A Low-level Approach in a High-Level World

Terry Morris **Peter Brook**

Otago Polytechnic
tmorris@tekotago.ac.nz

Abstract

The educational value of low-level content, especially programming at machine level, has been hotly contested since the advent of higher level languages with a common view then and now being categorically dismissive (University of Teeside, 2007). Whilst acknowledging the direct importance of assembler language programming skills in the microcontroller application industry it is argued that the wider educational aspects of machine level experience in a general IT degree program such as the Bachelor of Information Technology (BIT) considerably aids the deeper understanding of a wide range of topics including general high level programming, operating systems, hardware etc.

Keywords: assembler, low-level, machine, programming, education, microprocessor, microcontroller, IT, BIT, understanding, OS, hardware

1 Introduction

The constant educational debate about course content and approach is an inevitable consequence of information technology's continuous evolution. An example provided by the University of Tessdale (2007) states :

"If you regard computer science as being primarily concerned with the *use* of the computer, you can argue that assembly language is an irrelevance."

Although the first electronic computers were essentially low-level systems especially in respect of their I/O and programming, over the years operation and programming have been refined and engineered to the more generally intuitive interfaces familiar today. With increasing abstraction there begs the question of the suitability of teaching and learning about low-level systems in today's current computing courses.

It is reasonable to accept that an effective way of understanding microprocessors is to use them. A common reaction to questions relating to the educational value of low-level programming is often as rhetorical as it is negative.

This quality assured paper appeared at the *20th Annual Conference of the National Advisory Committee on Computing Qualifications* (NACCQ 2007), Nelson, New Zealand. Samuel Mann and Noel Bridgeman (Eds). Reproduction for academic, not-for profit purposes permitted provided this text is included. www.naccq.ac.nz

The following views (Hyde, 1995) are typical:

- Assembly is hard to learn.
- Assembly is hard to read and understand.
- Assembly is hard to debug.
- Assembly is hard to maintain.
- Assembly is hard to write.
- Assembly language programming is time consuming.
- Improved compiler technology has eliminated the need for assembly language.
- Today, machines are so fast that we no longer need to use assembly.
- If you need more speed, you should use a better algorithm rather than switch to assembly language.
- Machines have so much memory today, saving space using assembly is not important.
- Assembly language is not portable.

Notwithstanding the arguments developed later - most if not all of the points noted above. can be shown in current literature (University of Teesdale, 2007) to have little justification. Show the potent piece of PERL code in List 1 to a competent programmer who has no knowledge of PERL and ask for a detailed explanation:

```
while (<>)
{
    $count++ while (/\bthe\b/ig);
}
```

List 1: Interesting but not entirely obvious PERL code counting the number of occurrences of "the" in the text files supplied via the command line.

Although it is not the most obscure piece of PERL code it is nevertheless hard to read and understand; by the same token it will be hard to debug and maintain, it looks syntactically challenging and therefore hard to learn. (In passing, please note that the author of this paper is a strong advocate of PERL although not of code presented in this manner.) Indeed, many of the apparent problems associated with assembler language programming -

inscrutability, for example - are rather more general and relate to bad programming or commenting techniques rather than to assembler in particular and to which the PERL code in List 1 bears eloquent testimony.

It can be argued with some justification that it is unnecessary to understand how a video cassette recorder works to successfully use one or Reed-Solomon codes to work with optical media. What, therefore, is the place for a study of microprocessors and their programming in machine or assembler language?

This paper takes as understood the gathering momentum of embedded processing and its industrial importance in arguing that that is not the only and not even necessarily the strongest argument for including low-level experiences within the curriculum.

Although it is self evident that technologists acquire knowledge and understanding only when and where necessary it is arguable that it is not always applicable to the educational process especially in areas where there are opportunities for meaningful cross-fertilisation.

This paper argues that the carefully chosen study of low-level components and techniques offers value extending well beyond the immediate subject matter and that it will continue to do so for the foreseeable future.

2 Glimpsing the future, accentuating the present

The software realisation of hardware capability is noted for its latency, one of the most obvious examples being the lag between the release of first of the widely adopted Windows operating systems (Windows 3.0) and the IA32 Intel 80386 processor on which it was based.

The gap between the development of Inmos' Transputer and the recent embrace of parallel processing in the author's opinion borders on the astounding. This paper argues that an undergraduate review of carefully selected microprocessors overcomes developmental latency and yields a view of the future difficult if not impossible by other means.

Processors lie at the core of computer systems; peripherals and software layers depend upon processor architecture and performance. Predicting the future of IT is an occupation of noted frailty but a study of microprocessor development has revealed some unusually clear paths and some illogical time lines.

Processor development is generally but not always driven by the demands of software. Whilst the development of pretend multitasking was a logical outcome following the release of the Intel 80386 with its support for task state switching the importance of massively parallel processing systems was heralded with the design in 1979 and the production in 1985 of the Inmos Transputer (Birkby, 1995).

The Transputer was capable of forming dynamically configurable arrays with other Transputers and thus offered massively parallel processing at a time when industry was set to espouse its contemporary, the Intel 8086. Its programming language, OCCAM, was probably more integrated into Transputer architecture than any other processor - language combination

Ironically the IA32/64 architecture is possibly a dead end should the future endorse massive parallelism whilst an important thread of discussion currently links OCCAM developments such as OCCAM-pi with the IBM Cell processor for the Sony PS/3 - a further irony because OCCAM's Transputer optimisation might have ruled it out of contention for efficient implementation on other processors. Might it be that the Transputer's design had generic implications for the future - implications also of considerable educational importance?

It is argued that it is axiomatic that tertiary undergraduate courses must look to the future and the importance of parallelism, notwithstanding the admission of many industry commentators that the view two decades hence is hazy at best. Whilst there always has been a place for the Transputer's honourable mention in courses like the BIT which should acknowledge the importance of parallelism, new processors now form the vanguard, notably the aforementioned PS/3 Cell processor and Parallax's eight-core Propeller microcontroller (Martin, 2006)..

If it achieves nothing else, the Propeller's innovative architecture has the capability of changing the way we teach microcontrollers. Yet, whilst doing so, it also emphasises the practical importance of bedrock operating system concepts that often remain remote abstractions at undergraduate level.

3 Interrupts - why bother?

In the single-processing world the interrupt remains the most practical solution the problem of multiple device-handling; a vital component of contemporary business and personal computer systems their understanding is of importance in the educational treatment of hardware and operating systems.

Tacitly ignoring the requirements of protected mode I/O virtualisation on IA-32 processors for example, the general principles of interrupt configuration vary little from processor to processor – their only drawback is the fastidious attention to their configuration.

But they have no meaning and no place in the Propeller's multi-core architecture. With its abundance of cores the dedication of a core to a device represents not only sensible but recommended processor use - the Propeller has no interrupts because it doesn't need them. Cores immediately provide service without delay, without the possibility of side effects and without involving the main program. The global availability of the system clock helps make light work of what would have been timer interrupts and core synchronisation is as precise as it is trivial. Shared resources, e.g. memory, are protected in the expected mutually exclusive manner. Moreover, with no interrupts there are no interrupt constraints.

Gone are the time penalties and overheads often associated with service routines; the need to complete the service before either the main program is significantly delayed or before another, perhaps higher priority device, requires service. That doesn't necessarily imply there are no time constraints; clearly a device-serving core and its associated routine must not be outrun with the demands of the device.

Re-entrant interrupt code is unnecessary with a 1:1 core:device ratio. Each core runs truly in parallel with its own variables and work space if necessary and with an independence and robustness that requires laborious preparation in a single processor, interrupt-driven environment.

If nothing else, the Propeller illustrates there are other approaches to efficient device handling. Although future microcontrollers might sport somewhat different architectures it appears unlikely that the valuable gains accruing from a multi-core approach will be discarded and thus the Propeller currently offers educational value well beyond its initial design brief.

4 The OS connection

The Propeller's parallel process synchronisation around blocking devices and critical code access leans heavily on what for many undergraduate students is the heady abstraction of semaphores. A vital, low-level operating system function is fleshed out with immediate and discernible practical purpose because the Propeller employs semaphores for resource locking.

Propellor semaphores are accessible from both assembler and Spin (higher-level) code and it is at the machine level that another interesting and important operating system issue is writ large; that of critical (re-entrant) code and statement atomicity.

It is one thing to write the high-level statement for a 32-bit counter in a 16-bit processor environment:

```
        counter++;
```

but it is another to truly appreciate that, ignoring the situation where variables may be assigned to dedicated registers, the statement isn't atomic at machine level but typically requires several instructions within which a task switch to another process manipulating the same variable changes it with typically disastrous results.

In the following and oft-recounted example, one problem is the possible partial change in the counter value when a task switch occurs following the writing of the low word:

```
addCounter:
mov ax, [counterLo]
add ax, 1    ; for the low word
             ; add 1 and ignore the
             ; carry flag
mov [counterLo, ax]
mov ax, [counterHi]
adc ax, 0    ; include the carry in
             ; the addition to the
             ; high word
mov [counterHi], ax
```

List 2: x86 assembler code incrementing a 32 bit variable

Machine level experience renders the false atomicity of high-level statements obvious rather than remote and abstract, and renders almost palpable the need to guard such code sections.

With parallel processes and locked resources deadlock is a distinct possibility. As always, programmers must be the devil's advocate and assume the worst; the Propellor's capacity to shut down and start cores under program control suggests a number of candidate mechanisms for deadlock recovery. Parallel process coding for the Propeller clearly has strong curricular links with an important component of multitasking operating systems.

5 Hardware reality

Programmers have long celebrated the freedom from needing detailed hardware and architectural knowledge courtesy of modern high-level languages and operating systems. But it is believed that a firm grasp of low-level issues makes for better and more robust programming at any level. Some of the more important are (Harvard University, 2002, abbreviated)

1. **Assembly language illustrates pointers.** Pointers are one of the most difficult concepts for novice students to grasp. Once students have written assembly language code to implement address arithmetic and have used load and store instructions, pointers and addresses in higher-level languages become much more intuitive.

2. **Assembly language teaches essential design skills and style.** Even very short pieces of assembly language code are difficult to write unless they are well designed and structured, and nearly impossible to read unless properly documented.

3. **Assembly language makes clear the architecture of the underlying machine.** Understanding the underlying machine demystifies the computer and programming in assembly language helps students understand how extremely complex programs can be reduced to sequences of very simple instructions.

After all, machine language is the only language which is ever executed and the CPU's perception of the outside world is only one of memory.

6 Machine-Human Disparity

Humans are essentially experiential creatures if their inability to learn either from the lessons of history or from the mistakes of others is anything to go by; a vexed state that caused George Bernard Shaw to exclaim "If history repeats itself, and the unexpected always happens, how incapable must Man be of learning from experience".

That processor and human response times are markedly different is axiomatic but it is argued that practical experience is formative to understanding its implications. That understanding in turn becomes a key skill in the many undergraduate Projects that demand low-level, direct device interaction.

If there is a classic example, one that has initially perplexed generations of students, it must undoubtedly be that of the repeated button-push which causes some rapidly executed event. Consider a system of eight LEDs linked to an output port such that successive single LEDs are lit by repeated button pushes. The sample code of List 3 illustrates the defeat of logic by speed.

A student's initial reaction is to prepare a solution without the code highlighted in bold. Why not indeed? It all seems so straightforward. Wait for the button press and carry out the processing which handles the LEDs. Few would perform the essential preliminary of comparing execution speed with that of a human key press.

Consider the code shown in List 3, extracted from a program driving an ATMEL Mega16 clocked at 3.686MHz in an STK500 environment, 48 instructions taking a total of approximately 13.02µs are needed to walk through all the LEDs with the reasonable assumption of one instruction per clock.

What is the problem? It is that most people find it difficult to leave the switch down for less than about 0.05s, time for the entire operation to take place around 3840 times. And that is what happens - the LEDs never appear to light because they are switched on and off so quickly.

As each iteration completes, the switch still appears to be down; the program as yet has no way of telling whether it is the same or another key press! Waiting for the switch to come up after it is pressed is a useful solution to the problem - assuming immediacy is not of the essence. Adding around 5ms delay to pass over any switch bounce is a worthwhile enhancement.

Apart from the issue of speed disparity the program illustrates all too clearly the time-wasting character of polling especially in the context of a single-core processor and serves as a springboard for the introduction of interrupts.

7 Algorithms and data structures at machine level

Assuming the educational value of working with component technology, especially intelligent devices then the interpretation and manipulation of data at machine level must be regarded as a valuable if not an essential skill. There is far too much pressure on content inclusion in undergraduate papers to deliver material not of proven value (University of Teeside, 2007)

"The availability of high-performance hardware and the drive to include more and more new material in the CS curriculum, has put pressure on academics to justify what they teach. In particular, many are questioning the need for courses on assembly language".

```
main:
sbic PIND, 0 ; is active low SW0 down?
             ; read the pins not the
             ; latches
rjmp main
switchIsDown:
sbis PIND, 0 ; wait for SW0 to come up
rjmp switchIsDown
switchIsOn:
com LEDs     ; invert LEDs register
             ; prior to writing to
             ; PORTB
out PORTB, LEDs
com LEDs     ; restore LEDs essential
             ; for walking 1's
lsl LEDs     ; walk the "1" one place
             ; to the left
rcall waitAround  ; hang around to
             ; eliminate switch bounce
brcs programFinished ; can't walk any
             ; further, "1" falls into
             ; the carry flag
noLEDAction:
rjmp main
```

List 3: Responding to a button press at machine level – logic defeated by perception

In the opinion of the author it is correct to question the value of teaching number systems and their associated arithmetic and it is difficult to refute if they are considered out of context.

But when working with processors, especially microcontrollers, which have limited memory, it becomes apparent that a purpose-built, code-compact, arithmetic routine is preferable to the inclusion of a large library or even a specific general purpose routine. Confrontation with a device as simple as the Dallas D1620 combination thermometer - thermostat demands the understanding of signed binary numbers because it returns temperature data as a 9-bit two's complement number.

In a microcontroller environment, too, there are decisions to be made about precision and magnitude, decisions that can only be made with an understanding of number and its machine representation. Experience in performing arithmetic with multiple precision numbers has immense educational value well outside the immediate context because of the practice in developing algorithms for what is not only abstract but eminently useful in the cause of efficiency. Perhaps there is nothing quite so illustrative about pointers than working with them at machine level.

Many higher level languages such as Pascal exhibit "obscure" behaviour when numeric variables are used outside of their magnitude maxima or minima. Look no further than the remarkable PicAxe microcontroller from

Revolution Education Ltd (UK) increasingly used within New Zealand schools. It happily interprets the assignment to the byte variable b0:

```
b0  =  -9
```

- only to display its value as 247.

The BASIC interpreter is well-endowed with clever and useful I/O but it is an integer processor which often has to be called upon to deliver pseudo floating point results and thus an understanding of data representation at machine level is essential to maintaining precision during computation.

A useful feature is its capacity to interface with a number of industry standard devices such as the Dallas DS18B20 temperature sensor..

The PicAxe reads 10-bit temperature data and neatly deposits it into a register variable as the binary Celsius value. `However` although there is a 12-bit instruction, `readtemp12`, it only retrieves the raw binary data from the sensor and it is left to the programmer to transform it into the Celsius value. Once again it is clear that the understanding of numeric data storage is underpins the computation.. This and similar low-level numeric techniques are within the boundary of the school technology curriculum as the Electrotechnology Industry Training Organisation (ETITO) and others vigorously promote control technology, especially through the PicAxe. It is believed these developments illustrate the universal need for understanding of low-level data depiction and manipulation.

8 Conclusion

The burgeoning complexity of computer systems moves programming environments closer to the human condition and ever more distant from the executive machine. With the increasing distance comes an inevitably reduced perception of the importance of machine architecture and processor behaviour. The review of curricular content of necessity must be a continuous process, it is also almost inevitable that the inclusion of low-level material in an undergraduate course will elicit a debate that can only grow stronger with the passage of time.

The case for the continued place of low-level understanding within an undergraduate course has hopefully been established not just in its own right but because it serves to support many other facets of IT education, delivering a robust foundation difficult to achieve by other means.

However, the curricular position of low-level content and its attendant skills needs careful thought. Clearly it is not an ideal candidate for a first programming experience but it arguably represents an important thread running through at least the first two years of an IT degree. Otago Polytechnic treats it as such through an introductory Microware paper encountered in the second semester of the 1st year and its 2nd year development, the Microprocessors paper. It serves as a springboard for the second year hardware paper and the third year Control Technology paper.

9 References

Ant Course Modules: CS1 Assembly Language Programming, Harvard University. http://www.ant.harvard.edu/modules/alp8/index.shtml. Accessed 18 Mar 2007.

Assembler Is Not For Dummies, Open Source Software Educational Society. http://www.softpanorama.org/Lang/assembler.shtml Accessed 17 March 2007

Birkby, R. (1995): *A Brief History Of The Microprocessor*. Lexikon Services. http://www.computermuseum.li/Testpage/MicroprocessorHistory.htm Accessed 12 Jun 2007

Maxim Integrated Products Ltd. *Dallas 18B20 Datasheet*.

http://datasheets.maxim-ic.com/en/ds/DS18B20.pdf. Accessed 23 Oct 2006

Dimmich, D.J., Jacobsen, C.L. and Jadud, M.C. (2006): A Cell Transterpreter. *Communicating Process Architectures*, Napier University, Edinburgh, UK. 64:215-224, IOS Press.

Hyde, R. (2003): *The Art of Assembly Language.* San Francisco, No Starch Press. 928pp

Martin, J. (2006). *Propellor Manual.* Parallax Inc. http://www.parallax.com/detail.asp?product_id=122-32000. Accessed 7 Dec 2006.

New Zealand Ministry of Education. (1995): *Technology in the New Zealand Curriculum*. Wellington, Learning Media Limited. 86pp

The Case For and Against Assembly Language, School of Computing, University of Teeside. http://www-scm.tees.ac.uk/users/u0000408/CaseFor.htm Accessed 17 Mar 2007