Computational Thinking

Topic 3

How does hardware execute software?

3.2 Introduction

As to what constitutes software and what constitutes hardware has been a changing landscape over the years. Nevertheless, a constant within this landscape has been the abstraction linking algorithm and machine. At the top end has been an algorithm, implemented in a variety of programming languages, each aimed at making life for the programmer as easy as possible, and at the bottom end has been collections of transistors, gates, and so on, through which the algorithm is ultimately brought to life (or even cogs and wheels if one goes back far enough!).

In what follows we'll look at the final bridge between top and bottom; that is, what emerges from the compiler or translator, as studied in Topic 2; and the electronic circuits, as studied in Topic 1. We'll also look at the internal 'software organisation' of the computer as it tries to hide essential actions and other activities from the computer users yet still facilitate the things the users want to do; namely, the operating system. We'll start by looking at the instruction set architecture (ISA) and its primary component: its assembly language. We'll look at a few instructions from the MIPS ISA and see how a high-level C program might be compiled into an assembly program. Whilst an assembly program is low-level, it is still not suitable for execution by the CPU; so we'll see how MIPS instructions are encoded as machine language for CPU execution. Now that we've gone all the way from a high-level program to machine code, we'll turn to looking at how the operating system manages all the activities going on within a computer. We'll take a look at the many different functions of the operating system and some key aspects of operating systems, namely processes, virtual memory, process control blocks and context switching.

3.3 Layers of abstraction in modern computer systems

Recall our earlier abstraction of a modern computer system, as visualized in Fig. 1. Our focus in what follows is on the levels labelled 'instruction set architecture (ISA)' and 'operating system'.

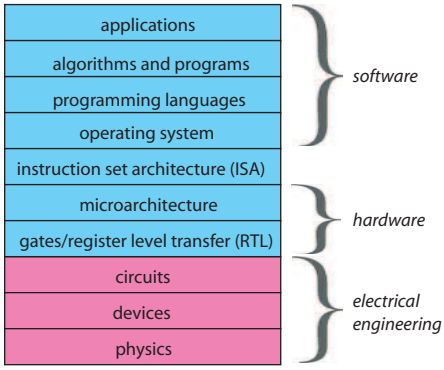

Fig. 1: layers of abstraction in a modern computer system.

3.4 The instruction set architecture

The **instruction set architecture** (**ISA**) is the interface between hardware and software. It is the view the programmer has of hardware and includes everything programmers need to know in order to control the processor. The ISA interface can be envisaged as the surface of the water where above the water the swan progresses serenely (analogously to the high-level abstraction of an executing program) whilst below the water the swan's feet are paddling furiously (analogously to the activity of the processor and the electronic circuits composing it). Many different ISAs exist (dependent upon the actual processor), e.g., MIPS, ARM and SPARC (we'll take a brief look at MIPS in a moment).

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

The ISA includes:

- the organisation and structure of programmable storage (namely the registers and main memory that is useable by the programmer);

- the instruction sets and formats (namely the processor instructions available to the user in order to access and manipulate data residing in memory);
- the methods for addressing and accessing data items within memory, e.g., absolute ('go to this location'), indirect ('go to the location stored in this location'), using offsets ('go to this location plus this offset number'); and
- the process of execution, e.g., the fetch-decode-fetch-execute cycle.

A primary component of the ISA is its **assembly language**. As we have already seen, a high-level program is compiled first into assembly language and then into machine code. Assembly language instructions are very low-level and facilitate the subsequent translation into machine code.

3.5 Our processor

Let's look now at a typical ISA, namely the **MIPS** (**Microprocessor without Interlocked Pipeline Stages**) ISA (MIPS processors are widely used by, for example, Silicon Graphics, Nintendo and Cisco). Our presentation of MIPS is much simplified and it is only the essence of MIPS that we want to get across and not the precise details.

We assume that:

- memory has 2^{32} memory locations (so, our address bus is 32 bits wide);
- words are 4 bytes long (so our data bus is 32 bits wide);
- when we request the contents of a memory address m, the contents of memory locations m, $m+1$, $m+2$ and $m+3$ are returned; and
- there are 32-bit registers $\$zero$ (which always holds 0), $\$s1$, $\$s2$, ..., $\$s7$.

Every assembly language instruction is 32 bits wide (we'll see what this means in a second). Instructions include:

`lw $s1, ($s2)`	register $\$s1$ takes the value of the memory location held in register $\$s2$
`addi $s1, $s2, x`	register $\$s1$ takes the value of register $\$s2$

	plus the number x
`beq $s1, $s2, m`	if the value of register $\$s1$ is equal to the value of register $\$s2$ then go to memory location m
`slt $s1, $s2, $s3`	if the value of register $\$s2$ is less than the value of register $\$s3$ then register $\$s1$ takes the value 1 otherwise it takes the value 0
`j m`	jump to memory location m

(we can substitute different registers in these instructions). Note that whilst arithmetic and other operations were done through the accumulator in our simple von Neumann architecture as presented earlier, in our more sophisticated MIPS processor a variety of registers can play the role of the accumulator).

3.6 A simple searching program

Consider the following C program that looks for the largest integer in a list of positive integers held in an array $A[0 \ldots n-1]$ (where we assume $n \geq 1$; an **array** in C corresponds to a **list** in Python):

```
x = A[0];
for (i = 1; i < n; i++)
{
  if (A[i] > x)
  {
    x = A[i];
  }
}
```

A C compiler converts this program into assembly language and the assembly language is then converted to machine code.

3.7 Searching in assembly language

Here's how a compiler *might* translate our C code to assembly language:

- register $\$s1$ contains the current value of the variable x and starts with the value 0;

- register $\$s2$ is initialized to 44 and in general contains the memory address of the array item being considered, and the values of the list elements $A[0], A[1], A[2], \ldots$ are in memory locations $44, 48, 52, \ldots$;
- register $\$s3$ contains the current value of the variable i and starts with the value 0; and
- register $\$s4$ contains the value n, with registers $\$s5$ and $\$s6$ used as work registers.

```
0    lw $s1, ($s2)          $s1 := contents of memory location $s2
4    addi $s3, $s3, 1       $s3 := $s3 + 1
8    beq $s3, $s4, 40       if $s3 = $s4 then jump to address 40
12   addi $s2, $s2, 4       $s2 := $s2 + 4
16   lw $s5, ($s2)          $s5 := contents of memory location $s2
20   slt $s6, $s1, $s5      if $s1 < $s5 then $s6 := 1 else $s6 := 0
24   beq $zero, $s6, 32     if $s6 = 0 then jump to address 32
28   addi $s1, $s5, $zero   $s1 := $s5
32   addi $s3, $s3, 1       $s3 := $s3 + 1
36   j 8                    jump to address 8
40   break
```

3.8 Assembly language in action

[Animation]

'interpreting code as data and data as code'

3.9 Assembly to machine code

All that remains to do is to explain how (MIPS) assembly language programs are converted into machine code.

- All instructions are 32 bits wide.
- There are three different types of instructions:
  - an **I-type instruction** involves data transfer:

    - `addi`, `lw`, `beq`
  - an **R-type instruction** works on registers:
    - `slt`
  - a **J-type instruction** involves jumps:
    - `j`

The MIPS ISA actually has 32 registers including:

- $\$zero$
- $\$s0$, $\$s1$, $\$s2$, ..., $\$s7$
- $\$t0$, $\$t1$, $\$t2$, ..., $\$t7$.

The above registers are encoded using 5 bits:

- $\$s0$, $\$s1$, ..., $\$s7$ are encoded as 10000, 10001, ..., 10111 (decimal 16-23)
- $\$t0$, $\$t1$, ..., $\$t7$ are encoded as 01000, 01001, ..., 01111 (decimal 8-15).

3.10 I-type instructions

An I-type instruction has format as depicted in Fig. 2.

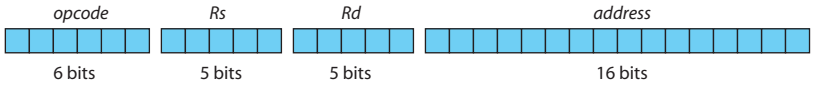

Fig. 2: format of an I-type instruction.

where:

- *opcode* is the instruction code;
- *Rs* is the source register;
- *Rd* is the destination register; and
- *address* is a memory address or a constant (depending upon the instruction).

For example:

- `addi $s3, $s3, 1` in machine code is

  001000 10011 10011 0000000000000001

- `beq $s3, $s4, 40` in machine code is

  000100 10011 10100 0000000000101000

- `lw $s1, ($s2)` in machine code is

  100011 10001 10010 0000000000000000

3.11 R-type instructions

An R-type instruction has format as depicted in Fig. 3.

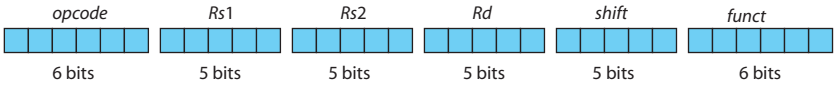

Fig. 3: format of an R-type instruction.

where:

- *opcode* is the instruction code;
- $Rs1$ is the first source register;
- $Rs2$ is the second source register;
- Rd is the destination register;
- $shift$ is a shift amount for shift instructions; and
- $funct$ is a supplement for opcode for some instructions.

For example:

- `slt $s6, $s1, $s5` in machine code is

  000000 10110 10001 10101 00000 101010

3.12 J-type instructions

A J-type instruction has format as depicted in Fig. 4.

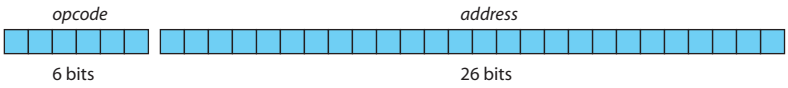

Fig. 4: format of an J-type instruction.

where:

- *opcode* is the instruction code; and
- *address* is a memory address.

For example:

- `j 8` in machine code is

  000010 00000000000000000000001000

3.13 Machine code for our program

So, the machine code for our program is:

0	`lw $s1, ($s2)`	10001110001100100000000000000000
4	`addi $s3, $s3, 1`	00100010011100110000000000000001
8	`beq $s3, $s4, 40`	00010010011101000000000000101000
12	`addi $s2, $s2, 4`	00100010010100100000000000000100
16	`lw $s5, ($s2)`	10001110101100100000000000000000
20	`slt $s6, $s1, $s5`	00000010110100011010100000101010
24	`beq $zero, $s6, 32`	00010000000101100000000000100000
28	`addi $s1, $s5, $zero`	00100010001101010000000000000001
32	`addi $s3, $s3, 1`	00100010011100110000000000000001
36	`j 8`	00001000000000000000000000001000
40	`break`	00110100000000000000000000000000

Of course, there are no 'end-of-lines', and so on, in our machine code; it is just a long string of 0s and 1s.

3.14 Layers of abstraction in modern computer systems

All that is left to study is the layer labelled 'operating system' in Fig. 1. We can then move on to looking at issues relating to problem solving and some real-life algorithms. Actually, our abstraction in Fig. 1 is a little rough and can be refined a bit. It is better to speak about **system software** as encompassing the **operating system** and other system programs (programs that 'make the computer work') such as compilers and virus protection. This refinement can be visualized in Fig. 5.

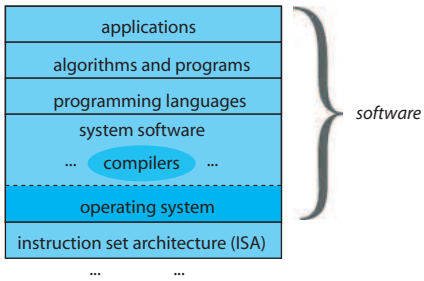

Fig. 5: refined layer of system software.

3.15 Operating systems

The primary component of the system software is the operating system. An **operating system** provides the interface between different programming applications and hardware. The whole need for an operating system resides around the fact that although a computer generally has one CPU, nevertheless it is concurrently undertaking a whole series of tasks (even when a computer has more than one processor, there are usually more processes executing than there are processors). In short, the operating system manages a machine's resources, e.g., processor cycles, memory and devices such as modems, disks, network interfaces, video interfaces and so on, so that they are used efficiently and safely.

So that this might happen, the operating system provides an abstraction of the hardware for use by an application; that is, the operating system controls and organises the use of hardware by different programs and all access to hardware is through the operating system. The operating system has access to hardware through ISA instructions and also through specialised **system calls**. It is through system calls that an operating system organises

access to memory by individual programs, for example, and ensures that memory allocated to each program is kept separate and secure from other programs. This is called **data security**.

The main component of an operating system is the **kernel** which is the single program that is loaded at boot-time and which has primary control of the processor. The main functions of an operating system can be described as follows.

- The operating system **virtualizes** a machine. Hardware has low-level physical resources with complicated, idiosyncratic interfaces. An operating system provides abstractions that present clean interfaces so as to make the computer easier to use. For example, whilst the hard disk is a mechanical beast consisting of sectors and tracks, the operating system presents it as a well-organised file store. Also, virtualization means that the operating system can make a physical machine behave like an idealised virtual machine so as to aid portability. For example, Unix runs on many very different computer systems and programs can be ported across systems with very little effort.

- The operating system starts and stops programs. For example, it ensures that when a program is stopped, the memory allocated to this program is freed up for use by other programs and the processor is allocated to some other task (which is presumably in some queue waiting for processor access).

- The operating system manages memory. To see what we mean by this, consider compiling some C program. The compiled program will require memory so that it might run. However, it may be that the memory required is already being used by some currently executing program. The operating system will ensure that the compiled C program can still run by using memory that the compiled program *thinks* is the memory requested whereas in reality the memory actually *used* is different.

- The operating system handles input/output. Whilst programs are running, users are typing on keyboards, outputs are being sent to screens and so on. The operating system ensures that all of these activities are coordinated. For example, an operating system handles **interrupts** which might be caused, for example, by pressing some keyboard key and result in the execution of some program being paused or aborted.

- The operating system maintains and provides access to the file system. It ensures that programs and users only have access to their own files and not those of others, and that users can access their files in a well-structured way.
- The operating system deals with networking. For example, it continuously monitors the network for incoming traffic and sends data packets as output when requested by some program.
- The operating system maintains security through firewalls, and tries to ensure that executing programs don't have access to parts of the system from which they can cause malicious behaviour.
- The operating system facilitates error handling and recovery when programs don't do what they are supposed to do. It tries to provide some limitation as to the potentially disastrous effects of badly written or maliciously written programs.

'modularizing in anticipation of multiple users'

3.16 Input/output

Let's take a closer look at how the operating system deals with input/output. Input/output devices might include a hard disk, a floppy disk, a CD drive, a graphics card, a sound card, an ethernet card, a modem and so on. Such devices tend to be connected to the CPU via a bus (or buses).

As we have already seen, a **bus** consists of a series of communication wires (or signals or lines), the number of which is determined by the **width** of the bus. Buses are cheap and versatile but are typically slow in comparison with the speed at which a CPU operates, and thus can result in a communication bottleneck. The operating system helps alleviate this bottleneck.

In order to do this, the operating system uses **interrupts**. Imagine a program requiring input from the keyboard. The time taken to request this input and wait for this input to be entered is considerable. The operating system allows the CPU to continue with other tasks whilst our program waits for input. The input from the keyboard is only dealt with when the operating systems's **interrupt handler** announces that an interrupt has

been **raised** (that is, that such keyboard input has been entered). The interrupt is signalled on one of the lines of the bus connecting the CPU and the input/output device. The interrupt handler makes the necessary arrangements, amongst other processes, so that our paused program resumes, with access to the keyboard input.

3.17 Processes

As we have explained, in general there will be a lot of programs executing simultaneously on a CPU. Each executing program is called a **process**. It is important to make the distinction between a program (residing on a disk) and an *executing program*, i.e., a process. There may be many concurrent processes, with more than one process resulting from executing the same program more than once.

'processing in parallel'

In more detail, a process consists of a thread or threads (of execution) together with an address space, where:

- a **thread** is a sequence of instructions in the context of a sequential execution (that is, a coordinated execution resulting from some program); and
- an **address space** consists of some memory locations (in main memory) that the corresponding process can read from and write to.

A process within which there are multiple threads generally needs to have these threads communicate with one another and synchronize; similarly, a collection of processes need to communicate and synchronize.

This leads to the problem of **mutual exclusion**; that is, ensuring that two threads are not in the critical section at the same time, where the **critical section** is exclusive access to some shared resource such as a memory location.

Consider the situation where two threads have access to the same counter c and where each thread is to increment this counter. This is done by each thread reading the counter, incrementing it by 1 and writing the counter

(back to its memory location). The critical section is acquiring access to the counter for the purpose of incrementing it. Suppose that the operating sequence does not enforce mutual exclusion. One execution sequence might be:

c has value 0
    thread 1 reads the value of counter c, namely 0
    thread 1 increments its value, to 1
    thread 1 writes its value back to c
so now c has value 1
    thread 2 reads the value of counter c, namely 1
    thread 2 increments its value, to 2
    thread 2 writes its value back to c
so now c has value 2

However, another execution sequence might be:

c has value 0
    thread 1 reads the value of counter c, namely 0
    thread 1 increments its value, to 1
    thread 2 reads the value of counter c, namely 0
    thread 1 writes its value back to c
so now c has value 1
    thread 2 increments its value, to 1
    thread 2 writes its value back to c
so now c has value 1

The second execution sequence has clearly not achieved what was intended.

'understanding and using the power of non-determinism'

3.18 Virtual memory

Having many simultaneous processes means a lot of RAM might be required and often there is not enough. The operating system will make use of **virtual memory** by **paging**, that is, moving memory (or, more, precisely, the values from memory) backwards and forwards between RAM and the hard disk as

and when required. In general, the processor time-shares in order to execute a number of processes simultaneously. However, this time-sharing may not be equitable as some processes might have a higher priority than others with the other processes waiting until the high priority processes have finished before executing. The operating system moves the address space of the lower priority processes to the hard disk so that the higher priority processes can execute more quickly (recall that the further away memory is from the CPU, the higher the access time). Of course, if there are still too many higher priority processes so that all their address spaces fit into RAM, the execution time of every process will be slower because there will be at least one process slowing things down.

Whilst each process might think it has access to its chosen address space and to the CPU, the reality might be that it is not only sharing the CPU with other processes but its actual address space might reside not in RAM but on the hard disk.

3.19 Process life-cycle

A typical abstraction of the life-cycle of a process is as a **finite state machine**, as can be visualized as in Fig. 6.

There are 5 states:

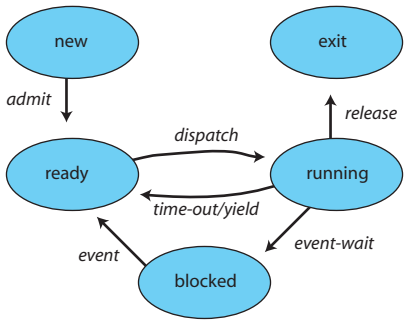

Fig. 6: the life-cycle of a process.

- **new**: the process is being created;
- **ready**: the process is not executing on the CPU but is ready to execute;
- **running**: the process is executing on the CPU;
- **blocked**: the process is waiting for an event (and so not executable); and

- **exit**: the process has finished.

There are the following state transitions, named according to the action:

- **admit**: the control overheads as regards a process have been set up and the process is moved to the run queue;
- **dispatch**: the scheduler allocates the CPU to an executable process (the **scheduler** is the component of the operating system that determines which processes should be run and when);
- **timeout/yield**: the executing process is forced to/volunteers to release access to the CPU;
- **event-wait**: a process is waiting for an input/output event, for example, and gives up access to the CPU;
- **event**: an event occurs and wakes up a process; and
- **release**: a process terminates and releases access to the CPU (and other resources).

3.20 Process control block

Given that the operating system is to manage each process, for example, by paging its address space in and out of the hard disk or by pausing the process (whilst waiting for an I/O event or executing a higher priority process), there needs to be some book-keeping undertaken so that the current situation of any process is fully understood. A **process control block** (**PCB**) is a data structure that the kernel uses in order to manage a process. It consists of:

- the process's unique ID;
- the current state of the process (new, exit, ready, etc.);
- CPU scheduling information such as the process priority, whether there are events pending and so on;
- the program counter, detailing the next instruction to be executed (this is only available for processes that are not running, having been paused for some reason);

- the current values of other CPU registers (again, this is only available for processes that are not running, having been paused for some reason);
- memory management information, such as the (current) location of the address space;
- scheduling and accounting information, such as when the process was last run and the amount of CPU time that has elapsed so far; and
- a reference to the next and previous PCB in the list of PCBs.

3.21 Context switching

The whole process whereby the operating system pauses one process and resumes another process is called **context switching** and can be very time consuming. It can be visualized as in Fig. 7. Context switching entails managing the PCBs as different processes are given access to the processor.

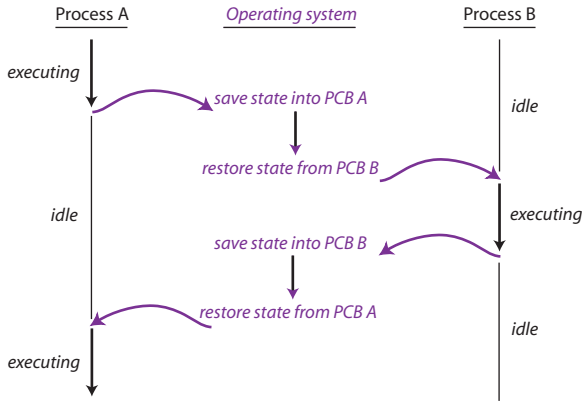

Fig. 7: context switching.

3.22 Research glimpse: Operating Systems

There have been numerous technological advances in recent years including the move to multi-core, cloud computing and embedded systems. There has also been an increase in user expectations as more computing power means that users expect a corresponding increase in such things as real-time multimedia quality of service (QoS), security, dependability and power awareness. Such properties are not well catered for and supported by today's (commodity) operating systems.

One next-generation operating system is **Tessellation**, being developed at the University of Berkeley, U.S.A. This operating system is structured around '*resource distribution, performance isolation and QoS guarantees*'. The concept of 'process' as an operating system primitive has been replaced by that of 'cell', with processors clustered within cells and with cells carrying with them 'guarantees' of resources such as processors, caches, network or memory bandwidth, fractions of system services and so on.

The name Tessellation derives from the space-time partitioning approach that tessellates cells across the space of hardware resources. The operating system Tessellation is targeted at both existing and future many-core machines (recall, a many-core machine is a multi-core machine in which the number of processors is large enough so that traditional multi-processor techniques are no longer efficient).

References

The books below all include introductory material as regards basic aspects of ISAs and operating 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 from library].

[8] A. Silberschatz, P.B. Galvin and G. Gagne, *Operating System Concepts* (7th edition), Wiley (2005) [005.43 SIL].

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

[10] A.S. Tanenbaum, *Modern Operating Systems* (2nd edition), Prentice Hall (2001) [005.43 TAN].

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

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