Computational Thinking

Topic 1

What is computer hardware?

1.2 Introduction

In short, computer hardware is the physical 'nuts and bolts' of a computer; it's the parts of a computer that you can touch (and that break if you drop them!). However, the notion of 'a computer' means different things to different people: to you it might be a PC (that sits on your desktop); to Cray or IBM (makers of supercomputers, like the Cray Jaguar and the IBM Blue Gene) it might be a machine that fills a room; to the guys at, for example, the Berkeley Quantum Information and Computation Center in the USA, who are trying to build a quantum computer (a computer that operates according to the laws of quantum mechanics) it might be a tangle of lasers, wires, glass tubes and so on; to Dynamics Inc. (a Silicon Valley company specialising in intelligent advance payment cards) it might be an embedded chip; and to those involved in SETI@home (an experiment that uses Internet-connected computers to search for extra-terrestrial life) it might be the Internet itself!

For us, computer hardware will be the physical PC that sits on our desks. We're going to open one up and take a closer look at how it works; in particular, at how the central processing unit (the CPU; the heart of the PC) enables the computer to do an incredible array of things as prompted by the particular computer program (or one of them) it happens to be running at the time.

We undertake a rapid overview of computer hardware, beginning by opening up a PC and examining its motherboard, on which the CPU sits, before looking at the CPU and its fabrication process in more detail. We'll delve even deeper into the CPU and examine its building blocks, namely transistors and the logic gates they go to build, and we'll see how we can build some useful circuits out of NOT-, AND- and OR-gates, such as a half-adder and a full-adder. The design of integrated circuits is incredibly complex and we'll describe the basic process and how there are (programming) languages that assist the designer. We'll look at the microarchitecture of the CPU and its fundamental components, before examining the most basic computer

architecture, namely the von Neumann architecture. After looking at fundamental aspects of memory we'll examine the fetch-decode-fetch-execute cycle and see how a basic processor computes at the (low) level of hardware.

1.3 Let's open up a PC!

When we remove the 'lid' of our typical PC, we generally see pretty much the same thing: a **motherboard** upon which are housed an array of different electronic components; a fan (which actually covers the PC's CPU and provides some cooling relief to it when it is working hard); and some wires leading from the motherboard to 'metallic boxes' which actually provide the interface between the PC and the outside world (via the monitor, the keyboard, the mouse, the Internet connection, the disk drives and so on).

We're going to be interested in the CPU and so let's take a closer look at the motherboard on which it is housed. Actually, there is more than one processor in a PC. The motherboard is traditionally partitioned according to the CPU, the northbridge and the southbridge. In short, the **northbridge** is the **chipset** (that is, set of mini-processors) that handles (fast) communications amongst the CPU, the memory and the graphics processing unit (GPU), whilst the **southbridge** is the chipset that handles (slower) communications involving external hard-disks, the mouse, the keyboard, the Internet, the printer and other such devices. However, nowadays the northbridge is increasingly part of the CPU itself. Communication within the CPU is undertaken using a variety of **buses**.

1.4 Silicon chips

Let's tunnel further into the motherboard and take a closer look at the CPU and how it is constructed. Our CPU (or **silicon chip** or simply **processor**) is actually an **integrated circuit** (**IC**), which consists of millions of **transistors** interconnected by microscopic wires on a footprint of around 1 cm^2 (transistors are the building blocks of electronic circuits). The incredible complexity of an IC can only really be appreciated by looking at it using a scanning electron microscope.

Having designed an IC (which, given the number of components in the IC and the miniscule footprint of an IC, is an extremely complex task), the IC is **etched** onto a wafer of silicon (the chemical and electrical semiconductor properties of silicon make it ideal for building transistors). Initially (and,

in part, in order to counter errors in the production process), hundreds of copies of the same IC are etched on a wafer of silicon with each copy called a **die**. Dies are tested and each error-free die is cut and **mounted** in a **package** with the die's **pads** connected to the package **pins**. Essentially, what results is the silicon chip encased (or mounted) in a package with the external contacts of the IC (the pads) connected to pins in the package to enable direct contact with other components within the PC.

The general process of moving from silicon ingot to chip, as depicted in Fig. 1, is error-prone and calls for continual error-testing throughout the process, with some dies not making it through to final package.

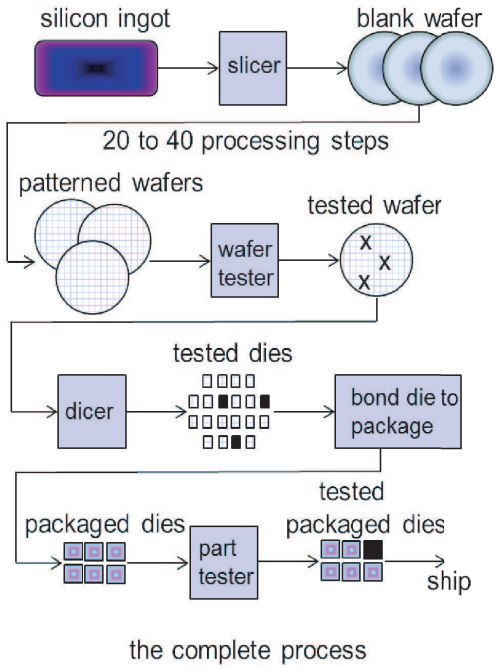

Fig. 1: from silicon ingot to chip.

A modern phenomenon is the **multi-core processor**. Here, multiple independent **cores** (CPUs) are manufactured on the *same* IC but these cores share things like main memory. At present (because of technological limitations), only a relatively small number of cores have been incorporated within the same IC and used in PCs and laptops (typically 2 or 4). However, the 10-core Xeon Westmere-EX has 2.6 billion transistors and future technological advances will result in multi-core processors with hundreds of cores (so-called

many-core processors). Whilst it will be tremendous to have this extra computing power available, best utilising this power will mean programmers having to move from 'thinking sequentially' to 'thinking in parallel'. This is a significant paradigm shift and calls for a very different approach to general problem solving than has hitherto been the case!

'processing in parallel'

1.5 Moore's Law

If we look at how many transistors computer manufacturers have managed to cram into integrated circuits, we can see that over the years this number has followed **Moore's Law**, a 'law' coined in 1965 by Gordon Moore (a co-founder of Intel) which says: '*Transistor capacity doubles every* 18-24 *months*.' Of course, this 'law' really just provides a rough description of a long-term trend.

There are physical limitations as regards Moore's Law and there is currently much debate as to how long Moore's Law will hold good. In 2005 Gordon Moore said: '*In terms of size* [*of transistors*] *you can see that we're approaching the size of atoms which is a fundamental barrier, but it'll be two or three generations before we get that far - but that's as far out as we've ever been able to see. We have another* 10 *to* 20 *years before we reach a fundamental limit. By then they'll be able to make bigger chips and have transistor budgets in the billions.*'

Power dissipation is also a problem, where the power is proportional to the number of transistors switched multiplied by the frequency of switching (that is, the **clock speed** of the CPU). The problem with CPU power dissipation has driven the growth of multi-core processors where one CPU with a high clock-speed has been replaced with a number of CPUs with lower clock-speeds but which, when working together, can give better computational power (well, that's the theory anyway!).

1.6 Transistors

Given that transistors are key building blocks of ICs, let's take a closer look at them. A **transistor** is a semiconductor device used to amplify and switch

electronic signals. They exist in just about all electronic equipment but we're only concerned with transistors appearing in integrated circuits.

Regardless of where they are used, they have the same fundamental operation. Every transistor has: a **collector** from which electricity can flow; an **emitter** to which electricity can flow; and a **base** to which a voltage can be applied (so that it is 'on' or 'off').

There are two essential types of transistor.

- In an **NPN transistor**, if the base voltage is 'on' then current can flow; otherwise, there is no current flow.
- In a **PNP transistor**, if the base voltage is 'off' then current can flow; otherwise, there is no current flow.

1.7 Gates

Current technology is building transistors of size getting towards the atomic level, with new nanotechnology, such as quantum dots, carbon nanotubes and even bio-circuitry, promising even smaller units (such new technology is still in its infancy and will require much more research before making it into the mainstream). Transistors can be composed together to build (**logic**) **gates**; that is, very small circuits that take **Boolean** (or **binary**) inputs, namely 0 and 1, interpreted as there being a low or a high voltage, respectively, and produce a Boolean output (or outputs).

From a logical perspective, there are three fundamental gates:

- the **NOT-gate**:
  - input X, output Z
  - $X = 1$ if, and only if, $Z = 0$.
- the **AND-gate**:
  - inputs X, Y, output Z
  - $Z = 1$ if, and only if, $X = 1$ and $Y = 1$.
- the **OR-gate**:
  - inputs X, Y, output Z

– $Z = 0$ if, and only if, $X = 0$ and $Y = 0$.

Icons used for these logic gates can be visualized in Fig. 2. We can build a NOT-gate using 2 transistors, and an AND-gate and an OR-gate using 6 transistors (actually, a NOT-gate can be built using just one transistor but the set-up suffers from deficiencies as regards power consumption and processing speed).

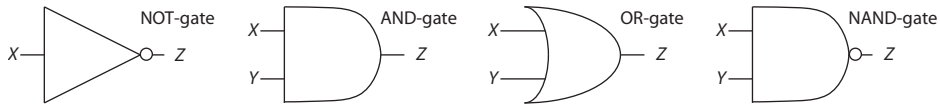

Fig. 2: icons used to depict logic gates.

While the NOT-, AND- and OR-gates are the fundamental gates from a logical perspective (as we shall see in a moment), when building gates electronically the NAND-gate can be thought of as the fundamental logic gate. The NAND-gate is such that the output is 0 if, and only if, both the inputs are 1 (the icon for a NAND-gate can be visualized in Fig. 2 too). The NAND-gate can be built out of just 2 transistors, and it is a simple exercise to show that NOT-, AND- and OR-gates can be built out of NAND-gates. For example, a NOT-gate is just a NAND-gate where the input to the NOT-gate is duplicated; an AND-gate is just a NAND-gate composed with a NOT-gate; and an OR-gate is just a NAND-gate where the two inputs are prefixed with a NOT-gate.

Irrespective of which logic gates we choose to use, we can think of a silicon chip as a massively complex circuit including gates and wires, and where there are only two input and output values: 0 and 1.

1.8 A binary world

If we are given *any* function $f(X_1, X_2, \ldots, X_n) : \{0,1\}^n \to \{0,1\}$ then using only NOT-, AND- and OR-gates, we can build a circuit that computes f.

For example, for any $n \geq 1$, we can build a circuit that has:

- inputs $X_n, X_{n-1}, \ldots, X_1, Y_n, Y_{n-1}, \ldots, Y_1$; and
- outputs $Z_{n+1}, Z_n, \ldots, Z_2, Z_1$

and is such that the sum of the two base-2 strings $X_n, X_{n-1}, \ldots, X_1$ and $Y_n, Y_{n-1}, \ldots, Y_1$ is computed. For example, when $n = 3$ our circuit will undertake binary computations such as the following:

$$\begin{array}{cccccccc} & X_3 & X_2 & X_1 & = & & 0 & 1 & 1 \\ & Y_3 & Y_2 & Y_1 & = & & 1 & 1 & 0 \\ \hline Z_4 & Z_3 & Z_2 & Z_1 & = & 1 & 0 & 0 & 1 \end{array}$$

We'll build our circuit in the case $n = 3$. We begin by building a **half-adder**. This circuit takes x and y as inputs and computes the Boolean sum of x and y, namely $s(x, y)$, where the Boolean sum of a collection of inputs is 1 if, and only if, an odd number of the inputs are 1, and also the resulting carry bit $c(x, y)$, which is 1 if, and only if, both x and y are 1. The half-adder can be visualised in Fig. 3.

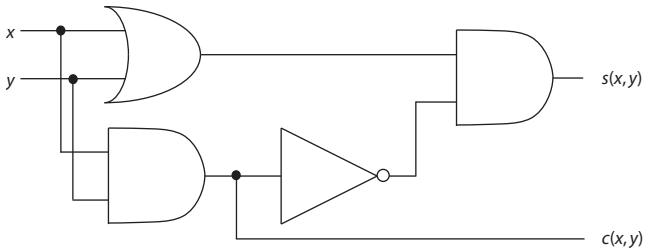

Fig. 3: a half-adder.

We now use the half-adder to build a **full-adder** which takes inputs x, y and c and computes the Boolean sum, s, of these 3 inputs together with a carry bit d, which is 1 if, and only if, at least 2 of the inputs are 1. The full-adder can be visualised in Fig. 4. So, the full-adder takes 3 bits as input and outputs their Boolean sum s along with their carry bit d.

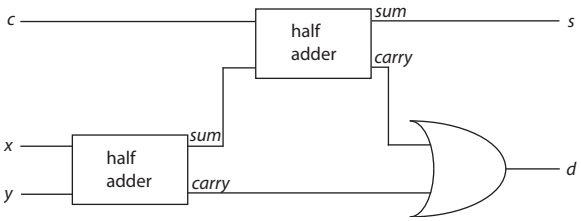

Fig. 4: a full-adder.

Now we use both the half-adder and the full-adder to build our circuit. Our circuit (or its design, at least) can be visualised in Fig. 5.

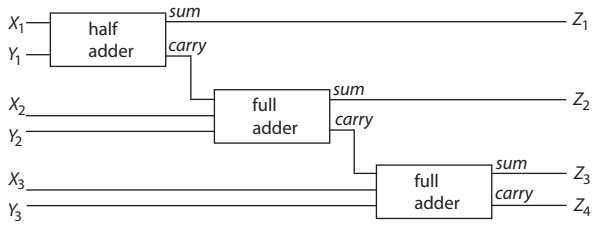

Fig. 5: a circuit to add two 3-bit numbers.

Follow through some inputs and convince yourself that the circuit works! You should be able to amend the above construction so as to build a circuit that adds 2 n-bit numbers for *any* fixed $n \geq 1$.

'using abstraction and decomposition in tackling a large complex task'

1.9 Integrated circuit design

Integrated circuit design is deciding how a system's components will be interconnected on a chip in order to build a CPU. As might be expected, the whole process of IC design is incredibly complex and fraught with difficulties. There are essentially three phases to IC design.

- In the first phase, a **functional specification** of the chip is derived. This functional specification describes exactly *what* the IC is supposed to do. Factors influencing the functional specification include, for example, chip area, power, speed and cost.
- The next stage involves the **register transfer level** (**RTL**) design. Here, the functional specification is used to describe the *exact behaviour* of the digital circuits on the chip, as well as the interconnections to inputs and outputs, with this description being in terms of logic gates and interconnecting wires. Testing is undertaken to ensure that this logical design conforms to the functional specification. The logical design is then converted into an electronic circuit design, involving transistors and so on, with yet more testing undertaken (especially as regards timing considerations, where different delays in physical components can cause unwanted effects if not catered for).

- The final stage involves the mapping of the RTL design to a **physical layout** in silicon using **computer-aided design** (**CAD**) tools. Specific physical attributes must be respected; for example, it is crucial that appropriate spacing between transistors is maintained in the physical layout. Testing is continual with testing in this phase undertaken to ensure that the physical layout conforms to the RTL design.

'using abstraction and decomposition in tackling a large complex task'

To assist with testing (in all phases but especially as regards ensuring that the RTL design meets the function specification), formal methods are used. **Formal methods** encompasses a wide spectrum of research within Computer Science where the general aim is to *mathematically prove* properties of designs, programs and so on, so as not just to rely on empirical testing. Within formal methods lie **model checking** and **computer-aided verification** (these topics are especially relevant to the design of ICs). Here, the aim is to improve the quality of digital systems by using logical reasoning, supported by software tools, to analyse their designs. The idea is to build a mathematical model of a system and then try to prove properties of it that validate the system's correctness - or at least help discover subtle bugs. The proofs can be millions of lines long, so specially-designed computer programs are used to search and check these proofs.

There is a lot to be gained by adopting formal methods, especially within the chip manufacturing industry. The **Pentium FDIV bug** (1994) was a bug in the Intel P5 Pentium **floating point unit**, which is an IC, also called the **maths co-processor**, that deals with arithmetic on floating-point numbers, where **floating-point numbers** are representations of real numbers within a computer and, as such, are sometimes necessarily approximations of the actual numbers involved. This bug was disastrous for Intel. Certain floating point division operations were observed to produce incorrect results and, when investigated, Intel announced that 5 out of 1066 rows in a particular lookup table used by the divide operation were incorrect. Intel was forced to offer to replace, for free, every chip sold until they could fix the bug, at a cost of $475 million! If only they had used formal methods to check their designs! Events such as the Pentium bug helped to establish the use of formal methods across many industries.

1.10 Hardware description languages

A **hardware description language** (**HDL**) is a language whose purpose is the formal description of digital logic circuits. It describes the behavior of an electronic circuit or system, that is, how it operates, as well as its design and organisation, from which the actual physical circuit or system can then be attained. The principal feature of an HDL is that it enables the function of hardware to be described without building an actual implementation (which could be costly, especially if errors are subsequently discovered). As well as a circuit's behaviour, design and organisation, an HDL can, in addition, describe specific tests to verify the circuit's operation by means of simulation.

In contrast to a normal programming language, an HDL includes explicit notations for expressing **time** and **concurrency**, which are primary attributes of computer hardware. An HDL allows a programmer to write an **executable specification** which can then be used by a simulation program to assist with testing. Thus, a piece of hardware can be modelled (in software) and tested before being physically created (which is costly). There are a number of hardware description languages including: Verilog; VHDL; RHDL (based on Ruby, a general-purpose programming language similar in ethos to Python); and MyHDL (based on Python).

1.11 Pause: From a chip to a fully working computer

[Video]

From 'NakedScientists' on *YouTube* (5:04 minutes):

`http://www.youtube.com/watch?v=gkg_7y-dwJI`

1.12 The Pentium 4 processor microarchitecture

Let's have a closer look at a chip. In Fig. 6, we can see the **microarchitecture** of the Pentium 4 processor; that is, the organisation of the different components of the processor. It is not important that we fully understand every aspect of the chip but there are some key components.

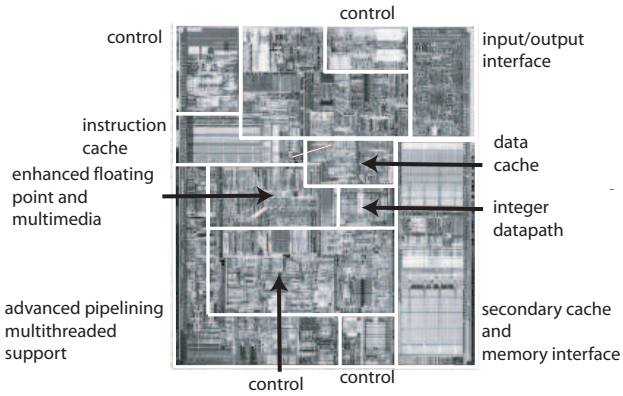

Fig. 6: Pentium 4 microarchitecture.

- The **datapath** performs all the data processing operations (it is the 'brawn' of the processor) and includes the **arithmetic logic unit** (**ALU**), which is the part of the CPU that performs all the arithmetic and logical operations on data. The datapath includes (a limited number of) memory locations called **registers** (we'll say more about registers later).

- The **control** tells the datapath, memory and input/output (I/O) devices what to do (it is the 'brains' of the processor). Essentially, the control is the conduit between the datapath and the main memory.

- A **cache** consists of small, fast and relatively expensive on-chip memory. Data caches are used to store memory items that need to be regularly accessed (sometimes there is more than one such cache). Similarly, instruction caches are used to store instructions that need to be regularly executed (we'll say more about caches later).

1.13 von Neumann architecture

The **von Neumann architecture** is a fundamental computer architecture (also known as the **stored program architecture**). It was proposed by the pioneering computer scientist John von Neumann in 1945. A key aspect of the von Neumann architecture is that the stored program and the data are both held in memory, which allows for self-modifying programs (of course, programming errors might result in a program obliterating itself or modifying itself in unintended ways!). The basic von Neumann architecture can be

visualized as in Fig. 7(i) (modern computer architectures are more complex than this). Although not depicted separately, there is actually a **data bus**, an **address bus** and a **control bus** as conduits between the processor and main memory, where a **bus** is simply a communications device that can be thought of as a series of 'parallel wires' down which data can flow, with the number of these 'wires' being the **width** of the bus (we'll see these buses in more detail soon). Although there are three distinct buses, program instructions and data items are given the same priority and have to be accessed sequentially.

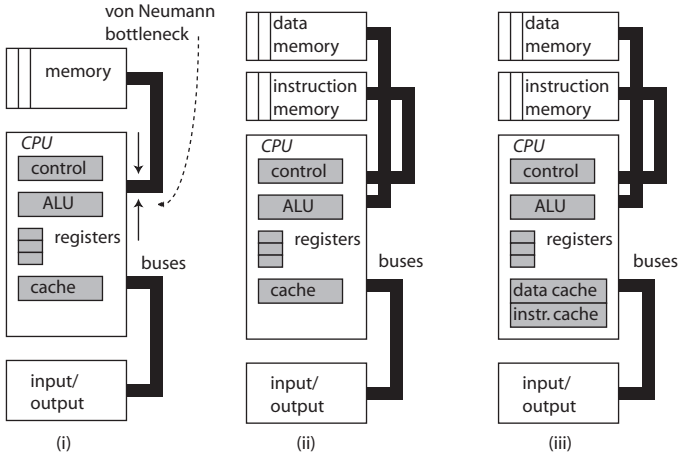

Fig. 7: von Neumann, Harvard and modified Harvard architectures.

What can result in the von Neumann architecture is the **von Neumann bottleneck**; that is, a limitation in the data transfer rate between the CPU and memory (as stated above, data and instructions have to be fetched in sequential order and the CPU spends a considerable time idle whilst waiting for data items or instructions to be fetched from memory). The von Neumann bottleneck gave rise to the use of caches.

'prefetching and caching in anticipation of future use'

The **Harvard architecture**, where memory is partitioned into data memory and instruction memory with dedicated buses for each of them, does not suffer from the von Neumann bottleneck. Now, program instructions and data items can be fetched simultaneously. The Harvard architecture can be visualised as in Fig. 7(ii).

Most modern computers have a **modified Harvard architecture** where although data memory and instruction memory are separated, each with their own dedicated bus, the processor allows data to be treated as instructions and instructions to be treated as data; this allows for self-modifying or self-optimising code. In addition, there are separate CPU caches for instructions and for data. The modified Harvard architecture can be visualised as in Fig. 7(*iii*). In fact, modern computer architectures are considerably more complicated than those depicted here; however, our architectures suffice to describe some fundamental concepts.

1.14 Memory

Memory is essentially a set of 'pigeon-holes' into which data can be placed. It comes in various forms (and, as we'll see, to some extent is characterized by how far it is away from the CPU). Each 'pigeon-hole' has a unique **address** (a natural number) and holds 1 **byte** of storage (which consists of 8 **bits**, where a bit is either 0 or 1). Larger words can be formed by dealing with batches of bytes, with a **word** being the amount of data that can be handled by a processor as one unit. Every computer architecture has an associated **word-size**, with the word-size chosen being a design choice (my laptop happens to have a word-size of 32 bits, or 4 bytes).

As we have heard, within the processor architecture there are (at least) 3 buses.

- The **address bus** holds addresses of locations in main memory. The width of the address bus determines the size of addressable memory. For example, an address bus that is 32 bits wide can carry at most 2^{32} different addresses. So, as each memory address holds 1 byte then addressable memory has size 4 Gbytes.

- The **data bus** carries the contents of memory locations. The width of the data bus determines the word-size of the computer. Note that although each memory location holds 1 byte of data, a computer with a word-size of 32 bits, for example, can be considered to hold 32 bits of data in each location, for the contents of some 'location' are considered to consist of the contents of 4 contiguous bytes of memory.

- The **control bus** is used to transfer information between the CPU and various other devices within the processor. Typical information

includes signals saying whether data is being read or written and detailing information transmitted from input/output devices.

1.15 More on memory

The cost and performance of memory is generally proportional to its physical distance from the CPU.

- The **hard disk** usually consists of rapidly rotating disks with magnetic heads (although solid state drives, which have no moving parts, are becoming more common). Hard disks are used for secondary storage and tend to be cheap but relatively slow.
- **Random Access Memory** (**RAM**) takes the form of an integrated circuit and comes in two basic types.
  - **Dynamic RAM** (**DRAM**) is where a bit of data is stored using a transistor/capacitor combination. It is relatively cheap although it needs to be refreshed (because capacitors tend to 'leak') and is slow.
  - **Static RAM** (**SRAM**) is where a bit of data is stored by a **flip-flop**, which incorporates 4-6 transistors. It is stable and fast although takes up more space.
- **Caches** are expensive memory that are used to store rapidly accessed items. They are generally organised hierarchically into levels. A **level 1 cache** usually consists of static RAM and is built into the CPU. A **level 2 cache** again usually consists of static RAM and might be either built into the CPU or be at a short physical distance away.

1.16 Memory, registers and datapath in a (very) simple von Neumann processor

A *basic* von Neumann processor might be viewed architecturally as in Fig. 8 (nowadays, computer architectures can be significantly more complicated although they still possess some of the fundamental features of the von Neumann architecture).

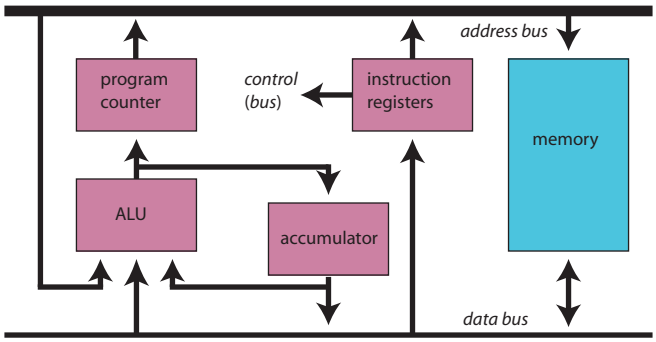

Fig. 8: a basic processor architecture.

- **Registers** are on-chip memory locations that are limited in number (the actual number results from a design decision) and are the 'bricks' of computer construction. Processor registers are at the top of the memory hierarchy and provide the fastest way to access data.
- The **accumulator** is a special register which temporarily stores the results of arithmetic and logical operations (on data in registers).
- The **program counter** is a special register that holds the memory address where the next instruction can be found.

Basic processor operation is via the **fetch-decode-fetch-execute** cycle (repeated *ad infinitum*!):

- the 'instruction fetch' phase involves the supply of the instruction address (via the address bus) and the return from memory (via the data bus) of the instruction;
- next, the 'instruction decode' phase involves interpreting the stored instruction within the CPU;
- next, the 'operand fetch' phase involves the supply of the address of any required data (via the address bus) and the return from memory (via the data bus) of this data; and finally
- the 'execute instruction' phase involves the CPU performing the necessary actions (this phase is sometimes split into two with the 'execute instruction' phase followed by a 'write-back' phase where data is written back to memory, if needs be).

'interpreting code as data and data as code'

1.17 Instruction fetch

The CPU outputs the value of the program counter (the address of the next instruction to be executed) on the address bus. Memory reads the address bus and puts the contents of the particular memory address onto the data bus; that is, the instruction to be executed. The actual instruction is stored in instruction registers within the CPU.

[Animation]

1.18 Instruction decode

The instruction word stored in the instruction registers is decoded by the internal logic to provide control signals to the ALU and to other internal circuits inside the CPU. These control signals essentially tell the ALU what to do in order to execute this instruction. The program counter value is pushed onto the address bus. The ALU (having read the program counter from the address bus) increments this value by the word-size (in bytes) and puts it back into the program counter; so, the program counter now contains the address in memory of the next instruction to be executed (or so it thinks). However, and as we'll see, the value in the program counter might change.

[Animation]

1.19 Operand fetch

The instruction registers provide the address of the data to be processed to the address bus; that is, the operand address. Memory supplies the operand data to the data bus and thus to the CPU, ready for processing by the ALU.

[Animation]

1.20 Execute instruction

Processing is performed on the operand by the ALU, according to the instruction, and the result (if any) is put back into the accumulator. Also, it might be the case that the program counter is updated; for the instruction just executed might have been a branch instruction and the next instruction might actually be in a different memory location to that expected.

[Animation]

1.21 Write-back (if necessary)

The result from the accumulator is written back into memory (via the data bus). In many processors, the write-back is separate from the execute.

[Animation]

1.22 Layers of abstraction in modern computer systems

We can abstract a modern computer system in layers as shown in Fig. 9. This topic has provided a very brief look at the bottom five layers.

'using abstraction and decomposition in tackling a large complex task'

1.23 Research glimpse: Multi-core Computing

A **multi-core computer** is a computer where more than one processor (typically) is integrated on one IC. The architecture can be: **shared-memory**, where different processors communicate with one another by writing and reading messages left in shared-memory locations; or **distributed-memory**, where processors are connected according to some **interconnection network** (e.g., as a ring or a mesh) and communicate by message-passing.

Multi-core computing allows for the use of more and slower processors, with a consequent drop in heat generation and power consumption in comparison with one fast, high-power processor. However, brand new 'parallel

architectures' need to be designed and there is a massive impact upon a range of aspects of computer architecture including cache organisation, programming, performance analysis, task scheduling and synchronization.

Intel's **Teraflops Research Chip** (**Polaris**) has 80 cores connected by an 8×10 mesh, and its own programming language called Ct.

'processing in parallel'

1.24 Research glimpse: GPGPU Computing

General purpose graphics processing units (**GPGPUs**) have traditionally been used to deal with aspects of graphics with, essentially, one thread dealing with one screen pixel. However, these threads can be used as computational threads! That is, rather than deal with particular pixels in the screen they can simply be used as computational devices. What results is a massively parallel desk-top computer.

Whilst GPGPUs are shared-memory devices, programming such a shared-memory device can be extremely difficult (poorly designed programs can be slower rather than faster than sequential ones!). Nevertheless, there is currently much interest from the users of high-performance computers, such as Mathematicians, Physicists, Chemists and other scientists, who perform massive 'number crunching' computations and who have, up until now, had to buy time on extremely expensive and geographically distant supercomputers in order to undertake their calculations.

Research issues include the acceleration of applications, developing programmers' toolkits, designing useful and efficient GPGPU algorithms and clustering GPGPUs (into 'parallel parallel' machines!). At present, devising efficient GPGPU programs that make best use of the underlying hardware is more of an art than a science!

'processing in parallel'

References

The books below all include introductory material as regards basic aspects of computer architecture and computer systems.

[1] M. Abd-El-Barr and H. El-Rewini, *Fundamentals of Computer Organization and Architecture*, Wiley (2005) [004.22 ABD].

[2] A.S. Berger, *Hardware and Computer Organization*: *The Software Perspective*, Newnes (2005).

[3] N. Dale and J. Lewis, *Computer Science Illuminated* (4th edition), Jones and Bartlett (2009).

[4] I. Englander, *The Architecture of Computer Hardware, Systems Software, and Networking*: *An Information Technology Approach* (4th edition), Wiley (2010) [004.22 ENG].

[5] D.M. Harris and S.L. Harris, *Digital Design and Computer Architecture*, Morgan Kaufmann (2007).

[6] M.M. Mano, *Digital Design* (4th edition), Prentice Hall (2006) [621.3815 MAN; 2nd edition 2002].

[7] D.A. Patterson and J.L. Hennessy, *Computer Organization and Design*: *The Hardware/Software Interface* (4th edition), Morgan Kaufmann (2008) [e-book available].

[8] W. Stallings, *Computer Organization and Architecture*: *Designing for Performance* (8th edition), Pearson (2010) [004.22 STA].

[9] A.S. Tanenbaum, *Structured Computer Organization* (5th edition), Pearson (2006) [005.1 TAN].

[10] J.F. Wakerly, *Digital Design*: *Principles and Practices* (3rd edition), Prentice Hall (1999) [621.395 WAK; 1st edition 1990].