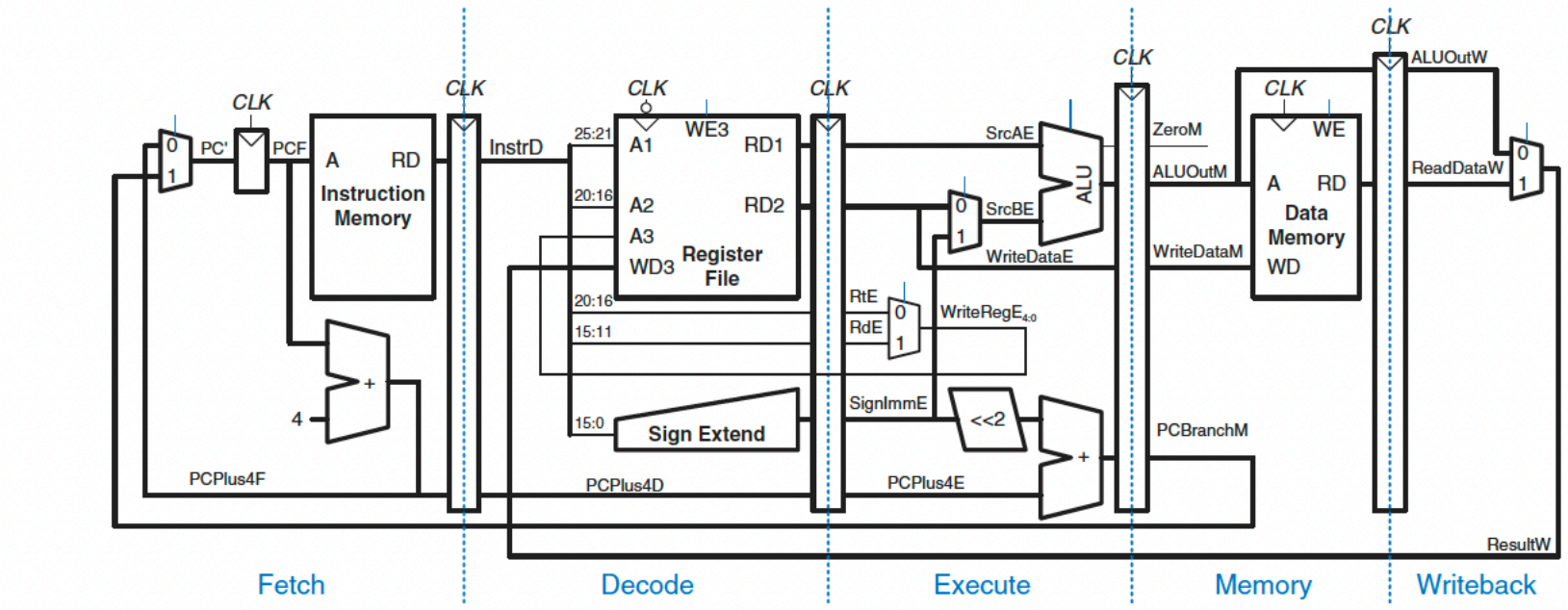
The Pipelined Datapath

- A critical aspect of pipelining is that **all signals associated with an instruction must advance through the pipeline in unision**
- Failure to do so leads to hazards. There is one in the figure. Can you identify it?



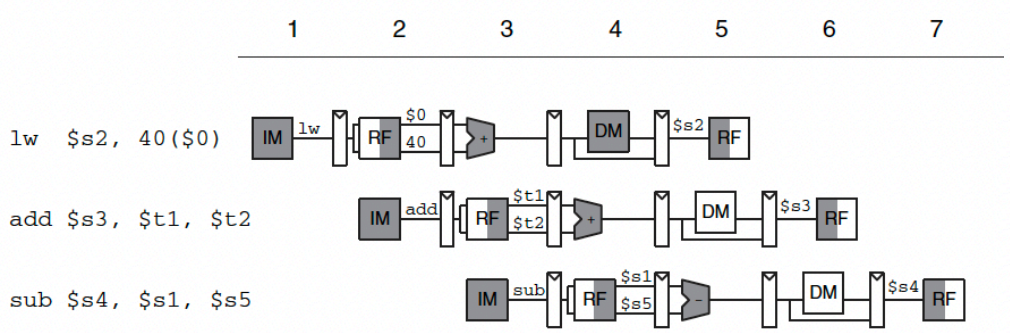
The Pipelined Datapath

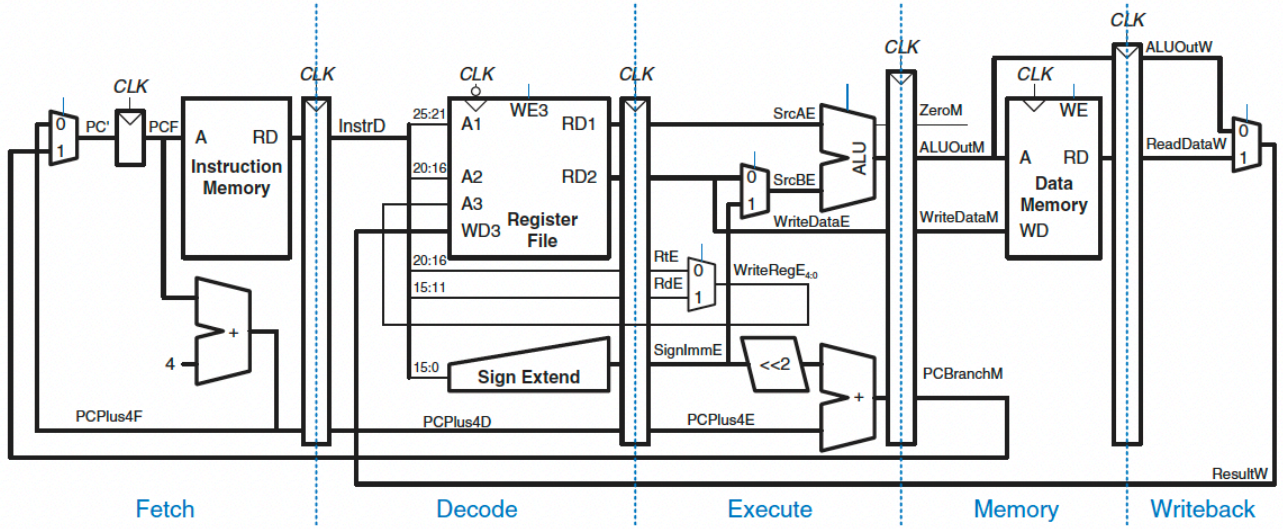
- The address A3 in the register file comes from the Execute stage (*WriteRegE*)
- The data WD3 in the register file comes from the Writeback stage (*ResultW*)



The Pipelined Datapath

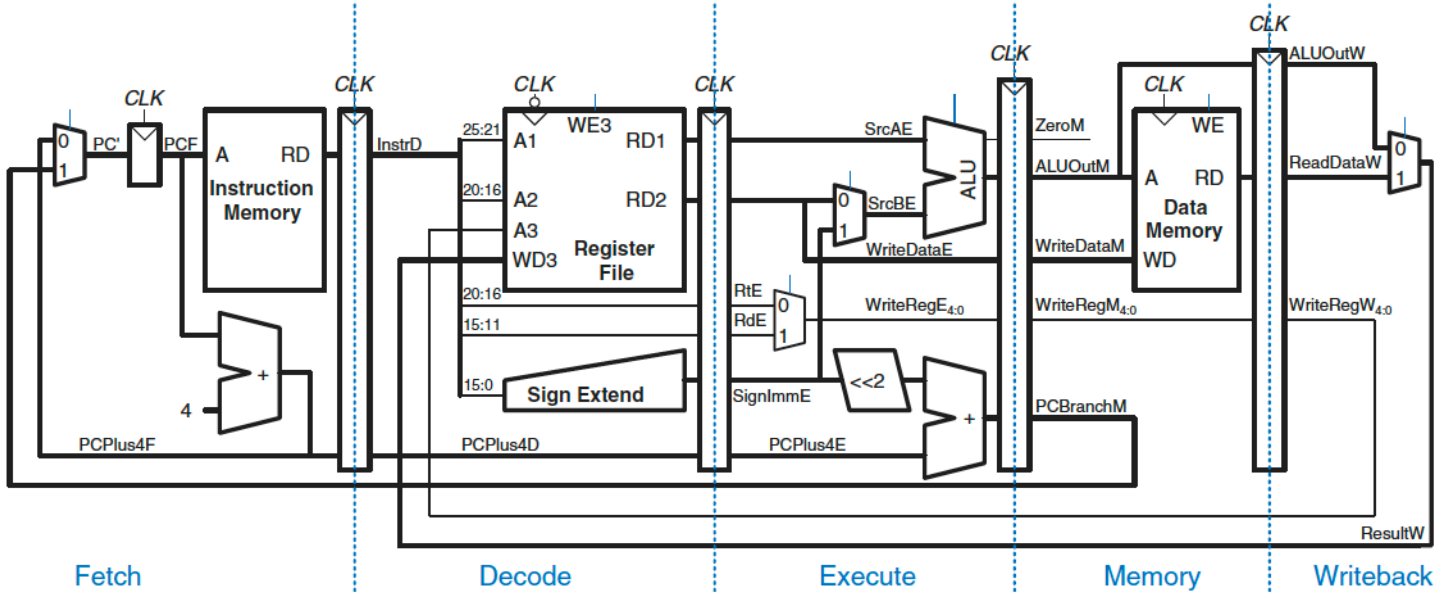
- Consequently, the `lw` instruction may incorrectly write to **$s4** because in cycle 5 the address in A3 of register file will come from the `rd` field of the `sub` instruction.





The Pipelined Datapath

- To fix this, the *WriteReg* signal should be pipelined along through Memory and Writeback stages to sync it with the rest of the instruction.
- *WriteRegW* and *ResultW* should be fed back together to the register file in the Writeback stage.

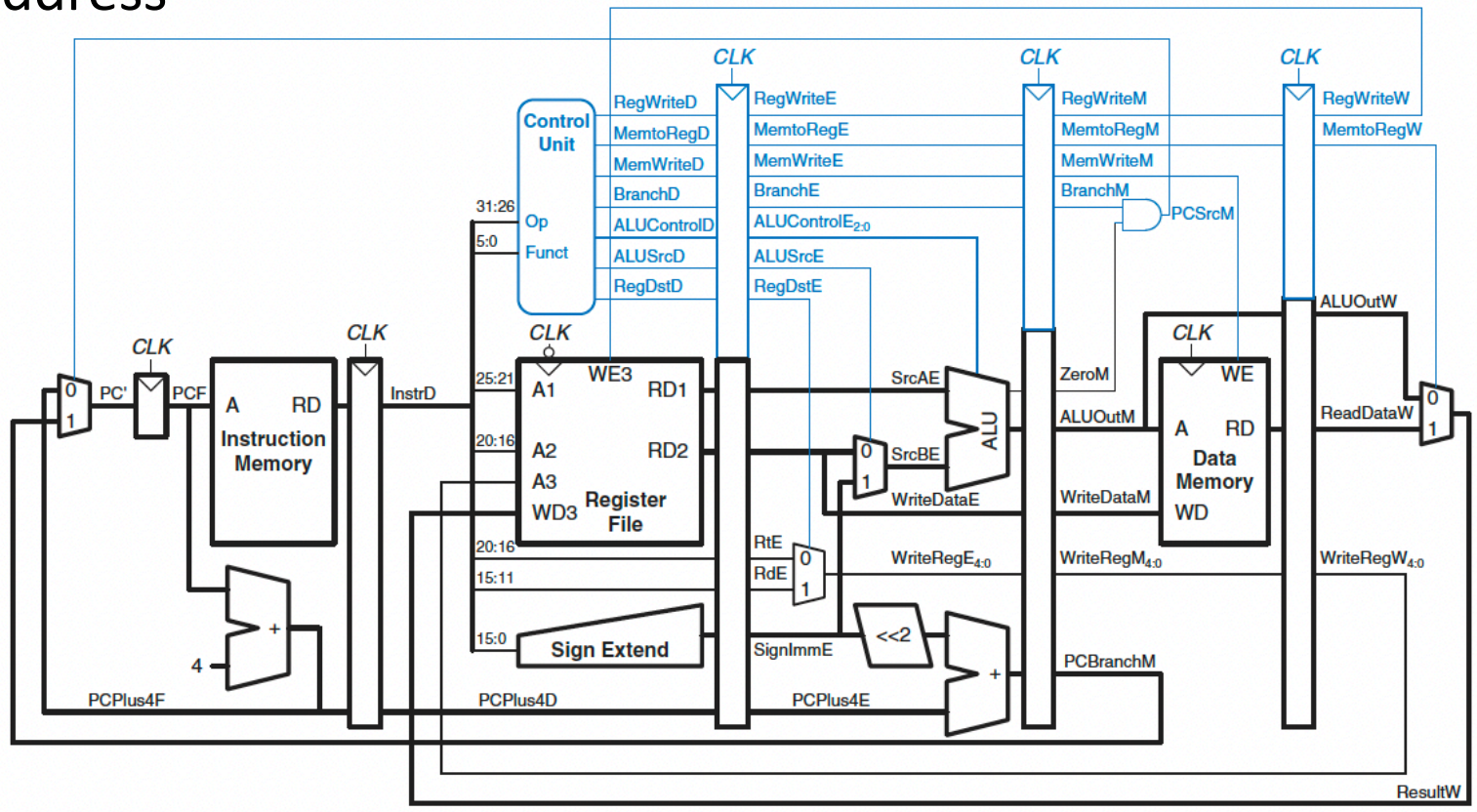


The Pipelined Control

- We can use the same control unit from single-cycle
- After all the pipelined processor uses the same control signals
- But the control signals must be be pipelined through the pipeline registers along with the data so they remain synchronized with the instruction. E.g.,
  - The *RegWrite* signal must be pipelined into the Writeback stage before it feeds back to the register file (same as *WriteReg*)
  - Recall *RegWrite* signal controls if write to register should happen; *WriteReg* contains the address where the write will happen.

The Pipelined Control

- Notice how the *RegWrite* is forwarded and then fed back
- Notice how the PCPlus4 is forwarded because it may be required to calculate the branch address

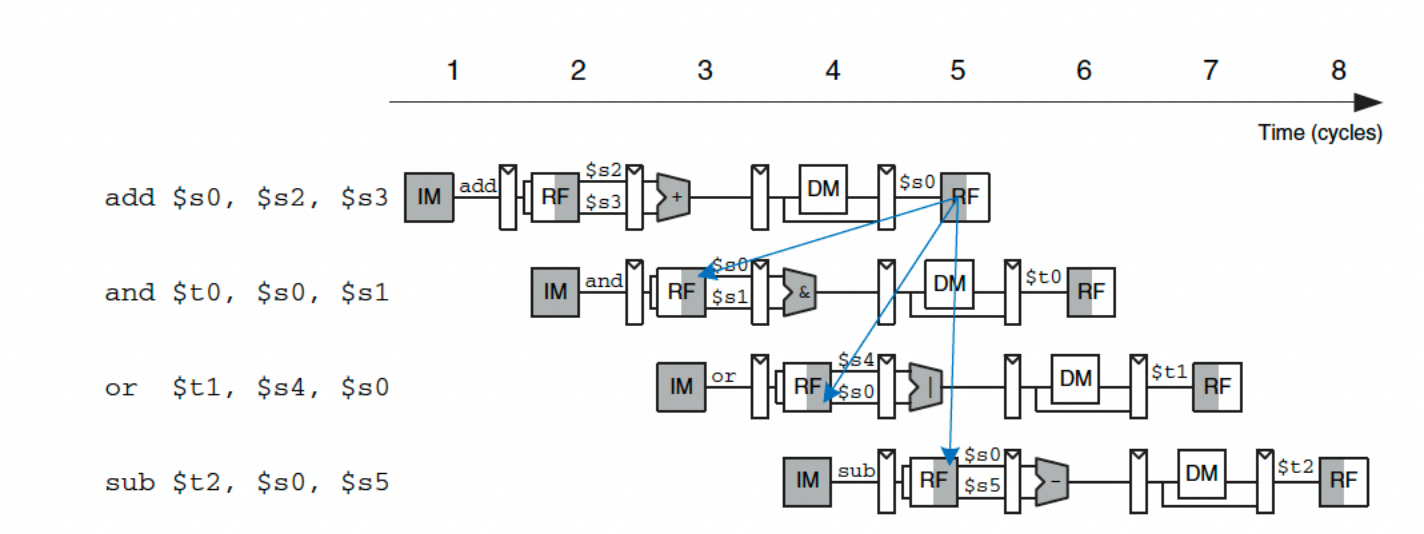


Hazards

- Recall a hazard occurs when one instruction *depends* on the results of another that has not yet completed.
- We know that the register file is read and written in the same cycle.
- So, does that cause a hazard?
  - No, because write takes place in 1st half of a cycle and read in the 2nd half.
- So, what constitutes a hazard?

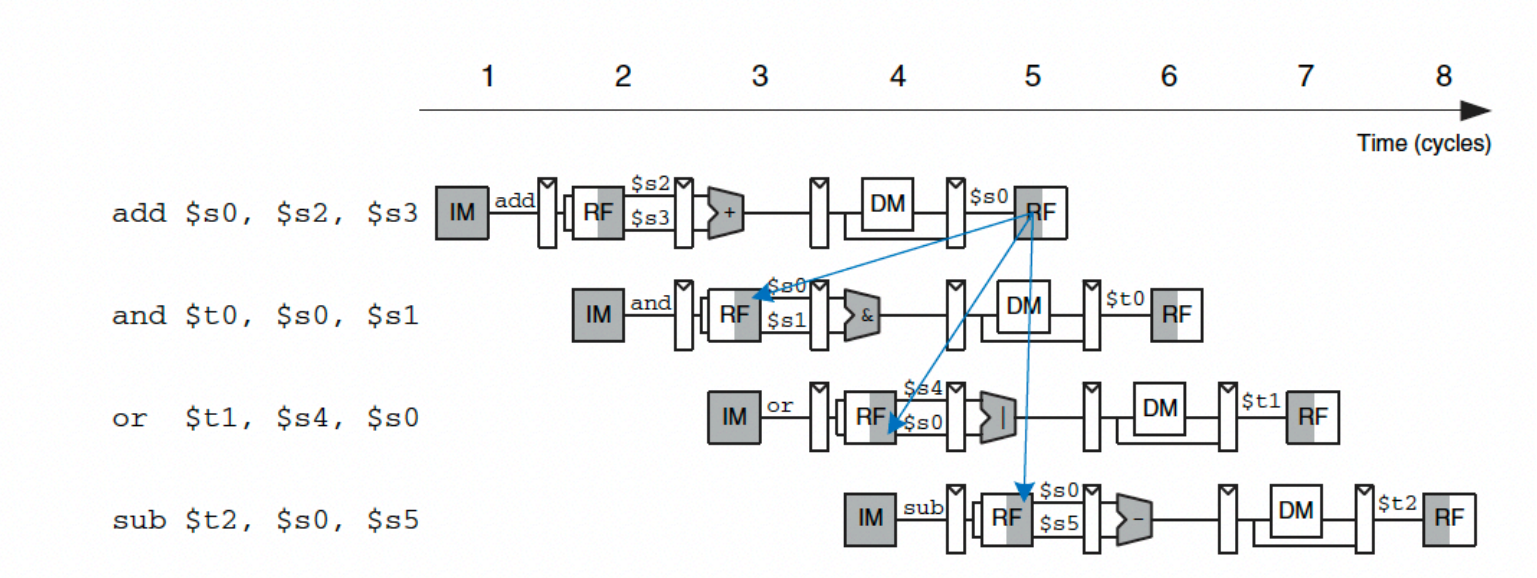
Hazards

- Consider an example where one instruction writes a register ($s0) and subsequent instruction read this register.



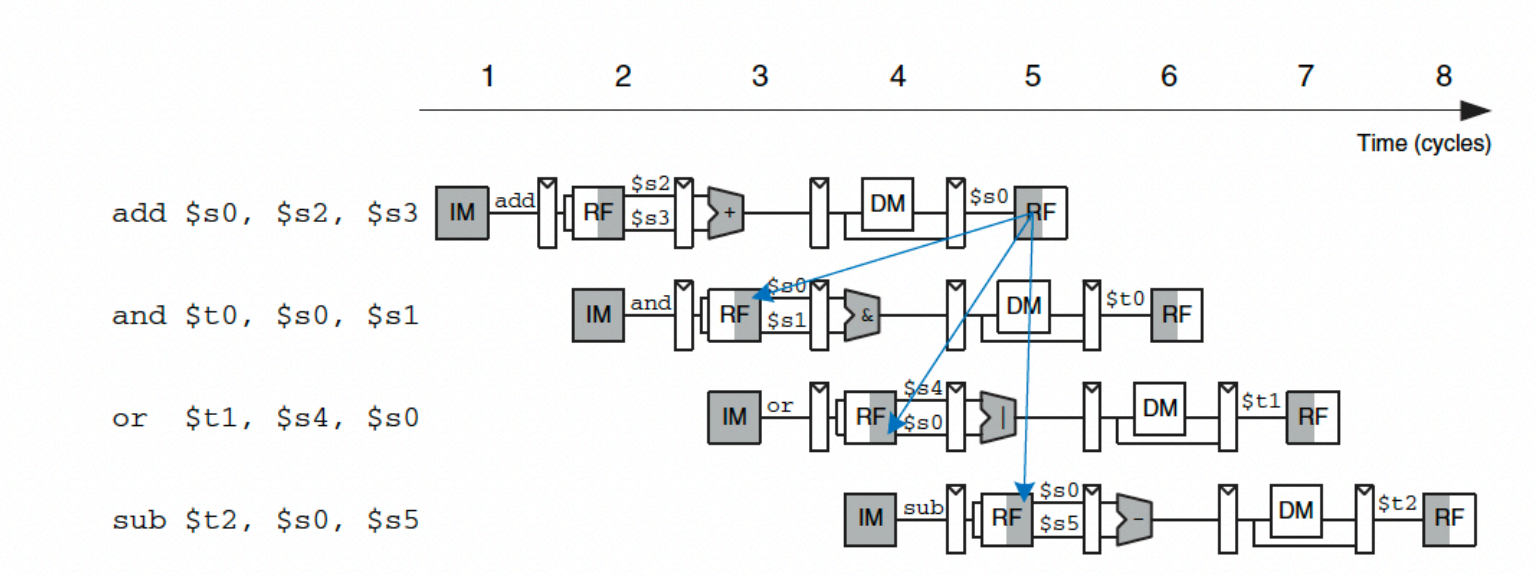
- A dependency exists between (`add`, `and`) , (`add`, `or`) , and (`add`, `sub`).

Hazards



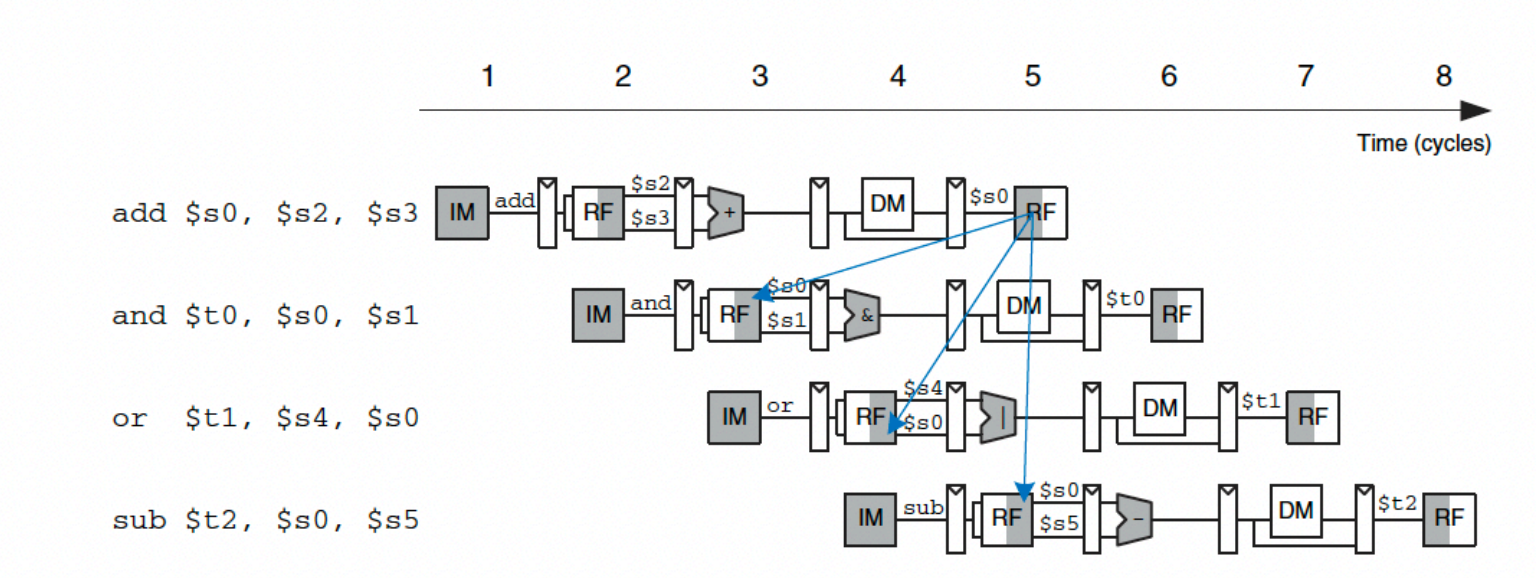
- `add` writes `$s0` in the 1st half of cycle 5.
- `and` reads `$s0` in cycle 3. Hazard!
- `and` reads wrong value from `$s0`

Hazards



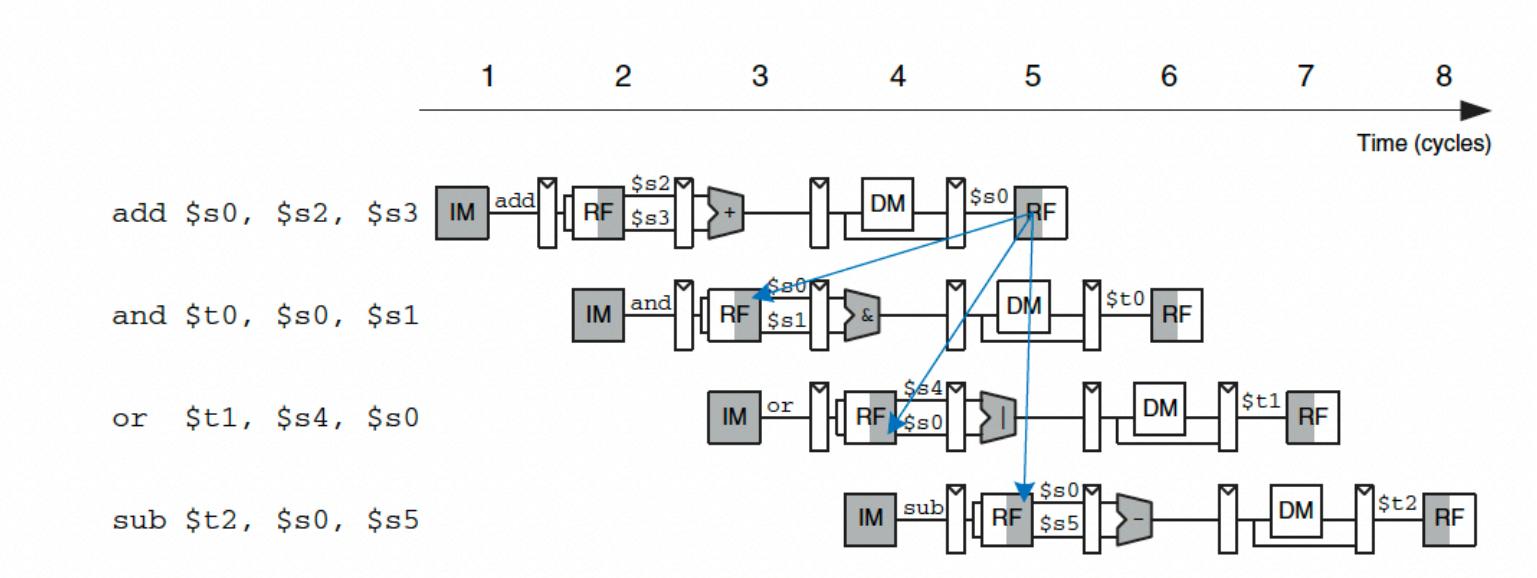
- `add` writes `$s0` in the 1st half of cycle 5.
- `or` reads `$s0` in cycle 4. Hazard!
- `or` reads wrong value from `$s0`

Hazards



- `add` writes `$s0` in the 1st half of cycle 5.
- `sub` reads `$s0` in 2nd half of cycle 5. No Hazard!
- `sub` reads correct value from `$s0`
- What about subsequent instructions if any?
  - They will read the correct value from `$s0`

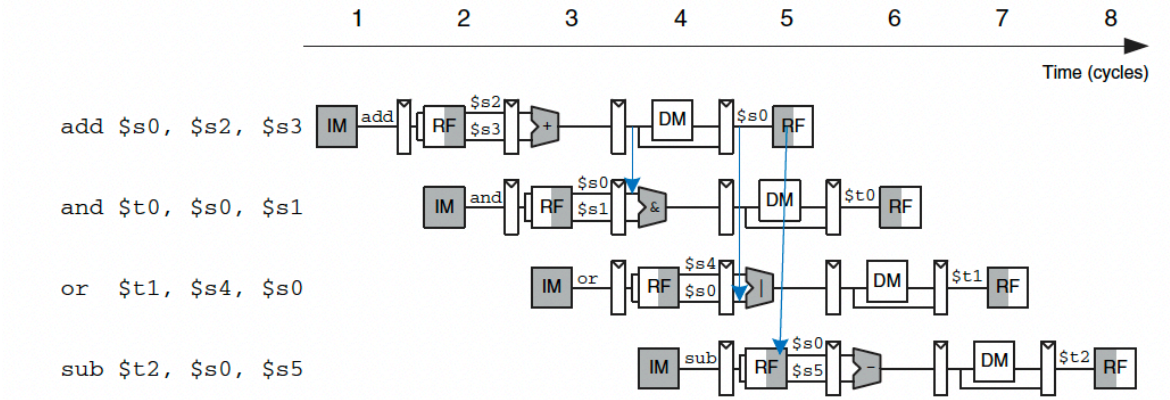
Hazards



- The illustration shows that hazards may occur in this pipeline when an instruction writes a register and either of the two subsequent instructions read that register.
- This is called **read after write (RAW)** hazard.
- Resolving hazards needs special treatment.

Solving Data Hazards With Forwarding

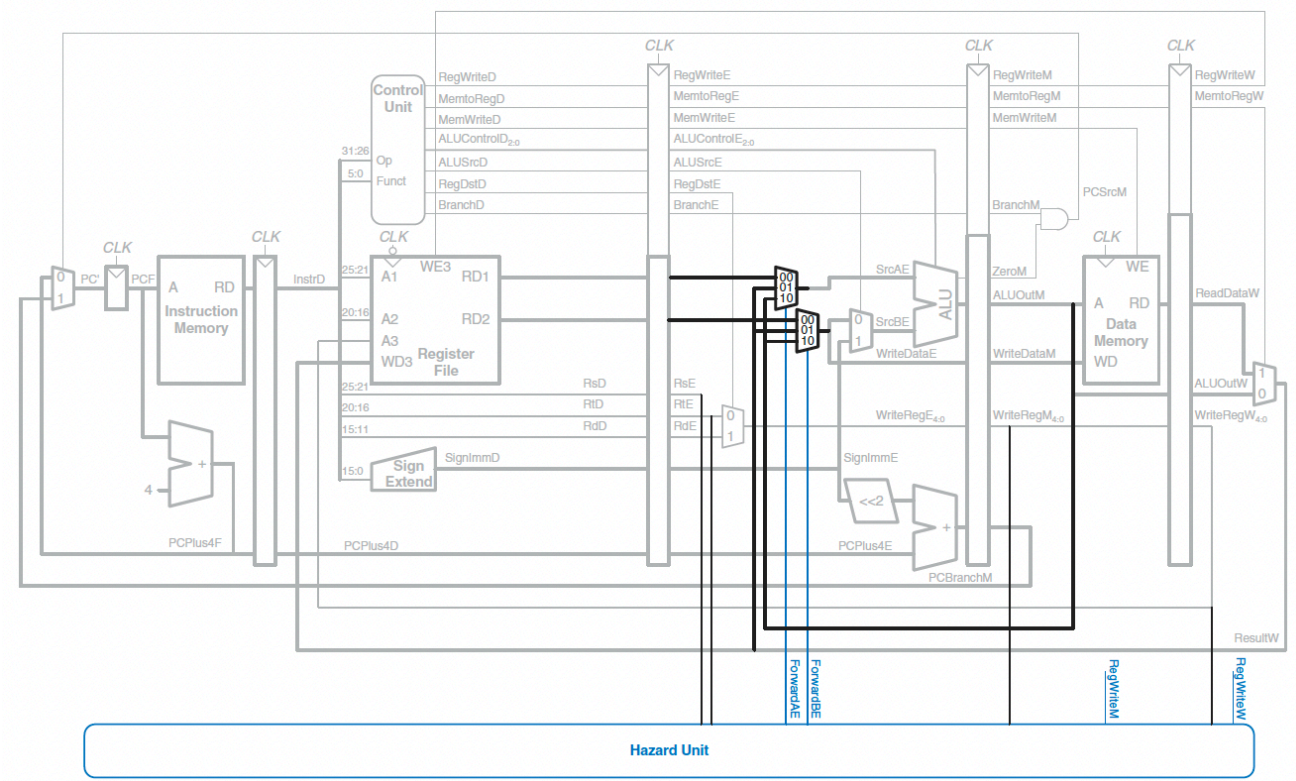
- Some data hazards are solved by simply forwarding the data **from the memory or writeback stage to the execution stage**.
- When is this necessary?
  - Instruction I1 in Execute stage has a source register matching the destination register of instruction I2 in Memory or Writeback stage.



- Notice in cycle 4, `$s0` can be forwarded from memory stage of `add` to execution stage of `and`.
- Also, in cycle 5, `$s0` can be forwarded from writeback stage of `add` to execution stage of `or`.

Solving Data Hazards With Forwarding

- How is forwarding implemented?
  - We need a hazard detection unit (HDU) and two multiplexers in front of the ALU to select the operands from the register file or the forwarded signals.
  - HDU receives the **two source registers** from the instruction in the *execute* stage and the **destination register** from the instructions in the *memory* and *writeback* stage.
  - HDU also receives the **RegWrite** signals (controls if write will happen to the register?)
  - Based on the registers and the RegWrite signals, it generates control signals for the ALU multiplexers.
  - The **RegWrite** signals are indicated by short wires labelled with RegWriteX to avoid clutter

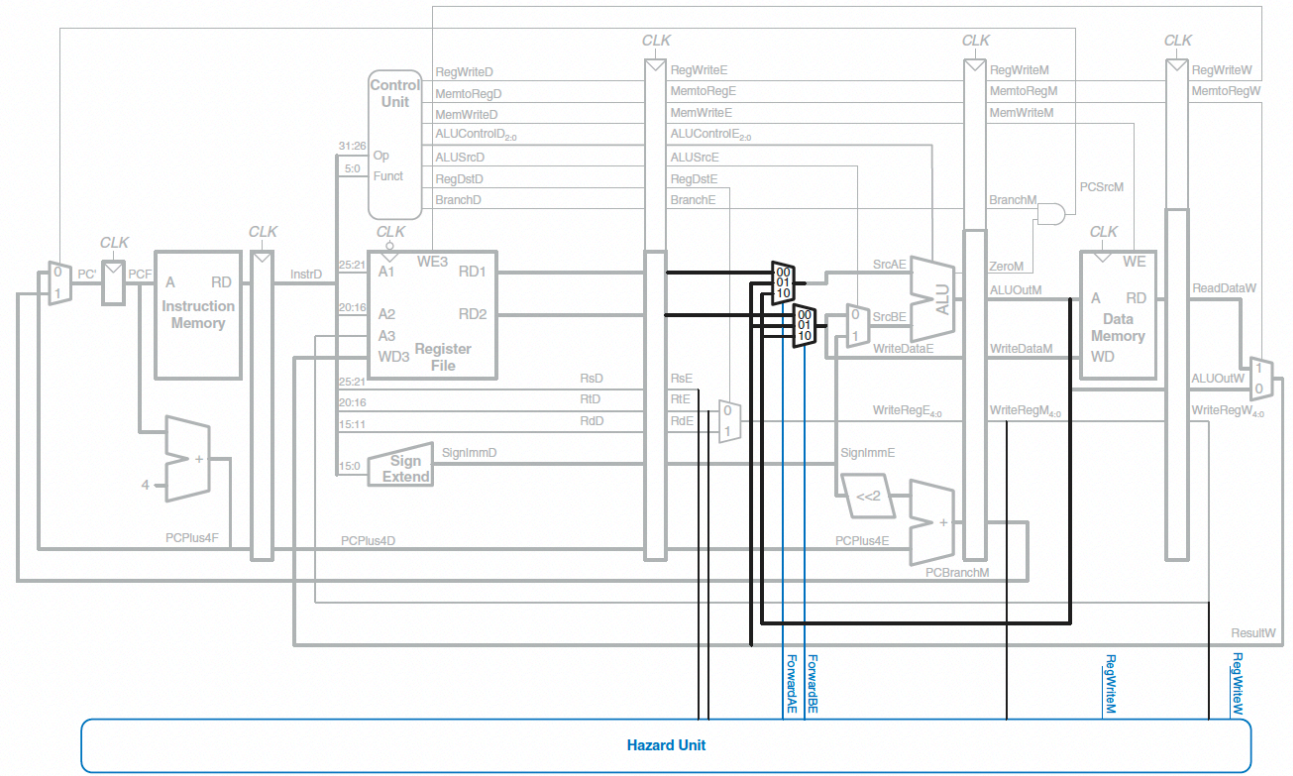


Solving Data Hazards With Forwarding

- Forwarding logic for `SrcA`:

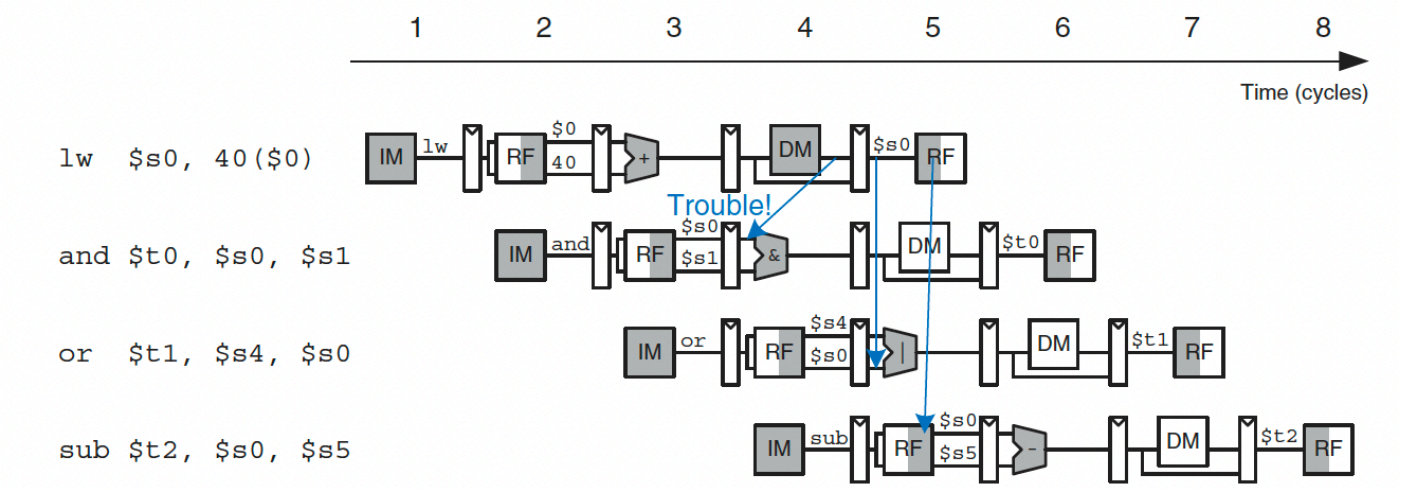
```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
                                ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
                                ForwardAE = 01
else                            ForwardAE = 00
```

- Note `rs` cannot be 0 as `$0` is hardwired and should not be forwarded.
- If Memory and Writeback contain matching destinations, then memory is prioritized.
- The logic for `SrcB` is the same except, `rs` is replaced by `rt`.



Solving Data Hazards With Stalls

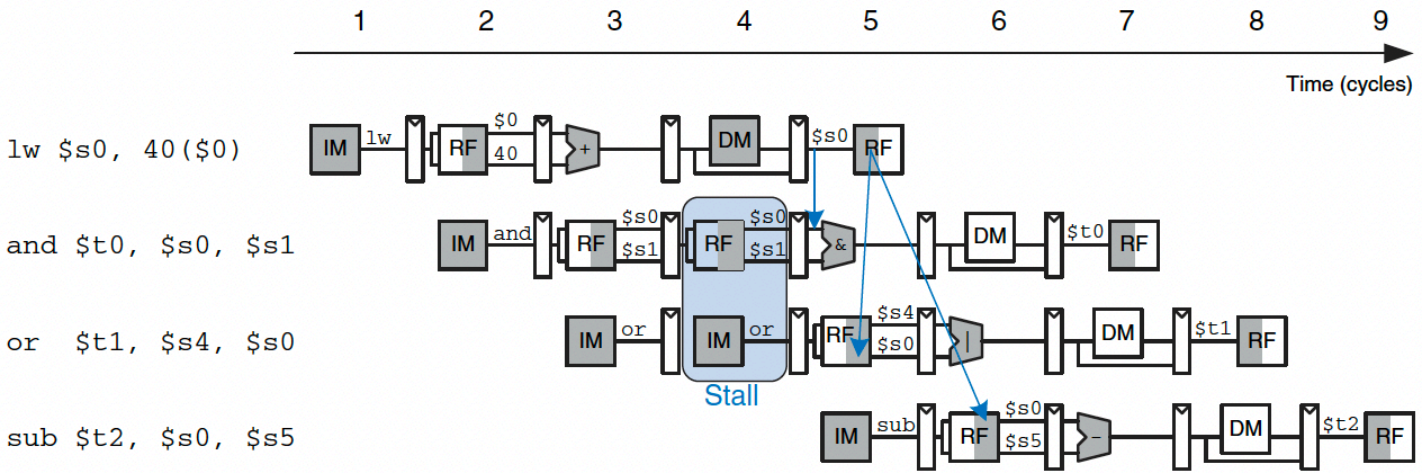
- Consider an instruction whose result is not available until 2 cycles later. E.g.,
  - `lw` receives data from memory at the end of cycle 4.
  - But `and` needs the result beginning of cycle 4!



- Such instructions are said to have a **two-cycle latency**.
- Forwarding *cannot solve data hazards* involving two-cycle latency instructions.

Solving Data Hazards With Stalls

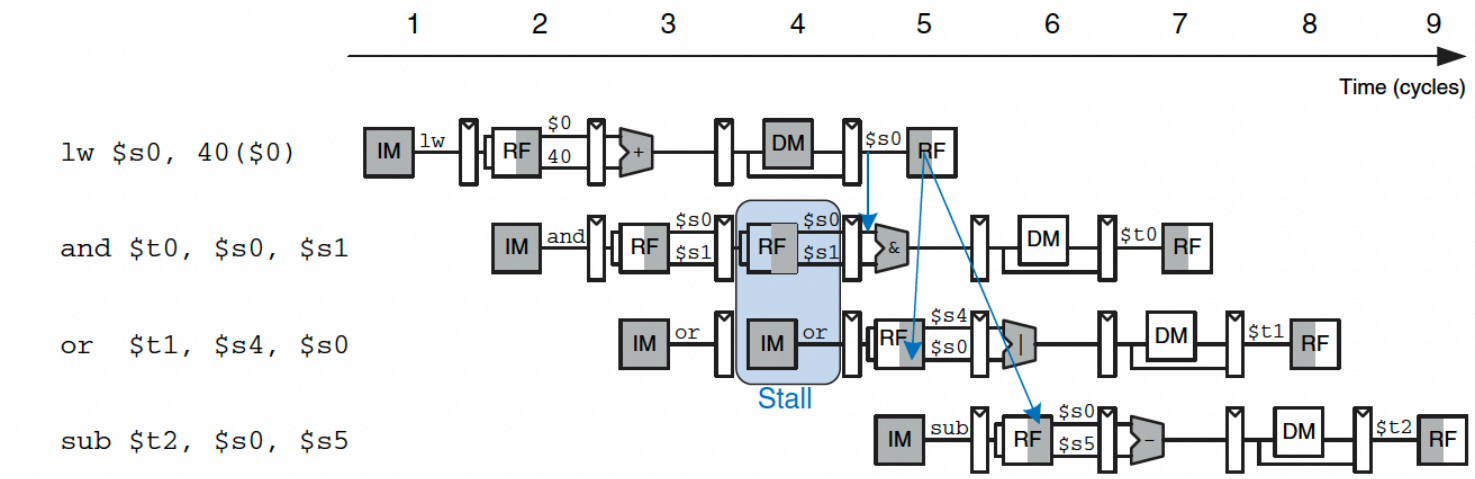
- The alternative is to **stall** the pipeline, i.e., hold up operation till data is available
- So, `and` enters decode stage in cycle 3 and is stalled or delayed till cycle 4.
- This stalls all subsequent instructions too!



- Stalling allows result to be forwarded from Writeback stage of `lw` to Execute stage of `and`.
- In cycle 5, `or` reads from register file, forwarding not needed.

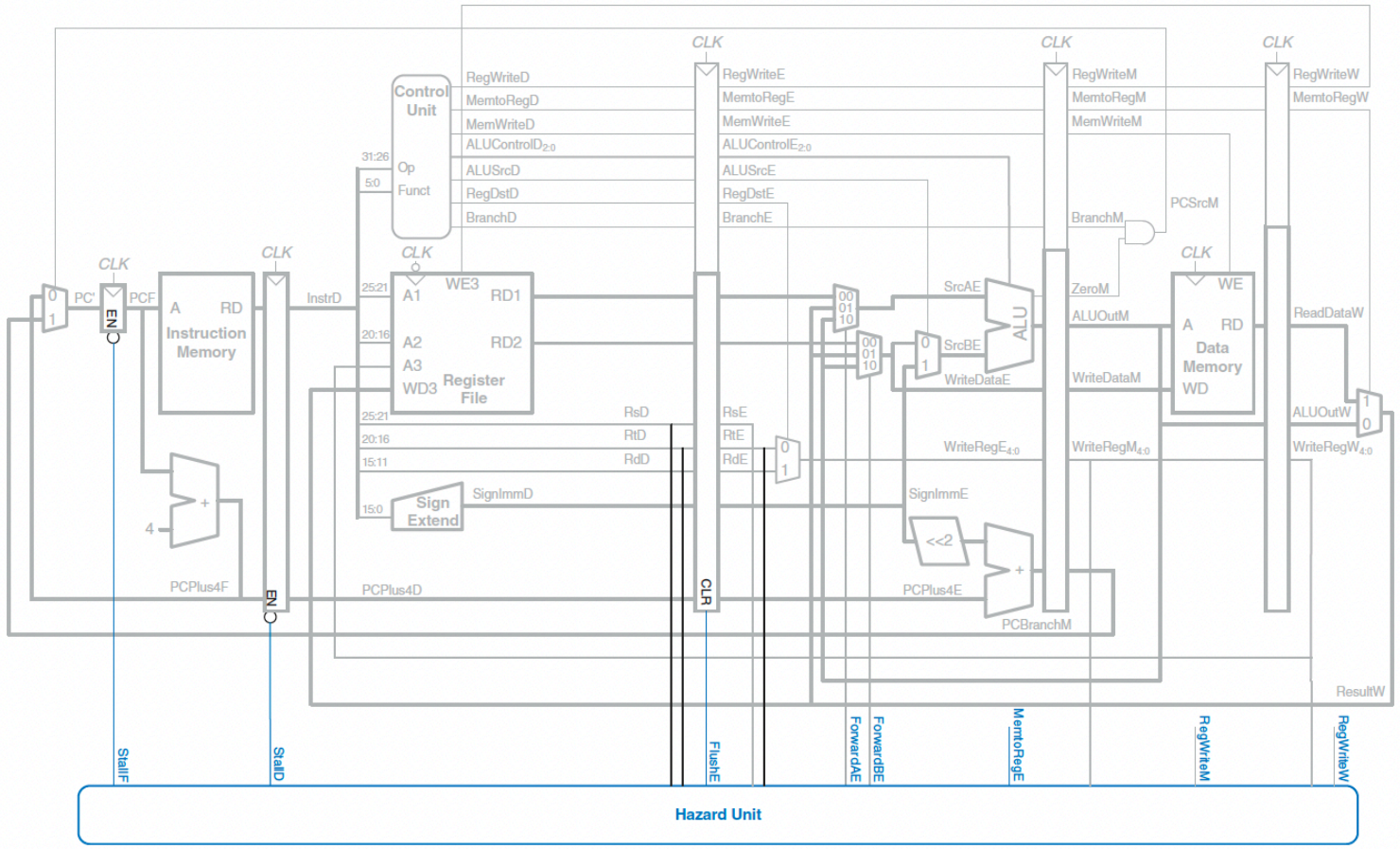
Solving Data Hazards With Stalls

- How is stalling implemented?
  - Disable the pipeline register so contents do not change
  - When a stage is stalled, all previous stages must be stalled, so no subsequent instructions are lost
  - The pipeline register directly after the stalled stage must be cleared to avoid propagating bogus information
  - Stalling degrades performance and should be used only when necessary.
- Notice how Execute is unused in cycle 4, so is Memory in cycle 5.
- The unused stage (also called **bubble**) propagates through the pipeline and behaves like a *nop* instruction.



Solving Data Hazards With Stalls

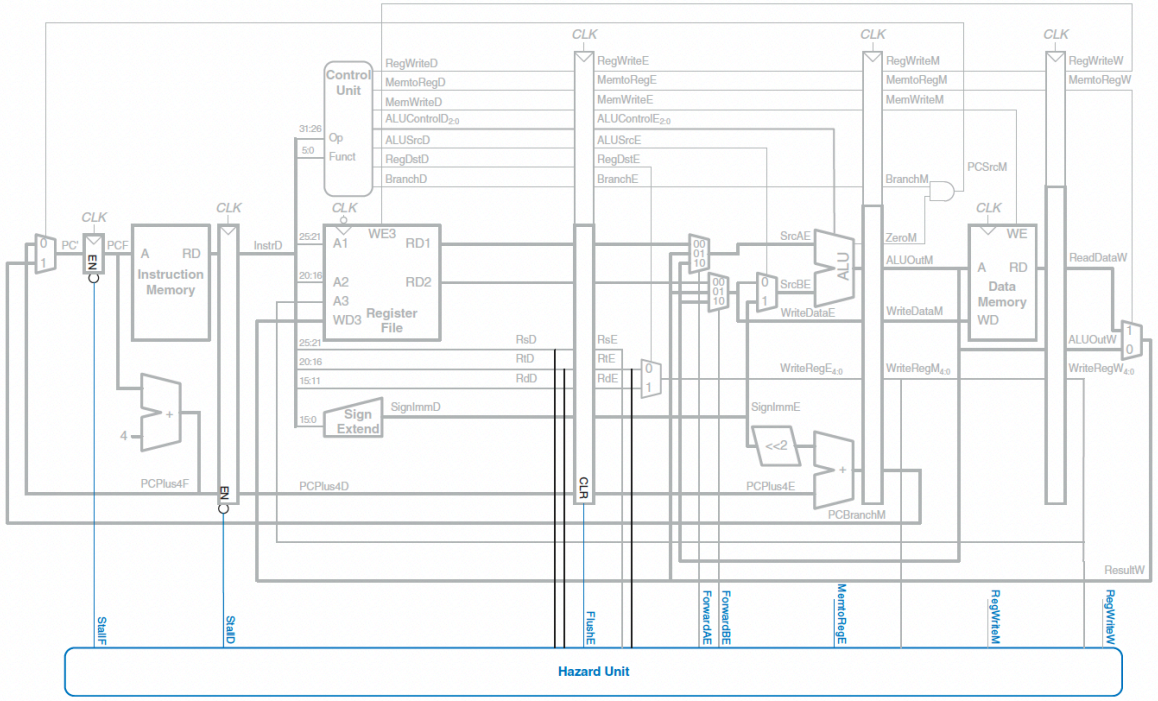
- For a `lw` stall, the hazard unit
  - Examines the instruction in the Execute stage.
  - If instruction is lw, then its destination register is matched with either of the source operands in the decode stage.
  - If match occurs, then instruction in the Decode stage is stalled until the source operand is ready.
- Stalls are supported by adding adding enable inputs (EN) to the Fetch and Decode pipelines and a synchronous reset/clear (CLR) input to the Execute pipeline register.
- When a `lw` stall occurs,
  - StallD and StallF are asserted to hold old values in the respective pipeline registers and
  - FlushE is asserted to clear the Execute stage pipeline
  - The MemtoRegE signal is asserted for the lw instruction




```
lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE
StallF  = StallD = FlushE = lwstall
```

Control Hazards

- Branch instructions (e.g., `beq`) present a control hazard.
- The branch decision (based on `PCSrc`) is not available in time to make the correct decision about the next instruction to fetch!

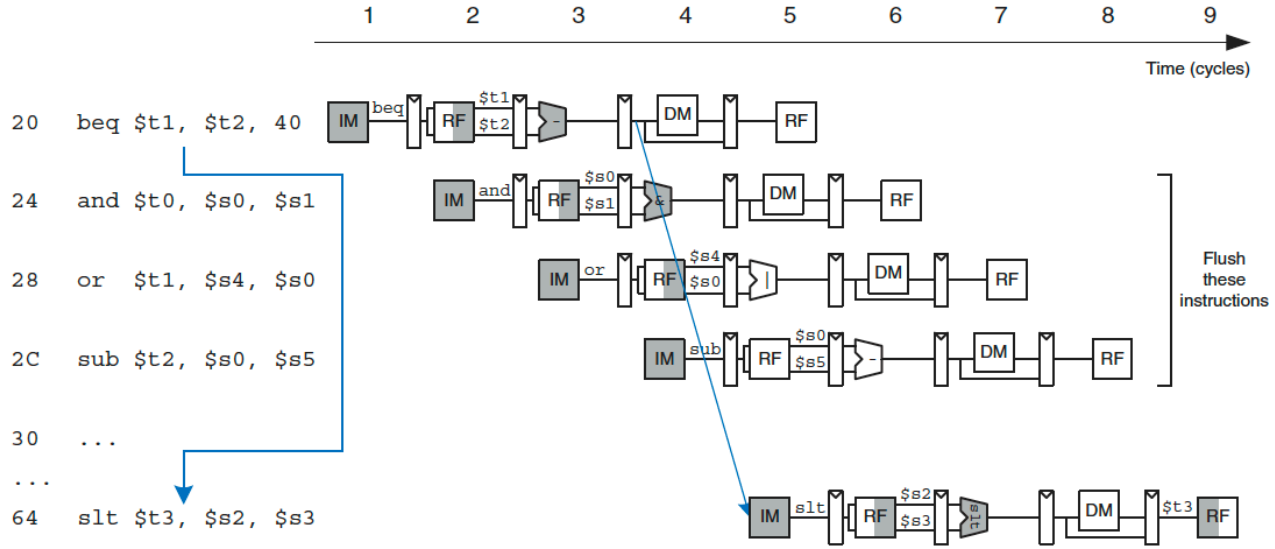


Solving Control Hazards

- One way to is to stall the pipeline for 3 cycles
- Why 3 cycles?
  - Because the decision is made in the *Memory* stage.
- This will degrade performance!

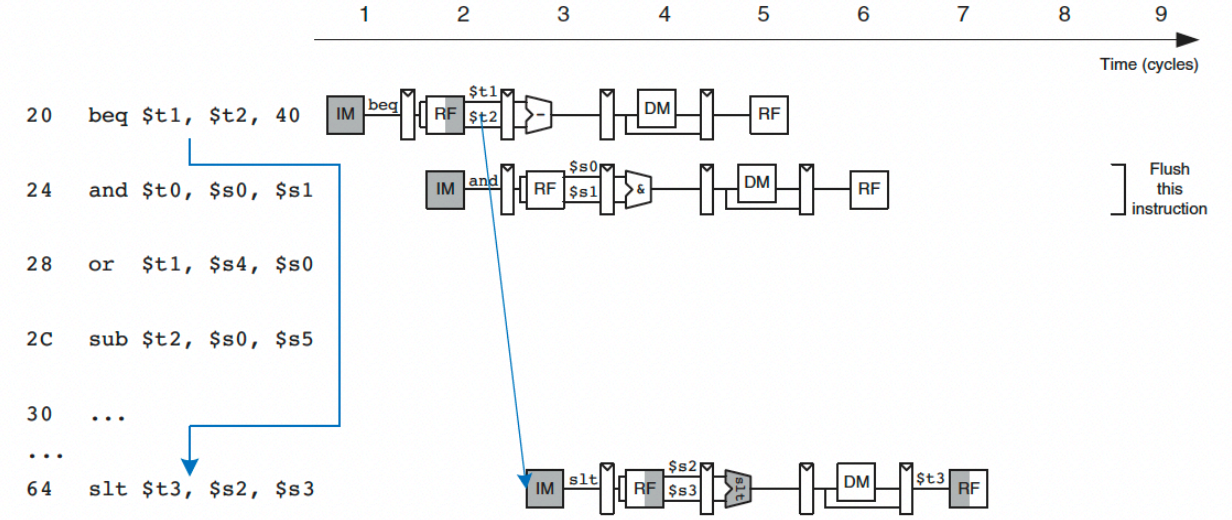
Solving Control Hazards

- The alternative is **branch prediction**
  - Predict branch is not taken and continue execution.
  - When decision is available, either continue (prediction was correct) or flush out the instructions executed as a result of the wrong prediction and fetch the correct instruction.
  - The cost of incorrect prediction wastes cycles and is called the **branch misprediction penalty**.



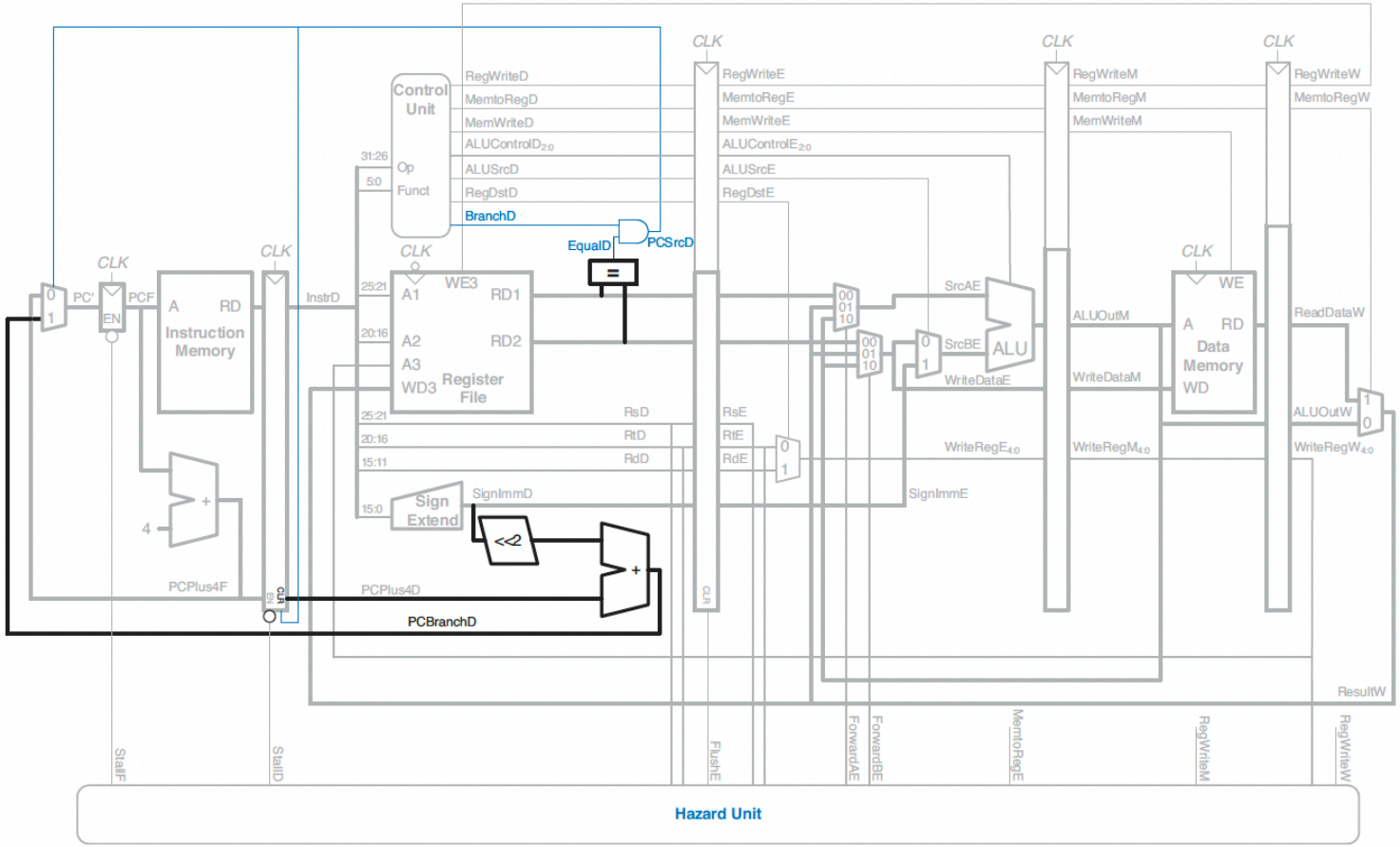
Solving Control Hazards

- We can reduce **branch misprediction penalty** by making the prediction earlier (if possible).
- For example, we could move the prediction logic to the decode stage of the pipeline.
- Hence, we will only have to flush out 1 instruction, in case of misprediction.



Solving Control Hazards

- Making early branch prediction requires changes to the hardware:
  - An equality comparator in the decode stage for the branch comparison.
  - The `PCSrc` AND gate moved to the Decode stage for accurate branch decision.
  - The `PCBranch` adder moved to the Decode stage for accurate computation of destination address.
  - `PCSrcD` must be connected to the CLR input of the pipelined register in the Decode stage so wrong instructions can be flushed.



Solving Control Hazards

- Unfortunately, early branch prediction causes RAW hazards!
  - If one of the source operands is written by a previous instruction and the operand is not available yet.
- How do we resolve these hazards?
  - Forwarding or stalling

Solving Control Hazards

- The pipelined processor needs to ne modified to handle the Decode stage dependencies.
  - If a result is in the Writeback stage => no hazards exists as writes happen in 1st half and reads in 2nd half of the cycle.
  - If ALUResult is in the Memory stage => forward ALUResult to the equality comparator through two new multiplexers
  - If ALUResult is in the Execute stage => stall at the Decode stage till result is ready
  - If `lw` result is in the Memory stage => stall at the Decode stage till result is ready

Solving Control Hazards

- Forwarding logic:

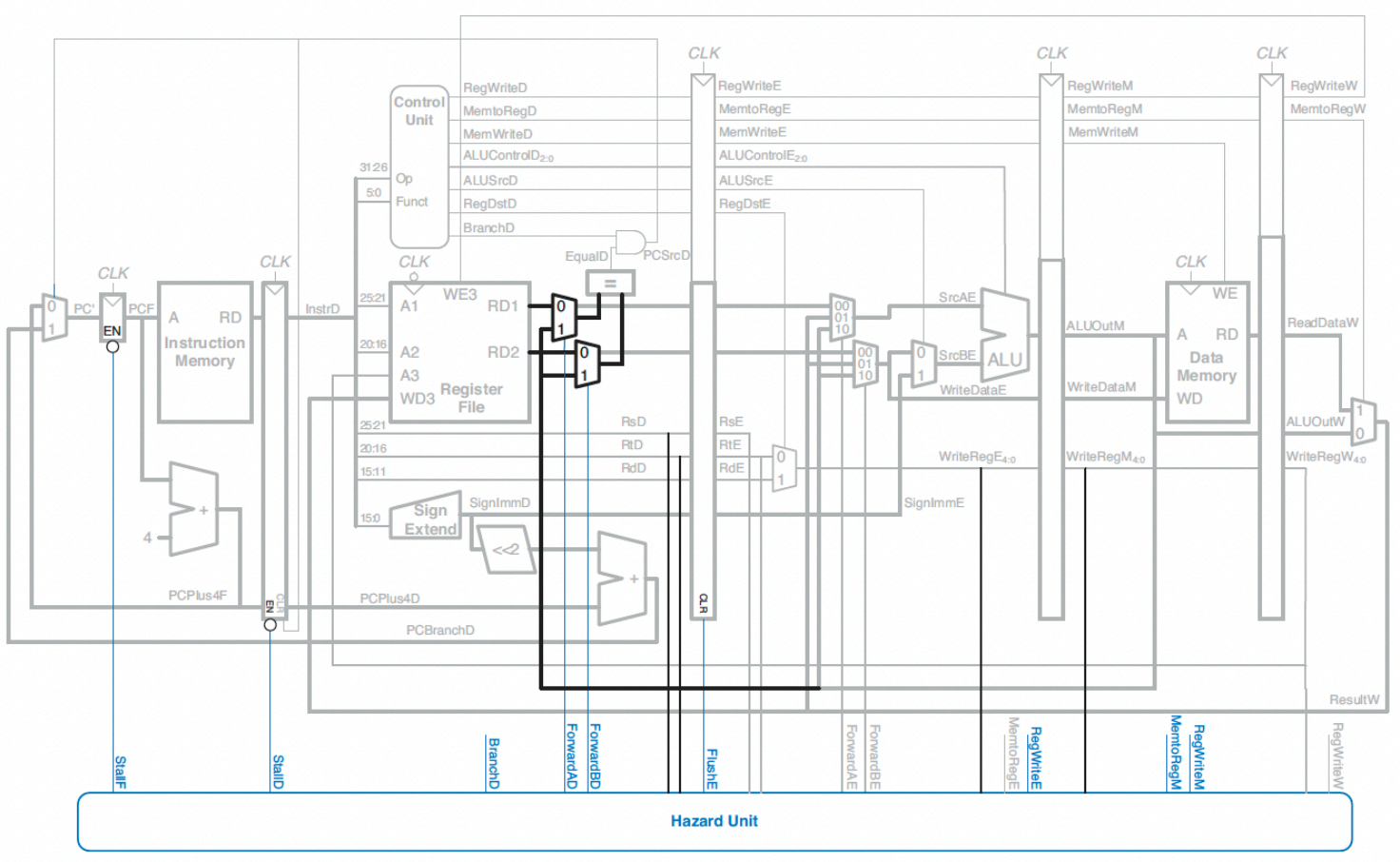
```
ForwardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM
```

- Stall logic:

```
branchstall =
   BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)
                          OR
   BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)
```

- Processor may stall due to a load or a branch hazard:

```
StallF = StallD = FlushE = lwstall OR branchstall
```



Hazard Summary

- RAW hazards occur when an instruction depends on the result of another instruction that hasn't been written to the register file yet.
- RAW hazards are resolved by
  - Forwarding if the result is computed in time.
  - Stalling the pipeline until result is available.
- Control hazards occur when the decision of next instruction has not been made in time.
- Control hazards are resolved by branch prediction and flushing instructions when prediction is wrong.
- Flushing instructions can be minimized by early branch prediction.

Example #1: Hazards

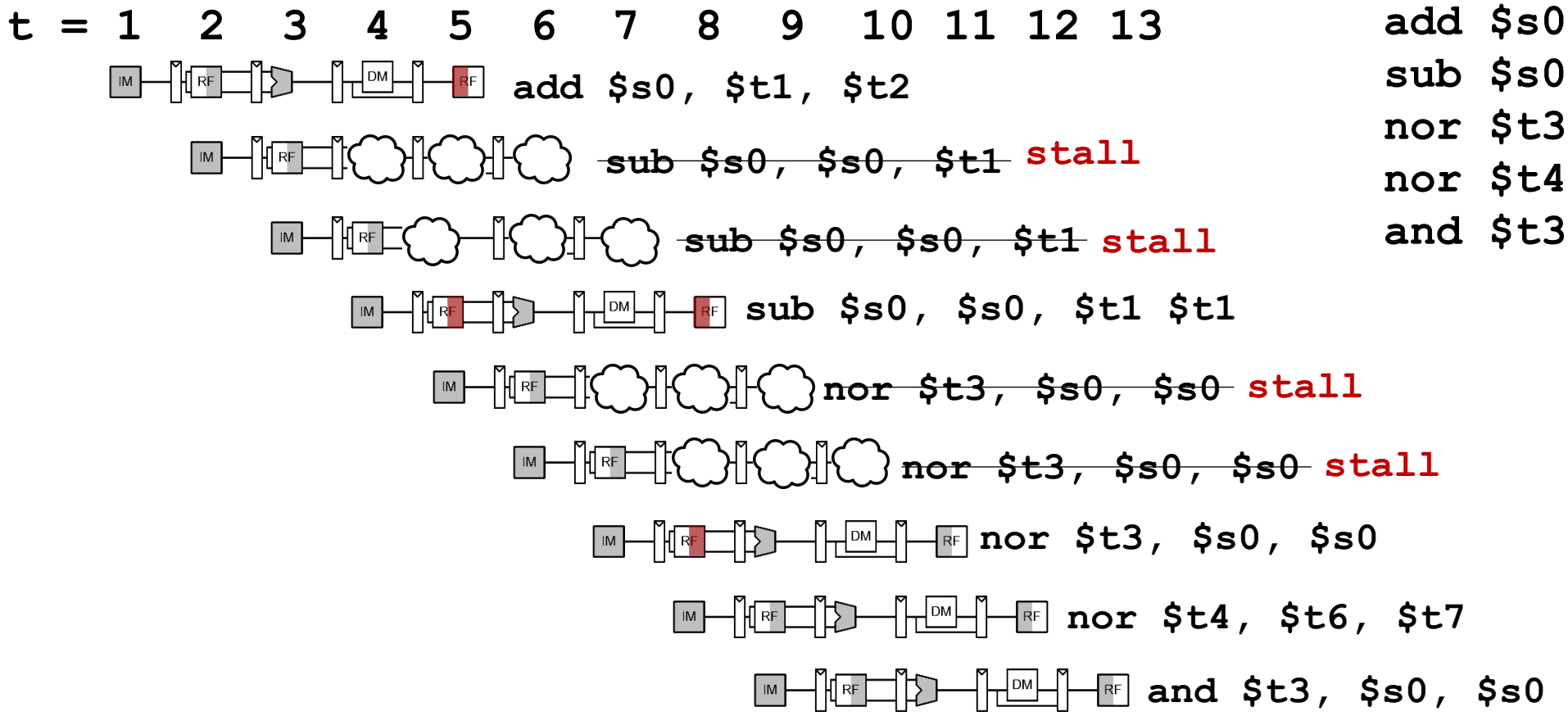
- Suppose we have the following MIPS code. Circle registers to indicate dependencies between instructions that cause RAW hazards.

```
add $s0, $t1, $t2
sub $s1, $s0, $t1
nor $t3, $t7, $s1
nor $t4, $t6, $t7
and $t3, $t4, $s0
```

Example #2: Hazards

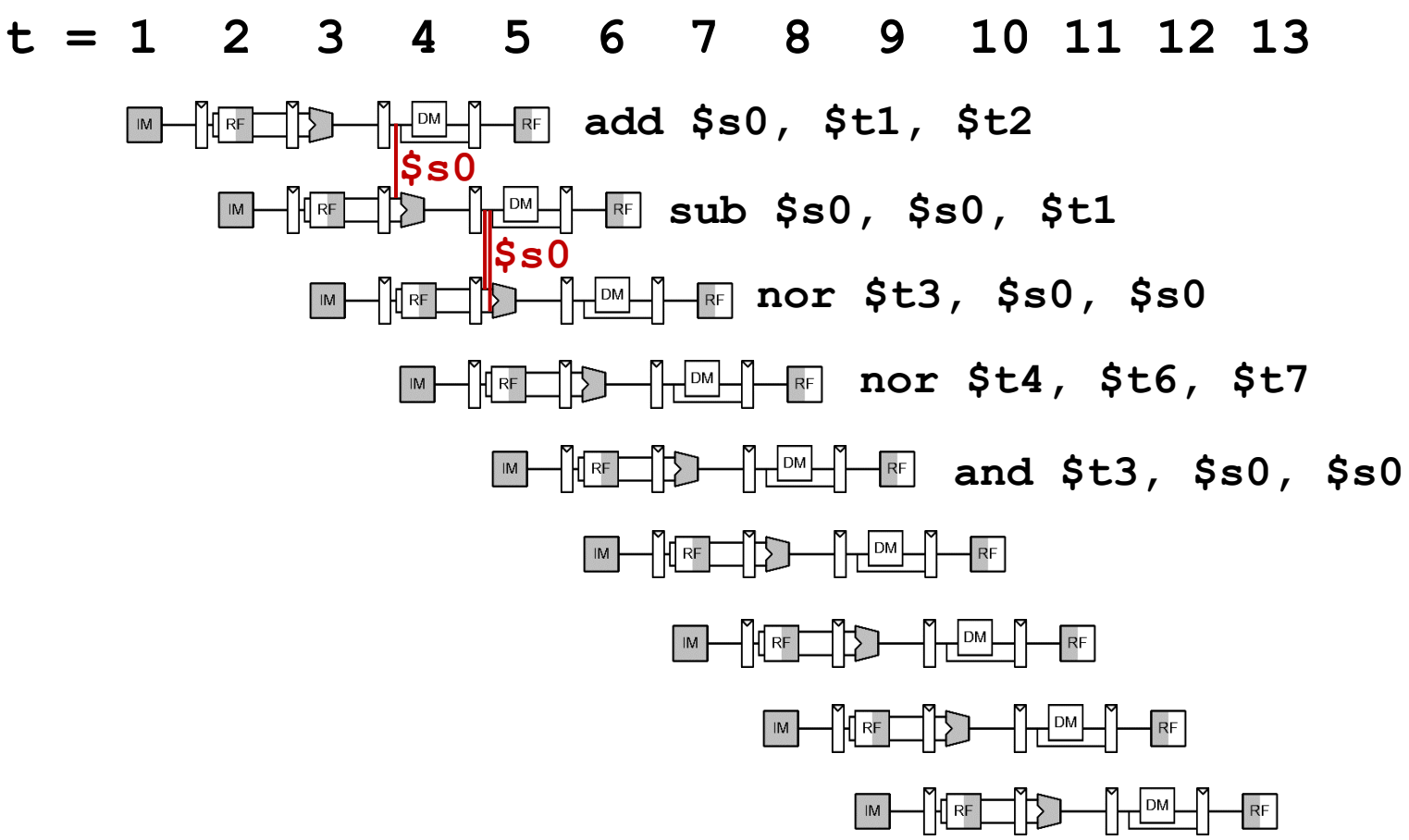
- Assuming no forwarding hardware is available, draw bubbles to show how the pipeline would be stalled to correctly execute the code.

```
add $s0, $t1, $t2
sub $s0, $s0, $t1
nor $t3, $s0, $s0
nor $t4, $t6, $t7
and $t3, $s0, $s0
```



Example #3: Hazards

- Assuming forwarding hardware is available, show the forwarding paths and stalls needed to execute the same instructions

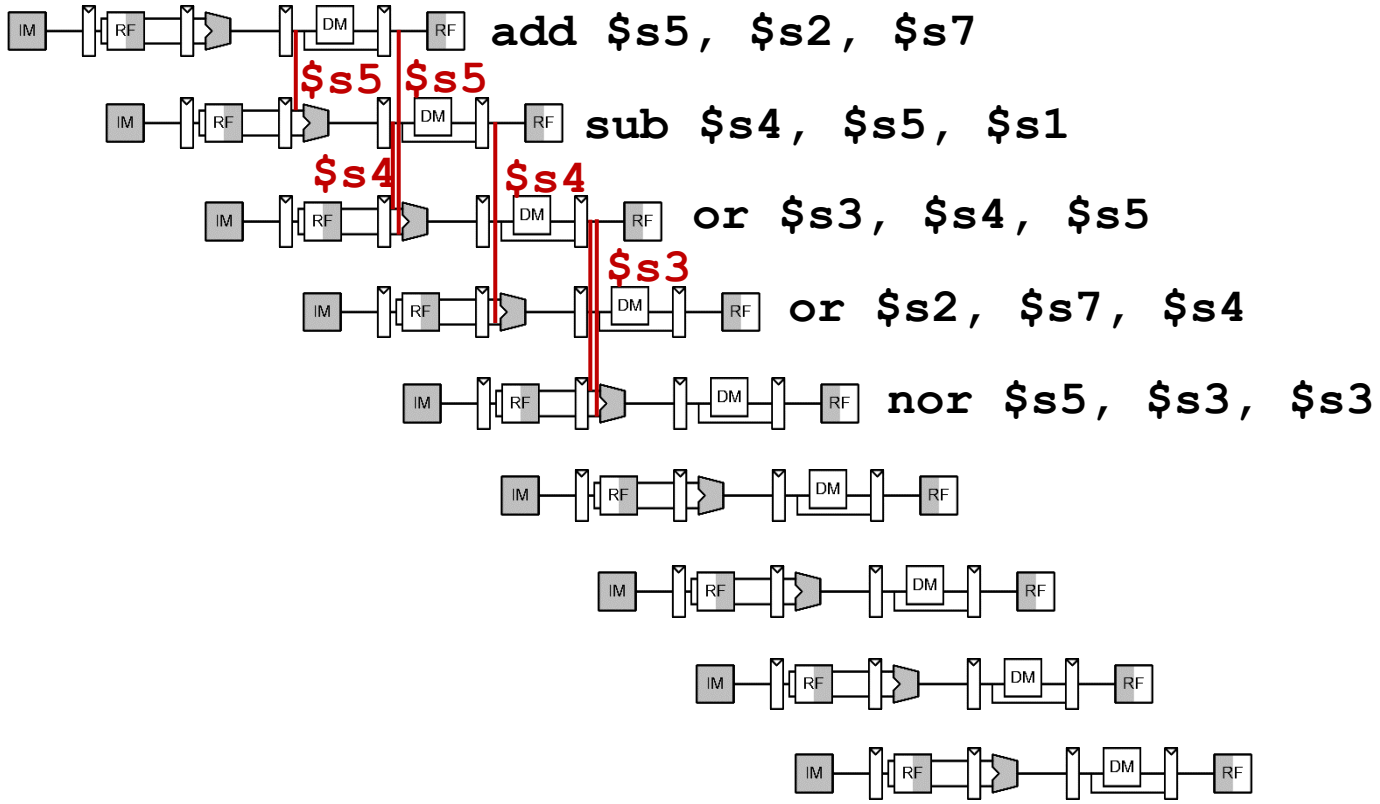



```
add $s0, $t1, $t2
sub $s0, $s0, $t1
nor $t3, $s0, $s0
nor $t4, $t6, $t7
and $t3, $s0, $s0
```

Example #4: Hazards

- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions.

```
t = 1   2   3   4   5   6   7   8   9   10 11 12 13
```

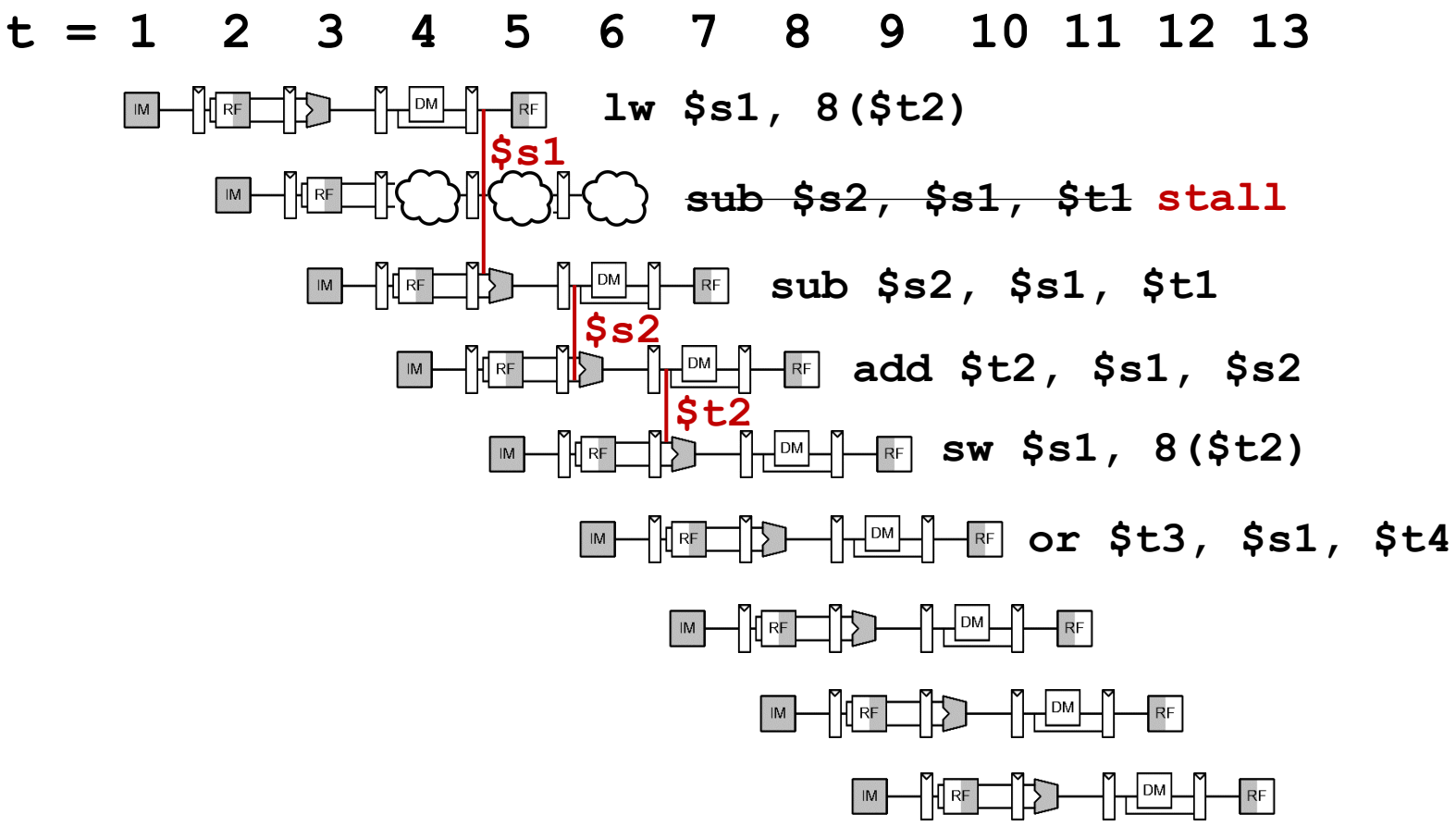



```
add $s5, $s2, $s7
sub $s4, $s5, $s1
or $s3, $s4, $s5
or $s2, $s7, $s4
nor $s5, $s3, $s3
```

Example #5: Hazards

- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions.

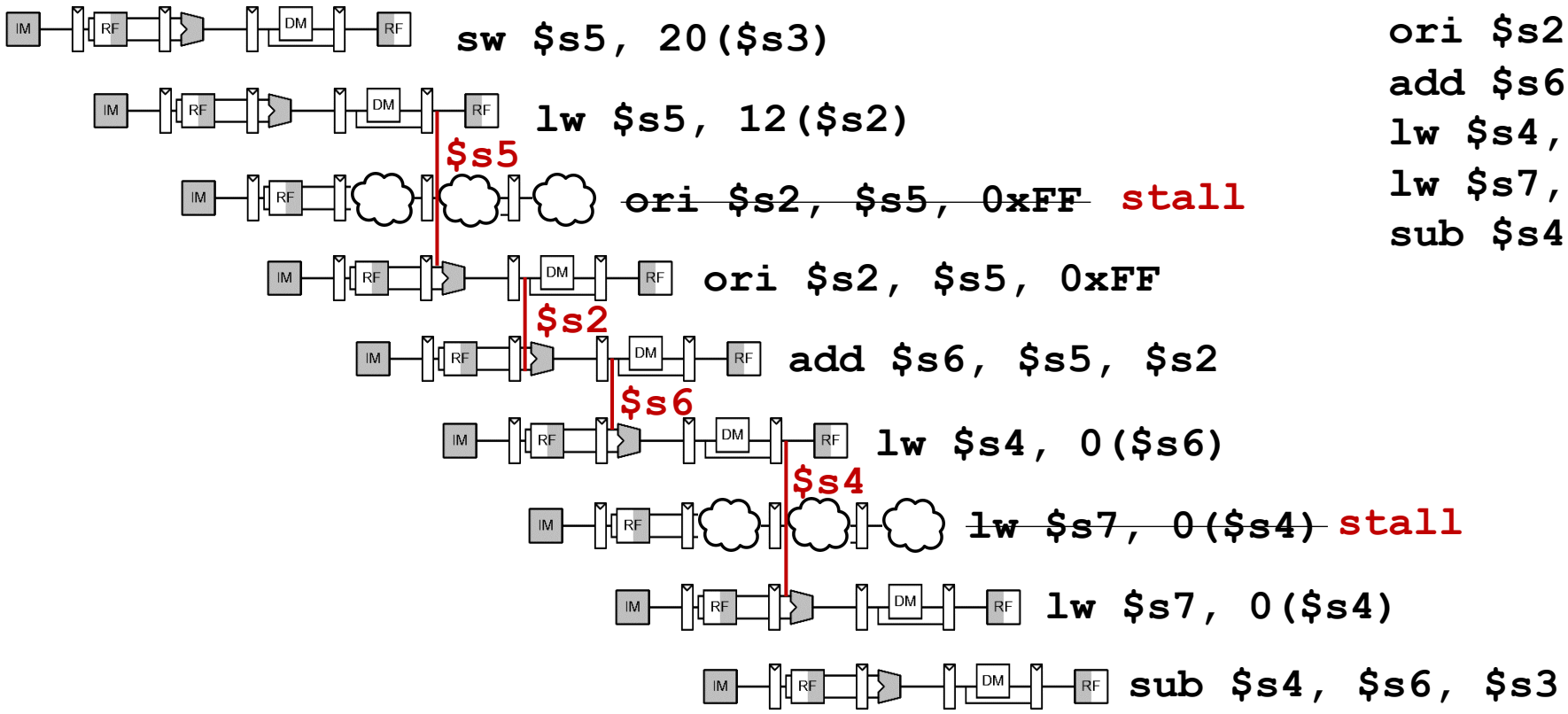
```
lw $s1, 8($t2)
sub $s2, $s1, $t1
add $t2, $s1, $s2
sw $s1, 8($t2)
or $t3, $s1, $t4
```



Example #6: Hazards

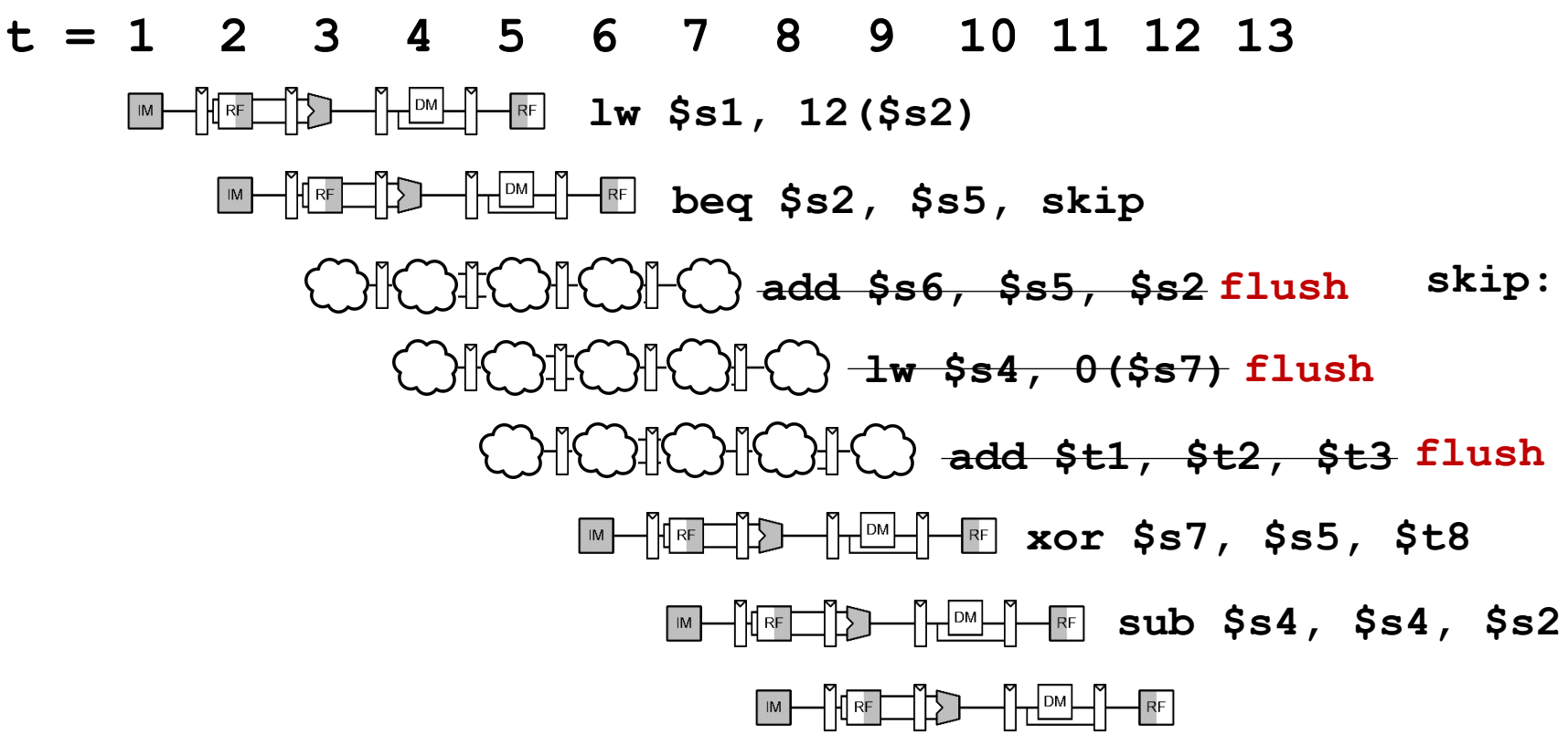
- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions.

```
sw $s5, 20($s3)
lw $s5, 12($s2)
ori $s2, $s5, 0xFF
add $s6, $s5, $s2
lw $s4, 0($s6)
lw $s7, 0($s4)
sub $s4, $s6, $s3
```



Example #7: Hazards

- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions. Assume the branch is taken and there is no comparator. Branches are predicted to be not taken.

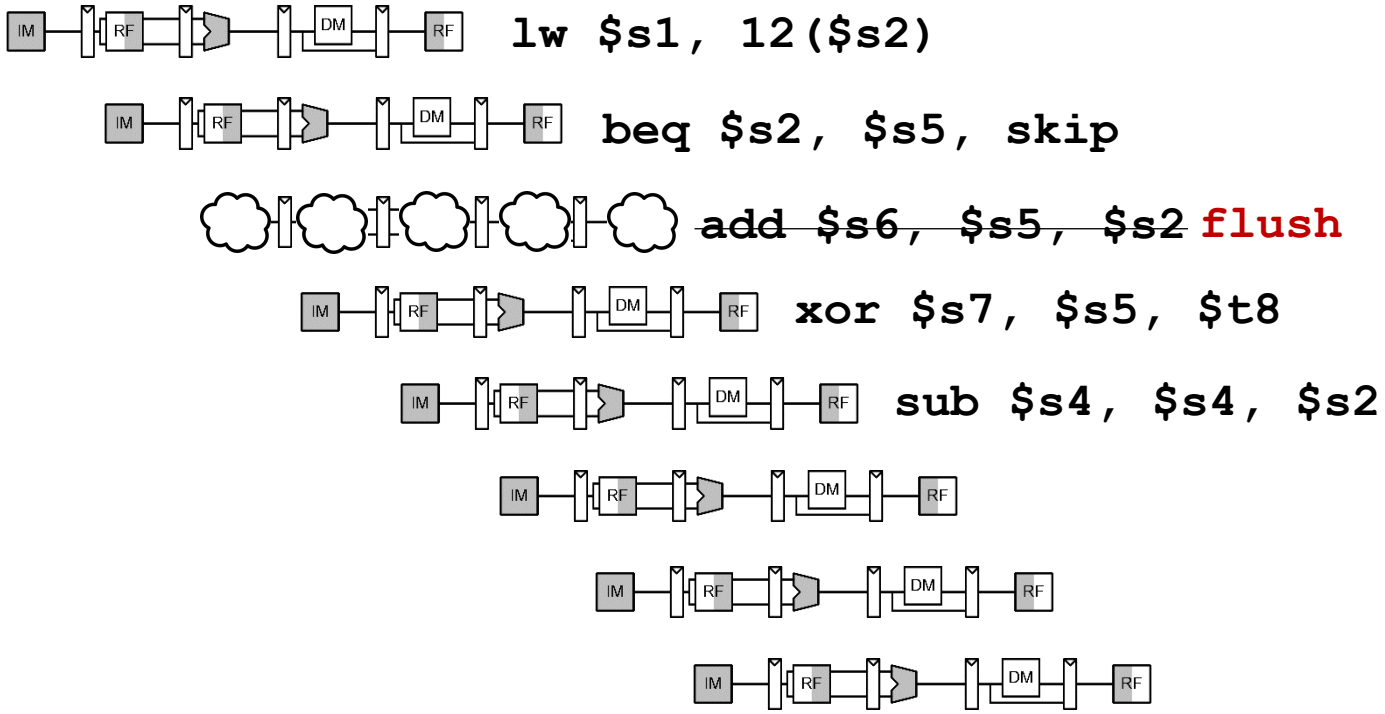



```
lw $s1, 12($s2)
beq $s2, $s5, skip
add $s6, $s5, $s2
lw $s4, 0($s7)
add $t1, $t2, $t3
skip: xor $s7, $s5, $t8
sub $s4, $s4, $s2
```

Example #8: Hazards

- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions. Assume the branch is taken and there is a comparator in the ID/RF stage.

```
t = 1   2   3   4   5   6   7   8   9   10 11 12 13
```




```
        lw $s1, 12($s2)
        beq $s2, $s5, skip
        add $s6, $s5, $s2
        lw $s4, 0($s7)
        add $t1, $t2, $t3
skip:   xor $s7, $s5, $t8
        sub $s4, $s4, $s2
```