Pipelined MIPS Microarchitecture

Digital Design and Computer Architecture
Mohammad Sadrosadati
Frank K. Gürkaynak

http://safari.ethz.ch/ddca

Adapted from Digital Design and Computer Architecture, David Money Harris & Sarah L. Harris ©2007 Elsevier

What Will We Learn

- **How to do more per unit time**
  - Parallelism
  - Pipelining
- **Single Cycle vs Pipelined**
- **Pipelined MIPS architecture**
- **Hazards and how to solve them**
  - Data hazards
  - Control hazards
- **Performance of the Pipelined architecture**

Parallelism

- **Two types of parallelism:**
- **Spatial parallelism**
  - duplicate hardware performs multiple tasks at once
- **Temporal parallelism**
  - task is broken into multiple stages
  - also called pipelining
  - for example, an assembly line

Parallelism Definitions

- **Some definitions:**
  - ***Token***: A group of inputs processed to produce a group of outputs
  - ***Latency***: Time for one token to pass from start to end
  - ***Throughput***: The number of tokens that can be produced per unit time
- **Parallelism increases throughput.**

Parallelism Example

- **Example:**

  Ben Bitdiddle is baking cookies to celebrate the installation of his traffic light controller. It takes 5 minutes to roll the cookies and 15 minutes to bake them. After finishing one batch he immediately starts the next batch. What is the latency and throughput if Ben doesn't use parallelism?

Latency =

Throughput =

Parallelism Example

- **Example:**

  Ben Bitdiddle is baking cookies to celebrate the installation of his traffic light controller. It takes 5 minutes to roll the cookies and 15 minutes to bake them. After finishing one batch he immediately starts the next batch. What is the latency and throughput if Ben doesn't use parallelism?

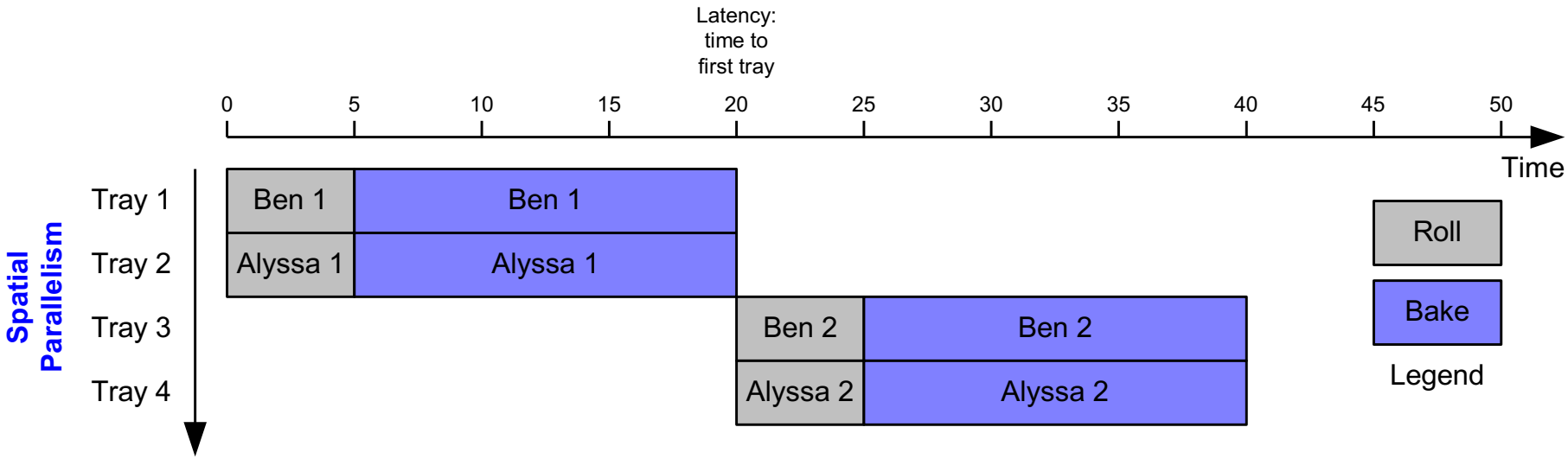
Latency **= 5 + 15 = 20 minutes** **= 1/3 hour**

Throughput **= 1 tray/ 1/3 hour** **= 3 trays/hour**

Parallelism Example

- **What is the latency and throughput if Ben uses parallelism?**
  - **Spatial parallelism**: Ben asks Allysa P. Hacker to help, using her own oven
  - **Temporal parallelism**: Ben breaks the task into two stages: roll and baking. He uses two trays. While the first batch is baking he rolls the second batch, and so on.

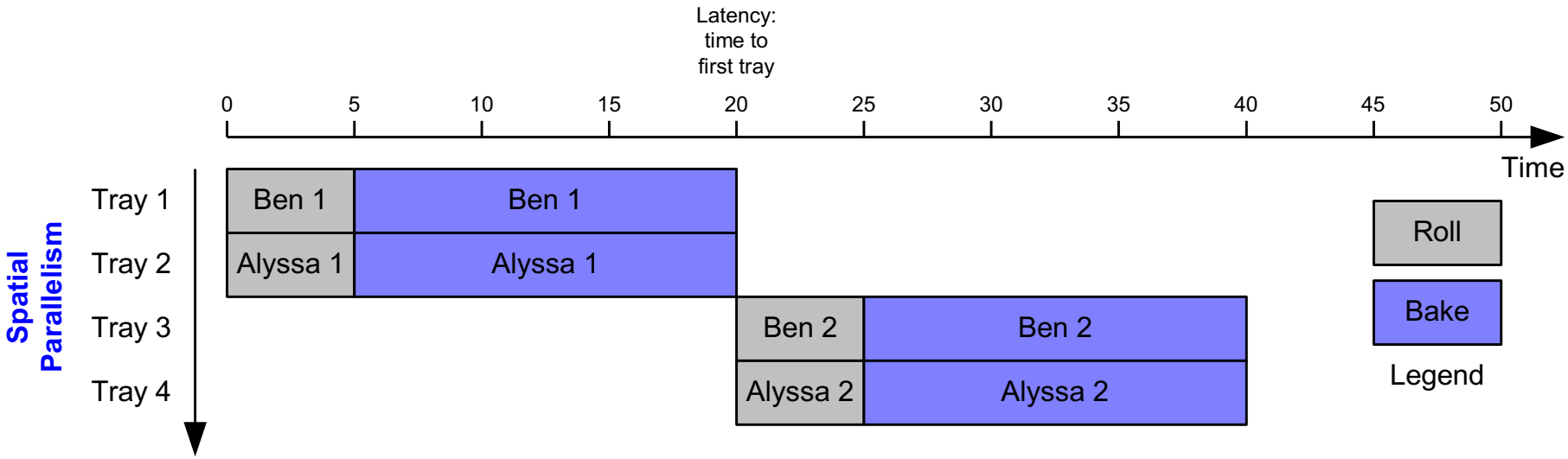
Spatial Parallelism



Latency =

Throughput=

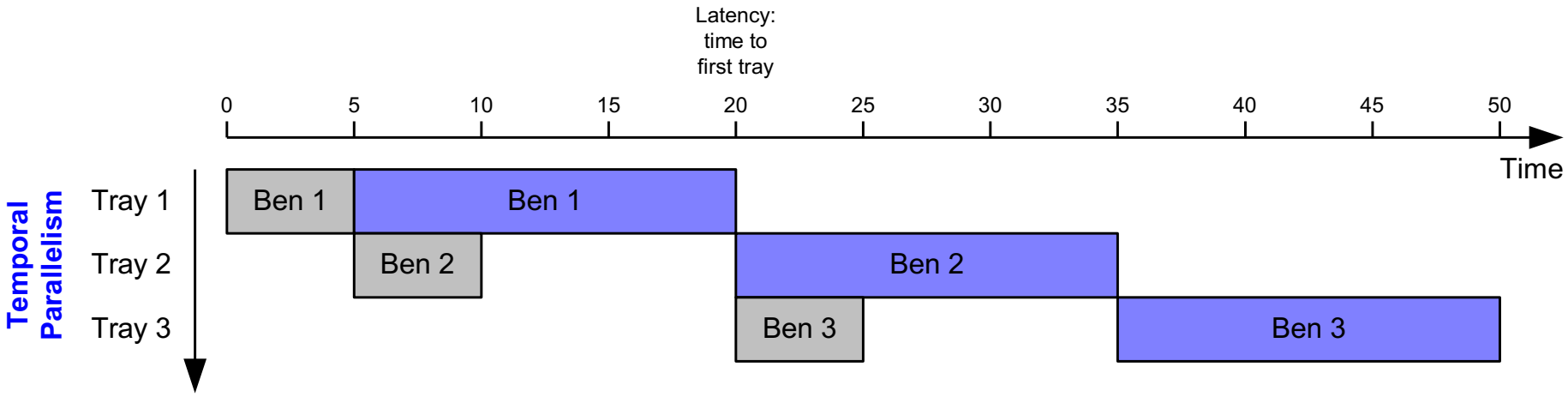
Spatial Parallelism



Latency = 5 + 15 = 20 minutes = 1/3 hour

Throughput= 2 trays/ 1/3 hour = 6 trays/hour

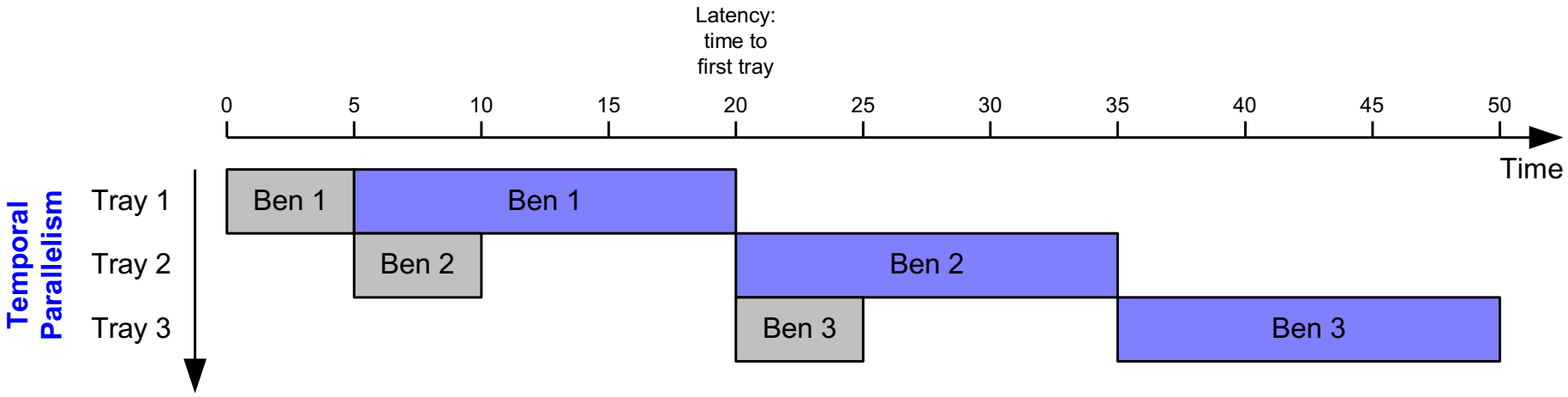
Temporal Parallelism



Latency =

Throughput=

Temporal Parallelism



Latency = 5 + 15 = 20 minutes = 1/3 hour

Throughput= 1 trays/ 1/4 hour = 4 trays/hour

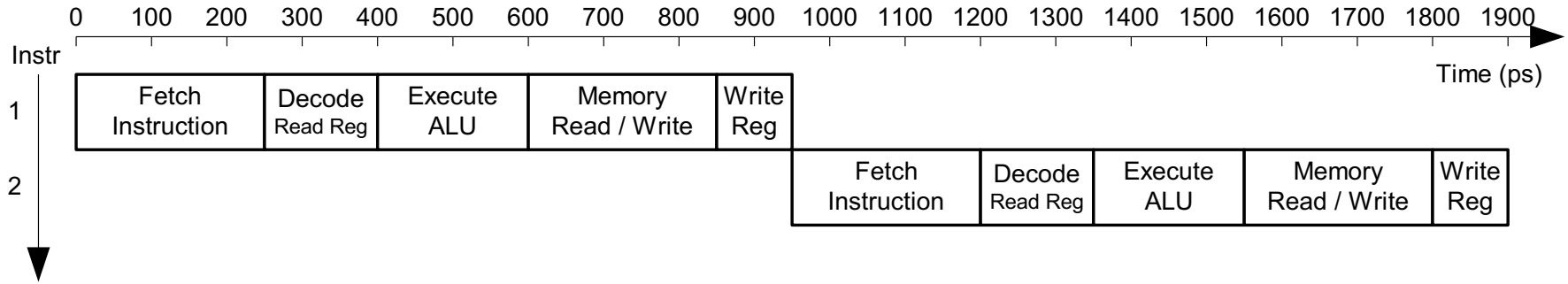
Using both techniques, the throughput would be 8 trays/hour

Pipelined MIPS Processor

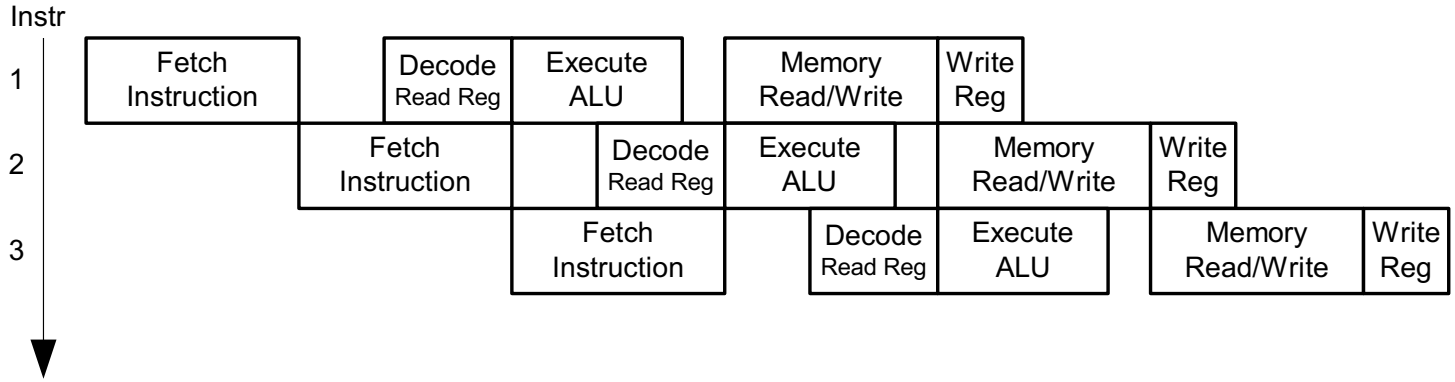
- **Temporal parallelism**
- **Divide single-cycle processor into 5 stages:**
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- **Add pipeline registers between stages**

Single-Cycle vs. Pipelined Performance

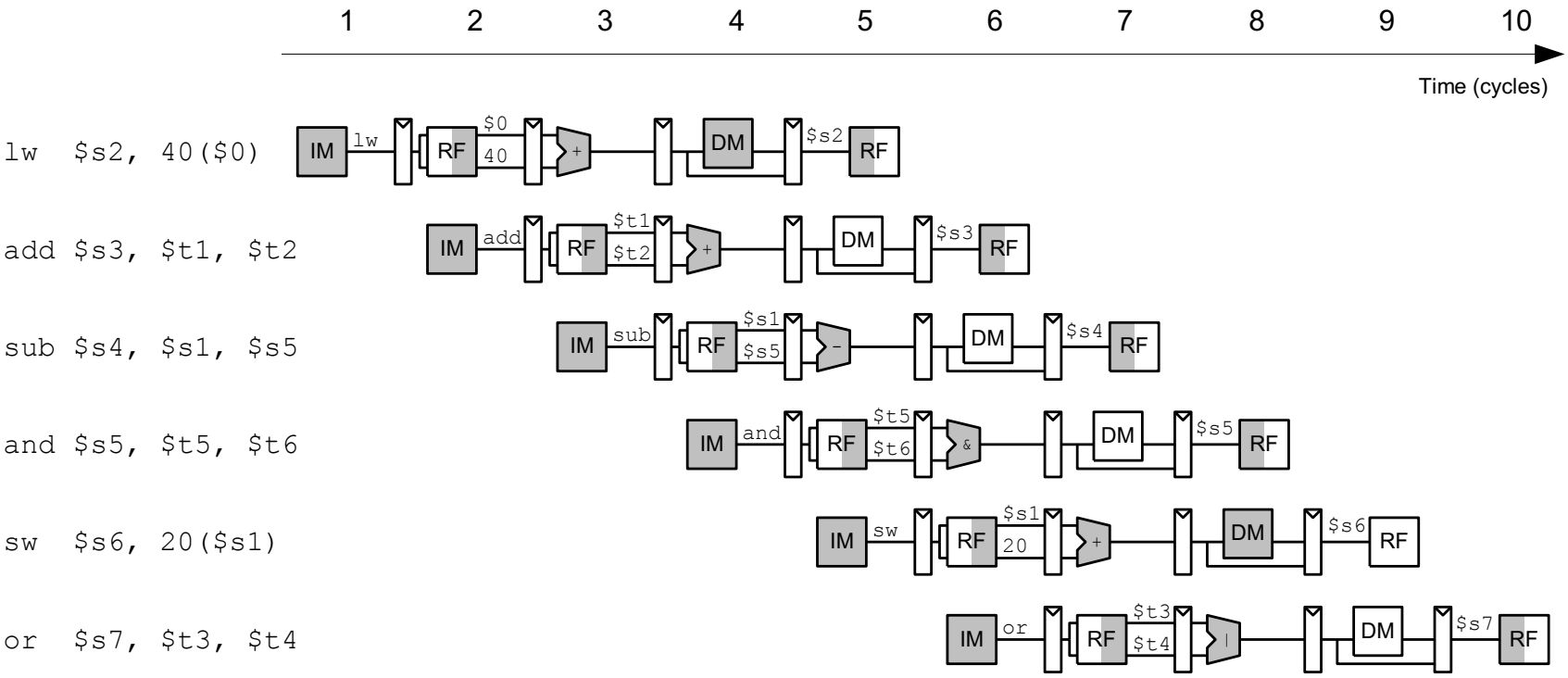
Single-Cycle



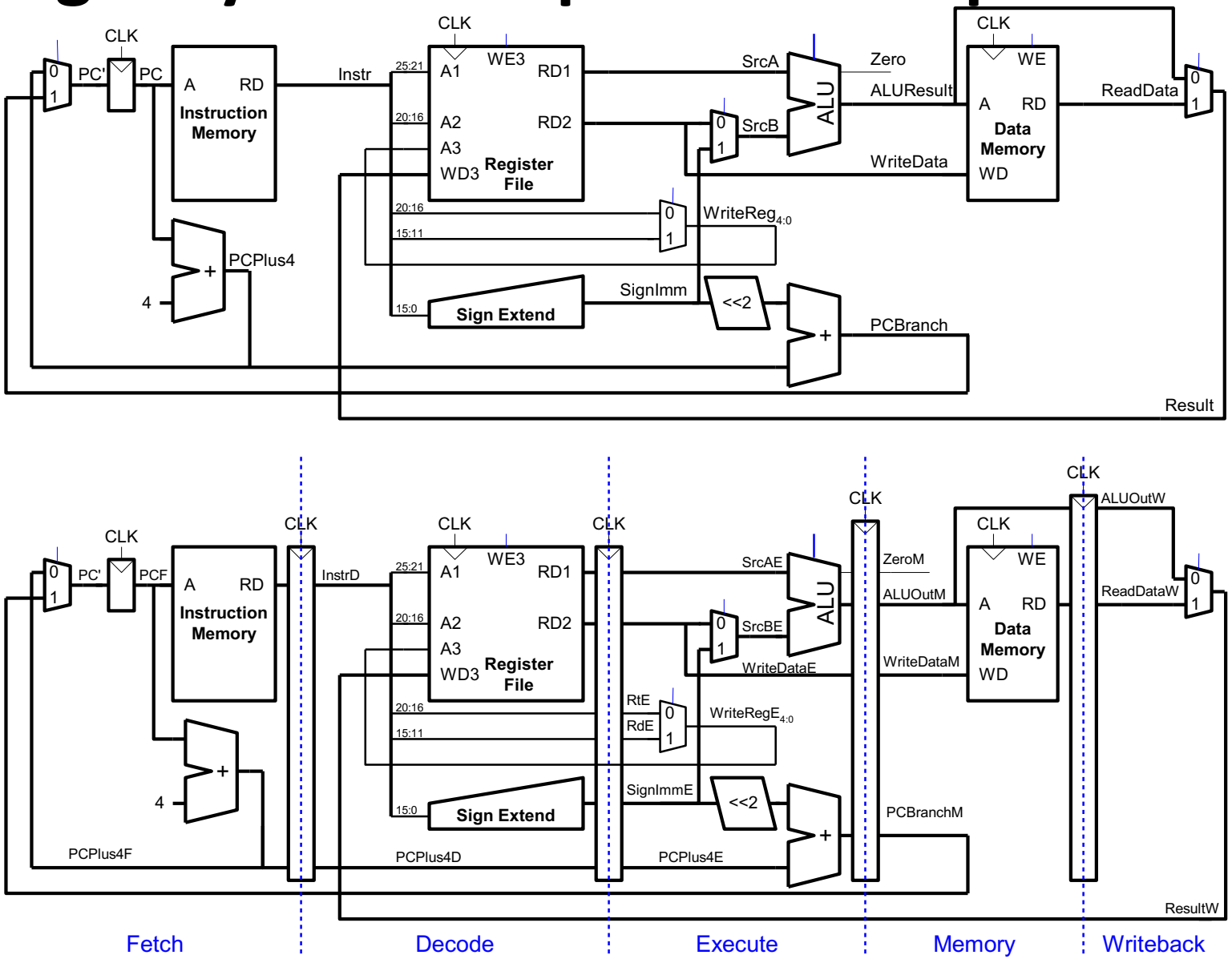
Pipelined



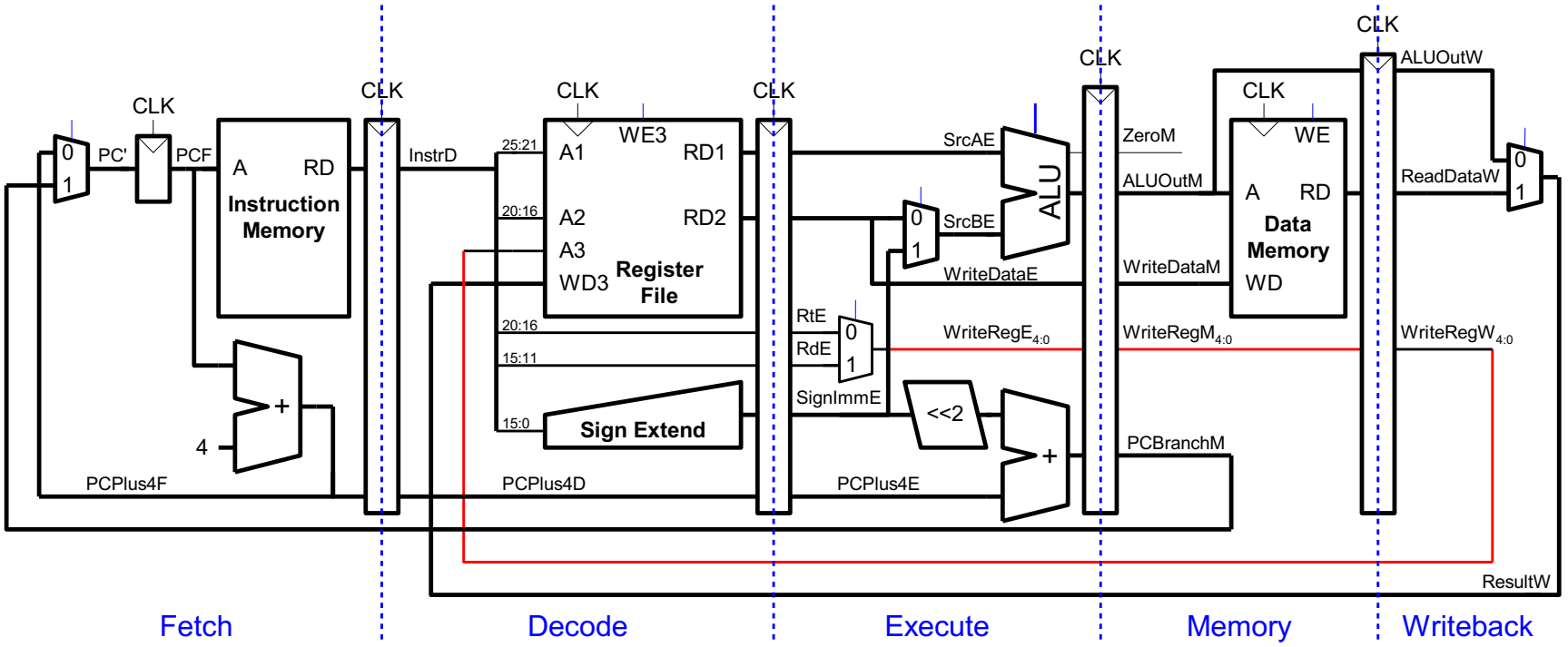
Pipelining Abstraction



Single-Cycle and Pipelined Datapath

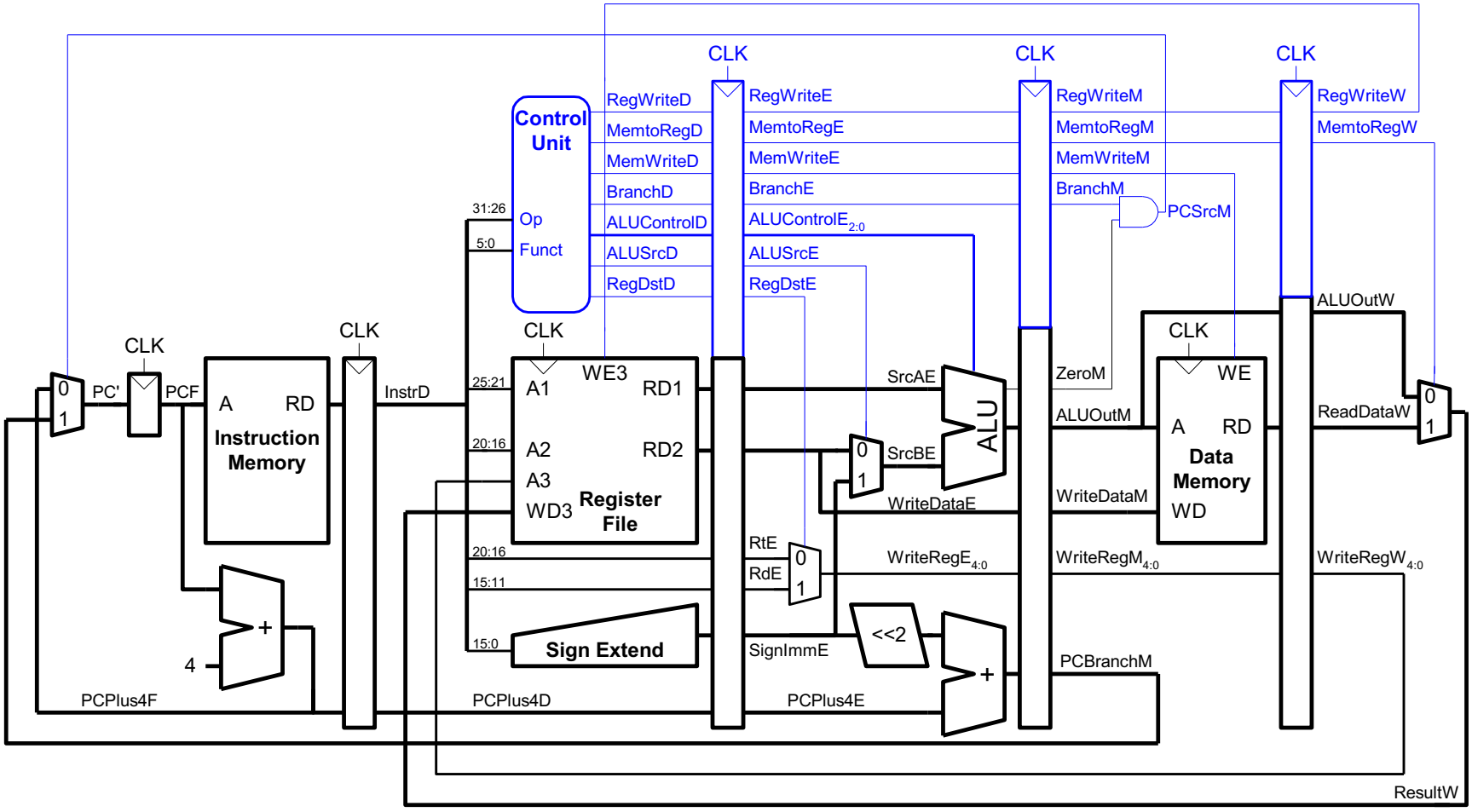


Corrected Pipelined Datapath



- **WriteReg must arrive at the same time as Result**

Pipelined Control



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

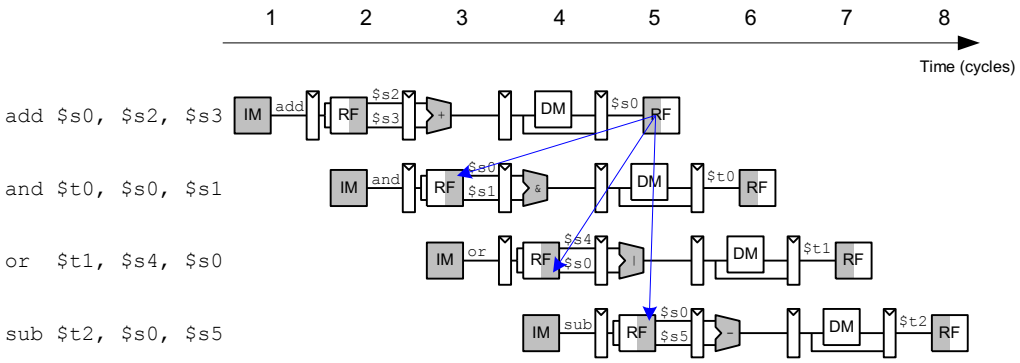
Pipeline Hazard

- **Occurs when an instruction depends on results from previous instruction that hasn't completed.**

- **Types of hazards:**
  - ***Data hazard:*** register value not written back to register file yet
  - ***Control hazard:*** next instruction not decided yet (caused by branches)

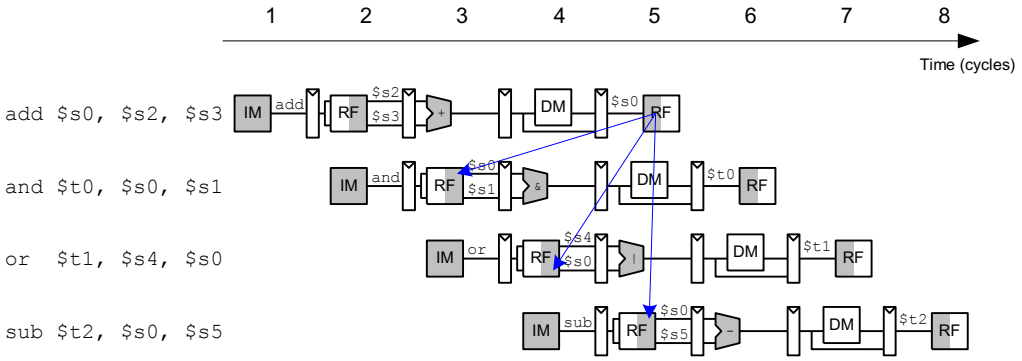
Data Hazard

- **The register file can be read and written in the same cycle:**
  - write takes place during the 1st half of the cycle
  - read takes place during the 2nd half of the cycle => no hazard !!!
  - However operations that involve register file have only ***half a clock cycle*** to complete the operation!!



Data Hazard

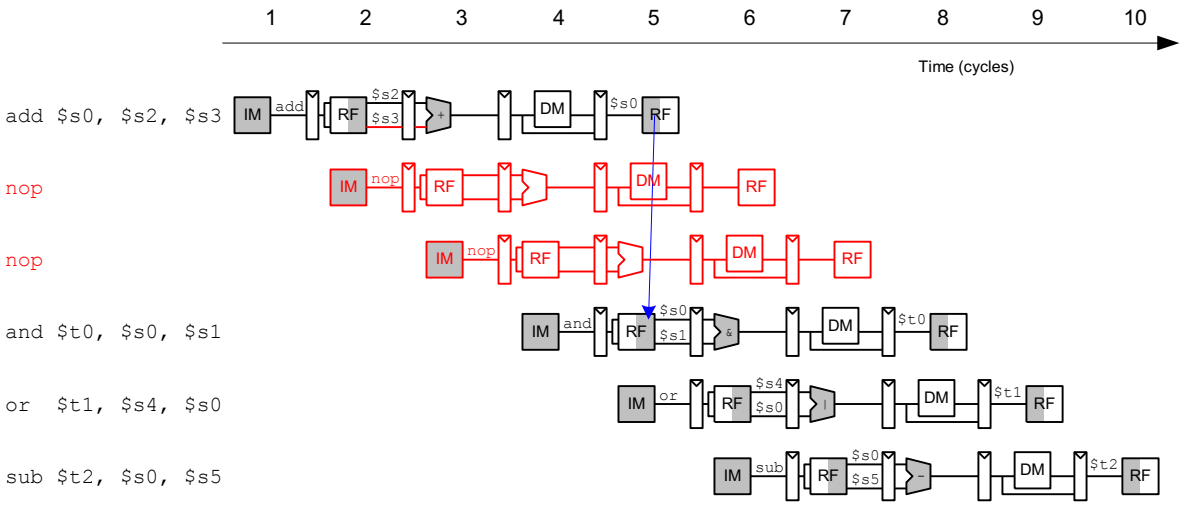
- **One instruction writes a register ($s0) and next instructions read this register => read after write (RAW) hazard.**
  - ***add*** writes into $s0 in the first half of cycle 5
  - ***and*** reads $s0 on cycle 3, obtaining the wrong value
  - ***or*** reads $s0 on cycle 4, again obtaining the wrong value.
  - ***sub*** reads $s0 in the second half of cycle 5, obtaining the correct value
  - subsequent instructions read the correct value of $s0



How Can You Handle Data Hazards?

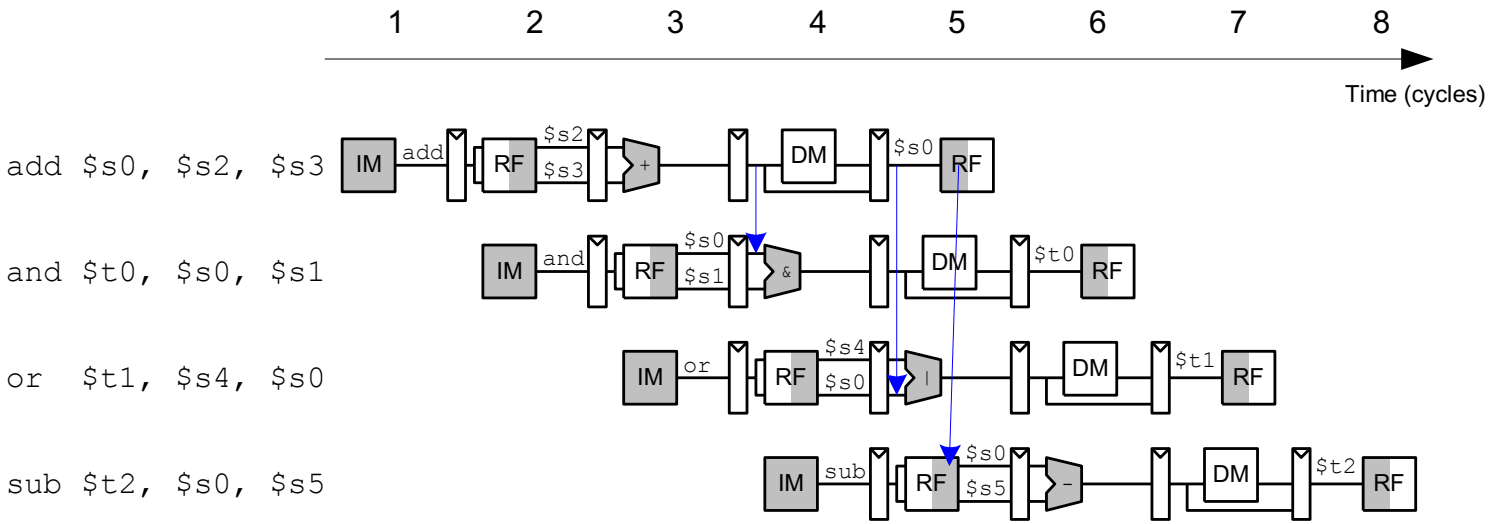
- Insert “NOP”s (No OPeration) in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Compile-Time Hazard Elimination

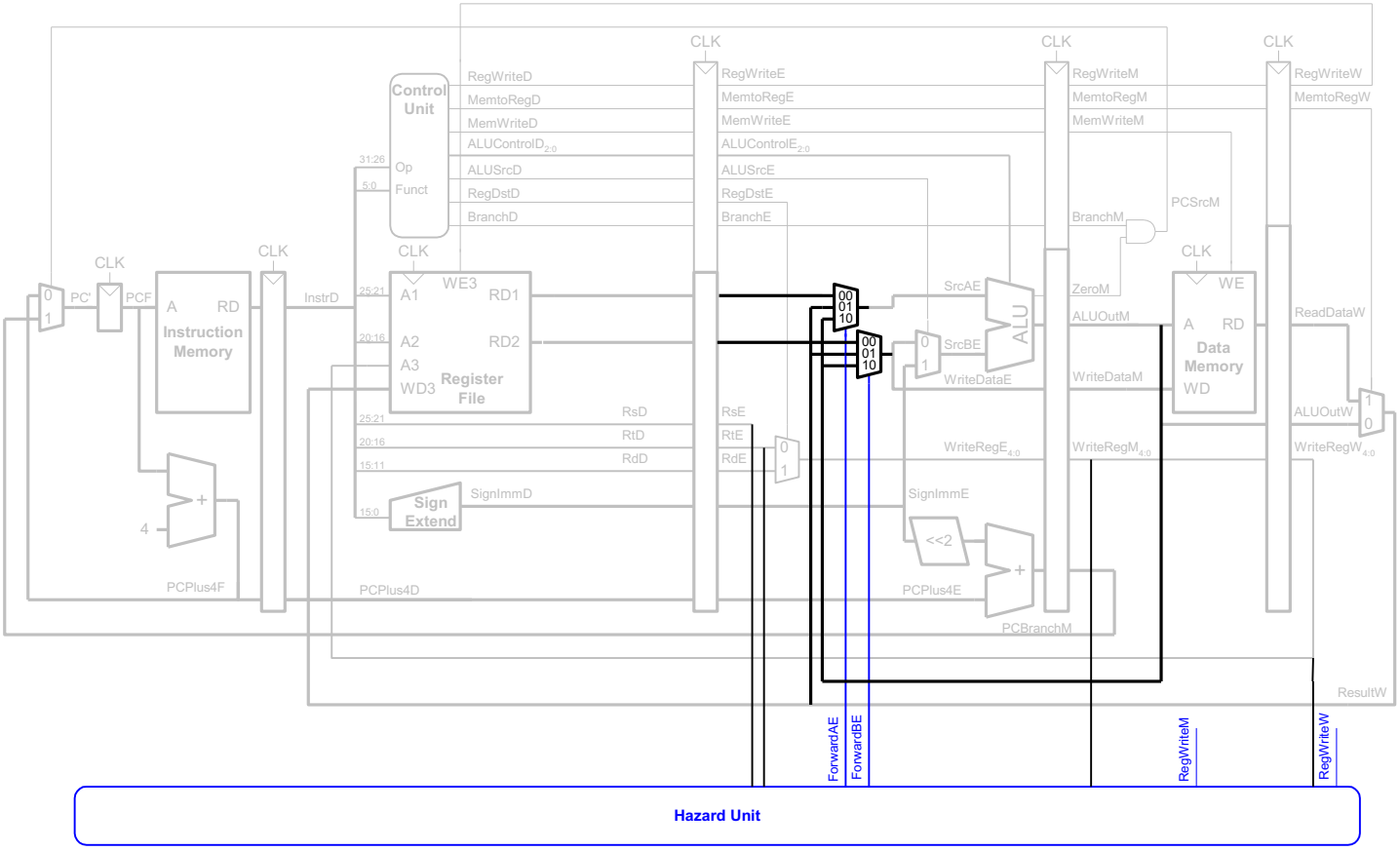


- Insert enough NOPs for result to be ready
- Or (if you can) move independent useful instructions forward

Data Forwarding



Data Forwarding



Data Forwarding

- **Forward to Execute stage from either:**
  - Memory stage or
  - Writeback stage
- **When should we forward from one either Memory or Writeback 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, the Memory stage should have priority, because it contains the more recently executed instruction.

Data Forwarding

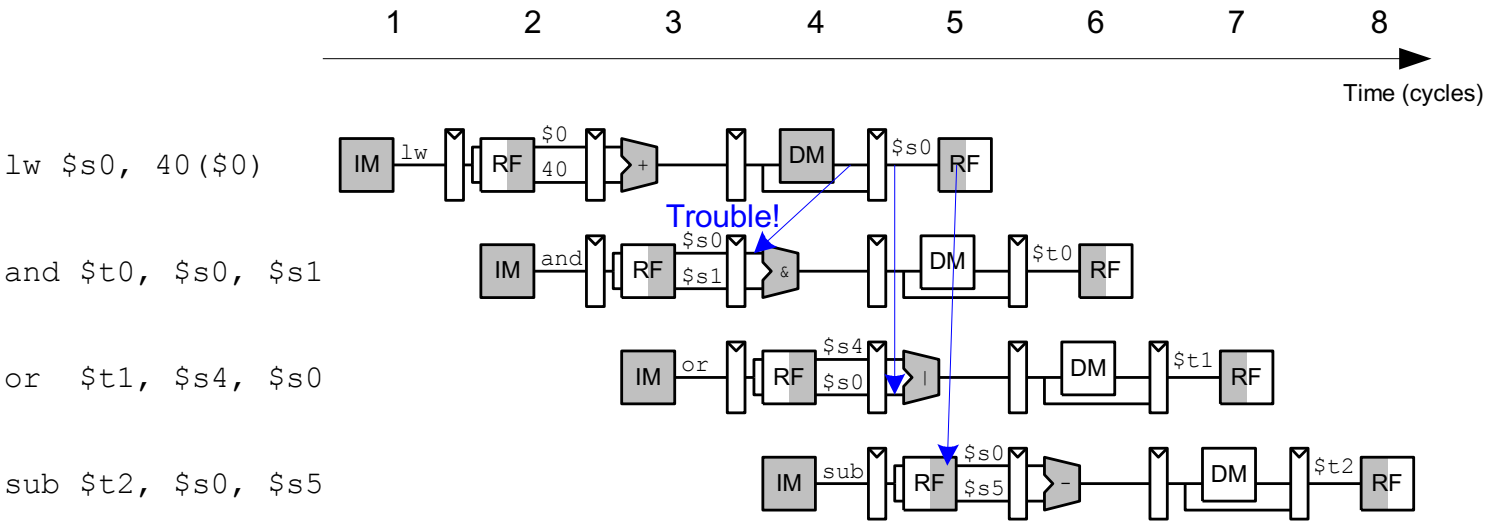
- **Forward to Execute stage from either:**
  - Memory stage or
  - Writeback stage

- **Forwarding logic for *ForwardAE (pseudo code)*:**

```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
        ForwardAE = 10  # forward from Memory stage
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
        ForwardAE = 01  # forward from Writeback stage
else
        ForwardAE = 00  # no forwarding
```

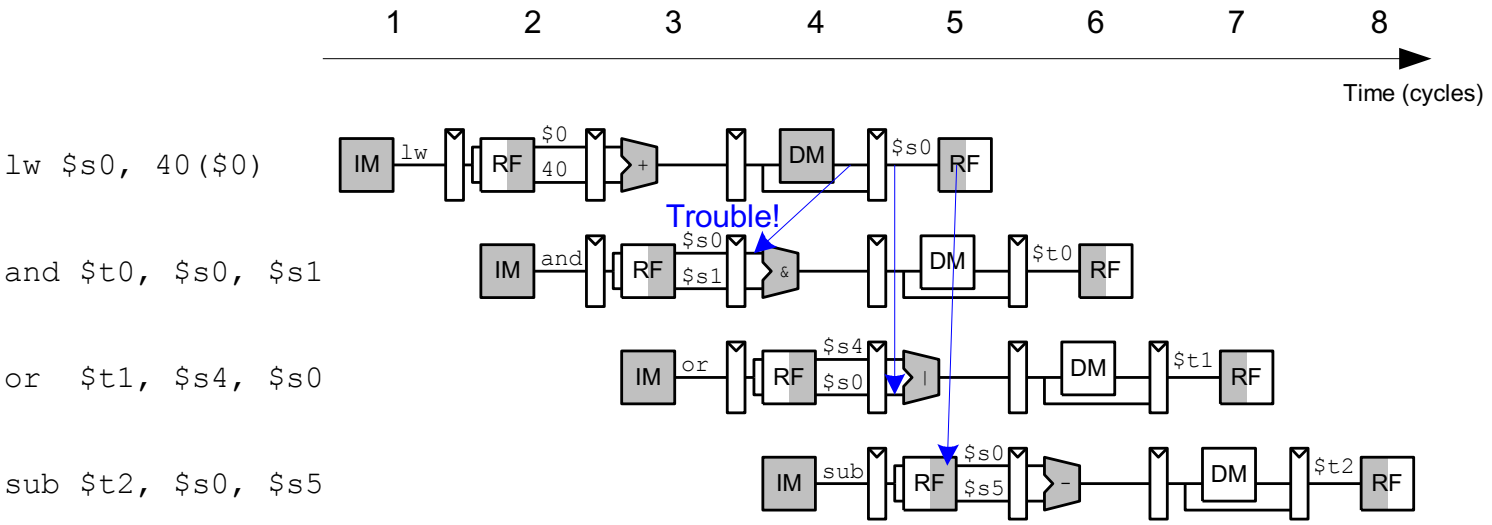
- **Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE***

Stalling



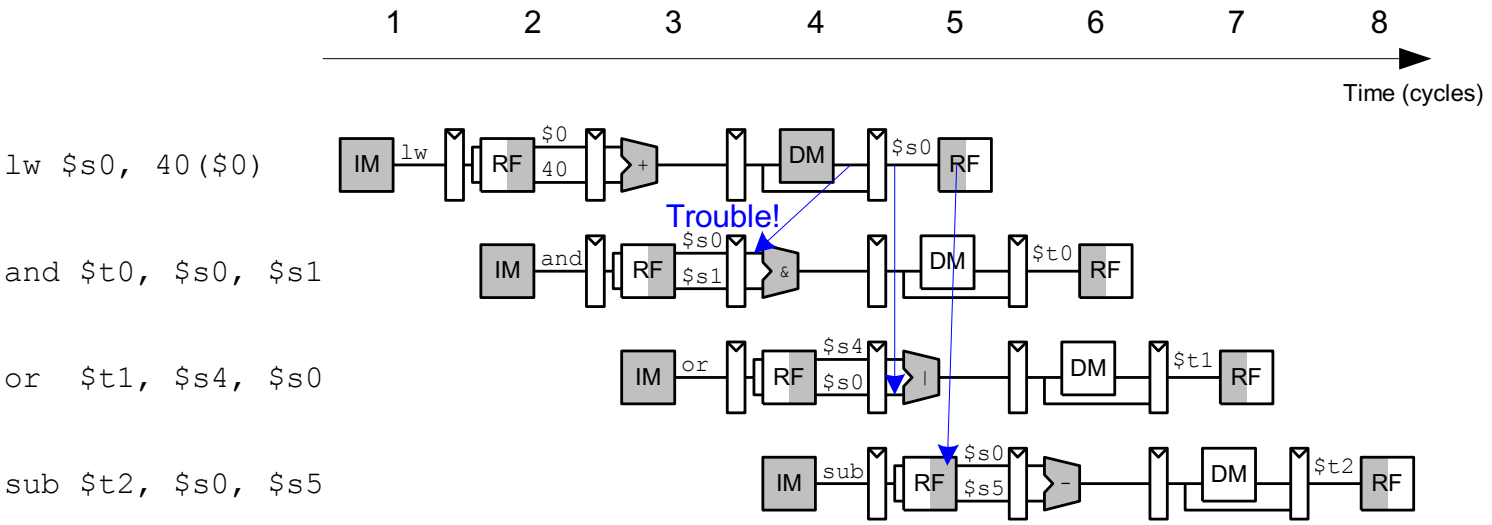
- **Forwarding is sufficient to solve RAW data hazards**
- ***but …***

Stalling



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.

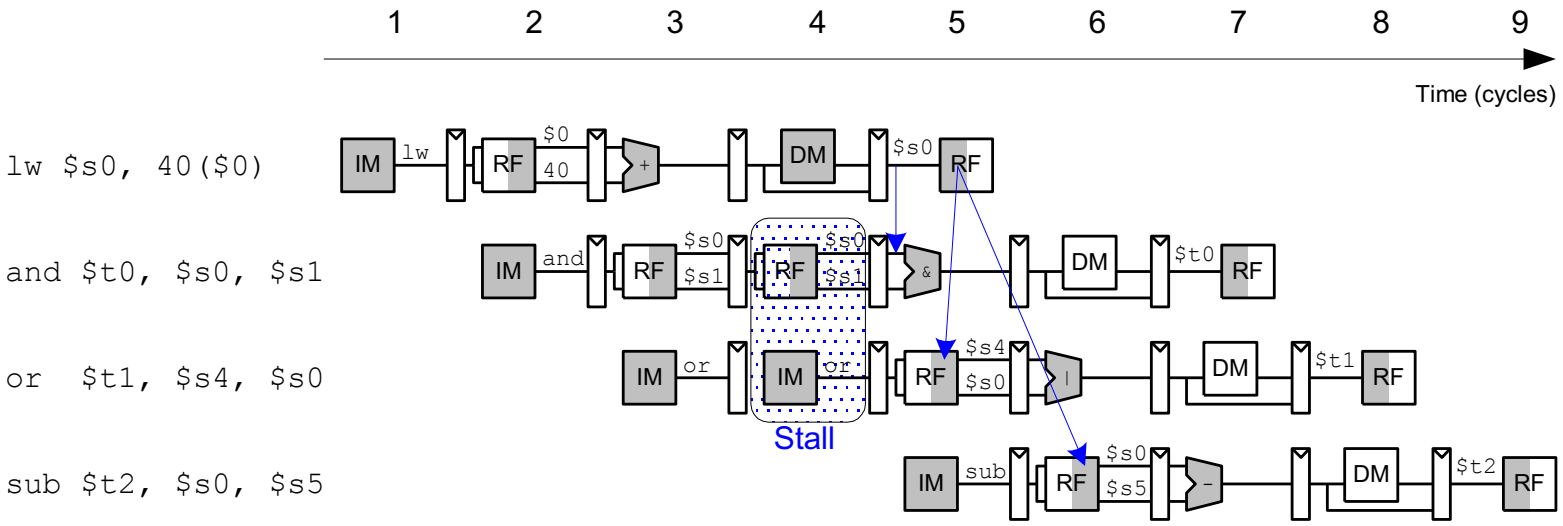
Stalling



The **lw** instruction has ***a two-cycle latency***, therefore a dependent instruction cannot use its result until two cycles later.

The **lw** instruction receives data from memory at the end of cycle 4. But the **and** instruction needs that data as a source operand at the beginning of cycle 4. ***There is no way to solve this hazard with forwarding.***

Stalling

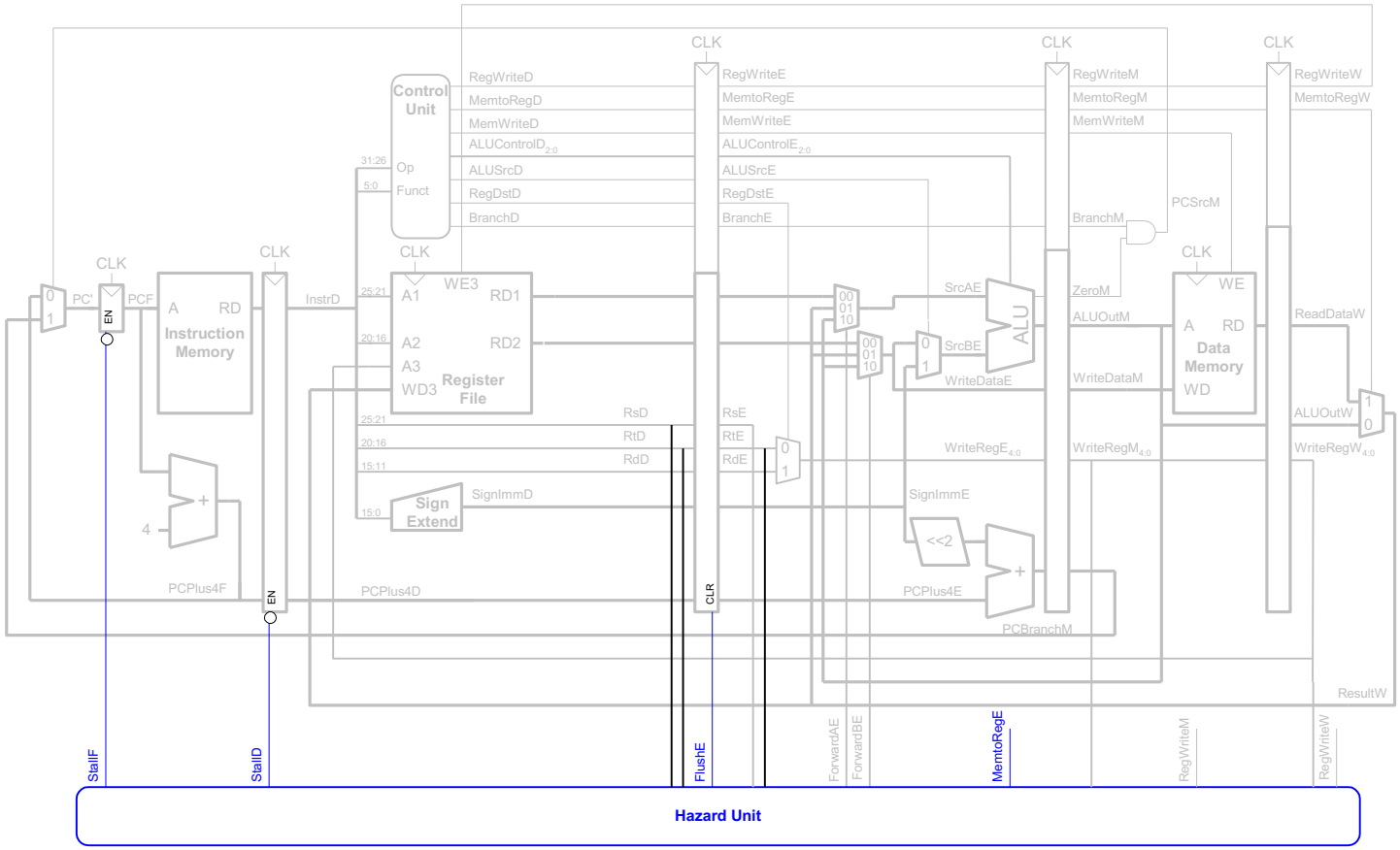

1
2
3
4
5
6
7
8
9
Time (cycles)
lw $s0, 40($0)
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
IM
lw
RF
$0
40
+
DM
$s0
RF
IM
and
RF
$s0
$s1
RF
$s0
$s1
&
DM
$t0
RF
IM
or
IM
or
RF
$s4
$s0
|
DM
$t1
RF
Stall
IM
sub
RF
$s0
$s5
-
DM
$t2
RF

Stalling Hardware

- **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.

- **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, introducing a bubble

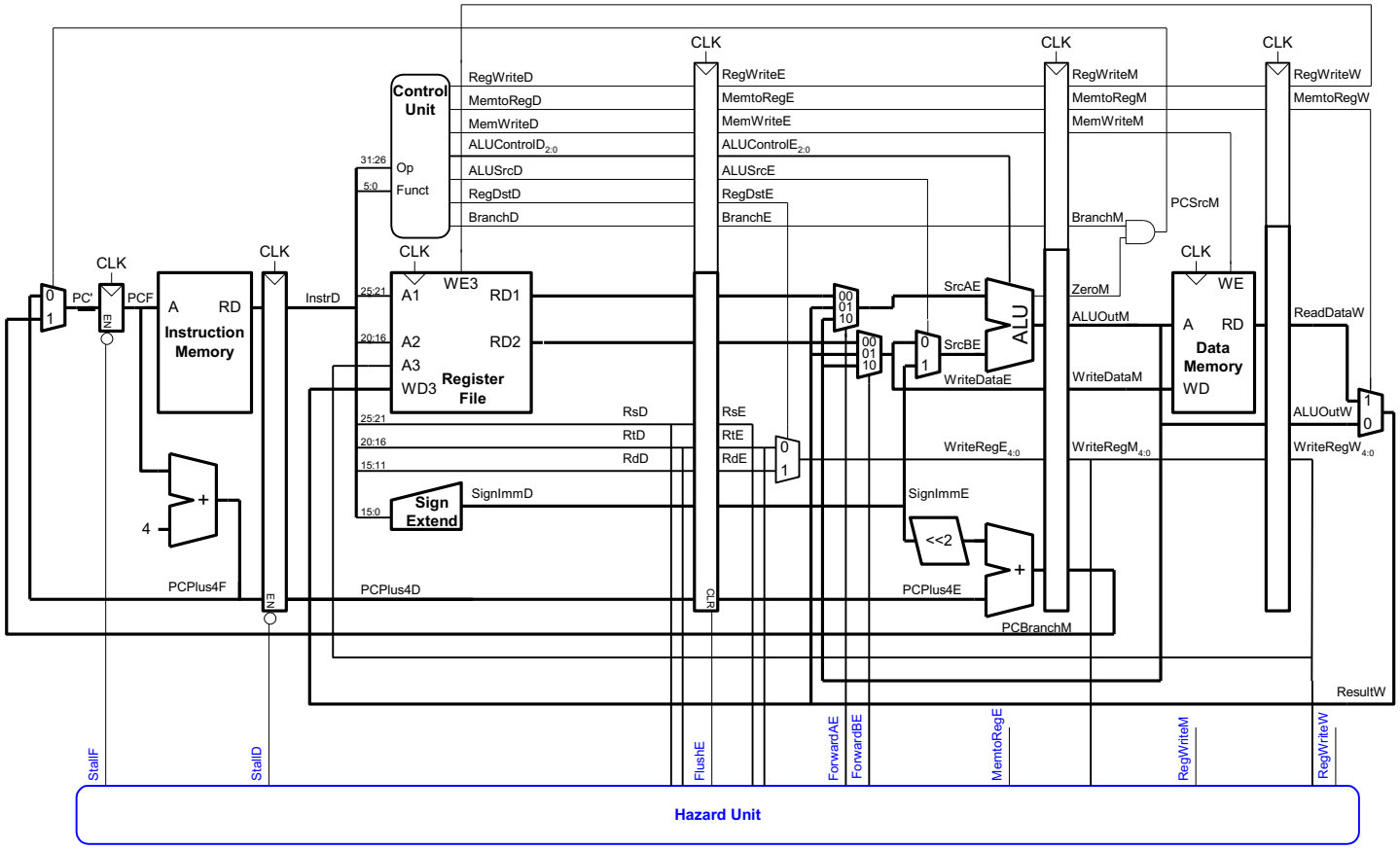
Stalling Hardware



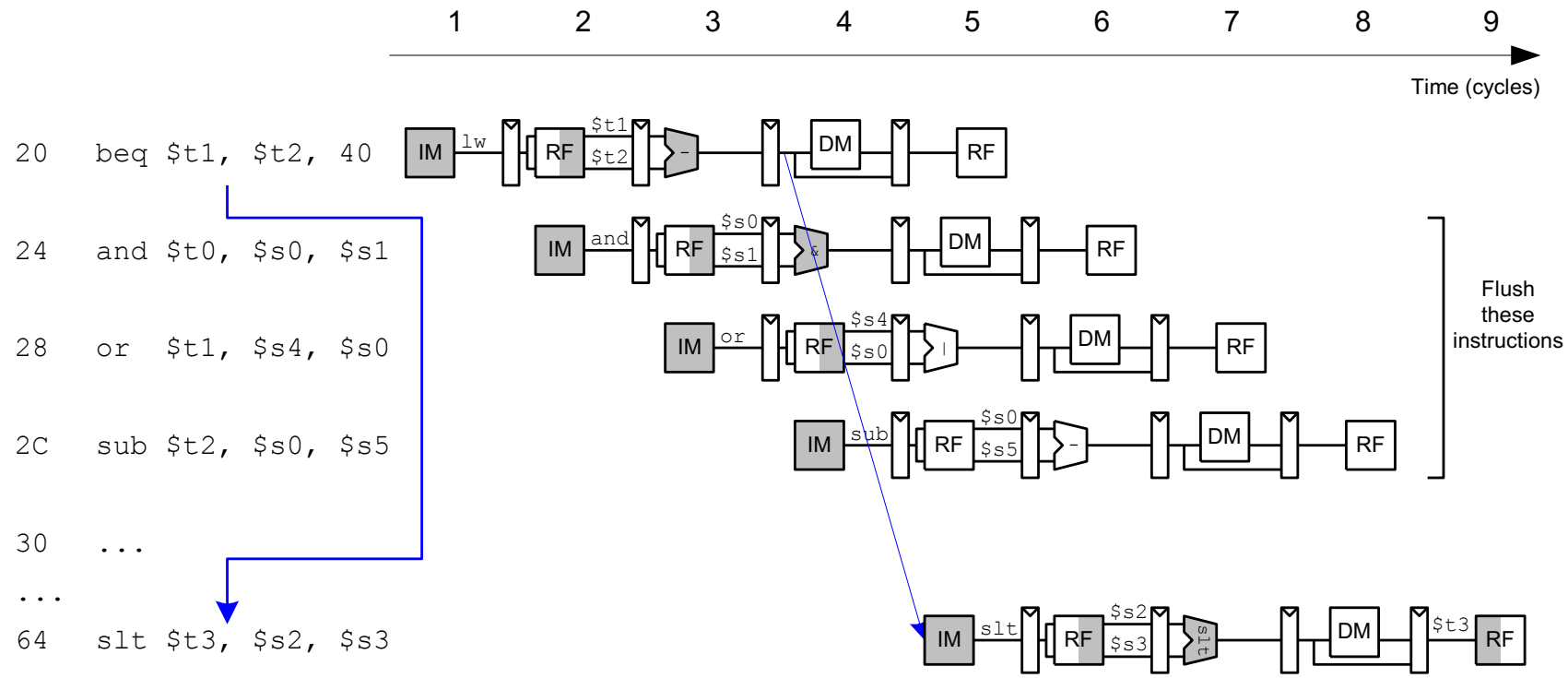
Control Hazards

- **beq:**
  - branch is not determined until the fourth stage of the pipeline
  - Instructions after the branch are fetched before branch occurs
  - These instructions must be flushed if the branch happens
- **Branch misprediction penalty**
  - number of instruction flushed when branch is taken
  - May be reduced by determining branch earlier

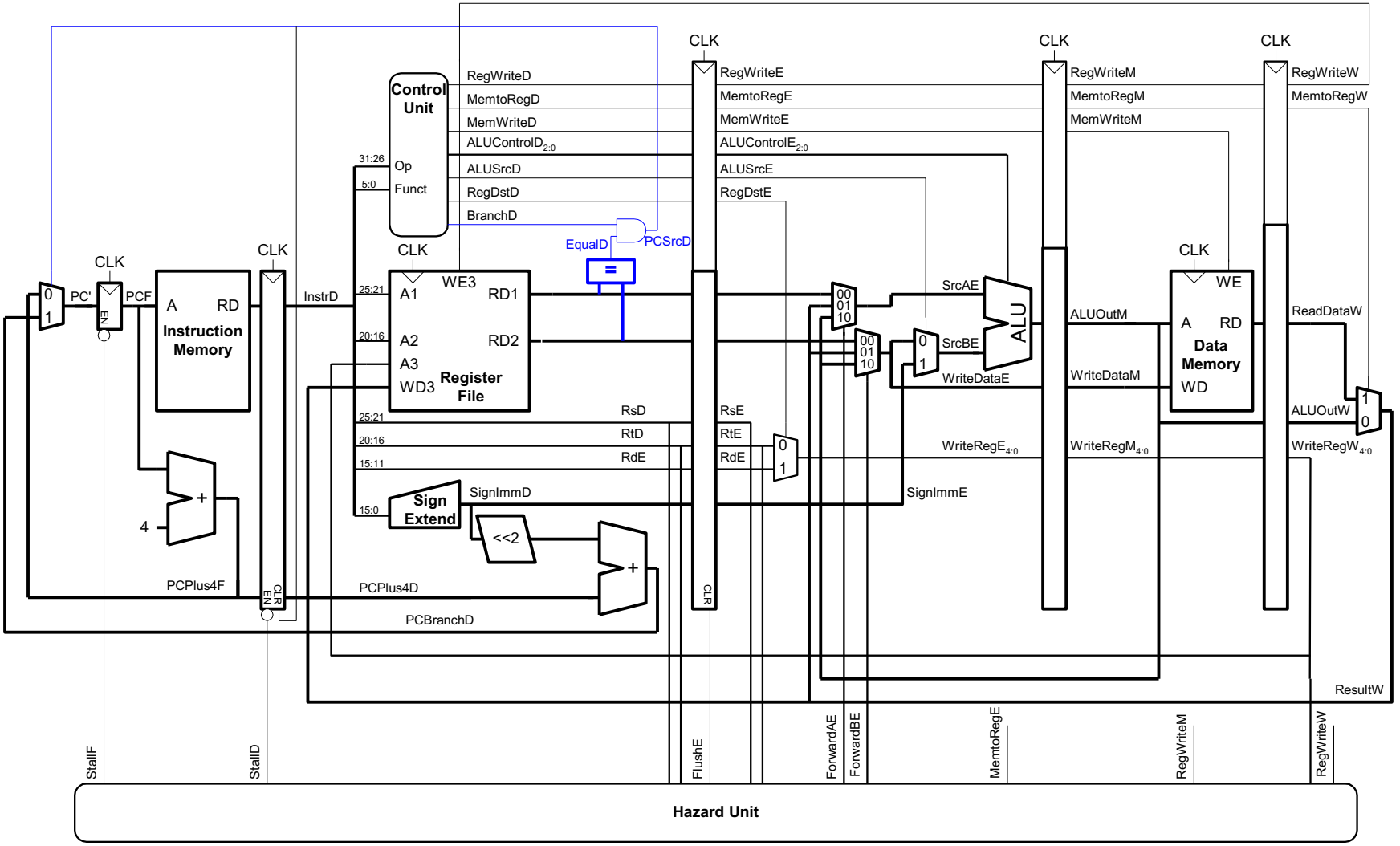
Control Hazards: Original Pipeline



Control Hazards

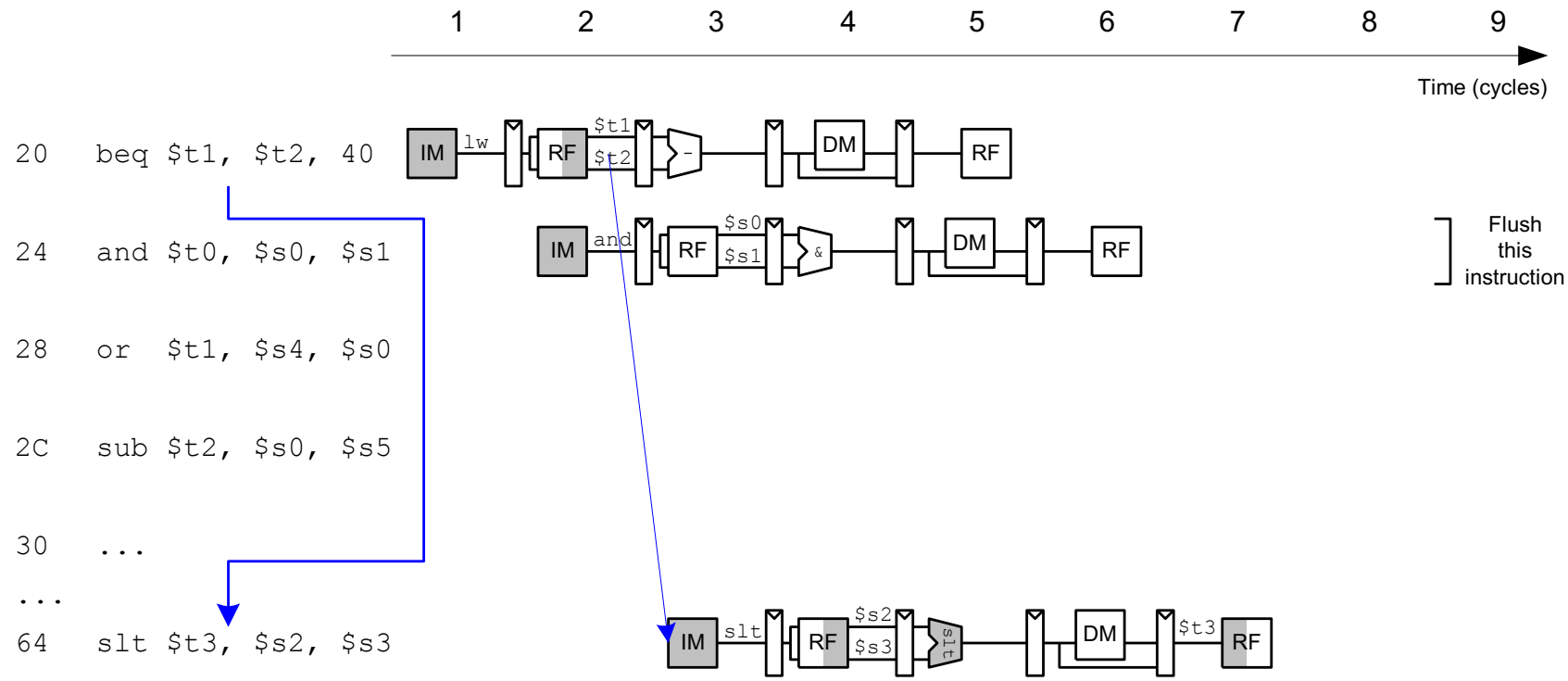


Control Hazards: Early Branch Resolution

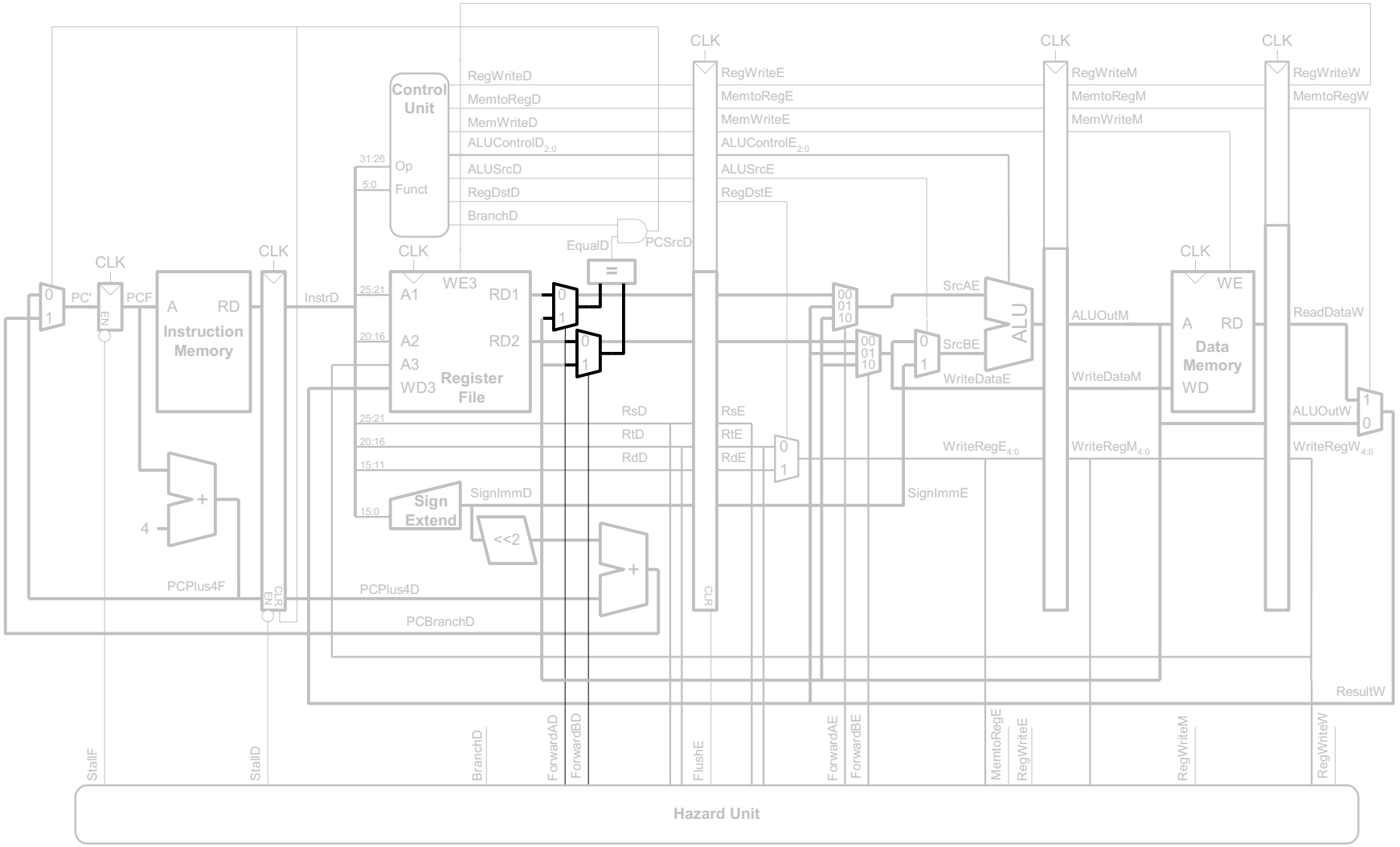


Introduced another data hazard in Decode stage.. this is another story

Early Branch Resolution



Handling Data and Control Hazards



Possible solution to data hazard in Decode stage.

Control Forwarding and Stalling Hardware

```
// Forwarding logic:
assign ForwardAD = (rsD != 0) & (rsD == WriteRegM) & RegWriteM;
assign ForwardBD = (rtD != 0) & (rtD == WriteRegM) & RegWriteM;

//Stalling logic:
assign lwstall = ((rsD == rtE) | (rtD == rtE)) & MemtoRegE;

assign branchstall = (BranchD & RegWriteE &
                      (WriteRegE == rsD | WriteRegE == rtD))
                      |
                      (BranchD & MemtoRegM &
                      (WriteRegM == rsD | WriteRegM == rtD));

// Stall signals;
assign StallF = lwstall | branchstall;
assign StallD = lwstall | branchstall;
assign FLushE = lwstall | branchstall;
```

Branch Prediction

- **Guess whether branch will be taken**
  - Backward branches are usually taken (loops)
  - Perhaps consider history of whether branch was previously taken to improve the guess
- **Good prediction reduces the fraction of branches requiring a flush**

Pipelined Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
- **All jumps flush next instruction**
- **What is the average CPI?**

Pipelined Performance Example Solution

- **Load/Branch CPI = 1 when no stalling, 2 when stalling. Thus:**
  - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$ *Average CPI for load*
  - $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$ *Average CPI for branch*
- **And**
  - ***Average CPI*** =

Pipelined Performance Example Solution

- **Load/Branch CPI = 1 when no stalling, 2 when stalling. Thus:**
  - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$ — *Average CPI for load*
  - $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$ — *Average CPI for branch*
- **And**
  - ***Average CPI*** = (0.25)(1.4) + *load*
    (0.1)(1) + *store*
    (0.11)(1.25) + *beq*
    (0.02)(2) + *jump*
    (0.52)(1) *r-type*

    = **1.15**

Pipelined Performance

- There are 5 stages, and 5 different timing paths:

$T_c = \max \{$

$t_{pcq} + t_{mem} + t_{setup}$	*fetch*
$2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$	*decode*
$t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}$	*execute*
$t_{pcq} + t_{memwrite} + t_{setup}$	*memory*
$2(t_{pcq} + t_{mux} + t_{RFwrite})$	*writeback*

$\}$

- The operation speed ***depends*** on the ***slowest operation***
- Decode and Writeback use register file and have only half a clock cycle to complete, that is why there is a 2 in front of them

Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	$T_{memwrite}$	220
Register file write	$t_{RFwrite}$	100

$$T_c = 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$$
$$= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps}$$
$$= 550 \text{ ps}$$

Pipelined Performance Example

- **For a program with 100 billion instructions executing on a pipelined MIPS processor:**
  - CPI = 1.15
  - T_c = 550 ps

- **Execution Time = (# instructions) × CPI × T_c**
  = $(100 \times 10^9)(1.15)(550 \times 10^{-12})$
  = 63 seconds

Performance Summary for MIPS arch.

Processor	Execution Time (seconds)	Speedup (single-cycle is baseline)
Single-cycle	95	1
Multicycle	133	0.71
Pipelined	63	1.51

- **Fastest of the three MIPS architectures is *Pipelined.***
- **However, even though we have 5 fold pipelining, it is not 5 times faster than single cycle.**

What Did We Learn?

- **How to design a pipelined architecture**
  - Break down long combinational path by registers
  - Shortens the clock period
  - You can start processing next instruction once the first part is complete
- **Problems of the pipelined architecture**
  - Data you need for the next instruction may not be ready (***Data Hazard***)
  - You may not yet know the next instruction (**Control Hazard**)
- **Solutions to Hazards**
- **Performance of pipelined MIPS architecture**