Ollscoil na Gaillimhe
University of Galway

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213
Computing Systems & Organisation

Programming Models

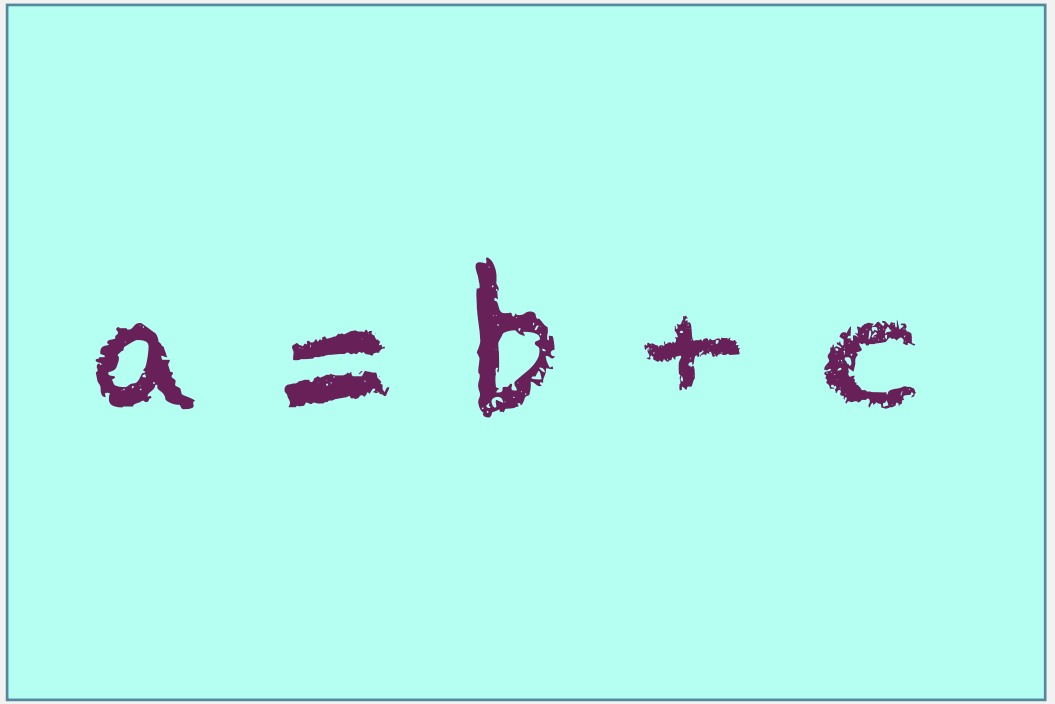
Outline

- Instruction types
- Stack
- Stack architectures
- GPR architectures
- Stack used to implement procedure calls

Programming Models

A processor programming model defines how instructions access their operands and how instructions are described in the processor's assembly language

Processors with different programming models can offer similar sets of operations but may require very different approaches to programming

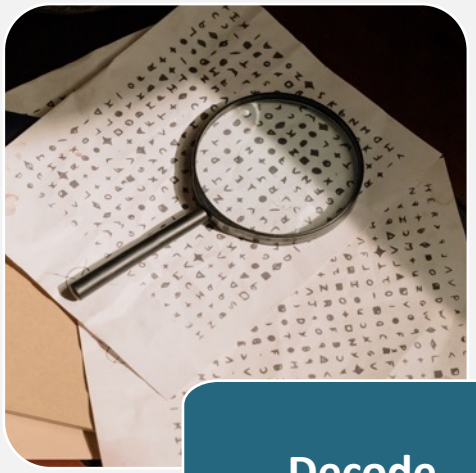


The Processor - Instruction Cycles

- The instruction cycle is the **procedure of processing an instruction** by the microprocessor:

Fetch

or read the instruction from the memory



Decode

what is to be done

Execute

Perform the operation

- Each of the functions fetch -> decode -> execute consist of a sequence of one or more operations inside the CPU (and interaction with the subsystems)

Types of Instructions

- **Data Transfer** Instructions
  - Operations that **move data** from one place to another
  - These instructions **don't modify** the data, they just copy it to the destination

- **Data Operation** Instructions
  - Instructions do modify their data values
  - They typically perform some operation (e.g., +/-/*) using one or two data values (operands) and store the result
- **Program Control** Instructions
  - **Jump** or **branch** instructions used to **go in another part** of the program; Jumps can be **absolute** or **conditional** (e.g., if then else)
  - Instructions that can generate **interrupts** (software interrupts)

Data Transfer Instructions (1)

Load data from memory into the microprocessor

These instructions copy data from memory into microprocessor registers (i.e., LD)

Store data from the microprocessor into the memory

Similar to load data, except that the data is copied in the opposite direction (i.e., ST)

Data is saved from internal microprocessor registers into the memory

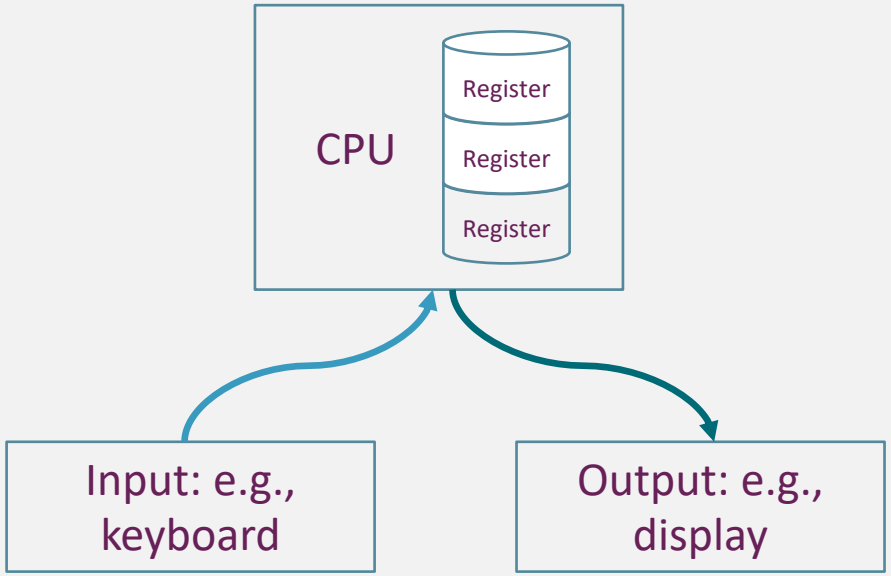
Move data within the microprocessor

These instructions move data from one microprocessor register to another (i.e., MOV)

Data Transfer Instructions (2)

- **Input data** to the microprocessor
  - A microprocessor may need to input data from the outside world, these are the instructions that input data from the input device into the microprocessor
  - An example: microprocessor needs to know which key was pressed (i.e., IORD)

- **Output data** from the microprocessor
  - The microprocessor copies data from one of its internal registers to an output device
  - In example: microprocessor may want to show on a display the content of an internal register (the key that has been pressed) (i.e., IOWR)



Data Operation Instructions

- **Arithmetic instructions**
  - add, subtract, multiply or divide
    - ADD, SUB, MUL, DIV, etc.
  - Instructions that increment or decrement one from a value
    - INC, DEC
  - Floating point instructions that operate on floating point values
    - FADD, FSUB, FMUL, FDIV

- **Logic Instructions**
  - AND, OR, XOR, NOT, etc.

- **Shift Instructions**
  - SR, SL, RR, RL, etc.

Program Control Instructions (1)

- **Jump and branch instructions (**Conditional or unconditional**):**
  - JZ: Jump if the zero flag is set
  - JNZ: Jump if the zero flag is NOT set
  - JMP: Unconditional jump; flags are ignored
  - Etc.

- **Comparison instructions:**
  - TEST: logical BITWISE AND

- C**alls and returns** a/from a routine **(**Conditional or unconditional**):**
  - CALL: call a subroutine at a certain line
  - RET: return from a subroutine
  - IRET: interrupt and return

Program Control Instructions (2)

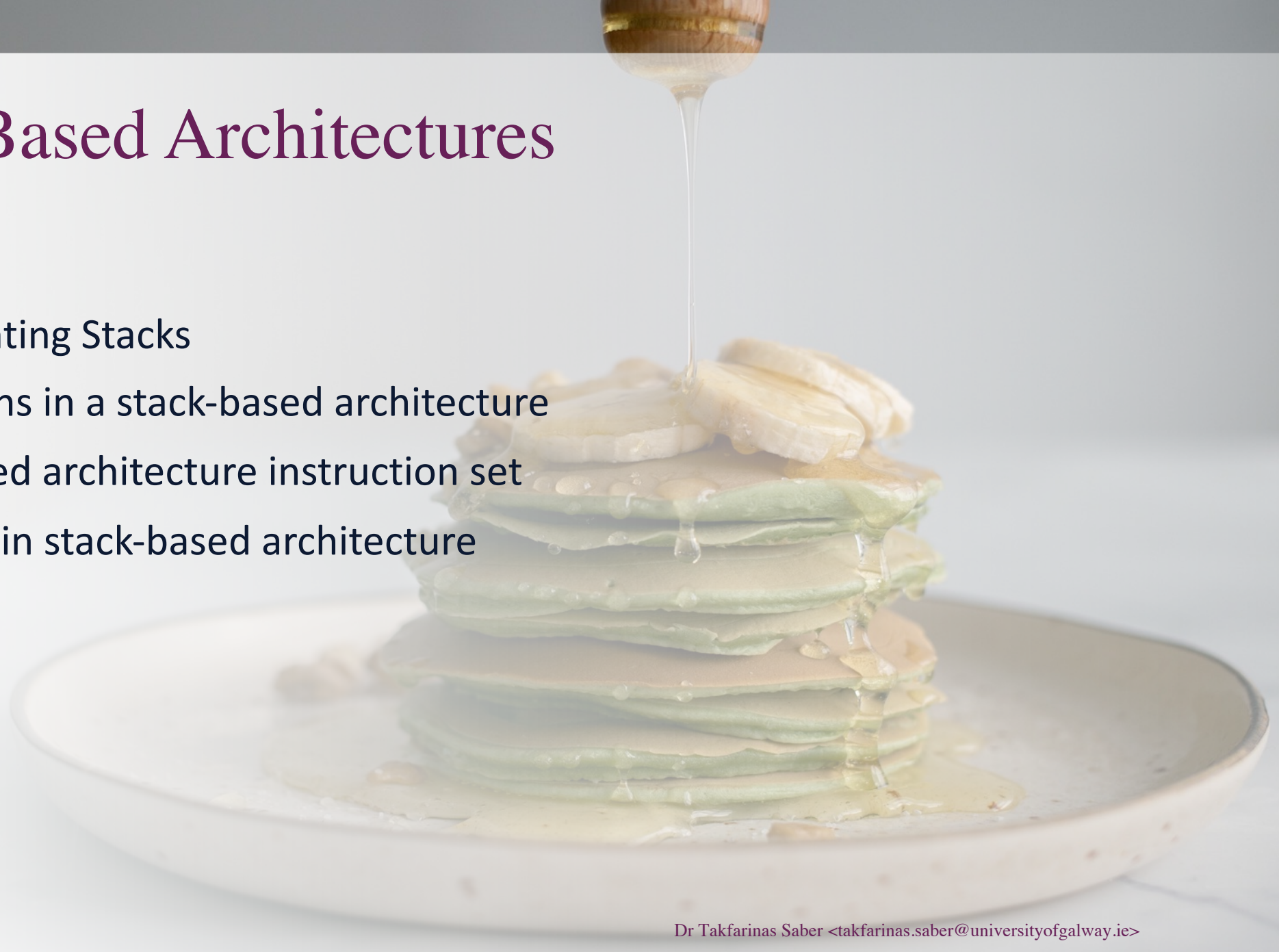
- **Software interrupts:**
  - Generated by devices outside of a microprocessor (not part of the instruction set)
  - called **hardware interrupts**
  - INT
- **Exceptions and traps:** triggered when valid instructions perform invalid operations,
  - E.g., dividing by zero
- **Halt instructions:** causes the processor to stop executions,
  - E.g., at the end of a program
  - HALT

https://www.tutorialspoint.com/assembly_programming/index.htm

Stack Architectures

Stack Based Architectures

- The Stack
- Implementing Stacks
- Instructions in a stack-based architecture
- Stack based architecture instruction set
- Programs in stack-based architecture



The Stack (1)

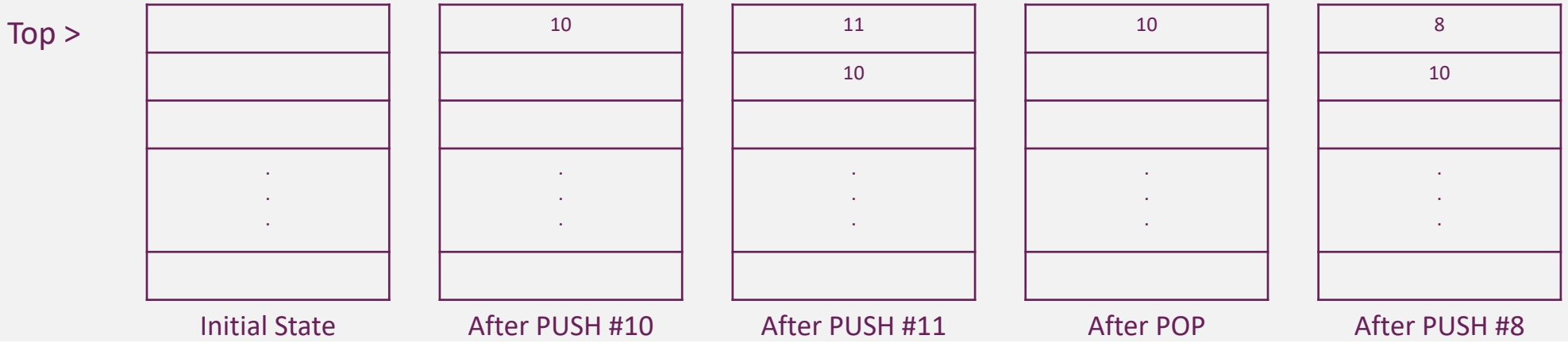
- **Last In First Out** (LIFO) data structure
- Consists of **locations**, each of which can hold a *word of data*
  - It can be used explicitly to **save/restore** data
- Supports two operations
  - **PUSH** – takes one argument and places the value of the argument in the top of the stack
  - **POP** – removes one element from the stack, saving it into a predefined register of the processor
- Used implicitly by procedure call instructions (if available in the instruction set)

The Stack (2)

When new data is added to the stack, it is placed at the top of the stack, and all the contents of the stack are pushed down one location

Consider the code:

```
PUSH #10
PUSH #11
POP
PUSH #8
```



Implementing Stacks

Two ways to implement a Stack:

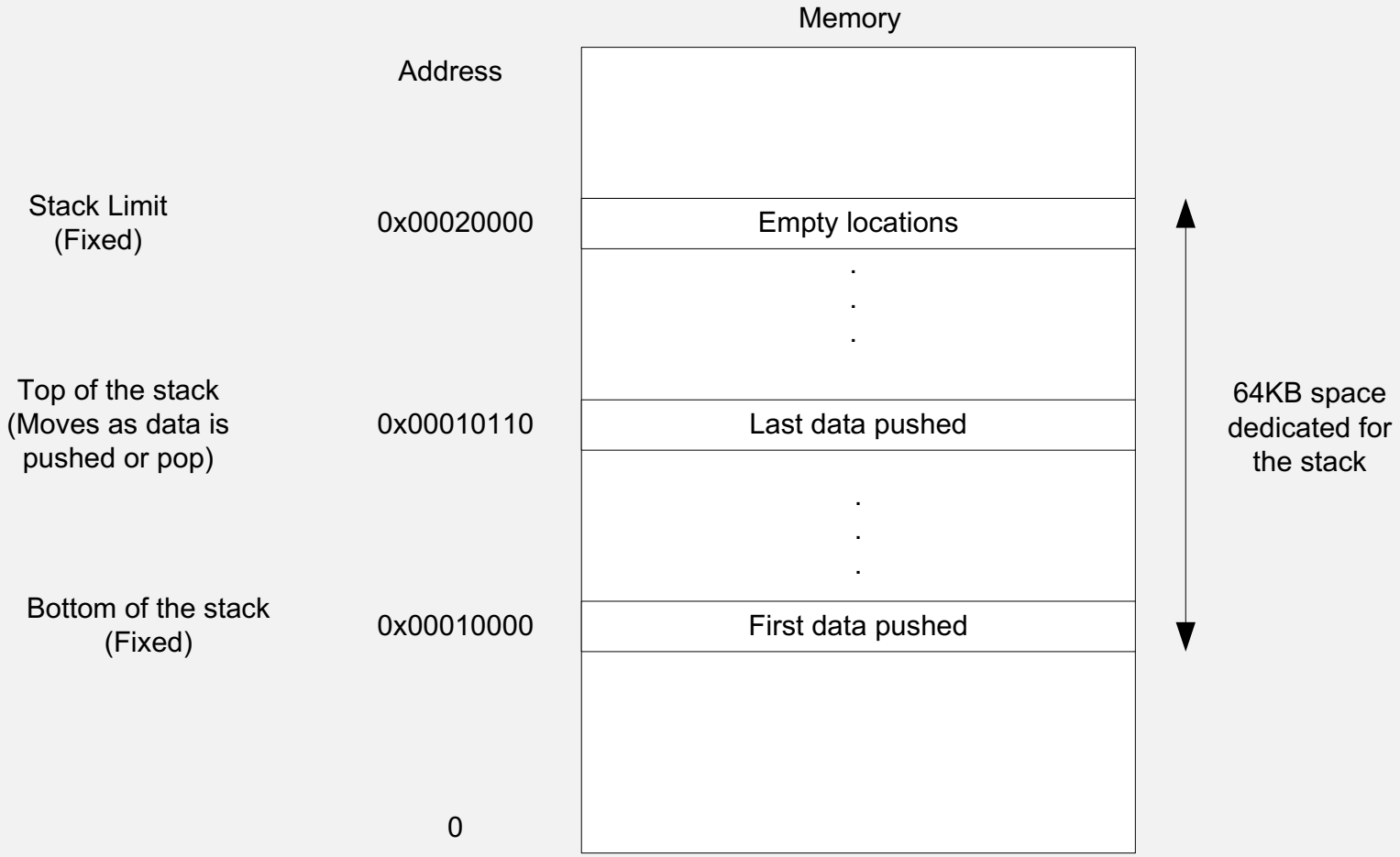
1. Dedicated hardware stack
   - It has a hardware limitation (limited number of locations)
   - Very fast

2. Memory implemented stack
   - Limited by the physical memory of the system
   - Slow compared with hardware stack, since extra memory addressing has to take place for each stack operation

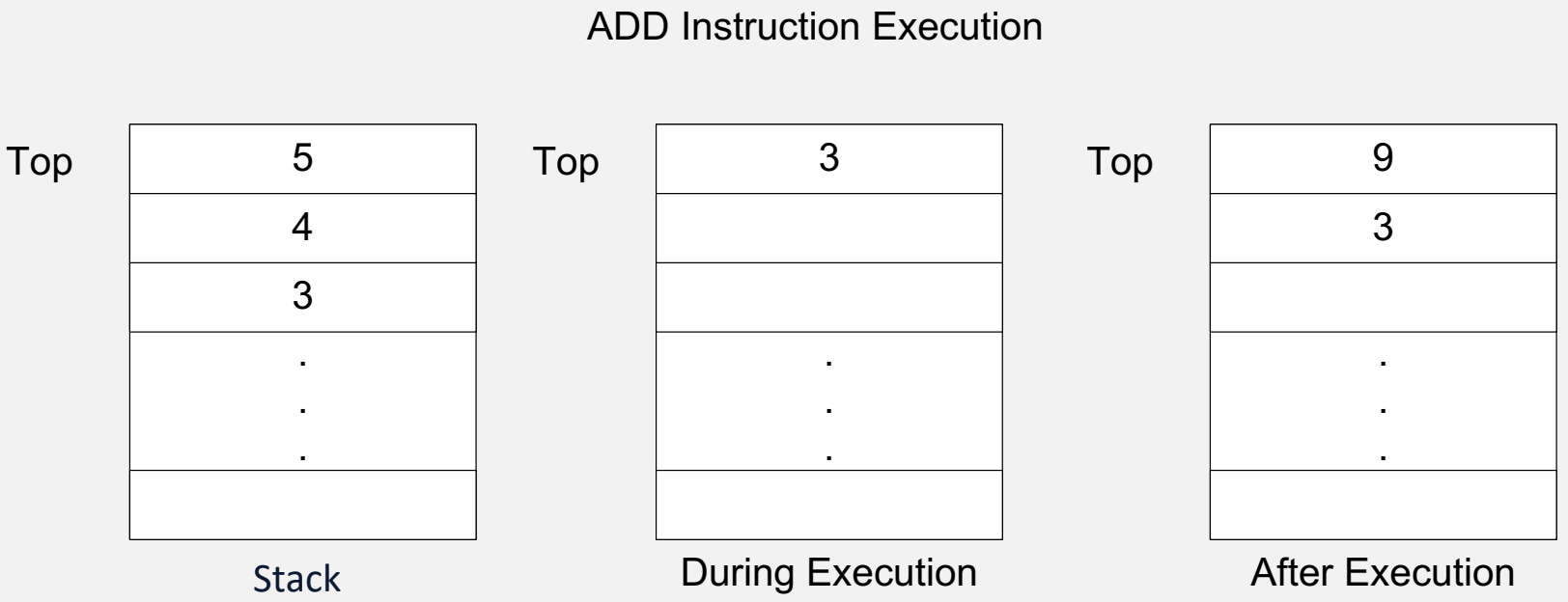
Stack overflows can occur in both implementations
   - When the amount of data in the stack exceeds the amount of space allocated to the stack (or the hardware limit of the stack)

Stack Implemented in Memory

- Every push operation will increment the top of the stack pointer (with the word size of the machine)
- Every pop operation will decrement the top of the stack pointer



Instructions in a Stack Based Architecture



- Get their operands from the stack and write their results to the stack
- Advantage - Program code takes little memory (no need to specify the address of the operands in memory or registers)
  Push is one exception, because it needs to specify the operand (either as constant or address)

Programs in Stack Based Architecture (1)

- Writing programs for stack-based architectures is not easy
  - Stack-based processors are better suited for postfix notation rather than infix notation
- **Infix** notation is the traditional way of representing math expressions, with operation placed between operands
  - E.g., a + b
- **Postfix** notation – the operation is placed after the operands
  - E.g., a b +
- Once the expression has been converted into postfix notation, implementing it in programs is easy
- Exercise: Create a stack-based program that computes: $A*(B-C)+(D+E)$

Simple Stack Based Instruction Set

Operation: *A**(*B*-*C*)+(*D*+*E*)

	# stack contents (leftmost = top = most recent)
push A	# A
push B	# B A
push C	# C B A
subtract	# B-C A
multiply	# A*(B-C)
push D	# D A*(B-C)
push E	# E D A*(B-C)
add	# D+E A*(B-C)
add	# A*(B-C)+(D+E)

General Purpose Register Architecture

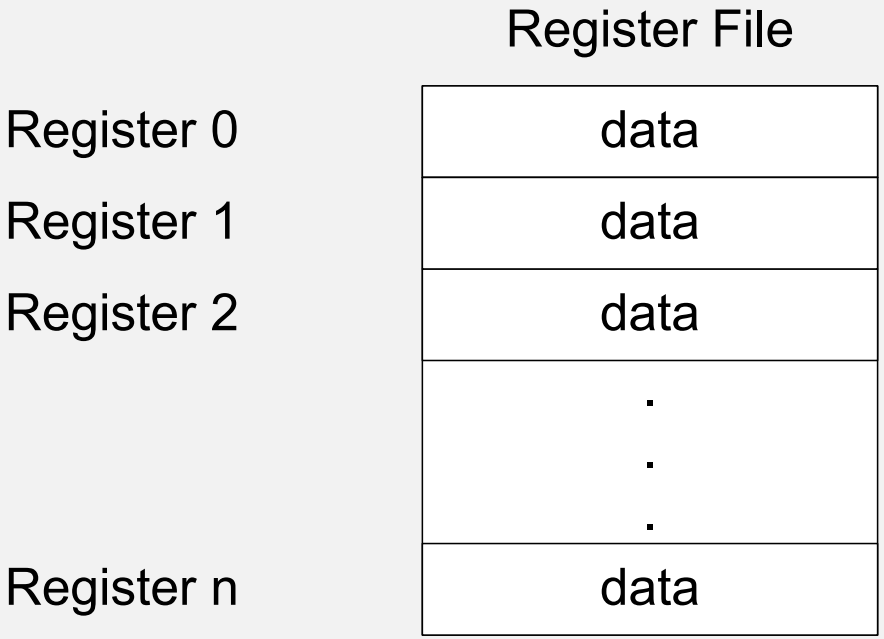
General Purpose Register Architecture

- Instructions in a GPR architecture
- A GPR instruction set
- Programs in GPR architecture

General Purpose Register Architecture (1)

- The instructions read their operands and write their results to **random access register file**.
- The general purpose register file allows the **access of any register in any order** by specifying the number (register ID) of the register
- The **main difference** between a general purpose register and the stack is that reading repeatedly a register will produce the same result and will not modify the state of the register file.
  - Popping an item from a LIFO structure (stack) will modify the contents of the stack

General Purpose Register Architecture (2)



- Many GPR architectures assign special values to some registers in the register file to make programming easier
  - I.e., sometimes, register 0 is hardwired with value 0 to generate this most common constant

Instructions in GPR Architecture (1)

- GPR instructions need to **specify:**
  - **the register** that hold their **input operands**
  - and the register that will hold the **result**
- The most common format is the **three operands instruction format**
  - E.g., **ADD r1, r2, r3** instructs the processor to read the contents of r2 and r3, add them together and write the result in r1
- Instructions having two or one input are also present in GPR architecture

Instructions in GPR Architecture (2)

- A significant difference between GPR architecture and stack-based architecture:
  - Programs can choose **which values should be stored in the register file** at any given time, allowing them to cache most accessed data
- In stack based architectures, once the data has been used, it is gone.
- GPR architectures have **better performance** from this point of view, at the expense of needing **more storage space** for the program
  - larger instructions need to encode the addresses of the operands

Simple GPR Instruction Set

ST (ra), rb	(ra) <- rb
LD ra, (rb)	ra <- (rb)
ADD ra, rb, rc	ra <- rb + rc
SUB ra, rb, rc	ra <- rb - rc
AND ra, rb, rc	ra <- rb & rc
OR ra, rb, rc	ra <- rb | rc
MOV ra, rb	ra <- rb

(ra): The memory location whose address is contained in ra

Programs in a GPR Architecture (1)

- Programming a GPR architecture processor is **less structured** than programming a stack based architecture one.
- There are **fewer restrictions on the order** in which the operations can be executed
- On stack based architectures, instructions should execute in the order that would leave the operands for the next instructions on the **top of the stack**
- On GPR, any order that places the operands for the next instruction **in the register file** before that instruction executes is valid.
- Operations that access different registers can be **reordered** without making the program invalid

Programs in GPR Architecture (2)

- Create a GPR based program that computes:
  - 2 + (7&3)

- GPR programming uses infix notation:

```
MOV R1,  #7
MOV R2,  #3
AND R3, R1, R2
MOV R4, #2
ADD R4, R3, R4
```

- The result will be placed in R4

Comparing Stack based and GPR Architectures

- **Stack-based architectures**
  - Instructions take **fewer bits** to encode
  - **Reduced amount of memory** taken up by programs
  - Manages the **use of register automatically** (no need for programmer intervention)
  - Instruction set does not change if size of register file has changed

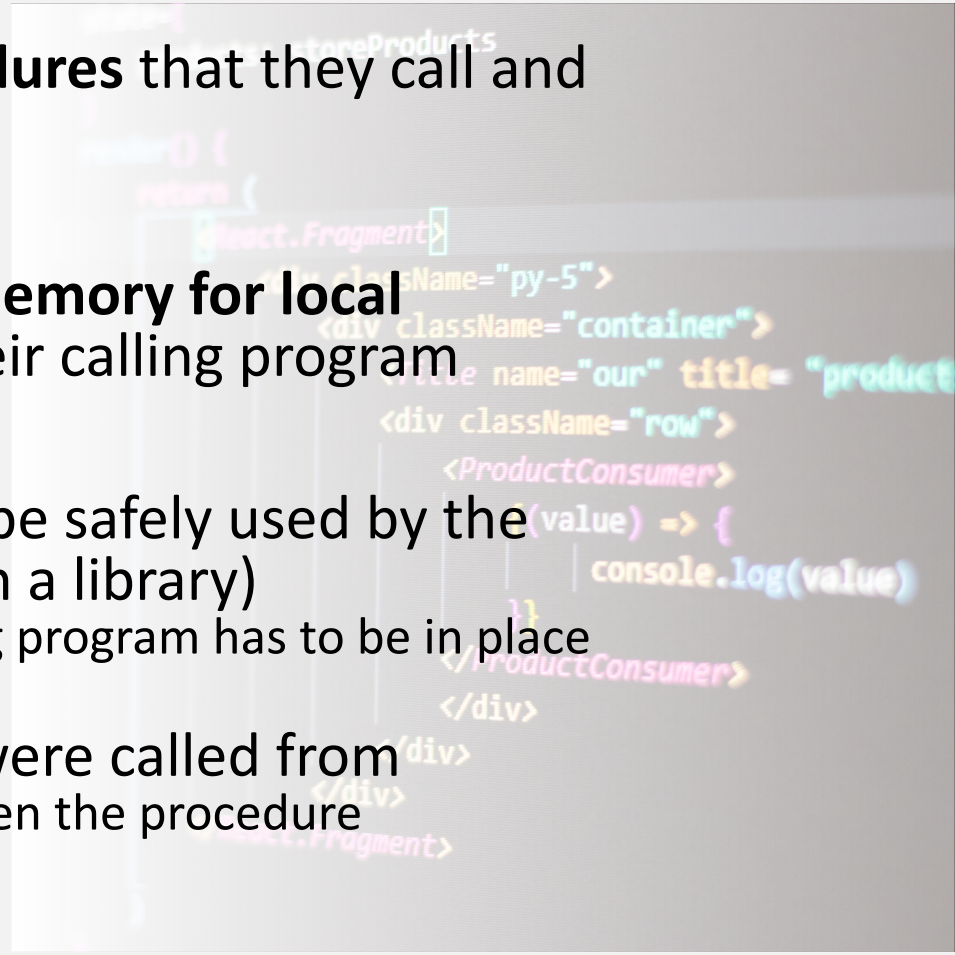
- **GPR architectures**
  - With evolution of technology, the amount of space taken up by a program is less important
  - Compilers for GPR architectures achieve **better performance** with a given number of general purpose registers than those on stack-based architectures with same number of registers
    - The compiler can **choose which values to keep** (cache) in register file at any time

- Stack based processor are still attractive for certain embedded systems. GPR architectures are used by modern computers (workstations, PCs, etc.)

Stacks for Procedure Calls

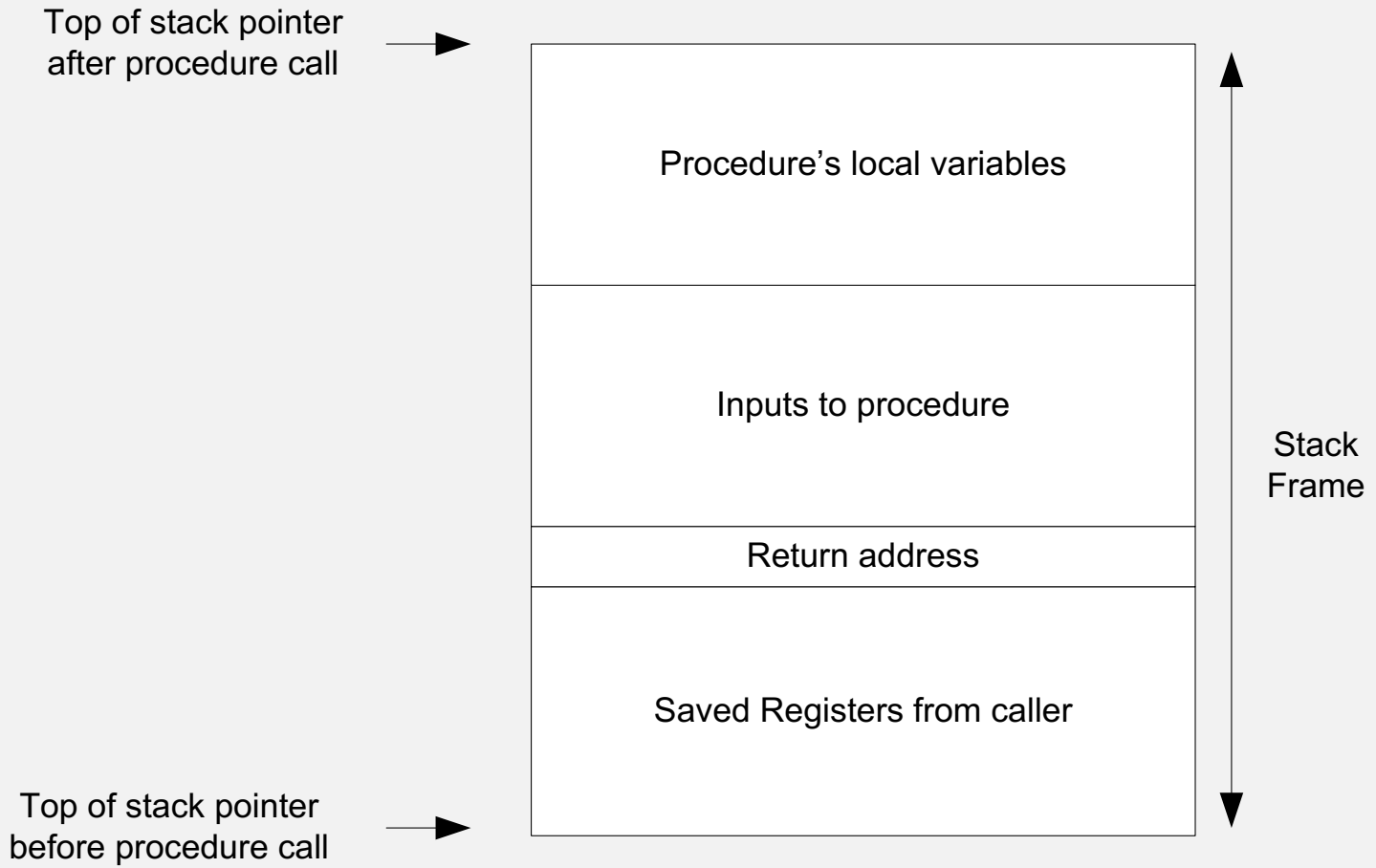
Using Stacks to Implement Procedure Calls (1)

- Programs need a way to **pass inputs to the procedures** that they call and to receive outputs back from them
- Procedures need to be able to **allocate space in memory for local variables**, without overriding any data used by their calling program
- It is impossible to determine which registers may be safely used by the procedure (especially if the procedure is located in a library)
  - So, a mechanism to **save/restore registers** of the calling program has to be in place
- Procedures need a way to figure out where they were called from
  - So, the execution can **return to the calling program** when the procedure completes (they need to restore the program counter)



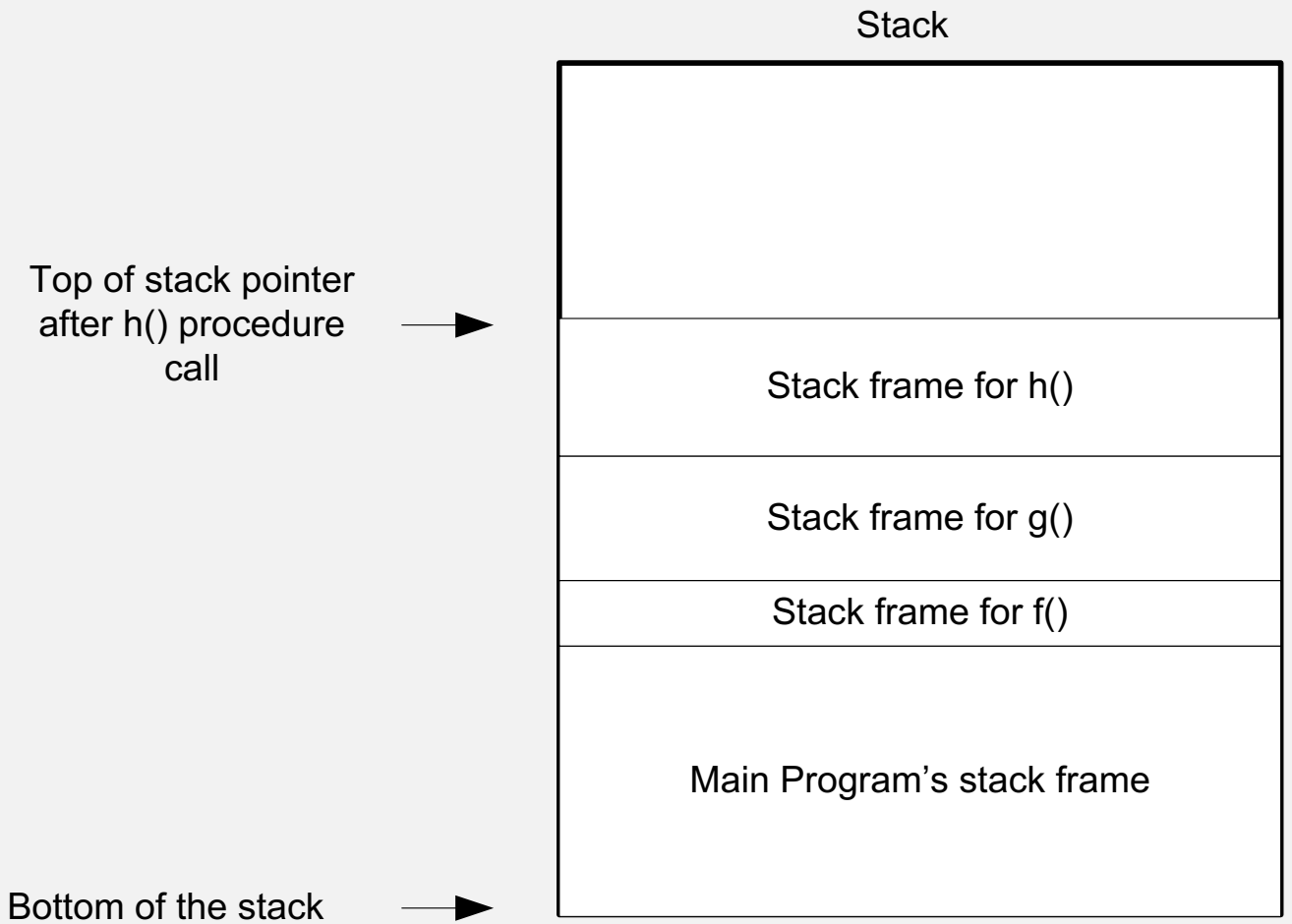
Using Stacks to Implement Procedure Calls (2)

- When a procedure is called, a **block of memory** in the stack is allocated. This is called a stack frame
- The top of the stack pointer is incremented by the **number of locations** in the stack frame
- When a procedure finishes, it jumps to the **return address** contained in the stack and execution of the calling program resumes.



Using Stacks to Implement Procedure Calls (3)

- Nested procedure calls:
  - main program calls function f(),
  - function f() calls function g(),
  - function g() calls function h()



References

- “Computer Systems Organization & Architecture”, John D. Carpinelli, ISBN: 0-201-61253-4
- “Computer Architecture”, Nicholas Charter, ISBN – 0-07-136207

- **Images taken from Pexels**:
- Photo of dog by Jozef Fehér
- Photo of magnifying glass and fencers by cottonbro
- Photo of mirror by sum+it
- Photo of hardware by Valentine Tanasovich
- Photo of tree by Johannes Plenio
- Photo of stop sign by Mwabonje
- Photo of stack by Monstera
- Photo of drawers by Stephan Streuders
- Photo of code by Antonio Batinić