Edinburgh Napier UNIVERSITY

ALGORITHMS & DATA STRUCTURES SET08122

LECTURE 01: COMPUTER ARCHITECTURE

Dr Simon Wells
s.wells@napier.ac.uk
http://www.simonwells.org

TL/DR

- There are many Data Structures & Algorithms, as many as there are ways to organise & manipulate computer memory, but they are **specified** against an *abstract machine* and **optimised** for a *specific architecture/implementation*

Edinburgh Napier
UNIVERSITY

At the end of this lecture you will be able to:

- Construct a working definition for the terms "**data structure**" & **"algorithm"**
- Relate data structures to the von Neumann Architecture
- Compare data types
- Begin building a working understanding of data types & structures

OVERVIEW

1. Data Structures & Algorithms (preliminary definitions)
2. The von Neumann Architecture
3. Data & Memory
4. Primitive Datatypes
5. Composite & Linear data structures
6. Arrays & their limitations
7. The costs of Computation

DATA STRUCTURES & ALGORITHMS

- Programming is about manipulating information
- Information is stored as data
- How we store data affects how easily we can manipulate it
- **Data structures** are organised collections of data
  - How that organisation is achieved differentiates data structures
- **Algorithms** are recipes for manipulating data.
  - A series of operations that transform our initial data into a desired form
- *We'll look at data & data structures for now, & return to algorithms a little later*

DATA STRUCTURES

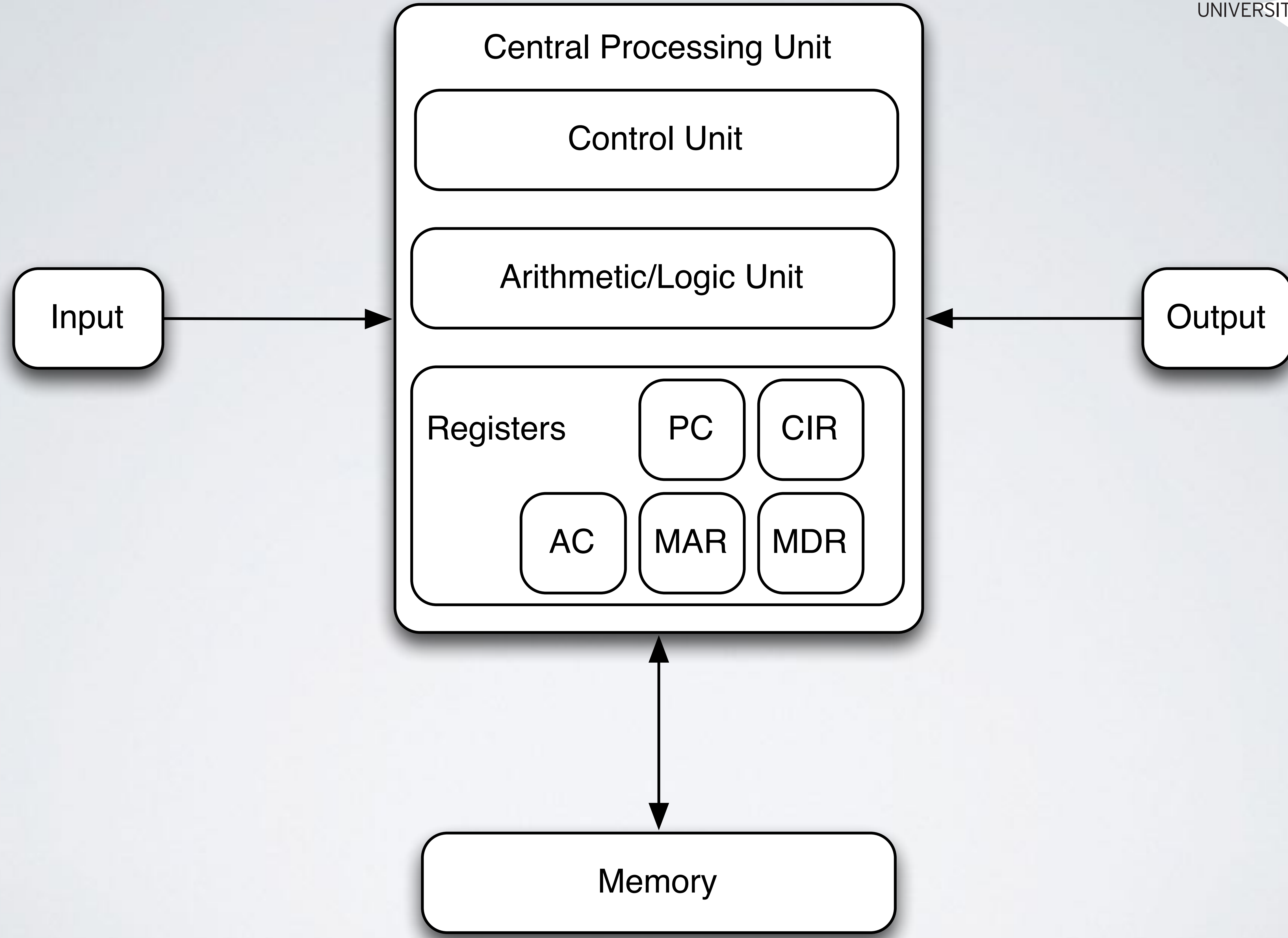
- Specific data structures are strongly associated with
  - The architecture & organisation of computational systems,
  - Language design decisions
- Wide range of ways to implement a given architecture
- Many language have similar, but not identical primitive datatypes

JOHN VON NEUMANN

- John von Neumann introduced what is now referred to as the von Neumann Architecture in 1945
- Based on the first draft of the EDVAC machine (one of the earliest electronic computers)
    - NB. There are other architectures, e.g. the Harvard Architecture & other names for von Neuman architecture, e.g Princeton Architecture
- The architecture came about due to hardware constraints of the time
- So simple, elegant, and powerful that it is still the basis for modern programming models
- Despite 60 years of technical innovations still an accurate abstraction of modern computer hardware

VON NEUMAN ARCHITECURE

- Design of a digital architecture made up of:
  - Central Processing Unit (CPU)
  - Arithmetic Logic Unit (ALU)
  - Processor Registers
  - Control Unit (containing instruction register & program counter)
  - Memory (storing data & instructions)
  - Mass storage
  - Input/Output (IO) Mechanisms
- All connected by a bus (communication pathways)



Edinburgh Napier
UNIVERSITY

VON NEUMANN BOTTLENECK

- A computer basically works by moving information (data stored in memory cells [e.g. processor registers, various levels of cache, RAM, Mass Storage] around and performing operations on it.
- Operations performed by looking in specific memory locations for inputs, performing operations, then writing result to an output
- Data is moved around on a **bus** (generic name for hardware that moves information)
- Moving information around has a cost
- There is limited bandwidth to move data from one location to another - if there is too much data then the limiting factor on speed is the bus - this is a known limitation of many computer architectures & is known as the **von Neumann Bottleneck**

SPECIFICATION VS OPTIMISATION

- Data Structures & Algorithms are **specified** against an *abstract machine* but **optimised** for a *specific architecture/ implementation*
  - *Consider the difference between describing an implementation of an array for an abstract machine versus implementing that array from scratch on a given piece of hardware, or adding an array implementation to an existing language*

STORING DATA IN MEMORY

- Data is stored in memory
- There are many types of hardware implementations of memory
- For analysis we often "idealise" the architecture & e.g. assume infinite memory (although we only use a finite amount at any given time)
- Generally, memory is made up of discrete cells which individually store data.
- Cells are arranged in a regular pattern and of fixed size (think of an array)
- Addressable - each cell can be uniquely referred to
- Each cell can store a word (usually assume an integer per cell)

DATA & TYPES

- Depending upon your language, you may have some mix of:
  - **primitives** [integers, characters, floats, doubles, strings, references (pointers/handles), &c.]
  - **composite types** [arrays, structs records, &c.]
  - **linear types** [arrays, lists, vectors, matrices]
  - **associative types** [dict, maps, sets, tuples]
  - **abstract data types** [Lists, Stacks, Queues, Deques, Trees, Graphs]
  - **concrete data types** (may or may not be identical to the abstract types)

PRAGMATISM & LANGUAGE DESIGN

- For many reasons, whilst a data structure has a theoretical shape the implementation must take account of practical & design issues:
  - Avoiding duplication (e.g. Python has lists but no more primitive collection like arrays)
  - Reusing existing data structures (the behaviour of stacks, queues, deques can all be achieved using the Python list methods)
  - Fitting with design of the language
  - Giving the coder sufficient syntactic sugar to make common tasks easy
  - Optimisation
- So sometimes we need to consider behaviour (practical), behaviour (ideal), performance tradeoffs between the two.

THE C PROGRAMMING LANGUAGE

- A relatively old language
- High level (compared with assembly)
- Low level (compared with Java, C#, Python, &c.)
- Compiled C runs directly on the CPU - No Virtual Machine
- Optimisation occurs at compile time - has advantages for this class
- Already met it in Programming Fundamentals
- (Historically) used for tasks that need performance or efficiency, e.g O.S.

C PRIMITIVES

- char (signed, unsigned) - usually 8 bits
- int (short, signed, unsigned, long, long long [c99]) - usually 16 bits
- float
- double (long)
- Actual size varies depending upon implementation (*perhaps we'll investigate this in the lab*…)
- Each primitive datatype has an associated **pointer** datatype

STRUCTS

- A **composite** type (in contrast to primitives)
- Aggregation of multiple (potentially differing) primitive datatypes…
- … into a single memory block…
- … that is referenced by a single variable
- Can contain pointers to other structures (used to build linked data structures [*we'll explore this over the next few weeks*])

SEQUENTIAL/LINEAR STRUCTURES

- A way to organise primitive datatypes in relation to each other - as various kinds of sequences
- We could use individual items of data (e.g. variables), but this quickly becomes problematic:
  - *which would you rather deal with a thousand individually named integer variables or a single array of thousand integers?*
  - Arrays, {single/doubly}, Vectors, Matrices, (Linked) Lists

ARRAYS

- Another composite datatype (in contrast to primitives)
- A **linear** data structure - the elements form a sequence
- Static data structures
- A collection of values of the same type
- Stored contiguously in memory
- Can build an array of most primitive datatypes (including structs)

ARRAY LIMITATIONS

- Programming languages, particularly higher level ones, generally implemented using lower level languages (some get to the point where they are self-hosted)
  - Some weirdness when you get to higher-level interpreted languages like Java, C#, Python due to VM. Language itself *may* be self hosted, but the VM is written in a lower level language.
- Often used within the implementation of more advanced data structures
- Overcoming the drawbacks of the basic, fixed size array is a spur to development of many other structures:
  - e.g. ArrayList (Java), Lists (Python), Dynamic arrays in general
    - Desire to store an unknown amount of data without having to deal with the details of increasing the size of the array whenever it runs out of room
    - Always want more space for new data, but don't want to pre-allocate space unless needed.

STANDARD LIBRARIES

- Also known as builtins, frameworks, &c.
- C has the C Standard Library (which unfortunately doesn't add new data structures but there are other libraries)
- Java has the Collections framework (standard with most distributions, import the library, occasionally implement an interface, sometimes a ready made class)
- Python has a bunch of powerful data structures as standard (available without importing additional libraries)
- In all cases, specialist data structures are available in specific libraries to solve particular problems, e.g. Pandas & Sci-Py are data analysis Python libraries which provide specialist structures.

ALGORITHMS

- An *unambiguous specification for how to solve a problem*
- Can be used to do computation, process data, calculate results, reason, bake a cake, service a car, etc. etc.
- *Effective* algorithms are expressed within a finite time & space using a well-defined language

    1. **Start with an initial state & (possibly empty) input...**
    2. **... proceed through a finite number of intermediate states by executing instructions...**
    3. **... to produce an output and termination state.**

- Many programs are also algorithms under this definition (if they terminate)
- We'll return to this topic repeatedly over the next few weeks and build a more complete understanding

COSTS OF COMPUTATION

- There are costs involved in computation:
  - Storing data uses memory
  - Finding data uses CPU
  - Moving data around uses CPU & memory
- All take **time** (abstraction from CPU usage)
- All use **space** (abstraction from memory usage)
- Data Structures & Algorithms is concerned with evaluating & trading off between time & space usage
- We'll return repeatedly to the topic of time & space calculations (complexity) over the next few weeks

SUMMARY

- We started with a simple working definition of the terms "Data Structure" & "Algorithm" which we will elaborate as we progress
- We considered the von Neumann architecture as an abstract model of computation
- We then considered real world machines & programming language
- We spent some time considering the C programming language, how it aligns with abstract and real-world models of computation, and the facilities that C gives us for representing data, and how they can be elaborated upon to produce different data structures
- Finally we began the process of considering the costs of computation which will eventually build into some tools for reasoning about both abstract and real world performance.

QUESTIONS ???

WHAT DID WE LEARN?

- *We can now…*

- Construct a working definition for the terms “**data structure” & “algorithm”**
- Relate data structures to the von Neumann Architecture
- Compare data types
- Begin building a working understanding of data types & structures