

# HARDWARE COURSES AND THE UNDERGRADUATE COMPUTER SCIENCE CURRICULUM AT SMALL COLLEGES

## ABSTRACT

*Hardware courses have always been a part of the undergraduate Computer Science (CS) curriculum at small colleges. However, these courses are often viewed by many faculty members either as a necessity to satisfy accreditation requirements or, at best, as a traditional but useless part of the CS program in liberal arts settings. The material in these courses is often considered self-contained, and the relationship with other courses in the CS curriculum is rarely emphasized. In this paper we take a closer look at the place and relevance of hardware courses in the CS curriculum of small colleges.*

## 1. INTRODUCTION

The Computer Science curriculum at a typical small college invariably includes at least one "hardware" course. Most commonly this course is Computer Organization and Architecture, but often it is substituted by an Assembly Language course, which offers students an abbreviated introduction to computer architecture. In addition, some Discrete Mathematics courses offer a brief overview of Boolean algebra, logic gates, and combinational circuit analysis and design. In rare cases there is a second "hardware" course, usually offered as an elective to seniors. Usually, none of the above courses has a lab, except perhaps the Assembly Language course, where students use the allotted lab time to write assembly language programs on a PC.

By comparison, the Computer Science curriculum at a typical "engineering" college requires students to take a well-balanced mix of hardware, software and theoretical courses. The usual hardware course sequence includes Digital Design, Computer Organization, and Microprocessors - all required courses for CS majors, followed by a required hardware elective selected from Advanced Computer Architecture, Parallel Computing, Embedded Systems, etc. Each course is usually accompanied by a lab, which allows students to gain hands-on experience designing, building, verifying/debugging, and programming hardware systems, and getting a first-hand look at the interaction between software and hardware.

The reasons for this disparity are many. The small number of faculty and the heavy teaching load in a typical Computer Science department at a small college makes allocating resources for preparing and teaching a sound hardware course sequence difficult. The limited funds are often considered better spent on updating the existing general purpose computing lab equipment, rather than investing into a specialized hardware lab. In addition, the number of general liberal arts requirements leaves little room for adding hardware courses to the curriculum, short of reducing further the small number of elective courses.

More significantly, however, many faculty members consider hardware courses essentially useless, kept in the Computer Science undergraduate curriculum to offer students a glimpse of a traditional but no longer needed aspect of Computer Science. Computer Science Programs aspiring for accreditation are mandated by the CSAB/ABET requirements to include hardware courses as part of the curriculum. However, the material covered in these courses is often treated as self-contained, and not referenced in any other CS course in the curriculum. Instead of exploring the complexity of modern computer systems, and the intricacies of hardware-software interaction, many "hardware" courses limit the discussion of machine organization and architecture to a brief enumeration of the general purpose registers, spending most of the allotted course time on Assembly Language programming. As a result, students consider hardware courses to be just another hurdle to be overcome on the way to the Bachelor's degree, never gaining knowledge or even appreciation for the power of modern computer systems, and their intimate interdependence with operating systems, programming languages, and applications.

In this paper we examine (from a personal perspective) the place and relevance of hardware courses in the CS curriculum at small, liberal-arts colleges. We emphasize many aspects of hardware-software interaction, point out connections to other courses in the Computer Science curriculum, and study the significance of hardware courses as an integral part of the professional development of young computer scientists with respect to the demands of computer industry. Our sincere hope is that this paper will stimulate further discussion on the role of hardware courses in the Computer Science curriculum, and bring forth other opinions on the subject.

## 2. MYTHS AND MISCONCEPTIONS

The disregard for the importance of hardware courses in the CS curriculum is based on a number of fundamental misconceptions:

### **"Computer Scientist = Programmer"**

The reluctance to incorporate hardware courses more deeply into the CS curriculum at small colleges comes in part from the fact that many people - students, the industry, and even faculty - consider computer scientists to be nothing but programmers. A true computer scientist, however, is one who struggles to better understand the foundations of his/her discipline in order to develop innovative approaches to creating improved software and hardware technology. To achieve this, the computer scientist needs to be well versed in all three aspects of Computer Science - theory, software, and hardware. Thus, the goal of Computer Science education should be to reveal the hidden interdependencies among these three spheres of knowledge, and develop the student's ability to draw upon all of them when faced with a professional challenge. For example, algorithm design is almost invariably presented in the context of a programming-based Data Structures course, tied to a specific programming language such as C++ or Java. Algorithms, however, are abstract notions, and should be presented without reference to any particular programming language, *without reference to any particular implementation*. An important part of designing an algorithm should be verifying its correctness and its estimating the complexity - both theoretical notions. Furthermore, an algorithm may be implemented in software or in hardware. The tradeoffs between the two approaches - "hardware is fast, software is cheap" - are worth discussing with the students, especially in the context of a particular real-world example, e.g. z-buffering implemented in software or with a special video hardware.

### **"If you use high-level languages, you don't need to know hardware"**

The advent of high-level languages has allowed the semantics of programs to be arbitrarily complex and removed from the issues of the actual hardware on which the program is to be executed. However, knowledge of hardware is still essential. When writing a program in a high-level language, the programmer relies on the correct and efficient implementation of the compiler he/she is using. Whenever speed and compactness must be guaranteed (as is the case with some aspects of operating systems design for example), a low-level language is usually the choice. This requires at least a certain level of familiarity with the underlying hardware of the machine.

In addition, knowledge of how the hardware handles the execution of programs helps the programmer create more efficient code. For example, knowing that the CPU is a pipeline processor, and that control hazards are a major problem in the pipeline will help the programmer use loops and recursion more judiciously. The concept of delayed branch slots, implemented by optimizing compilers to avoid CPU pipeline control hazards, is not available to programmers who use interpreted languages. They need to understand the issues of control hazard avoidance in order to be able to restructure their programs accordingly for maximum performance. Knowing that the system uses a virtual memory tells the careful programmer to limit the size of loops and functions so they can fit on a single page, thus reducing frequency of page faults.

### **"The road to computer hardware is through Assembly Language"**

Many small colleges opt to teach an Assembly Language course instead of Computer Organization and Architecture. The course briefly introduces students to the programmer significant architectural features of the machine such as the general purpose registers, and quickly moves on to examine every aspect of the assembly language of the chosen machine.

There is no doubt that assembly languages are important, but only when studied in conjunction with the architecture and organization of a computer. Instead of limiting the discussion to the assembly language of a specific machine, students should be introduced to a variety of assembly languages - one for each type of architecture they encounter - stack, accumulator, general purpose register, etc. In each case, only the most significant aspects of the language should be introduced, emphasizing the advantages and disadvantages of the particular language and architecture with respect to other architectures discussed in class. Assembly language programming should be the foundation on which a deeper understanding of program execution by the hardware is based.

It is also important to point out that modern programming practices rarely require the sole use of assembly languages. Even low-level programming tasks such as writing device drivers are often carried out in C. If an assembly language is used, it is often embedded in a high-level language program. Therefore, it may be useful to students to study the low-level features of the C programming language, and learn how to incorporate assembly code into high-level language programs.

### **3. HARDWARE COURSES AND ...**

The knowledge gained in hardware courses can be easily and efficiently integrated into other courses in the Computer Science undergraduate curriculum. This will emphasize the connection between various disciplines of Computer Science, and improve the algorithmic thinking and problem solving skills of our students.

#### **3.1 ... Operating Systems**

The interaction and deep interdependence between hardware and software is probably most apparent in the Operating Systems course, which is usually required of all CS majors. The operating system is the intermediary between the user and the underlying machine hardware. Thus, to design an operating system, the students must be familiar with the specifics of the machine for which the operating system is being designed. Various aspects of operating system design (and the efficiency of that design) depend directly on the availability of hardware support. For example, the concept of a context switch, i.e. switching the execution from one process to another, and the time it requires depend on the number of registers the CPU has. RISC processors, which usually have a large number of general-purpose registers will take longer to carry out a context switch since there are more register values to be updated in the process control block of the process being preempted, and more values to be loaded from memory into the registers for the process selected for execution next. A CISC processor, which has fewer general-purpose registers, will require less context switch time. Another example arises when discussing process scheduling. Implementing Round-Robin scheduling is contingent on the availability of a timer, which can interrupt the CPU when the time quantum allotted to an executing process is over. As a third example, consider the fact that implementing paging and virtual memory depends on the availability of either a set of page table registers into which the contents of the page table is loaded on a context switch, or on the availability of a Page Table Base Register pointing to the beginning of the page table of the executing process in memory. The availability of a Table Lookaside Buffer - a specialized cache for storing the frame numbers of frequently referred to pages - can dramatically increase the performance of an operating system, which uses paging as a memory allocation policy [1]. Many more examples can be presented from every aspect of operating system design - process management, memory management, IO and secondary storage management, networking, security, etc.

#### **3.2 ... Compilers and Programming Languages**

The job of the compiler is to convert source code written according to the rules of a particular high-level language into machine instructions that can be executed on the hardware of the computer. Thus, the compiler writer must be quite familiar with the architecture of the machine for which the compiler is targeted. Details of the instructions set, internal CPU storage and its organization and access principles are all factors, which contribute to the efficiency of the compiler being created. Register allocation is one of the most fundamental tasks of a high-level language compiler. For example, when a variable is being accessed often (e.g. in a loop) the compiler may allocate a general-purpose register for temporary storage of the variable's value. After the loop is completed, the final value is loaded back from the temporary general purpose register into the original memory variable. Moving beyond simple compilers, optimization techniques often require significant level of hardware support, and are aimed at improving the performance of the computer system. For example, the delayed branch slot technique involves selecting an instruction independent of the

branch outcome and fetching it into the instruction pipeline right after the branch instruction, avoiding the need to predict whether the next instruction after the branch or the target of the branch will need to be fetched. For this to work, of course, the pipeline of the CPU must be set up so that the branch outcome is computed as soon as possible, i.e. immediately after the branch instruction is fetched [3]. It also requires that the compiler be able to rearrange the order of execution of instructions without changing the semantics of the program. This type of out-of-order execution can also be implemented in hardware with the help of a hardware unit referred to as an instruction window [2], into which instructions are deposited as soon as they are decoded. Any functional unit may then fetch a waiting instruction from the instruction window and proceed with the execution stage. The advantage of the hardware instruction window approach is that it can be done at execution time, while the compiler approach allows for a deeper, more accurate search for independent instructions. This is a perfect example of how closely intertwined issues of hardware and software are.

### **3.3 ... Data Structures**

As mentioned above, the Data Structures course has traditionally been tied to a particular programming language. The algorithms presented throughout the course are usually given in pseudo-code, which often closely resembles the programming language which the students are most familiar with from their Introduction to Programming courses. As homework, students are expected to convert the pseudo-code into real code using a specific high-level language. There are several problems with this approach. First and foremost, students learn to equate algorithms with high-level language programs. The idea that an algorithm can be implemented with anything else but C++ or Java never occurs to them. Moreover, the specifics of the implementation, which depend on the features of the particular language often shift the attention of the student away from the main ideas of the algorithm.

There are many ways in which “hardware” courses and a Data Structures course can benefit from each other. The essential thing is to convince our students early on in the Data Structures course that algorithms are abstract notions, and need not always be implemented with a high-level language. A simple example to illustrate how algorithms can be implemented in hardware is Booth's Recording used for multiplying signed numbers. The instructor may assign the implementation of Booth's Recording as a programming project, and then demonstrate in class the hardware implementation of the algorithm using simple shift registers and adders. Other such algorithms, which will similarly prove the point are the unsigned multiplication algorithm, and the non-restoring- and restoring division algorithms [3].

On the other hand, the algorithms introduced in the data structures course can be used as the basis for discussing various important concepts in hardware design. For example, the notion of stack architecture is based on the familiar LIFO stack principle. The actual implementation in hardware can be done with registers, with memory, or a combination of the two. For example, a CPU may use several general-purpose registers in a stack fashion. Whenever the depth of the stack exceeds the number of available registers, the stack is "extended" into memory. The access mechanism of this hardware stack is microcoded into the control unit of the CPU. An example of such a stack architecture is the CPU in fig.1 [4], or the architecture of the InMOS Transputer series of microprocessors, T414, T800, T9000 [5].

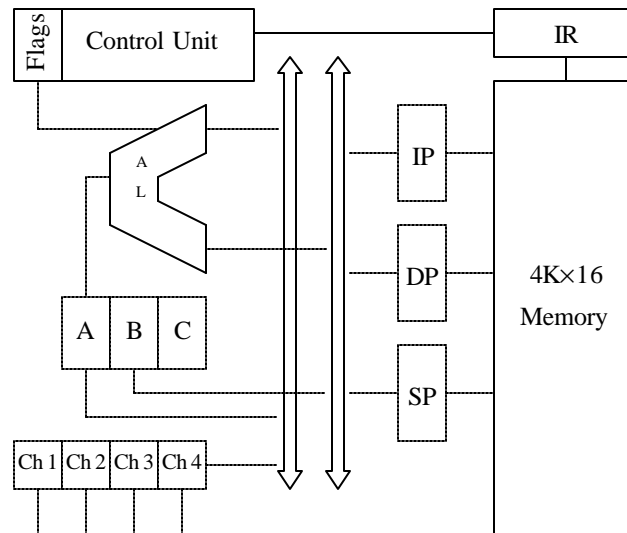


Fig.1 An example of a CPU with a Stack Organization

Besides hardware stacks, various other data structures abound in the domain of hardware design. Array multipliers, instruction prefetch queues, tree architectures, and shuffle networks are among the many applications of fundamental ideas presented in the Data Structures course and implemented in hardware.

### 3.5 ... Other CS courses

Many advanced courses and senior electives can also benefit from an improved hardware course sequence. Below we briefly discuss some of the issues related to only two of these courses:

#### - Networking and Telecommunication

Many Computer Science programs offer their students the opportunity to specialize in Networking and Telecommunication. Advanced courses within this area of specialization offer a unique blend of hardware and software, since the high level network features exist on top of specific communication hardware devices governed by network protocols implemented in software and/or hardware. However, the instructor is often forced to spend valuable classroom time discussing the basics of digital circuits and the design of simple devices such as decoders, encoders, multiplexors, demultiplexors, etc. This leaves less time for advanced networking issues, and decreases the overall level of the course. A good knowledge of basic hardware devices, gained early in the curriculum will allow students to focus on the advanced principles of networking and telecommunication, and help the instructor concentrate on advanced topics.

#### - Computer Graphics

Another usual senior elective is the Computer Graphics course, which almost always centers on the implementation of graphics algorithms in software. Students are given pseudo-code, and asked to implement the algorithms discussed in class using their usual programming language. This once again reinforces the students' beliefs that algorithms are equivalent to programs. At the same time, hardware has become an

indispensable component of Computer Graphics. Practically all video cards today employ a powerful Graphics Processing Unit (GPU), in which numerous rendering algorithms are implemented in hardware. Instead of learning to tap into and use this power, students are forced to “reinvent the wheel”, by implementing trivial and obsolete algorithms, kept in the textbooks by tradition. An applications-oriented course on Computer Graphics should, therefore, concentrate on teaching students how to use the available advanced graphics resources, while a more theoretical course should weigh the advantages and disadvantages of the software and/or hardware implementation of graphics algorithms, and offer students both perspectives.

#### **4. A GLIMPSE TOWARD THE FUTURE**

The flood of hand-held programmable devices such as PDAs, cell-phones with Internet access capabilities indicates that the future of computer science does not diverge from the issues of hardware. On the contrary, there is a higher and higher industrial demand for computer scientists with a solid knowledge of hardware as well as software, able to integrate the two into a portable embedded system of ever-increasing complexity. If students graduating from the Computer Science programs of small colleges are to remain marketable and competitive, it is imperative that we begin to integrate hardware education much more seriously into the Computer Science curriculum, so as to better prepare our students for the demands of the work-place of tomorrow.

#### **5. CONCLUSIONS**

This paper examined the balance of hardware, software, and theoretical courses in the undergraduate Computer Science curriculum at small, liberal arts colleges. We are concerned with the fact this balance has shifted significantly towards programming and away from hardware and theory. Many of the notions and ideas presented in this paper are not new, but we felt we needed to bring them into focus so as to spark a new, serious discussion about the significance of hardware courses in the overall education of Computer Science undergraduates. We hope that reading this paper will convince our colleagues to reexamine their views of the relevance of hardware education, and help them find new balance between software and hardware in their lectures. This will only help improve the quality of undergraduate education at our schools, and better prepare our students for the new demands of computer industry.

#### **6. REFERENCES**

- [1] Silberschatz, Galvin, Gagne “*Operating System Concepts*”, 6<sup>th</sup> ed, Wiley Publ, 2002
- [2] W. Stallings, “*Computer Organization and Architecture*”, 5<sup>th</sup> Edition, Prentice Hall, 2000
- [3] J.Hennessy, D.Patterson, “*Computer Architecture: A Quantitative Approach*”, 2<sup>nd</sup> Edition, Morgan Kaufmann Publishing, 1996
- [4] L.Ivanov “*Formal Verification of a Microprocessor Control*”, Proceedings of IEEE MWSCAS'01, 2001
- [5] Sima, Fountain, Kacsuk, “*Advanced Computer Architectures: A Design Space Approach*”, Addison Wesley, 1997
- [6] P.Garret, “*Making, Breaking Codes: An Introduction to Cryptology*”, Prentice Hall, 2001