COMS12200 Introduction to Computer Architecture

Simon Hollis (simon@cs.bris.ac.uk)

Summary from last time

We saw three useful guidelines when creating control flow for state machines:

1. MUXs are useful for selecting inputs
2. DEMUXs are useful for decoding outputs
3. OR gates are useful for combining states and producing feedback signals

CONTROLLING PROCESSING WITH INSTRUCTIONS

Recap: selecting the input

- Where do the *select* signals come from?
  - They could come from instructions
    - Instructions will dictate behaviour

  - They could come from feedback
    - Data values will dictate behaviour

Processor blocks

- A processor is made of a number of component blocks
- Each has a specific role
- Each is enabled or disabled, as seen fit when executing a particular instruction

Instructions and control

- Can we format the instructions to aid control signal selection?
- Many architectures do this by grouping common functionality in certain bits, e.g. below (and more in ISA lecture).

Instruction	Op-code	ALU active?	Shifter active?
Add	100	Y	N
Subtract	101	Y	N
Left shift	010	N	Y
Right shift	011	N	Y
Add and left shift 1	110	Y	Y

Instructions and ALUs

- How can we create a controllable ALU based on these instructions?

Instructions and machines

- So, the needed instructions determine what computational blocks the machine needs to include
- The particular blocks change based on the machine type being constructed
- There are many different approaches, which we will now explore

Topic 7: Different machine types

COMPUTING MACHINE TYPES AND TAXONOMY

Overview

- We’ll now look at the various common paradigms used for implementing a computing system.
  - We’ll look at how processors are structured and operates;
  - Later, we’ll look at the memory sub-system is configured and interacts with the processor execution.

Architectural paradigms

PROCESSOR ARCHITECTURES

Processor architectures

- Last lecture, we saw that programs are first stored then executed. What happens to the program internally in the processor when it is executed, and how is computation performed?
- There are three common flavours of implementation paradigm:
  - **Accumulator machine**
  - **Stack machine**
  - **Register machine**

Accumulator machine

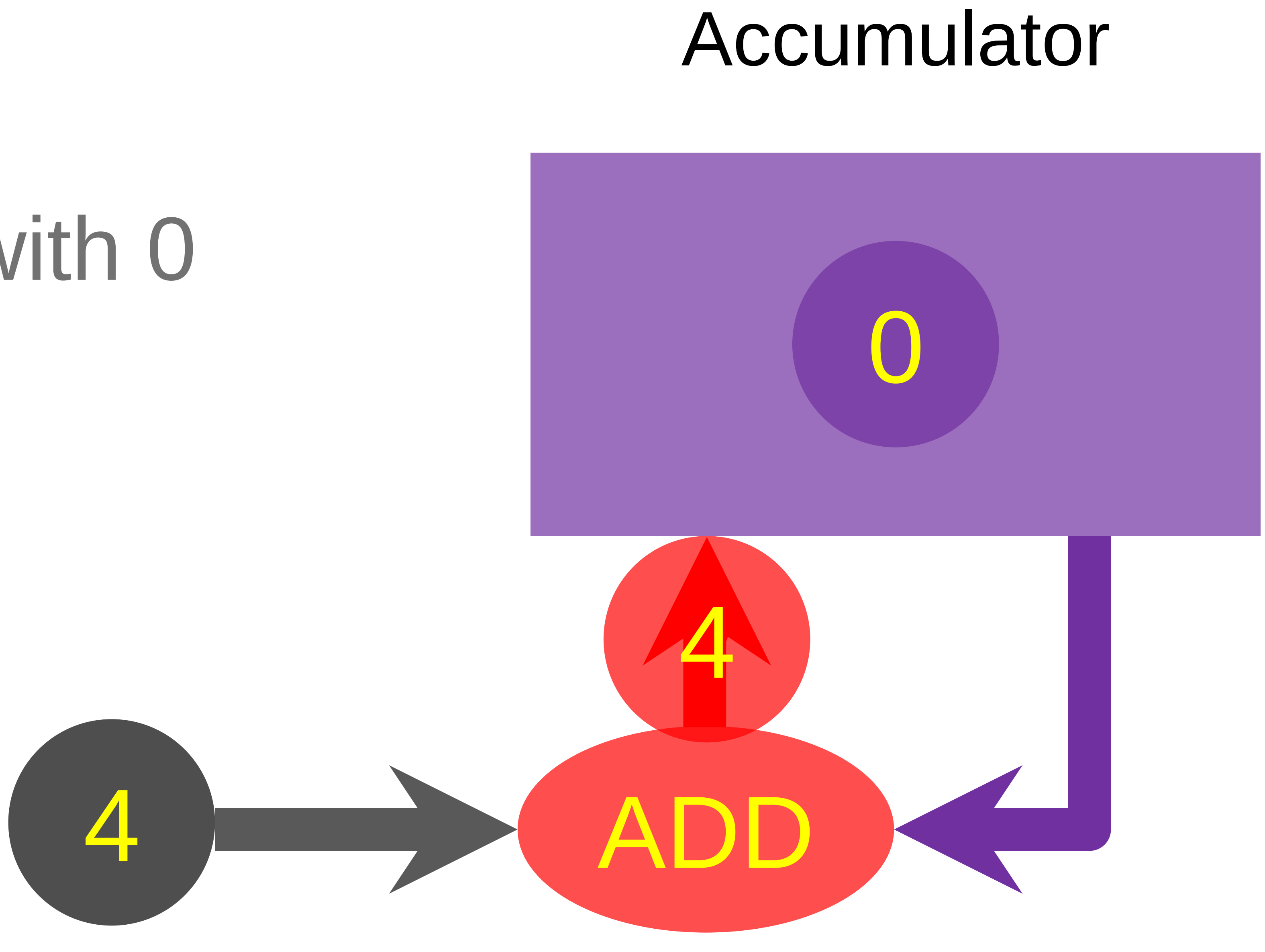
- A central accumulating store called "***the accumulator***"
- All instructions manipulate this store
  - May also access memory
- All data must flow through the store
- They are simple, fast, but require many instructions to do tasks.

Accumulator machine

e.g. “4 + 2”

→Initialise with 0

→Add 4



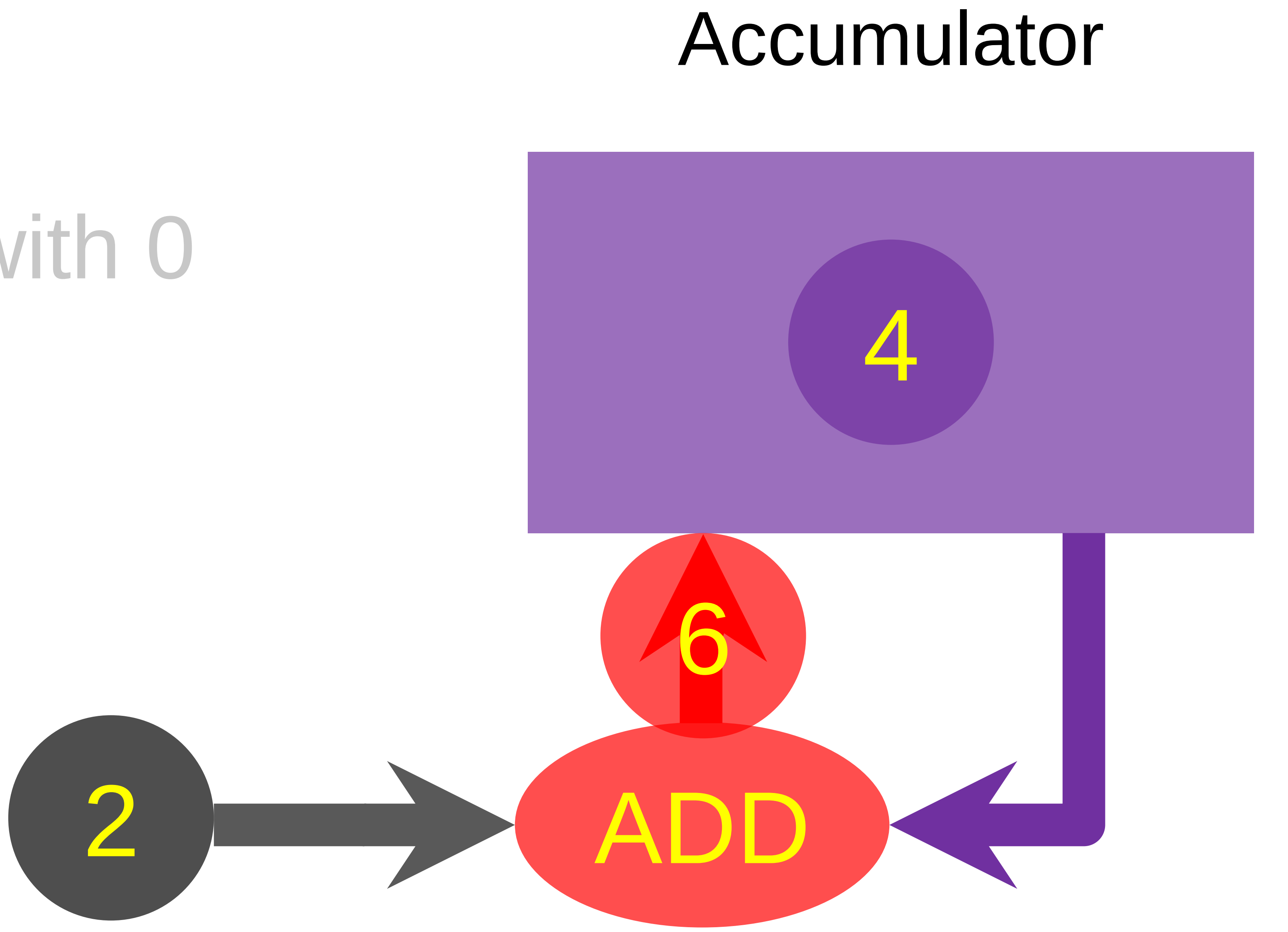
Accumulator machine

e.g. “4 + 2”

→ Initialise with 0

→ Add 4

→ Add 2

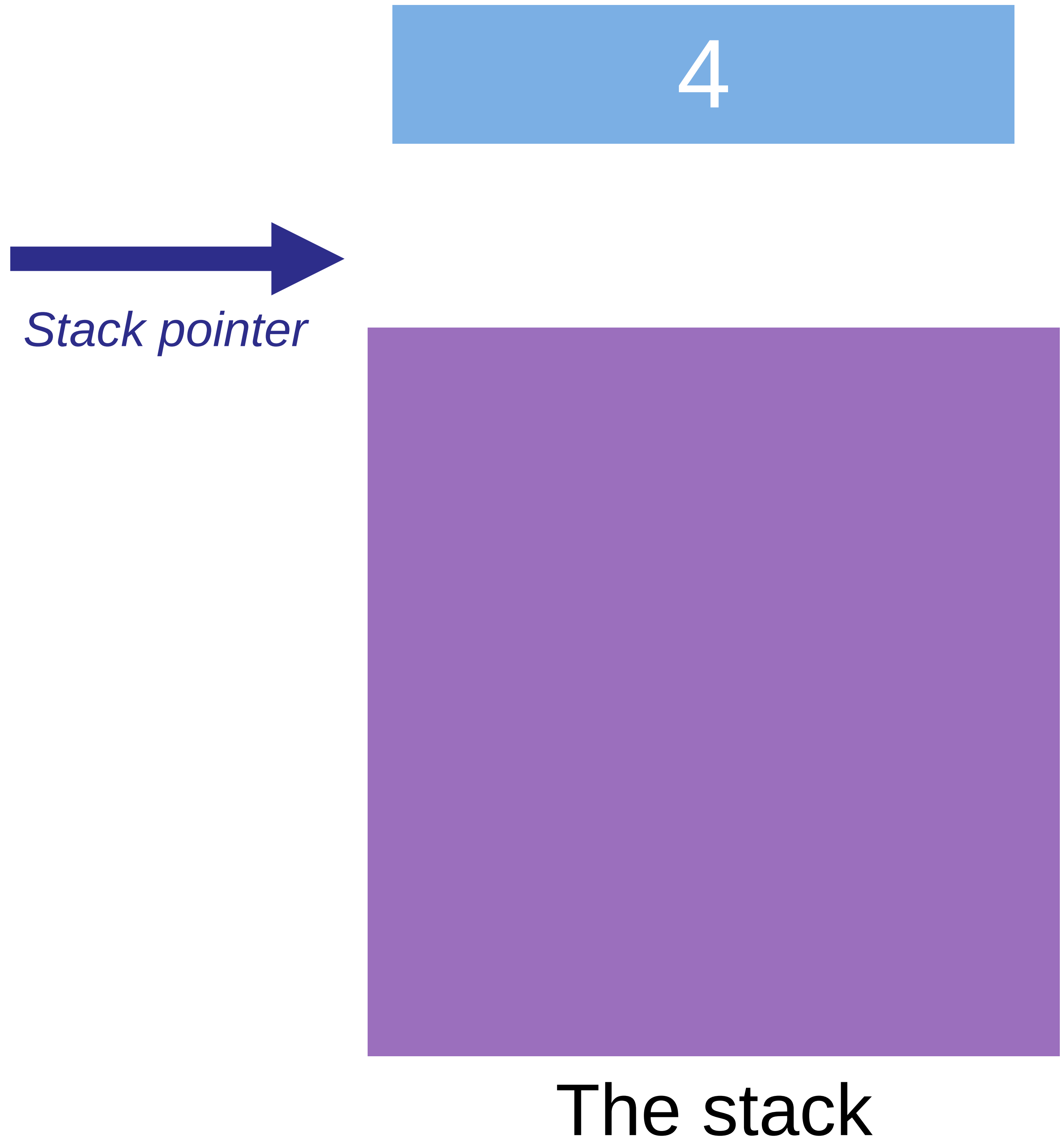


Stack machines

- A stack machine uses a processor ***stack*** to store information during execution.
- A ***stack pointer*** (a hidden register) keeps track of the current top of the stack.
- All operations modify the stack or stack pointer.
- The stack is a bottleneck.

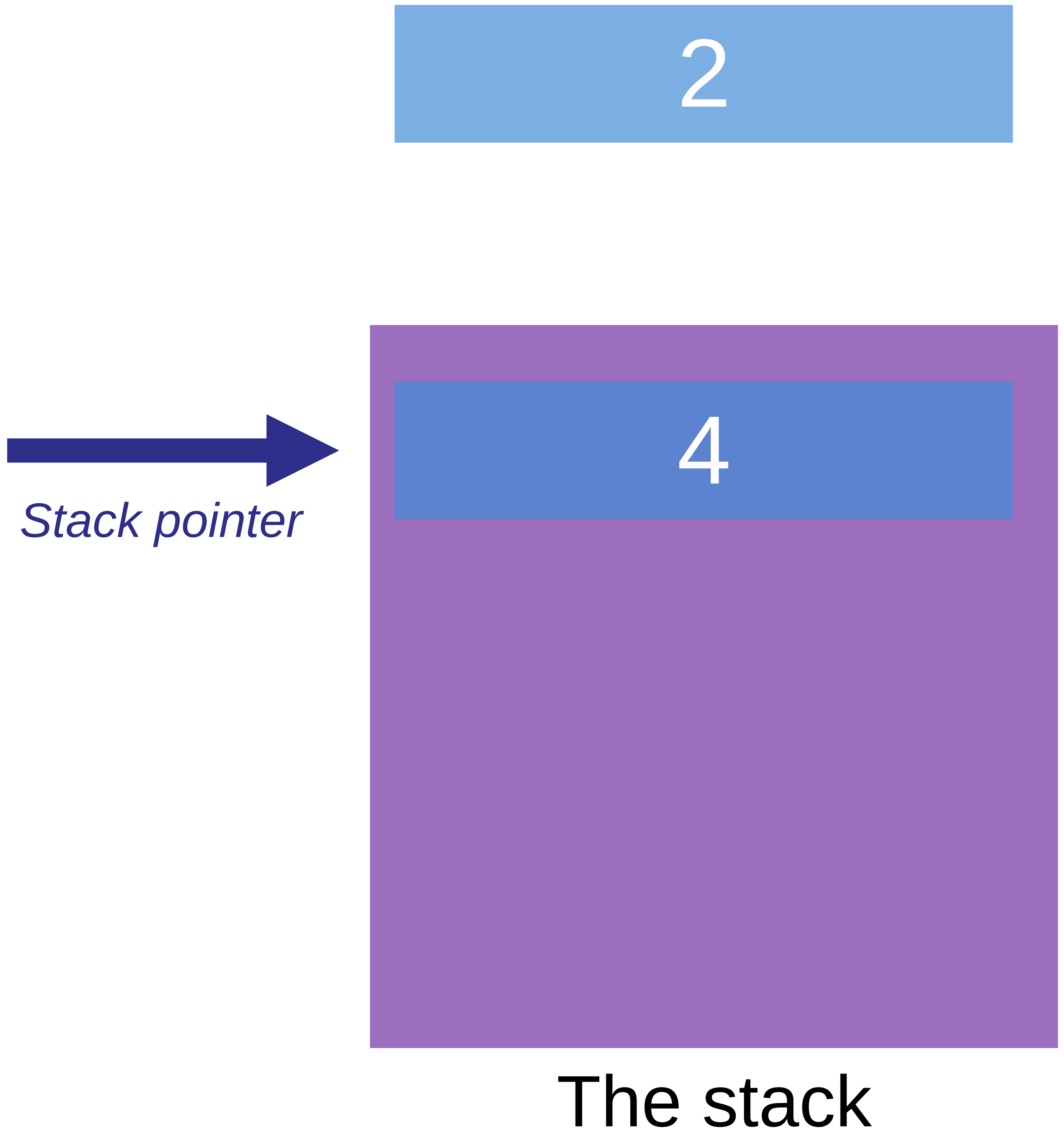
Stack machines

e.g. “4 + 2”



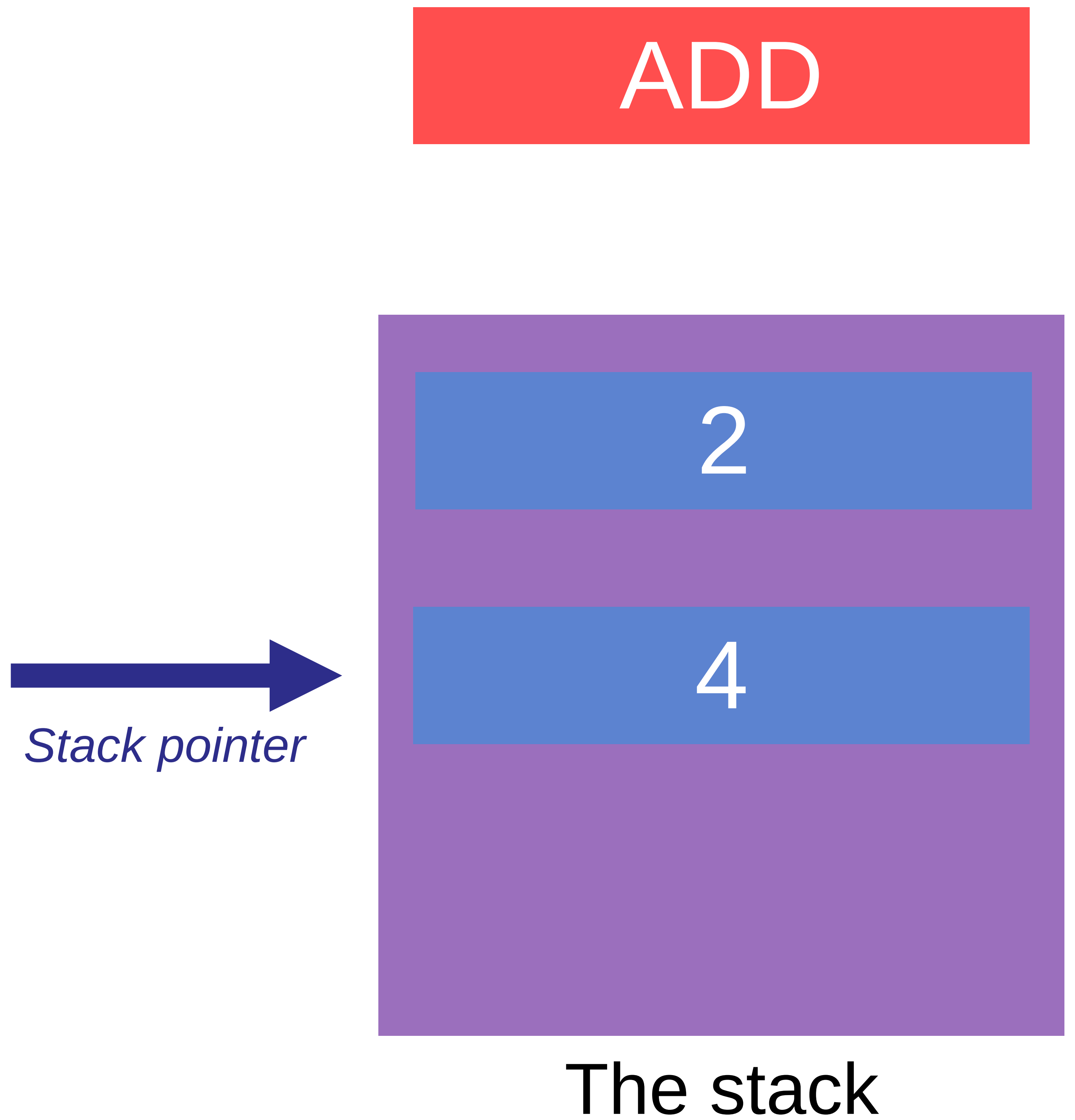
Stack machines

e.g. “4 + 2”



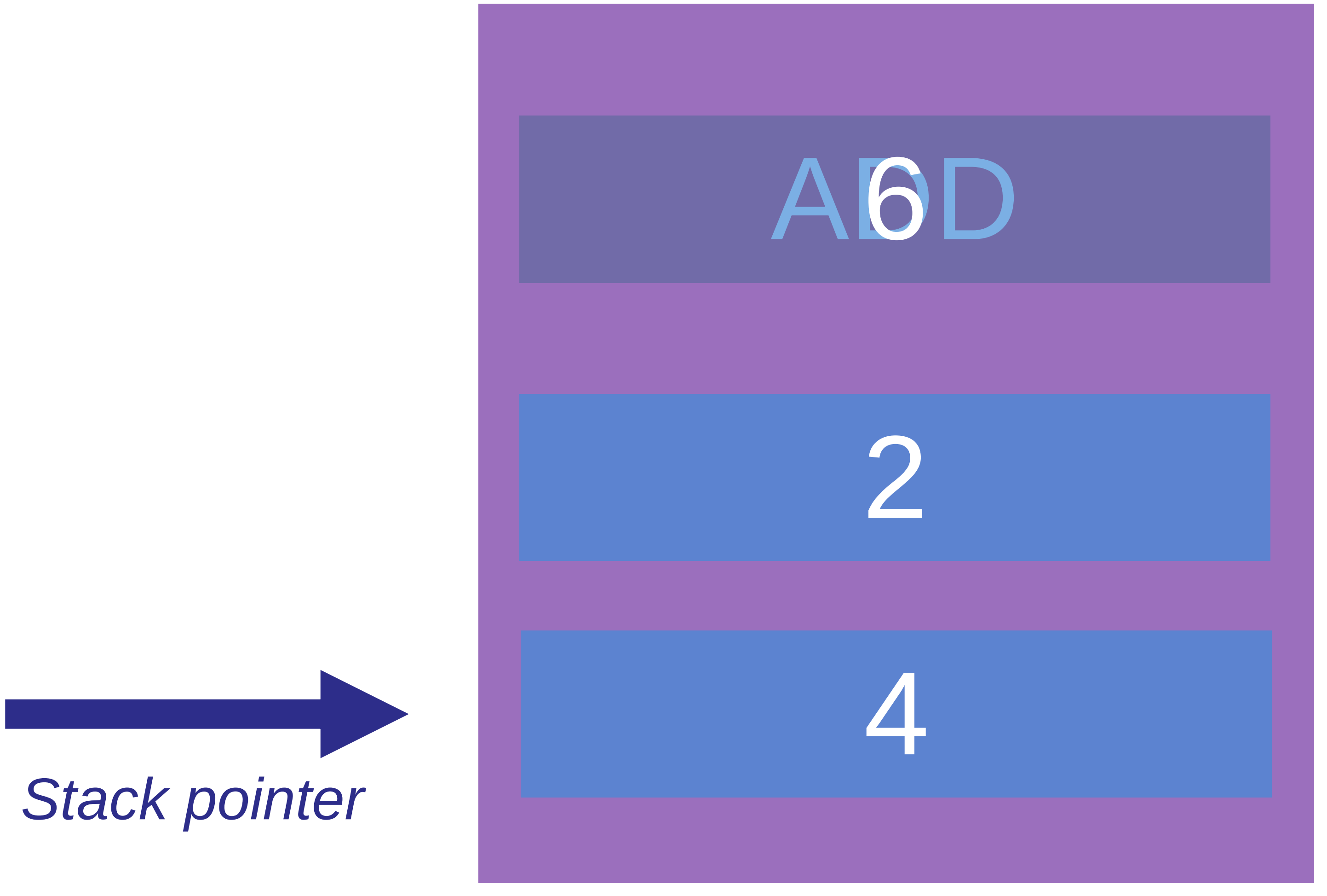
Stack machines

e.g. “4 + 2”



Stack machines

e.g. “4 + 2”



The stack

Reverse Polish notation

- We just saw that operations must be re-ordered to work-correctly.
- We can write them down in a format that makes sense for a stack machine, called "Reverse Polish" notation.
- Powerful way of mapping the problem and also for reasoning about stack machines.

Reverse Polish notation

- Simple rules of associativity and precedence.
- Operators always appear *after* their input data.
- Example:
  - Desired operation: 4 + 2
  - Reverse Polish: 4, 2, ‘+’
- ‘+’ is a binary operator, so consumes the two previous data values.