The Elusive Low Level

Graham White [1]

Abstract. We start with a example of assembler programming, and show how even at this low level the structure of the programming language does not directly mirror the structure of the hardware, but that it is also decisively influenced by the human practices surrounding computer use. We give several historical examples and illustrate the changing pattern of mutual accommodation between human practices and computer technology.

1 Introduction: Three Surprises

We are told – in fact, if, like me, "we" teach computer architecture, we *tell* people – that computer languages can be either high level or low level, and that the low level ones, specifically the various assembly languages, all reflect more or less directly the configuration of the hardware. So it comes as a bit of a shock when we get to know a particular assembly language – I am thinking specifically of MIPS32 [15] – and we write programs which are sequences of instructions, and we are tempted to believe that the hardware will execute the instructions in the order that we write them in, but no: if there is a branch instruction, which tells the computer, depending on the equality or otherwise of two numbers, to resume execution from some other instruction it will not execute the branch *right then* but only when the instruction after the branch has been executed [7, A-59]. (This is because a branch takes a comparatively long time to execute, so the computer might as well have something to get on with while it does so).

So that, then, is surprise 1. Now generally people learn assembler by working on a simulator, rather than by actually executing the code on the appropriate hardware: and surprise 2 is that, with the simulator,[2] you have a choice of whether it gives you this rather unintuitive behaviour or not. So you can choose between a simulator which simulates the hard-to-learn behaviour of the real hardware, or the somewhat easier behaviour of fictitious hardware. Clearly the question of "what is the real hardware", or "what is the real low level", is not as straightforward to answer as we might like to think.

I shall be arguing in this paper that this phenomenon is quite typical: at the low level of assembly programming, you might expect that human factors are rather minimally in play, if they are in play at all, and that the programming language somewhat directly mirrors what the hardware does. But not so: humans who work with computers at this level are presented with a carefully achieved view of what the hardware is doing, and it would be an oversimplification to imagine that this view was a direct presentation of the hardware. To some extent this is inevitable: modern computer hardware, and especially CPU chips, is extraordinarily complex, most of it is designed by computer programs, and nobody can have any sort of overview of what the CPU actually does. Furthermore, the behaviour of modern computers is nondeterministic and chaotic, so that we cannot predict their behaviour, at least in the short term. If you deal with this nondeterminism in its own terms (measuring the statistics of hard disc access, caching and the like) then you are an electronic engineer and not a computer scientist: Hennessy and Patterson [6] will give you a good idea of what this entails.

But there is also a role for computer scientists, namely the people who design and implement algorithms, design higher level programming languages, database systems, and the like. For this, one needs a rather more abstract view than the one given by electronic engineering. As it happens, the programming models that computer scientists use (students, obviously, but also professionals) present computers to them as something vaguely like Turing machines: however, in reality, the computers themselves are not Turing machines, but something much stranger.

What we have, then, is a system with three components: the human designers and programmers, the hardware, and the software and assembly languages that present a view of the hardware to the programmers. All three sides of this triangle are important. It is a system that is constantly under strain, mostly due to the ongoing rapid progress in technology coupled with the rather stable nature of programming language design. Consequently, the picture that emerges is of continuous frantic improvisation in order to present the same (or nearly the same) view to the programmers while the underlying hardware is constantly in flux. Currently, there is a great deal of action around multicore processors, concurrency, and caching: these are hard issues, but dealing with them is probably they key to having models of modern computers that humans can, in some way, program.

Which brings us to Surprise 3. We might like to think that the sophisticated intellectual background behind modern programming languages took a long time to arrive, just as it took a long time to get from small, old, slow hardware to modern, fast, capable hardware. But, as we find from [10], a great deal of that intellectual infrastructure was in place very early, by the 50s or early 60s at least.

1.1 Methodology

There will be a large historical component to this argument, and for the historical facts I shall be mostly relying on Mark Priestley's PhD thesis [10]. My own methodology, however, will deviate somewhat from Priestley's: I shall be influenced by ethnomethodology, although I will not be using ethnomethodological methods, and I will view the members of the relevant community as engaged in a constant process of, in Garfinkel's words, achieving and sustaining their practices of programming and their interactions with each other when they talk about programming and language design. I shall rely to some extent on textbooks for an account of the community view of particular technologies: in particular, I shall rely on Hennessy and

[1] Electronic Engineering and Computer Science, Queen Mary, University of London, email: g.graham.white@gmail.com, url: http://www.eecs.qmul.ac.uk/∼graham

[2] Specifically, this one `http://spimsimulator.sourceforge.net/`

Patterson [7] for an account of the current state of hardware and of good practice in assembly programming.

I shall take it that programs are written in order to implement algorithms, and that, correspondingly, they have a semantics: integer variables in a program will correspond to integers in the algorithm, and, depending on the algorithm, they can be updated, become the arguments of arithmetical expressions, and so on. This semantics is not straightforward to define formally (see Priestley [10, §4.9] and White [14]), but I shall only be concerned here with an informal concept along the lines of "what the programmer intends when they are writing the program" (an informal concept which could, in principle, be elucidated by asking the programmer what, for example, a particular variable represents).

I shall definitely not be viewing the universal Turing machine as an idea which dropped from heaven, and which was then triumphantly implemented in both hardware and the design of programming languages. Furthermore, many of the ideas of Turing and his contemporaries were actually influential in the practice of programming and in the design of hardware, without passing through the idea of the universal Turing machine: Turing had more ideas than that, and those other ideas were also influential. The connection between computers and universal Turing machines was not appreciated by many of Turing's contemporaries (Priestley [10, §3.7]). And, although the idea of the Turing machine has had a considerable influence on, for example, language design, modern computers are not in any simple way Turing machines, as we will come to see later.

Furthermore, this process is historical, and everything is always changing. For all of his influence, modern programming languages and programming practice deviate quite markedly from Turing's own ideas: for example, Turing was extraordinarily attached to the idea of self-modifying code (Priestley [10, §3.4]), an idea the attractiveness of which is hard to appreciate from the current perspective.

2 The Idea of a Variable

One of the key ideas in programming is the idea of a variable: there are basically two stories to tell about it, one being about the idea of a variable *in general*, the other being the idea of a variable in assembly language programming. We will tell the first story first, because these ideas will be more familiar to the modern audience: nevertheless, there are certain distinct features of variables in assembler, which will will also describe.

2.1 In General

In modern terms, a *variable* is an area of memory with a name attached [13, pp. 37f.]; names are declared in computer programs, and they persist during the execution of the program (so long as they are *in scope*, a concept which we will come to later) and these declarations are laid down in the *program*, rather than being artefacts generated in a particular execution of the program. Variables may have values, and these values can be inspected and updated.

There is nothing like this in the idea of a Turing machine: Turing machines certainly have tapes, and locations in the tape store values, which can be read or written. But locations of the tape do not have *names* (in fact, although the locations are arranged in order along a tape with a beginning but no end, and so implicitly can be numbered, these numbers are not available to the machine). Furthermore, although a Turing machine can read and write to the tape, there is no concept of updating *the same entity* which we have when we update a variable: if a Turing machine writes to a tape location, this could correspond to updating a value in the algorithm which the Turing machine program implements, but it could equally well be the replacement of a temporary value with another unrelated temporary value.

Variables in programming languages are also not the same as variables in mathematics, since they can be updated, and mathematical entities, whatever they are, do not change. *Assignment statements* are the statements by which variables receive values or have their values updated: they are usually written with an equality sign, thus:

```
x = x + 1
```

These equality signs do not stand for mathematical equalities: they are not symmetric, for example (the assignment statement `x+1 = x` does not make sense, because `x+1` is not a variable, although `x` is).

However, variables certainly appear in Turing's work, even in 1936: he described the implementation of algorithms in Turing machines by means of a notation which he called "machine tables", or, for short, m-tables. These tables would describe the behaviour of the machine when it was presented with a particular symbol on the tape and a particular internal configuration: the behaviour was represented by means of mathematical functions, and these functions could contain free variables which would hold the value of the currently scanned symbol. So the idea of the variable – in the *mathematical* sense – certainly enters here.

One possible influence for the idea of updating variables is the practice of numerical analysis, that is, the systematic calculation of solutions to mathematical equations. Much of numerical analysis is concerned with progressive approximation to, for example, the value of an integral, or to an irrational number: one would start with a rough approximation and then progressively improve it, using something like Newton's method. Although numbers are mathematical entities, and thus cannot change, there is a very good sense in which one can talk about changing, or updating, an *approximation* to a number; one can see this at work in old textbooks of numerical analysis, which frequently have quite detailed instructions about how to lay out such calculations on a page (see Hartree [5]). And the practice of numerical analysis was, in fact, referred to by early computing researchers: Howard Aiken and Grace Hopper write, describing the design of an early computer, that

> The development of numerical analysis ... [has] reduced, in effect, the processes of mathematical analysis to selected sequences of the five fundamental operations of arithmetic: addition, subtraction, multiplication, division, and reference to tables of previously computed results. The automatic sequence controlled calculator was designed to carry out any selected sequence of these operations under completely automatic control. (Aiken and Hopper [2, p. 386], cited in Priestley [10, p. 92])

Now early computers were largely used for carrying out numerical calculations (Priestley [10, §3.1]) – indeed the early computation groups consulted established numerical analysts such as Hartree – and so it is not inconceivable that the idea of a variable was at least partly influenced by the practice of numerical analysis.

2.2 In Assembly Language

Computers typically hold numbers in memory, and memory locations have addresses, which are numbers (in this respect, computers differ from Turing machines). Addresses are used in several scenarios:

for data If we want to perform an operation, such as addition, on data, then we will need the addresses of the arguments and the address where the result is to be stored

for instructions if we are to perform a jump instruction, then we will need the address of the instruction that execution is to jump to; if we are to execute a subroutine, then we need the address of the instruction where execution of the subroutine starts, and, when the subroutine returns, we need the address of the instruction after the calling instruction

Now it is generally speaking impractical to simply write instructions with these numerical addresses in them: for one thing it is very error prone, numbers not being very memorable. But the main disadvantage is that there is no practical way of telling where in memory a particular piece of code or data is going to reside. This is particularly acute in the case of subroutines, because a subroutine may be called more than once in a particular program, so that the return address simply cannot be specified in advance.

This was recognised very early on. Turing discusses how to deal with return addresses for subroutines by putting the return address of the subroutine into memory:

> When we wish to start on a subsidiary operation we need only make a note of where we left off the major operation and then apply the first instruction of the subsidiary. When the subsidiary is over we look up the note and continue with the major operation (Turing, [12], cited in Priestley [10, §4.5, p. 104])

We should note that, although the precise details of "mak[ing] a note of" have varied, this is still the same way that the return addresses of subroutines are handled.

There still remains the problem of jumps. Turing, too, worked on this problem: instructions were to be written in what Turing called a "popular" format, and they were to be written, one instruction per card, in "groups" (in practice, things like subroutines and the like: they should be sequences of instructions which could be guaranteed to end up in contiguous places in memory). Each instruction would have associated with it the name of the group and its "detail figure", or place within the group. Programs would be constructed by taking all the relevant cards (for the main routine, subroutines and so on), collating them, and then assigning memory locations: from this one could translate from group name and detail figure to a memory location. Then one would have to replace all the memory addresses in instructions (which would, of course, have been written in the popular format with group names and detail figures) with the actual memory addresses. Turing seems to have envisaged this being done by hand (assisted by punched card manipulation machines, which were common technology at the time), but he did recognise that it would be something which could be done "within the machine". (Priestley [10, §4.5, p. 104])

The task of showing how such address translation could be done within the machine was begun by Goldstine and von Neumann [4]; they proposed first loading the program and its subroutines, and then running a "preparatory routine" which would perform the required address modification. (Priestley [10, §4.5, p. 105])

2.3 Accommodations

We have seen here the problems that were raised by introducing variables and subroutines into computer programming: there is no direct support for these things in the Turing machine, and there was very rudimentary support, in the form of memory and instruction addresses, in the early computers. There were already human practices – namely those developed for numerical analysis – which computers needed to support, and the solution was to develop coding languages and practices which were not too distant from what people were already doing by hand, and to fill the gap between human coding and machine execution by a process of translation. So we see a process of accommodation, driven by the gap between the hardware and the pre-existing human practices.

3 Modern Hardware

In this section we will look at how modern hardware typically works. One of the main problems here is that of speed disparity: the central processing unit (CPU) of a modern computer executes a basic instruction in less than a nanosecond (complex instructions, like multiplication, might of course take longer), whereas hard disk access will take tens of milliseconds. RAM access is intermediate between the two. The ratio between CPU speed and hard disk speed is somewhat over a million. If the CPU had to wait milliseconds every time it needed data for an instruction, then computers would be much slower than they actually are. There are several technologies that we can use to deal with this. None of them is directly under the control of applications or systems programmers: they are designed by chip designers, they (hopefully) make the hardware run faster, and they are *transparent*: that is, programmers simply write the instructions they would have written anyway, but the computer executes them faster than it would have done without caching, pipelining, and so on.

3.1 Caching

This is the strategy of fetching data in relatively large units, and storing it in fast memory near where it is to be used. For example, modern CPUs have caches where they store data that they get from RAM: they fetch data in large contiguous blocks, store it in their cache, and, when they need more data from RAM, they check the cache first [7, §5.3, pp. 383ff]. This generally works: with modern hardware, CPUs will find the data in the cache about 90% of the time [6, p. B-10].

The reason why caching works, when it works, is what is called *data locality* [7, §5.1, pp. 374ff]. This is the idea that data which is relevant for a particular calculation is usually held contiguously: if we have large amounts of data, then it will generally be in an array or some large data structure of that sort, and that data structure will (hopefully) be held contiguously. So if we are iterating along an array, and if we cache the array in contiguous blocks, then most data will be found in the cache: the only reason to go to RAM for data is when we have already iterated through the data in the cache.

Similarly, instructions are stored in RAM in the order that they occur in the program, and they are *generally* executed in that order too. So caching tends to win here also.

However, data locality may fail in both of these scenarios. Two-dimensional arrays (that is, arrays with two indices) can be thought of as big matrices, and there are two ways of iterating over them: in the first, you select column 0, iterate over that, then select column 1, iterate over that, and so on. For the other way of iterating, you do the same thing but with rows rather than columns. Now if you write array code in Java, it turns out that one of these ways gives you data locality but the other way does not, so there is a large performance penalty for doing it one way rather than another.

Consequently, although there are large gains to be made by caching, they are not automatic: both for instructions and for data the gains depend on the statistical character of the code or data concerned.

3.2 Pipelining

Instruction execution in a CPU is generally performed in several stages: first the instruction is fetched from RAM, then the CPU decides what sort of instruction it is (which affects whether the instruction needs data, etc.), then the instruction is executed, and finally the result is written back to RAM. These stages are generally performed by different parts of the CPU, but they each depend on the previous stage having been performed. But there is nothing wrong with executing stage 1 for a particular instruction while the CPU is executing Stage 2 for the previous instruction: this is called *pipelining* [7, §4.5, pp. 272ff].

So, if we have a sequence of instructions like this

```
i1
i2
i3
 . . .
```

then we could have a pipeline which, at successive times, was doing the following:

```
t1  stage 1 of i1
t2  stage 1 of i2  stage 2 of i1
t3  stage 1 of i3  stage 2 of i2 stage 1 of i1
```

What can go wrong? Suppose, on the other hand, we have a loop:

```
for(i=0;i<4;i++) {
  i1
  i2
  i3
}
i4
```

so that (if we have a 4-stage pipeline as above) we will execute instructions in the following order:

```
i1 i2 i3 i1 i2 i3 i1 i2 i3 i1 i2 i3 i4
```

However, the pipeline will fill up with, successively, `i1`, `i2`, `i3`, and then (because it is the next instruction in the listing) `i4`. However (unless this is the last time round the loop) the next instruction to be executed after `i3` is `i1`: but we cannot in general tell whether we are going round the loop again until the last instruction of this loop has been executed. And if then we have to start refilling the pipeline from `i1`

Because of this, modern CPUs generally do *branch prediction*: that is, at places where a branch might happen (end of a `for` loop, beginning of a `while` loop, and so on) they try to guess what branch might be taken, and keep filling the pipeline accordingly. For example, it is generally a good idea to guess that, when you get to the end of a `for` loop, you will continue execution from the beginning of it: this is because most loops get executed more than once, and often much more than once (there is no point in having them otherwise), so that you will generally win by guessing that way.

Branch prediction, then, is like data locality: it is a statistical matter, and depends on code being written in a certain way. It would certainly possible to write code in such a way that it would break pipelining by forcing branch prediction to misbehave, but it would probably be quite hard to do, and it would need some specialised knowledge of what hardware it was to be run on (how many stages in the pipeline, some details of its branch prediction, and so on).

3.3 Registers

It has always been recognised that it was advantageous to have fast memory locations inside the CPU, and that these could be used for storing frequently used data. These locations are called *registers*, or *accumulators*, and have been part of computer design since the early days (for example, the ENIAC – which started running in 1945 or so – had 20 accumulators [10, p. 61]).

Registers can often achieve a considerable speedup. For example, if we have a loop like this:

```
for (i=0;i<100;i++) {
  i1
  i2
  i3
. . .
}
```

then we could, theoretically, read the loop variable `i` from the hard disk every time it was to be used, and write it to hard disk every time it was to be updated, but that would be very wasteful: the sensible thing to do is to keep `i` in a register, initialise it to zero when we start executing the loop, and to release the register after the end of the loop: it need never even be stored in RAM. Assembly code allows access to registers: in fact, many assemblers (MIPS, for example) only allow arithmetic operations between the contents of registers, and only allow the result to end up in a register. There are other instructions for reading data from RAM to a register (*loading* the data) and for writing data from a register to RAM (*storing* the data). This is the so-called *load-store paradigm*.

Now because they involve RAM access, `load` and `store` instructions are comparatively slow, so there is a great deal to be gained by eliminating them as much as possible. For example, a `store` of data from a register to a location, followed by a `load` of the same data to the same register from the same location, can generally be eliminated; the same situation, only with different registers, can generally be replaced by simply moving data from one register to another, and so on. But a given CPU will only have a certain number of registers, so one will inevitably encounter situations where one has to store data in a particular register in order to make room for new data. This is the problem of *register allocation*: how to decide in what registers old data should be replaced by new data.

Now compilers, from languages like C or Fortran, would generally emit assembly code for the relevant architecture, which would then be assembled and run. It used to be the case that human programmers could generally write better and more efficient assembler than compilers could produce (better register allocation, and so on): this is no longer the case, because of progress with register allocation algorithms. Thus, programmers' contact with modern computers generally does not reach inside the CPU, and, in particular, it generally does not deal with registers.

3.4 Hard Disks

Hard disks are, as is well-known, composed of a stack of rotating magnetic platters together with a set of read/write heads which are mounted on an arm that can move the heads in or out over the platters (Hennessy and Patterson [7, §5.2, pp. 381ff]). The surfaces of each platter are divided into concentric *tracks*: that is, if the heads are at a fixed position, they will read or write from a particular track on each surface as the disks go round. The set of all of the tracks which are under the heads with the arm at a fixed position is called a *cylinder*. Each track is divided into *sectors*.

Thus, to access the data in a particular sector, the disk must do the following:

1. move the heads to the appropriate cylinder,
2. wait until the appropriate sector rotates under the head, and
3. read or write the data from that sector.

The average time for 1 is between 3 and 13 milliseconds, and the average time for 2 is about 5 milliseconds: these are comparatively long times in computer terms. It is obviously advantageous to take advantage of contiguity: that is, if we successively read data, it would be best if it came from the same cylinder.

However, this is very difficult to achieve. Modern hard disks have hardware in them called *disk controllers*, which do several things. One thing they do is to cache data as they read it from the disk or before they write it to the disk: because they do this, they can reorder reads and writes in order to minimise head movement. They also do error checking (they record extra check bits with the data in order to detect errors if they occur), they monitor errors, and they can move data off a sector and onto another if they think a sector is deteriorating. Because of this, it is very hard to tell whether data is located contiguously or not: blocks of data could start off close to each other, but end up distant because of data movement. None of this is under the control of programmers, and so optimisations that used to be possible are now no longer possible. On the other hand, disk are now genuinely faster and more reliable, due to more intelligent disk controllers, so this is probably a net gain.

3.5 Accommodations

We have seen that the assembly programmer sees a particular view of the workings of a computer: the programmer can load and store data from the RAM to the CPU, can work on it in the CPU, and gets a generally sequential view of the execution of instructions. The programmer can also access hard disks, and these hard disks are organised in cylinders and tracks and sectors. None of this is strictly speaking true: data is cached between RAM and the CPU, and in the disk controller. Instructions are not strictly executed one at a time, but they overlap due to pipelining. And the hard disk geometry as seen by the programmer is an idealisation: data movement makes it very difficult to optimise the location of data on a hard drive.

So, again, there are accommodations between the programmers' view and what goes on in hardware. The programmer has a particular view of the hardware: as we have seen, it was rather carefully constructed in order to allow for entities like variables (and, on top of these, a universe of complex data types). The various optimisations that we have seen are all built on top of this view, and, to a great extent, they work to maintain it, at the cost of the strict correctness of the view.

Furthermore, most of these optimisations are not guaranteed to work, but only work given programs and tasks with a particular statistical distribution. Optimisation is generally done to make the common case (i.e. what programs mostly do) fast. So human behaviour has an effect on this: the things that people want to do are usually what is optimised for (computer games are a case in point). Given all this, it is not surprising that computer benchmarks are a contentious, and ultimately political, subject (Hennessy and Patterson [7, §1.9, pp.46ff]).

4 Semantics

4.1 Compositionality

Since there have been programming languages, there has been work on their semantics: this is not surprising, firstly because many of the people who developed programming languages were logicians and thus naturally thought in terms of semantics, and secondly because computer scientists wanted to do things like prove that programs ran correctly, which is a question which could conceivably be answered by investigating the semantics of programs.

This proved to be a difficult problem. As Priestley remarks, "semantics for logic are typically compositional" [10, p. 113]: that is, logical formulae are generally built up from smaller components, and we can get the semantics of the larger pieces (that is, the mathematical objects they stand for) by composing the semantics for smaller pieces. It would be natural to try to define a semantics for programming language in the same sort of way. However, it is not straightforward: consider, for example, one of the simplest ways of combining commands in programming languages, namely concatenating them:

```
command1
command2
```

which says that command 2 should be executed after command 1. What mathematical objects correspond to these commands? How do we combine them?

It was some time before good answers to these problems emerged: decisive breakthroughs were made by Strachey and his school in Oxford in the 70s [14, 11]. These results did not merely allow a mathematical analysis, but they led towards an understanding of the space of possible programming languages. In particular, they put into perspective the so-called *functional* languages: these are languages in which variables have fixed values, that is, their values can only be defined and read, but not updated. These languages have good mathematical properties, which made their semantics quite perspicuous. In particular, Abramsky [1] has produced a semantically-based taxonomy of a large collection of languages, based on functional programming but with added features; the semantics is based on the game-theoretic approach to logic pioneered by Lorenzen [8].

4.2 Accommodations

This development is essentially the development of mathematical tools to investigate what had gone before. However, there is an added complication because, as we have seen, previous stages of computer use were based not merely on computer hardware but on human practices of using that hardware. There was, of course, already a great deal of mathematics associated with the use of computers to solve numerical problems in physics or engineering, and particularly there was a great deal of sophisticated numerical analysis and the analysis of algorithms. However, the work of Strachey and his school was explicitly non-numerical: it was concerned to develop an abstract semantics of computer languages. Abramsky's game-theoretic semantics is especially noteworthy: Lorenzen's concept of game is quite abstract, and thus can be applied on both the human side and the computer side, but it also enables a very rich theory of both human-computer and computer-computer interaction.

As well as the techniques, we should also consider the results of this development. It has allowed a certain reflective grasp of the technology: people are now asking questions not just about the programming languages that we happen to have, but also about possible programming languages and what advantages and disadvantages they might offer.

5 Further Pressures

5.1 The Advent of Multicore Computers

Over the last several decades there has been a remarkable trend: computers have become faster and more powerful, while becoming, if anything, cheaper (Hennessy and Patterson, [7, §1.7, pp. 40ff]). This now seems to be coming to an end: CPU clock speeds are not increasing any more. Rather, hardware manufacturers are turning to *multicore* hardware,[3] that is, hardware designs which have more than one processor on a chip (Hennessy and Patterson [7, §1.7, pp. 40ff]). This change to multicore hardware means a change from serial to parallel processing, that is, processing in which many computations can go on at one time. This is a fundamental change, the emotional impact of which is perhaps best captured in James Mickens' essay "The Slow Winter" [9].

Why might this be so fundamental? Since very early in the development of computers, there was a general pressure towards strictly serial computers: programmers should be able to write a series of instructions and be assured that the machine would execute them in that order. Early computers quite often had some capacity for parallel programming, but programmers – and competent programmers at that – generally found this very difficult to cope with: as Eckert writes of programming the ENIAC,

> In thinking out the various operations of this machine, if they can be thought out in a purely serial fashion, it is not necessary to worry about any irrelevant timing between the two steps. . . . The human brain does not think in several parallel channels at the same time: it usually thinks these things out step by step. Therefore, in all ways, it is found exceedingly desirable to build the machine so that only single steps are performed at any time. The ENIAC is usually used in this way. (J.P. Eckert Jr [3], cited in [10, p. 94])

Note "usually": it was obviously possible to use the ENIAC in a non-serial way. However, because of the perceived difficulty of parallel programming, seriality was enforced. And programming has generally remained quite serial ever since; there are niches in which parallel programming is used (mostly out of necessity), but, for example, the average undergraduate computer science curriculum has very little parallel programming in it.

5.2 Accommodations

First some history: it has been recognised for a long while that functional programming might offer certain advantages, mainly because of its more perspicuous mathematical structure. On the other hand, it has also been the case that functional programming is not very popular: it has the reputation of being hard to learn, and functional programs have the reputation of not performing well.

Several things are changing, however. Universities are starting to teach functional programming, or are resuming the teaching of it (my university, for example, has recently started teaching functional programming again after about a decade in which it was not taught). There are also several prestigious applications of either pure functional programming or of programming in a functional idiom: for example, there is a framework called Hadoop,[4] which is used for the parallel processing of big data, and which, although it is written in the non-functional language Java, is based on a functional programming paradigm called map-reduce. And it turns out that, provided that people program with Hadoop in a fairly disciplined way, they can get the advantages of functional programming without having to learn to cope with purely functional languages. So this constitutes a successful accommodation: it is not the functional programming paradise, in which everyone would program in beautiful functional programming languages, that messianic functional programming enthusiasts used to dream about. It is also a rather niche application, and it does not tell us much about how to do parallel programming in other contexts. But it is an example of the sort of accommodations that people make in order to deal with the technology that they have available, and the sort of human systems that they have available. And, as well as the purely technological issues, some of the issues at play here are how the humans make sense of the technology that they have, and how they adapt the technology so that they can make sense of it.

[3] This whole historical development is related in some way to Moore's law [16], that is, the observation that the number of transistors per chip has been doubling every eighteen months since 1970 or so. The change to multicore hardware means that this increase can continue, although, since each chip now has several cores (i.e. processors), the number of transistors per core has levelled off. Similarly, the clock speed of CPUs has levelled off at about 2-4 GHz.

REFERENCES

[1] Samson Abramsky, 'Games in the semantics of programming languages', in *Proceedings of the 11th Amsterdam Colloquium,*, eds., M. Stokhof P. Dekker and Y. Venema, (1997).

[2] Howard Aiken and Grace Hopper, 'The automatic sequence controlled calculator', *Electrical Engineering*, **65**, 384–391, 449–454, 522–528, (1946).

[3] J.P Eckert Jr, 'A preview of a digital computing machine'. Lecture delivered 15 July 1946.

[4] H. H. Goldstine and J von Neumann, 'Planning and coding problems for an electronic computing instrument, part ii, volume 3', Technical report, Institute for Advanced Study, Princeton, (1948').

[5] Douglas Rayner Hartree, *Numerical Analysis*, Clarendon, Oxford, 1952.

[6] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 5th edn., 2012.

[7] John L. Hennessy and David A. Patterson, *Computer Organisation and Design: The Hardware-Software Interface*, Morgan Kaufmann, 5th edn., 2014.

[8] P. Lorenzen, *Normative Logic and Ethics*, Bibliographisches Institut, 1984.

[9] James Mickens. The slow winter. Available online at `http://research.microsoft.com/en-us/people/mickens/theslowwinter.pdf`.

[10] Peter Mark Priestley, *Logic and the Development of Programming Languages, 1930–1075*, Ph.D. dissertation, University College London, 2008.

[11] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.

[12] Alan M. Turing, 'Proposal for development in the mathematics department of an automatic computing engine (ACE)', Technical report, National Physical Laboratory, Teddington, UK, (1946).

[13] David A Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, 1990.

[14] G. Graham White, 'The philosophy of programming languages', in *The Blackwell Guide to the Philosophy of Computing and Information*, ed., L. Floridi, 237–247, Blackwell, (2004).

[15] Wikipedia. MIPS architecture — wikipedia, the free encyclopedia, 2013. [Online; accessed 30-December-2013].

[16] Wikipedia. Moore's law — wikipedia, the free encyclopedia, 2014. [Online; accessed 3-January-2014].

[4] See `http://hadoop.apache.org/`