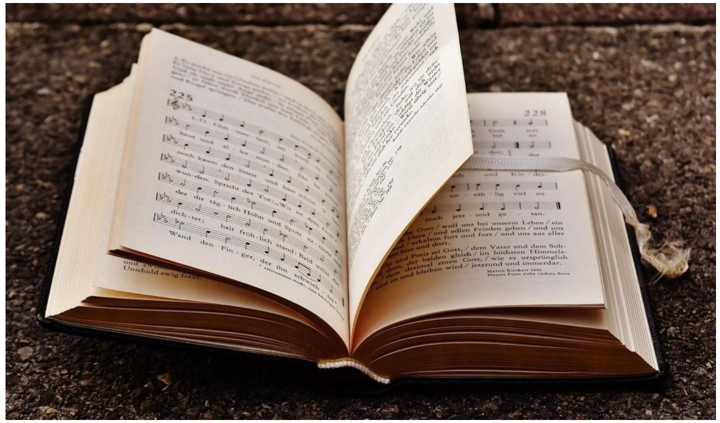
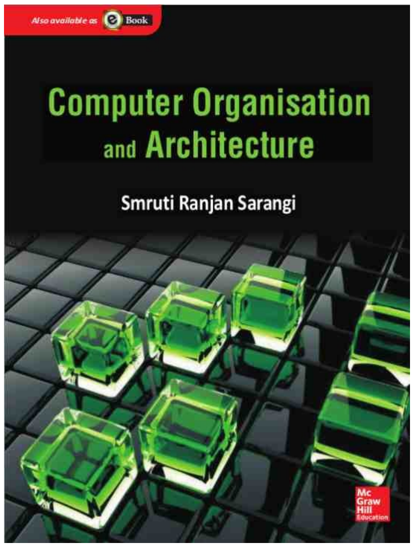


Chapter 2:
ut-of-Order Pipelines

Background Required to Understand this Chapter

- Assembly Languages
- Basic Processor Design
- Basic Pipeline Design

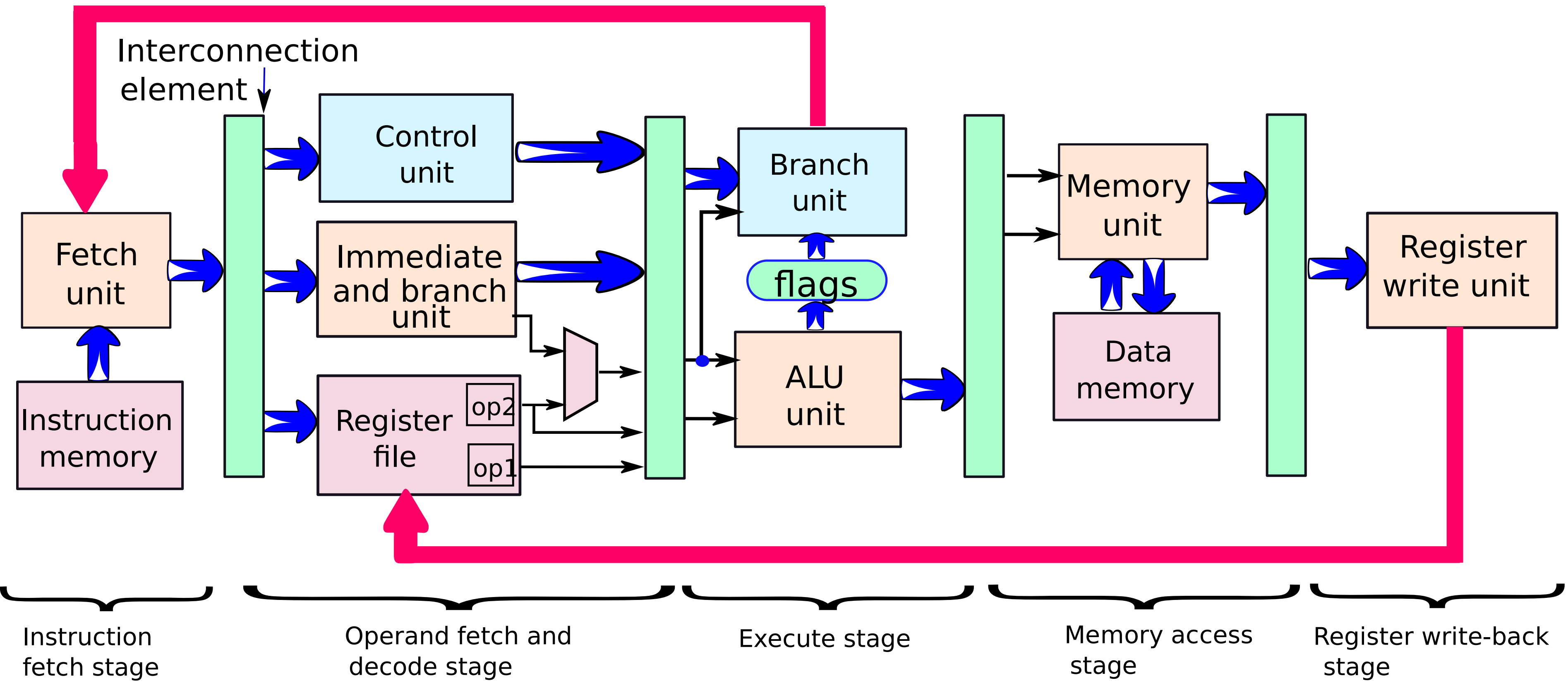


http://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

Outline

1. In-Order Pipelines
2. Performance Considerations
3. Out-of-Order Pipelines

A Simplified Diagram of a Processor with 5 Stages



[IF] Instruction fetch
[OF] Operand fetch and decode
[EX] Execute

[MA] Memory access
[RW] Register write-back

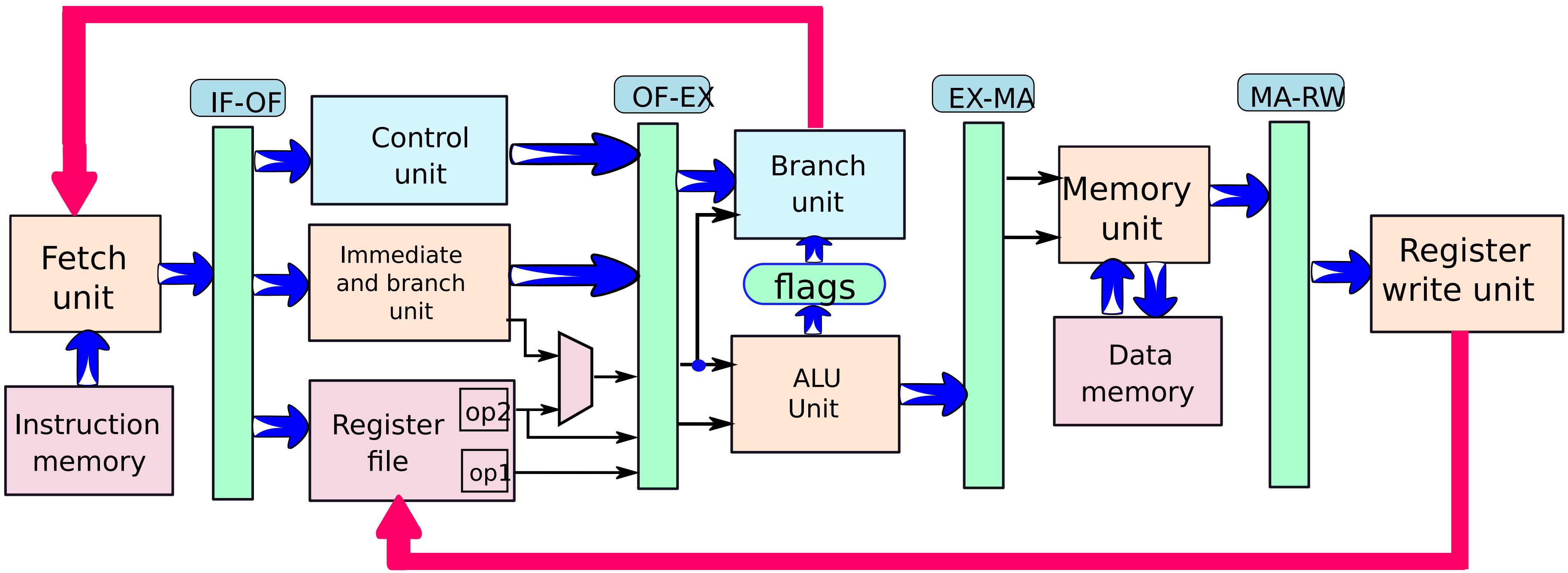
Pipelines

For more efficiency, we can pipeline the design. This will eliminate idleness in the processor.

In-order Pipelines
Instructions enter the pipeline in program order

5-Stage Pipeline

Pipelined Version of the Processor



Note the positions of the pipeline latches.

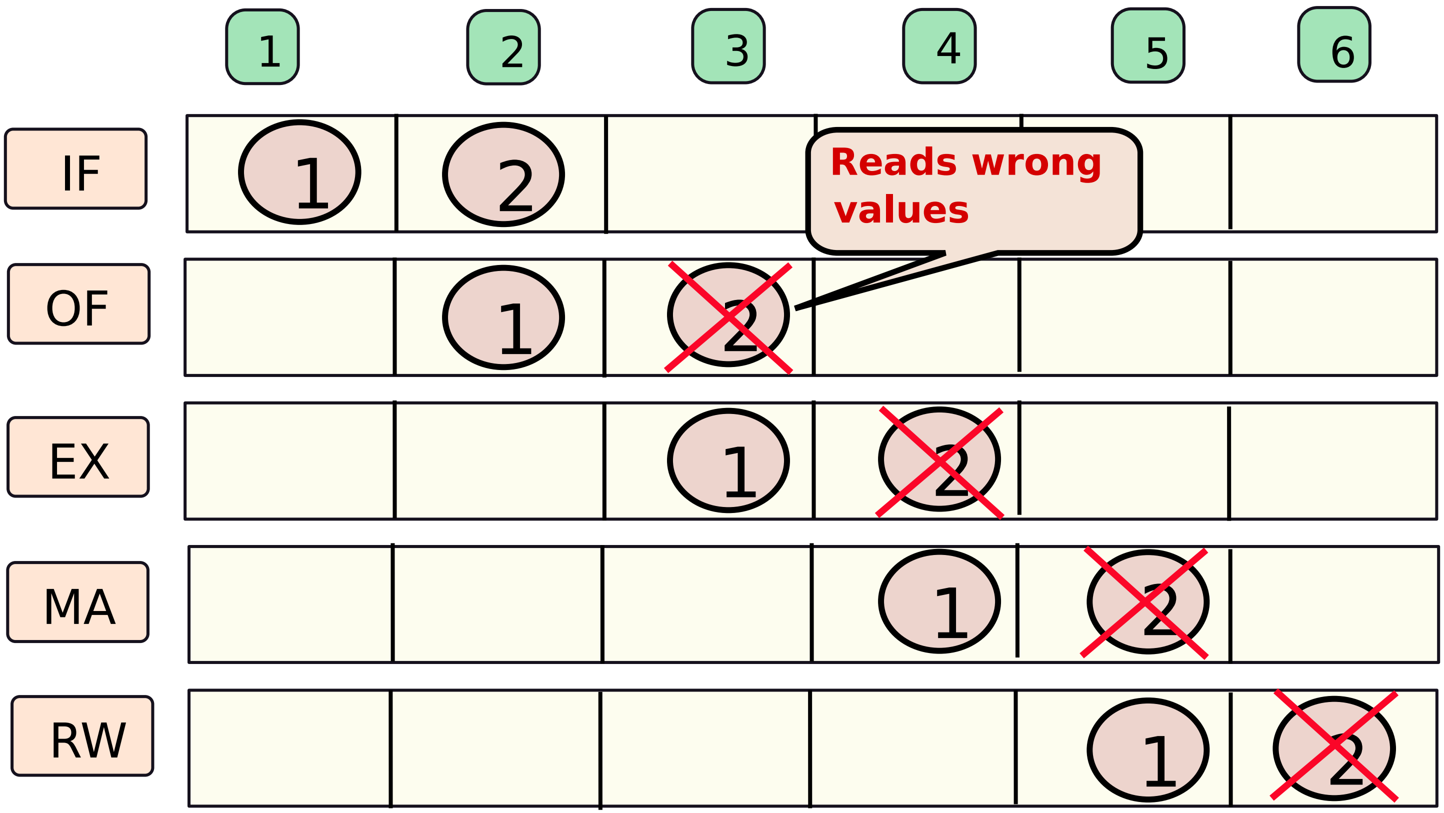
Problems with In-order Pipelines

Hazards

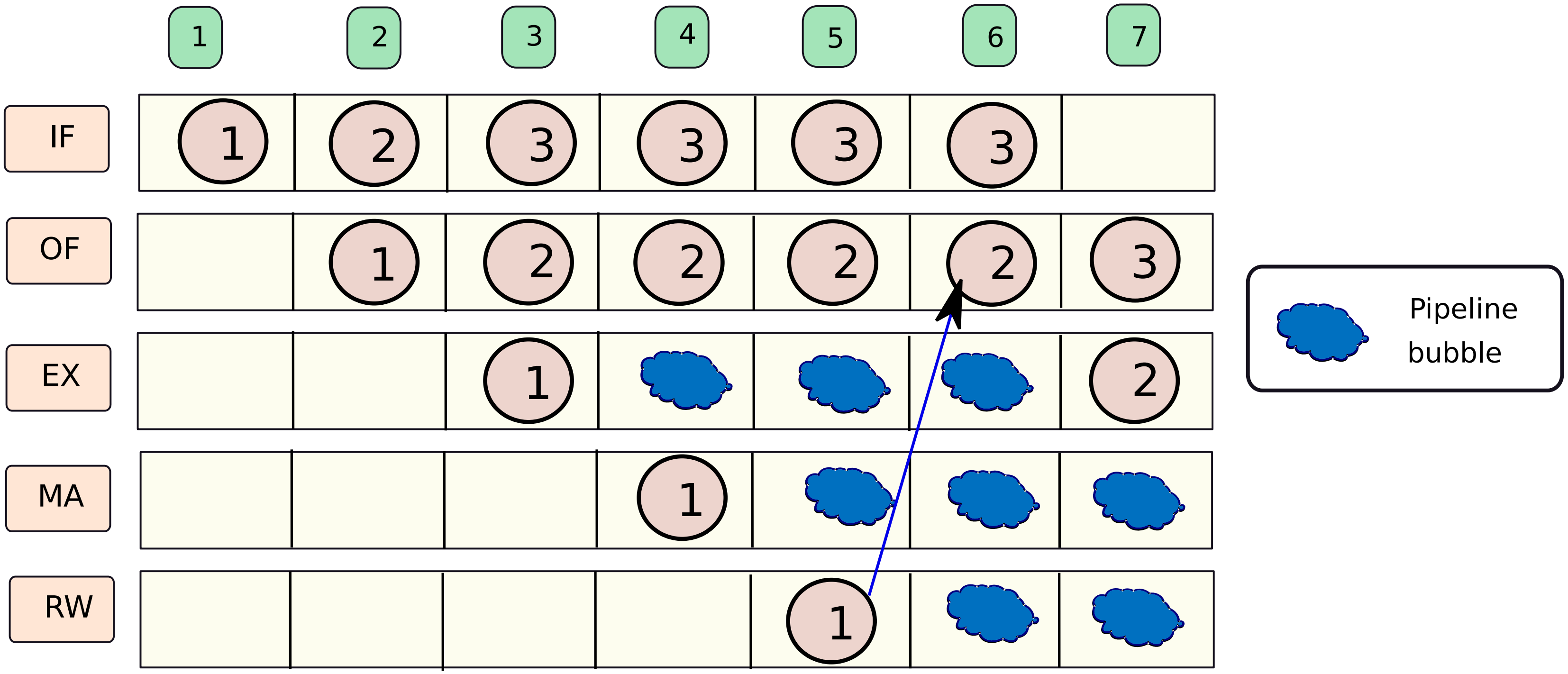
- Structural Hazards → Two instructions vie for the same resource (NOT possible in simple 5-stage pipelines)
- Data Hazards → An instruction stands to read or write the wrong data.
- Control Hazards → Instructions are fetched from the wrong path of the branch

Pipeline Diagrams

1	*add r1, r2, r3*
2	*add r4, r1, r3*



Pipeline Interlocks

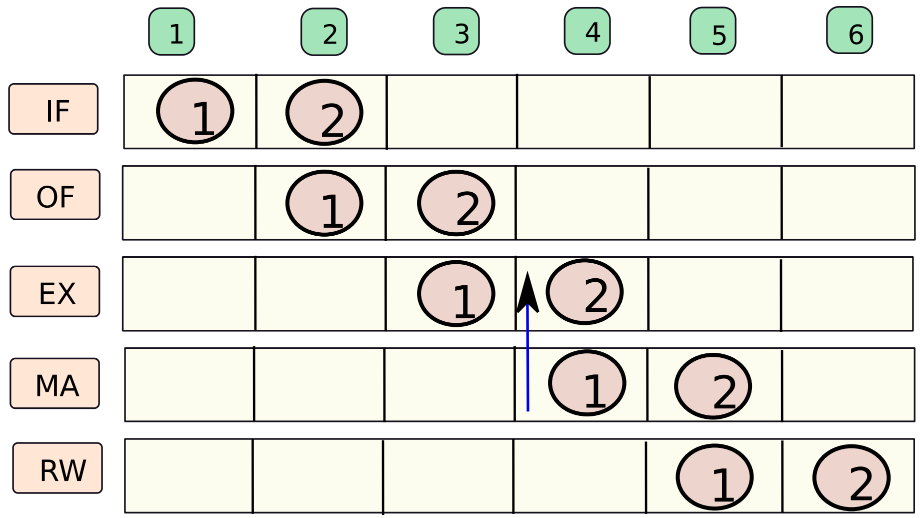


An interlock inserts a *nop* instruction (bubble) in the pipeline

Forwarding from the MA to the EX stage – No stalls

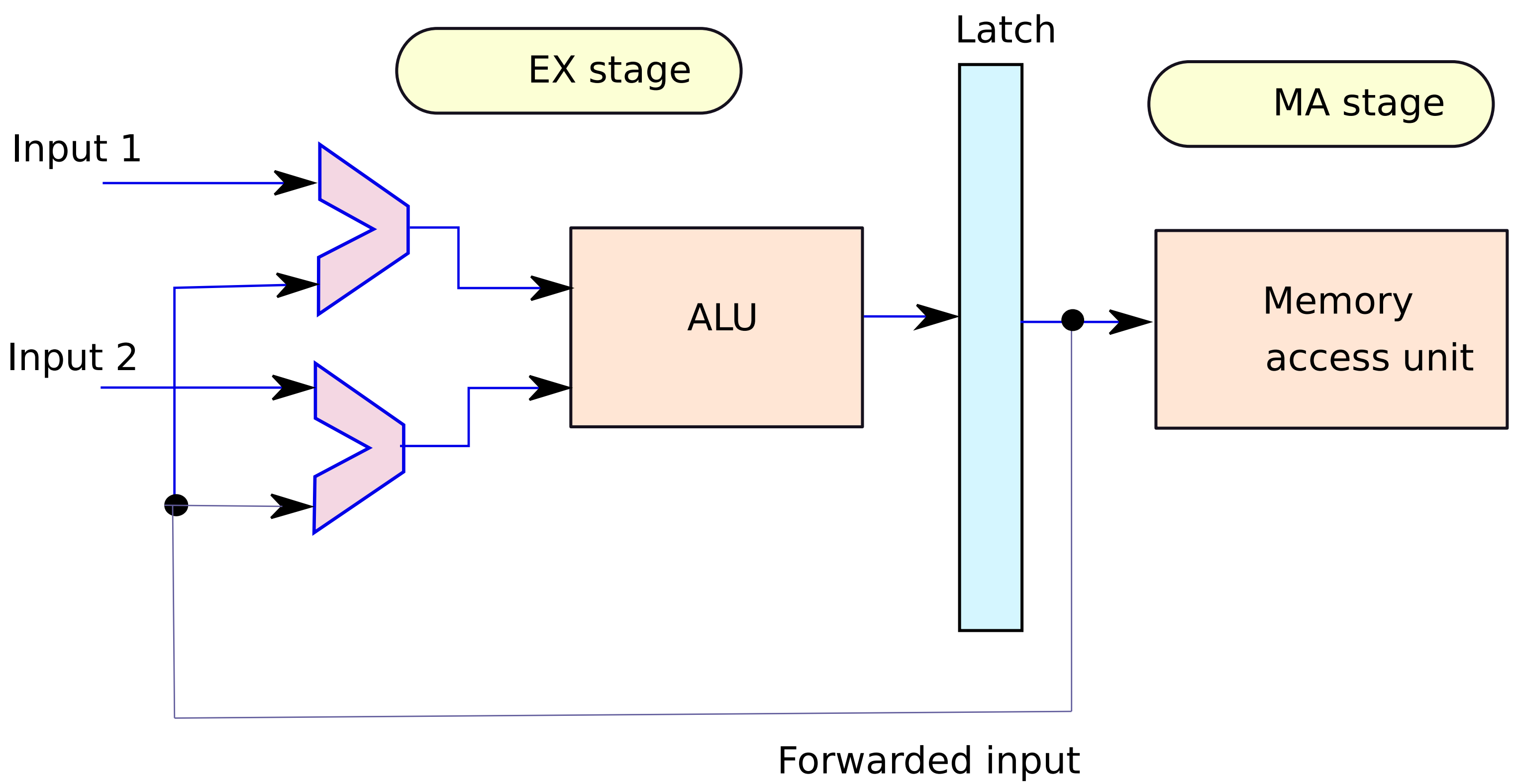
1: add r1, r2, r3
2: add r5, r1, r4

(a)



(b)

Forwarding Multiplexers



We need 4 Forwarding Paths

Forwarding Paths	Example
RW → MA	ld r1, 8[r2] st r1, 8[r3]

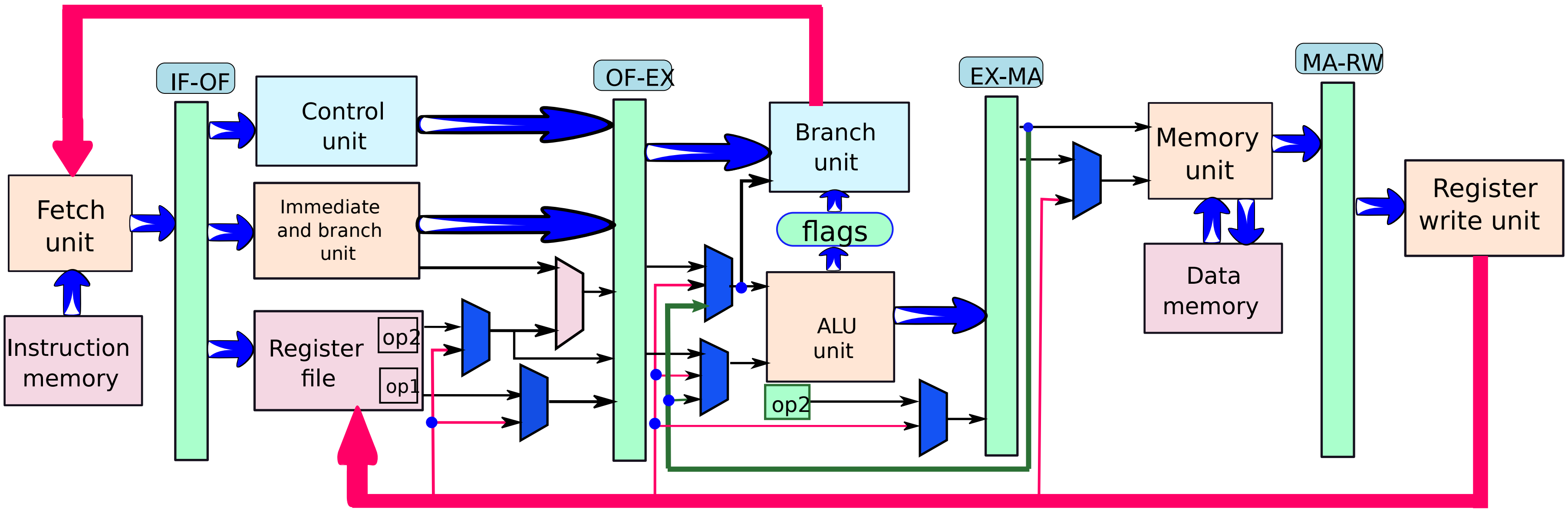
RW → EX	ld r1, 8[r2]
	sub r5, r6, r7
	add r3, r2, r1

RW → OF	ld r1, 8[r2]
	sub r5, r6, r7
	sub r8, r9, r10
	add r3, r2, r1

MA → EX	add r1, r2, r3 sub r5, r1, r4

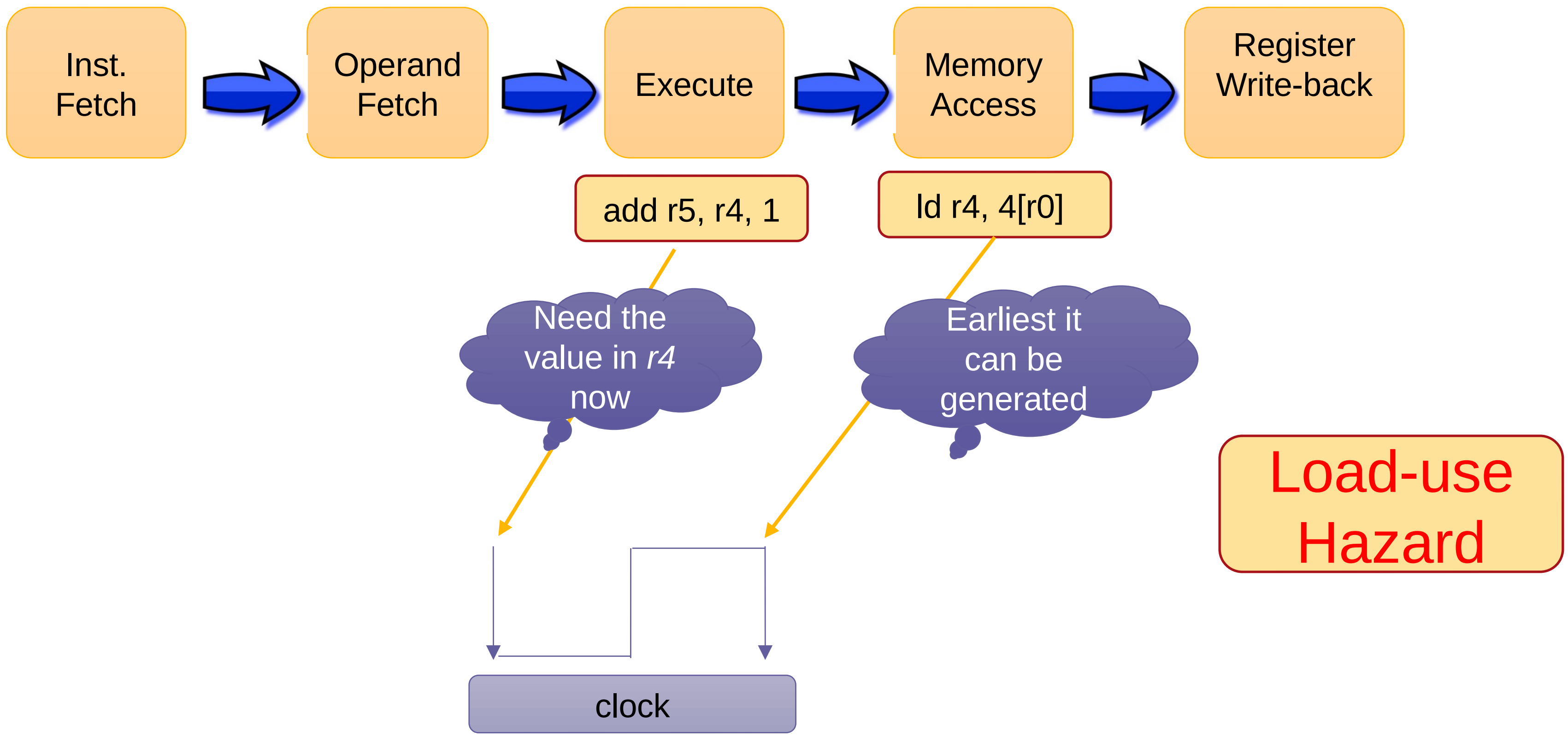
Forward as late as possible

Final View of the Pipelined Processor with Forwarding Multiplexers

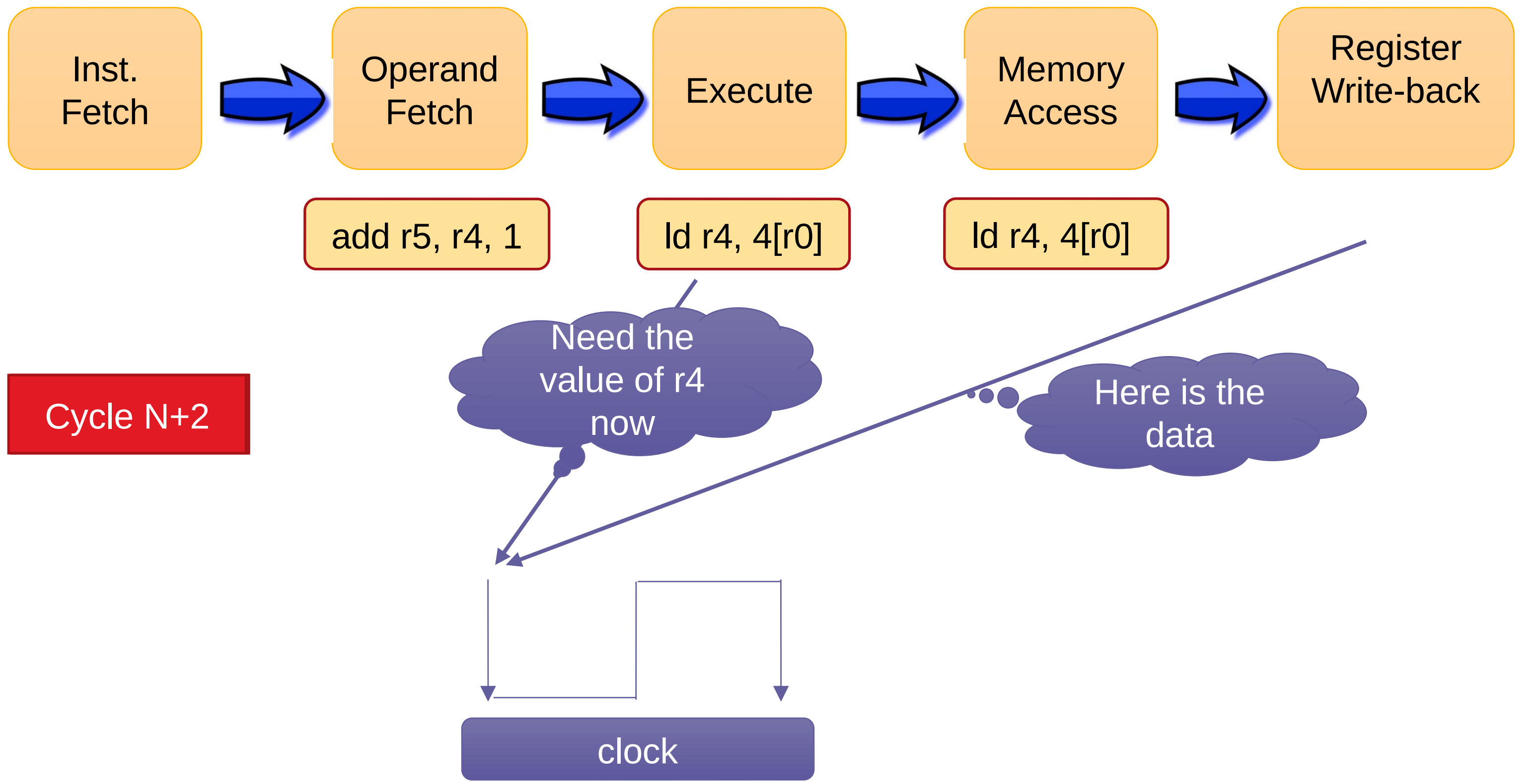


We add 6 forwarding multiplexers

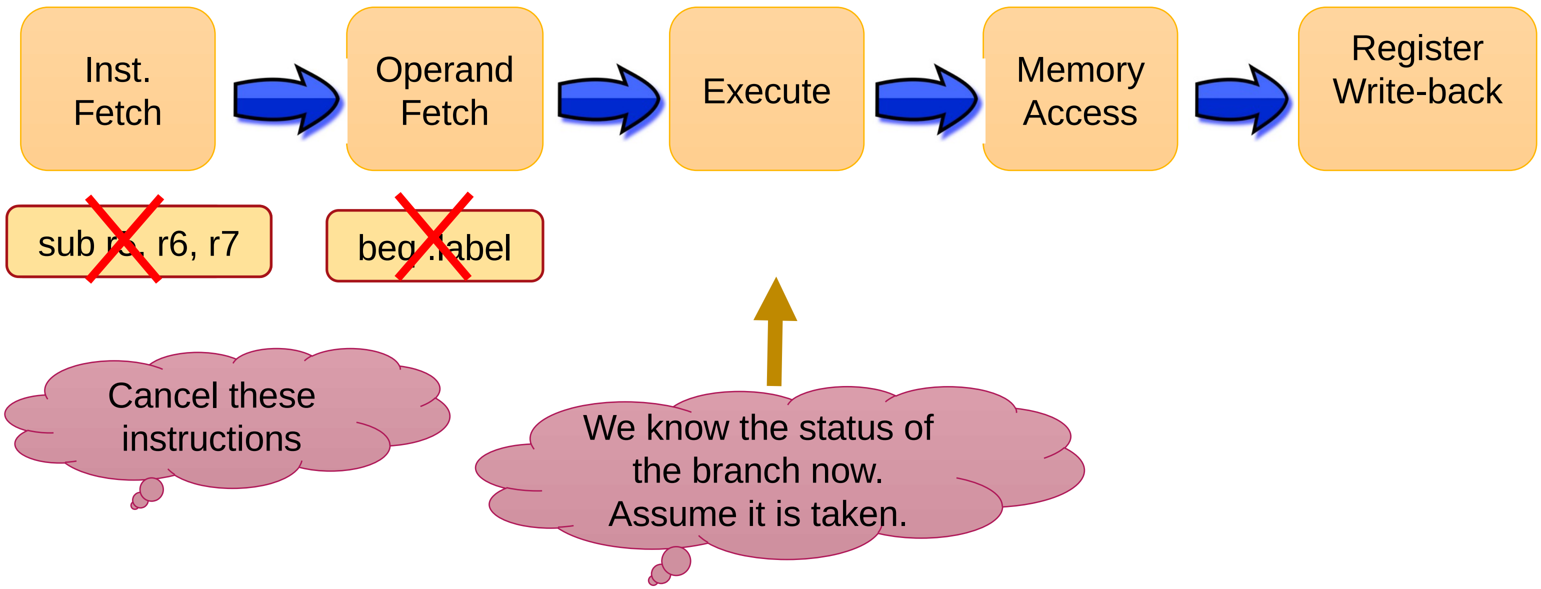
Data Hazards in In-order Pipelines with Forwarding



Solution: Stall the Pipeline



Control Hazards



Two instruction slots are wasted

Outline

1. In-Order Pipelines
2. Performance Considerations
3. Out-of-Order Pipelines

Performance Equation - I

Is Computer A faster that Computer B

- **Wrong Answers:**
  - More is the clock speed, faster is the computer
  - More is the RAM, faster is the computer

What does it mean for computer A to be faster than computer B

Short Answer: **NOTHING**

Performance is always with respect to a program. You can say that a certain program runs faster on computer A as compared to computer B.

Performance Equation - II

= * *

=

= (assume just 1 program)

- IPC is the number of instructions per cycle
- Let us loosely refer to the reciprocal of the time per program as the **performance**

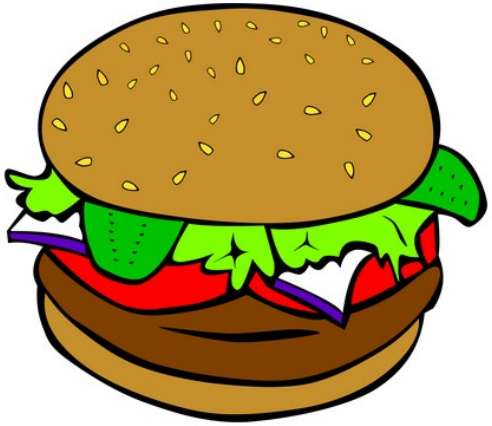
So, what does performance depend on …

#instructions in the program

- Depends on the compiler

Frequency

- Depends on the transistor technology and the architecture
  - If we have more pipeline stages, then the time to traverse each stage reduces roughly proportionally
  - Given that each stage needs to be processed in one clock cycle, smaller the stage, higher the frequency
  - To increase the frequency, we simply need to increase the number of pipeline stages



IPC

- Depends on the architecture and the compiler
- A large part of this book is devoted to this aspect.

How to improve performance?

There are **3** factors:

- IPC, #instructions, and frequency
- #instructions is dependent on the compiler not on the architecture

Let us look at IPC and frequency

IPC

- What is the IPC of an in-order pipeline?

1 if there are no stalls, otherwise < 1

Methods to increase IPC

- Forwarding
- Having more not-taken branches in the code
- Faster instruction and data memories

What about frequency?

What is frequency dependent on …

Frequency = 1 / clock period

Clock Period:

- 1 pipeline stage is expected to take 1 clock cycle
- Clock period = maximum latency of the pipeline stages

How to reduce the clock period?

- Make each stage of the pipeline smaller by increasing the number of pipeline stages
- Use faster transistors

Limits to Increasing Frequency

Assume that we have the fastest possible transistors

Can we increase the frequency to 100 GHz?

Reasons

Limits to increasing frequency - II

What does it mean to have a very high frequency?

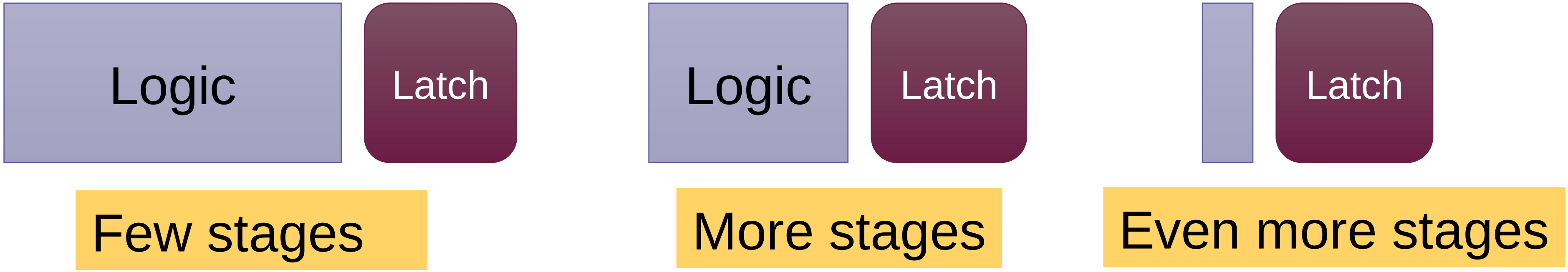
Before answering, keep these facts in mind:

1. **Thumb Rule**: $P \propto f^3$
2. **Thermo-dynamics**: T
3. We need to increase the number of pipeline stages → more hazards, more forwarding paths

P → power
f → frequency

T → Temperature

How many pipeline stages can we have?



- We are limited by the latch delay
- Even with an infinite number of stages, the minimum clock period will be equal to the latch delay

Pipeline Stages vs IPC

$$CPI = \frac{1}{IPC}$$

$$CPI = CPI_{ideal} + stall\_rate * stall\_penalty$$

- The stall rate will remain more or less constant per instruction with the number of pipeline stages
- The **stall penalty** (in terms of cycles) will however **increase**
- This will lead to a net increase in CPI and **loss** in IPC

Summary

As we increase the number of stages, the IPC goes down.

Summary: Why we cannot increase frequency by increasing the number of pipeline stages?

- Power
- Temperature
- Effect of the Latch Delay
- Stall penalties will increase

Since we cannot increase frequency …

Increase IPC

Increase IPC

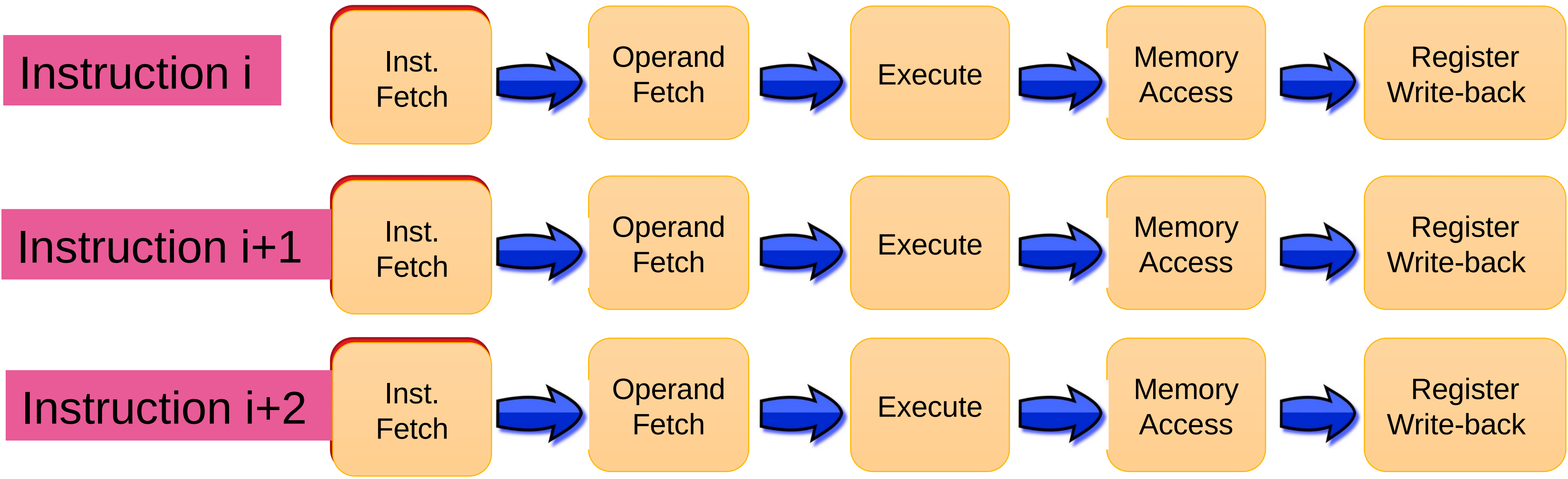
Issue more instructions per cycle

2, 4, or 8 instructions

Make it a **superscalar** processor A processor that can execute multiple instructions per cycle

In-order Superscalar Processor

Have multiple in-order pipelines.



In-order Superscalar Processor - II

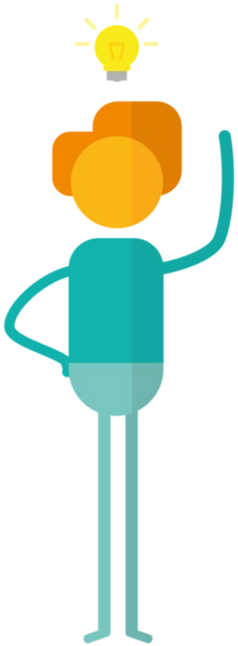
- There can be dependences between instructions
- Have $O(n^2)$ forwarding paths for an *n*-issue processor
- Complicated logic for detecting dependences, hazards, and forwarding
- Still might not be enough ...
- To get the peak IPC (= n) in an *n*-issue pipeline, we need to ensure that there are no stalls
- There will be no stalls if there are no taken branches, and no data dependences between instructions.
- Programs typically do not have such long sequences of instructions without dependences

Contents

Outline

1. In-Order Pipelines
2. Performance Considerations
3. Out-of-Order Pipelines

What to do ...



Don't follow program order

mov r1, 1
add r3, r1, r2
add r4, r3, r2
mov r5, 1
add r6, r5, 1
add r8, r7, r6

Too many dependences

Execute out of order

Execute on a 2-issue OOO processor

```
mov r1, 1           mov r5, 1
add r3, r1, r2      add r6, r5, 1
add r4, r3, r2      add r8, r7, r6
```

Execute 2 instructions in parallel

Continuation ...

	issue slot 1	issue slot 2
cycle 1	mov r1, 1	mov r5, 1
cycle 2		add r6, r5, 1
cycle 3	add r3, r1, r2	add r8, r7, r6
	add r4, r3, r2	

In Out-of-order (OOO) processors, the execution is not as per program order. It is as per the data dependence order the consumer is executed always after the producer.

Basic Principle of OOO Processors

Create a pool of instructions

Find instructions that are mutually independent and have all their operands ready

Execute them out-of-order

ILP

Instruction level parallelism

The number of ready and independent instructions we can simultaneously execute.

Revisit the Example

Pool of Instructions

mov r1, 1 | add r3, r1, r2 | add r4, r3, r2 | mov r5, 1 | add r6, r5, 1 | add r8, r7, r6

Issue ready and mutually independent instructions

Pool of Instructions: Instruction Window

- Needs to be large enough such that the requisite number of mutually independent instructions can be found.
- Typical instruction window sizes: 64 to 128
- How do we create a large pool of instructions in a program with branches? We need to be sure that **all the instructions** are on the correct path

```
for (i = 1; i < m; i++) {
        for (j = 1; j < i; j ++ ) {
                if (j %2 == 0) continue;
                ....
        }
}
```

Example

Problems with creating an Instruction Pool

Typically 1 in 5 instructions is a branch

Predict the directions of the branches, and their targets

Motivation for Branch Prediction

1 We need high IPC

2 This means that we need a large instruction window

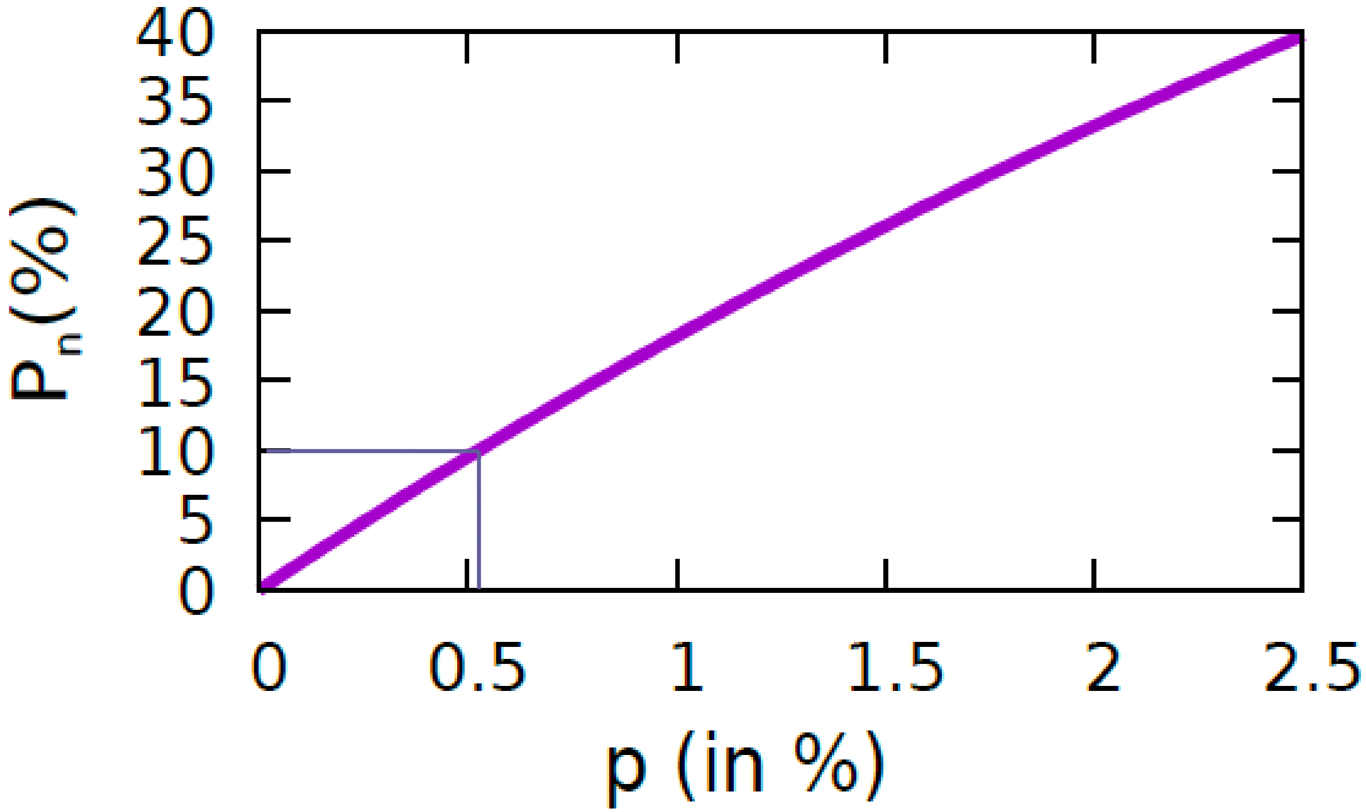
3 It will have a lot of branches.

4 We need to predic **ALL** the branches correctly.

The Maths of Branch Prediction

Number of instructions	***n***
Number of branches	**n/5**
Probability of predicting any given branch incorrectly	***p***
Probability of predicting **ALL the branches** correctly	
Probability of making at least a single mistake (branch misprediction) in a pool of *n* instructions.	

For (n=100) : A plot of P_n vs p



If P_n = 10%, p has to be as low as 0.5% !!!

If we need a large instruction window, we need a very accurate branch predictor. The accuracy of the branch predictor limits the size of the instruction window.

Nature of Dependences

Dependences between Instructions

Program Order Dependence

```
mov r1, 1
mov r2, 2
```

- One instruction appears after the other in program order

The program order is the order of instructions that is perceived by a single cycle in-order processor executing the program.

Data Dependences

RAW ▭ Read after Write Dependence (True dependence)

```
mov r1, 1
add r3, r1, r2
```

- It is a producer-consumer dependence.
- The earlier instruction produces a value, and the later instruction reads it.

Data Dependences - II

WAW ▭ Write after Write Dependence (Output dependence)

```
mov r1, 1
add r1, r4, r2
```

- Two instructions write to the same location
- The later instruction needs to take effect after the former

Data Dependences - III

WAR □ Write after Read Dependence (Anti dependence)

```
add r1, r2, r3
add r2, r5, r6
```

- Earlier instruction reads, later instruction writes
- The later instruction needs to execute after the earlier instruction has read its values

Control Dependences

```
beq .label
.....
.label
        add r1, r2, r3
```

- The *add* instruction is control dependent on the branch(*beq*) instruction
- If the branch is taken then only the *add* instruction will execute, not otherwise

Basic Results

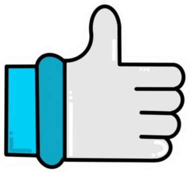
In-order processors respect all program order dependences. Thus, they automatically respect all data and control dependences.

OOO processors respect only data and control dependences.

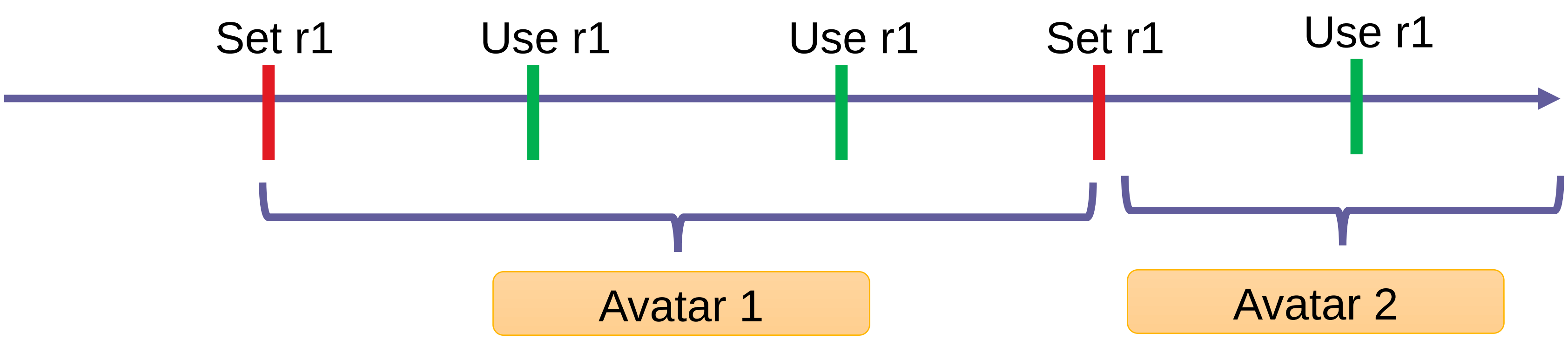
Can output and anti dependences be removed?

```
mov r1, 1
add r5, r6, r7
add r1, r4, r2
add r8, r9, r10
```

```
add r1, r2, r3
add r5, r6, r7
add r2, r5, r6
add r8, r9, r10
```



- Don’t you think that these dependences are there because we have a finite number of registers.
- What if we had an infinite number of registers?



Solution: Assume infinite number of physical registers

Architectural register → Physical register

Format in this example: rx is mapped to px<avatar number>

```
mov r1, 1
add  r1, r2, r3
add  r4, r1, 1
mov r2, 5
add r6, r2, r8
mov r1, 8
add r9, r1, r2
```

Code with architectural registers

```
mov p11, 1
add  p12, p2, p3
add  p41, p12, 1
mov p21, 5
add p61, p21, p8
mov p13, 8
add p91, p13, p21
```

Code with physical registers

Renaming

Program with real (architectural) registers

Program with physical registers

RAW dependences

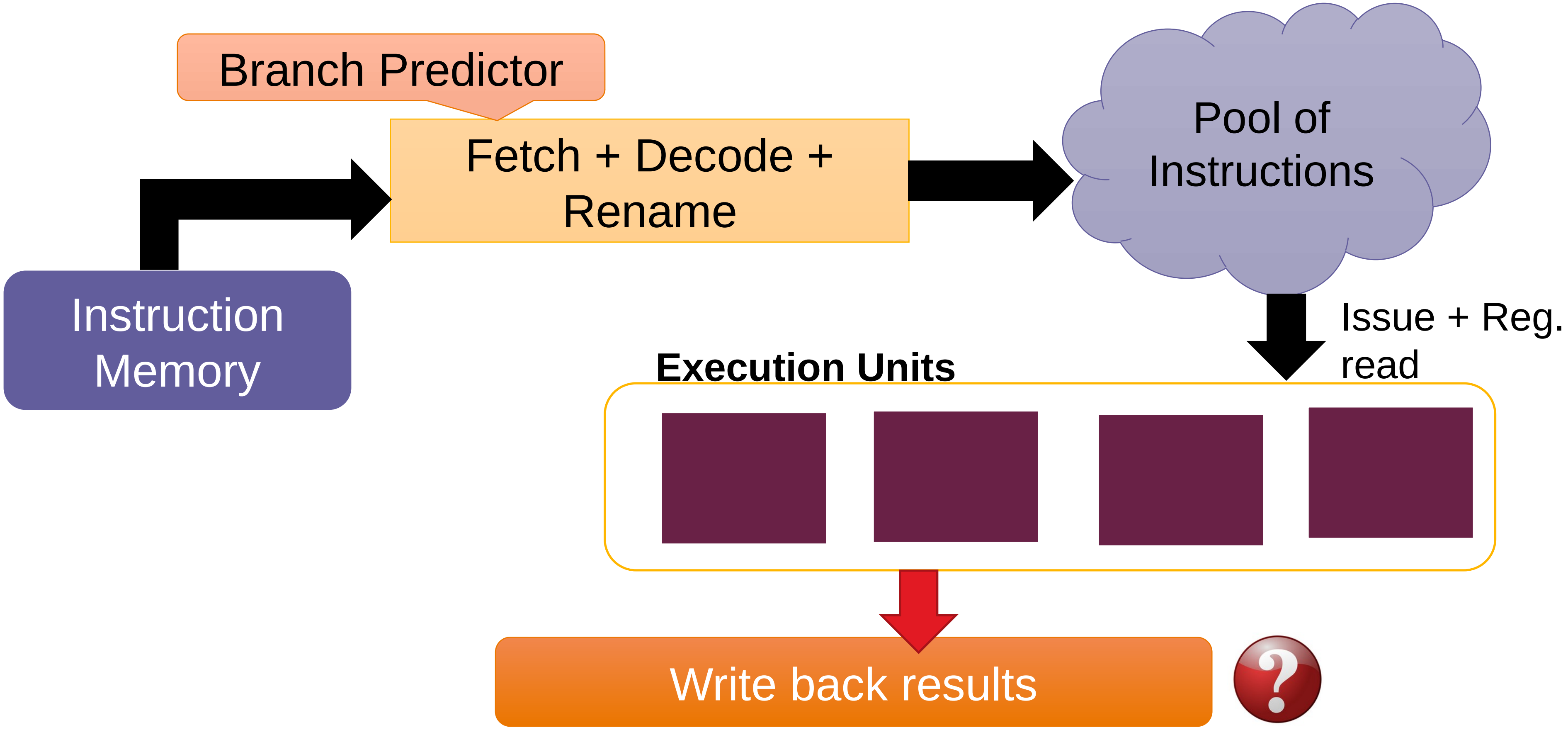
WAR dependences

WAW dependences

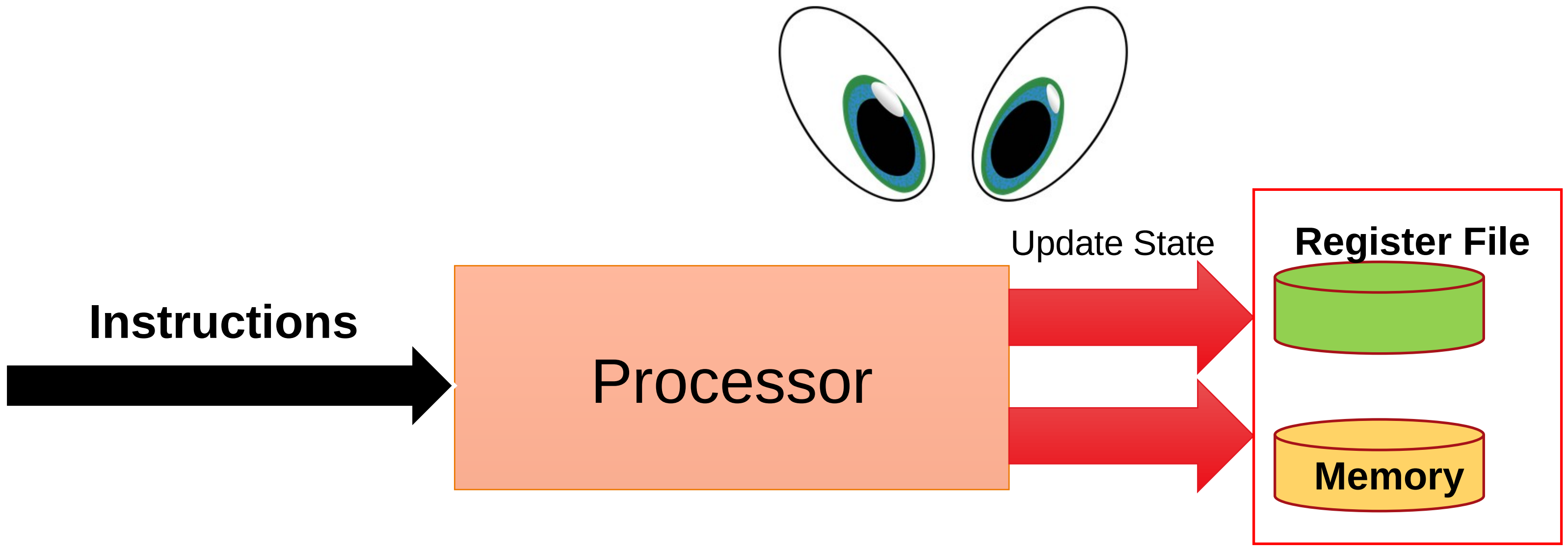
RAW dependences

Higher instruction level parallelism (ILP)

Where are we now ...



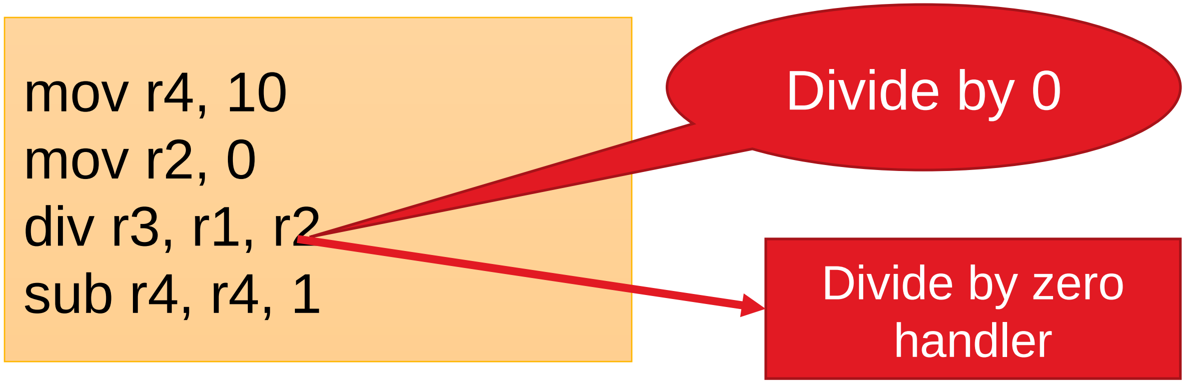
Issue with Write-back



To an outsider should it matter if the processor is in-order or OOO

NO

Assume that there is an exception or interrupt



Languages like C or Java have dedicated functions that are called if there is a divide-by-zero in the code.

The question is:

- What if the sub instruction has executed when we enter the exception handler?
- An in-order processor will never do this.

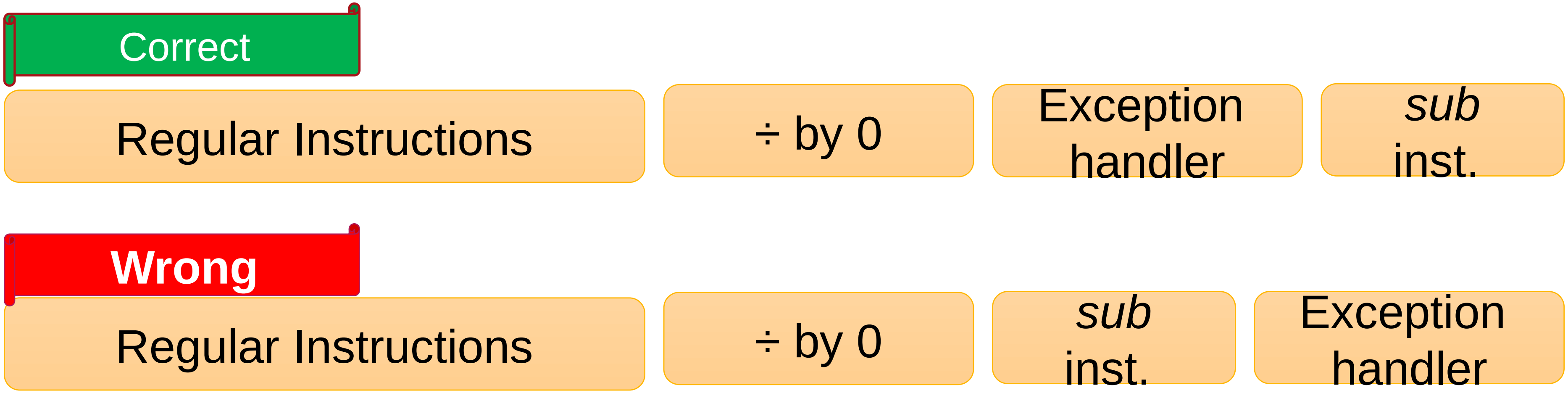
Precise Exceptions

Flow of actions

Regular Instructions	÷ by 0	Exception handler	sub inst.

- Assume that the exception handler decides to do nothing and return back
- After this the *sub* instruction should be executed
- This is exactly what will happen in an in-order processor
- In an OOO processor there is a possibility that the *sub* inst. can execute out of order
- The outsider (exception handler) will see a different view as compared to the view it will see with an in-order processor.

Precise Exceptions - II



To an external observer

- The execution should always be correct and as per program order
- Even in the presence of interrupts and exceptions

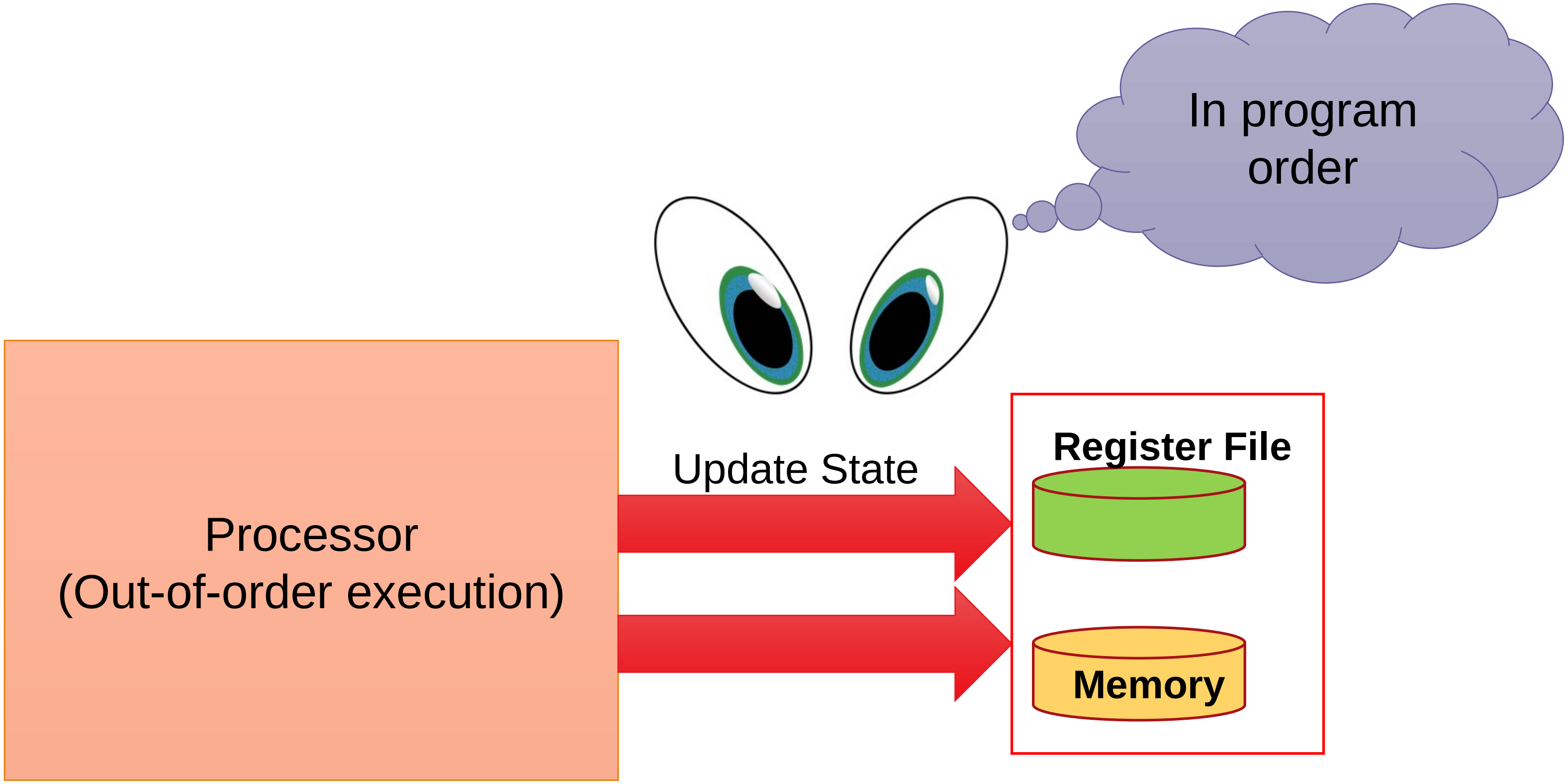
Precise Exceptions - III

- We thus need precise exceptions
- Assume that the dynamic instructions in a program (ordered in program order) are: ins_1, ins_2, ins_3 ... ins_n
- Assume that the processor starts the exception/interrupt handler after it has just finished writing the results of instruction: ins_k

ins_1 | *ins_2* | *ins_3* | • • • | *ins_k* | *Exception* | *ins_{k+1}* | • • •

- Then instructions: ins_1 ... ins_k should have executed completely and written their results to the memory/register file
- AND, ins_{k+1} and later instructions should not appear to have started their execution at all
- Such an exception or interrupt is precise

Precise Exceptions in an OOO Processor



Conclusion

- der pipelines have a limited IPC because zards and branches
- Multi-issue in-order pipelines do not solve the problem. Reason: dependences
- (OOO). We need a large instruction window to find sufficient independent instr.
- tain a large instruction window, we need a very te branch predictor.
- ose additional ILP, we can remove WAR/WAR s. Finally, we need to have precise exceptions.

The End