Lancaster University

SCC131: Digital Systems

Topic 5: Instruction set architecture (ISA)

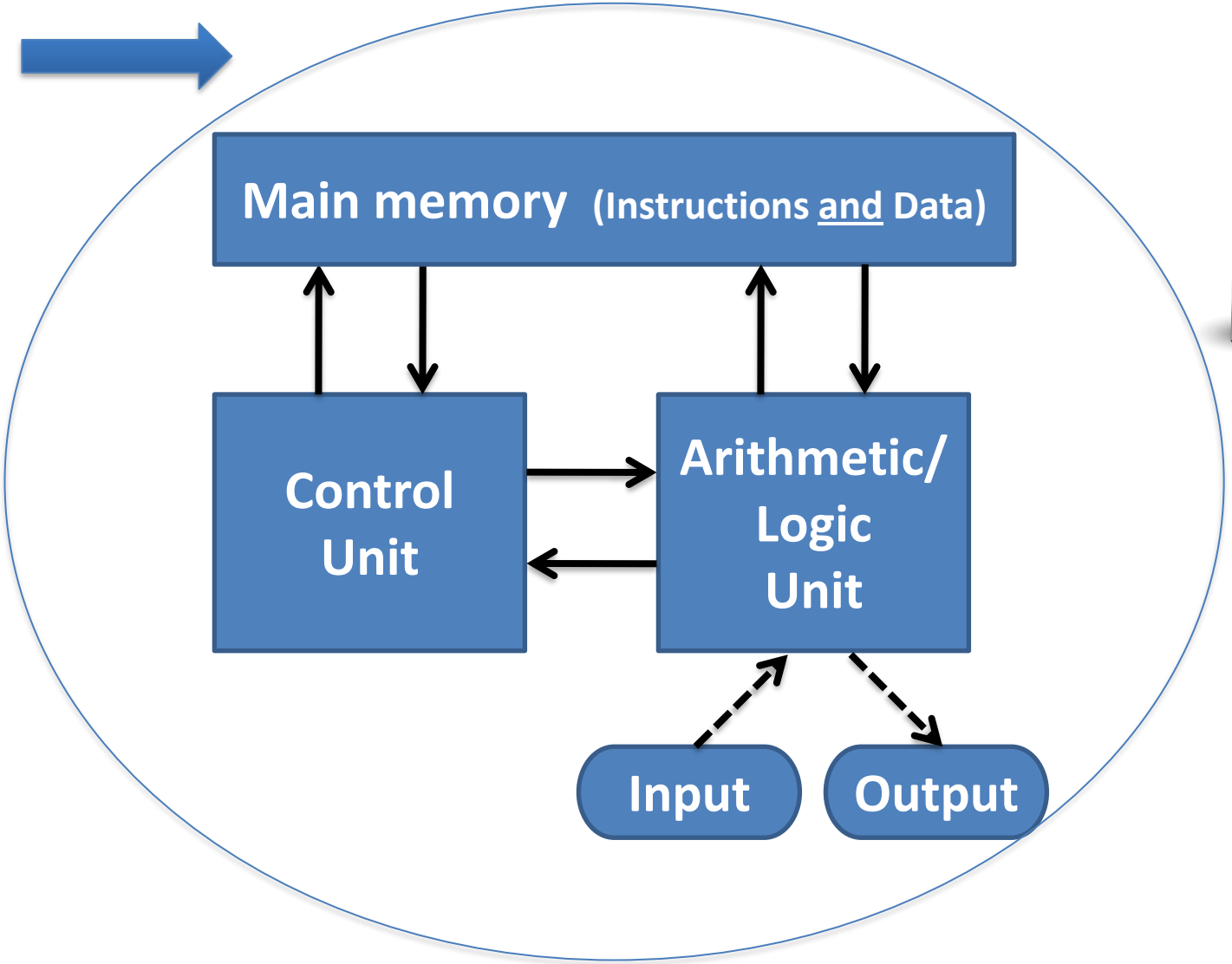
Week 5 Quiz

- This is a great opportunity to check your progress.
- The Quiz will cover modules 111, 121, and 131 (if you do CS, Software Engineering, Data Science, CyberSecurity).
- 3 Sections, one from each module.
- You should spend 30 min per section.
- We will ask you questions about material taught between Week 1-4.

- You are only allowed to take it during your lab slot on Week 5
- If you have any problems or need to swap labs contact scc-teaching-office@lancaster.ac.uk

Reminder of the von Neumann architecture

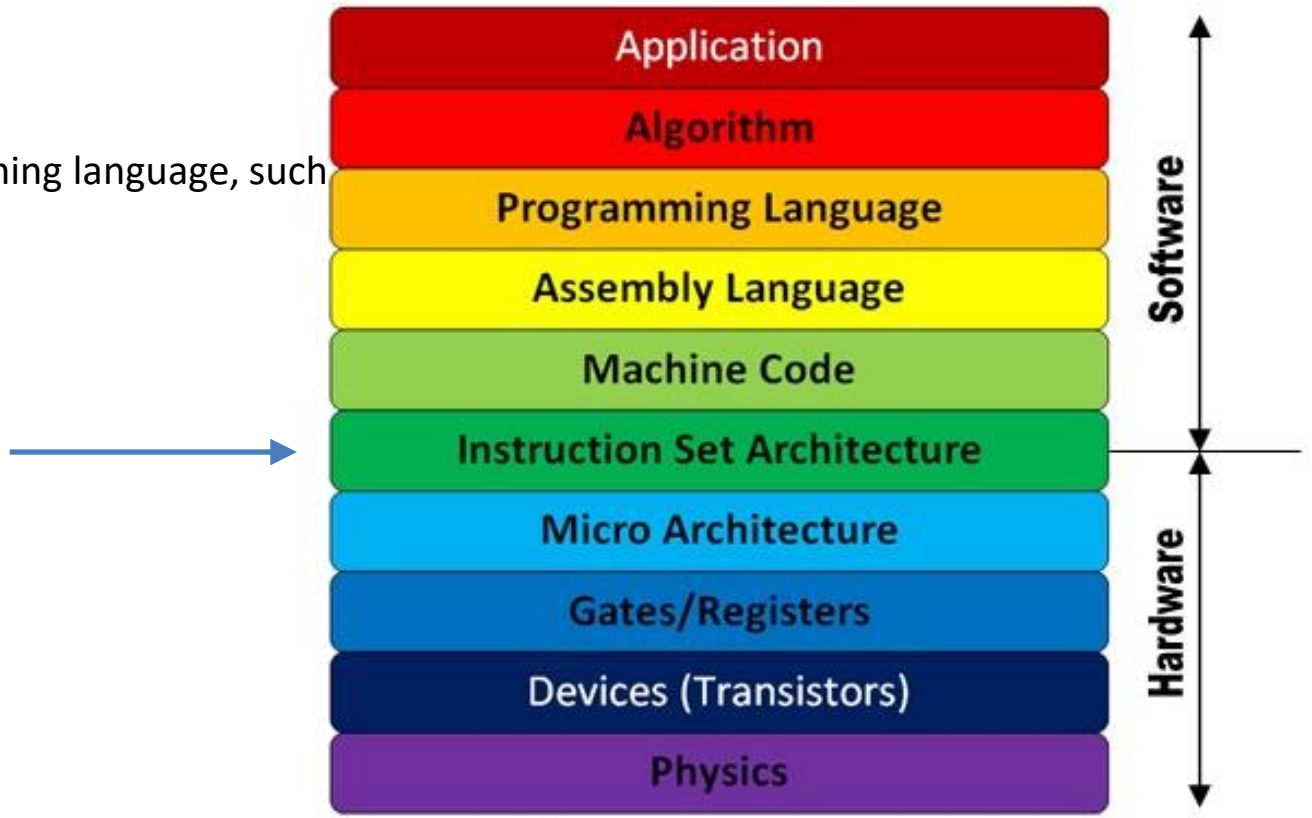
our focus



An abstraction view of computer architecture

– ISA is the interface between the hardware and software

High level programming language, such as Python, Java, etc



What is an “instruction set architecture (ISA)”?

– A set of commands that a processor can understand and execute

```
muli $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```

- What is the set of available instructions?
- What is the set of available registers?
- How many operands do instructions need?
- What are the sizes and types of the operands?
- How are operands accessed (e.g., stack-based ISAs don’t support random access to memory – see next slide)?
- How many operands can be in registers (vs. in memory)?
- How many clock ticks does it take to execute an instruction?
- …

Some examples of “families”of ISAs (1)

- Accumulator-based ISAs
  - Mainly used in early computers
  - Most instructions operate on two operands – a register and a memory location
- General-purpose-registers ISAs
  - There are multiple registers, and most instructions can access registers and memory locations in various combinations

Some examples of "families"of ISAs (2)

- Reduced Instruction Set Computers (RISC)
  - (Contrast with *Complex Instruction Set Computers* – or CISC)
  - Small number of instructions, all with a common operand format
  - All instructions apart from LOAD and STORE operate on registers (a.k.a. "load/store architecture")
  - Typically, instructions take only 1 clock cycle to execute

- Stack-based ISAs
  - Instructions can only access operands at the top of a ***stack*** – special PUSH and POP instructions are used to manipulate the stack (see more later on stacks)

Case study: the MIPS 1 ISA (1)

- A very influential instruction set developed by a company called MIPS Technologies
  - Started as the *Microprocessor without Interlocked Pipeline Stages (MIPS) project from Stanford researchers.*
  - Version 1 (1985) –> MIPS32/64 release 6 (April 2017)
  - Can be categorized as RISC

Case study: the MIPS 1 ISA (2)

- The MIPS 1 ISA
  - Has the following categories of instructions
    - Arithmetic and logic instructions
    - Data transfer instructions
    - Control transfer instructions
  - MIPS32 has 32 general purpose registers, $0-$31
    - Each register holds 32 bits
  - No floating point capability

MIPS arithmetic and logic instructions

- *(n.b.* ***all these instructions operate on registers inside the processor; not directly on main memory locations****)*
- **addu** $d, $s, $t
  - $d = $s + $t; unsigned addition; ignore any overflow
- **add** $d, $s, $t
  - $d = $s + $t; 2's complement addition; on overflow, execute a **trap**
- **addi** $t, $s, C
  - $t = $s + constant C (also used to copy one register to another: addi $1, $2, 0)
- **and** $d, $s, $t
  - $d = $s AND $t
- **andi** $d, $s, C
  - $d = $s AND constant C
- **or** $d, $s, $t
  - $d = $s OR $t
- **sll** $d, $t, constant C
  - "Shift left logical" immediate – shifts $t to the left by C bits and puts the result into $d (i.e., multiplies by 2^C)
- Also: addu, sub, subu, addiu, mult, multu, div, divu, ori, xor, nor, slt ("*set on less than*"), slti, srl, sra ("*shift right arithmetic*"), sllv ("*shift left logical*"), srlv srav

MIPS data-transfer instructions

- Whereas the arithmetic and logic instructions operate on registers inside the processor…
  - The **data-transfer instructions** enable data to be transferred, in both directions, between these registers and main memory
- **lw** $t,C($s)
  - Load the 32-bit unit (“word”) that starts at MEM[$s+C] into $t
- **sw** $t,C($s)
  - Store the word in $t into MEM[$s+C] and the 3 bytes following
- Also: lh, lhu, lb, lbu, sh, sb, lui, mfhi, mflo, mfcZ, mtcZ

The “MEM[]” notation abstracts main memory as an array of bytes

MIPS control-transfer instructions

- By default, instructions execute sequentially, i.e., as they are laid out in main memory…
  - **Control-transfer instructions** enable exceptions to that default path of execution

conditional

- **beq** $s, $t, C
  - If ($s == $t) jump to the instruction at PC+4+(4*C)
- **bne** $s, $t ,C
  - If ($s != $t) jump to the instruction at PC+4+(4*C)

unconditional

- **j** C
  - Jump (*unconditionally*) to the instruction at the specified address
- **jr** $s
  - Jump (*unconditionally*) to the address contained in register $s
- **jal** C
  - Call a subroutine; we'll look more at this later…

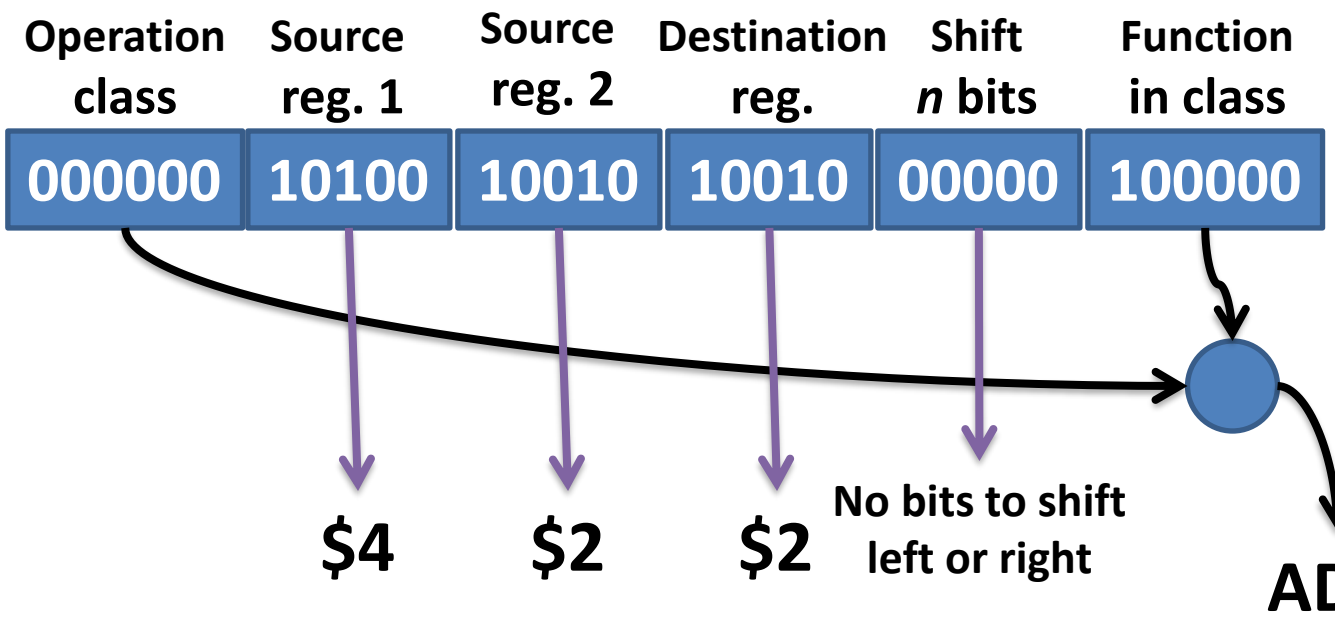
Instruction encoding (example)

Assembly Language:

```
swap:
muli $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```

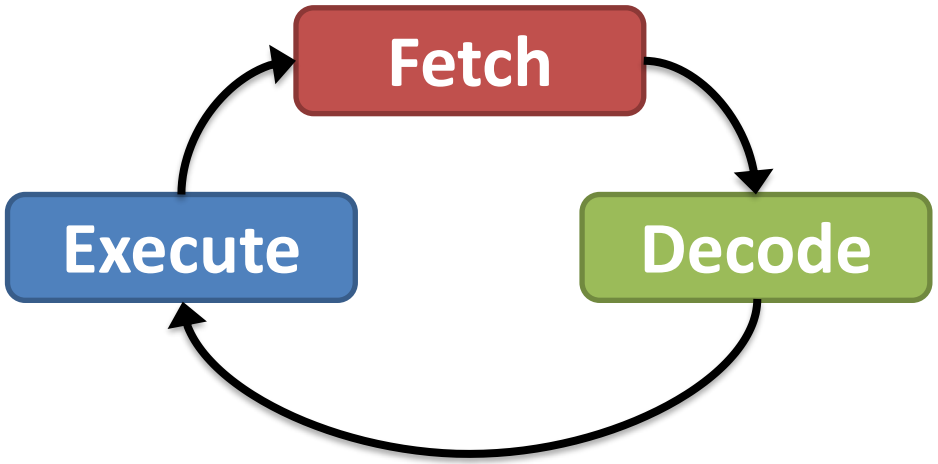
Recall once more that everything stored in a computer is **just bits** – and that it's humans who impose interpretations on these bits!

Machine Language: *(as laid out in memory: 32 bits)*



Basic processor operation: the fetch-decode-execute cycle

- All the CPU ever does in its life is endlessly repeat this cycle:
  1. Fetch the **next instruction** (from address held in the "PC" – see next slide)
  2. Decode it
  3. Execute it



- We call this the ***fetch-decode-execute cycle***
  - We will see in more detail how this works inside the processor when we will look at the "control unit" later on

PC refers to: Program Counter

Registers (1)

- We've looked at instructions, now let's look more at *registers*…

- Registers are *holding areas for data being worked on inside the CPU*
  - Often (as in MIPS), arithmetic and logic instructions are designed to work *on registers only*, not on main memory
    - This is because registers are much *faster* than main memory
    - We use the data-transfer instructions for register ↔ main memory transfers

Registers (2)

- General purpose and special registers
  - The registers used by the arithmetic and logic instructions are called **general purpose registers**
    - In computers like MIPS we have a largish set of these—a "file" of 32 x 32-bit registers
      - Because its registers are 32 bits wide, MIPS is said to have a *word size* of 32 bits
      - This is also the size of the basic unit of transfer between the registers and main memory
        (although main memory can also be accessed at the granularity of half-words and individual bytes)
  - Processors also have **special registers** such as the *program counter* (PC) and the *stack pointer* (SP)…

The program counter (PC)

- The PC is a “special register” that you can’t load and store like you can the general-purpose registers
- Think of the PC as a ***pointer*** to the next instruction to be executed
- In the (default) case of sequential execution, the value in the PC “moves over” each instruction after it has been fetched

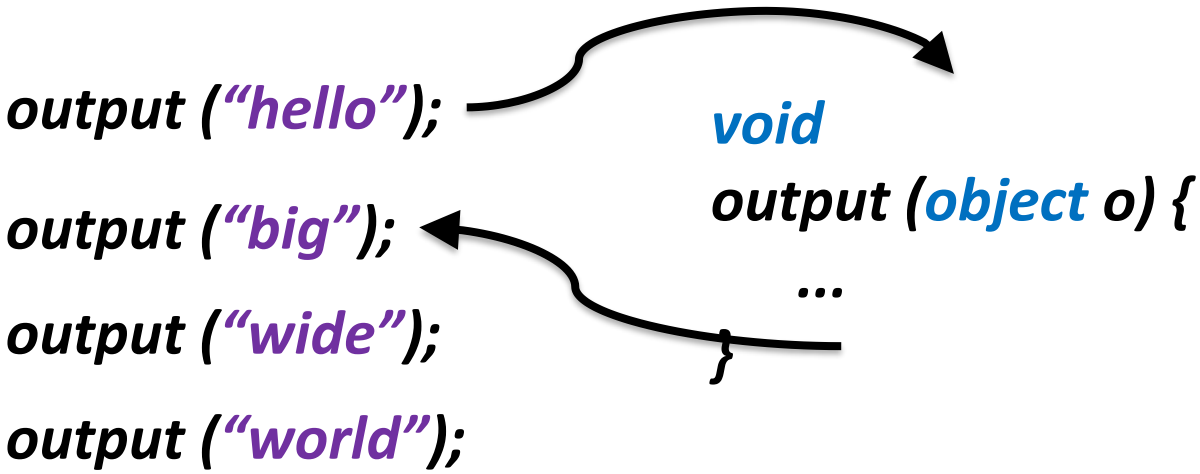
  PC ← PC + *length of current instruction and its operands*
- If we explicitly change the contents of the PC by using a control transfer instruction, execution skips immediately to that point in memory

  PC ← *memory address of some instruction*

  – This is how we implement jumps/ gotos, loops, conditional branches, etc in programming languages…

Handling subroutines

- In programming, we often need to transfer control from one subroutine/function/procedure/method to another *...and then go back again when its done*



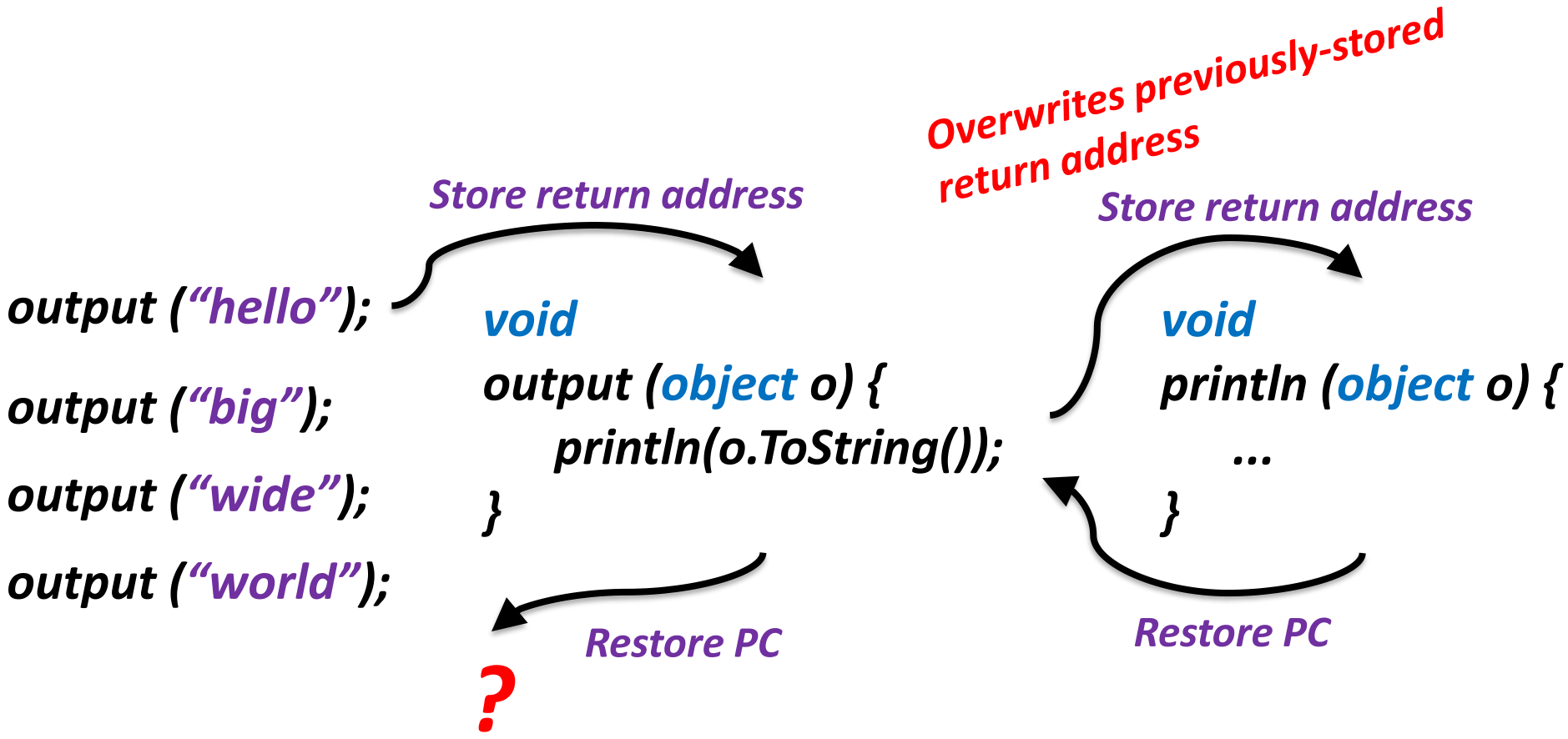
A subroutine is called from many places, so we can't simply "hard wire" a return address into the program – we must "remember" the return address each time we make each call

A simple solution...

1. Before making each call, the processor notes the value in the PC *(this is called the "return address" – it is the address of the next instruction in the calling routine)*
2. The processor jumps to the start of the subroutine and run it
3. At the end of the subroutine, the processor retrieves the previously-noted return address and restore it back to the PC
   ...the first routine will continue where it left off

- In MIPS, this is exactly what the **jal** ("jump and link") instruction does
  - **jal** C
    - Requires us (i.e. the program) previously to have put the return address in $31
    - Then, we do **jal** C to jump to the subroutine at address C
    - To return from the subroutine we do **jr** $31

A complication: subroutines are often *nested*



We've overwritten/ lost the first return address…
So, how can we store "earlier" return addresses so they don't get lost?

Solution: the stack

- Processors typically implement a stack
  - A stack is a ***Last In, First Out*** queue mapped to adjacent locations in memory (MEM[])
  - Think: plate stack in a restaurant

- The address of the **top** of the stack is remembered in a special register called the *stack pointer* **(SP)**

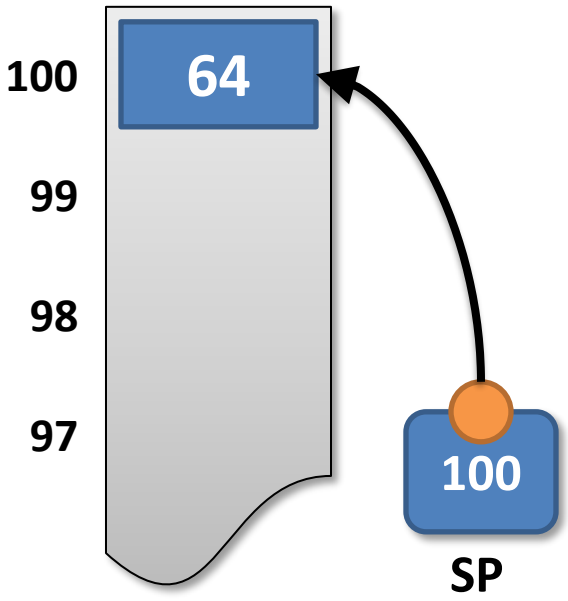
- Conceptually, there are two operations we can do on a stack: we can "***PUSH"*** values onto the stack, and "***POP"*** values off the stack

*(Many processors have explicit PUSH and POP instructions; MIPS, however, builds everything on top of **jal**—we have to explicit manipulate a register acting as the stack pointer)*

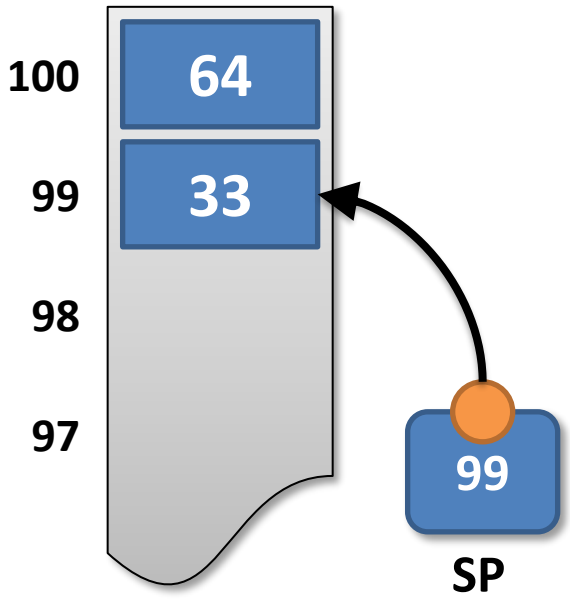
PUSH $s

We assume initially that SP = 101…

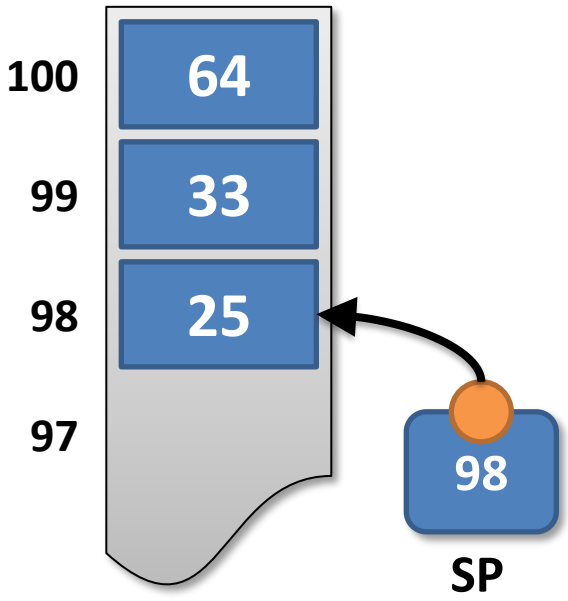
lw $s, 64
PUSH $s…



lw $s, 33
PUSH $s…

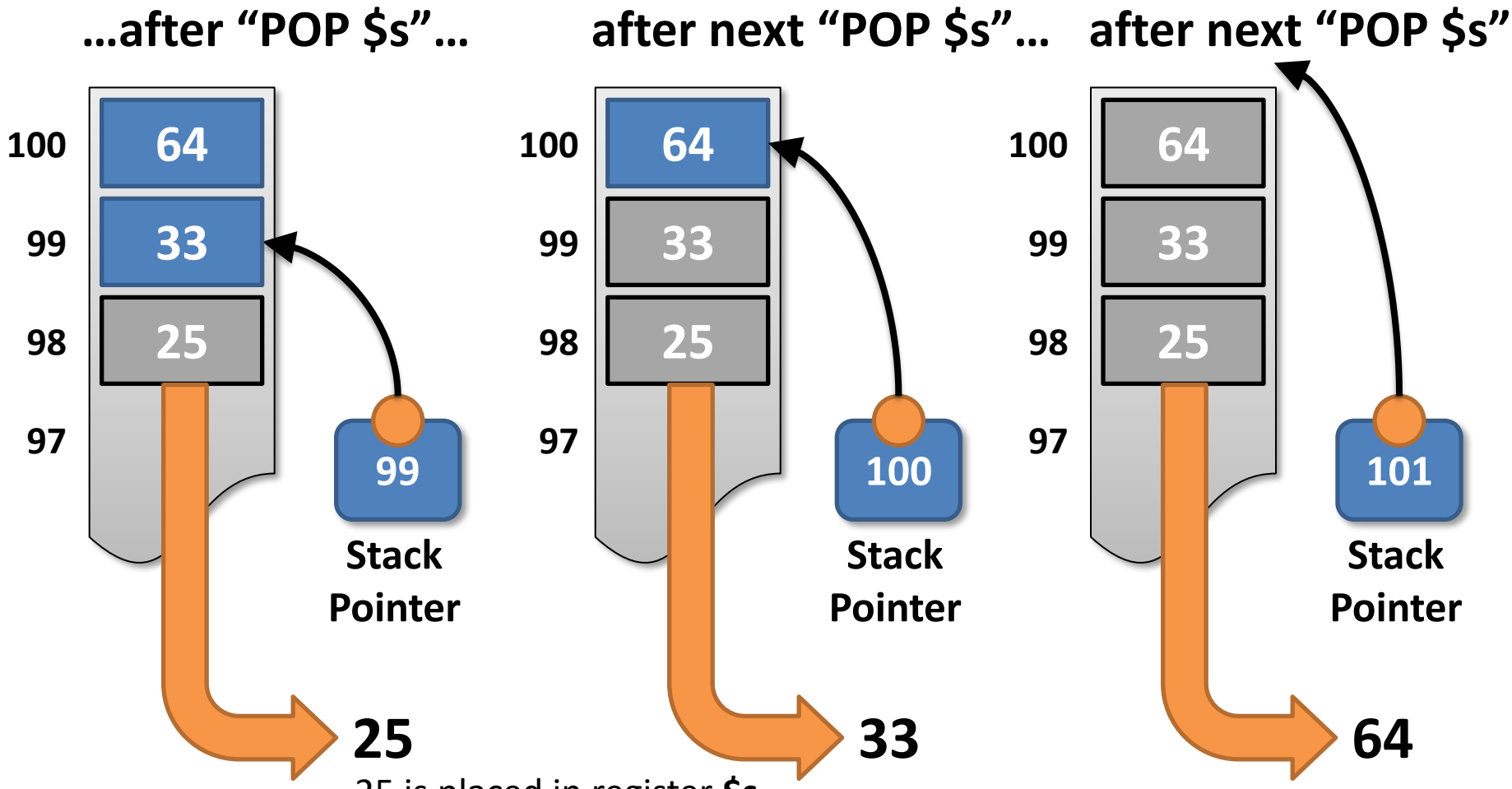


lw $s, 25
PUSH $s



Note that as we repeatedly PUSH, the stack grows **downwards** from high to low memory addresses!

POP $s

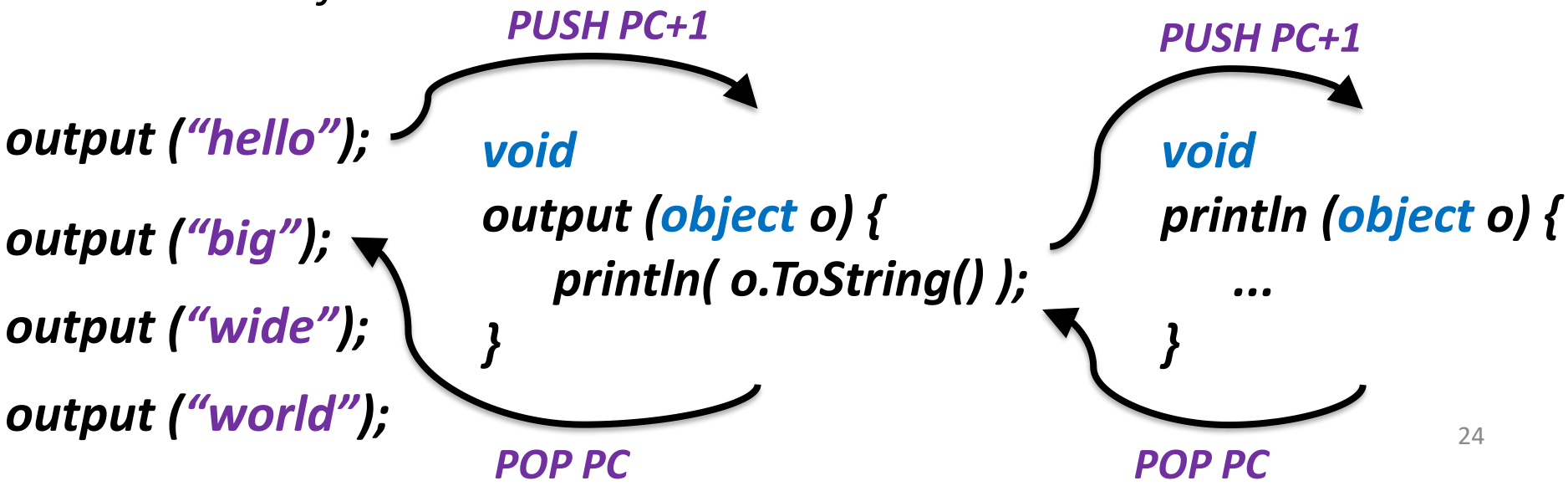


25 is placed in register **$s**…

Note that retrieved values are *not deleted*

Stack-based subroutine calls

- So, this is how we make stack-based subroutine calls
  1. PUSH PC + <*length of current instruction*> [i.e. push ret addr]
  2. Jump to start of subroutine S and run it
  3. At end of S: POP PC *...the initial subroutine will continue at the return address: the instruction immediately following the call of S*



Summary

- We know what an *instruction set architecture* is, and have seen some types/families of ISAs
- We understand the basic structure of the MIPS ISA (as a case study) which is in the RISC family
- We understand the processor's *fetch-decode-execute* cycle
- We understand that instructions are binary patterns in memory(!) – and are thus different from the textual representation we see in assembler programs
  - It's that old "representation" point again!
- We know what general purpose and special purpose registers are
- We understand how a program can manipulate the PC (directly and indirectly) to cause control transfers
- We understand how to handle nested subroutine calls by manipulating the stack (logical operations: PUSH and POP)