CS-200
Computer Architecture

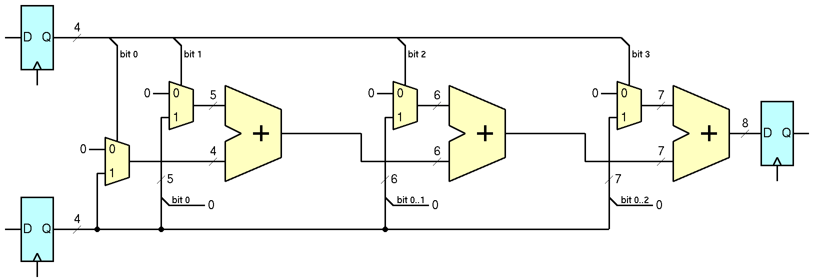
—

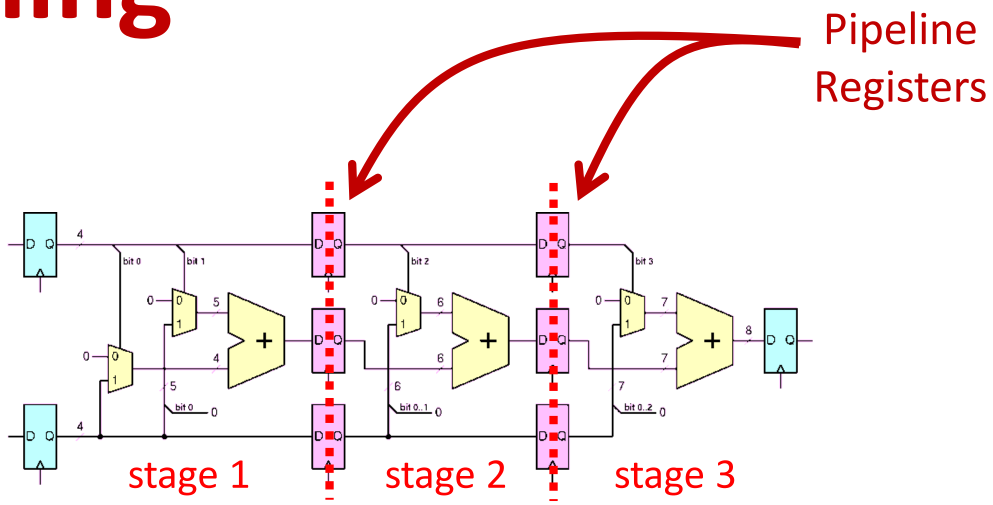
Part 4c. Instruction Level Parallelism
Pipelining

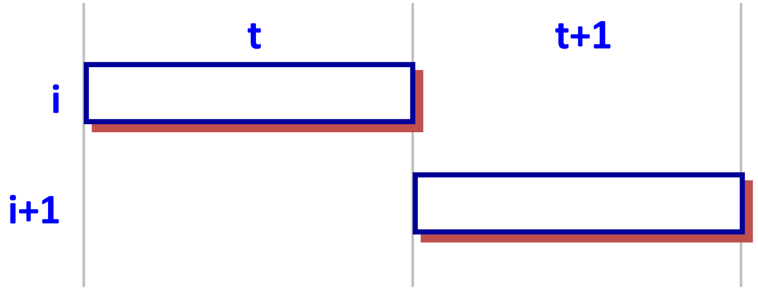
Paolo Ienne

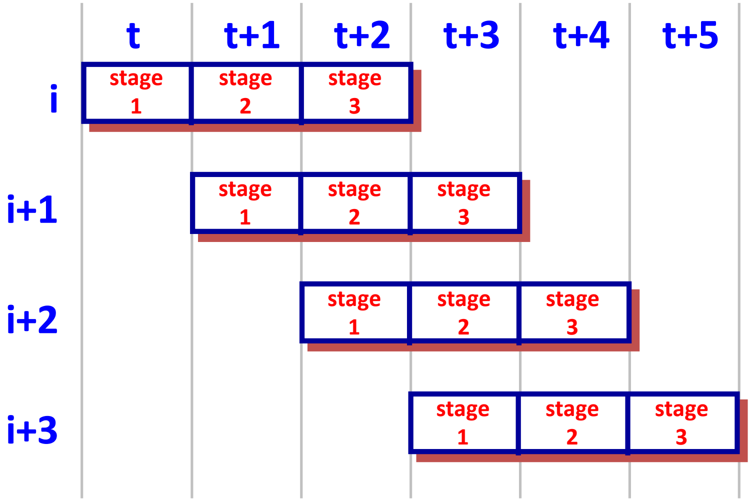
<paolo.ienne@epfl.ch>

Pipelining

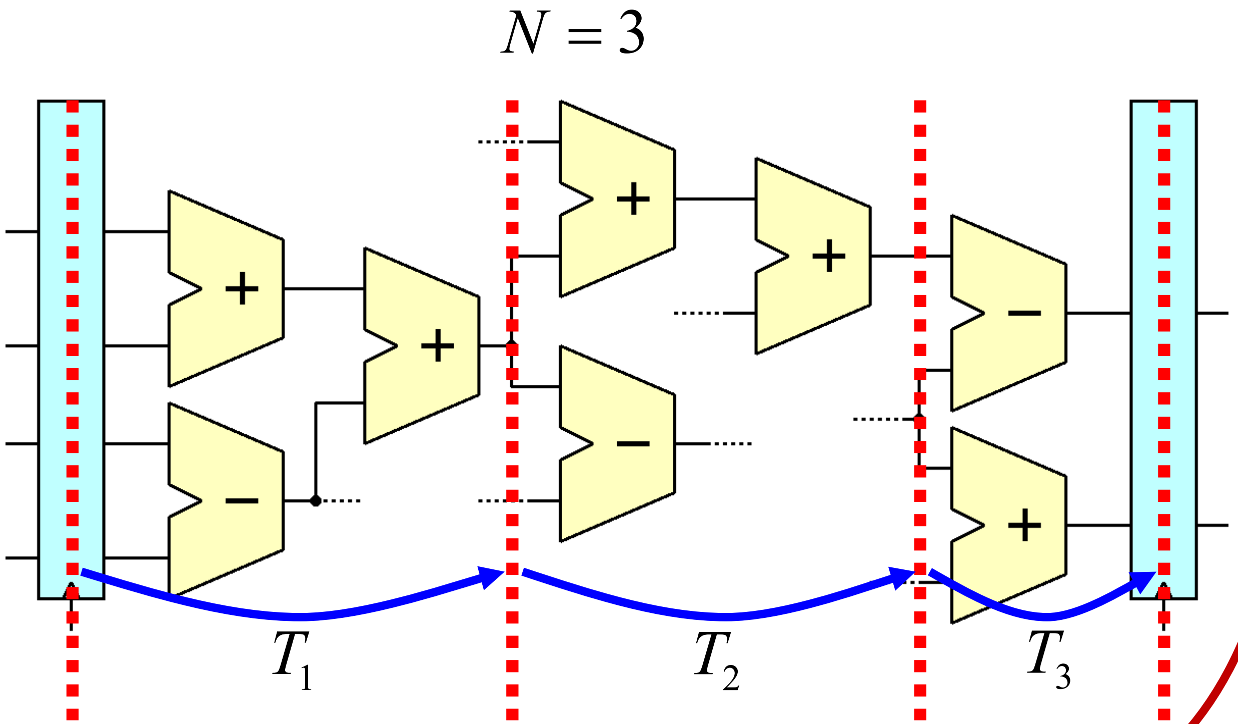



Pipeline Registers
stage 1
stage 2
stage 3


t
t+1
i
i+1


t
t+1
t+2
t+3
t+4
t+5
i
i+1
i+2
i+3
stage 1
stage 2
stage 3

Practical Pipelining



But flip-flops introduce a delay of their own…

But one might not be able to split **evenly** the circuit…

Ideally,

$$T_i \cong T_{CLK,comb} / N$$

and

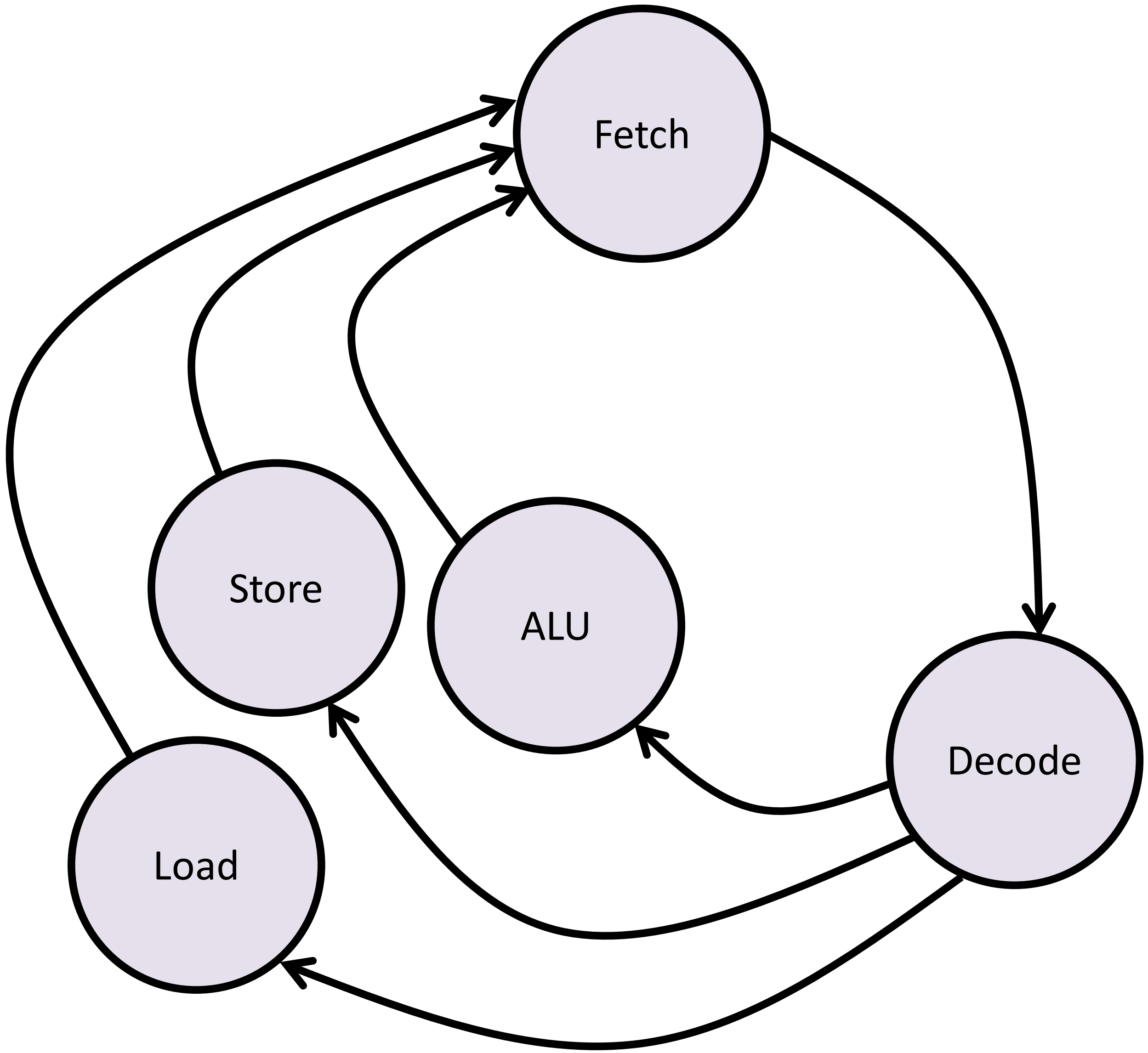
$$T_i \cong T_{CLK,pipe}$$

- **Latency** $\lambda_{pipe} = N \cdot \max_{i=0..N-1} (T_i + T_{FF}) = N \cdot T_{CLK,pipe} = N / f_{pipe} > \lambda_{orig}$
- **Throughput** $\phi_{pipe} = 1 / \max_{i=0..N-1} (T_i + T_{FF}) = f_{pipe}$

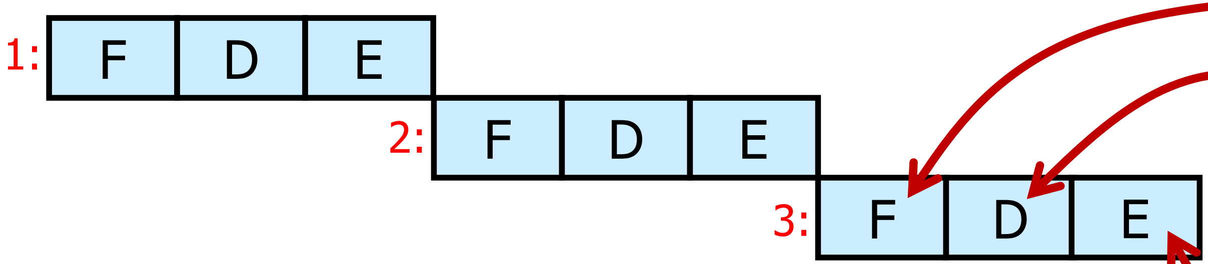
Pipelining for Processors?

- Pipelining useful only if the activity in object needs to be repeated many time
  - We have plenty of **instructions** to execute!
- Pipelining needs to split the single activity in object into many subactivities
  - We have a **good logical split** into *fetch*, *decode*, *execute*, etc.

Example: A Simple Multi-Cycle Processor



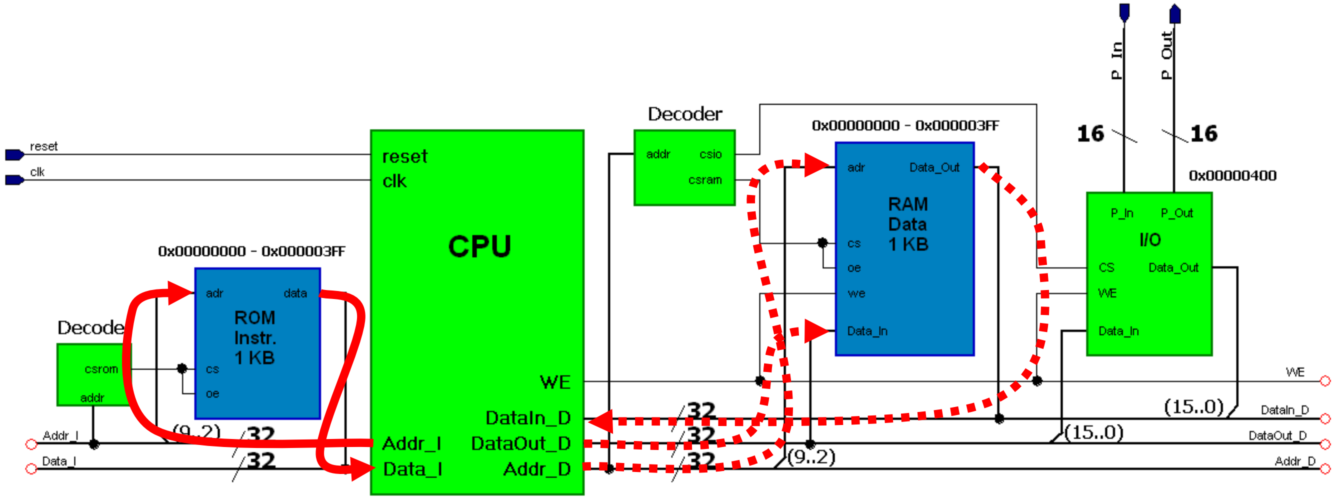
Example: A Simple Schedule



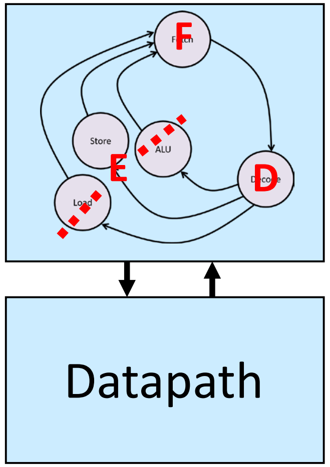
FETCH
Get the instruction to execute from the instruction memory

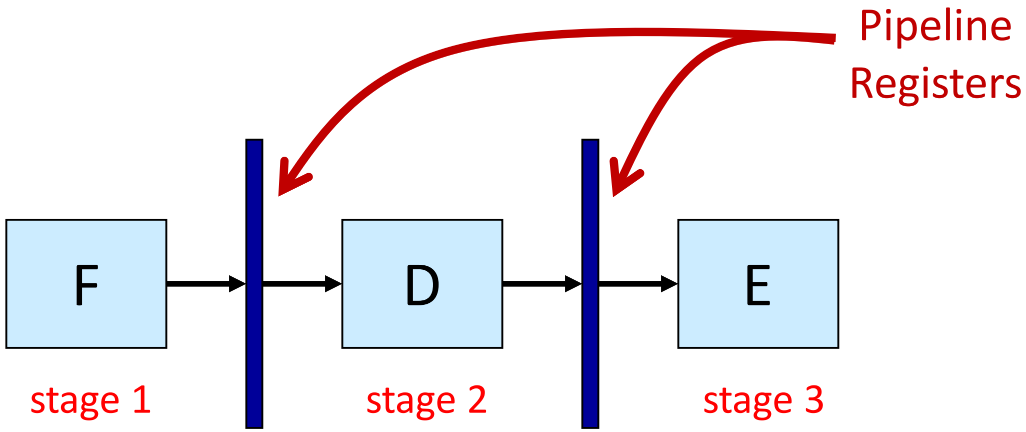
DECODE
Understand/decode the instruction and get ready for execution

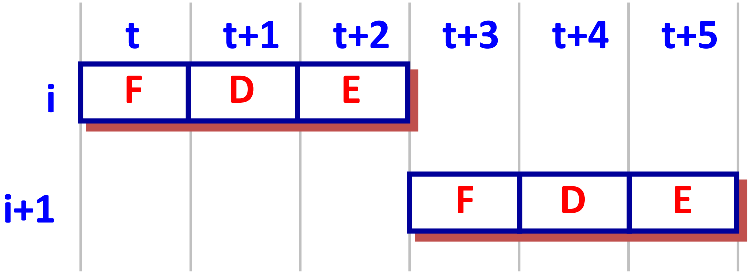
EXECUTE
Perform the required operation (including accessing data memory in case of loads and stores)

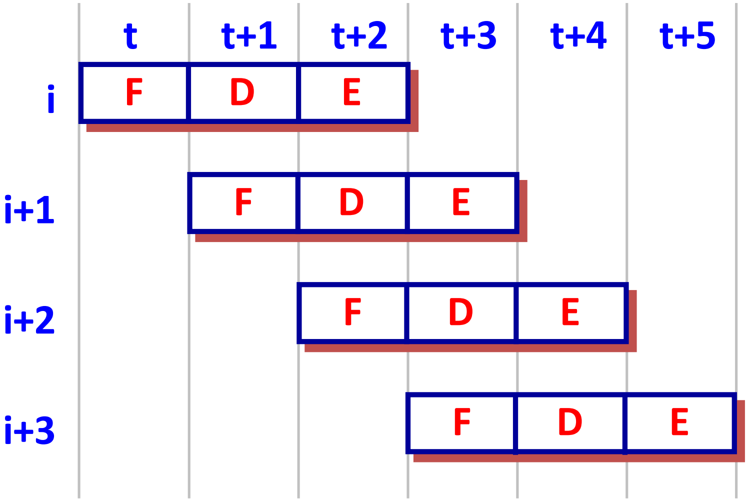


Pipelining the Processor?









No Hardware Reuse across Stages

- In a multicycle processor, some hardware components may be shared across **states**:
  - **FETCH** typically requires an **adder** to increment the program counter
  - **EXECUTE** naturally needs an **ALU**
  - They are **never used at the same time**, so the ALU can be used to increment the program counter
- In a pipelined processor, there cannot be sharing across **stages**, in general:
  - All stages **active all the time**!
  - Hardware needs to be **replicated** where appropriate

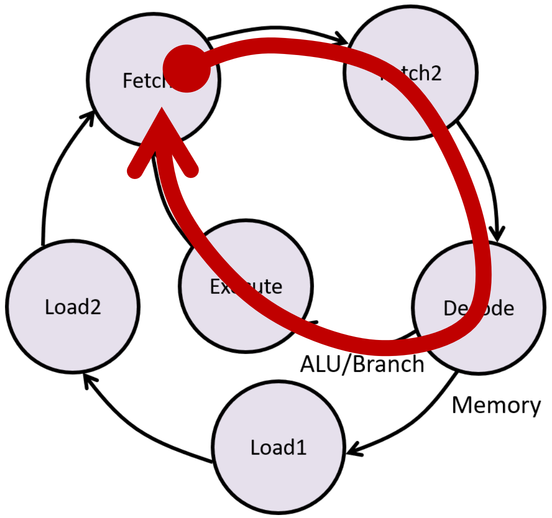
Two Main Problems

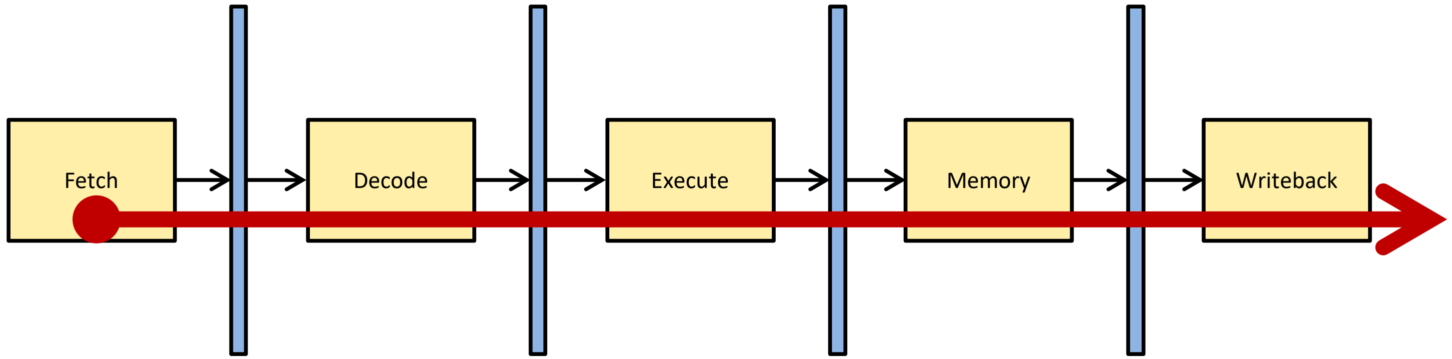
1. CISC vs. **RISC**
   - Can we **build equally well a pipeline** for a Complex Instruction Set Computer as for a Reduced Instruction Set Computer?
   - What does that mean?!

2. Instructions are **not** independent
   - Can we execute code **correctly**?

FSM vs. Pipeline

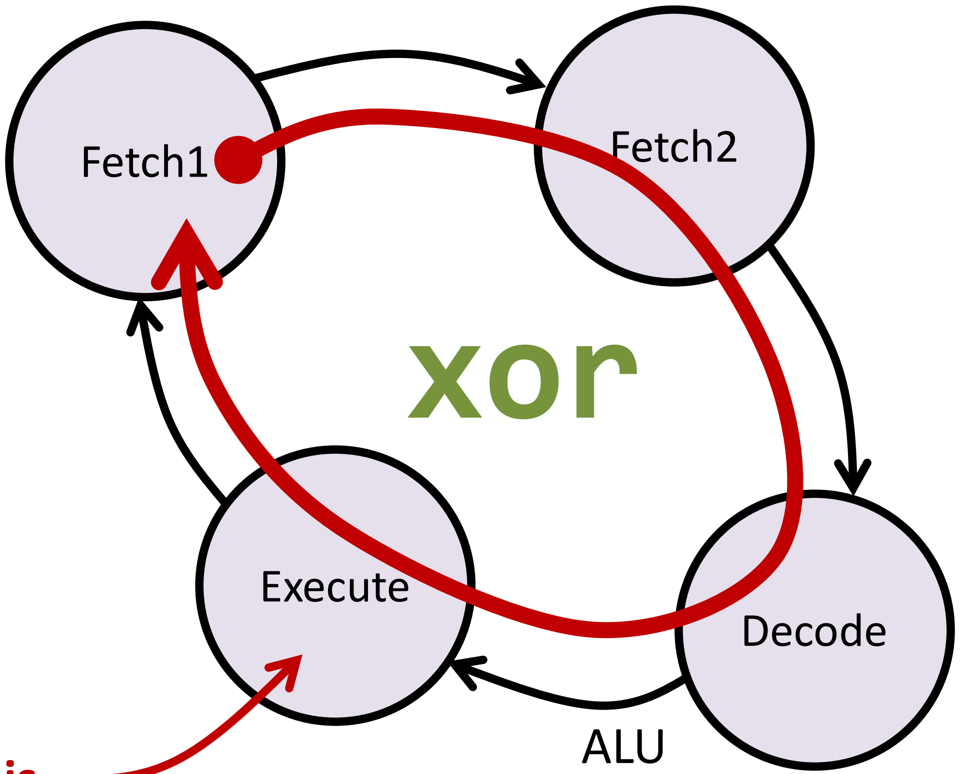
- **Any path** through our FSM represents the **sequence of necessary steps** for the execution of **an** instruction
- **The ordered path** through the pipeline is the **sequence of all possible steps** for the execution of **any** instruction





Adding Instructions to a Multi-Cycle Processor

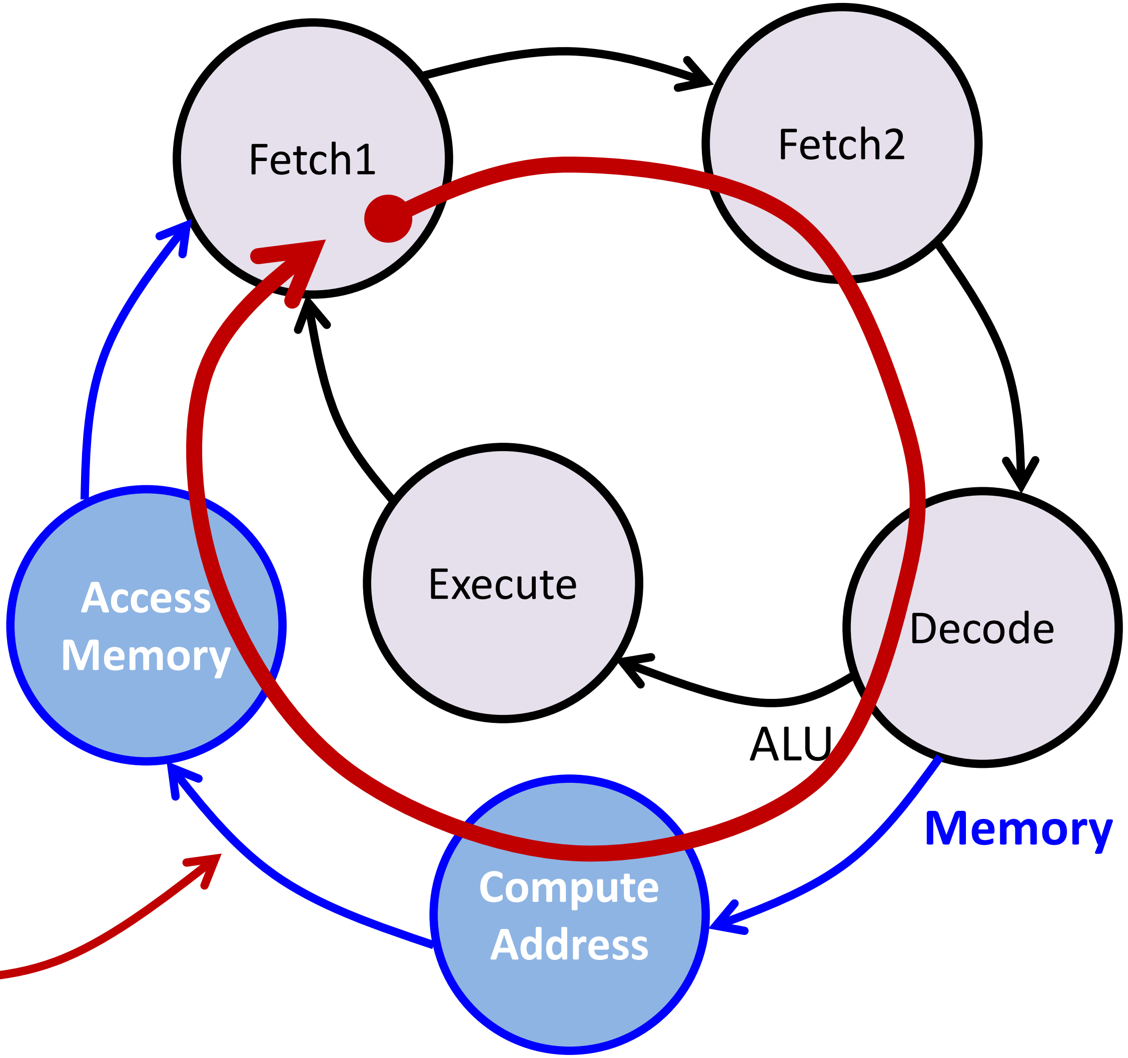
How do we support **add**?



If the **sequence of steps to execute is the same** (fetch the instruction, read registers, use the ALU, save result in the register), we just need to **an ALU that can perform additions**

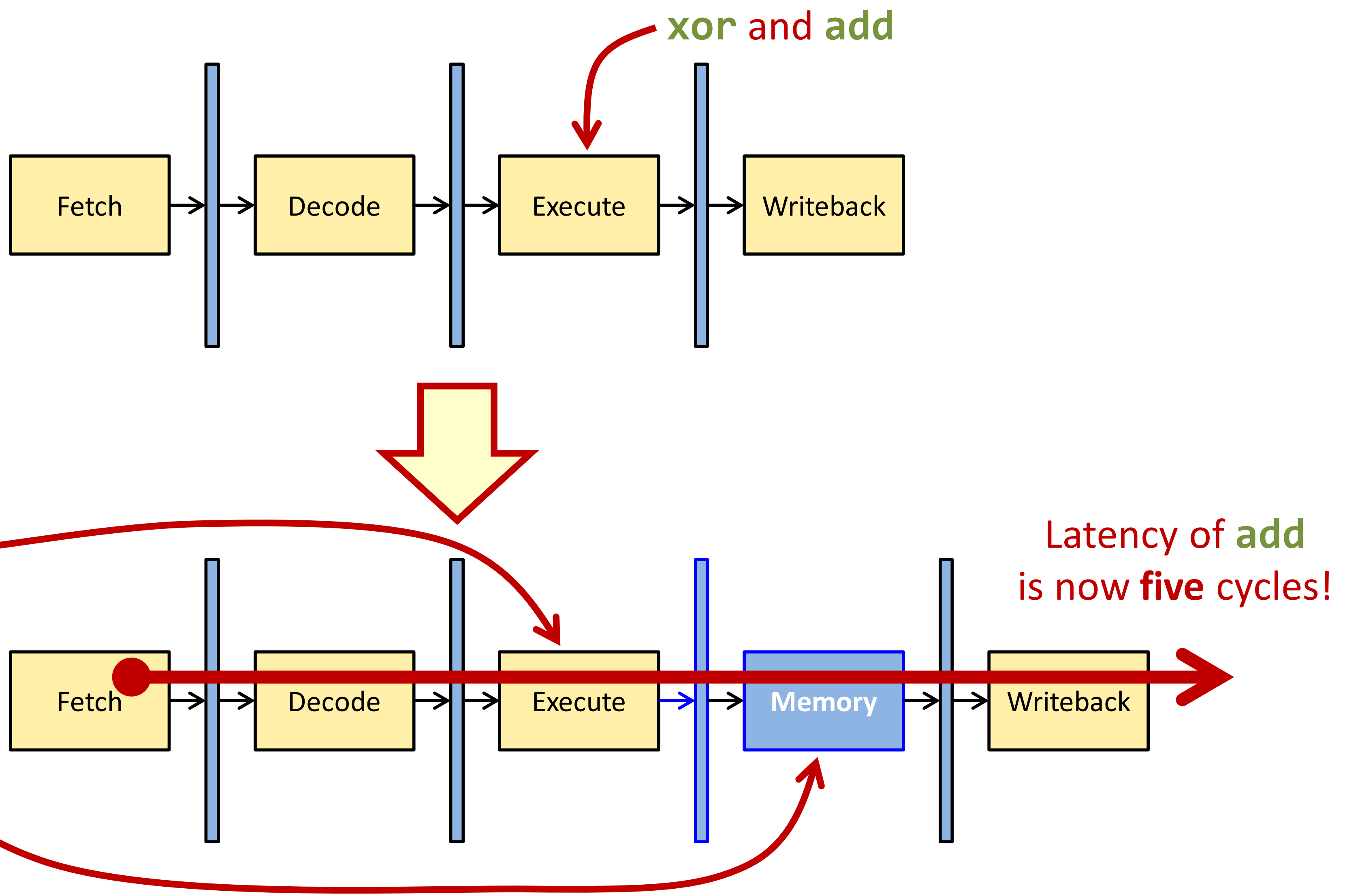
Adding Instructions to a Multi-Cycle Processor

How do we support `lw`?



The sequence of steps to execute is **not** the same (**compute address and access memory**) so we add new steps and a (partially) new path

Adding Instructions to a Pipelined Processor



How do we support lw?

We need to accommodate the steps **compute address** and **access memory**

How?!

The Importance of the ISA

Imagine that we want to have an instruction

```
sub 8(t4), 0(t1), 0(t2)
```

A CISC instruction!

Complex
Instruction-Set
Computer

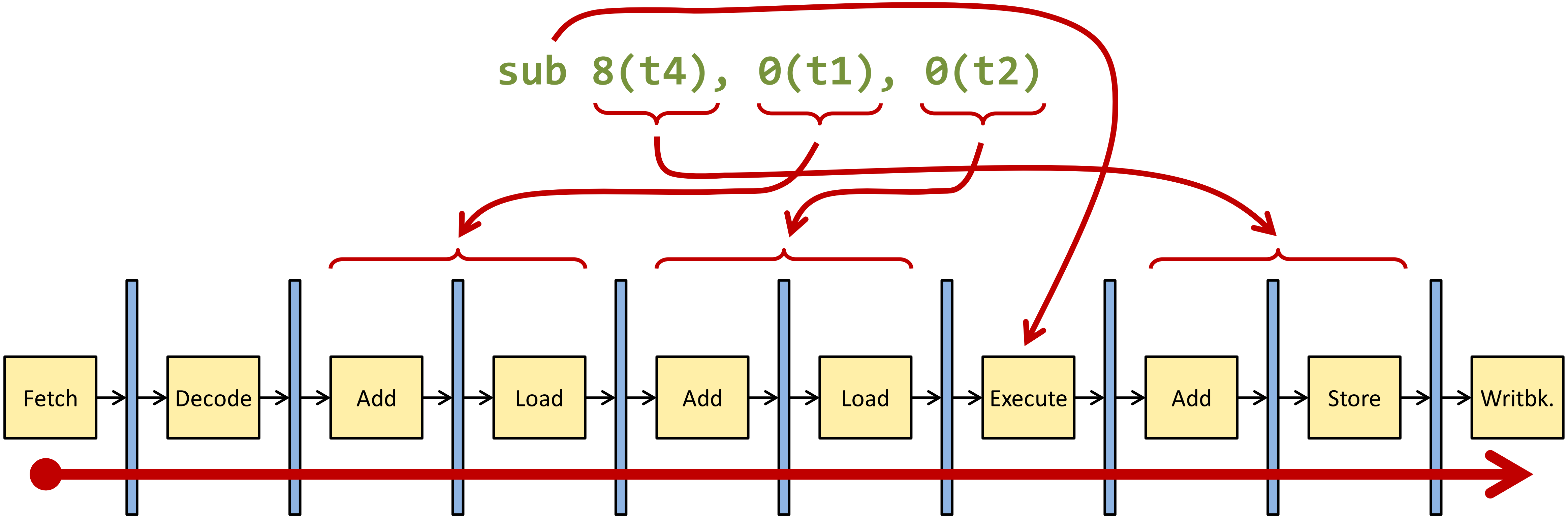
Read the value in memory
at the address `t2 + 0`

Subtract the value in memory at the address `t2 + 0`
from the value in memory at the address `t1 + 0`
and store the result in memory at the address `t4 + 8`

The Importance of the ISA

Imagine that we want to have an instruction

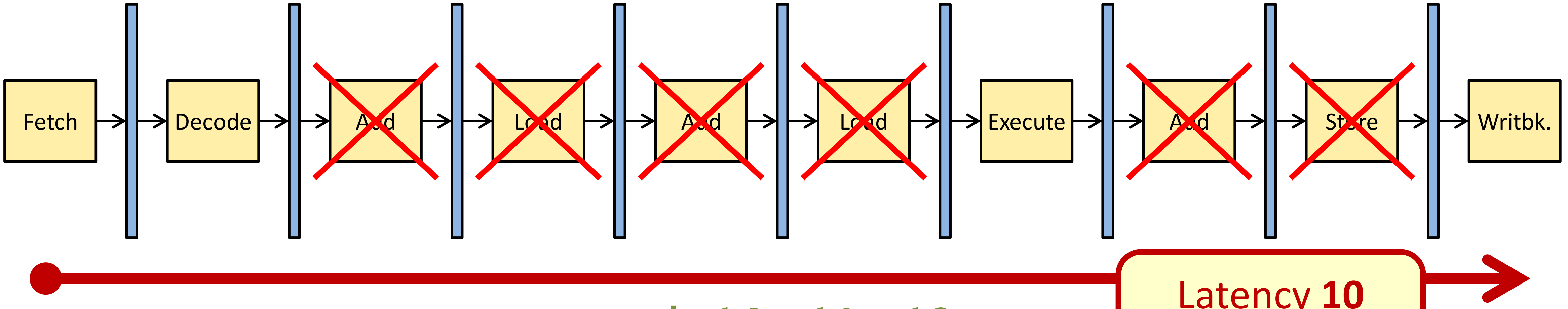
```
sub 8(t4), 0(t1), 0(t2)
```



The Importance of the ISA

Imagine that we want to have an instruction

```
sub 8(t4), 0(t1), 0(t2)
```




```
sub t4, t1, t2
```

Latency **10** instead of **4**!

Reduced Instruction-Set Computer

- Instead of imposing a **huge penalty to every simple instruction** by making complex instructions possible, let's **have only similarly simple instructions** and build our programs with those:

```
sub  8(t4), 0(t1), 0(t2)
```

```
lw   t3, 0(t1)
lw   t5, 0(t2)
sub  t3, t3, t5
sw   t3, 8(t4)
```

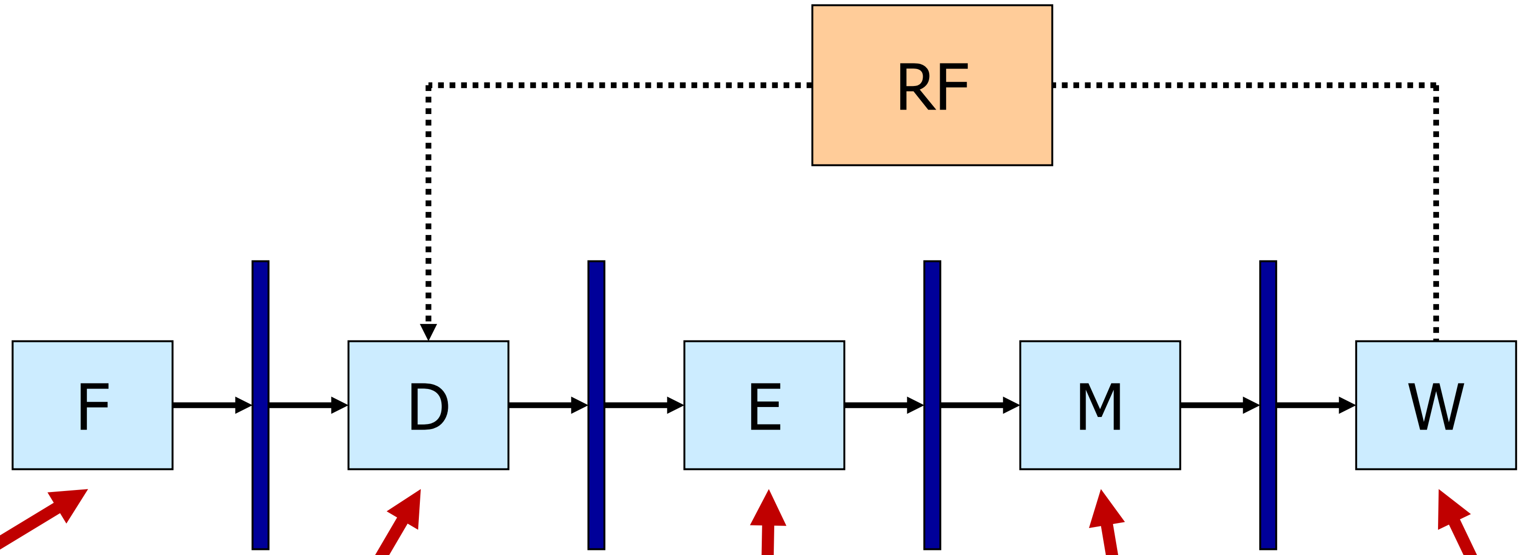
- It turns out that it is not the only way to go, but it is a **good one** and we will follow it…

Two Main Problems

1. CISC vs. **RISC**
   - Can we **build equally well a pipeline** for a Complex Instruction Set Computer as for a Reduced Instruction Set Computer?
   - What does that mean?!

2. Instructions are **not** independent
   - Can we execute code **correctly**?

Simple 5-Stage MIPS Pipeline



FETCH:
get the instruction to execute from the *instruction* memory

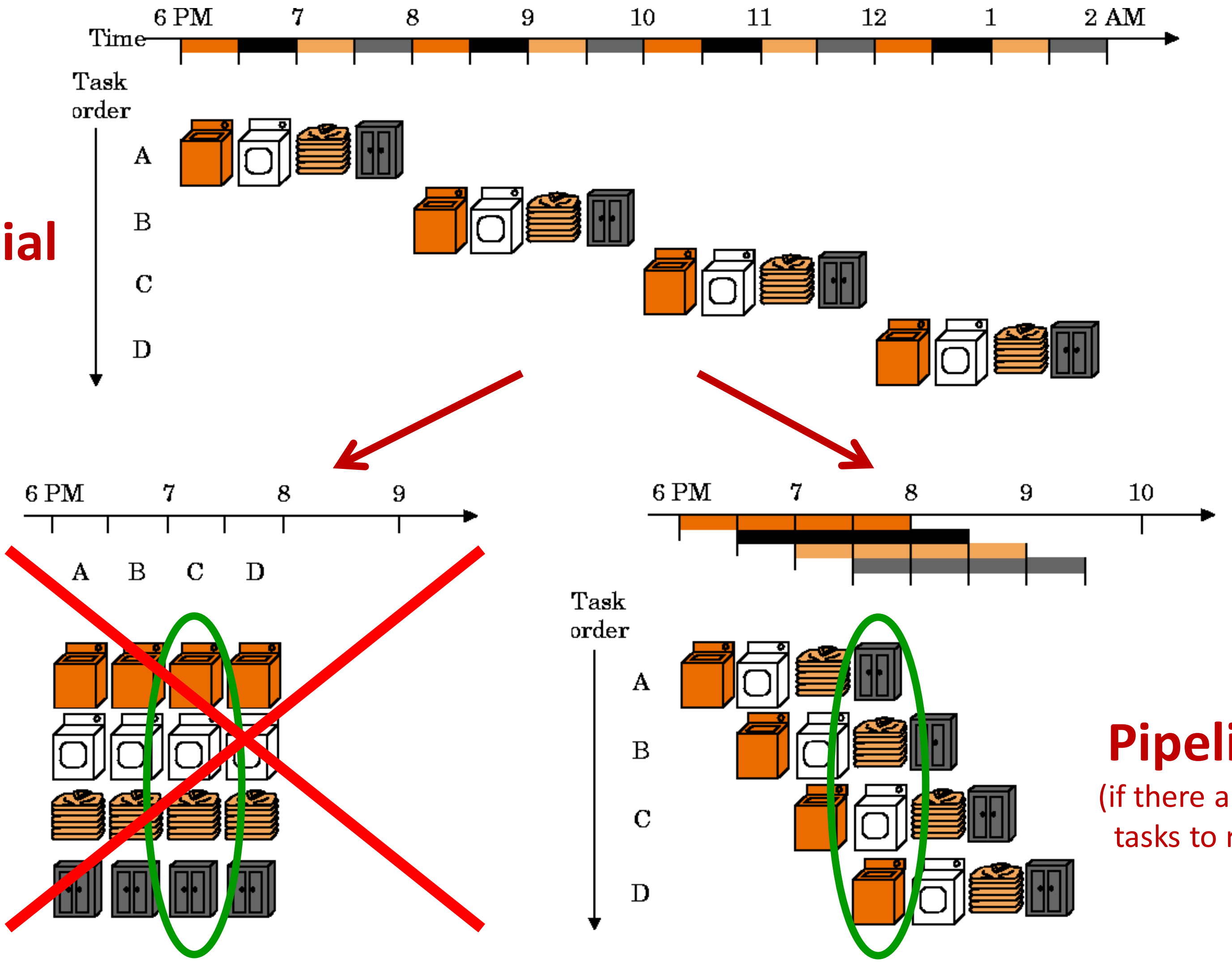
DECODE:
understand/decode the instruction **and** obtain arguments from the register file

EXECUTE:
perform the required operation in the ALU (including address calculations for loads and stores)

MEMORY:
access the *data* memory, if needed (only for loads and stores)

WRITEBACK:
write the result of the operation, if any, to the register file (either produced from the ALU or received from the data memory)

The Laundry Metaphor

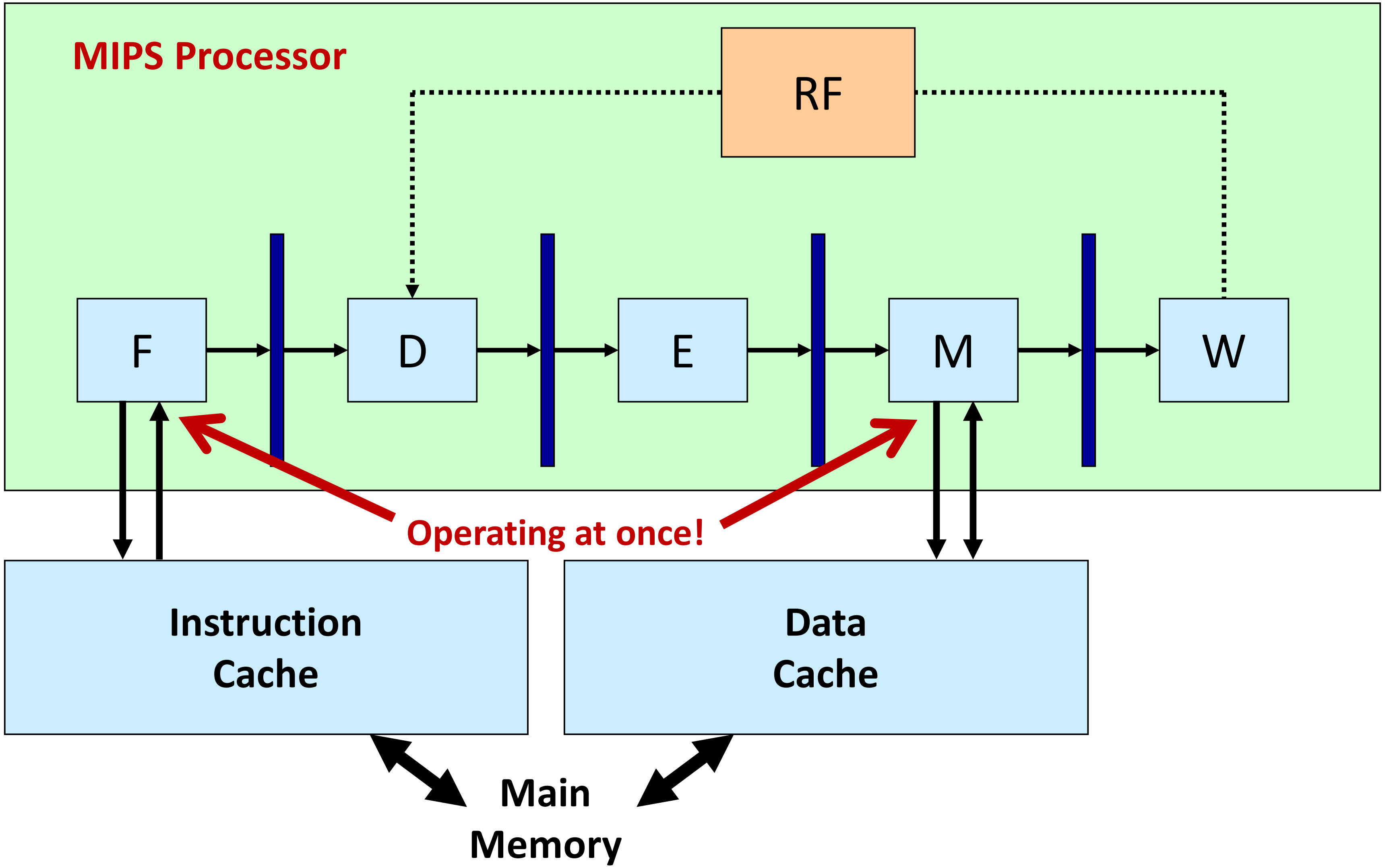


Sequential

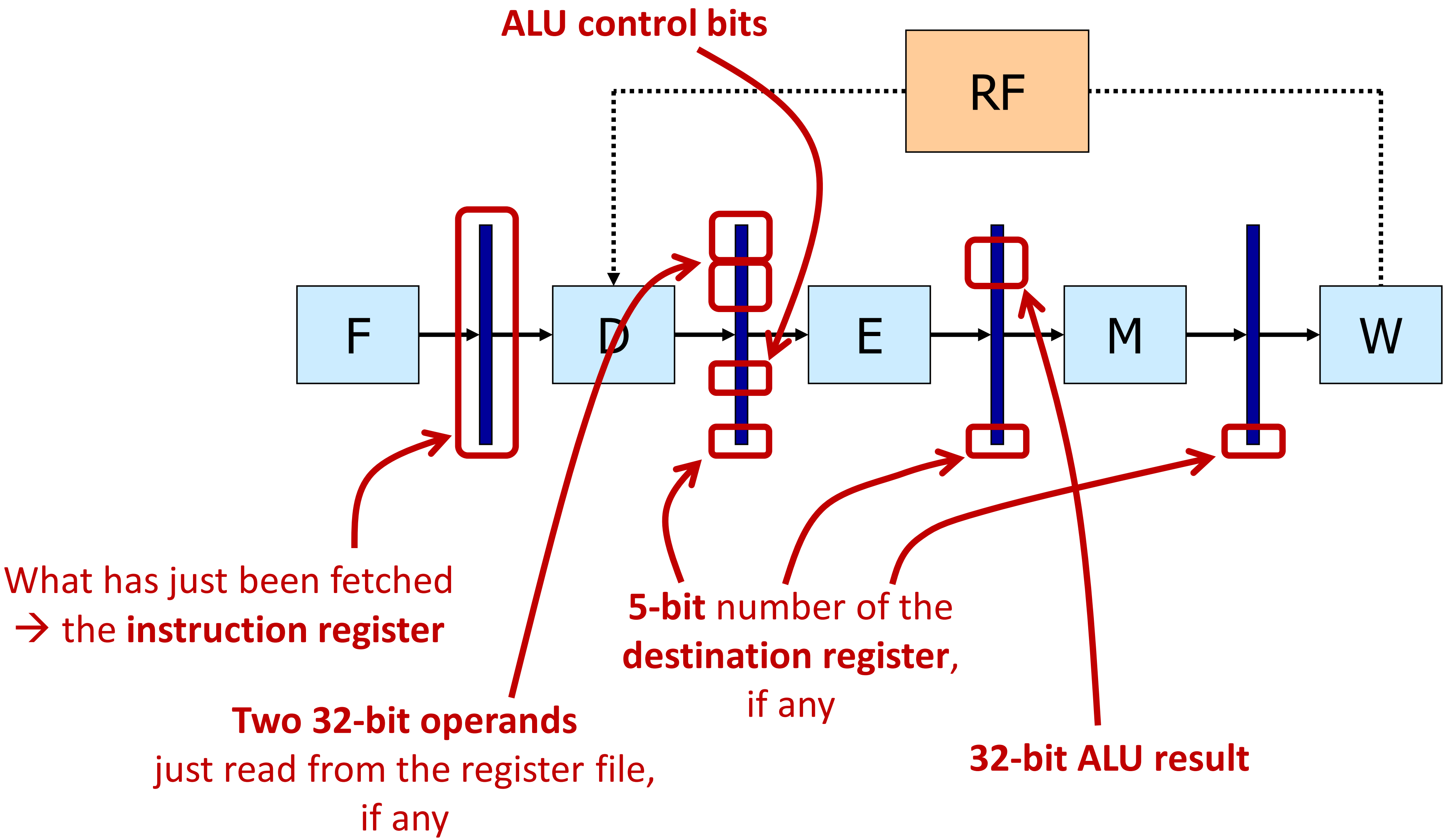
Parallel
(if subtasks are independent)

Pipelined
(if there are many tasks to repeat)

Two Distinct Memory Interfaces

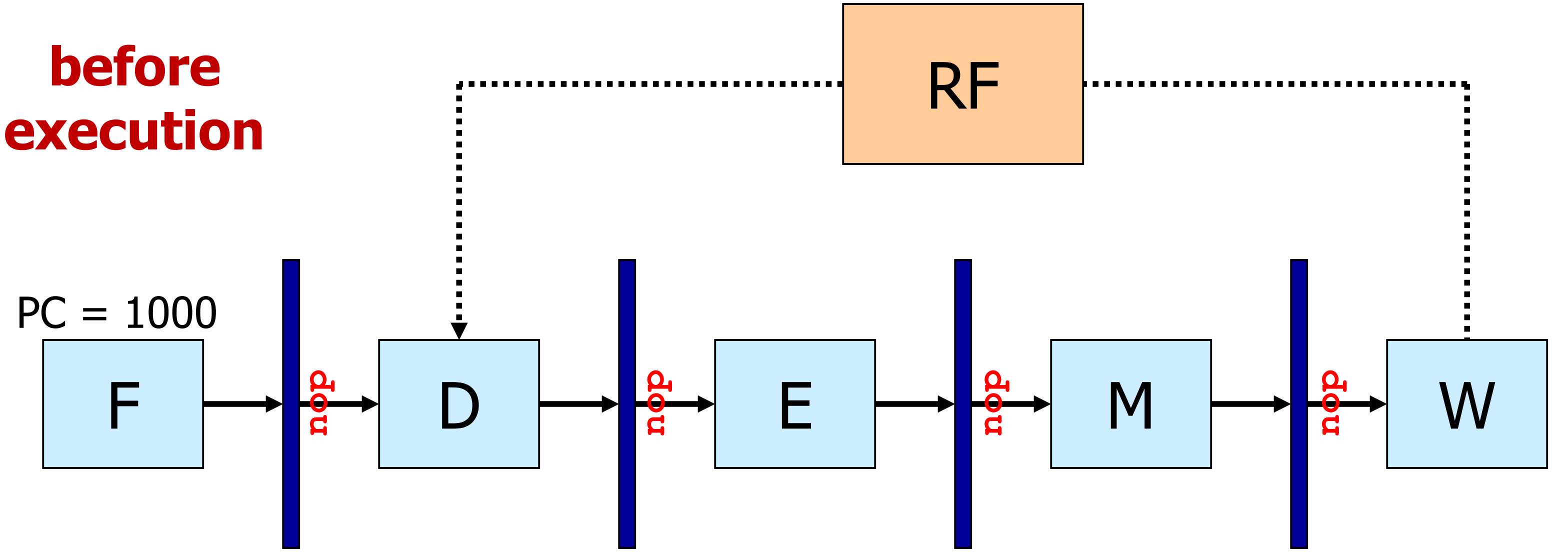


What Is in the Pipeline Registers?



Example of Pipelined Execution

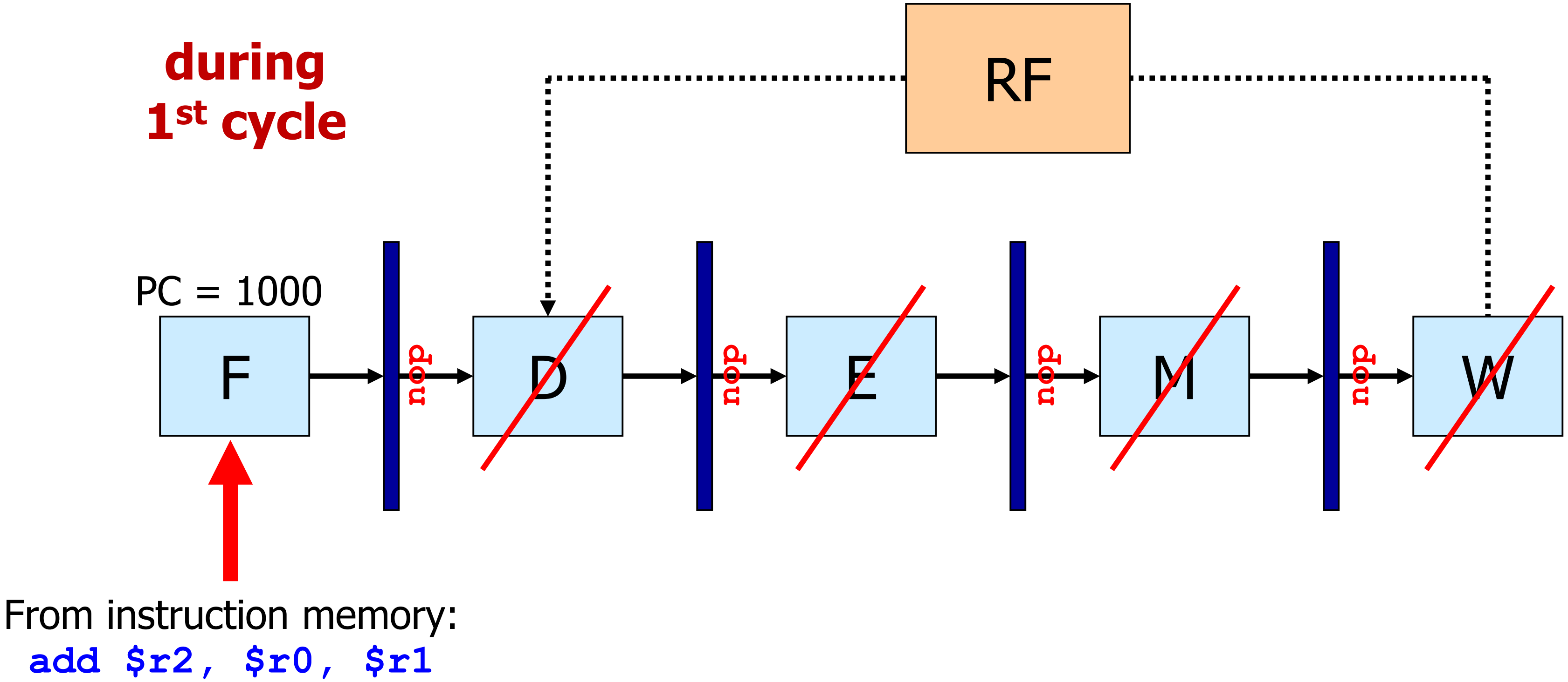
before execution



Empty pipeline: all stages at **nop** (no operation)

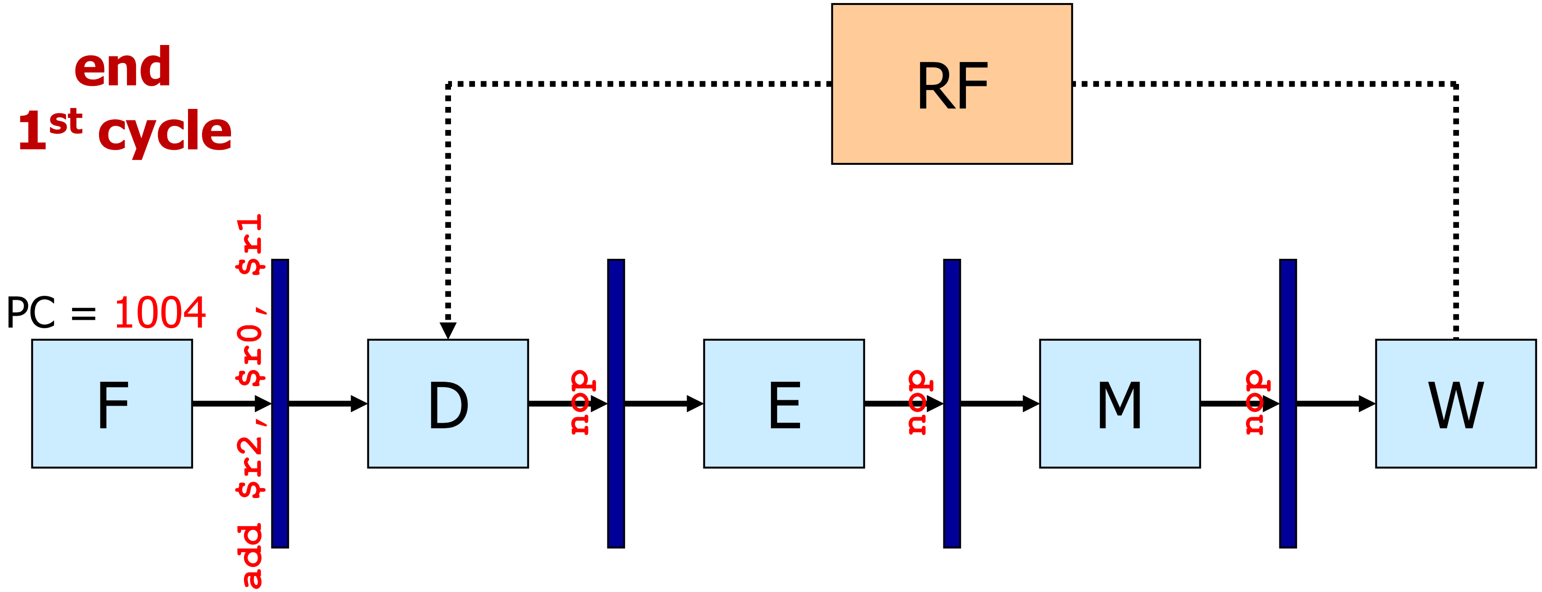
```
1000: add      $r2,  $r0,  $r1
1004: sub      $r5,  $r3,  $r4
1008: sw       $r6,  50($r7)
1012: lw       $r9,  20($r8)
1016: mul      $r12, $r10, $r11
```

Example of Pipelined Execution



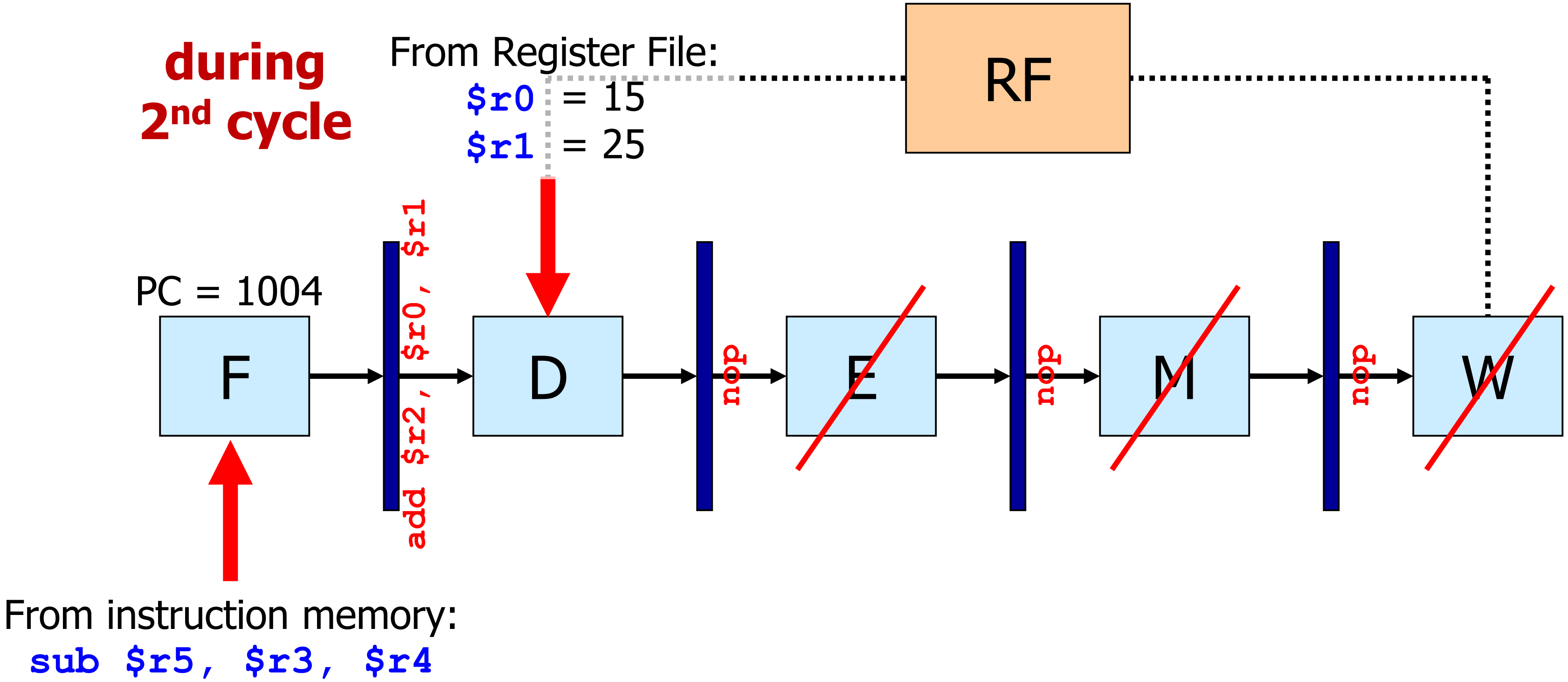

```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

Example of Pipelined Execution



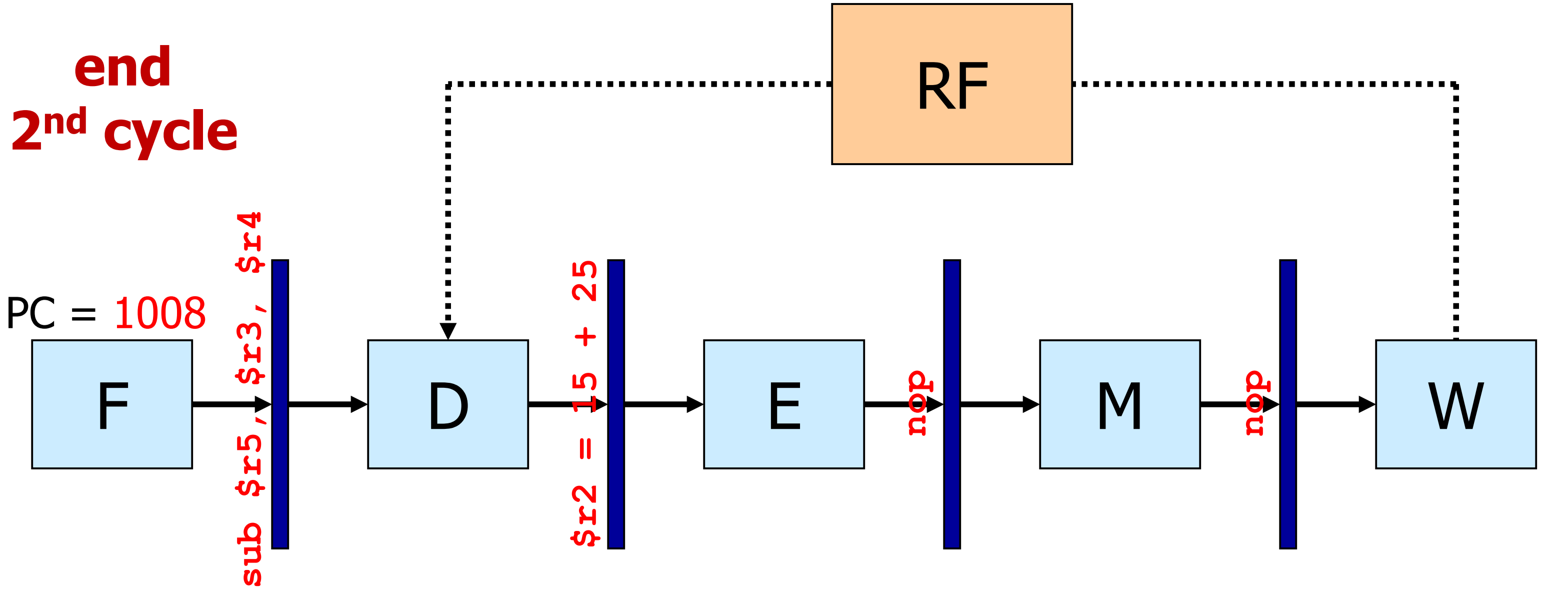

```
1000: add     $r2,  $r0,  $r1
1004: sub     $r5,  $r3,  $r4
1008: sw      $r6,  50($r7)
1012: lw      $r9,  20($r8)
1016: mul     $r12, $r10, $r11
```

Example of Pipelined Execution



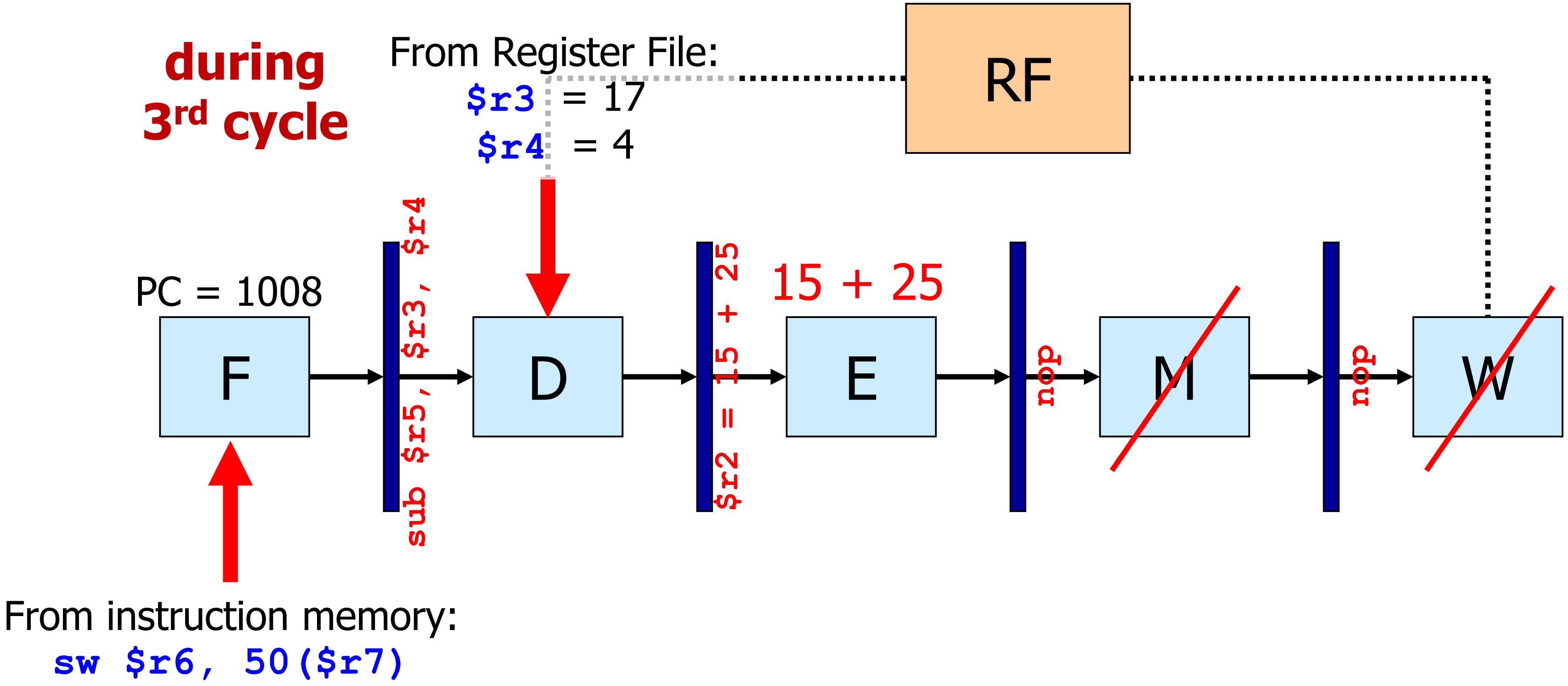

```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

Example of Pipelined Execution




```
1000: add     $r2,  $r0,  $r1
1004: sub     $r5,  $r3,  $r4
1008: sw      $r6,  50($r7)
1012: lw      $r9,  20($r8)
1016: mul     $r12, $r10, $r11
```

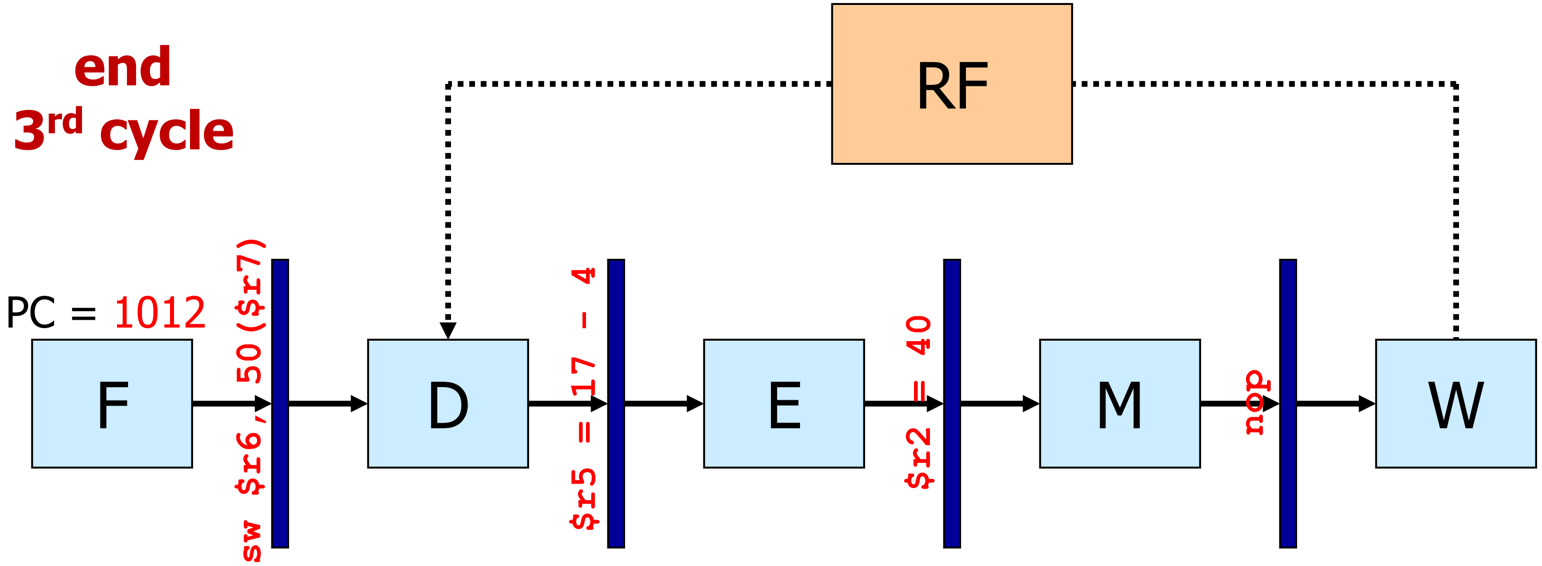
Example of Pipelined Execution




```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

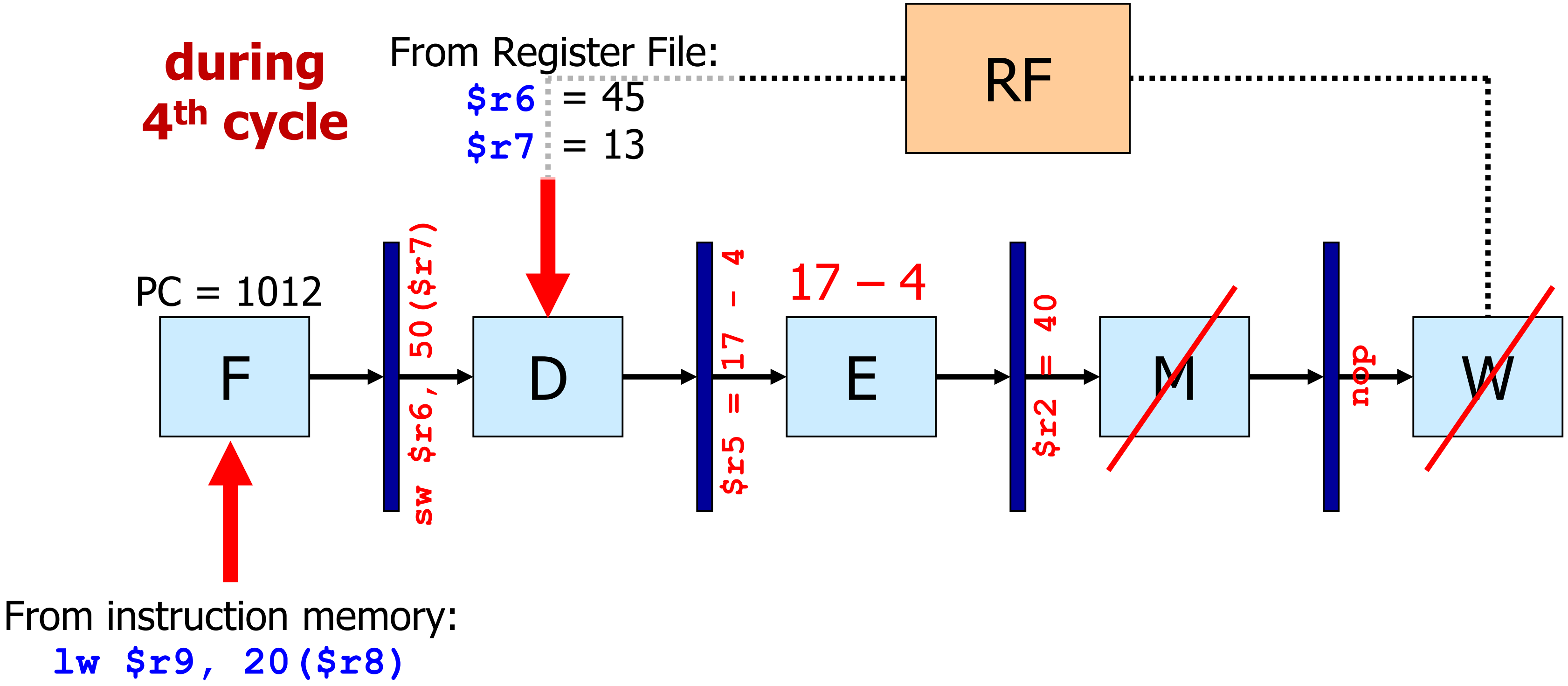
Example of Pipelined Execution

end 3rd cycle



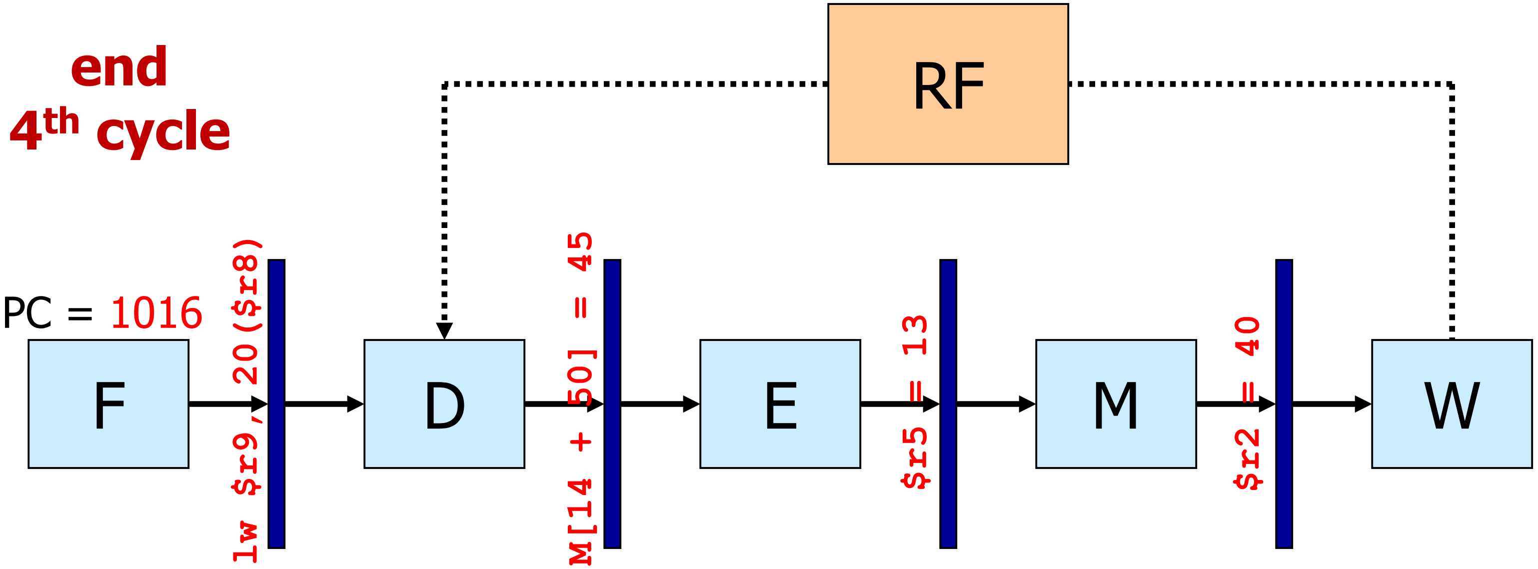

```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

Example of Pipelined Execution



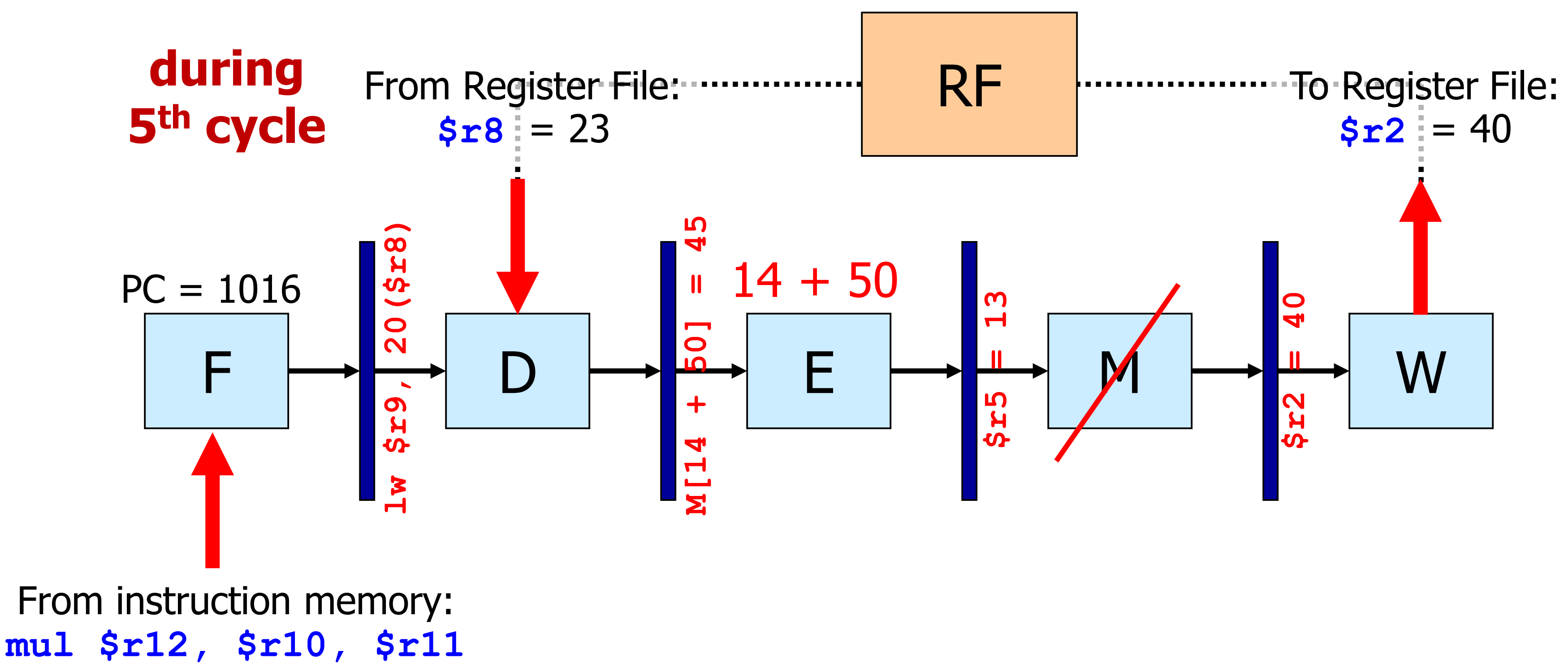

```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

Example of Pipelined Execution



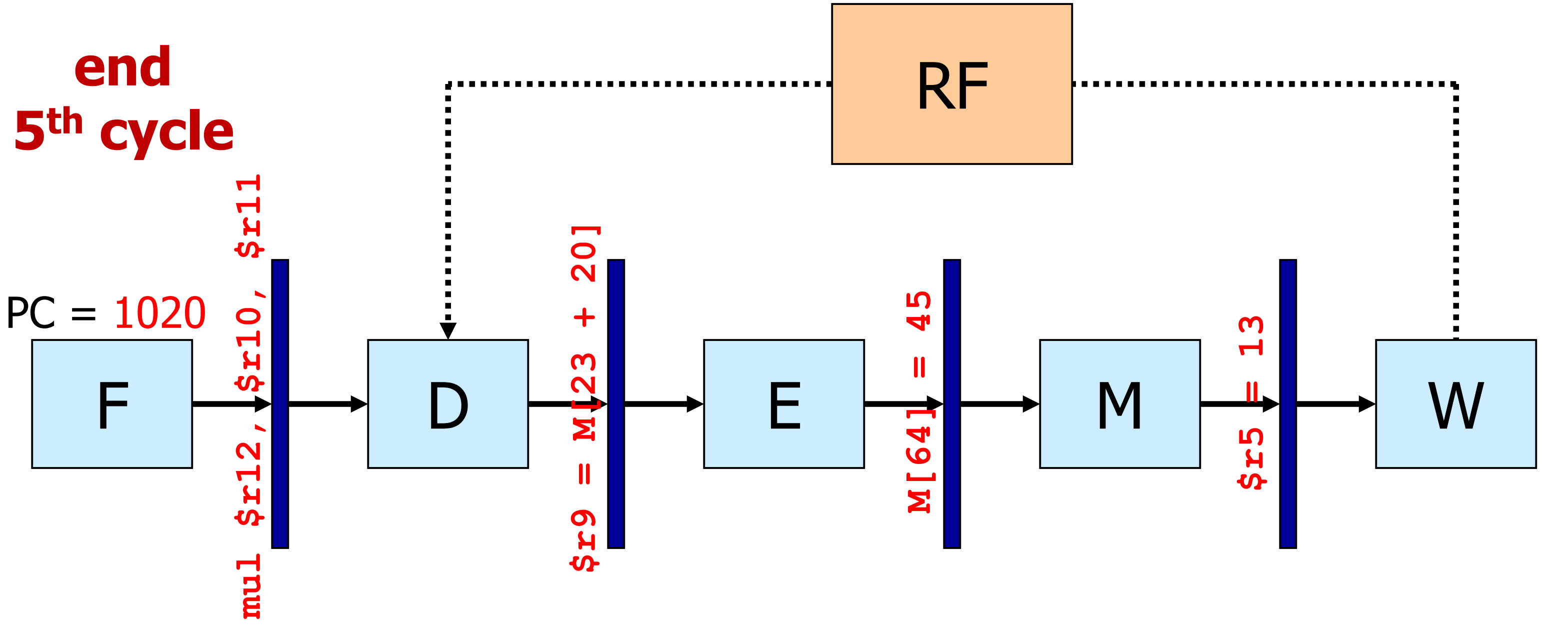

```
1000: add     $r2,  $r0,  $r1
1004: sub     $r5,  $r3,  $r4
1008: sw      $r6,  50($r7)
1012: lw      $r9,  20($r8)
1016: mul     $r12, $r10, $r11
```

Example of Pipelined Execution



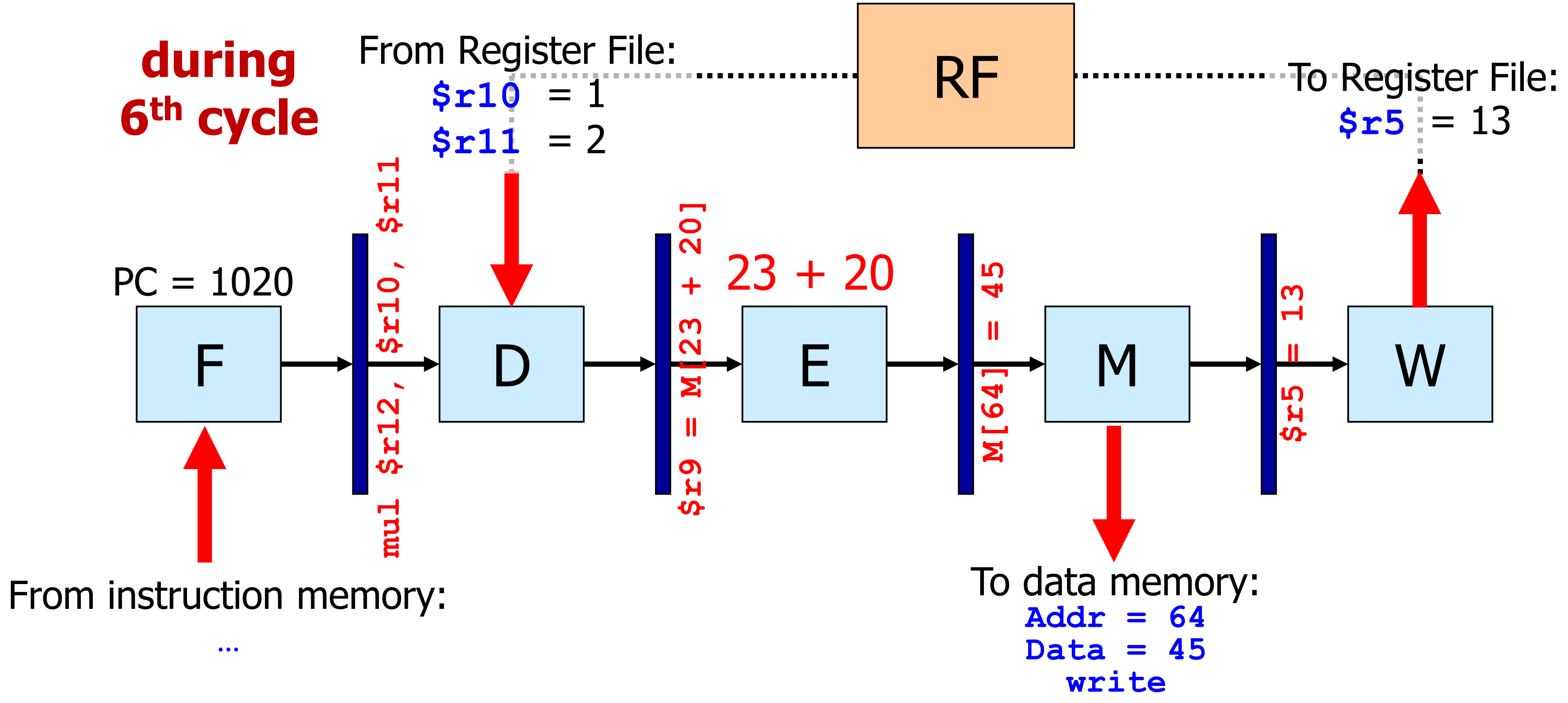

```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

Example of Pipelined Execution



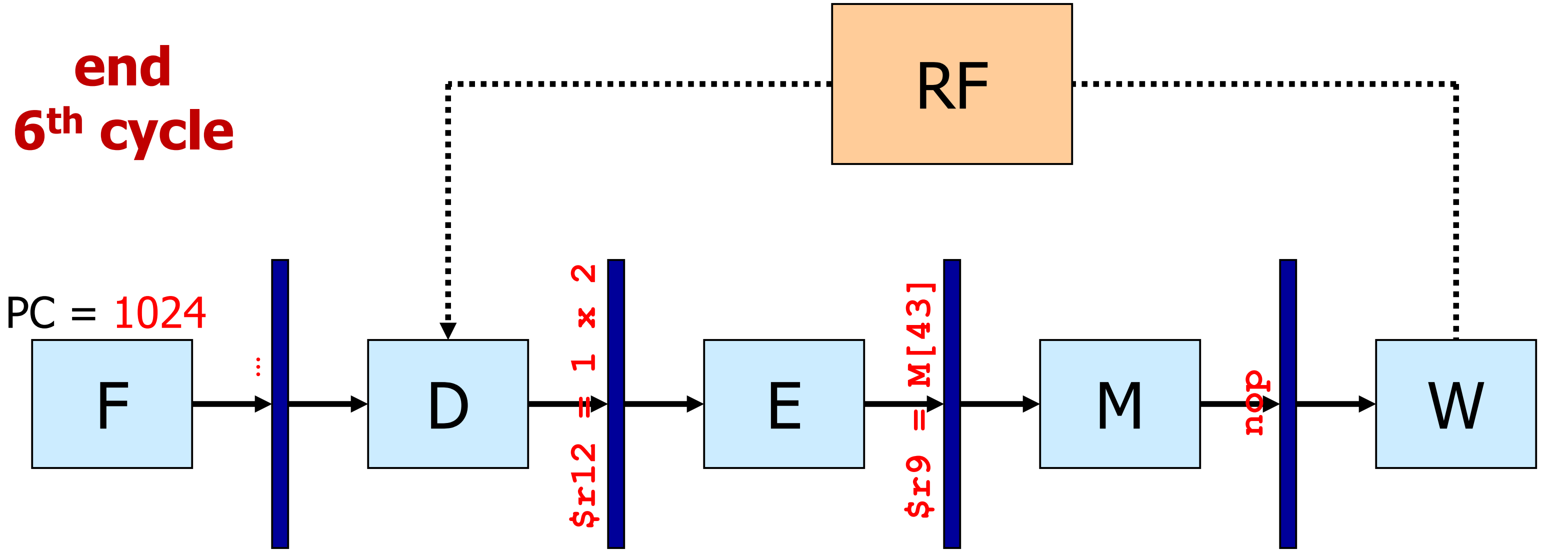

```
1000: add     $r2,  $r0,  $r1
1004: sub     $r5,  $r3,  $r4
1008: sw      $r6,  50($r7)
1012: lw      $r9,  20($r8)
1016: mul     $r12, $r10, $r11
```

Example of Pipelined Execution



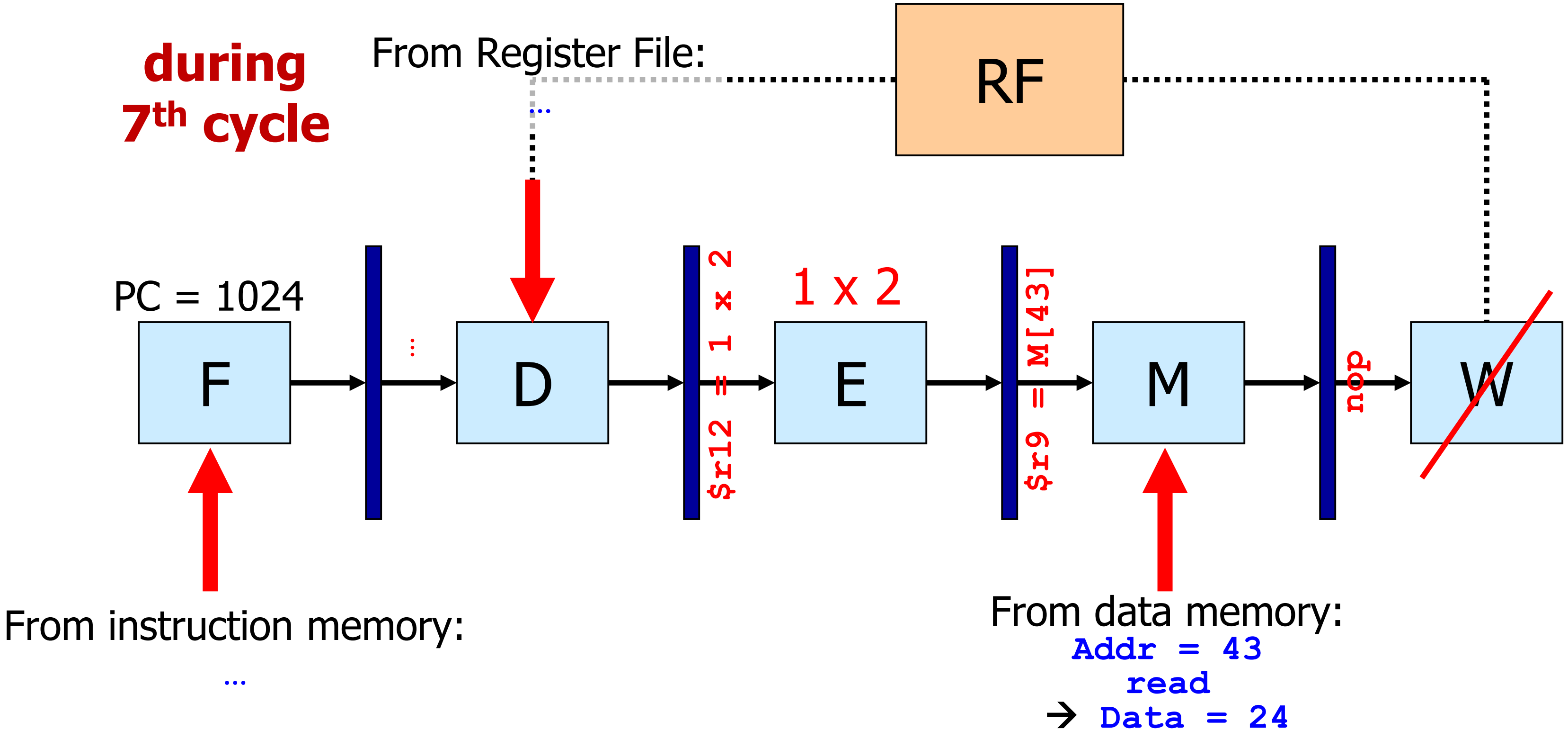

```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

Example of Pipelined Execution



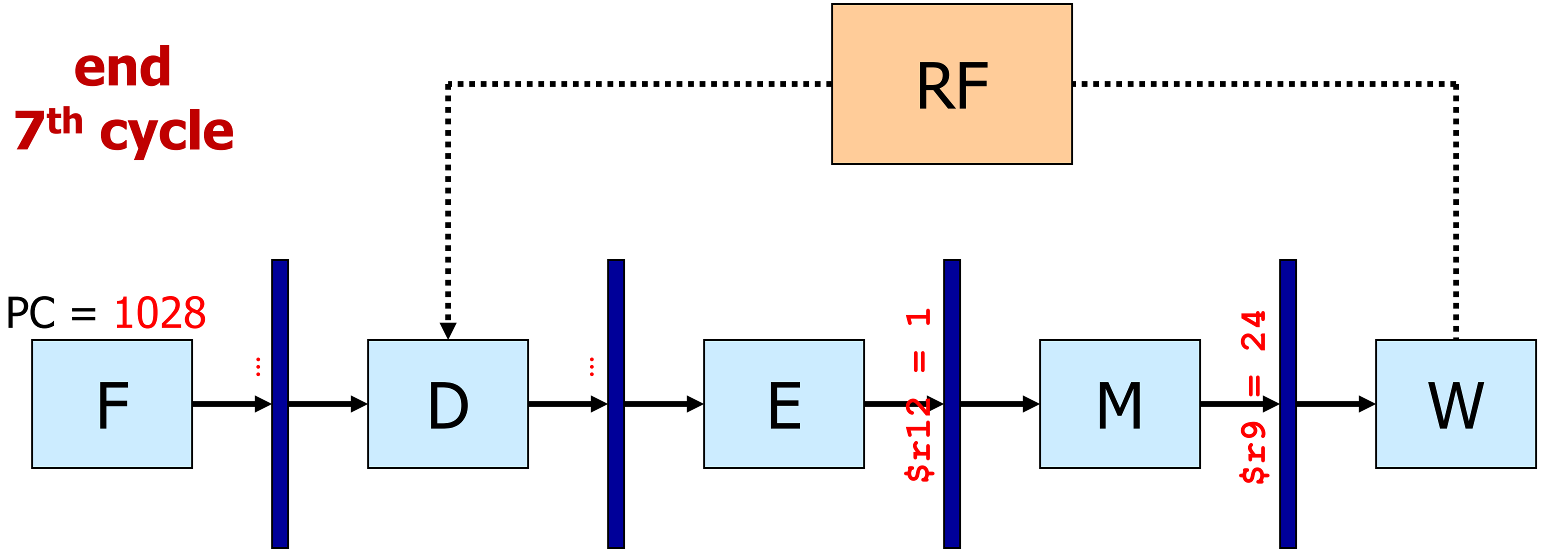

```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

Example of Pipelined Execution



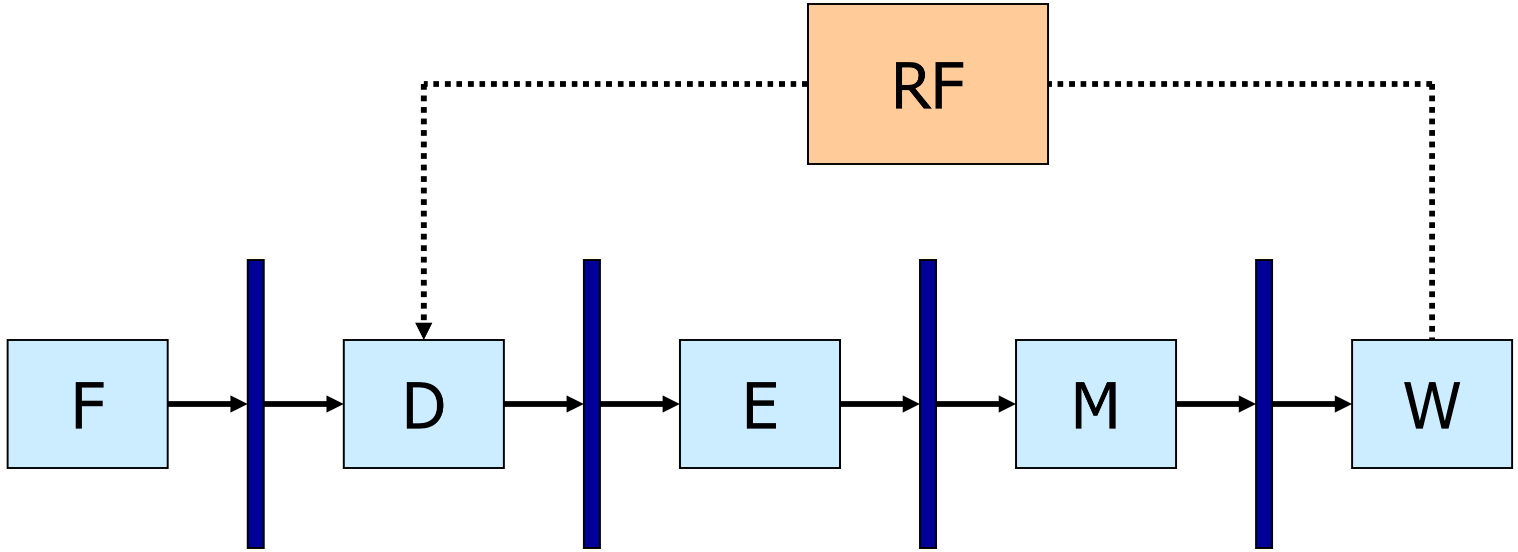

```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

Example of Pipelined Execution



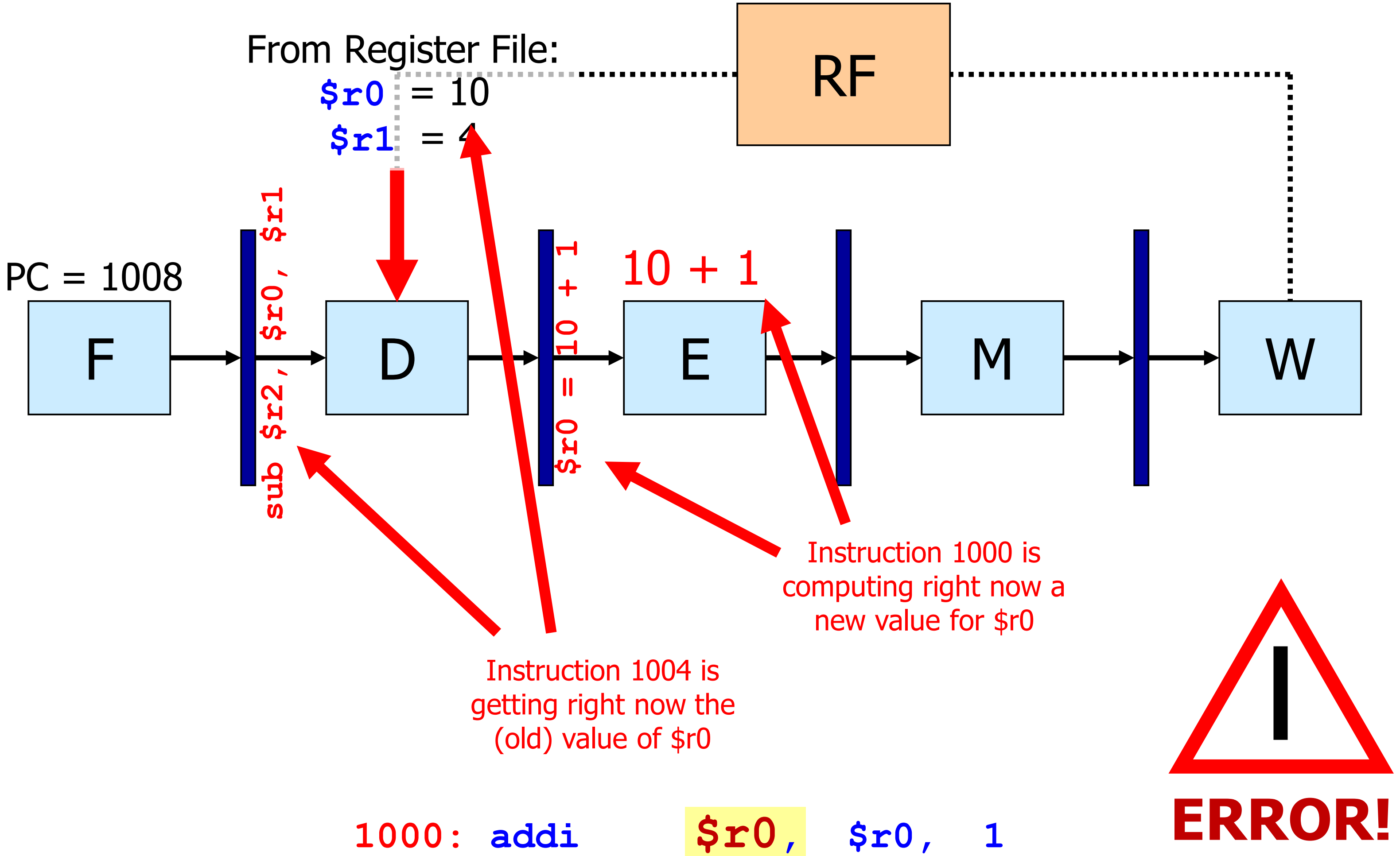

```
1000: add    $r2,  $r0,  $r1
1004: sub    $r5,  $r3,  $r4
1008: sw     $r6,  50($r7)
1012: lw     $r9,  20($r8)
1016: mul    $r12, $r10, $r11
```

Pipeline Schedule



1000:	F	D	E	M	W				
1004:		F	D	E	M	W			
1008:			F	D	E	M	W		
1012:				F	D	E	M	W	
1016:					F	D	E	M	W

Problem!




```
1000: addi   $r0,  $r0,  1
1004: sub    $r2,  $r0,  $r1
```

Two Main Problems

1. CISC vs. **RISC**
   – Can we **build equally well a pipeline** for a Complex Instruction Set Computer as for a Reduced Instruction Set Computer?
   – What does that mean?!

2. Instructions are **not** independent
   – Can we execute code **correctly**?

Write After Read

RAW, WAR and WAW Dependences

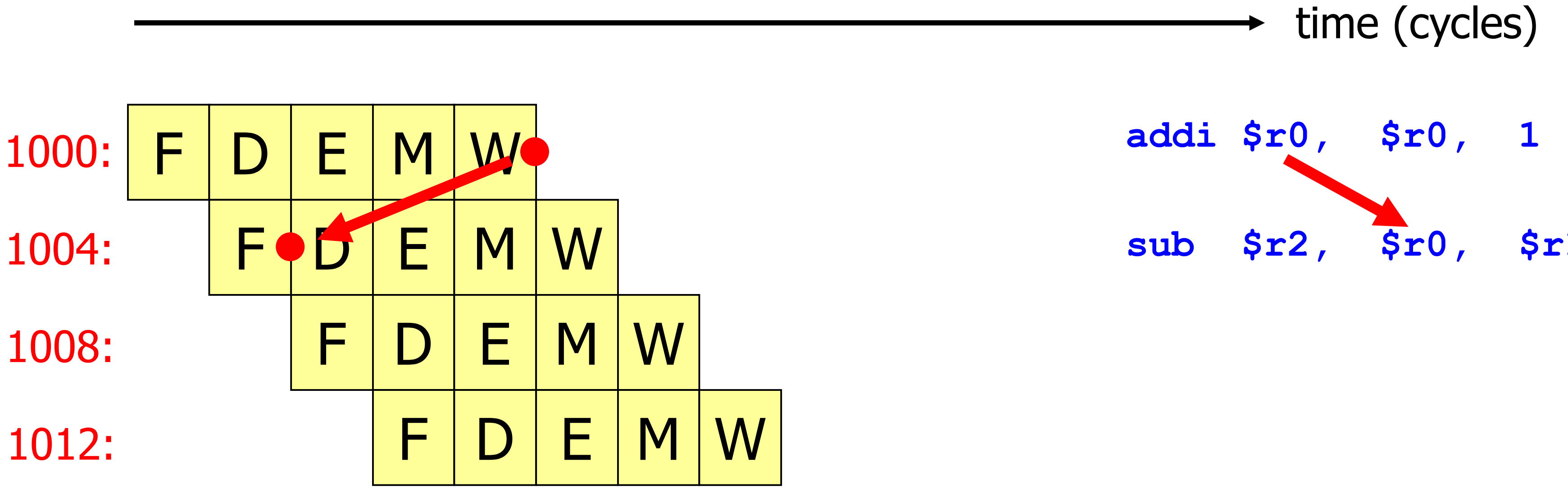
```
divd    $f0, $f1, $f2
addd    $f3, $f0, $f4
subd    $f4, $f5, $f6
adddi   $f0, $f5, 10
```

- **addd** has a RAW dependence on **divd**
- **subd** has a WAR dependence on **addd**
- **adddi** has a WAW dependence on **divd**

True or **data** dependencies

Name dependencies

Data Hazards

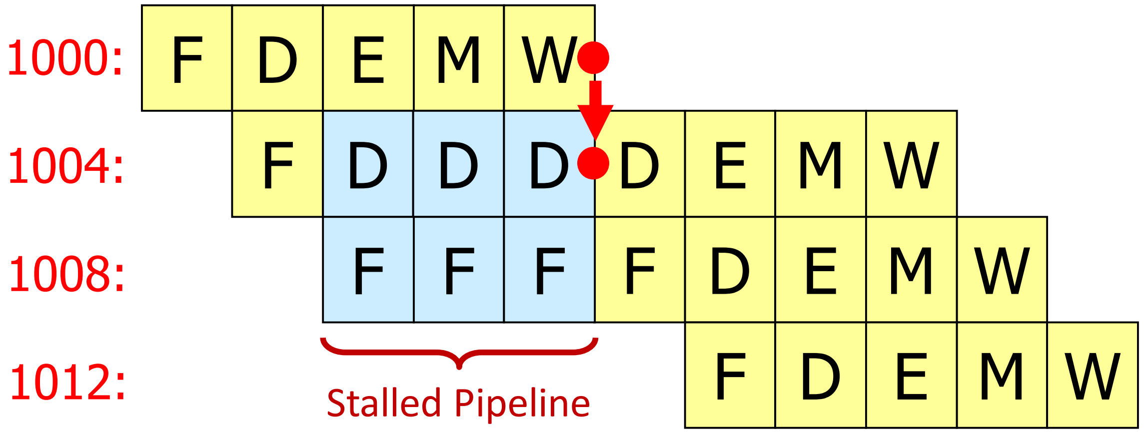



```
addi  $r0,  $r0,  1

sub   $r2,  $r0,  $r1
```

Causality violation!
We try to use a result before it is produced!

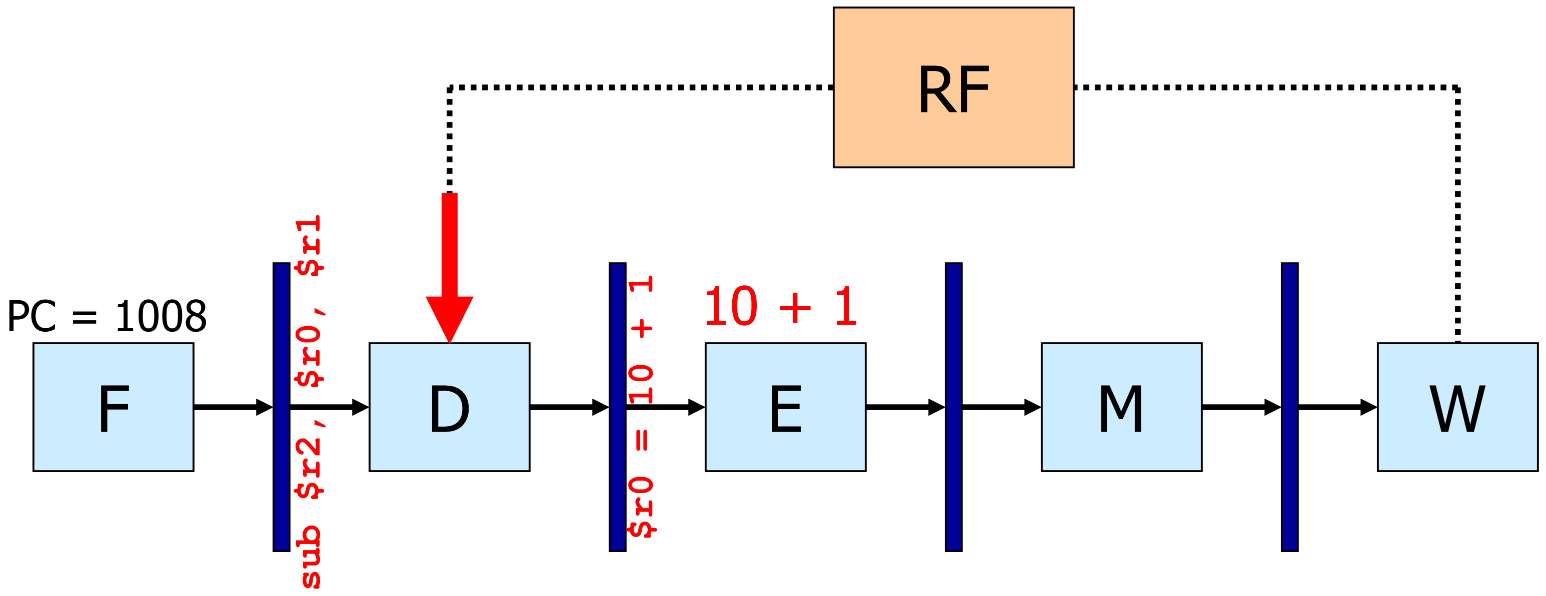
Data Hazards Solved by Stalling the Pipeline




```
addi $r0, $r0, 1
sub  $r2, $r0, $r1
```

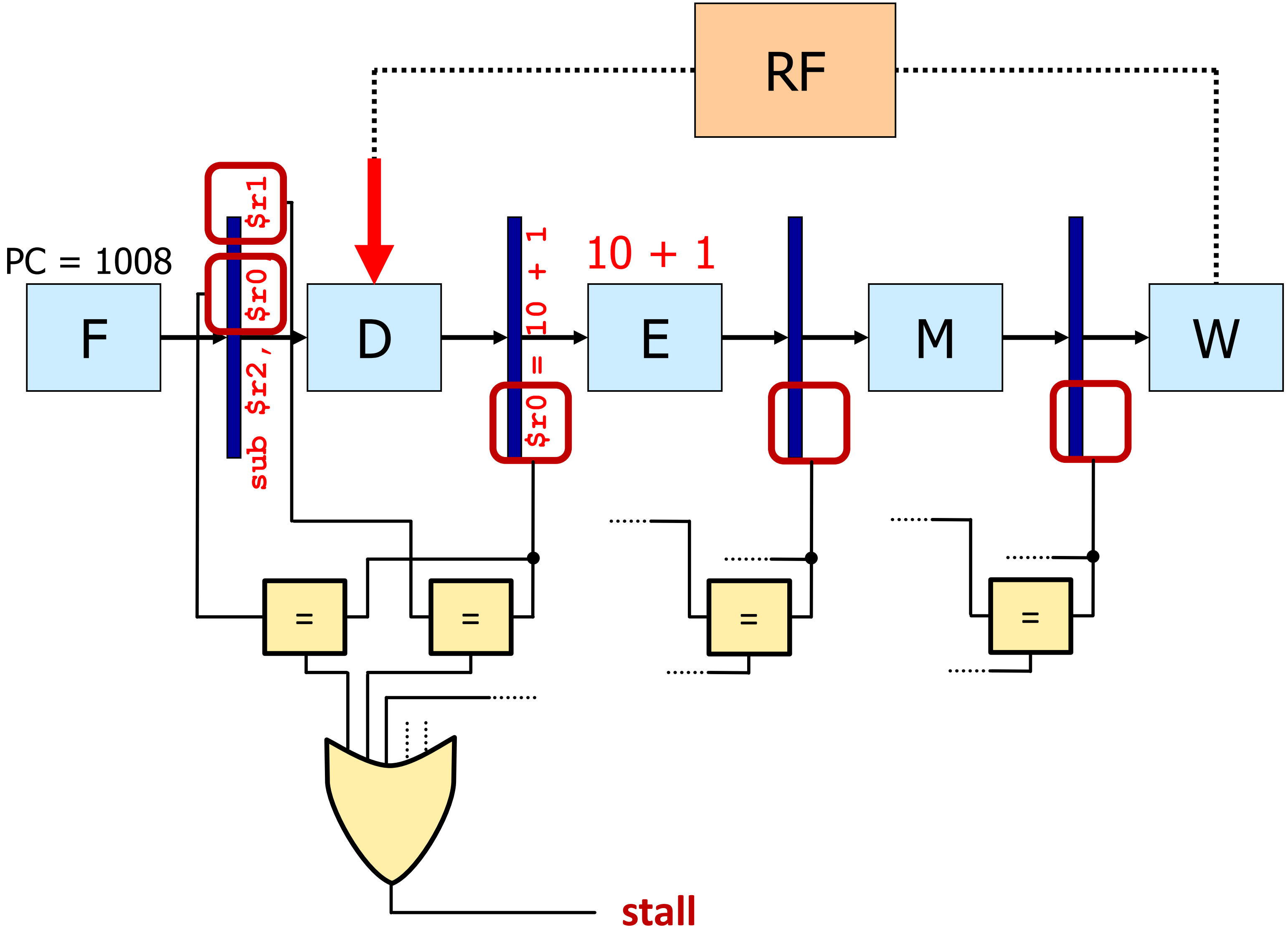
- The natural solution to **Data Hazards** caused by **RAW** dependences is to implement some logic in the processor to stop/repeat the decoding until the required value is available
- "**Stalling**" roughly means introducing `nop`'s in the pipeline
- Due to the rigidity of the pipeline, if one stage is stalled (**D** in the example), all the preceding ones must be stalled too (e.g., **F**)

```
1000: addi    $r0,  $r0,  1
1004: sub     $r2,  $r0,  $r1
```

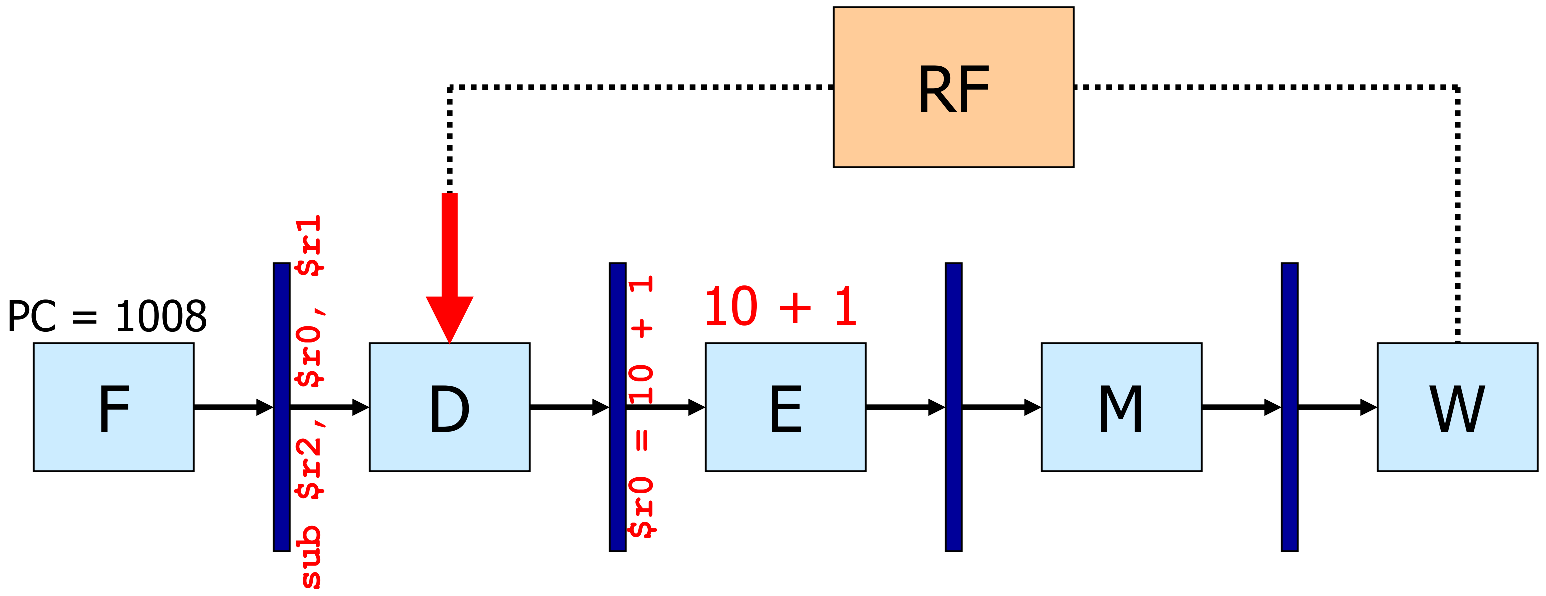


```
1000: addi    $r0,  $r0,  1
1004: sub     $r2,  $r0,  $r1
```

Detecting

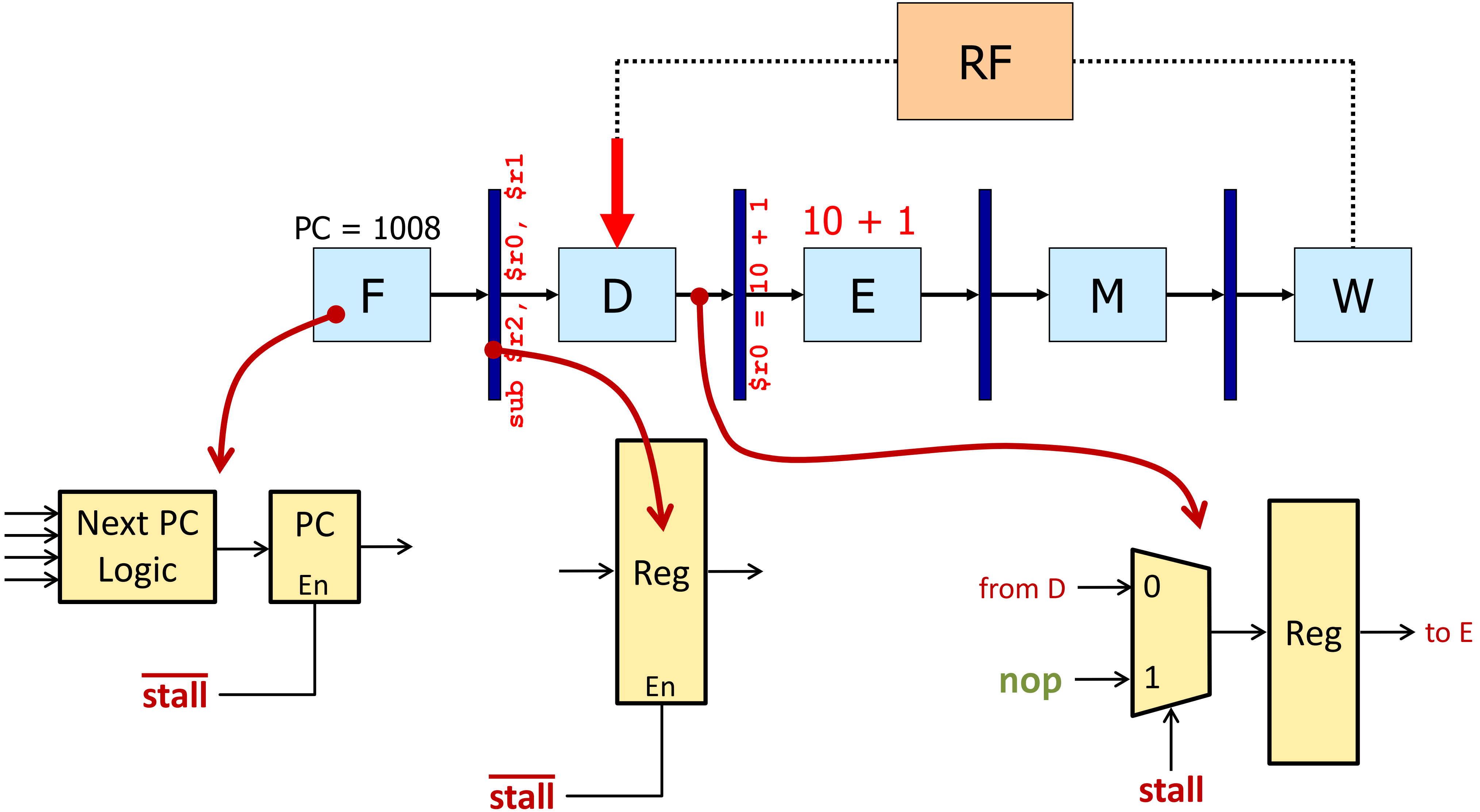


```
1000: addi    $r0,  $r0,  1
1004: sub     $r2,  $r0,  $r1
```

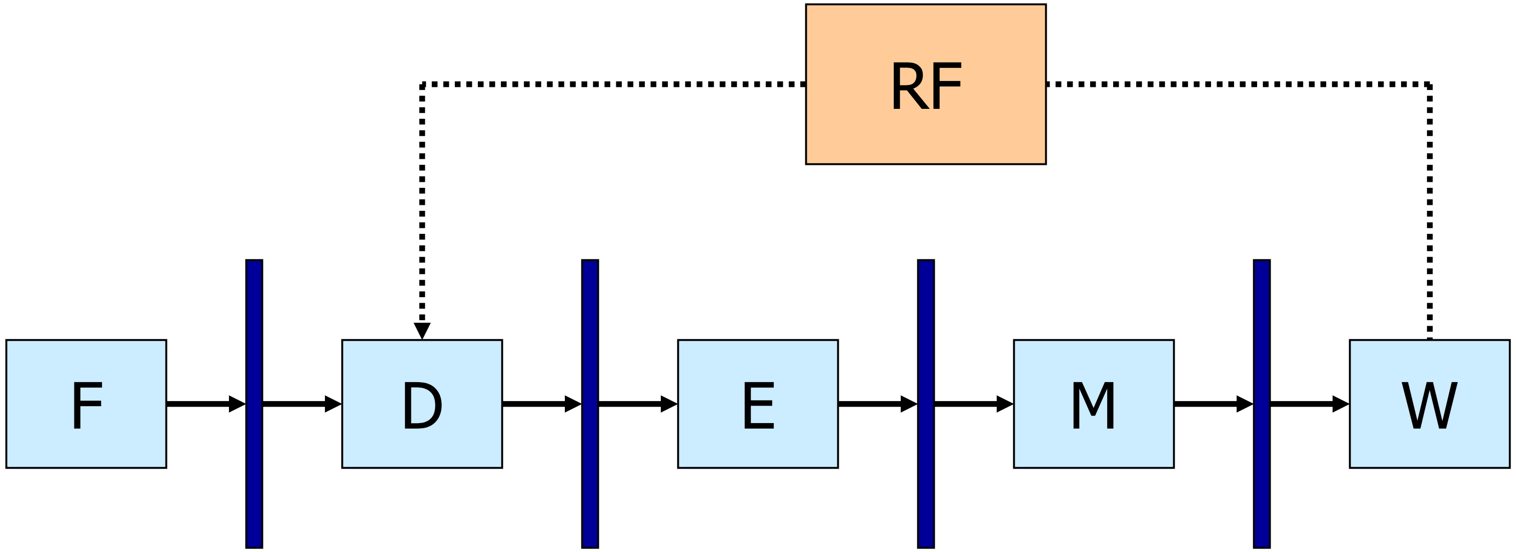


```
1000: addi    $r0,  $r0,  1
1004: sub     $r2,  $r0,  $r1
```

Stalling

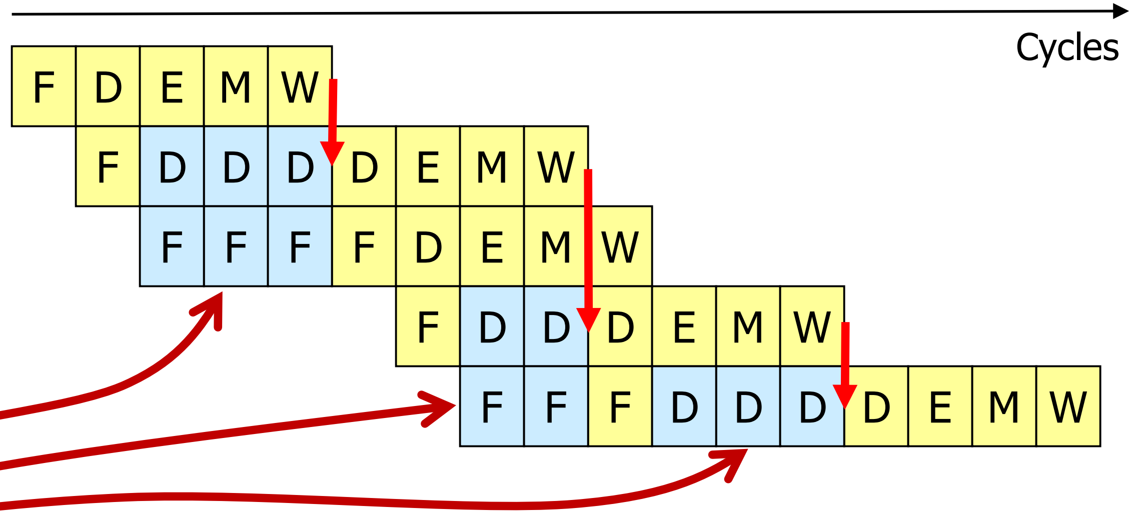


```
1: addi   $r5, $r1, 1
2: add    $r7, $r5, $r2
3: xor    $r1, $r1, $r2
4: lw     $r9, 0($r7)
5: subi   $r3, $r9, 1
```

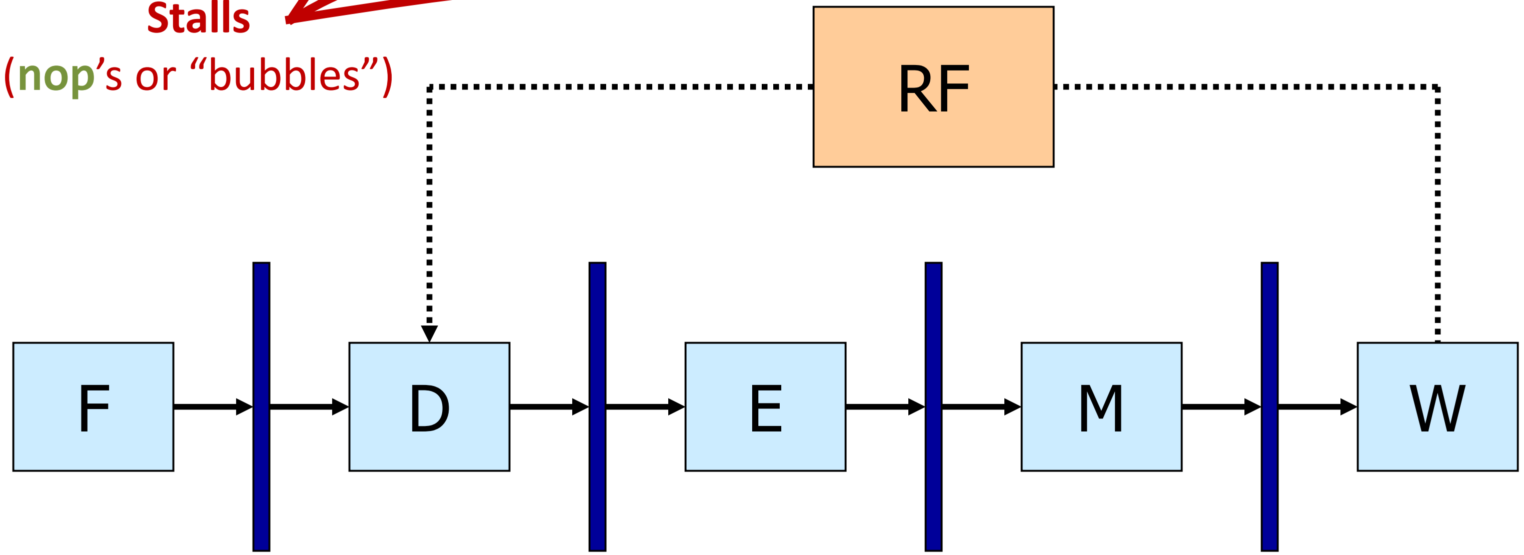


Data Hazards

```
1: addi   $r5, $r1, 1
2: add    $r7, $r5, $r2
3: xor    $r1, $r1, $r2
4: lw     $r9, 0($r7)
5: subi   $r3, $r9, 1
```



Stalls
(nop's or "bubbles")

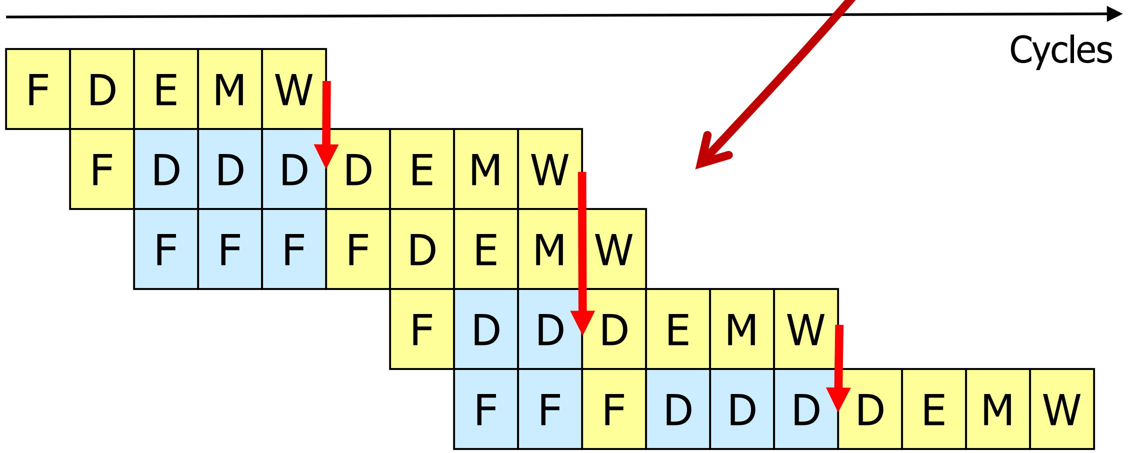


Another Solution?

```
1:  addi   $r5, $r1, 1
2:  add    $r7, $r5, $r2
3:  xor    $r1, $r1, $r2
4:  lw     $r9, 0($r7)
5:  subi   $r3, $r9, 1
```

This solution needs some **hardware**

This solution needs a good **compiler**



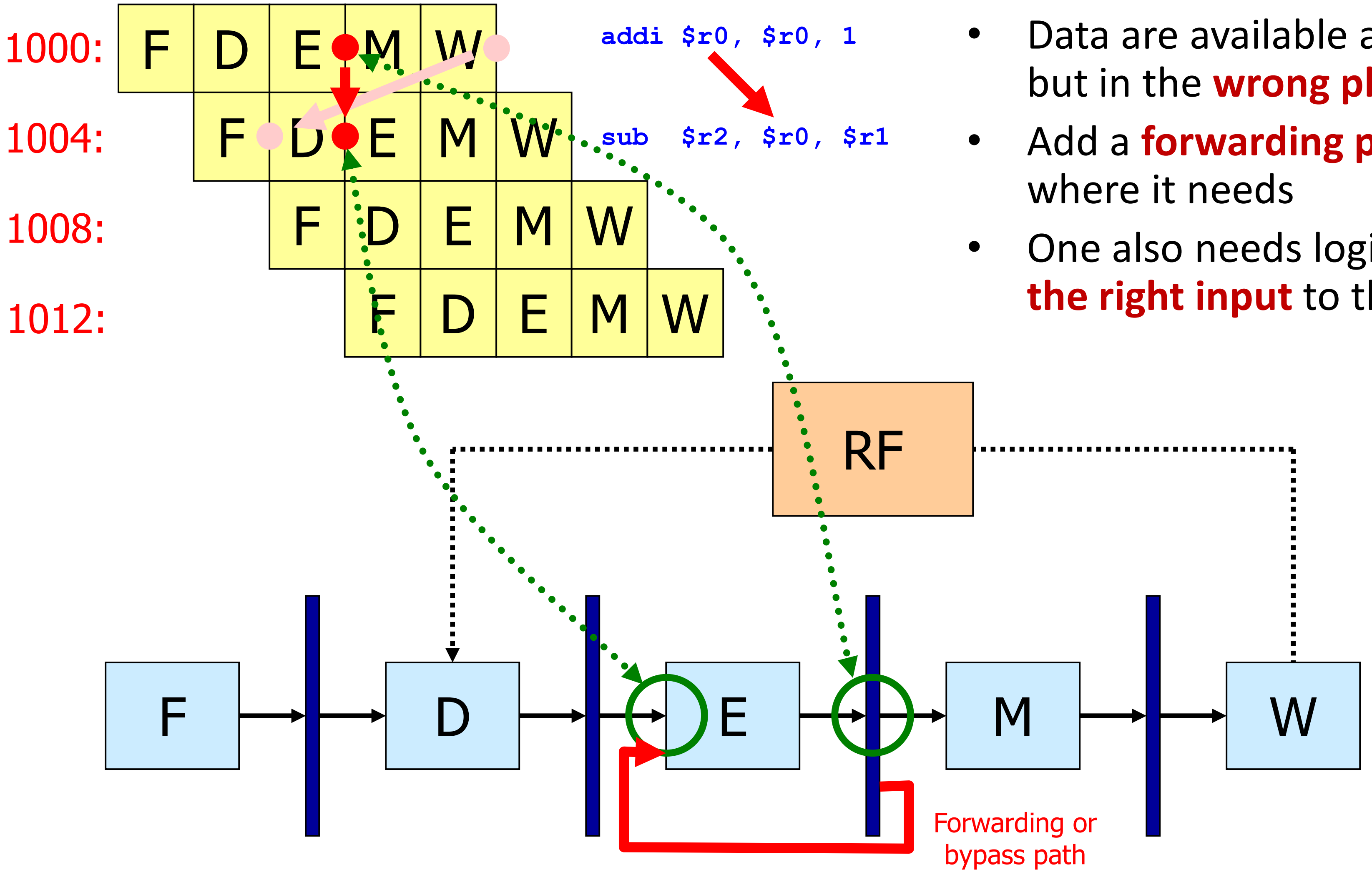

```
1:   addi   $r5, $r1, 1
2:   nop
3:   nop
4:   nop
5:   add    $r7, $r5, $r2
6:   xor    $r1, $r1, $r2
7:   nop
8:   nop
9:   lw     $r9, 0($r7)
10:  nop
11:  nop
12:  nop
13:  subi   $r3, $r9, 1
```

Architecture and Microarchitecture

- **Architecture**: what is in the ISA contract
  - Instructions, registers, etc.
- **Microarchitecture**: what is specific of an implementation
  - Multicycle vs. pipelined, FSM or pipeline structure, etc.

Some of the solutions evoked (reschedule instructions, add **nop**’s, delay slots) **expose typically microarchitectural aspects** (pipeline structure) **in the architecture** → the same binary **<u>does not run</u>** on different processors!

Data Hazards Solved by Forwarding Values



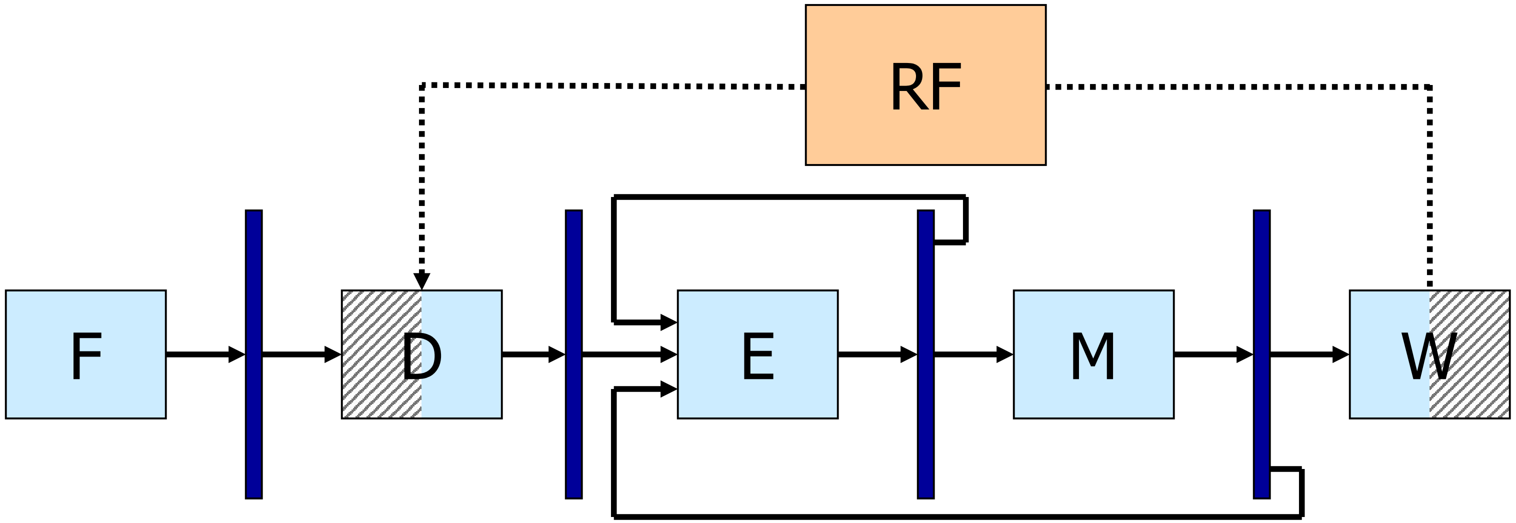
- Data are available at the **right time** but in the **wrong place**
- Add a **forwarding path** to bring it where it needs
- One also needs logic to **select (MUX) the right input** to the **E** stage

Classic MIPS Pipeline

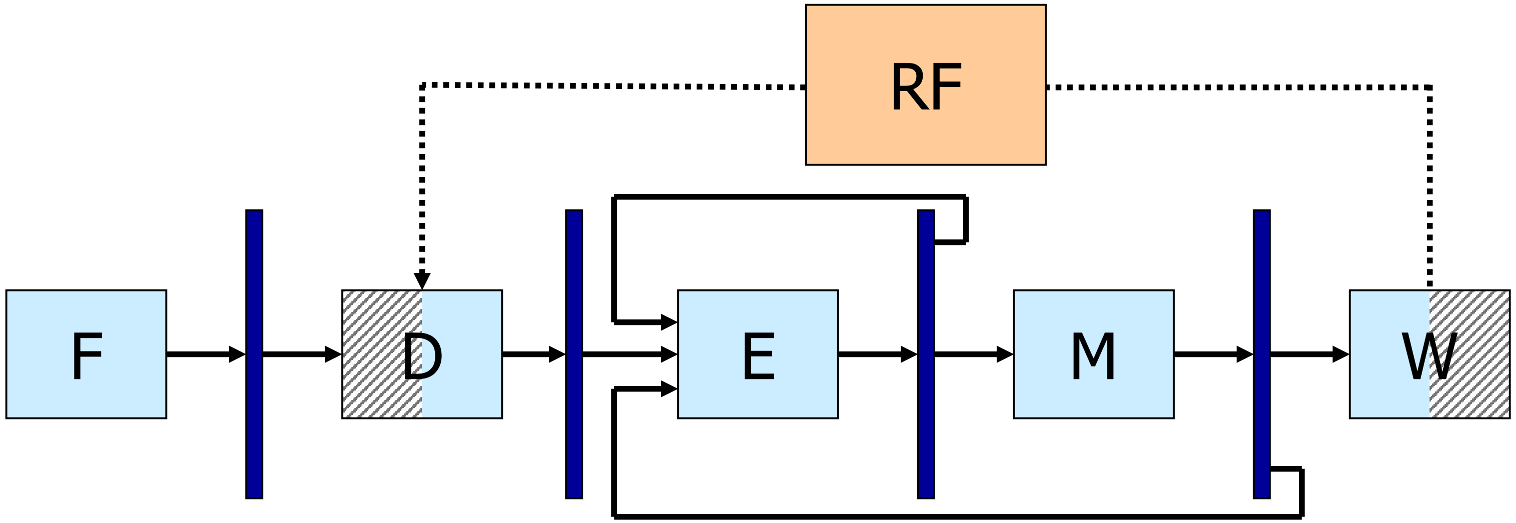
- 5-stage pipeline with **all forwarding paths**:
  - **E→E, M→E**
  - **W→D**
- The **register-file forwarding** (**W→D**) is a special case:
  - During **W**, registers are written in the **first half** of the cycle
  - During **D**, registers are read in the **second half** of the cycle

→ a register can be correctly written and read in the same cycle

- Not always all forwarding path exist…

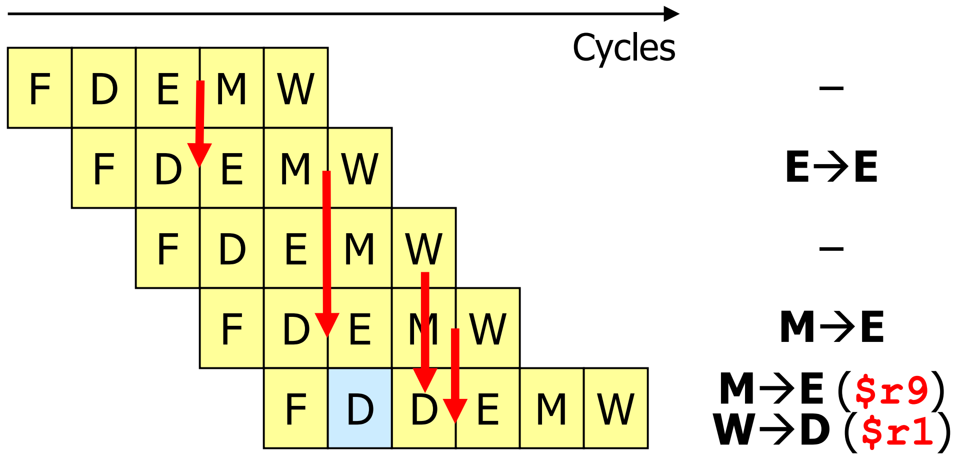


```
1: addi   $r5, $r1, 1
2: add    $r7, $r5, $r2
3: xor    $r1, $r1, $r2
4: lw     $r9, 0($r7)
5: subi   $r3, $r9, 1
```

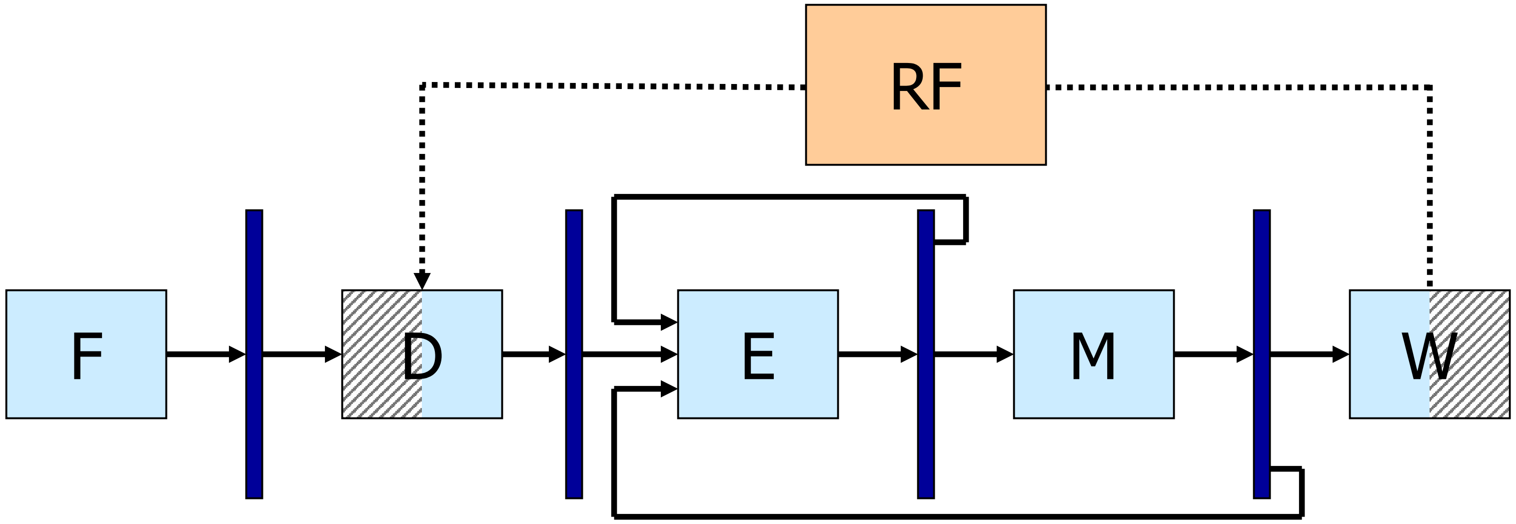


Reduced Data Hazards

```
1: addi   $r5, $r1, 1
2: add    $r7, $r5, $r2
3: xor    $r1, $r1, $r2
4: lw     $r9, 0($r7)
5: sub    $r3, $r9, $r1
```

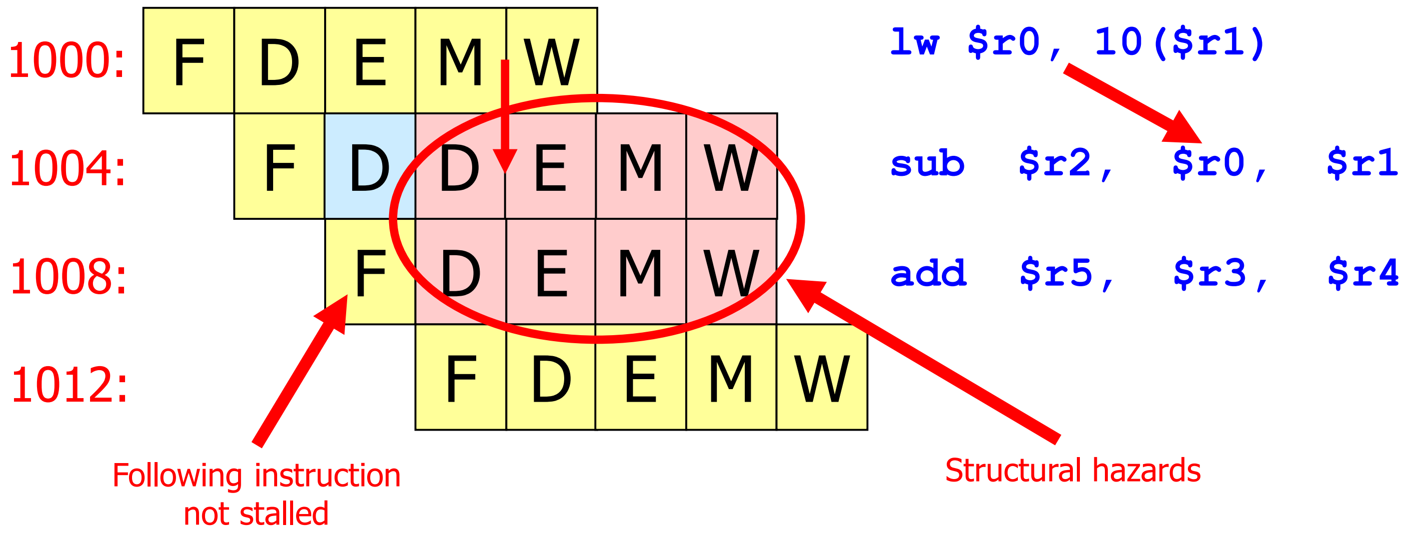


Used forwarding paths
(if any were missing, stalls would have been inserted)



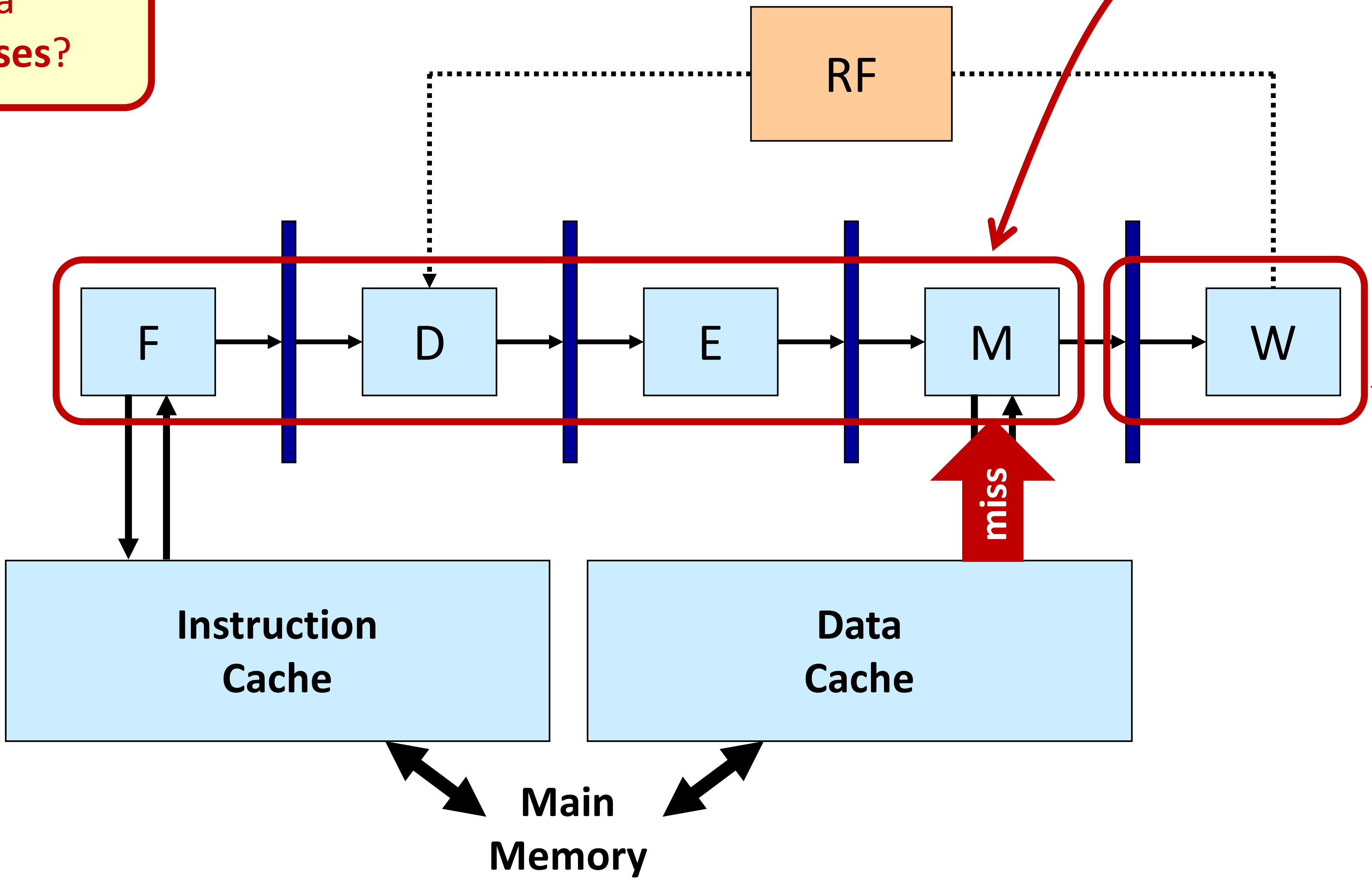
Structural Hazards

- A **structural hazard** happens when different instructions compete for the same resource (e.g., pipeline stage)
- Structural hazards **cannot happen** in our pipeline
- If we did not stall also instructions following one missing an operand, we could have structural hazards



Structural Hazards

What if a **caches misses?**

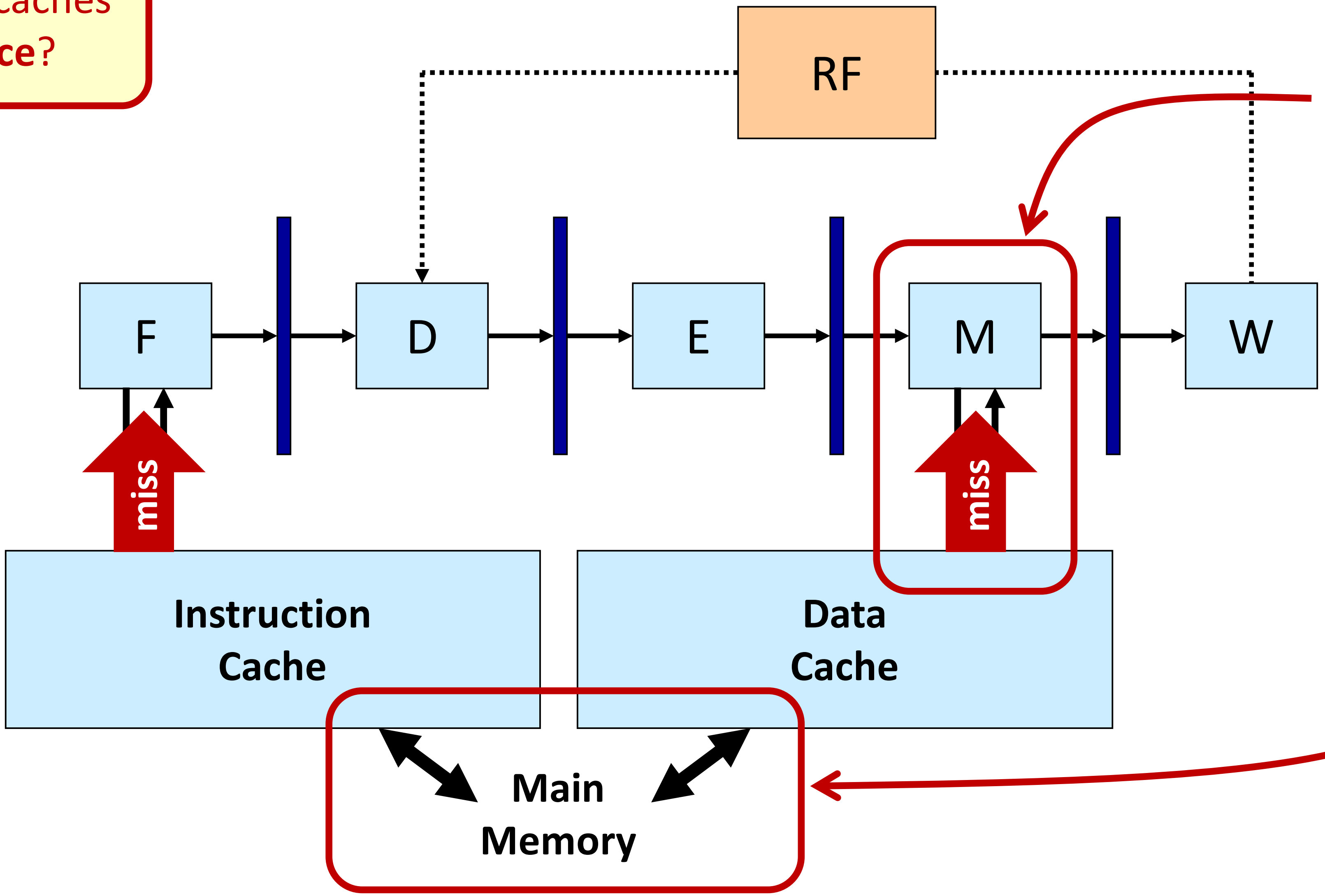


Stall the failing stage and all preceding ones

Let **downstream** stages **advance**

Structural Hazards

What if both caches miss **at once**?

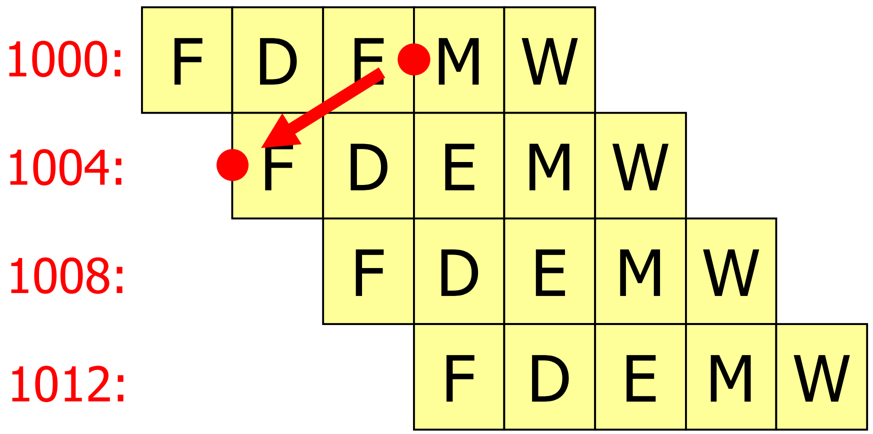


We give **priority** to the data cache (**F** is stalled, anyway…)

A **structural hazard** on main memory

Control Hazards

time (cycles)



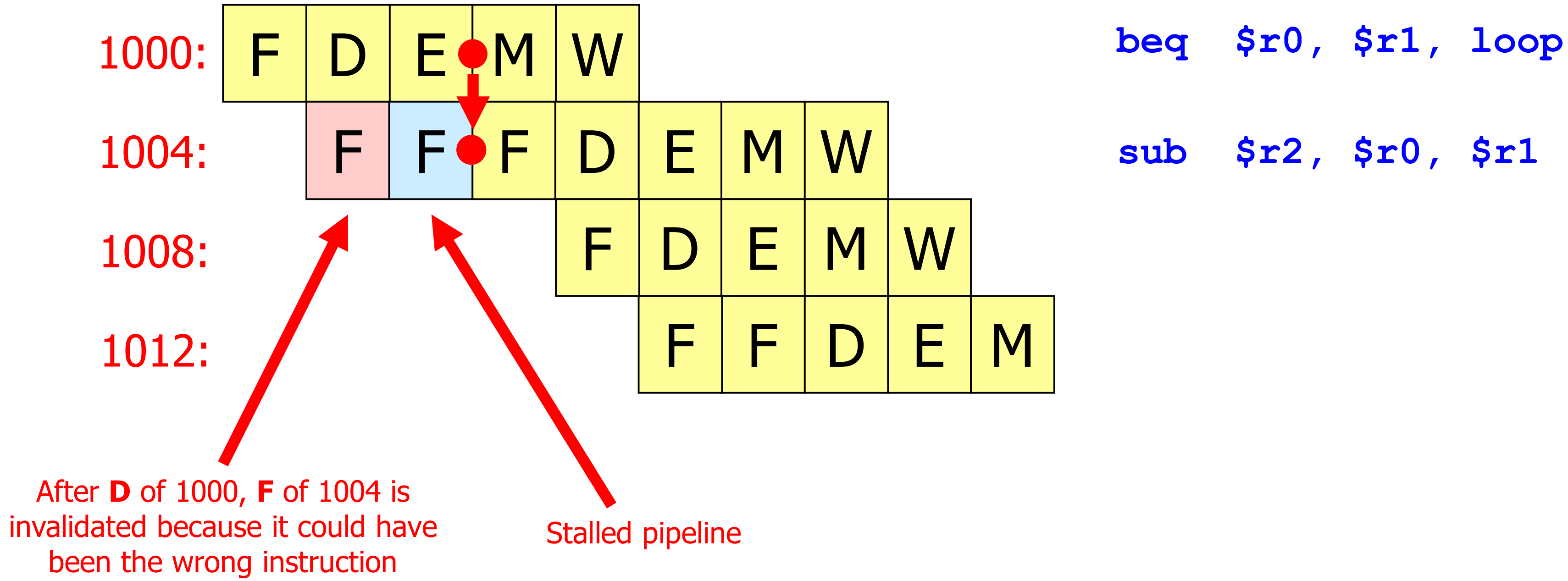

```
beq  $r0, $r1, loop

sub  $r2, $r0, $r1
```

Causality violation!

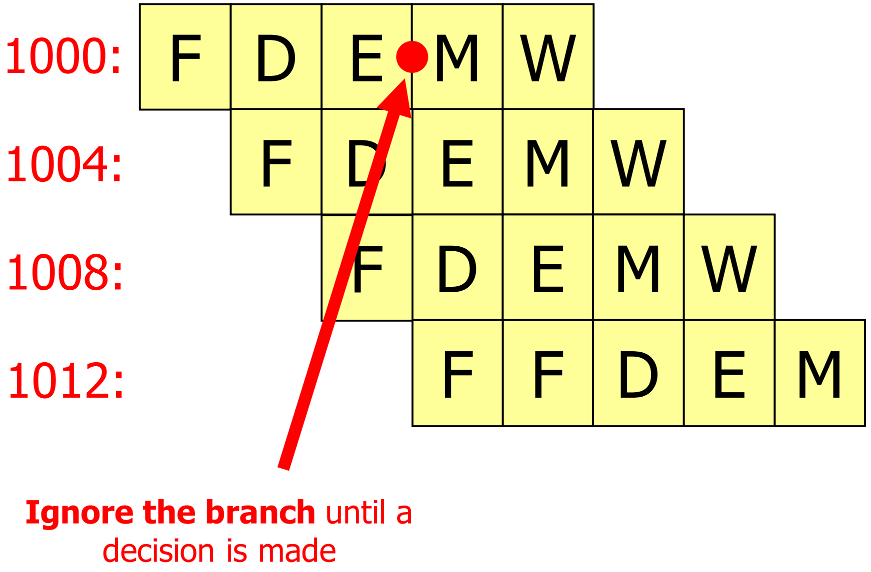
We fetch an instruction before we know which one!

Control Hazards Solved by Stalling the Pipeline



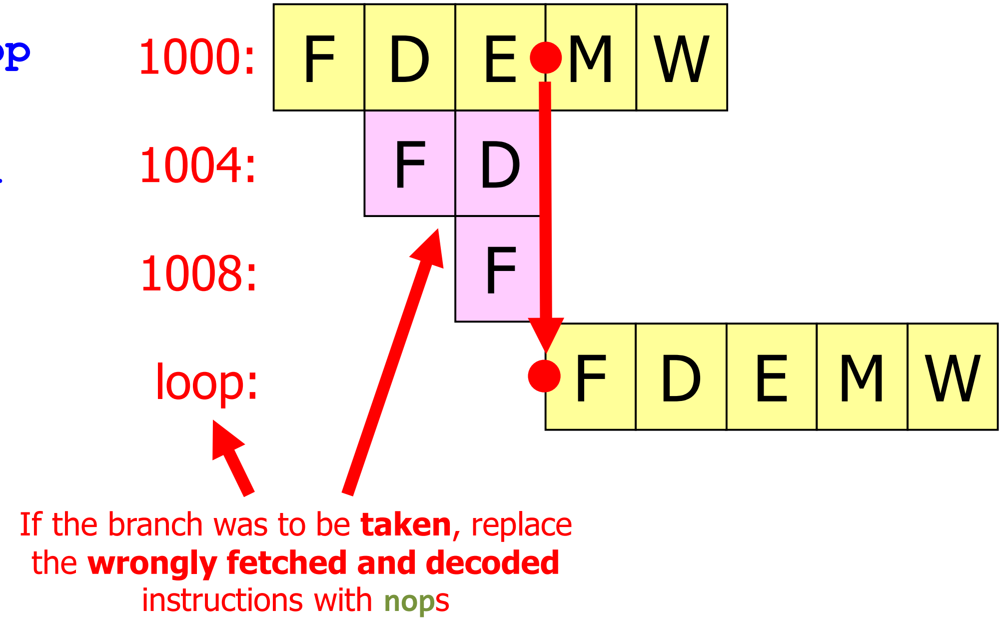
- Similarly to the way we solve data hazards, we can **stall the pipeline** (**F**), once it is discovered, after **D**, that an instruction was a branch, and this **until the branch is resolved**
- If, for instance, the correct address of the next instruction is known at the end of the **E** stage, **2 cycles are lost every branch**

Fetching and Decoding Do Not Do Any Damage




```
beq   $r0, $r1, loop

sub   $r2, $r0, $r1
```



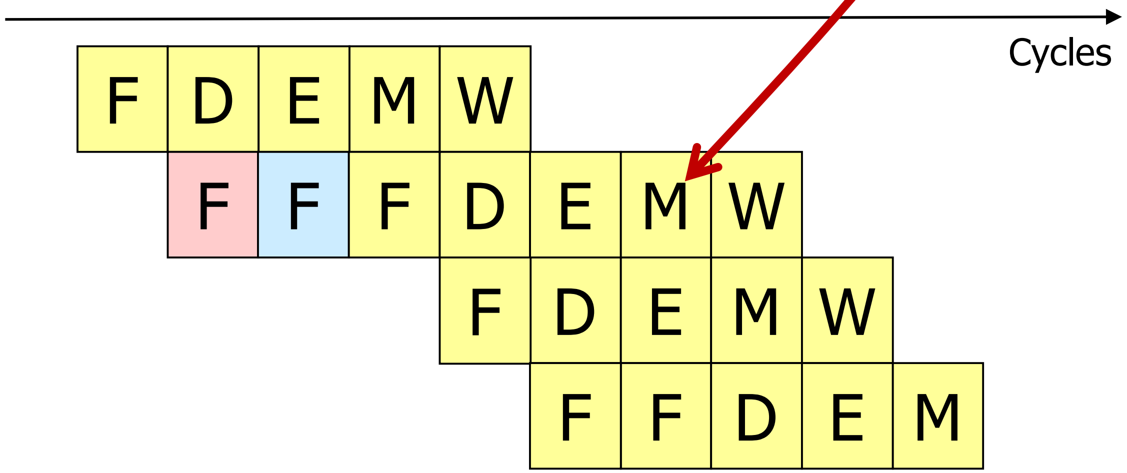
- Fetching or decoding a wrong instruction does not create any problem, provided that the instruction is not also **executed**
- If the outcome of a branch is known at the end of the **E** stage, we can **wait** until then to **conditionally kill** the following two instructions in the pipeline **if the branch happens to be taken**
- Now **2 cycles are lost only for taken branches** and **none is lost for nontaken ones**

Another Solution?

This solution needs some **hardware**

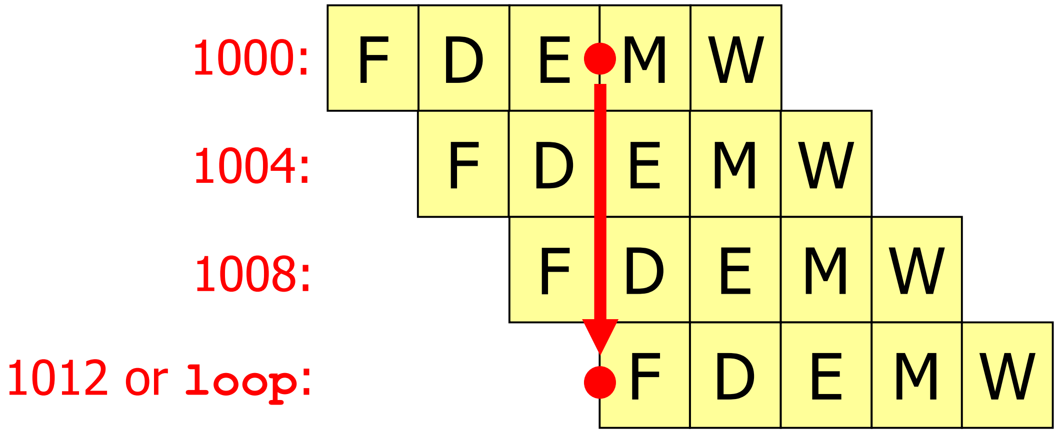
This solution needs a good **compiler**

```
1: beq    $r0, $r1, loop
2: sub    $r2, $r0, $r1
```




```
1: beq    $r0, $r1, loop
2: nop
3: nop
4: sub    $r2, $r0, $r1
```

Control Hazards Solved by Defining Delay Slots




```
beq  $r0, $r1, loop
sub  $r2, $r0, $r1
add  $r5, $r3, $r4
???
```

Delay Slots

- Alternatively, we can **modify the definition of the architecture** and decide that **the two instructions following a branch are executed in any case** (branch taken or not) as if they were before
- These instructions after the branches are called **delay slots**
- Note that code becomes **counterintuitive!!!**
- MIPS did it as some others, but quite rare in current architectures

Use of Delay Slots

- A simple way of using delay slots is to use them for **nop**'s—but then it is not better than stalling the pipeline
- A better idea is to put there instructions which precede the branch and on which the branch has **no dependence**
- Suppose an architecture with two delay slots:

```
1000:  sub  $r2, $r0, $r7
1004:  mul  $r1, $r6, $r7
1008:  add  $r5, $r3, $r4
1012:  beq  $r0, $r1, loop
1016:  nop                   } Delay
1020:  nop                   } slots
1024:  lw   $r8, 12($r9)
```

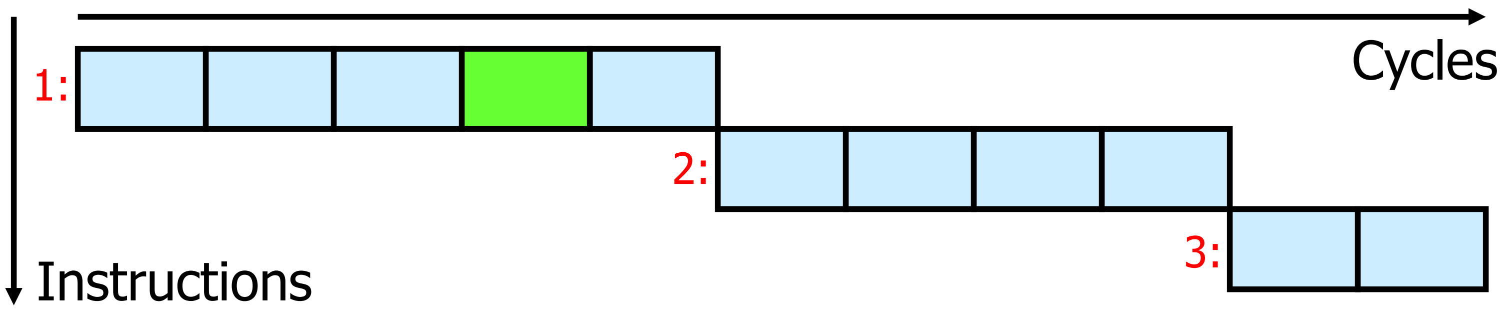
```
1000:  mul  $r1, $r6, $r7
                 RAW, cannot be moved
1004:  beq  $r0, $r1, loop
1008:  sub  $r2, $r0, $r7    } Delay
1012:  add  $r5, $r3, $r4    } slots
1016:  lw   $r8, 12($r9)
```

Branch Prediction

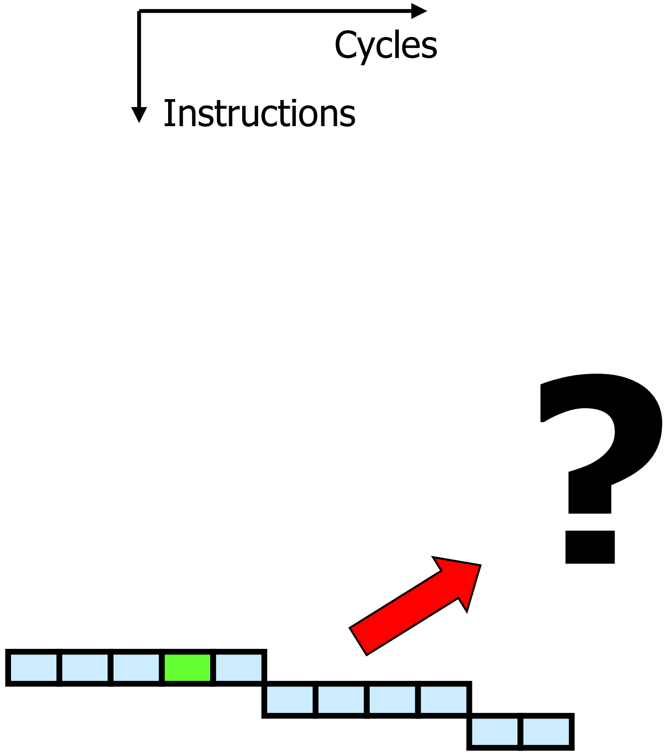
- A better strategy is to **guess the branch outcome** and fetch the corresponding instruction (either the next instruction or the branch destination, but not necessarily the former)
  - If the guess is correct, **no cycle is lost**
  - If the guess is wrong, what has been fetched and decoded is thrown away ("**squashed**")
- Branch predictors of modern processors are extremely sophisticated: dynamic predictors **learn from previous executions** of a branch…
- Complex predictors can be **correct up to 95-99%** of the time
- The quality of branch predictors has made architectures with delay slots extremely rare

Starting Point (Programmer Model)

- Sequential multicycle processor

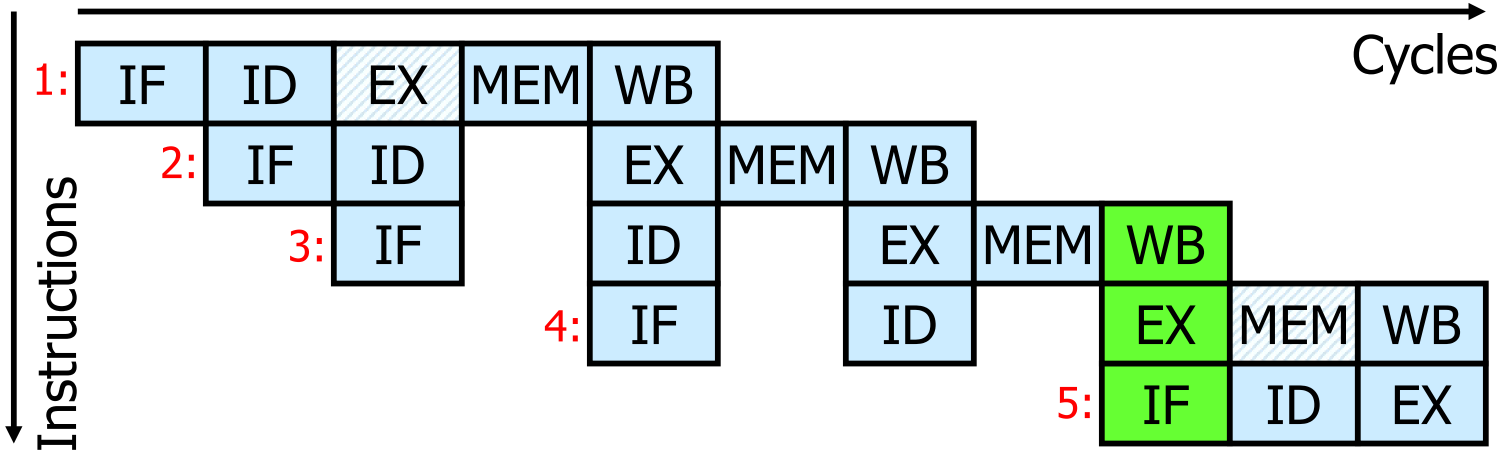


Instruction Level Parallelism?



Standard

First Step: Pipelining



- Simplest form of **Instruction Level Parallelism** (ILP): several instructions are now executed at once

Three Types of Hazards Hinder Pipelining

- **Data Hazards** (= data dependences)
  - *Solutions:*
    - Forwarding paths, wherever possible
    - Stalls, in all other cases
- **Control Hazards** (= jumps and branches)
  - *Solutions:*
    - Delay slots, if the architecture allows it
    - Branch prediction, to try to do the right thing
    - Stalls, if not
- **Structural Hazards** (= conflicting need for a resource)
  - *Solutions:*
    - Rigid pipelines which cannot have structural hazards by construction
    - Stalls, otherwise

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Sections 4.6-4.9**